Proceedings of the Eighth International Workshop on the Web and Databases

WebDB 2005

June 16-17, 2005
Baltimore, Maryland
Collocated with ACM SIGMOD/PODS 2005

In Cooperation with ACM SIGMOD

Edited by AnHai Doan, Frank Neven, Robert McCann, and Geert Jan Bex

FOREWORD

This volume contains 16 papers, 9 posters, and 2 demos that were presented at the Eighth International Workshop on the Web and Databases (WebDB), which was held June 16-17, 2005, in Baltimore, Maryland, in conjunction with the ACM SIGMOD/PODS conference. All contributions present preliminary reports on continuing research, and were selected by the Program Committee out of 59 submissions. During the selection process, particular emphasis was given to papers that promote novel research directions and to data integration, the underlying theme of this year's workshop edition. In addition to the contributed papers, posters, and demos, the WebDB 2005 program included a panel moderated by Alon Halevy (University of Washington), and two invited talks by Michael S. McQueen (W3C and MIT, joint with XIME-P) and by William W. Cohen (Carnegie Mellon University).

The organizers thank the Program Committee for providing thorough valuations within a very short time, Rob McCann and Geert Jan Bex for compiling the proceedings and maintaining the Web site, Microsoft for providing us with the Conference Management Toolkit, the SIGMOD organizers (especially Marianne Winslett and Lisa Singh) for their assistance in organizing the workshop, and Sihem Amer-Yahia, Luis Gravano, Juliana Freire - past WebDB organizers - for providing advice on WebDB organization.

PROGRAM COMMITTEE

Bruce Croft	University of Massachusetts at Amherst, USA
Kevin Chang	University of Illinois at Urbana-Champaign, USA
Junghoo Cho	University of California at Los Angeles, USA
Wenfei Fan	Bell Labs and University of Edinburgh, UK
Juliana Freire	University of Utah and OGI/OHSU, USA
Zack Ives	University of Pennsylvania, USA
H. V. Jagadish	University of Michigan, USA
Christoph Koch	Technische Universitaet Wien, Austria
Nick Koudas	University of Toronto, Canada
Chen Li	University of California at Irvine, USA
Bertram Ludaescher	University of California at Davis, USA
Maarten Marx	University of Amsterdam, The Netherlands
Ioana Manolescu	INRIA, France
Sergey Melnik	Microsoft Research, USA
Christopher Olston	Carnegie Mellon University, USA
Erhard Rahm	University of Leipzig, Germany
Arnaud Sahuguet	Bell Labs Research, USA
Jayavel Shanmugasundaram	Cornell University, USA
Gerhard Weikum	Max-Planck-Institut fuer Informatik, Saarbruecken, Germany
Clement Yu	University of Illinois at Chicago, USA

ORGANIZERS

AnHai Doan	University of Illinois, USA
Frank Neven	Hasselt University, Belgium

WEB CHAIR

Geert Jan Bex	Hasselt University, Belgium

PROCEEDINGS CHAIR

Robert McCann	University of Illinois, USA

WORKSHOP SCHEDULE

DAY 1: Thursday June 16, 2005

2:30-3:30pm	**Invited Talk** **What does XML have to do with Immanuel Kant?** *Michael S. McQueen*
3:30-4:00pm	Coffee Break
4:00-5:00pm	**Panel on future directions of information integration** *Organized by Alon Halevy*
5:00-5:10pm	Short Break
5:10-6:30pm	**Paper Session 1: Data Integration and Web** *Chair: Gerhard Weikum* **Searching for Hidden-Web Databases** *Luciano Barbosa, Juliana Freire* **iFuice - Information Fusion utilizing Instance Correspondences and Peer Mappings** *Erhard Rahm, Andreas Thor, David Aumueller, Hong-Hai Do, Nick Golovin, Toralf Kirsten* **Managing Integrity for Data Exchanged on the Web** *Gerome Miklau, Dan Suciu* **Design and Implementation of a Geographic Search Engine** *Alexander Markowetz, Yen-Yu Chen, Torsten Suel, Xiaohui Long, Bernhard Seeger*

DAY 2: Friday June 17, 2005

8:30-9:10am	**Paper Session 2: Web and Peer-to-Peer Systems** *Chair: Kevin Chang* **Using Bloom Filters to Refine Web Search Results** *Navendu Jain, Mike Dahlin, Renu Tewari* **JXP: Global Authority Scores in a P2P Network** *Josiane Xavier Parreira, Gerhard Weikum*
9:10-10:00am	**Short talks for posters (5 min each)** *Chair: Frank Neven* **XFrag: A Query Processing Framework for Fragmented XML Data** *Sujoe Bose, Leonidas Fegaras* **Analysis of User Web Traffic with A Focus on Search Activities** *Feng Qiu, Zhenyu Liu, Junghoo Cho* **Processing Top-N Queries in P2P-based Web Integration Systems with Probabilistic Guarantees** *Katja Hose, Marcel Karnstedt, Kai-Uwe Sattler, Daniel Zinn* **Context-Sensitive Keyword Search and Ranking for XML** *Chavdar Botev, Jayavel Shanmugasundaram* **Constructing Maintainable Semantic Mappings in XQuery** *Gang Qian, Yisheng Dong* **The Framework of an XML Semantic Caching System** *Wanhong Xu* **A Data Model and Query Language to Explore Enhanced Links and Paths in Life Science Sources** *George Mihaila, Felix Naumann, Louiqa Raschid, Maria Esther Vidal*

Malleable Schemas: A Preliminary Report
Xin Dong, Alon Halevy

Mining the inner structure of the Web graph
Debora Donato, Stefano Leonardi, Stefano Millozzi, Panayiotis Tsaparas

Managing Multiversion Documents & Historical Databases: a Unified Solution Based on XML
Fusheng Wang, Carlo Zaniolo, Xin Zhou, Hyun J. Moon

10:00-10:40am Coffee Break/Poster Presentation

10:40-12:00pm **Invited Talk**
Chair: AnHai Doan

A Century Of Progress On Information Integration: A Mid-Term Report
William Cohen

12:00-1:00pm Buffet Lunch

1:00-2:20pm **Paper Session 3: XML**
Chair: Zack Ives

On the role of composition in XQuery
Christoph Koch

An Empirical Evaluation of Simple DTD-Conscious Compression Techniques
James Cheney

Towards a Query Language for Multihierarchical XML: Revisiting XPath
Ionut Iacob, Alex Dekhtyar

Indexing Schemes for Efficient Aggregate Computation over Structural Joins
Priya Mandawat, Vassilis Tsotras

2:20-2:50pm Coffee Break/Poster Presentation

2:50-3:50pm **Paper Session 4: Keyword Search, Peer-to-Peer Systems, and Web**
Chair: Jayavel Shanmugasundaram

An Evaluation and Comparison of Current Peer-to-Peer Full-Text Keyword Search Techniques
Ming Zhong, Justin Moore, Kai Shen, Amy Murphy

Efficient Engines for Keyword Proximity Search
Benny Kimelfeld, Yehoshua Sagiv

Freshness-Aware Scheduling of Continuous Queries in the Dynamic Web
Mohamed Sharaf, Alexandros Labrinidis, Panos Chrysanthis, Kirk Pruhs

3:50-4:00pm Short Break

4:00-5:00pm **Paper Session 5: XML**
Chair: Christoph Koch

Vague Content and Structure (VCAS) Retrieval for Document-centric XML Collections
Shaorong Liu, Wesley Chu, Ruzan Shahinian

On the Expressive Power of Node Construction in XQuery
Wim Le Page, Jan Hidders, Philippe Michiels, Jan Paredaens, Roel Vercammen

Indexing for XML Siblings
SungRan Cho

INVITED TALKS

What does XML have to do with Immanuel Kant?

Michael S. McQueen, World Wide Web Consortium

This talk by one of the authors of the XML 1.0 specification will attempt to distinguish technical wheat from marketing chaff and disentangle truths and fictions about XML as a data format. He will describe the origins of XML and the goals of the original design, identify the technically and socially most interesting aspects of XML, and sketch out the roles it can play in solving the problems of information management. The relation of syntax to semantics in XML markup, the relationship of XML to the concept of 'semistructured data', and the role of schemas and data validation in XML will be discussed. Particular attention will be paid to the ways in which XML and database systems complement each other or compete.

A Century Of Progress On Information Integration: A Mid-Term Report

William Cohen, Carnegie Mellon University

Over the last half-century, information integration has progressed from the study of narrow problems, of interest only to obscure technical communities, to a broad field of great scientific, social and economic importance. I believe that over the next half-century, our lives will be profoundly affected by the degree to which today’s problems of information integration can be solved, and the uses to which these solutions are put. In this talk I will survey some recent work in information integration, and how this work relates to two important technical questions. First, when is it best to resolve uncertainty about object identity, and when is it best to propagate this uncertainty? Second, how can structured information be integrated with information in unstructured formats, such as genomic data, images and text?

PROCEEDINGS CONTENTS

Searching for Hidden-Web Databases

Luciano Barbosa
University of Utah
lab@sci.utah.edu

Juliana Freire
University of Utah
juliana@cs.utah.edu

ABSTRACT

Recently, there has been increased interest in the retrieval and integration of *hidden-Web data* with a view to leverage high-quality information available in online databases. Although previous works have addressed many aspects of the actual integration, including matching form schemata and automatically filling out forms, the problem of *locating relevant data sources* has been largely overlooked. Given the dynamic nature of the Web, where data sources are constantly changing, it is crucial to automatically discover these resources. However, considering the number of documents on the Web (Google already indexes over 8 billion documents), automatically finding tens, hundreds or even thousands of forms that are relevant to the integration task is really like looking for a few needles in a haystack. Besides, since the vocabulary and structure of forms for a given domain are unknown until the forms are actually found, it is hard to define exactly what to look for.

We propose a new crawling strategy to automatically locate hidden-Web databases which aims to achieve a balance between the two conflicting requirements of this problem: the need to perform a broad search while at the same time avoiding the need to crawl a large number of irrelevant pages. The proposed strategy does that by focusing the crawl on a given topic; by judiciously choosing links to follow within a topic that are more likely to lead to pages that contain forms; and by employing appropriate stopping criteria. We describe the algorithms underlying this strategy and an experimental evaluation which shows that our approach is both effective and efficient, leading to larger numbers of forms retrieved as a function of the number of pages visited than other crawlers.

Keywords

hidden Web, large scale information integration, focused crawler

1. INTRODUCTION

Recent studies estimate the hidden Web contains anywhere between 7,500 and 91,850 terabytes of information [2, 14]. As the volume of information in the hidden Web grows, there is increased interest in techniques and tools that allow users and applications to leverage this information. In this paper, we address a crucial problem that has been largely overlooked in the literature: how to efficiently locate the searchable forms that serve as the entry points for the hidden Web. Having these entry points is a necessary condition to perform several of the hidden-Web data retrieval and integration tasks. The searchable forms can be used as the starting point for deep crawls [18, 1] and for techniques that probe these databases to derive source descriptions [11]; and in form matching they can serve as inputs to algorithms that find correspondences among attributes of different forms [12, 24, 13].

Eighth International Workshop on the Web and Databases (WebDB 2005), June 16-17, 2005, Baltimore, Maryland.

Several factors contribute to making this problem particularly challenging. The Web is constantly changing – new sources are added, and old sources are removed and modified. A scalable solution, suitable for a large-scale integration or deep-crawling task, must *automatically find the hidden-Web sources*. In addition, even for a well-defined domain (*e.g.,* books for sale), it is hard to specify a schema (or schemata) that accurately describes the relevant forms. Since there is a wide variation both in the structure and vocabulary of forms, if the definition is too strict, we risk missing relevant forms that use a slightly different schema vocabulary or structure. And in order to obtain a general definition that covers the domain well, it is necessary to have the forms to discover the correspondences among attributes [12, 24, 13]. Thus, we *need to perform a broad search.* But forms are very sparsely distributed. A recent study estimates that there are 307,000 deep Web sites, and an average of 4.2 query interfaces per deep Web site [7]. Thus, searching for tens, hundreds or even thousands of forms, that are relevant to the integration task among billions of Web pages is really like looking for a few needles in a haystack. In order to be practical, the search process must be *efficient* and avoid visiting large unproductive portions of the Web.

A possible approach to address this problem would be to perform a full crawl of the Web, but this would be highly inefficient. An exhaustive crawl can take weeks; and as the ratio of forms to Web pages is small, this would lead to unnecessarily crawling too many pages. Another alternative would be to use a focused crawler. Focused crawlers try to retrieve only a subset of the pages on the Web that are relevant to a particular topic. They have been shown to lead to better quality indexes and to substantially improved crawling efficiency than exhaustive crawlers [6, 19, 5, 10, 21]. However, existing strategies fail to meet the requirements of our problem.

Crawlers that focus the search based solely on the contents of the retrieved pages, such as the best-first crawler of [6], are not effective. Since forms are sparsely distributed even within a restricted domain, the number of forms retrieved per total of visited pages can be very low (see Section 4). Renee and McCallum [19] used reinforcement learning to build a focused crawler that is effective for sparse concepts. Instead of just considering the content of individual pages and crawling through pages that give immediate benefit, they train a learner with *features collected from paths leading to a page*. They do this by repeatedly crawling sample sites to build the connectivity graphs with the optimized paths to the target documents. However, this approach was designed for tasks which, unlike searching for hidden-Web databases, consist of well-defined search problems within a well-defined set of Web sites. For example, locating the name of the CEO in a given company site; and locating research papers available in the sites of computer science departments [19].

We propose a new crawling strategy that combines ideas from these two approaches. Similar to [6], we use a page classifier to guide the crawler and focus the search on pages that belong to a specific topic. But in order to further focus the search, like in [19], our crawler learns to identify promising links, including links whose benefit may not be immediate – in our case, a link classifier selects links that are likely to reach pages that contain forms (in one or more steps). However, instead of explicitly building the Web graph through repeated crawls of selected sites (which can be prohibitively expensive for a broad search), we rely on the backward crawling facilities provided by search engines in order to approximate this graph [3, 10]. In addition, based on form-specific characteristics, we introduce new stopping criteria that are very effective in guiding the crawler to avoid excessive speculative work in a single site.

Our experimental results over three distinct domains show that, even using an approximated connectivity graph, our crawler is more efficient (up to an order of magnitude) than a set of representative crawlers. Not only it is able to perform a broad search and retrieve a large number of searchable forms, but, for a fixed number of visited pages, it also retrieves a significantly larger number of forms than other crawlers. The experiments also show an added benefit of combining a focused crawl with the identification of links that lead to pages that contain forms: focusing the crawl on a topic helps improve effectiveness of the link classifier, since the features that are learned for links are often specific to a topic/domain.

The outline of the paper is as follows. We review related work in Section 2. In Section 3, we describe the architecture of our crawler, its implementation and underlying algorithms. An experimental evaluation and comparison against other approaches is given in Section 4. We conclude in Section 5, where we discuss directions for future work.

2. RELATED WORK

In what follows, we give an overview of previous works on focused crawling. We also briefly review approaches that address different aspects of retrieval and integration of hidden-Web data.

Focused Crawling. The goal of a focused crawler is to select links that lead to documents of interest, while avoiding links that lead to off-topic regions. Several techniques have been proposed to focus web crawls (see *e.g.,* [5, 6, 10, 19, 21]). In [6], Chakrabarti et al describe a best-first focused crawler (called *baseline* in the remainder of this paper) which uses a page classifier to guide the search. The classifier learns to classify pages as belonging to topics in a taxonomy (*e.g.,* dmoz.org). Unlike an exhaustive crawler which follows each link in a page in a breadth first manner, this focused crawler gives priority to links that belong to pages classified as relevant. Although this best-first strategy is effective, it can lead to suboptimal harvest rates, since even in domains that are not very narrow, the number of links that are irrelevant to the topic can be very high. An improvement to the baseline strategy was proposed in [5], where instead of following all links in relevant pages, the crawler used an additional classifier, the apprentice, to select the most promising links in a relevant page. The baseline classifier captures the user's specification of the topic and functions as a *critic* of the apprentice, by giving feedback about its choices. The apprentice, using this feedback, learns the features of good links and is responsible for prioritizing the links in the crawling frontier. Although our approach also attempts to estimate the benefit of following a particular link, there are two key differences. Whereas the apprentice only considers links that give immediate benefit, our link classifier learns to predict the distance between a link and a target page, and thus, our crawler considers links that may be multiple steps away from a target page. Besides, the goal of our link classifier is complementary to that of [5] – we want to learn which links lead to pages that contain searchable forms, whereas the goal of [5] to avoid off-topic pages. In fact, our approach would also benefit from such an apprentice, since it would reduce the number of off-topic pages retrieved and improve the overall crawling efficiency. Integrating the apprentice in our framework is a direction we plan to pursue in future work.

One issue with focused crawlers is that they may miss relevant pages by only crawling pages that are expected to give immediate benefit. In order to address this limitation, strategies have been proposed that train a learner with features collected from *paths leading to a page*, as opposed to just considering a page's contents [10, 19]. Rennie and McCallum [19] use reinforcement to train a classifier to evaluate the benefit of following a particular link. Their classifier learns features of links which include words in the title and body of the document where the link is located, and words in the URL, anchor and text in the neighborhood of the link. Given a link *(u,v)*, the classifier returns an estimate of the number of relevant pages that can be reached by following *(u,v)*. Diligenti et al [10] also collect paths to relevant pages. But their classifier estimates the distance from a page u to some relevant page w; it does not distinguish among the links in u – if the classifier estimates that a page u has high benefit, all pages v directly reachable from u are retrieved. Similar to [19], we estimate the benefit of individual links and select the ones that are more likely to reach pages that contain forms. However, in order to train our classifier, instead of explicitly building the Web graph through an exhaustive crawl of selected sites, we use the same optimization applied in [10] to build context graphs, *i.e.,* we rely on the backward crawling facilities provided by search engines in order to approximate the Web connectivity graph.

Retrieving and Integrating Hidden-Web Data. MetaQuerier [8] is a system that enables large-scale integration of hidden-Web data. It consists of several components that address different aspects of the integration. One of these components is a crawler for locating online databases, the *Database Crawler*. Unlike our approach, the Database Crawler neither focuses the search on a topic, nor does it attempt to select the most promising links to follow. Instead, it uses as seeds for the crawl the IP addresses of valid Web servers; then, from the root pages of these servers, it crawls up to a fixed depth using a breadth-first search. Their design choice is based on the observation that searchable forms are often close to the root page of the site [7]. As we discuss in Section 3, our crawler prioritizes links that belong to pages close to the root of a site. However, we also show that just limiting the depth of a breadth-first search leads to low crawling efficiency (see Section 4).

Raghavan and Garcia-Molina [18] proposed HiWe, a task-specific hidden-Web crawler. The key problem they addressed was how to automatically fill out structured Web forms. Although this pioneering work automates deep crawling to a great extent, it still requires substantial human input to construct the label value set table. In [1], we proposed a completely automated strategy to crawl through (simpler) unstructured forms (*i.e.,* keyword-based interfaces). Both crawlers can benefit from our system and use the returned form pages as the starting point for deep crawls.

To further automate the process crawling through structured forms,

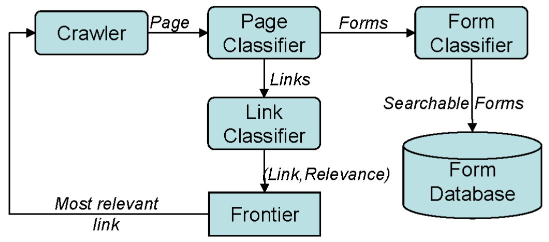


Figure 1: Form Crawler Architecture.

a hidden-Web crawler must *understand* the form interfaces so that it can generate meaningful submissions. Several techniques have been proposed that improve form understanding by finding matches among attributes of distinct forms (see *e.g.,* [12, 13, 24]). The forms we find can serve as inputs to these techniques.

Finally, it is worth pointing out that there are directories specialized on hidden-Web sources, *e.g.,* [4, 17, 20]. Hidden-Web directories organize pointers to online databases in a searchable topic hierarchy. Chang et al [7] note that these directories cover a small percentage of the hidden-Web databases; and they posit this low coverage is due to their "apparent manual classification". Being focused on a topic makes our crawler naturally suitable for automatically building a hidden-Web directory.

3. FORM-FOCUSED CRAWLER

To deal with the sparse distribution of forms on the Web, our Form Crawler avoids crawling through unproductive paths by: limiting the search to a particular topic; learning features of links and paths that lead to pages that contain searchable forms; and employing appropriate stopping criteria. The architecture of the Form Crawler is depicted in Figure 1.

The crawler uses two classifiers to guide its search: the page and the link classifiers. A third classifier, the form classifier, is used to filter out useless forms. The *page classifier* is trained to classify pages as belonging to topics in a taxonomy (*e.g.,* arts, movies, jobs in Dmoz). It uses the same strategy as the best-first crawler of [6]. Once the crawler retrieves a page *P*, if *P* is classified as being on-topic, forms and links are extracted from it. A form is added to the *Form Database* if the *form classifier* decides it is a searchable form, and if it is not already present in the Form Database.[1] The *link classifier* is trained to identify links that are likely to lead to pages that contain searchable form interfaces in one or more steps. It examines links extracted from on-topic pages and adds to the crawling frontier in the order of their importance. In the remainder of this section we describe the core elements of the system in detail.

3.1 Link Classifier

Since forms are sparsely distributed, by selecting only links that bring immediate return (*i.e.,* links that directly point to pages containing searchable forms), the crawler may miss "good" target pages that can only be reached with additional steps. Thus, the link classifier aims to identify links that may bring *delayed benefit*, *i.e.,* links that *eventually* lead to pages that contain forms. It learns the following features of links: anchor, URL, and text in the proximity of the URL; and assigns a score to a link which corresponds to the distance between the link and a relevant page that is reachable from that link.

Learning Distance through Backward Crawling. In order to learn the features of "good" paths, the link classifier needs examples of paths that lead to pages that contain searchable forms. These examples can be obtained from the connectivity graphs for a set of representative sites. Note that to build this graph, it may be necessary to perform exhaustive crawls over the sites. While this is possible for a small set of sites [19], the task would be extraordinarily expensive and time-consuming to apply in a large-scale crawling task that may involve thousands of sites.

Instead of building the exact connectivity graph, we get an approximation of this graph by performing a backward crawl using Google's "link:" facility, which returns pages that point to a given document [3, 10]. The backward crawl consists of a breadth-first search starting from pages that contain searchable forms. Each level *l+1* is constructed by finding all documents that point to the documents in level *l*. The resulting graph is an approximation because: Google does not provide complete connectivity information; and since the number of backlinks can be large, we select only a subset of these backlinks. Nonetheless, this approximation is sufficient to train the link classifier. As we show in Section 4, using multiple backward crawl levels leads to substantial gains in the form harvest rates.

For each level of the backward crawl, we extract the features of the links in pages that belong to that level. The classifier then learns the distance between a given link (from its associated features) and the target page which contains a searchable form. Intuitively, a link that matches the features of level *1* is likely to point to a page that contains a form; and a link that matches the features of level *l* is likely *l* steps away from a page that contains a form.

Feature Space Construction and Focused Crawl. The effectiveness of the link classifier is highly dependent on the features it considers. We experimented with different sets of features, as well as with different ways of extracting them. Due to space limitations, we discuss only the best of the strategies we examined.

For each level *l* in the backward crawl, we extract the words in the neighborhood of the links in *l*. We consider three *contexts*: URL, anchor, and text around the link. Since the number of extracted features tends to be large (and most of them have very low frequency), we remove stop-words and stem the remaining words. Then, for each context, we select only words with frequency larger than a fixed threshold. Note that features are associated with a context. For example, if the word "search" appears in both in the URL and in the anchor text of a link, it is added as a feature in both contexts.

Table 1 shows an excerpt of the feature space we constructed for the *jobs* domain; for each context, it shows the most common words and their frequencies. For illustrative purposes, we also show the common words in page title and text of the page where the link is located. Note that:

- Link features contain words that are clearly associated with the domain as well as with searchable forms: common words include "search", "career" and "job". We have observed similar behavior in other domains we explored, for example, for *cars*, common words in the anchor included "search", "used" and "car";
- The document text is a very good indicator of the relevance of a page. For example, words such as "job", "search" and "career" have very high frequencies in the document text in all levels;
- As a link gets farther from the target page, the frequency of clearly related words decreases. For example, whereas the anchor frequency of the word "job" in level 0 is 39, it goes down to 11 in level 2. And although the number of words that are apparently related to topic decreases with the distance, many of the words in the higher levels are still related to the topic.

[1]We check for duplicates because many Web sites have the same form interface in multiple pages.

level/field	URL	Anchor	Around the link	Title of page	Text of page	Number of pages
1	**job 111**	**job 39**	**job 66**	job 77	job 186	187
	search 38	**search 22**	**search 49**	career 39	search 71	
	career 30	ent 13	**career 38**	work 25	service 42	
	opm 10	**advanced 12**	**work 25**	search 23	new 40	
	htdocs 10	**career 7**	home 16	staffing 15	career 35	
	roberthalf 10	width 6	keyword 16	results 14	work 34	
	accountemps 10	popup 6	help 15	accounting 13	site 27	
2	**job 40**	**job 30**	**job 33**	job 46	job 103	212
	classified 29	**career 14**	home 20	career 28	search 57	
	news 18	today 10	ticket 20	employment 16	new 36	
	annual 16	ticket 10	**career 18**	find 13	career 35	
	links 13	corporate 10	program 16	work 13	home 32	
	topics 12	big 8	sales 11	search 13	site 32	
	default 12	list 8	sports 11	merchandise 13	resume 26	
	ivillage 12	find 6	search 11	los 10	service 22	
3	ivillage 18	**job 11**	**job 21**	job 17	font 37	137
	cosmopolitan 17	advertise 8	new 17	ctnow 8	job 33	
	ctnow 14	web 5	online 11	service 8	service 24	
	state 10	oak 5	**career 11**	links 7	cosmo 20	
	archive 10	fight 5	contact 10	county 7	new 19	
	hc-advertise 10	**career 5**	web 9	career 7	career 19	
	job 9	against 5	real 9	employment 7	color 16	
	poac 9	military 5	home 9	work 6	search 16	

Table 1: Excerpt of feature space for *jobs* domain. This table shows the most frequent words in each context for 3 levels of the backward crawl, as well as the total number of pages in examined in each level. The selected features used for the different contexts in the link classifier are shown in bold.

These observations reinforce our decision to combine a focused crawler, that takes the page contents into account, with a mechanism that allows the crawler to select links that have delayed benefit. The alternative of using a traditional breadth-first (non-focused) crawler in conjunction with our link classifier would not be effective. Although such a crawler might succeed in retrieving forms reachable from links with generic features (*e.g.,* “search”), it is likely to miss links whose features are domain-dependent (*e.g.,* “used”, “car”) if the frequencies of these features in the feature table are below the fixed threshold.

In addition to determining the distance of a particular link in relation to the target, *i.e.,* its category, it is also interesting to obtain the probabilistic class membership of this link in the category. This enables the crawler to prioritize links with higher probability of belonging to a given class. For this reason, we chose a naïve Bayes classifier [16] to classify the links. It is worthy of note that other crawlers have used this type of classifier to estimate link relevance [5, 19].

3.2 Page Classifier

We used Rainbow [15], a freely-available naïve Bayes classifier, to build our page classifier. In the Form Crawler, Rainbow is trained with samples obtained in the topic taxonomy of the Dmoz Directory (dmoz.org) – similar to what is done in other focused crawlers [5, 6]. When the crawler retrieves a page *P*, the page classifier analyzes the page and assigns to it score which reflects the probability that *P* belongs to the focus topic. If this probability is greater than a certain threshold (0.5 in our case), the crawler regards the page as relevant.

3.3 Form Classifier

Since our goal is to find hidden-Web databases, we need to filter out non-searchable forms, *e.g.,* forms for login, discussion groups interfaces, mailing list subscriptions, purchase forms, Web-based email forms. The form classifier is a general (domain-independent) classifier that uses a decision tree to determine whether a form is searchable or not.

The decision tree was constructed as follows. For positive examples we extracted 216 searchable forms from the UIUC repository [22], and we manually gathered 259 non-searchable forms for the negative examples. For each form in the sample set, we obtained the following features: number of hidden tags; number of checkboxes; number of radio tags; number of file inputs; number of submit tags; number of image inputs; number of buttons; number of resets; number of password tags; number of textboxes; number of items in selects; sum of text sizes in textboxes; submission method (post or get); and the presence of the string “search” within the form tag.

Algorithm	Error test rate
C4.5	8.02%
Support Vector Machine	14.19%
Naive Bayes	10.49%
MultiLayer Perceptron	9.87%

Table 2: Test error rates for different learning algorithms.

We performed the learning task using two thirds of this corpus, and the remaining one third was used for testing. We selected decision trees (the C4.5 classifier) because it had the lowest error rate among the different learning algorithms we evaluated [23]. The error test rates are shown in Table 2.

Cope et al [9] also used a decision tree to classify searchable and non-searchable forms. Their strategy considers over 550 features, whereas we use a much smaller number of features (only 14); and their best error rate is 15%, almost twice the error rate of our form classifier.

3.4 Crawling

The search frontier consists of *N* queues, where *N* is the number of levels used by the link classifier; the *i*-th queue is associated to the *i*-th level. The crawler prioritizes links that are closer to the target pages, *i.e.,* links that are placed in the queues corresponding to the lowest levels. Within a queue, links are ordered by the likelihood of belonging to the respective level. However, links that belong to pages close to the root of a Web site are given higher priority in the queue. Our decision to prioritize such links comes from the observation that forms often occur close to the main pages of

Web sites [7]. Note, however, that "just" prioritizing these pages is not enough – as we discuss in Section 4, a strategy that simply fixes the search depth is not effective.

Before the crawl starts, the seed links are placed in queue 1. At each crawl step, the crawler gets the most relevant link in the queues, *i.e.,* it pops the link with the highest relevance score from the first non-empty queue. If the page it downloads belongs to the domain, its links are classified by link classifier and added to the *persistent frontier*. When the queues in the crawling frontier become empty, the crawler loads a subset of the queues in the persistent frontier (the most relevant links are given priority). By keeping the persistent frontier separate, we ensure some fairness – all links in the crawling frontier will eventually be followed.

Stopping Criteria. Due to the sparseness of searchable forms, it is important for the Form Crawler to determine when to stop crawling a given site to avoid unproductive searches. The Form Crawler uses two stopping criteria: 1) the crawler leaves a site if it retrieves a pre-defined number of distinct forms; or 2) if it visits the maximum number of pages on that site. The intuition behind the first criterion is that there are few searchable forms in a hidden-Web site. Chang et al [7] observed that deep Web sites contain a small number of query interfaces. They estimate that, on average, a deep Web site has 4.2 query interfaces. Thus, after the crawler finds these forms, it can stop since it is unlikely to find additional forms. Since the Form Crawler performs a broad search, it visits many sites that may contain fewer than 4.2 forms, and sites that do not contain searchable forms. The second criterion ensures that the crawler will not waste resources in such sites. As we discuss below, these stopping criteria are key to achieving a high crawling efficiency.

4. EXPERIMENTAL EVALUATION

The key distinguishing feature of the Form Crawler is that it performs broad crawls to locate forms which are sparsely distributed over the Web. In our experiments, we compare the efficiency of our Form Crawler against that of three other crawlers:

- *Baseline*, a variation of the best-first crawler [6]. The page classifier guides the search: the crawler follows all links that belong to a page classified as being on-topic;
- *Fixed depth* follows the strategy adopted by the Database Crawler [8]. It performs a breadth-first search starting from the root page of a site up to a fixed depth. In our experiments, we set the depth to 3, since according to [7], most query interfaces (91.6%) appear within this depth;
- *Baseline SC* is an extension of the baseline crawler which adopts the stopping criteria described in Section 3.4.

In order to verify the effectiveness of using the distance between a link and a relevant target page as a predictor of link importance, we used different configurations for our Form Crawler:

- *Form Crawler with 1 level*, which corresponds to considering only links that give immediate benefit, *i.e.,* which lead to a form page in a single step;
- *Form Crawler with multiple levels*, which besides links that give immediate benefit also considers links that are multiple steps away from a target page. We ran experiments using from 2 to 4 levels; since the improvements obtained from level 4 are small, we only show the results for configurations with 2 and 3 levels.

We ran these crawlers over three distinct domains: jobs, cars and books. Seed pages for the actual crawl were obtained from the categories in the Google directory that correspond to these domains.

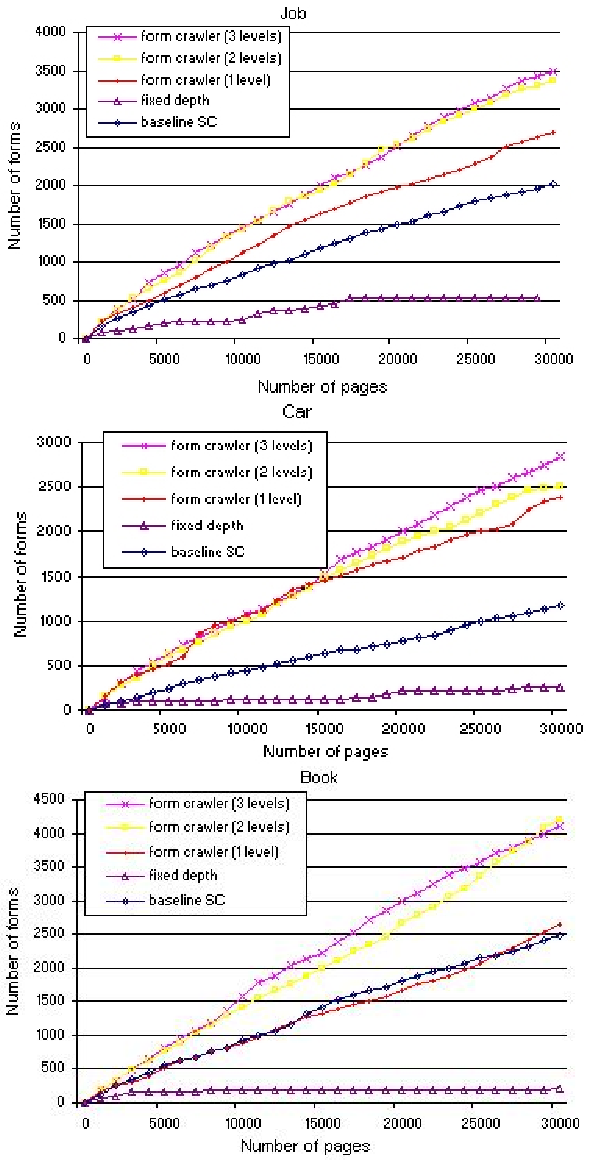


Figure 2: Performance of different crawlers for 3 domains.

For each domain, we created instances of the link and page classifiers. In order to train the link classifier, we obtained a sample of URLs of pages that contain forms (level 1) from the UIUC repository [22], and from these links we performed a backward crawl up to level 4. For the page and form classifiers we followed the procedures described in Sections 3.2 and 3.3, respectively.

An accepted measure for the effectiveness of a focused crawler is the amount of *useful work* it performs. For our crawler, this corresponds to the number of *distinct* relevant forms it retrieves as a function of the number of pages visited. Recall that the relevance of a form is determined by the form classifier (Section 3.3). Figure 2 shows the performance of the different crawlers we considered for each domain. The multi-level Form Crawler performed uniformly better than the other crawlers for all domains. In particular, multi-level always beats Form Crawler with only 1 level. Note that the amount of improvement varies with the domain. Considering the total number of forms retrieved from crawling 30,000 pages, using 3 versus 1 level leads to improvements that range between 20% and 110%. This indicates that the use multiple levels in the link classifier results in an effective strategy to search for forms. The two multi-level configurations (with 2 and 3 levels) have similar performance for both the jobs and books domains. The reason was that for these domains, the sample links in level 3 contain many *empty* features. This is illustrated in Table 1: very few of the selected features (shown in bold) are present in level 3. Note, however, that using 3 levels in the cars domain leads to a marked improvement – for 30,000 pages, the 3-level crawler retrieves 2833 forms, whereas the 2-level retrieves 2511 forms. The feature table for cars, unlike the ones for the other domains, contains many more of the selected features in level 3.

While running the baseline crawler, we noticed that it remained for a long time in certain sites, overloading these sites without retrieving any new forms. For example, in the jobs domain, after crawling 10000 pages it had retrieved only 214 pages. The baseline SC crawler avoids this problem by employing the stopping conditions we described in Section 3.4. The stopping conditions lead to a significant improvement in crawling efficiency compared to the standard baseline. Nonetheless, as Figure 2 indicates, by further focusing the search, our multi-level strategies retrieve a substantially larger number of forms than baseline SC.

The performance of the fixed-depth crawler was similar to that of the baseline crawler (without stopping conditions). As the density of forms in a site is very low, even performing a shallow crawl (using depth 3) can be inefficient. Our multi-level strategies outperform the fixed-depth crawler by over 1 order of magnitude for both cars and books, and for jobs, the gain is 5-fold.

5. CONCLUSION

In this paper we described a new crawling strategy to automatically discover hidden-Web databases. Our Form Crawler is able to efficiently perform a broad search by focusing the search on a given topic; by learning to identify promising links; and by using appropriate stop criteria that avoid unproductive searches within individual sites. Our experimental results show that our strategy is effective and that the efficiency of the Form Crawler is significantly higher than that of a representative set of crawlers.

Our initial prototype makes use of a decision-tree-based classifier to identify searchable forms. Although the test error rate for this classifier is low, it is hard to determine how well it performs with the actual forms retrieved by the Form Crawler. Since our crawls retrieve thousands of forms, it is not feasible to manually check all these forms. In future work, we plan to investigate automated techniques for evaluating the quality of the forms harvested by the Form Crawler.

Since our system uses learning algorithms to control the search, it can be used as a general framework to build form crawlers for different domains. We are currently using the Form Crawler to build a hidden-Web database directory – because it focuses the crawl on a topic, the Form Crawler is naturally suitable for this task.

Acknowledgments. This work was partially supported by the National Science Foundation under grant EIA 0323604, and by the Oregon University System.

6. REFERENCES

[1] L. Barbosa and J. Freire. Siphoning Hidden-Web Data through Keyword-Based Interfaces. In *Proc. of SBBD*, pages 309–321, 2004.

[2] M. K. Bergman. The Deep Web: Surfacing Hidden Value (White Paper). *Journal of Electronic Publishing*, 7(1), August 2001.

[3] K. Bharat, A. Broder, M. Henzinger, P. Kumar, and S. Venkatasubramanian. The connectivity server: Fast access to linkage information on the Web. *Computer Networks*, 30(1-7):469–477, 1998.

[4] Brightplanet's searchable databases directory. http://www.completeplanet.com.

[5] S. Chakrabarti, K. Punera, and M. Subramanyam. Accelerated focused crawling through online relevance feedback. In *Proc. of WWW*, pages 148–159, 2002.

[6] S. Chakrabarti, M. van den Berg, and B. Dom. Focused Crawling: A New Approach to Topic-Specific Web Resource Discovery. *Computer Networks*, 31(11-16):1623–1640, 1999.

[7] K. C.-C. Chang, B. He, C. Li, M. Patel, and Z. Zhang. Structured Databases on the Web: Observations and Implications. *SIGMOD Record*, 33(3):61–70, 2004.

[8] K. C.-C. Chang, B. He, and Z. Zhang. Toward Large-Scale Integration: Building a MetaQuerier over Databases on the Web. In *Proc. of CIDR*, pages 44–55, 2005.

[9] J. Cope, N. Craswell, and D. Hawking. Automated Discovery of Search Interfaces on the Web. In *Proc. of ADC*, pages 181–189, 2003.

[10] M. Diligenti, F. Coetzee, S. Lawrence, C. L. Giles, and M. Gori. Focused Crawling Using Context Graphs. In *Proc. of VLDB*, pages 527–534, 2000.

[11] L. Gravano, P. G. Ipeirotis, and M. Sahami. QProber: A system for automatic classification of hidden-Web databases. *ACM TOIS*, 21(1):1–41, 2003.

[12] B. He and K. C.-C. Chang. Statistical Schema Matching across Web Query Interfaces. In *Proc. of SIGMOD*, pages 217–228, 2003.

[13] H. He, W. Meng, C. T. Yu, and Z. Wu. Automatic integration of Web search interfaces with WISE-Integrator. *VLDB Journal*, 13(3):256–273, 2004.

[14] P. Lyman and H. R. Varian. How Much Information? Technical report, UC Berkeley, 2003. http://www.sims.berkeley.edu/research/projects/how-much-info-2003/internet.htm.

[15] A. McCallum. Rainbow. http://www-2.cs.cmu.edu/ mccallum/bow/rainbow/.

[16] T. M. Mitchell. *Machine Learning*. McGraw Hill, 1997.

[17] Profusion's search engine directory. http://www.profusion.com/nav.

[18] S. Raghavan and H. Garcia-Molina. Crawling the Hidden Web. In *Proc. of VLDB*, pages 129–138, 2001.

[19] J. Rennie and A. McCallum. Using Reinforcement Learning to Spider the Web Efficiently. In *Proc. of ICML*, pages 335–343, 1999.

[20] Search engines directory. http://www.searchengineguide.com/searchengines.html.

[21] S. Sizov, M. Biwer, J. Graupmann, S. Siersdorfer, M. Theobald, G. Weikum, and P. Zimmer. The BINGO! System for Information Portal Generation and Expert Web Search. In *Proc. of CIDR*, 2003.

[22] The UIUC Web integration repository. http://metaquerier.cs.uiuc.edu/repository.

[23] Weka 3: Data Mining Software in Java. http://www.cs.waikato.ac.nz/ ml/weka.

[24] W. Wu, C. Yu, A. Doan, and W. Meng. An Interactive Clustering-based Approach to Integrating Source Query interfaces on the Deep Web. In *Proc. of SIGMOD*, pages 95–106, 2004.

iFuice – Information Fusion utilizing Instance Correspondences and Peer Mappings

Erhard Rahm, Andreas Thor, David Aumueller, Hong-Hai Do, Nick Golovin, Toralf Kirsten
University of Leipzig, Germany
{rahm, thor, aumueller, hong, golovin, tkirsten}@informatik.uni-leipzig.de

ABSTRACT

We present a new approach to information fusion of web data sources. It is based on peer-to-peer mappings between sources and utilizes correspondences between their instances. Such correspondences are already available between many sources, e.g. in the form of web links, and help combine the information about specific objects and support a high quality data fusion. Sources and mappings relate to a domain model to support a semantically focused information fusion. The iFuice architecture incorporates a mapping mediator offering both an interactive and a script-driven, workflow-like access to the sources and their mappings. The script programmer can use powerful generic operators to execute and manipulate mappings and their results. The paper motivates the new approach and outlines the architecture and its main components, in particular the domain model, source and mapping model, and the script operators and their usage.

Keywords: Data integration, Peer-to-peer system, mappings

1. INTRODUCTION

Most proposed data integration approaches rely on the notion of a global schema to provide a unified and consistent view of the underlying data sources [10]. This approach has been especially successful for data warehouses, but is also used for virtual integration of web data sources. Unfortunately, the manual effort to create such a schema and to keep it up-to-date is substantial, despite recent advances, e.g. in the area of semi-automatic schema matching [14]. Likewise, the effort to integrate new sources is usually high making it difficult to scale to many sources or to use such systems for ad-hoc (explorative) integration. Notwithstanding the high effort associated with a global schema, it cannot guarantee good data quality at the instance level. Integrating the real data, e.g. during query processing over different web sources, may still require extensive data cleaning to achieve good results, e.g. to deal with duplicate data [15].

The iFuice approach (information Fusion utilizing instance correspondences and peer mappings) focuses on the instance data of different sources and mappings between them. Many web sources expose explicit, high quality instance-level correspondences to other sources, e.g. in the form of web links. Such correspondences represent one type of mapping iFuice uses to fuse information from different sources. Sources and mappings are related to a domain model to support semantically meaningful information fusion. The iFuice architecture incorporates a mapping mediator offering both interactive and script-driven, workflow-like access to the sources and their mappings. The script programmer can use powerful generic operators to execute and manipulate mappings and their results.

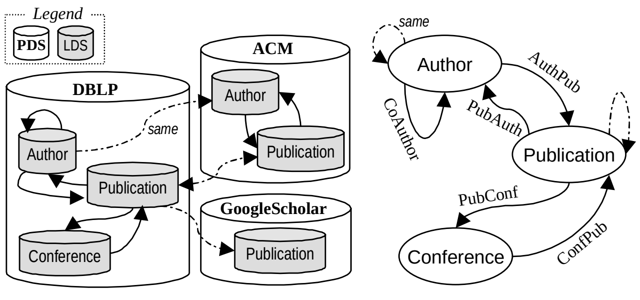
Bioinformatics is one area where this approach holds great promise. There are hundreds of web-accessible data sources on molecular-biological objects such as genes, proteins, metabolic pathways, etc. which highly cross-reference each other [5]. Creating a global schema for a sizable fraction of these sites is virtually impossible due to the high diversity, complexity and fast evolution of the data. The existing cross-references represent a low-level way to obtain additional information from other sources for a specific object, e.g. a gene. The additional information is typically of high quality since links are mostly established and maintained by domain experts. However, the manual navigation is unsuitable for evaluating large sets of objects, e.g. for gene expression analysis, so that there is a strong need for a more powerful integration approach. Moreover, the semantics of the links is typically not made explicit so that the user has to know exactly what kind of relationship they represent. The bioinformatics area also has a strong demand for experimental workflows to repetitively perform a series of analysis steps interrelating and aggregating information from different sources. The iFuice script facility aims at supporting such requirements with little development effort on the user side.

Eighth International Workshop on the Web and Databases (WebDB 2005), June 16-17, 2005, Baltimore, Maryland.

Instance-level cross-references are available in many other domains or can be generated with little effort, e.g. to interrelate bibliographic information, product descriptions and prices, etc. For simplicity we use examples from bibliographic data sources throughout this paper to illustrate the approach. Figure 1a shows three sample data sources and associated mappings. A physical data source (PDS), e.g. DBLP, may offer objects of different types. We call the object types of one PDS the logical data sources (LDS), e.g., Author, Publication and Conference as provided by DBLP. Object types combined across sources are represented in the abstract domain model (Figure 1b). Each mapping between source instances has a *mapping type* which is also represented in the domain model. Mappings map instances of an input object type to instances of an output object type, e.g. all mappings of type AuthorPubs relate author instances to their associated publications.

An important mapping type is signified by the *same-mappings* interrelating instances of the same object type across PDS, and provides a means to fuse the information for the respective instances. Typically, same-mappings are based on unique object ids, e.g. accession numbers in molecular-biological data sources or stable web URIs. Figure 1a indicates three such same-mappings, of which some already exist (e.g. DBLP links its author pages to the ACM author entries).

All mappings of the source-mapping model are executable, e.g. implemented by a query or web service. iFuice allows for explorative data fusion by browsing along these mappings, e.g. to derive from a DBLP author all publications from the directly or transitively connected LDS. The execution of several mappings and manipulation of their results can be specified within scripts to allow repeated executions for different input objects or to use the script as an implementation of a complex mapping. For example, we may want to have a script determining for a given conference



a) source mapping model b) domain model

Figure 1. Fusion scenario for a bibliographic domain (*same-mappings* are denoted by dashed lines)

X its most frequently referenced papers, e.g. to determine candidates for a 10-year best paper award. An iFuice representation of such a script is shown later. Informally, it locates conference X in DBLP, executes the PubConf mapping to get all publications of that conference, uses the same-mapping to Google Scholar to get the corresponding publications together with an attribute indicating the number of citations, sorting the publications on the number of citations, and returning the top-most publications. The example shows that mappings need to be executable on a set of input objects and return a set of output objects. Mapping execution can be restricted to specific sources or to all sources of a specific type for which a corresponding mapping implementation exists.

The main contribution of the paper is a new generic way to dynamic information fusion based on instance correspondences and executable mappings between sources. Source and mapping semantics are reflected in a domain model which is at a higher abstraction (ontological) level than a global schema and easier to construct. Mappings are executable on sets of objects and highly composable thereby supporting powerful aggregation of information over several sources. We propose different types of mappings on web data sources, including basic and aggregated mappings, same- and association mappings, and id- and query mappings. Furthermore, we introduce a set of declarative operators to execute the different kinds of mappings, to perform data aggregation (fusion), and to manipulate mapping results. Lastly, we show the usage of the operators for script programming.

In the next section, we introduce the representation of sources, mappings and the domain model, and how to add new sources and mappings. Section 3 introduces the iFuice operators on mappings and mapping results including operators for data fusion. We illustrate the use of operators by a script for the introductory example. Section 4 briefly outlines the architecture of the mapping mediator. In Section 5, we discuss related work. Finally, we conclude with a summary and outlook.

2. SOURCES AND MAPPINGS

In this section we describe the metadata used by the mapping mediator to provide uniform access to the sources and their mappings both for interactive exploration and script execution. The mediator's metadata is held in a repository and specified by a metadata model as shown in Figure 2. It consists of two main parts, a *source-mapping model* and a *domain model*. The source-mapping model describes both the accessible data sources and their associated mappings. The domain model specifies object types and mapping types. The sample models of Figure 1 conform to metadata model of Figure 2.

In the following, we describe the modeling and use of sources, mappings and the domain model. Finally we discuss the steps for adding a new source and mapping to the system. The description introduces several kinds of mappings, for which specific operators will be defined in Section 3.

2.1 Sources

We distinguish between a physical data source (PDS) and logical data source (LDS). A PDS can be a database, website, private user files or any other information base. A PDS can hold instances of different object types. We separate a PDS into LDS's each containing instances of exactly one object type of the domain model. The structure of a LDS is described by a set of attributes (Fig. 2). We only mandate the specification of an identifying key attribute per LDS to access its instances and to ensure that each instance is provided with a unique object id. Website instances are typically identified by URLs. Uniqueness for database instances can be established by concatenating instance key values with the ids of the corresponding PDS and LDS.

Requiring only an id attribute per LDS allows us to integrate a variety of heterogeneous data sources including unstructured and semi-structured data sources, e.g. websites. Furthermore, it makes it easy to add new data sources, and helps to insulate the mediator information against structural changes in the sources and thus to support a high degree of data source autonomy.

Each LDS has to provide a source-specific mapping for id-based instance access, i.e. to return for a given id the associated instance including all attribute values. We call these mappings *getInstance* mappings. The implementation of such mappings may be very simple (e.g. database lookup), but may also extract specific attributes from a webpage. The actual attributes returned are thus determined by the mapping implementation and depend on the current source content. The mapping mediator supports such variably structured result sets and their dynamic fusion with data from other sources at runtime.

In addition to the id attribute, further attributes can be optionally specified in the mapping mediator for enhanced functionality, e.g. to offer query access on the attributes or to select sources based on the availability of specific result attributes (e.g. number of citations for publications). Query access can optionally be provided by LDS-specific mappings, which we call *QueryInstances* mappings. The LDS attributes on which queries are supported should explicitly be registered in the mediator metadata.

Another optional specification is correspondences between attributes of two sources. Such correspondences are useful to enhance the fusion of instance values (see Figure 3). For instance, specifying that DBLP.Author.name corresponds to ACM.Author.author can help avoid author names appearing twice in author instances fused from DBLP and ACM. Attribute correspondences can also be used to map queries specified on one source to equivalent queries on other sources (query transformation).

2.2 Mappings

Mappings describe directed relationships between instances of two object types. They are used to uniformly interrelate instances within and between physical data sources irrespective of the underlying data management systems. The semantics of the mapping relationship is expressed by a mapping type of the domain model.

We distinguish between several kinds of mappings, including basic (simple) vs. aggregated mappings, same- vs. association mappings and id- vs. query mappings. *Basic mappings* interrelate instances of one input and one output LDS and return a result set from the output LDS. All mappings shown in Fig. 1a are such basic mappings. Note that the input and output LDS may be the

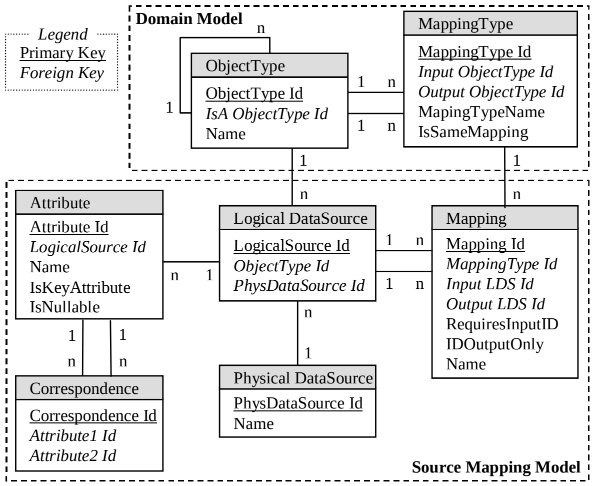


Figure 2. Metadata model of mapping mediator

same, e.g. for mappings of type CoAuthor or the source-specific getInstance and queryInstances mappings. *Aggregated mappings* are the result of mediator operations or scripts, and interrelate aggregated objects from several LDS of the same type. An example is a mapping to combine authors with their publications from several LDS (e.g., DBLP, ACM and Google Scholar).

Same-mappings represent semantic equality relationships between physical data sources at the instance level, i.e. each correspondence should refer to the same real-world object. They thus interrelate LDS of the same object type from different PDS (e.g. DBLP.Author and ACM.Author). These correspondences at the instance level are much more specific than attribute correspondences and can guide a high quality data fusion. Note that the composition of same-mappings results in new same-mappings which can thus be used to interrelate data from many sources. *Association mappings* are non-same-mappings and mostly represent domain-specific relationships between LDS of the same PDS (e.g. AuthorPubs). By composing them with same-mappings they can relate to and fuse with data of other PDS (see section 3).

Mappings can be further categorized on whether they are id-based or query-based with respect to their input instances (id- vs. query mapping). The output of basic mappings is always assumed to include the id of the returned instances. A mapping may in fact only return ids, e.g. as input for subsequent id-mappings and to limit the amount of data to transfer to the mediator and to process there.

Id-mappings interrelate ids (and thus instances) of two LDS or PDS and can easily be composed. The result of id-mappings can be represented by a set of instance correspondences (id1, id2). *Query mappings* are helpful to find relevant instances and their ids in the first place. One example are source-specific QueryInstance mappings. Their results include instance ids and can thus be combined with id-mappings to obtain related query results from different sources. The attribute values of a query result may also be used to query another source.

The mapping specification in Figure 2 only considers basic mappings for simplicity. It derives the distinction between same- and association mapping from the used mapping type of the domain model. The attribute *RequiresInputID* indicates whether or not the mapping is id-based.

The implementation of mappings can use other mappings, utilize database queries, etc. To hide implementation differences, iFuice mappings are uniformly encapsulated as web service operations and use XML for data exchange. Ideally, id-mappings, including same-mappings, can be based on existing instance correspondences such as web links. Alternatively, they may be implemented by a query mapping, e.g. to use instance values from one source (e.g. obtained from a getInstance mapping) to search for corresponding instances of a second source (input for query mapping). For instance, the same-mapping between DBLP publications and Google Scholar can be implemented by using the name and author of a DBLP publication as a keyword query to Google Scholar.

For improved performance, the results of id-mappings, i.e. the set of instance correspondences, may be stored or cached in binary (id/id) mapping tables [8]. Composition between such mappings then becomes a join operation. Materialized id-mappings can also be inverted, even for n:m cardinalities (e.g., an AuthorPub mapping can be derived from an Id-based PubAuthor mapping and vice versa).

2.3 Domain model

The domain model defines domain-specific object types and mapping types to semantically (ontologically) categorize data sources and mappings. A hierarchical (taxonomical) categorization of object types is possible to classify sources in more detail (e.g., conferences based on discipline). We do not include attributes for object types to accommodate a large variety of data sources and to make it much easier to construct the domain model than a global schema. In many cases, we expect a small set of object types to be sufficient. New object types may be added as needed to accommodate new sources, i.e. the domain model can be incrementally extended in a bottom-up fashion. A mapping type interrelates two object types. A special attribute indicates whether the mapping type represents same-mappings (semantic equality relationship).

2.4 Adding Sources and Mappings

One goal of iFuice is to make it easy to add new sources and mappings. A new physical data source requires to register at least one logical data source. Registering a LDS requires to assign it to the corresponding object type, specification of an id attribute, and provision of a getInstance mapping. Furthermore, a peer mapping P to at least one other LDS should be provided to permit data fusion with other sources. Optionally, a QueryInstances mapping can be provided, and additional attributes (e.g. known output attributes of P or for query input) and attribute correspondences can be specified. For a LDS of a new object type, the domain model must be extended with the corresponding object type (e.g. Journal) and at least one associated mapping type (e.g. JournalPubs).

Provision of a new mapping requires its registration at the mapping mediator. This involves the specification of the mapping characteristics and possibly the registration of a new mapping type in the domain model. The mapping must be executable, i.e. an implementation must be provided. The mapping implementation can hide many details of the underlying data sources and typically exposes only selected input and output attributes at the interface. As discussed, we do not require that the input and output attributes of a mapping be registered in the source-mapping-model.

To illustrate the ease and benefit of providing data sources and mappings consider the following simple example: A user keeps a list of her favorite authors (including handpicked information like e-mail address or nationality, which are accessible by a *getInstance* operation) in a local file and wants to bind it to the mapping mediator so that she can periodically check the information about the authors' publications. This can be achieved by establishing a same-mapping between the local file and the LDS *DBLP author*, e.g. by providing a list of DBLP URLs. Thereafter the existing mappings between DBLP, ACM and Google Scholar can be used to gather the information of interest.

3. OPERATORS AND SCRIPTS

Interactive users and script programmers should not be limited to the execution of one mapping at a time but are provided with more powerful operators including data fusion capabilities. Therefore, the iFuice mapping mediator supports a variety of operators which can be used within script programs or to implement derived mappings. This idea is inspired by the script approach for model management, e.g. as implemented in Rondo [11]. While Rondo focuses on metadata manipulation, iFuice provides operators for mapping execution and manipulation of instances.

We designed operators of different complexity to allow users to focus access on specific data sources and mapping paths. Compared to transparent access on many sources, this not only helps improve performance but is also important for user acceptance and data quality. This is because users often have specific preferences for some data sources, and the cleanliness of merged data tends to decrease with more sources. Therefore we designed two sets of operators, one for processing basic mappings returning objects from one source (getInstances, traverse, map, queryInstances, queryTraverse queryMatch) and one for aggregated mappings and aggregated objects (aggregateSame, aggregateQueryTraverse aggregateMap, fuseAttributes). In both cases we have a set of operators to process the respective results, similar to query languages (union, intersect, project, sort, join, ...). All operators are set-oriented, i.e. they work on sets of simple or aggregated input objects and determine a set of result objects or a mapping.

The next two subsections introduce these two sets of operators. In 3.4 we present a script program using the operators.

3.1 Operators for basic mappings

Basic mappings relate instances of one input LDS with instances of one output LDS, i.e. we obtain a homogeneous set of result objects. An *object* (instance) o_i consists of a unique id plus a (possibly empty) list of attribute values. The id is assumed to also identify the logical data source to which the object belongs. We first introduce operators for id-mappings and then for query mappings.

3.1.1 Operators for id-mappings

Let L_1, L_2, … denote logical data sources, O_1, O_2, … sets of L_1 objects, L_2 objects … and m_1, m_2, … id-mappings (same- or association mappings).

traverse $(O_1, m_2, \ldots, m_k) \rightarrow O_k$

traverse $(O_1, m_2, \ldots, m_k) = m_k(m_{k-1}(\ldots m_2(O_1)))$

consecutively executes m_2, m_3 … thereby traversing via $L_2 - L_3$ to L_k. Of course, the input source of m_i must correspond to the output source of m_{i-1}. Note that both same- and association mappings can be used within a traversal path. For same-mappings the LDS names can be used instead of the mapping names. The output is required to be a set, i.e. no duplicates are allowed. *Example*: Let O_1 be a list of DBLP author URLs, traverse (O_1, ACM, ACMAuthorPub) returns their corresponding ACM publications. traverse (O_1, DBLPAuthPub, DBLPPubConf) returns the conferences in which the authors in O_1 have published (without returning the authors and publications).

For same-mappings we provide a variation of traverse

traverseSame $(O_1, LDS_k) \rightarrow O_k$

It is not restricted to a single traversal path but considers all paths of same-mappings from the input LDS_1 to LDS_k and takes the union of their LDS_k results.

The traverse and traverseSame operators only return the instances of the last LDS on the mapping path which can thus be the input for other operators on objects. Frequently one wants to see the correlations between objects of the first and last source. This is achieved by

map$(O_1, m_2, \ldots, m_k) \rightarrow O_1 \times O_k$

map$(O_1, m_2, \ldots, m_k) = \{(o_1,o_k) | o_1 \in O_1,\ o_k \in$ traverse$(\{o_1\}, m_2, \ldots, m_k)\}$

In the special case k=2, map returns the instance correspondences of a single mapping (there may be just id-id combinations, e.g. for same-mappings). For more than one mapping, the semantics corresponds to that of a classical compose operation. *Example*: map (O_1, DBLPAuthPub, DBLPPubConf) returns authors together with the conferences in which they published, i.e. these different instances are 'fused' together by the mapping result.

The support operator

getInstances $(O_1) \rightarrow O_1$

determines for the input instances in O_1 the available attribute values. This operator usually is applied to objects that only hold an id value or a subset of attributes. *Example*: Given a list of DBLP author URLs, getInstances adds attribute values, e.g. name, no. of co-authors etc. In the previous operators, the implementation of the mappings, especially m_k, determines which attributes are present in the output instances. Applying the getInstances operator on these instances helps to obtain additional attribute values if needed.

3.1.2 Operators for (basic) query mappings

In iFuice, queries are posed for one source and can then be propagated to other sources by applying id-mappings or query matching. For querying a source we use

queryInstances : $(L_1, \{cond\}) \rightarrow O_1$

which returns all object instances (i.e. at least their ids) from L_1 which fulfill the given set of attribute conditions $\{cond\}$.

To propagate a query, we use derived operators combining queryInstances with traverse or traverseSame.

queryTraverse $(L_1, \{cond\}, m_2, \ldots, m_k)$
 = traverse (queryInstances $(L_1, \{cond\})$, $m_2, \ldots, m_k) \rightarrow O_k$

queryTraverseSame $(L_1, \{cond\}, L_k)$
 = traverseSame (queryInstances $(L_1, \{cond\})$, L_k))

Example: queryTraverseSame (DBLP, {name= 'Bernstein'}, ACM) returns the ACM author objects of all DBLP authors with that name.

Operator queryMatch transforms an input query for one source to an equivalent one on a second source:

queryMatch $(L_1, \{cond\}, L_2)$ = queryInstances (L_2, attrTransf $(\{cond\})$)

The function attrTransf utilizes specified attribute correspondences to map the L_1 query condition into a corresponding L_2 query condition. Hence, queryMatch is only applicable to L_2 sources for which a queryInstances implementation and attribute correspondences have been provided. This operator typically is used to build the union of the source-specific results which leads to aggregated objects.

3.2 Operators for aggregated mappings

Same-mappings identify semantically equivalent objects (synonyms, duplicates) which should be combined to reduce redundancy and merge (complement) the available information from different sources. We separate this into two steps, called aggregation and fusion. Figure 3 exemplifies this for two semantically equivalent publication objects. In step 1 we combine them into one aggregated object which is a combination of all attributes from the original objects. In step 2 we fuse the attributes to reduce

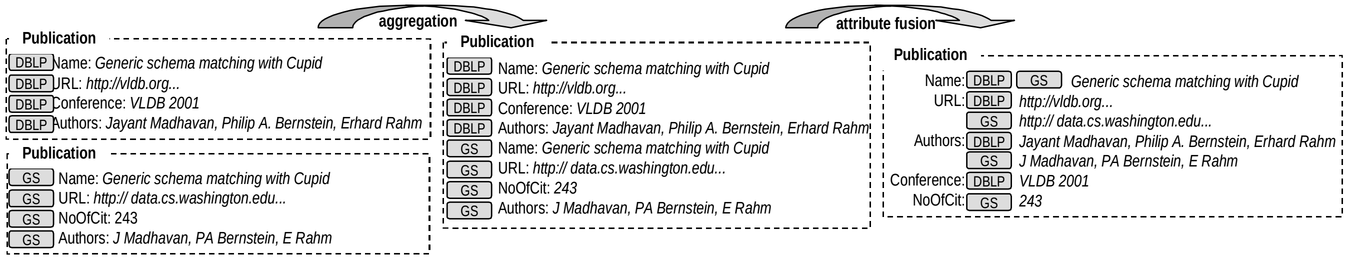

aggregation
attribute fusion
Publication
DBLP Name: Generic schema matching with Cupid
DBLP URL: http://vldb.org...
DBLP Conference: VLDB 2001
DBLP Authors: Jayant Madhavan, Philip A. Bernstein, Erhard Rahm
GS Name: Generic schema matching with Cupid
GS URL: http:// data.cs.washington.edu...
GS NoOfCit: 243
GS Authors: J Madhavan, PA Bernstein, E Rahm

Figure 3. Example for aggregation and attribute fusion

redundancy without losing information. The first step is usually the most difficult one, but is well supported in iFuice by the same-mappings thereby facilitating good data quality. The second step can use attribute correspondences or actually analyse the existing values for merge possibilities.

Most iFuice operators for aggregated mappings deal with aggregated objects where all attribute values are still available for further processing. The fusion of attribute values is performed by a separate operator, fuseAttributes, which is best applied before returning aggregated objects to the user.

Let $\{o_1, o_2, o_3, \ldots\}$) be a set of semantically equivalent objects of object type T from one or more logical data sources. The aggregation of these objects agg $(\{o_1,o_2,o_3,\ldots\}) = (o_1\text{-}o_2\text{-}o_3\text{-}\ldots)$ is called an *aggregated object* and also refers to object type T.

disagg $(o_1\text{-}o_2\text{-}o_3\text{-}\ldots) = \{o_1,o_2,o_3,\ldots\}$ returns the components of an aggregated object.

Aggregating objects of the same type

Let AO_1, AO_2, … be sets of aggregated objects of type T, and L_2 a LDS of object type T. The operator

$$\text{aggregateSame}\ (AO_1, L_2) \rightarrow AO_2$$
$$=\{\text{agg}(ao_1,\{\text{traverseSame}\ (\{o_1\},L_2) \mid o_1\in \text{disagg}(ao_1)\}) \mid ao_1\in AO_1\}$$

aggregates all AO_1 objects with semantically equivalent objects in L_2 by evaluating the same-mappings from the input objects to the corresponding L_2 objects. Note that the operator can also be applied to simple input objects from one input LDS. Note further that the same-mappings implement the duplicate detection and can thus support efficient and high quality data aggregation. The definition of aggregateSame can easily be generalized to more than one target LDS since the operator works on sets of aggregated objects, e.g. the output of a previous aggregateSame execution.

Combining the results for a propagated query at different sources leads to aggregated objects and is supported by

$$\text{aggregateQueryTraverse}\ (L_1,\{cond\}, L_k)$$
$$= \text{aggregateSame}\ (\text{queryInstances}\ (L_1,\{cond\}), L_k).$$

The standard set-oriented (relational) operators intersect, diff, union, restrict, project, sort etc. can be extended to deal with aggregated objects and duplicates. Due to space constraints we only define intersection.

$$\text{intersect}\ (AO_1, AO_2) = \{(ao_1\text{-}ao_2)|ao_1\in AO_1,\ ao_2\in AO_2,\ ao_1\approx ao_2\}$$

Thereby, $\approx$ denotes that two aggregated objects are semantically equivalent. This is the case if they share at least one component object or if they are related by a same-mapping.

Aggregating objects of different types

Association mappings typically interrelate objects of different types which should not be aggregated together like equivalent objects. For these mappings, we generalize the traverse operation to both mapping types and aggregated objects.

$$\text{aggregateTraverse}\ (AO_1, mt) \rightarrow AO_2$$
$$=\{\text{agg}(\{\ \text{traverse}(\{o_k\},m)|\ o_k\in \text{disagg}(ao_k),\ m \text{ of } mt\ \})|\ ao_k\in AO_1\}$$

applies all association mappings of type mt for all objects in AO_1 and aggregates the resulting objects. Similarly, we generalize the map operator for association mappings and aggregated objects to obtain binary aggregated mappings:

$$\text{aggregateMap}\ (AO_1, mt) \rightarrow AO_1 \times AO_2$$
$$=\{(ao_1, ao_2) \mid ao_1\in AO_1,\ \ ao_2\in \text{aggregateTraverse}\ (\{ao_1\}, mt)\}$$

Example: Given a set AO_1 of aggregated objects of DBLP and ACM authors, aggregateMap (AO_1, AuthPubs) returns author-publication pairs of aggregated objects that contain object instances from DBLP or ACM or both.

For aggregated mapping results $MR_1 \subseteq AO_1 \times AO_2$ and $MR_2 \subseteq AO_3 \times AO_4$ the operators join and compose are defined as follows

$$\text{join}\ (MR_1, MR_2)$$
$$= \{(ao_1,(ao_2\text{-}ao_3),ao_4)|(ao_1,ao_2)\in MR_1,(ao_3,ao_4)\in MR_2,\ ao_2\approx ao_3\}$$
$$\text{compose}(MR_1,MR_2) = (\{ao_1,ao_4\}|(ao_1,ao_2,ao_4) \in \text{join}\ (MR_1, MR_2))$$

The given join semantics refers to an inner join, but left outer join etc. can be specified analogously. Join and compose also need a duplicate detection to aggregate semantically equivalent objects.

3.3 Script Example

A script is a sequence of operator calls. Each operator call stores the results into a variable (denoted by a '\$'-prefix). The following simple script example presents an approach to determine candidates for the 10-Year Best Paper Award.

```
$SIGMODPubs := queryTraverse (LDS=DBLP.Conf, {Name="SIGMOD 1995"},
                                                        DBLPConfPubs)
$CombinedConfPub:= aggregateSame ($SIGMODPubs, GoogleScholar)
$CleanedPubs := fuseAttributes ($CombinedConfPub)
$Result := sort ($CleanedPubs, "NoOfCitings")
```

In this example, step 1 uses a queryTraverse operation to query on the LDS DBLP.Conf to determine the DBLP id for the conference of interest and traversing to the associated publications. The used mapping DBLPConfPubs is assumed to determine complete instances. Step 2 utilizes the same-mapping on publications between DBLP and Google Scholar to aggregate the DBLP values with the corresponding instances in Google Scholar. In step 3 we clean the aggregated objects by applying fuseAttributes. Finally, we sort the resulting set of fused publications on the attribute denoting the number of citations. The top items/publications in the final result set indicate likely candidates for the 10-Year Best Paper Award.

Since operators can process many input objects at a time, the script does not only apply to a single conference but many. To determine the most-cited publications of, say, a whole conference series, one could use a modified query in step 1 to select these conferences, e.g. SIGMOD or VLDB. The rest of the script can remain unchanged.

4. MEDIATOR ARCHITECTURE

Figure 4 gives an overview of the iFuice mediator architecture. Its main components are the *repository* (already described in Section

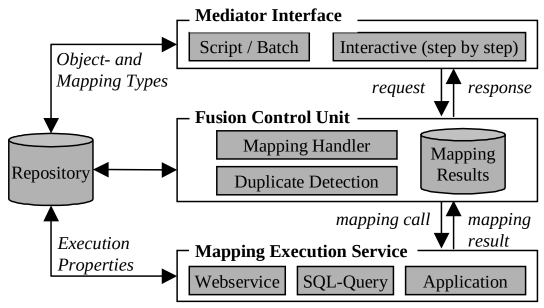


Figure 4. Architecture of the iFuice mediator

2), *mediator interface (MI),* the *fusion control unit (FCU),* and the *mapping execution service (MES)*. The MI provides two modes of operation. The interactive mode supports an explorative approach where users can execute mappings step by step. The script-based mode defines a batch of execution steps for the mediator, returning a final result set. Operations are executed within the FCU, the central unit of the system. The mapping handler coordinates single mapping calls according to the appropriate mapping definition of the metadata model. It manages (temporary) mapping results for further operations. It also performs duplicate handling for aggregating objects. The mapping execution service actually executes the called mappings and returns their results to the FCU.

As a proof of concept we have implemented a first subset of the mediator functionality for interactive mapping execution (explorative navigation). The prototype is implemented in Java and utilizes a relational database system for the repository and mapping results. It can execute mappings and operation sequences as in the examples shown, including the example on the 10-year best paper award.

5. RELATED WORK

Most previous data integration approaches for web data sources utilize a global schema and a query mediator. In contrast to iFuice, these approaches do not utilize peer mappings and instance correspondences. Moreover, the global schema tends to be much more complex than our domain model leading to increased effort to add sources or to deal with changing sources.

More related to our approach is recent work on P2P databases and biological databases. P2P prototypes such as PeerDB and Piazza [12][17] focus on query processing across peer mappings without a global schema. PeerDB propagates IR searches, whereas Piazza reformulates queries based on metadata mappings. Queries refer to the peer schema where the query was initially posed. By contrast, iFuice focuses on instance-level correspondences and can apply a variety of executable peer mappings including queries. Moreover, we support a set of powerful operators (including aggregation operators) and the execution of script programs.

Several integration approaches in the bioinformatics area [16], [9], [7] utilize cross-references at the instance level to combine data from different sources. Systems like SRS [4] and our GenMapper prototype [1] materialize instance correspondences in mapping tables for improved performance. These efforts lack a sufficient consideration of the semantics of the cross references but expect the users to know what the cross references mean. The iFuice domain model differentiating different mapping types, including different same-mappings, allows a much more focused data fusion. To our knowledge, the proposed framework of mapping operators and scripts is also unknown so far in the bioinformatics domain.

SEMEX [3] is an interesting personal information management system which utilizes a domain model and mappings similar to our approach. However, it only deals with centrally stored data, while we integrate data from different web data. Most previous work on data cleaning was done in the data warehouse area with a focus on duplicate identification [1][15]. The use of semantic object-ids for data integration in the TSIMMIS mediator [13] has similarities to the utilization of ids in our *same-mappings*.

6. CONCLUSION AND OUTLOOK

iFuice combines a set of techniques to a new approach for integrating information from diverse web data sources. It does not depend on a global schema and utilizes explicit instance correspondences and executable peer mappings. We proposed the use of a domain model and a mapping mediator to control the execution of a variety of such mappings. Furthermore, we introduced a set of powerful operators for mapping execution and data aggregation. An initial prototype for interactive mapping execution showed the viability and flexibility of the approach.

In future work, we will fully implement the outlined approach and investigate techniques based on caching mapping tables to improve performance. We plan to adopt the iFuice implementation to different domains and to support integration of both web data sources and local / private data sources.

Acknowledgements. A. Thor and N. Golovin are funded by the German Research Foundation (DFG) within the Graduiertenkolleg “Knowledge Representation”. H. Do and T. Kirsten are supported by DFG grant BIZ 6/1-1. Phil Bernstein and Sergey Melnik gave helpful comments.

7. REFERENCES

[1] Chaudhuri, S. et al.: *Robust and efficient fuzzy match for online data cleaning*. Proc. SIGMOD 2003

[2] Do, H.-H., Rahm, E.: *Flexible integration of molecular-biological annotation data: The GenMapper Approach*. Proc. of EDBT 2004

[3] Dong, X., Halevy, A. Y.: *A platform for personal information management and integration.* CIDR 2005

[4] Etzold, T. et al.: *SRS: An integration platform for databanks and analysis tools in bioinformatics*. In [9]: 109-145.

[5] Galperin, M.Y.: *The molecular biology database collection - 2004 update*. Nucleic Acids Research 32, Database issue, 2004.

[6] Greco, S. et al: *Integrating and managing conflicting data*. Proc. Of Conf. on Perspectives of System Informatics. 2001

[7] Hernandez, T, Kambhampati, S: *Integration of biological sources: current systems and challenges ahead*. SIGMOD Record 33(3), 2004

[8] Kementsietsidis, A. et al.: *Mapping data in peer-to-peer systems:semantics and algorithmic issues*. Proc. SIGMOD 2003

[9] Lacroix, Z., Critchlow T. (Eds.): *Bioinformatics: Managing Scientific Data*. Morgan Kaufmann, 2003

[10] Lenzerini, M: *Data integration: a theoretical perspective*. Proc. PODS 2002

[11] Melnik, S. et al.: *Developing metadata-intensive applications with Rondo*. Journal on Web Semantics, 2003

[12] Ng, W. S., *et al.*: *PeerDB: A P2P-based System for Distributed Data Sharing.* Proc. ICDE 2003

[13] Papakonstantinou, Y. et al.: *Object Fusion in Mediator Systems*. Proc. VLDB 1996

[14] Rahm, E., Bernstein, P. A.: *A survey of approaches to automatic schema matching*. VLDB Journal, 10(4), 2001

[15] Rahm, E., Do, H.-H.: *Data cleaning: problems and curren approaches.* IEEE Bull. Techn.Com. Data Engineering, 23 (4), 2000

[16] Stein, L. D.: *Integrating biological databases.* In Nature Review Genetics, 4, 2003

[17] Tatarinov, I., *et al.*: *The Piazza peer data management project.* SIGMOD Record, 32(3), 2003

Managing Integrity for Data Exchanged on the Web

Gerome Miklau   Dan Suciu
University of Washington
{gerome, suciu}@cs.washington.edu

ABSTRACT

The World Wide Web is a medium for publishing data used by collaborating groups and communities of shared interest. This paper proposes mechanisms to support the accuracy and authenticity of published data. In our framework, publishers annotate data with virtually unforgeable evidence of authorship. Intermediaries may query, restructure, and integrate this data while propagating the annotations. Final recipients of the data may then derive useful conclusions about the authenticity of the data they receive.

1. INTRODUCTION

The emergence of diverse networked data sources has created new opportunities for the sharing and exchange of data. In particular, the Web has become a medium for publishing data used by collaborating groups and communities of shared interest. Once published, it is common for other parties to combine, transform, or modify the data, and then republish it.

In such distributed settings there are few mechanisms to support users in trusting the accuracy and authenticity of data they receive from others. To address this problem, we investigate guarantees of *data integrity* for exchanged data. Integrity is an assurance that unauthorized parties are prevented from modifying data[1]. Integrity benefits both the authors of data (who need to make sure data attributed to them is not modified) and the consumers of data (who need guarantees that the data they use has not been tampered-with).

After publication, the owner of data can never directly prevent modification of the published data by recipients. But it *is* possible to annotate published data with virtually unforgeable evidence of its authenticity that can be verified by recipients. Data authors need techniques which allow them to annotate data with claims of authenticity. These claims should be difficult to forge or transfer, and must be carried along with the data as it is exchanged and transformed. Subsequent users must then be able to derive useful integrity guarantees from query results containing these claims. We explore here techniques to accomplish these goals.

[1]We adopt the meaning of integrity common to information security (not databases).

Copyright is held by the author/owner. Eighth International Workshop on the Web and Databases (WebDB 2005), June 16-17, 2005, Baltimore, Maryland.
.

To illustrate the importance of integrity in data exchange, we describe two applications: scientific data exchange and personal identity databases.

Scientific data exchange. As a representative scientific domain we consider the field of molecular biology. From primary sources containing original experimental data, hundreds of secondary biological sources [2] are derived. The secondary sources export views over primary sources and/or other secondary sources, and usually add their own curatorial comments and modifications [23]. These databases are often published on the Web, as structured text files – not stored in proprietary systems or servers that can provide security guarantees. The data consumers are scientists, and a significant fraction of research takes place in so-called "dry" laboratories using data collected and curated by others. An illustration of this scenario is provided in Figure 1.

The threat of malicious tampering with the data is usually not a primary security concern in this setting. Instead, the main issues are attributing and retaining authorship and avoiding the careless modification of data. To the best of our knowledge, security properties are rarely provided in scientific data exchange. Although in some cases authorship is traced, there is little evidence or verification of authorship.

Personal identity databases. A large class of databases, which we call *personal identity databases*, have in common the fact that they contain personally identifying information about individuals (e.g. census data, medical databases, organizations' member lists, business customer data). Such databases can be viewed as intermediate sources that collect data from primary source individuals donating their personal data. The secondary sources may disseminate or further integrate this identity data. For example, individuals present their personal data to a hospital database when admitted. Hospitals add treatment data and send patient records to an insurance company that integrates it with data of patients from other hospitals.

Integrity is critical as the data migrates between organizations: only authorized parties should be able to modify data, or create new records, and those parties should be identifiable after the fact. If an individual

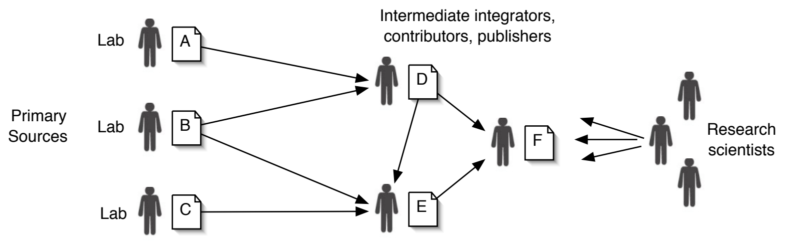


Figure 1: Data publishing and exchange scenarios for scientific data management.

applies for insurance coverage, the insurance company will evaluate the cost of insuring the individual based on the data in its database. An individual should have the right to verify the accuracy of that data. This can be accomplished if the data carries integrity metadata, and the insurance company is required to present the data and its evidence of integrity. To continue the example, some of the individual's personal data will be signed by the individual himself, some will be signed by individual's doctors, etc. Regulations need to be in place that make it illegal for the insurance company to base coverage decisions on data that is not verifiably authentic.

Origin authenticity and Origin-critical data

Our particular focus is an aspect of data integrity called *origin authenticity*: an assurance that data comes from an attributed source (and that it has not been modified from its original state) [16].

There are two important threats to origin authenticity. The first is the threat of copying data published by an author Alice. For example, an adversary, Mallory, may duplicate the data received from Alice, remove the original evidence of attribution and claim himself as the author. This threat, while important, is not our focus here. (It requires substantially different techniques like watermarking [1], or legal measures [13].)

Instead we focus on the threat of an adversary tampering with data authored by Alice. This can happen in two related ways. Alice may author some data, which is properly attributed to her, but Mallory changes the data while keeping the attribution. Or more directly, Mallory forges the attribution itself, applying it to data of his choosing. To justify our focus on this threat, we describe next *origin-critical data*, for which tampering is the primary concern.

Origin-critical data is data whose value or utility depends critically on its authenticity or the authority of its source[2]. A table of stock recommendations like {(IBM, buy), (MSFT, sell)} is an example of origin-critical data. The raw data can easily be fabricated or duplicated and therefore is worth little if its source is unknown or the claim of its origin is untrusted. For instance, proof that the author of the data is an expert equity analyst makes the data valuable. If authored by a high-school investment club however, the stock ratings are substantially less useful. Therefore, its origin is critically important. An mp3 file is an example of data whose origin is not critical. The utility of the mp3 file seems to consist solely in the contents of the file. Its source, and the authenticity of its attributed source, are usually not relevant to the listener who is happy to download the file from an unknown party on the Internet.

Many kinds of digital data are origin-critical, and a primary integrity concern is to maintain and manage verifiable claims of authorship. For example, in scientific data management the consumers of data (the scientists) are reluctant to use data that does not come from a reputable source. In e-business transactions, a party may not be willing to act on data received if it is not verifiably authentic. For origin-critical data it is usually in the interest of the participants to retain evidence of origin authenticity.

Our goal is therefore to develop a framework to (1) allow authors to annotate data with evidence of authorship, (2) allow recipients to query, restructure, and integrate this data while propagating the evidence, and (3) enable recipients to derive useful conclusions about the authenticity of the data they receive. In support of these goals, first a pair of integrity annotations are proposed, which are applied to data to represent useful claims of origin authenticity. Then cryptographic techniques are described that can be used to support these annotations so that the claims are not forgeable or transferable. Finally, we assess the requirements and challenges of managing data with integrity annotations.

2. INTEGRITY ANNOTATIONS

Annotations are applied to fragments of a database. For relational data, a fragment may be an individual attribute, a tuple, or a set of tuples. For XML data, a fragment may be a complex subtree, or a forest of subtrees. We propose in this section two related forms of annotation – signature and citation – which are used by data authors to represent claims of origin authenticity. We describe the semantics of these annotations and their use. In the next section we describe cryptographic techniques to implement these forms of annotation.

Signatures. On paper, signatures are intended as proof of authorship or an indication of agreement with the contents of a document. Signatures serve their purpose because they have some or all of the following properties [27]: the signature is unforgeable, the signature cannot be reused on another document, and the signed document is unalterable. For the present discussion we will assume a basic signature primitive possessing these

[2]Some of these ideas were inspired by [5].

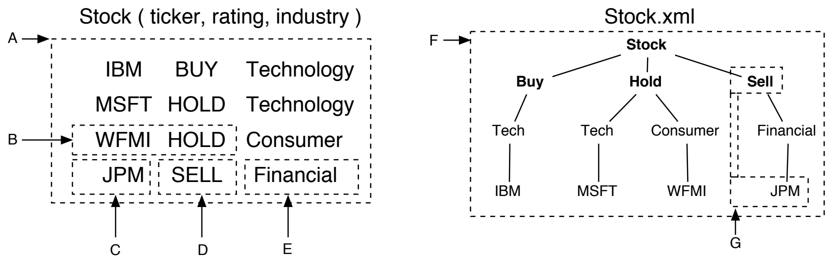


Figure 2: A relational table of stock recommendations (left), the same data represented as XML (right), and an illustration of fragments of the data to be signed.

properties that can be applied to any fragment of data. Realization of the signature primitive is discussed in the next section.

The author signs data to ensure that others cannot modify it. The granularity of signatures can vary: an author can sign an entire table, a tuple, a single column value.[3] Usually signatures are used to associate some data in an unmodifiable way, as shown in the next example. In what follows we use stock recommendations as a simple example of origin critical data, however the intended application domains remain those described in Section 1.

Example 2.1 Figure 2 shows stock recommendations represented as a relational table Stock(ticker, rating, industry) and as an XML document. The dotted regions illustrate portions of the data that are signed, called the target fragment of a signature. Signature $sig(A)$ is applied to target A, i.e. the entire table, and $sig(F)$ is similarly applied to the entire document. If the user wanted to compare the performance of the recommended portfolio represented by Stock, then these signatures provide integrity: poorly performing stocks cannot be removed and outperforming stocks cannot be added after signing. Signature $sig(B)$ and $sig(G)$ are applied to ticker-rating pairs. This associates the ticker name with the rating in an unmodifiable way, however a collection of such signed tuples does not prevent deletions, rearranging or additions to the collection. Signatures $sig(C)$, $sig(D)$, and $sig(E)$ are applied to individual attribute targets. By themselves, these three are probably not useful signatures since they do not authenticate the association between ticker and rating, which is of primary importance here.

The choice of signature granularity is application dependent. The signature of an entire database protects against all possible changes, but may be inefficient since verification must be performed on the entire database. In practice authors sometimes want to authorize smaller pieces of data. In many contexts, the author may wish to publish data signed in more than one way, with varying granularity. This allows a recipient to republish the data in various forms, retaining its evidence of authenticity. For example:

1. In Example 2.1, an author may wish to sign all subsets of tuples by sector, so that the data consumer can extract authenticated data for any relevant industry sectors, and omit others.
2. Consider a college transcript represented as a structured document, and signed by a academic administrator. The original form of the transcript may include fields the student wishes to hide when the transcript is submitted to a potential employer (e.g. date of birth). The administrator may wish to provide two signed versions of the document – one with the date of birth, and one without. (The administrator would not however, want to grant the student signed versions of the transcript that omit bad grades.)
3. If the order of elements is critical for an XML document, then a signature must secure the order. For other data, the author may wish to sign an unordered collection, allowing any ordering to be authenticated. In this case, the author would need to sign all possible orderings, or use a signature primitive that is order insensitive (we return to this in the next section).

Citations. We propose another integrity annotation that allows for the *citation* of signed data. We define a citation to be an annotation of a data fragment (the derived data), with a query and a reference to some signed data (the target). A citation represents a claim of authenticity: the derived data is the result of evaluating the query on the target fragment. The following examples provide some intuition, and an illustration of the flexibility of citations.

Example 2.2 Consider again the stock recommendations in Figure 2. Table 1 presents four examples of citations. For each, the first column is a derived fragment (tuples or sets of tuples in this case). The second column is the citation query which is expressed as a conjunctive query over the signed target fragment. Here A refers to the fragment signed by $sig(A)$. The last column indicates that the target is backed by a signature. We describe the meaning of each:

1. Citation (1) has derived data consisting of two tuples. Its citation claims that the fragment is the result of evaluating query C_1 on the target (the entire Stock table) which is signed by signature $sig(A)$.
2. Citation (2) is very similar with a different selection condition in the citation query.

[3]Although the main focus for data exchange is XML, we present some examples in relational form for simplicity of presentation.

3. Citation (3) consists of the same derived data as (2), however its citation query is different: it claims that the derived data is *contained* in the result of query C_3. Clearly this citation provides a different authenticity guarantee than citation (2).
4. The target data of a citation was signed in each of the examples above, but in other cases may instead be cited, resulting in a composition of citations. Assume that the derived fragment from Citation 1 is called T_1. Citation 4 therefore refers to a fragment that is itself cited. The claim of authenticity here is the composition of the individual claims. That is, the citation claims that tuple (MSFT, HOLD) is the result of query C_4 evaluated on table T_1 which itself is the result of citation query C_1 on the original data signed with $sig(A)$.

Since a citation is merely a claim, it must be verified by checking the signature of the cited source, and verifying that the cite fragment is in fact the result of the citation query evaluated on the citation source. (In the next section we mention techniques to make this verification procedure more efficient.) It is worth noting that a citation is a generalization of a signature. A citation whose query is the identity query is precisely a signature as described above.

Citations are useful because they do not require the compliance of the author, and provide additional flexibility if the signature on the source data does not permit the extraction a user desires. Citations are also useful for representing the relationship of an aggregation to its contributing values. However, citations may not offer the same level of integrity guarantee as a signature, and as the example shows, the same data may be cited using more than one citation query resulting in different authenticity conditions. Notice that Citations (2) and (3), as well as $Sig(B)$ from Example 2.1, are each annotations representing a claim of integrity about the target (WFMI, HOLD). Each of these integrity annotations has a different meaning. The distinction may be important, and careful reasoning about integrity semantics may be required.

3. CRYPTOGRAPHIC TECHNIQUES

Our objective is not merely to carry *claims* of authenticity along with data, but to propagate virtually unforgeable *evidence* of authenticity, verifiable by recipients. To do this we must employ cryptographic techniques. The most basic is the digital signature, which can be used to implement the basic signature annotation above. We then describe more advanced techniques which support extensibility. We show how Merkle hash trees can be used to design a signature primitive permitting controlled removal of elements from a signed collection. We then give an overview of more advanced techniques from the cryptographic literature that can be used to implement extensible signatures and citations.

Digital signatures

Digital signatures [11] are the basic tool for supporting origin authenticity. Digital signature schemes are generally based on public-key cryptography [11], and consist of two operations: signing and verification. Alice signs a piece of data by computing the one-way hash of the data and then encrypting the hash with her private key. The result is the *signature value*, and accompanies the data element when published. RSA [25] and SHA-1 [26] are examples of public key encryption and hash functions, respectively, from which a signature scheme can be built. Bob verifies a signature by retrieving Alice's public key, using it to decrypt the signature, and checking that the result is equal to the hash of the data element purportedly signed.

A verified digital signature provides extremely strong evidence of origin authenticity. The digital signature therefore provides the basic implementation for our signature annotation described above. Publishers generate public/private key pairs, add signature values to their data, and references to public key resources. Recipients verify signatures and propagate signatures to versions they in turn publish (as long as they publish the data in precisely the form it is signed).

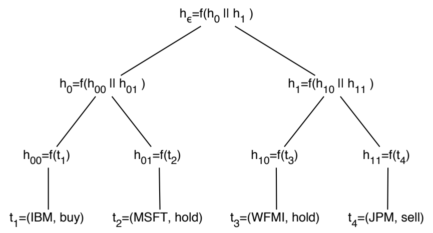


Figure 3: Merkle hash tree over stock tuples.

Extensible signature techniques

As mentioned above, it is often necessary to sign data in more than one way, or to sign data in such a way that certain modifications are permitted without invalidating the signature. The naive way to support this flexibility is for the author to publish many signatures along with the data. This can be very inefficient since, for example, providing signatures for every possible order in a collection would require an exponential number of signatures. We hope to avoid this by employing the techniques described next.

Merkle trees to allow authorized deletions. Suppose Alice wants to sign a collection of data items so that Bob can delete items but not add new items. If Carol receives a modified collection from Bob, she should be able to verify that each item was indeed authored by Alice, although some items may be missing. Alice's signature permits authorized deletions in this case, and this effect can be implemented using Merkle trees [17]. Note that signing each item in the original collection individually allows unauthorized mixing of items if Alice signs more than one collection over a period of time.

We illustrate how Alice would sign the collection of stock recommendations by building the hash tree illustrated in Fig. 3. Alice uses a collision-resistant hash function f to build a binary tree of hash values as follows. First, she computes the hash for each tuple t_i. Then she pairs these values, computing the hash of their

Table 1: Citations, referring to the relational data in Figure 2.

	Derived fragment	Citation query	Target fragment	Signature / Citation
(1)	(IBM, BUY) (MSFT, HOLD)	$C_1(t, r)$:- A(t, r, "Technology")	A	Sig A
(2)	(WFMI, HOLD)	$C_2(t, r)$:- A(t, r, "Consumer")	A	Sig A
(3)	(WFMI, HOLD)	$C_3(t, r) \subseteq$ A(t, "HOLD", i)	A	Sig A
(4)	(MSFT, HOLD)	$C_4(t, r)$:- T_1(t, "HOLD")	T_1	Cit T_1

concatenation (denoted || in the figure) and storing it as the parent. She continues bottom-up, pairing values and hashing their combination until a root hash value h_ϵ is formed. Note that h_ϵ is a hash value that depends on all tuples in her database. Alice publishes a description of f along with h_ϵ signed with her private key.

If Bob would like to delete t_3 from the collection, he will publish to Carol tuples t_1, t_2, t_4 along with h_{10} and the root hash signed by Alice. Carol will verify the authenticity of the data by recomputing the Merkle tree up to the root hash, and verifying Alice's signature on it. We assume a fixed order for the tuples and some specified structural information allowing Carol to deterministically reproduce the hash tree. Bob cannot add tuples not in the original collection without finding a collision in f, which is computationally infeasible. Note that, using this construction, Carol can tell how many items have been removed and from which positions in Alice's original data. This may be considered a feature and not a limitation for many applications.

The construction above is vulnerable to dictionary attacks by Carol, who can guess values for the omitted tuples and check them efficiently by hashing. In [14] a more secure signature scheme supporting controlled deletions is presented that avoids this vulnerability, along with other controlled operations including a version of deletion that does not reveal positions deleted.

The study of these extensible signature schemes is an active research area in cryptography, with a number of open problems mentioned in [24], and interesting extensibility features realized in [14, 19]. Our future goal is to adapt these techniques to our setting, as they provide an important efficiency improvement: when an extensible signature scheme exists for a useful operation, an author can effectively authorize many data elements by providing a single extensible signature. Naturally, extensible operations must be chosen carefully to avoid unintended forged signatures.

Query certification

The naive strategy for verification of a citation is to retrieve the original signed target data, compute the citation query and compare the result with the annotated data. This may be inefficient or impossible in a data exchange setting. Research into consistent query protocols [15, 22, 10, 9] can provide a more efficient verification process in some cases. These techniques allow a citation to carry proof that the derived data is the result of the citation query, relative to the summary signature on the original database. In particular, the techniques are again based on Merkle trees [17, 18] and allow signing of a database D such that given a query Q and an possible answer x to Q, a verification object can be constructed which proves that $x = Q(D)$. We omit further discussion of these techniques for lack of space; please see [20] for further details.

4. MANAGING ANNOTATED DATA

Managing data that contains annotations requires representing the annotated data, expressing queries over the data, propagating annotations through queries, and interpreting the data and annotations that result.

We are motivated by data exchange scenarios, and therefore focus on semistructured data enhanced with integrity annotations. The W3C Recommendation for XML digital signature syntax [12] can provide a basis for data representation and supports the signing of arbitrary fragments of an XML document. It is easy to support multiple signatures over the same document, overlapping signatures, and signatures by multiple authors. To this basic signature schema, we wish to add metadata for extensible signatures and citations.

Querying integrity-annotated data. A query over annotated data results in output data with integrity annotations. Queries must include selection conditions over the annotations (which for instance assert that certain data elements must be signed and verified) and propagation rules which determine how signatures should be propagated from the input to the output. For example, if data in the input has multiple signatures it may be sufficient to propagate just one, or it may be necessary to carry all signatures into the output. Further, because of the flexibility of citations, a wide variety of annotations could be propagated to the output, and the choice will depend on the setting. Recent work in data provenance provides some techniques for annotation propagation. In addition users should be able to query the integrity of data declaratively, without resorting to calls to low-level cryptographic routines.

A formal model of data authenticity. A number of challenges in this area call for a formal model to analyze claims of authenticity. First, it may not be clear in all applications how an author should sign data. For instance, in Example 2.1 we considered the case where signatures of stock recommendations were applied to tickers and ratings separately, and did not secure their association. In such a simple example this flaw was immediately evident. In a more complex setting the question of what to sign – and especially what extension semantics to permit for extensible signatures – could be a difficult issue for a data source. Second, decisions on propagation rules should be guided by the authenticity assertions that users want to verify. Finally, interpreting the meaning of multiple, possibly nested, signatures or complex citations may be very difficult. A formal model of authenticity will serve each of these issues. We have begun to address these issues by relating the authenticity guarantees of signatures to conventional database constraints [21].

5. RELATED WORK AND CONCLUSION

The authors of [4] use conventional digital signatures to implement "content extraction signatures" which allow an author to sign a document along with a definition of permissible operations of blinding (similar to redaction) and extraction. Recipients can extract data freely, but the verification procedure requires contacting the original author. The authors of [3] propose a framework of cooperative updates to a document by predetermined recipients constrained by integrity controls. In both cases, integrity properties are provided at the expense of flexible data exchange.

A number of projects [8, 6, 7] have studied data provenance, or lineage, which involves tracing and recording the origin of data and its movement among databases. These results provide important tools for managing integrity annotations, including complexity results for decision problems related to provenance and a background for propagation rules [6]. However, the emphasis of this work is not integrity, and we are concerned not just with carrying annotations, but providing cryptographic evidence of source attribution. Please see [20] for a full description of related research.

Conclusion. Today's web publishing applications require guarantees of integrity not provided by current technology. We have proposed primitives for expressing claims of origin, cryptographic techniques to implement these primitives, and have identified key problems in managing claims of integrity in the course of querying and restructuring of data. Solutions to these problems require formalizing the integrity guarantees of cryptographic primitives, and integrating these with query languages used for data management. We have tried to highlight the compelling challenge of preventing unauthorized modification of data while at the same time allowing innocuous modifications performed in the course of common collaboration and data integration.

6. REFERENCES

[1] R. Agrawal, P. J. Haas, and J. Kiernan. Watermarking relational data: framework, algorithms and analysis. *The VLDB Journal*, 12(2):157–169, 2003.

[2] Andreas D. Baxevanis. Molecular biology database collection. Nucleic Acids Research, available at www3.oup.co.uk/nar/database/, 2003.

[3] E. Bertino, G. Mella, G. Correndo, and E. Ferrari. An infrastructure for managing secure update operations on xml data. In *Symposium on Access control models and technologies*, pages 110–122. ACM Press, 2003.

[4] L. Bull, P. Stanski, and D. M. Squire. Content extraction signatures using xml digital signatures and custom transforms on-demand. In *Conference on World Wide Web*, pages 170–177. ACM Press, 2003.

[5] P. Buneman. Curated databases, November 2003. personal communication.

[6] P. Buneman, S. Khanna, and W. C. Tan. Why and where: A characterization of data provenance. In *ICDT*, pages 316–330, 2001.

[7] P. Buneman, S. Khanna, and W. C. Tan. On propagation of deletions and annotations through views. In *PODS '02*, pages 150–158, 2002.

[8] Y. Cui and J. Widom. Practical lineage tracing in data warehouses. In *ICDE*, pages 367–378, 2000.

[9] P. Devanbu, M. Gertz, A. Kwong, C. Martel, G. Nuckolls, and S. G. Stubblebine. Flexible authentication of xml documents. In *ACM Computer and Communications Security*, pages 136–145, 2001.

[10] P. T. Devanbu, M. Gertz, C. Martel, and S. G. Stubblebine. Authentic third-party data publication. In *IFIP Workshop on Database Security*, pages 101–112, 2000.

[11] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[12] D. Eastlake, J. Reagle, and D. Solo. Xml signature syntax and processing. http://www.w3.org/TR/xmldsig-core, February 12 2002. W3C Recommendation.

[13] H.r. 3261, to prohibit the misappropriation of certain databases. Introduced in the House, 108th Congress, available at http://frwebgate.access.gpo.gov, 2003.

[14] R. Johnson, D. Molnar, D. X. Song, and D. Wagner. Homomorphic signature schemes. In *RSA Conference on Topics in Cryptology*, pages 244–262, 2002.

[15] J. Killian. Efficiently committing to databases. Technical report, NEC Research Institute, February 1998.

[16] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1997.

[17] R. C. Merkle. Protocols for public key cryptosystems. In *IEEE Symposium on Security and Privacy*, pages 122–134, 1980.

[18] R. C. Merkle. A certified digital signature. In *CRYPTO*, pages 218–238, 1989.

[19] S. Micali and R. L. Rivest. Transitive signature schemes. In *RSA Conference on Topics in Cryptology*, pages 236–243. Springer-Verlag, 2002.

[20] G. Miklau. Research problems in secure data exchange. Univ. of Washington Tech Report 04-03-01, Mar 2003. Available at www.cs.washington.edu /homes/gerome.

[21] G. Miklau and D. Suciu. Modeling integrity in data exchange. In *Proceedings of VLDB 2004 Workshop on Secure Data Management*, August 2004.

[22] R. Ostrovsky, C. Rackoff, and A. Smith. Efficient consistency proofs on a committed database.

[23] Peter Buneman and Sanjeev Khanna and Wang-Chiew Tan. Data Provenance: Some Basic Issues. In *Foundations of Software Technology and Theoretical Computer Science*, 2000.

[24] R. Rivest. Two new signature schemes. Presented at Cambridge seminar, March 2001. See http://www.cl.cam.ac.uk/Research/Security/seminars/2000/rivest-tss.pdf.

[25] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[26] Secure hash standard. *Federal Information Processing Standards Publication (FIPS PUB)*, 180(1), April 1995.

[27] B. Schneier. *Applied Cryptography, Second Edition.* John Wiley and Sons, Inc., 1996.

Design and Implementation of a Geographic Search Engine

Alexander Markowetz [1] Yen-Yu Chen [2] Torsten Suel [2]
Xiaohui Long [2] Bernhard Seeger [3]

ABSTRACT

In this paper, we describe the design and initial implementation of a geographic search engine prototype for Germany, based on a large crawl of the `de` domain. Geographic search engines provide a flexible interface to the Web that allows users to constrain and order search results in an intuitive manner, by focusing a query on a particular geographic region. Geographic search technology has recently received significant commercial interest, but there has been only a limited amount of academic work. Our prototype performs massive extraction of geographic features from crawled data, which are then mapped to coordinates and aggregated across link and site structure. This assigns to each web page a set of relevant locations, called the geographic footprint of the page. The resulting footprint data is then integrated into a high-performance query processor on a cluster-based architecture. We discuss the various techniques, both new and existing, that are used for recognizing, matching, mapping, and aggregating geographic features, and describe how to integrate geographic query processing into a standard search architecture and interface.

1. INTRODUCTION

The World-Wide Web has reached a size where it is becoming increasingly challenging to satisfy certain information needs. While search engines are still able to index a reasonable subset of the (surface) web, the pages the user is really looking for may be buried under hundreds of thousands of less interesting results. Thus, search engine users are in danger of drowning in information. Adding additional terms to standard keyword searches often fails to drill the iceberg of results that are returned for common searches. A natural approach is to add advanced features to search engines that allow users to express constraints or preferences in an intuitive manner, resulting in the desired information to be returned among the first results. In fact, search engines have added a variety of such features, often under a special *advanced search* interface, though mostly limited to fairly simple conditions on domain, link structure, or last modification date.

In this paper we focus on how to constrain web queries geographically. Geography is a particularly useful criterion, since it most directly affects our everyday lives and thus provides an intuitive way to express an information request. In many cases, a user is interested in information with geographic constraints, such as local businesses, locally relevant news items, or tourism information about a particular region. When taking up yoga, local yoga schools are often of much higher interest than those of the world's ten biggest yoga schools.

We expect that *geographic search engines*, i.e., search engines that support geographic preferences, will have a major impact on search technology and associated business models. First, geographic search engines provide a very useful tool. They allow a user to express in a single query what might take multiple queries with conventional search engines. Thus, a user of a conventional search engine looking for a yoga school in or close to Brooklyn may end up trying queries such as `(yoga AND new york)` or `(yoga AND brooklyn)`, but even this might yield inferior results as there are many ways to refer to a particular area and since the engine has no notion of geographical closeness, e.g., a result across the bridge to Manhattan might also be acceptable. Second, geographic search is a fundamental technology for *location-based services* on mobile devices. Third, geographic search supports locally targeted web advertising, thus attracting advertisement budgets of small businesses. Other opportunities arise from mining geographic properties of the web, e.g., for market research.

Given these opportunities, it comes as no surprise that most leading search engines have made significant efforts to deploy some form of geographic web search. Our approach differs from these, both in the way geographic information is extracted from the web and how it is integrated into query processing. In particular, commercial engines focus on matching pages with data from business directories, supporting search for local businesses and organizations. While this is an important part of geographic search, we focus on more general information requests. A user may not just be interested in finding businesses listed in yellow pages, but may have broader interests that can best be satisfied by private or non-commercial web sites, such as local news and cultural events, or the history of a certain area. In order to facilitate such queries, we extract geographic markers, such as addresses or phone numbers, from web pages, independent of their listing in any directory. To extend search capabilities to those pages that contain no such markers, we employ a combination of new and previously proposed techniques based on link and site structure.

Before continuing, we briefly outline our basic approach. Our system is a crawl-based engine that starts by fetching a subset of the web for indexing and analysis, focusing on Germany and crawling the `de` domain. Afterwards, a standard text index is built. In addition, data extraction and mining is used to assign a set of relevant locations to each page, called a geographic footprint. Finally, search queries consisting of textual terms and geographic areas are evaluated against the index and footprint data using an appropriate ranking function. The goal of this project is to test and further develop ideas outlined in earlier work [21, 11, 19], by building a complete prototype.

Our contributions are: First, we provide the first in-depth description of an actual implementation of large-scale geographic web search. Our prototype, to be made available soon, is based on a crawl of over 30 million pages in the `de` domain, with plans to expand further. Second, we combine several known and new techniques for deriving geographic data from the web, using features such as town names, zip codes, phone numbers, link and site structure, and external sources such as `whois`. We represent the resulting geographic footprint of a page in a simple highly compressible format that is used during link and site analysis and query processing. Third, we provide the first discussion of efficient query execution in large geographic search engines. Due to space constraints, we have to omit many details. An expanded version of this paper is available as a technical report [20].

2. RELATED WORK

In this section, we describe related work on *geo coding*, existing geographic search engines, and the Semantic Web. Since we treat content, not hardware, we have omitted work on determining physical

[1]Department of Computer Science, University of Science and Technology, Kowloon, Hong Kong. `alexmar@cs.ust.hk`

[2]CIS Department, Polytechnic University, Brooklyn, NY. {`yenyu@photon.poly.edu,suel@poly.edu,xlong@cis.poly.edu`}. Research supported by NSF CAREER Award NSF CCR-0093400 and the New York State Center for Advanced Technology in Telecommunications (CATT) at Polytechnic University.

[3]Department of Mathematics and Computer Science, Philipps Universität, Marburg. `seeger@informatik.uni-marburg.de`

Eighth International Workshop on the Web and Databases (WebDB 2005), June 16–17, 2005, Baltimore, Maryland.

locations of servers. Most web pages today are hosted in server farms hundreds of miles away from either author or geographic regions they relate to.

Geo Coding: A good discussion of geographic hints commonly found in web pages is provided by McCurley [21], who introduces the notion of *geo coding*. He describes various geographic indicators found in pages, such as zip codes or town names, but does not discuss in detail how to extract these, or how to resolve the geo/geo or nongeo/geo ambiguity.

Work in [4, 11] introduced the idea of a page's *geographic scope*, i.e., the area it addresses in terms of readership. Initially, they assign a position to every web site based on its `whois` entry. Then a fixed hierarchy of administrative regions is superimposed, and some link analysis is performed. If enough links from a region point to a web site and these links are evenly distributed, the region is included in the site's geographic scope. The approach was applied to the United States using states, counties, and cities for the geographic hierarchy. Our approach in Section 4.6 is basically a generalization and refinement of the work in [4, 11], but differs in several ways. In general, [4, 11] is fairly coarse-grained as it focuses on sites instead of single pages and on relatively large geographic regions. It relies on the existence of a sufficient number of incoming links, and thus does not work well for pages and sites with moderate in-degree. The evaluation in [11] is limited to sites in the `edu` domain, where `whois` provides a good estimate of a site's location, and does not address more noisy data from page content.

The approach closest to ours is [1], using a hierarchical gazetteer with 40,000 names of cities all over the globe. It performs geo coding by looking for names of cities with more than 500,000 people, though decreasing the minimum size to 5,000 is reported to have a positive effect. The gazetteer's hierarchy is used for disambiguation when there are several towns of the same name, but the size of towns is not considered in this case. Similar to our geographic footprint, [1] focuses on a document's *geographic focus* rather than the more specialized *geographic scope* of [11]. In contrast to our system, the geographic focus of a page is not represented in geographic coordinates, but tied to a node in the hierarchical gazetteer. There can be several foci for a document, although the authors explicitly seek to avoid this by grouping.

Geographic Search Engines: Several geographic search engines are already available online. Some are academic prototypes based either on small specialized collections or a meta search approach. In particular, [16] performs automatic geo coding of documents based on the approach in [11]. Most other prototypes, such as [8], require pages either to carry special markup tags or to be manually registered with the search engine.

There are several lesser-known geographic engines by commercial players. Some, such as the extension to *Northern Light* by *Divine* [12], have already disappeared again. Others such as [14] rely on geographic meta data in the pages, or query internet directories such as the Open Directory Project. Of all these, the Swiss *search.ch* engine [22], which has been around for several years, is closest to our approach. It allows users to narrow down their initial search by specifying a more and more focused location, over several hierarchical levels such as cantons (states) and cities within Switzerland.

As mentioned, geographic search has recently received a lot of attention from the major search companies. Both *Google* and *Yahoo* have introduced localized search options [15, 24]. Their approaches appear to be quite different from ours, and seem to rely on an intermediate business directory. Users first retrieve entries for businesses that satisfy certain keywords and are close by, and can then extend the search to actually retrieve pages about these businesses and the area they are located in. The exact algorithms are not publicized. There are two main differences to our approach, the intermittent business directories and the location modeling to point coordinates, i.e., the street address of the businesses, to be displayed on detailed street maps. There seems to be no mechanism to model geographic footprints of pages that cover larger areas, such as a county or state.

Geographic Semantic Web: It seems natural to extend the Semantic Web to a *Geographic Semantic Web*, such as proposed in [13], where each web page contains some meta data, defining its geographic footprint. Several models are already available [3, 9]. Other models from the GIS community, such as GML from the Open GIS Consortium [7], can be adapted. However, there are two major problems, inherent to the Semantic Web, that make this approach infeasible for general web search (though it may be useful in other scenarios). First, there is a *chicken and egg* problem. Authors will only include meta information if search engines use them, while engines will wait for a sufficient amount of meta information to become available before building any services on it. Second, Web authors are not to be trusted, as they frequently provide misleading information to manipulate search engines. For this reason current engines pay little attention to existing meta tags.

3. UNDERLYING DATA

We now briefly describe the data, as used in our prototype. Using the *PolyBot* web crawler [23], we acquired about 31 million distinct web pages from the `de` domain in April 2004. We chose the `de` domain for two reasons. First, it is the right size both geographically and in terms of number of pages. It is quite dense with about 7.8 million registered domains within a relatively small area. It is also reasonably self-contained in terms of link structure. Thus, the domain provides a nice test bed, meaningful but not outside the reach of academic research. The `de` domain was estimated in 2000 at 3.67% of the entire web [2]. This translates to about 150 million pages to achieve the same coverage as 4 billion pages (the size of *Google* as of November 2004) on the entire web; this is within reach of our current setup. Second, availability of geographic data is a big issue. The `whois` entries for `de` domains are complete and well structured, allowing us to extract information without effort. We retrieved 680,000 `whois` entries for all the domains our crawl had touched; many of the 7.8 million registered domains do not actually have a live web server. We also had access to several other sources of geographic data for Germany, and an understanding of the language, administrative geography, and conventions for referring to geographic entities.

We focused on two geographic data sets for Germany. The first maps each of 5,000 telephone area codes to *one* city and also to the coordinates of the centroid of the region that the code covers. The second maps zip codes to 82,000 towns, and these towns to their positions. If the town was a village, it was also mapped to the associated city. This data set originated from a GIS application, where geographic positions are the database keys and town names are only for display to the user. Names were often misspelled or abbreviated in various nonstandard ways, requiring painstaking manual cleaning.

4. GEO CODING

The process of assigning geographic locations to web pages that provide information relevant to these locations is called *geo coding*. A document can be associated with one or multiple locations, for example when a company web page refers to several different outlets. We call this collection of locations the page's *geographic footprint*. For every location in the footprint, an integer value is assigned that expresses the *certainty* with which we believe the web page actually provides information relevant to the location.

In our approach, we divide geo coding into three steps, *geo extraction*, *geo matching*, and *geo propagation*. The first step extracts all elements from a page that might indicate a geographic location, including elements in URLs. The second step tries to make sense of these by mapping them to actual locations, i.e., coordinates, and leads to an initial geo coding of the pages. In these first two steps, we make use of databases of known geographic entities such as cities or zip codes. In the third step, we perform *geo propagation* to increase the quality and coverage of the geo coding through analysis of link structure and site topology. Before we proceed with the description of our geo coding process, we introduce our representation of a document's geographic footprints.

4.1 Geographic Footprints of Web Pages

In every GIS, a basic design decision has to be made between a vector data model and a raster data model, mapping data onto a discrete grid. A web page may contain several geographic hints, some referring to point positions, others (cities or zip codes) refer to polygonal areas. Thus, our data model has to handle both types. We decided to use a raster data model, representing geographic footprints in a bitmap-like data structure. In comparison to a vector model, we lose some precision by pressing the information into the grid. With a sufficiently fine grid however, the degree of imprecision is small, especially when compared to other uncertainties in the data and extraction process. In our case, we superimposed a grid of 1024×1024 tiles, each covering roughly a square kilometer, over Germany, and stored an integer amplitude with each tile, expressing the certainty that the document is relevant to the tile.

This representation has two advantages. First, it allows us to efficiently implement some basic aggregation operations on footprints. If a page contains several geographic features, the footprint for the page is defined as the sum of the footprints of the individual features, after suitable normalization. These operations are very useful during geo propagation and query processing. Second, since for most documents only a few tiles are non-zero, we can efficiently store the footprints in a highly compressed quad-tree structure. Moreover, we can use lossy compression (smoothing) on such structures to further reduce their size and thus facilitate efficient query processing.

We implemented a small and highly optimized library for operations such as footprint creation, aggregation, simplification (smoothing), and intersection (for query processing) based on quad-trees. Our focus here, as discussed earlier, is not on simple yellow page operations but more general classes of geographic search operations. Our grid model is particularly useful for the geo propagation and query processing phases, where exact locations are not that crucial.

4.2 External Databases

In addition to geographic markers extracted from pages, various external sources can also be used for geo coding, in particular business, web, and `whois` directories.

Business directories (yellow pages) map businesses and associated web sites to addresses, and thus to geographic positions. Some geographic search engines such as those of *Google* and *Yahoo* [15, 24] appear to make heavy use of business directories. The main problem with business directories is also their biggest strength. They require registration fees, and thus usually list mainly commercial companies, ignoring many personal or non-profit web sites. The fees however also often result in higher data quality.

Web directories such as Yahoo and ODP maintain geographic directories that categorize sites by region. They are difficult to maintain, far from complete, and often outdated. However, they can be useful as an additional data source in geo coding.

As an integral part of the Internet infrastructure and freely accessible, the `whois` directory is also a good source of geographic information. For every domain, it contains the address of the individual that registered it. An earlier study [25] showed a high degree of accuracy for `whois` entries. However, the quality of `whois` entries differs between top-level domains. For the `de` domain, they are highly structured and usually complete, with precise addresses and phone numbers. In contrast, entries for the `uk` domain typically contain less information and are fairly unstructured.

In Section 4.6.1 we discuss how to plug information from such databases into our geo coding process.

4.3 Germany's Administrative Geography

Effective geo coding requires an understanding of a country's *administrative geography* and common usage of geographic terms. Thus, one has to know how geographic names are composed, what the role of states and counties is, and how postal or area codes are used. Since every country is organized differently, the rules presented for Germany in this section will have to be adapted for other countries and languages. In the United States, for example, most addresses contain the state, which can be used to resolve ambiguities between towns with the same name. In German addresses, states are never mentioned. German telephone area codes and zip codes are highly clustered, i.e., codes with a common prefix tend to be in the same region. Large companies might have their own zip code, but we could infer their position from the positions of similar zip codes.

We give a brief summary of the usage of geographic terms in Germany. States, like counties and districts play little role in daily life and are rarely mentioned, and thus ignored. Area and zip codes are distributed in clusters; at least all entries with the same first digit are clustered. There is no simple relation between these numeric codes and towns. A town might cover several of these codes or several towns might share the same numeric code.

Towns fall into two categories, cities and villages (also boroughs), with a one-to-many relationship between the two. Every village is associated with exactly one city, but a city might be associated with several villages. Villages are often mentioned in conjunction with their cities. German town names consist of up to three parts. First, there is an optional single-term *descriptive prefix*, such as *Bad*. Second, there is a mandatory *main name*, such as *Frankfurt*, and third, extra *descriptive terms*, such as *bei Köln*, *am Main*, *Sachsen*.[1] Descriptive prefixes and large parts of the descriptive terms are often dropped or abbreviated in various ways. The city of *Frankfurt am Main* might be written as *Frankfurt M.*, *Frankfurt/Main*, *Frankfurt a.M.*, or just *Frankfurt*.

4.4 Geo Extraction

This step reduces a document to the subset of its terms that have geographic meaning. If there is any uncertainty whether a term is used in a geographic meaning or not (called *geo-nongeo* ambiguity [1]), then this is resolved at this point. We extract only those geographic markers that we know how to map to geographic positions: town names, phone numbers, and zip codes. In addition to page content, we also analyzed URLs. URLs are a very useful source of geographic information, but tricky to analyze since terms are often not well separated (e.g., finding a city name in `cheapnewyorkhotels.com` is not straightforward). We refer to [20] for details.

4.4.1 Town Names

When extracting terms that might refer to towns, we could simply write out all terms that appear as part of some town name. However, this would produce a lot of garbage; many terms from town names are also common German or English words or surnames. To avoid this, we manually divided the set of all terms that appear town names into 3,000 *weak terms* that are common language terms, and 55,000 *strong terms* that are almost uniquely used as town names. When parsing web pages, we first try to extract all strong terms. Next, we look for any weak terms that appear together with the extracted strong terms in the same town name. The underlying idea is that we try to find a town's main name first and then parse for weak terms (often found in the descriptive suffixes and prefixes) to resolve any ambiguity.

We assigned a distance to each weak term. A weak term would only be recognized if it appears within that distance from an associated strong term. Thus, if we find the strong term *Frankfurt*, we might accept the weak term *Main* anywhere on the page ($distance = \infty$), or the weak term *Oder* within $distance = 2$ since it is a much more common term.[2]

[1] near the city of Cologne, on the river Main, in the state of Saxony

[2] *Main* and *Oder* are names of rivers; however, *Oder* is also the Ger-

To further increase the precision of the extraction, we assigned *killer terms* and *validator terms* to the strong terms. Any appearance of a strong term will be ignored if one of its killer terms also appears within some distance. Also, if a strong term has a validator term assigned to it, then any appearance of the strong term will be ignored unless the validator term appears within some distance. This allows us to handle town names that are also normal German words. We also introduced a list of *general killers* such that any strong term within some distance of a general killer will be ignored. This list was filled with 3,500 common first names and titles such as *Mr.*, *Mrs.*, or *Dr.* to avoid mistaking surnames for town names. More details are given in [20], where we discuss phone numbers and zip codes.

4.5 Geo Matching

The previous step reduced documents to sets of terms that carry a geographic meaning. This step maps these terms to actual towns, and thus to geographic locations. The problem, is that some terms can point to several town names, called *geo-geo ambiguity* [1]. Some towns share the same main name, and a town's main name might even appear in another town's descriptive terms. We make two assumptions about the usage of town names that allow us to define rules to resolve these ambiguous cases.

The first assumption is that the author of a document mentioning a town name intends to talk about *a single town of this name*, not about several towns of that name. That is, someone mentioning *Frankfurt* intends to talk about either one of the two towns in Germany of that name. This assumption is called *single source of discourse* [1]. Even if this assumption fails, it only introduces a negligible error to a geographic search engine. Thus, in the rare case where a document discusses why "neither town named Frankfurt has a strong soccer team", it might be acceptable to only assign this page to one of the two towns.

The second assumption is that the author most likely meant the largest town with that name. There are for examples two towns with the name *Göttingen*, a larger city and a tiny village, situated about 150 km apart. One expects that there are more pages about the larger of the two towns. The page will therefore be assigned to the city, not the village, unless there are other strong indications. As before, it can be argued that the failure of this hypothesis only introduces a marginal error, especially when the difference in size is huge.

Our strategy consists of the following steps. First, a metric is used to evaluate matches between town names and terms. Second, we write out the town with the best match, and then delete its terms from the term set. Finally, we start over to find additional matches on the reduced term set. There are several measures for the quality of a match between a town name and a set of terms. The actual implementation of the algorithms is omitted, since it is tailored to Germany's administrative geography and to the databases available to us. The general strategy however is broad enough to be adapted to various countries and data sets.

4.5.1 *Measuring Geo Matching*

The degree to which a town name can be matched with a set of retrieved terms can be measured in various ways. None of them performs well on its own, but in combination they prove adequate for deciding the best of several possible matches.

One simple measure is the *number of matched terms*, i.e., the number of terms in the town name that are contained in the set of terms from the web page. A similar measure is the *fraction of matched terms*, i.e., the fraction of terms in the town name that were found in the page. For any of the above, one can find examples where they work really well and ones where they fail. Some other types of techniques are stronger. If a *numeric marker* such as a zip code is found, then this will usually resolve any ambiguity. Another approach is based on looking for *nearby towns*. If we find both `Frankfurt` and `Offenbach`, we can be pretty certain that the page intends to talk about *Frankfurt am Main*.[3] In our application we employed a simplified version, using Germany's administrative hierarchy as an indicator of distance. This measure can be looked up from a table quickly, without ever having to compute an actual distance.

man word for *or*.

4.5.2 *The Matching Strategy*

Since the implementation of our matching algorithm, called *BB-First*, is very specific to Germany, we will not show it in full detail but rather sketch the underlying strategy. The algorithm is called *BB-first* because it extracts the *best* of the *big* towns *first*. It starts with the set of all strong terms found in the document, called `found-strong`, and the set of all German towns, and proceeds as sketched in Table 1.

```
1. Group towns into several categories
   according to their size.
2. Start with the category of the largest
   towns.
3. Determine the subset of all towns from this
   category that contain at least one term in
   found-strong.
4. Rank them according to a mix of the measures
   described in Section 4.5.1.
5. Add the best matched town to the result.
6. Remove all terms found in this town name
   from the set found-strong.
7. Start this algorithm over at Step 3, as long
   as there are new results.
8. If there are no new results, repeat the
   algorithm for the next category down.
```

Table 1: Basic steps of the BB-First algorithm

In our implementation, we measured the size of towns only by sorting them into *villages* and *cities*, thus running the algorithm only with these two categories. The algorithm can be directly traced to the underlying assumptions. It clearly prefers large towns over small ones. It also assumes a single sense of discourse, since every strong term can cause at most one town to be matched before it is removed from `found-strong`. The extracted towns receive a *certainty value*, estimated with the same measures we used to determine how well towns were matched with the set of terms.

The results of this algorithm, i.e., the matched towns, are then finally mapped into our quad-tree based footprint structure with integer amplitudes. Note that cities are not mapped to a single tile but to a larger area of a few kilometers squared. Each tile in the grid receives as amplitude the sum over the certainties of towns that map to this tile. Applying this procedure to every document results in an initial geo coding of our web crawl that can be processed further during the next step. In this initial coding, each page that contains a geographic marker has an associated non-empty geographic footprint. In our set of 31 million pages, about 17 million had non-empty footprints based on page content, represented in an average of 137 bytes after compression. About 5.7 million pages had (separate) non-empty footprints based on extraction of markers from their URLs, represented in an average of 38 bytes since there are fewer extracted markers.

4.6 Geo Propagation

After applying the above techniques, and excluding `whois` entries, slightly more than half of all web documents have a non-empty geographic footprint associated with them. This is not unexpected, since not every document contains a geographic reference in its text. On the other hand, many of the pages that did have a footprint were not particularly valuable in terms of their actual content. For example, it seems that many sites return geographic information such as contact

[3]The city of Offenbach is a direct neighbor of Frankfurt am Main, and about 700km from the other Frankfurt.

addresses in separate pages from the actual content that a user might be looking for. These issues can be overcome by *geo propagation*, a technique that extends the basic radius-one, radius-two (co-citation), and intra-site hypotheses from Web information retrieval to the geographic realm.

According to the radius-one hypothesis, two web pages that are linked are more likely to be similar than a random pair of pages [10]. This assumption can be extended to geographic footprints. If one page has a geographic footprint, then a page it is linked to is more likely to be relevant to the same or a similar region than a random page. The radius-two hypothesis about pages that are co-cited can be extended similarly. The intra-site hypothesis states that two pages in the same site are also more likely to be similar. For documents from the same sub-domain, host, or directory within a site, even stronger statements can be made. This can also be extended to geographic properties. For Germany, it is particularly useful since there exists a law that any `de` site must have a page with the full contact address of the owner no more than two clicks from the start page. Thus, at least one page in any given site by law should provide rich geographic information which is supposed to apply to the entire site.

Geo propagation uses the above geographic hypotheses to propagate geographic footprints from one page to another. The idea is that if two pages are related in any of the above manners, they should inherit some dampened version of each others geographic footprint. We modeled the "inheritance" by simply adding the entire footprint of one page to the other, tile by tile, with some dampening factor α, $0 < \alpha < 1$. The exact value of α depends on the relationship between two pages. If two pages are in the same directory for example, α will be larger than if they are only within the same site.

Note that this process does not converge, and has to be handled with care. If geo propagation is performed too often, every single document could end up with a footprint spanning the entire country. In practice, one or two steps seem to give most of the benefit, and proper dampening factors plus lossy compression (simplification) prevents footprint sizes from getting out of hand. This results in an increased number of pages with non-zero footprints and an increased number of non-zero tiles therein.

4.6.1 Geo Propagation in our Prototype

Based on these general ideas, we implemented several forms of geo propagation. Starting out with about 17 million footprints, we separately performed forward and backward propagation across links as well as between co-cited pages. Thus, if page A has a footprint m_A and links to a page B with a footprint m_B, then we transmit m_A to B and compute a new footprint of the form $m_B + \alpha m_A$ for B. This is implemented using two ingredients: (1) our optimized implementation of footprint operations based on quad-tree structures described in Subsection 4.1, and (2) an I/O-efficient implementation for footprint propagation along links that resembles a single round of the Pagerank implementation in [6]. Footprints are sorted on disk by destination page and then aggregated into the footprint of the destination page. Propagation was also performed within sites. Finally, resulting footprints need to be normalized. In the end, we obtained about 28.4 million pages with non-empty footprints, for a page coverage of more than 90%. We also separately stored 490,000 footprints that apply to entire sites. This amounts to about 60% of all sites, which is smaller than expected, due to the large number of parked single-page sites. The site's footprints can be added into pages' footprints or used separately during query processing.

5. GEOGRAPHIC SEARCH

Geographic search engines allow users to focus a search on a specific geographic area by adding a query footprint to the set of keywords. There are a number of possible interface for specifying the query footprint and displaying search results, and we discuss here only some basic approaches. In particular, the area of interest could be automatically extracted from a keyword query, by looking for terms that match a city or other geographic term and replacing it by a suitable query footprint. The automatic identification of queries that are geographical in nature is discussed in [17]. Or alternatively, users could use an interactive map for this purpose. In a mobile environment, the current location of the user could be determined from the networking infrastructure and translated into a footprint. Results could be shown as lists or displayed on an interactive map, and additional geographic browsing operations may be supported. Note that a query footprint should not be seen as a simple filter for keyword-based results, but as a part of the ranking function. We will now describe the actual query processing in two passes, first on an abstract level and later in terms of our actual current implementation.

5.1 Basic Geographic Query Execution

We now outline the differences between geographic and conventional web search engines on an abstract level. In a nutshell, a conventional search engine works as follows: (i) The user inputs a set of search terms. (ii) The engine determines a set of pages that contain all the search terms, by using the inverted index. (iii) It then uses the frequencies, contexts, and positions of the term occurrences in these pages, together with other measures such as link structure, to rank the results. This is typically done concurrently with the second step. At first glance, the query processing in our geographic search engine works in a very similar way: (i) The user inputs search terms *and* a query region that is converted into a *query footprint*. (ii) The engine then uses the keywords *and* the query footprint to determine the set of pages that are in the intersection of the inverted lists and whose footprint has a nonempty intersection with the query footprint. (iii) The engine then uses the keywords *and* query footprint, plus other measures such as link structure, to rank the results.
Thus, the engine uses both keywords and query footprint, to retrieve candidates results during the second and third steps. In our case, the first step is simply a question of interface design. The second step is also fairly similar to conventional engines, at least on a high level, except that now only those pages survive that contain all search terms and have a non-empty intersection. The final ranking is a little different, since it has to merge two unrelated ranking measures, importance and geographic proximity.

5.2 Geographic Ranking

We now describe in detail how we rank pages based on both terms and geographic footprints. The user of a geographic search engine wants top results to satisfy two criteria: they need to be relevant as well as close to the query footprint. One approach would be to simply use the query footprint as a filter, removing all results "outside" the query area, and then use the standard ranking. At the other end of the spectrum, we could use the search terms as a filter, and rank all documents in the intersection of the inverted list by their distance to the center of the query area.

We decided on a general framework that includes these two cases as well as the continuum in between, allowing users to select their own preferences. First, they can choose different shapes for the query footprint as shown at the top of Figure 1. If a user prefers a sharp cutoff at a distance of say 10 km, she selects the footprint on the left, while the query footprint on the right models a more gradual approach. During the ranking phase, we compute a *geographic score* for each page in the intersection of the inverted lists of the query terms, based, e.g., on the volume of the intersection or the vector product between query and document footprint; see the bottom of Figure 1. If the score is zero, the document is discarded. Second, she can choose the relative weight of term-based and geographic components in the ranking. Thus, the total score of a document under the ranking function will be a weighted sum of its term-based score, its geographic score, and maybe an additional measure such as Pagerank. Both query footprint shape and relative weighting of the scores can be provided by the user through simple sliders, allowing interactive reordering of results.

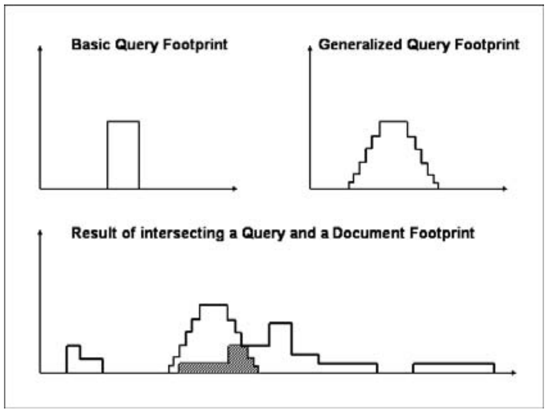


Figure 1: An illustration of footprints in a single spatial dimension. At the top, we have a query footprint with a distance threshold (left), and a footprint for a query that gives a lower score for documents that are farther away (right). At the bottom, we show an intersection between a query footprint and a document footprint.

5.3 Efficient Geographic Query Processing

Given this ranking approach, we now describe query processing in more detail. Figure 2 shows the example of a query with three search terms. After the query is issued, the inverted lists for the three terms are loaded into memory (shown here only as document IDs), and their intersection is computed. For any document in the intersection, there are two lookups. First, we maintain an in-memory table of conservatively approximated document footprints, obtained by lossy-compressing the footprint structures down to a size of at most 100 to 200 bytes each. We lookup in this table to check if the intersection between the query footprint and the document footprint is nonempty; if so, we compute an approximation of the geographic score of the document. Next, we perform a lookup into an in-memory table of Pagerank values to compute a final approximate document score.

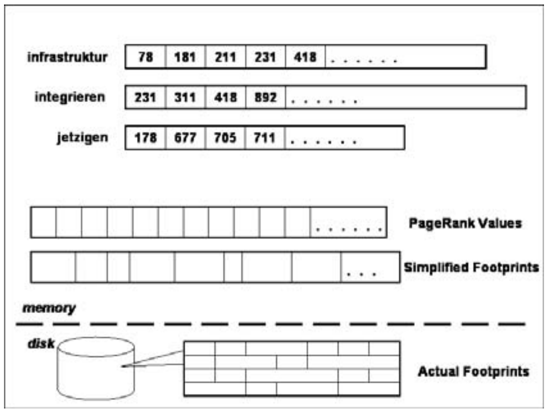


Figure 2: Organization of index structures, lookup tables, and geographic footprints in a scalable geographic engine.

After traversing the inverted lists and determining, say, the top-50 results, we can perform a more precise computation of their geographic scores by fetching footprints from disk. There are a number of other performance optimizations in search engines, such as index compression, caching, and pruning techniques [18], that are omitted here. By integrating these, we achieve query throughput comparable to that of a conventional non-geographic engine.

When compressed to 100 or 200 bytes, several million page footprints can be kept in memory by each node of the search engine cluster, a realistic number for large engines. In our prototype, we use a cluster of 7 Intel-based machines with reasonably large disks and main memories for our 31 million pages to sustain rates of a few queries per second.

6. CONCLUSION

This paper outlined design and implementation of a crawl-based geographic search engine for the German web domain. We described in detail how to extract geographic footprints from crawled pages through a *geo coding* process consisting of *geo extraction*, *geo matching*, and *geo propagation*, and discussed ranking and query processing in geographic search engines. Our prototype should be available online soon. One open issue for the near future is an appropriate evaluation of the quality of our footprints and query results.

Beyond this, there are many exciting open problems for future research in this area. On the most general level, many aspects of Web search and information retrieval, such as ranking functions, categorization, link analysis, crawling strategies, query processing, and interfaces, need to be reevaluated and adapted for the purpose of geographic search. We are particularly interested in the following directions. First, we are working on automatically identifying and exploiting terms such as "Oktoberfest" that are not listed in geographic databases but clearly indicate a particular location, through the use of data mining techniques. Second, we are looking at optimized query processing algorithms for geographic search engines. Third, we are studying focused crawling strategies [5] that can efficiently fetch pages relevant to geographic areas that run across many top-level domains. Finally, we are interested in *geographic mining* of the web.

Acknowledgement: We thank Thomas Brinkhoff for cooperation on earlier work [19] leading up to this project, and for continued feedback and support.

7. REFERENCES

[1] E. Amitay, N. Har'El, R. Sivan, and A. Soffer. Web-a-where: geotagging web content. In *Proceedings of the 27th SIGIR*, pages 273–280, 2004.
[2] Z. Bar-Yossef, A. Berg, S. Chien, J. Fakcharoenphol, and D. Weitz. Approximating aggregate queries about web pages via random walks. In *Proc. of 26th Int. Conf. on Very Large Data Bases*, September 2000.
[3] DCMI Usage Board. Dublin Core Qualifiers. Recommendation of the DCMI, Dublin Core Metadata Initiative, Jul 2000.
[4] O. Buyukkokten, J. Cho, H. Garcia-Molina, L. Gravano, and N. Shivakumar. Exploiting Geographical Location Information of Web Pages. In *WebDB*, 1999.
[5] S. Chakrabarti, M. van den Berg, and B. Dom. Focused crawling: A new approach to topic-specific web resource discovery. In *Proc. of the 8th Int. World Wide Web Conference*, May 1999.
[6] Y. Chen, Q. Gan, and T. Suel. I/O-efficient techniques for computing pagerank. In *Proc. of the 11th Int. Conf. on Information and Knowledge Management*, 2002.
[7] Open GIS Consortium. `http://www.opengis.org`.
[8] A. Daviel. April 1999. `http://geotags.com`.
[9] A. Daviel. Geographic registration of HTML documents. IETF Draft, July 2003. `geotags.com/geo/draft-daviel-html-geo-tag-06.html`
[10] B. Davison. Topical locality in the web. In *Proc. of the 23rd Annual Int. Conf. on Research and Development in Information Retrieval*, July 2000.
[11] J. Ding, L. Gravano, and N. Shivakumar. Computing geographical scopes of web resources. In *Proc. of the 26th VLDB*, pages 545–556, September 2000.
[12] Divine inc. Northern light geosearch. Last accessed February 2003.
[13] M. Egenhofer. Toward the semantic geospatial web. In *Proc. of the 10th ACM GIS*, pages 1–4, Nov 2002.
[14] Eventax GmbH. `http://www.umkreisfinder.de`.
[15] Google Inc. Google Local Search, 2003.
[16] L. Gravano. Geosearch: A geographically-aware search engine. 2003. `http://geosearch.cs.columbia.edu`.
[17] L. Gravano, V. Hatzivassiloglou, and R. Lichtenstein. Categorizing web queries according to geographical locality. In *Proc. of the 12th CIKM*, 2003.
[18] X. Long and T. Suel. Optimized query execution in large search engines with global page ordering. In *Proc. of the Int. Conf. on Very Large Data Bases*, 2003.
[19] A. Markowetz, T. Brinkhoff, and B. Seeger. Exploiting the internet as a geospatial database. In *Workshop on the Next Generation Geospatial Information*, Oct 2003. Also presented at the *3rd Int. Workshop on Web Dynamics*, 2004.
[20] A. Markowetz, Y. Chen, T. Suel, X. Long, and B. Seeger. Design and Implementation of a Geographic Web Search Engine. Technical Report TR-CIS-2005-03, CIS Department, Polytechnic University, February 2005.
[21] K. McCurley. Geospatial mapping and navigation of the web. In *Proc. of the 10th World Wide Web Conference*, pages 221–229, May 2001.
[22] Räber Information Management GmbH. `http://www.search.ch`.
[23] V. Shkapenyuk and T. Suel. Design and implementation of a high-performance distributed web crawler. In *Proc. of the Int. Conf. on Data Engineering*, 2002.
[24] Yahoo! inc. Yahoo Local Search, 2004. `http://local.yahoo.com`.
[25] M. Zook. Old Hierarchies or New Networks of Centrality? The Global Geography of the Internet Content Market. *American Behavioral Scientist*, 44(10), June 2001.

Using Bloom Filters to Refine Web Search Results*

Navendu Jain†
Department of Computer Sciences
University of Texas at Austin
Austin, TX, 78712
nav@cs.utexas.edu

Mike Dahlin
Department of Computer Sciences
University of Texas at Austin
Austin, TX, 78712
dahlin@cs.utexas.edu

Renu Tewari
IBM Almaden Research Center
650 Harry Road
San Jose, CA, 95111
tewarir@us.ibm.com

ABSTRACT

Search engines have primarily focused on presenting the most relevant pages to the user quickly. A less well explored aspect of improving the search experience is to remove or group all near-duplicate documents in the results presented to the user. In this paper, we apply a Bloom filter based similarity detection technique to address this issue by refining the search results presented to the user. First, we present and analyze our technique for finding similar documents using content-defined chunking and Bloom filters, and demonstrate its effectiveness in compactly representing and quickly matching pages for similarity testing. Later, we demonstrate how a number of results of popular and random search queries retrieved from different search engines, Google, Yahoo, MSN, are similar and can be eliminated or re-organized.

1. INTRODUCTION

Enterprise and web search has become a ubiquitous part of the web experience. Numerous studies have shown that the ad-hoc distribution of information on the web has resulted in a high degree of content aliasing (i.e., the same data contained in pages from different URLs) [14] and which adversely affects the performance of search engines [6]. The initial study by Broder et al., in 1997 [7], and the later one by Fetterly et al. [11], shows that around 29.2% of data is common across pages in a sample of 150 million pages. This common data when presented to the user on a search query degrades user-experience by repeating the same information on every click.

Similar data can be grouped or eliminated to improve the search experience. Similarity based grouping is also useful for organizing the results presented by meta-crawlers (e.g., vivisimo, metacrawler, dogpile, copernic). The findings by `searchenginejournal.com` [2] show a significant overlap of search results returned by Google and Yahoo search engines—the top 20 keyword searches from Google had about 40% identical or similar pages to the Yahoo results. Sometimes search results may appear different purely due to the restructuring and reformatting of data. For example, one site may format a document into multiple web pages, with the top level page only containing a fraction of the document along with a "next" link to follow to the remaining part, while another site may have the entire document in the same web page. An effective similarity detection technique should find these "contained" documents and label them as similar.

Although improving search results by identifying near-duplicates had been proposed for Altavista [6], we found that popular search engines, Google, Yahoo, MSN, even today have a significant fraction of near-duplicates in their top results[1]. For example, consider the results of the query "emacs manual" using the Google search engine. We focus on the top 20 results (i.e., first 2 pages) as they represent the results most likely to be viewed by the user. Four of the results, `www.delorie.com/gnu/docs/emacs/emacs_toc.html`, `www.cs.utah.edu/dept/old/texinfo/emacs19/emacs_toc.html`, `www.dc.urkuamk.fi/docs/gnu/emacs/emacs_toc.html`, and `www.linuxselfhelp.com/gnu/emacs/html_chapter/emacs_toc.html`, on the first page (top-10 results), were highly similar—in fact, they had nearly identical content but different page headers, disclaimers, and logo images. For this particular query, on the whole, 7 out of 20 documents were redundant (3 identical pairs and 4 similar to one top page document). Similar results were found using Yahoo, MSN[2], and A9[3] search engines.

In this paper, we study the current state of popular search engines and evaluate the application of a Bloom filter based near-duplicate detection technique on search results. We demonstrate, using multiple search engines, how a number of results (ranging from 7% to 60%) on search queries are similar and can be eliminated or re-organized. Later, we explore the use of Bloom filters for finding similar objects and demonstrate their effectiveness in compactly representing and quickly matching pages for similarity testing. Although Bloom filters have been extensively used for set membership checks, they have not been analyzed for similarity detection between text documents. Finally, we apply our Bloom filter based technique to effectively remove similar search results and improve user experience. Our evaluation of search results shows that the occurrence of near-duplicates is strongly correlated to: i) the relevance of the document and ii) the popularity of the query. Documents that are considered more relevant and have a higher rank also have more near-duplicates compared to less relevant documents. Similarly, results from the more popular queries have more near-duplicates compared to the less popular ones.

Our similarity matcher can be deployed as a filter over

*This work was supported in part by the Texas Advanced Technology Program, the National Science Foundation (CNS-0411026), and an IBM Faculty Partnership Award.

†This work was done during an internship at IBM Almaden.

[1]Google does have a patent [17] for near-duplicate detection although it is not clear which approach they use.

[2]Results for a recently popular query, "ohio court battle" from both Google and MSN search had a similar behavior, with 10 and 4 out of the top 20 results being identical resp.

[3]A9 states that it uses a Google back-end for part of its search.

any search engine's result set. The overhead of integrating our similarity detection algorithm with search engines only associates about 0.4% extra bytes per document and provides fast matching on the order of milliseconds as described later in section 3. Note that we focus on one main aspect of similarity—text content. This might not completely capture the human-judgement notion of similarity in all cases. However, our technique can be easily extended to include link structure based similarity measures by comparing Bloom filters generated from hyperlinks embedded in web pages.

The rest of the paper is organized as follows. Similarity detection using Bloom filters is described and analyzed in Section 2. Section 3 evaluates and compares our similarity technique to improve search results from multiple engines and for different workloads. Finally, Section 4 covers related work and we conclude with Section 5.

2. SIMILARITY DETECTION USING BLOOM FILTERS

Our similarity detection algorithm proceeds in three steps as follows. First, we use content-defined chunking (CDC) to extract document features that are resilient to modifications. Second, we use these features as set elements for generating Bloom filters[4]. Third, we compare the Bloom filters to detect near-duplicate documents above a certain similarity threshold (say 70%). We start with an overview of Bloom filters and CDCs, and later present and analyze the similarity detection technique for refining web search results.

2.1 Bloom Filter Overview

A Bloom filter of a set U is implemented as an array of m bits [4]. Each element u ($u \in U$) of the set is hashed using k independent hash functions $h_1, \ldots, h_k$. Each hash function $h_i(u)$ for $1 \leq i \leq k$ maps to one bit in the array $\{1 \ldots m\}$. Thus, when an element is added to the set, it sets k bits, each bit corresponding to a hash function, in the Bloom filter array to 1. If a bit was already set it stays 1. For set membership checks, Bloom filters may yield a *false positive*, where it may appear that an element v is in U even though it is not. From the analysis in [8], given $n = |U|$ and the Bloom filter size m, the optimal value of k that minimizes the false positive probability, p^k, where p denotes that probability that a given bit is set in the Bloom filter, is $k = \frac{m}{n} \ln 2$. Previously, Bloom filters have primarily been used for finding set-membership [8].

2.2 Content-defined Chunking Overview

To compute the Bloom filter of a document, we first need to split it into a set of elements. Observe that splitting a document using a fixed block size makes it very susceptible to modifications, thereby, making it useless for similarity comparison. For effective similarity detection, we need a mechanism that is more resilient to changes in the document. CDC splits a document into variable-sized blocks whose boundaries are determined by its Rabin fingerprint matching a predetermined marker value [18]. The number of bits in the Rabin fingerprint that are used to match the marker determine the expected chunk size. For example, given a marker 0x78 and an expected chunk size of 2^k, a rolling (overlapping sequence) 48-byte fingerprint is computed. If the lower k bits of the fingerprint equal 0x78, a new chunk boundary is set. Since the chunk boundaries are content-based, any modifications should affect only a couple of neighboring chunks and not the entire document. CDC has been used in LBFS [15], REBL [13] and other systems for redundancy elimination.

[4]Within a search engine context, the CDCs and the Bloom filters of the documents can be computed offline and stored.

2.3 Bloom Filters for Similarity Testing

Observe that we can view each document to be a set in Bloom filter parlance whose elements are the CDCs that it is composed of[5]. Given that Bloom filters compactly represent a set, they can also be used to approximately match two sets. Bloom filters, however, cannot be used for exact matching as they have a finite false-match probability but they are naturally suited for similarity matching.

For finding similar documents, we compare the Bloom filter of one with that of the other. In case the two documents share a large number of 1's (bit-wise AND) they are marked as similar. In this case, the bit-wise AND can also be perceived as the dot product of the two bit vectors. If the set bits in the Bloom filter of a document are a complete subset of that of another filter then it is highly probable that the document is included in the other. Web pages are typically composed of fragments, either static ones (e.g., logo images), or dynamic (e.g., personalized product promotions, local weather) [19]. When targeting pages for a similarity based "grouping", the test for similarity should be on the fragment of interest and not the entire page.

Bloom filters, when applied to similarity detection, have several advantages. First, the compactness of Bloom filters is very attractive for storage and transmission whenever we want to minimize the meta-data overheads. Second, Bloom filters enable fast comparison as matching is a bitwise-AND operation. Third, since Bloom filters are a complete representation of a set rather than a deterministic sample (e.g., shingling), they can determine inclusions effectively.

To demonstrate the effectiveness of Bloom filters for similarity detection, consider, for example, the pages from the Money/CNN web server (money.cnn.com). We crawled 103 MB of data from the site that resulted in 1753 documents. We compared the top-level page marsh_ceo/index.html with all the other pages from the site. For each document, we converted it into a canonical representation as described later in Section 3. The CDCs of the pages were computed using an expected and maximum chunk size of 256 bytes and 64 KB respectively. The corresponding Bloom filter was of size 256 bytes. Figure 1 shows that two other copies of the page one with the URI `/2004/10/25/news/fortune500/marsh\_ceo/index.htm` and another one with a dynamic URI `/2004/10/25/news/fortune500/marsh_ceo/index.htm?cnn=yes` matched with all set bits in the Bloom filter of the original document.

As another example, we crawled around 20 MB of data (590 documents) from the ibm web site (www.ibm.com). We compared the page `/investor/corpgovernance/index.phtml` with all the other crawled pages from the site. The chunk sizes were chosen as above. Figure 2 shows that two other pages with the URIs `/investor/corpgovernance/cgcoi.phtml` and `/investor/corpgovernance/cgblaws.phtml` appeared similar, matching in 53% and 69% of the bits in the Bloom filter, respectively.

To further illustrate that Bloom filters can differentiate between *multiple* similar documents, we extracted a technical documentation file 'foo' (say) (of size 17 KB) incrementally from a CVS archive, generating 20 different versions, with 'foo' being the original, 'foo.1' being the first version (with a change of 415 bytes from 'foo') and 'foo.19' being the last. As shown in Figure 3, the Bloom filter for 'foo' matched the most (98%) with the closest version 'foo.1'.

[5]For multisets, we make each CDC unique before Bloom filter generation to differentiate multiple copies of the same CDC.

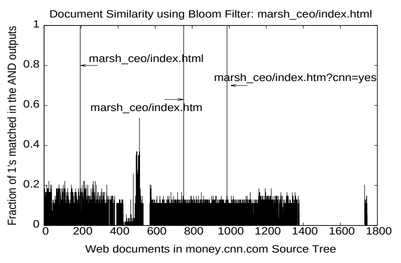


Figure 1: Comparison of the document marsh_ceo/index.html with all pages from the money.cnn.com web site

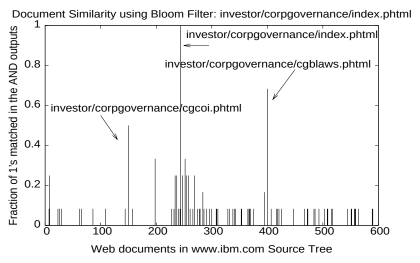


Figure 2: Comparison of the document investor/corpgovernance/index.phtml with pages from www.ibm.com

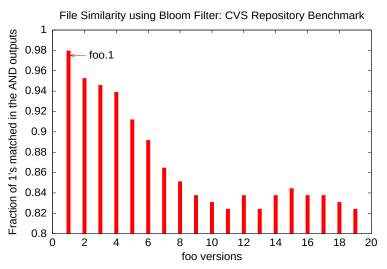


Figure 3: Comparison of the original file 'foo' with later versions 'foo.1', 'foo.2' $\cdots$ 'foo.19'

2.3.1 Analysis

The main consideration when using Bloom filters for similarity detection is the false match probability of the above algorithm as a function of similarity between the source and a candidate document. Extending the analysis for membership testing in [4] to similarity detection, we proceed to determine the expected number of *inferred* matches between the two sets. Let A and B be the two sets being compared for similarity. Let m denote the number of bits (size) in the Bloom filter. For simplicity, assume that both sets have the same number of elements. Let n denote the number of elements in both sets A and B i.e., $|A| = |B| = n$. As before, k denotes the number of hash functions. The probability that a bit is set by a hash function h_i for $1 \leq i \leq k$ is $\frac{1}{m}$. A bit can be set by any of the k hash functions for each of the n elements. Therefore, the probability that a bit is not set by any hash function for any element is $(1 - \frac{1}{m})^{nk}$. Thus, the probability, p, that a given bit is set in the Bloom filter of A is given by:

$$p \;=\; \left(1 - \left(1 - \frac{1}{m}\right)^{nk}\right) \;\approx\; 1 - e^{-\frac{nk}{m}} \qquad (1)$$

For an element to be considered a member of the set, all the corresponding k bits should be set. Thus, the probability of a false match, i.e., an outside element is inferred as being in set A, is p^k. Let C denote the intersection of sets A and B and c denote its cardinality, i.e., $C = A \cap B$ and $|C| = c$.

For similarity comparison, let us take each element in set B and check if it belongs to the Bloom filter of the given set A. We should find that the c common elements will definitely match and a few of the other $(n - c)$ may also match due to the false match probability. By Linearity of Expectation, the expected number of elements of B inferred to have matched with A is

$$\mathrm{E}[\#\text{ of inferred matches}] = (c) \;+\; (n - c)p^k$$

To minimize the false matches, this expected number should be as close to c as possible. For that $(n-c)p^k$ should be close to 0, i.e., p^k should approach 0. This happens to be the same as minimizing the probability of a false positive. Expanding p and under asymptotic analysis, it reduces to minimizing $(1 - e^{-\frac{nk}{m}})^k$. Using the same analysis for minimizing the false positive rate given in [8], the minima obtained after differentiation is when $k = \frac{m}{n} \ln 2$. Thus, the expected number of inferred matches for this value of k becomes

$$\mathrm{E}[\#\text{ of inferred matches}] = c \;+\; (n - c)(0.6185)^{\frac{m}{n}}$$

Thus, the expected number of bits set corresponding to inferred matches is

$$\mathrm{E}[\#\text{ of matched bits}] = m\left[1 - \left(1 - \frac{1}{m}\right)^{k\left(c \;+\; (n-c)(0.6185)^{\frac{m}{n}}\right)}\right]$$

Under the assumption of perfectly random hash functions, the expected number of total bits set in the Bloom filter of the source set A, is mp. The ratio, then, of the expected number of matched bits corresponding to inferred matches in $A \cap B$ to the expected total number of bits set in the Bloom filter of A is:

$$\frac{\mathrm{E}[\#\text{ of matched bits}]}{\mathrm{E}[\#\text{ total bits set}]} = \frac{\left(1 - e^{-\frac{k}{m}\left(c \;+\; (n-c)(0.6185)^{\frac{m}{n}}\right)}\right)}{\left(1 - e^{-\frac{nk}{m}}\right)}$$

Observe that this ratio equals 1 when all the elements match, i.e., $c = n$. If there are no matching elements, i.e., $c = 0$, the ratio $= 2(1 - (0.5)^{(0.6185)^{\frac{m}{n}}})$. For $m = n$, this evaluates to 0.6973, i.e., 69% of matching bits may be false. For larger values, $m = 2n, 4n, 8n, 10n, 11n$, the corresponding ratios are 0.4658, 0.1929, 0.0295, 0.0113, 0.0070 respectively. Thus, for $m = 11n$, on an average, less than 1% of the bits set may match incorrectly. The expected ratio of matching bits is highly correlated to the expected ratio of matching elements. Thus, if a large fraction of the bits match, then it's highly likely that a large fraction of the elements are common.

2.4 Discussion

Previous work on document similarity has mostly been based on shingling or super fingerprints. Using this method, for each object, all the k consecutive words of a document (called k-shingles) are hashed using Rabin fingerprint [18] to create a set of fingerprints (also called features or pre-images). These fingerprints are then sampled to compute a super-fingerprint of the document. Many variants have been proposed that use different techniques on how the shingle fingerprints are sampled (min-hashing, Mod_m, Min_s etc.) and matched [7, 6, 5]. While Mod_m selects all fingerprints whose value modulo m is zero; Min_s selects the set of s fingerprints with the smallest value. The min-hashing approach further refines the sampling to be the min values of say 84 random min-wise independent permutations (or hashes) of the set of all shingle fingerprints. This results in a fixed size sample of 84 fingerprints that is the resulting feature vector. To further simplify matching, these 84 fingerprints can be grouped as 6 "super-shingles" by concatenating 14 adjacent fingerprints [11]. In [13] these are called super-fingerprints. A pair of objects are then considered similar if either all or a large fraction of the values in the super-fingerprints match.

Our Bloom filter based similarity detection differs from the shingling technique in several ways. It should be noted, however, that the variants of shingling discussed above improve upon the original approach and we provide a comparison of our technique with these variants wherever applicable. First, shingling (Mod_m, Min_s) computes document similarity using the intersection of the two feature sets. In our approach, it requires only the bit-wise AND of the two Bloom filters (e.g., two 128 bit vectors). Next, shingling has a higher computational overhead as it first segments the document into k-word shingles ($k = 5$ in [11]) resulting in shingle set size of about $S - k + 1$, where S is the document size. Later, it computes the image (value) of each shingle by applying set (say H) of min-wise independent hash functions (|H|=84 as used in [11]) and then for each function, selecting the shingle corresponding to the minimum image. On the other hand, we apply a set of independent hash functions (typically less than 8) to the chunk set of size on average $\lceil \frac{S}{c} \rceil$ where c is the expected chunk size (e.g., $c = 256$ bytes for $S = 8$ KB document). Third, the size of the feature set (number of shingles) depends on the sampling technique in shingling. For example, in Mod_m, even some large documents might have very few features whereas small documents might have zero features. Some shingling variants (e.g., Min_s, Mod_{2^i}) aim to select roughly a constant number of features. Our CDC based approach only varies the chunk size c, to determine the number of chunks as a trade-off between performance and fine-grained matching. We leave the empirical comparison with shingling as future work.

In general, a compact Bloom filter is easier to attach as a document tag and can be compared simply by matching the bits. Thus, Bloom filter based matching is more suitable for meta crawlers and can be added on to existing search engines without any significant changes.

3. EXPERIMENTAL EVALUATION

In this section, we evaluate Bloom filter-based similarity detection using several types of query results obtained from querying different search engines using the keywords posted on Google Zeitgeist `www.google.com/press/zeitgeist.html`, Yahoo Buzz `buzz.yahoo.com`, and MSN Search Insider `www.imagine-msn.com/insider`.

3.1 Methodology

We have implemented our similarity detection module using C and Perl. The code for content defined chunking is based on the CDC implementation of LBFS [15]. The experimental testbed used a 933 MHz Intel Pentium III workstation with 512 MB of RAM running Linux kernel 2.4.22. The three commercial search engines used in our evaluation are Google *www.google.com*, Yahoo Search *www.yahoo.com*, and MSN Search *www.msnsearch.com*. The Google search results were obtained using the GoogleAPI [1], for each of the search queries, the API was called to return the top 1000 search results. Although we requested 1000 results, the API, due to some internal errors, always returned less than 1000 entries varying from 481 to 897.

For each search result, the document from the corresponding URL was fetched from the original web server to compute its Bloom filter. Each document was converted into a canonical form by removing all the HTML markups and tags, bullets and numberings such as "a.1", extra white space, colons, replacing dashes, single-quotes and double-quotes with single space, and converting all the text to lower case to make the comparison case insensitive. In many cases, due to server unavailability, incorrect document links, page not found errors, and network timeouts, the entire set of requested documents could not always be retrieved.

3.1.1 *Size of the Bloom Filter*

As we discussed in the section 2, the fraction of bits that match incorrectly depends on the size of the Bloom filter. For a 97% accurate match, the number of bits in the Bloom filter should be 8x the number of elements (chunks) in the set (document). When applying CDC to each document, we use the expected chunk size of 256 bytes, while limiting the maximum chunk size to 64 KB. For an average document of size 8 KB, this results in around 32 chunks. The Bloom filter is set to be 8x this value i.e., 256 bits. To accommodate large documents, we set the maximum document size to 64 KB (corresponding to the maximum chunk size). Therefore, the Bloom filter size is set to be 8x the expected number of chunks (256 for document size 64 KB) i.e., 2048 bits or 256 bytes, which is a 3.2% and 0.4% overhead for document size of 8 KB and 64 KB respectively.

Example. When we applied the Bloom filter based matcher to the "emacs manual" query (Section 1), we found that the page `www.linuxselfhelp.com/gnu/emacs/html_chapter/emacs_toc.html` matched the other three, `www.delorie.com/gnu/docs/emacs/emacs_toc.html`, `www.cs.utah.edu/dept/old/texinfo/emacs19/emacs_toc.html`, and `www.dc.turkuamk.fi/docs/gnu/emacs/emacs_toc.html`, with 74%, 81% and 95% of the Bloom filter bits matching, respectively. A 70% matching threshold would have identified and grouped all these 4 pages together.

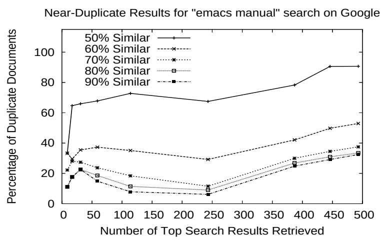


Figure 4: "emacs manual" query search results (Google)

3.2 Effect of the Degree of Similarity

In this section, we evaluate how the degree of similarity affects the number of documents that are marked similar. The degree of similarity is the percentage of the document data that matches (e.g., a 100% degree of similarity is an identical document). Intuitively, the higher the degree of similarity, the lower the number of documents that should match. Moreover, the number of documents that are similar depends on the total number of documents retrieved by the query. Although, we initially expected a linear behavior, we observed that the higher ranked results (the top 10 to 20 results) were also the ones that were more duplicated.

Using GoogleAPI, we retrieved 493 results for the "emacs manual" query. To determine the number of documents that are similar among the set of retrieved documents, we use a union-find data structure for clustering Bloom filters of the documents based on similarity. Figure 4 shows that for 493 documents retrieved, the number of document clusters were 56, 220, 317, 328, 340, when the degree of similarity was 50, 60, 70, 80, 90%, respectively. Each cluster represents a set of similar documents (or a single document if no similar ones are found). We assume that a document belongs to a cluster if it is similar to a document in the cluster, i.e., we assume that similarity is transitive for high values of the degree of similarity (as in [9]). The fraction of duplicate documents as shown in figure 4, decreases from 88% to 31% as the degree of similarity increases from 50% to 90%. As the number of retrieved queries increase from 10 to 493, the fraction of duplicate documents initially decrease and then increase forming a minima around 250 results. The decrease was due to the larger aliasing of "better" ranked documents. However, as the number of results increase, the initial set of documents get repeated more frequently, increasing the number of duplicates. Similar results were obtained for a number of other queries that we evaluated.

3.3 Effect of the Search Query Popularity

To get a representative collection of the types of queries

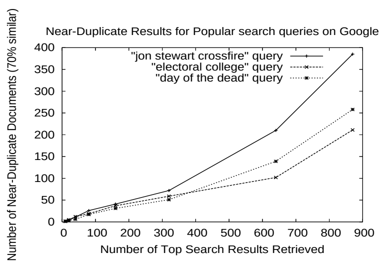


Figure 5: Search results for the top 3 queries on Google

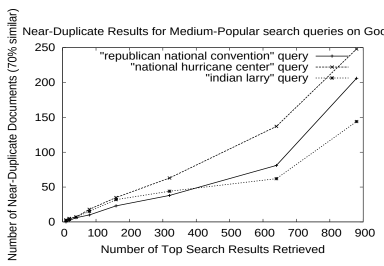


Figure 6: Search results for 3 medium-popular queries on Google

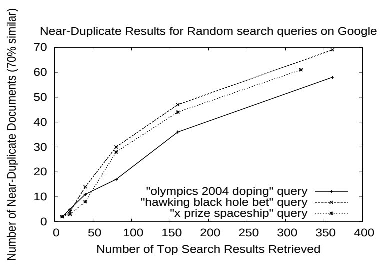


Figure 7: Search results for 3 random queries on Google

performed on search engines, we selected samples from Google Zeitgeist (Nov. 2004) of three different query popularities: i) Most Popular, ii) Medium-Popular, and iii) Random.

For most-popular search queries, the three queries selected in order were—"jon stewart crossfire"(TP1), "electoral college"(TP2) and "day of the dead"(TP3). We computed the number of duplicates having 70% similarity (atleast 70% of the bits in the filter matched) in the search results. Figure 5 shows the corresponding number of duplicates for a maximum of 870 search results from the Google search API. The TP1 query had the maximum fraction of near-duplicates, 44.3%, while the other two TP2 and TP3 had 29.7% and 24.3%, respectively. Observe that the most popular query TP1was the one with the most duplicates.

For the medium popular queries, we selected three queries from the list "Google Top 10 Gaining Queries" for the week ending Aug. 30, 2004 on the Google Zeitgeist—"indian larry" (MP1), "national hurricane center"(MP2) and "republican national convention"(MP3). Figure 6 shows the corresponding search results having 70% similarity for a maximum of 880 documents from the Google search engine. The fraction of near-duplicates among 880 search results ranged from 16% for MP1 to 28% for MP2.

For a non-popular query sample, we selected three queries at random—"olympics 2004 doping", "hawking black hole bet", and "x prize spaceship". The Google API retrieved only about 360 results for the first two queries and 320 results for the third query. Figure 7 shows the number of near-duplicate documents in the search results corresponding to the three queries. The fraction of near-duplicates in all these queries were in the same range, around 18%.

As we observed earlier, as the popularity of queries decrease so do the number of duplicate results. The most popular queries had the largest number of near-duplicate results, the medium ones fewer, and the random queries the lowest.

3.4 Behavior of different search engines

The previous experiments all compared the results from the Google search engine. We next evaluate the behavior of all three search engines, Google, Yahoo and MSN search in returning near-duplicate documents for the 10 popular queries featured on their respective web sites. To our knowledge, Yahoo and MSN search do not provide an API similar to the GoogleAPI for doing automated retrieval of search results. Therefore, we manually made HTTP requests to the URLs corresponding to the first 50 search results for a query.

We plot minimum, average and maximum number of near-duplicate (atleast 70% similar) search results in the 10 popular queries. The three whiskers on each vertical bar in Figures 8,9,10 represent min., avg., and max. in order. Figure 8 shows the results for Google, with average number of near-duplicates ranging from 7% to 23%. Figure 9 shows near-duplicates in Yahoo results ranging from 12% to 25%. Figure 10 shows the results for MSN, where the near-duplicates range from 18% to 26%. Comparing the earlier "emacs manual" query, MSN had 32% near duplicates while Yahoo had 22%. These experiments support our hypothesis that current search engines return a significant number of near-duplicates. However, these results do not in any way suggest that any particular search engine performs better than the others.

3.5 Analyzing Response Times

In this section, we analyze the response times for performing similarity comparisons using Bloom filters. The timings include (a) the (offline) computation time to compute the document CDC hashes and generating the Bloom filter, and (b) the (online) matching time to determine similarity using bitwise AND on Bloom filters and time for insertions and unions in a union-find data structure for clustering.

Exp. Chunk Sizes File Size	256 Bytes (ms)	512 Bytes (ms)	2 KB (ms)	8 KB (ms)
10 KB	0.3	0.3	0.2	0.2
100 KB	4	3	3	2
1 MB	29	27	26	24
10 MB	405	321	267	259

Table 1: CDC hash computation time for different files and expected chunk sizes

Document Size	# of chunks (n)	k = 2 (ms)	k = 4 (ms)	k = 8 (ms)
10 KB	35	11	12	14
100 KB	309	118	120	126
1 MB	2959	961	1042	1198
10 MB	30463	11792	11960	12860

Table 2: Time (ms) for Bloom filter generation for different document sizes (expected chunk size 256 bytes)

Bloom Filter Size (Bits)	100	300	625	1250	2500	5000
Time (μsec)	1.9	2.4	2.9	3.9	6.2	10.7

Table 3: Time (microseconds) for computing the bitwise AND of Bloom filters for different sizes

Table 1 shows the CDC hash computation times for a complete document (of size 10 KB, 100 KB, 1 MB, 10 MB) for different expected chunk sizes (256 bytes, 512 bytes, 2 KB, 8 KB). The Bloom filter generation times are shown in Table 2 for different values (2, 4, 8) of the number of hash functions (k) and different number of chunks (n). Although the Bloom filter generation times appear high relative to the CDC times, it is more an artifact of the implementation of the Bloom filter code in Perl instead of C and not due to any inherent complexity in the Bloom filter code. A preliminary implementation in C reduced the Bloom filter generation time by an order of magnitude.

For the matching time overhead, Table 3 shows the pairwise matching time for two Bloom filters for different filter

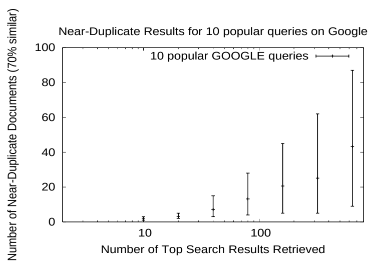


Figure 8: Search results for 10 popular queries on Google

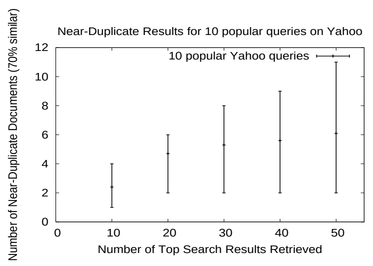


Figure 9: Search results for 10 popular queries on Yahoo Search

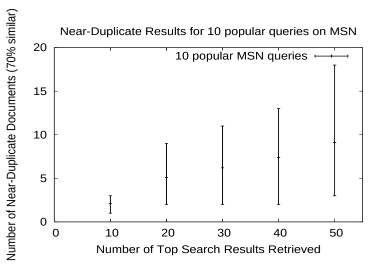


Figure 10: Search results for 10 popular queries on MSN Search

No. of Results / Search Query	10	20	40	80	160	320
"emacs manual"	1	4	15	66	286	1233
"ohio court battle"	1	7	24	98	369	1426
"hawking black hole bet"	1	6	23	88	364	1407

Table 4: Matching and Clustering time (in ms)

sizes ranging from 100 bits to 5000 bits. The overall matching and clustering time for different query requests is shown in Table 4. Overall, using untuned Perl and C code, for clustering 80 results each of size 10 KB for the "emacs manual" query would take around 80*0.3 ms + 80* 14 ms + 66ms = 1210 ms. However, the Bloom filters can be computed and stored apriori reducing the time to 66 ms.

4. RELATED WORK

The problem of near-duplicate detection consists of two major components: (a) extracting document representations aka features (e.g., shingles using Rabin fingerprints [18], super-shingles [11], super-fingerprints [13]), and (b) computing the similarity between the feature sets. As discussed in Section 2, many variants have been proposed that use different techniques on how the shingle fingerprints are sampled (e.g., min-hashing, Mod_m, Min_s) and matched [7, 6, 5]. Google's patent for near-duplicate detection uses another shingling variant to compute fingerprints from the shingles [17].

Our similar detection algorithm uses CDC [15] for computing document features and then applies Bloom filters for similarity testing. In contrast to existing approaches, our technique is simple to implement, incurs only about 0.4% extra bytes per document, and performs faster matching using only bit-wise AND operations. Bloom filters have been proposed to estimate the cardinality of set intersection in [8] but have not been applied for near-duplicate elimination in web search. We recently learned about Bloom filter replacements [16] which we will explore in the future.

Page and site similarity has been extensively studied for web data in various contexts, from syntactic clustering of web data [7] and its applications for filtering near duplicates in search engines [6] to storage space and bandwidth reduction for web crawlers and search engines. In [9], replica identification was also proposed for organizing web search results. Fetterly et al. examined the amount of textual changes in individual web pages over time in the PageTurner study [12] and later investigated the temporal evolution of clusters of near-duplicate pages [11]. Bharat and Broder investigated the problem of identifying mirrored host pairs on the web [3]. Dasu et al. used min hashing and sketches to identify fields having similar values in database tables [10].

5. CONCLUSIONS

In this paper, we applied a Bloom filter based similarity detection technique to refine the search results presented to the user. Bloom filters compactly represent the entire document and can be used for quick matching. We demonstrated how a number of results of popular and random search queries retrieved from different search engines, Google, Yahoo, MSN, are similar and can be eliminated or re-organized.

6. ACKNOWLEDGMENTS

We thank Rezaul Chowdhury, Vijaya Ramachandran, Sridhar Rajagopalan, Madhukar Korupolu, and the anonymous reviewers for giving us valuable comments.

7. REFERENCES

[1] Google web apis (beta), *http://www.google.com/apis.*

[2] Yahoo results getting more similar to google `http://www.searchenginejournal.com/index.php?p=584&c=1`.

[3] K. Bharat and A. Broder. Mirror, mirror on the web: a study of host pairs with replicated content. *Comput. Networks*, 31(11-16):1579–1590, 1999.

[4] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[5] A. Z. Broder. On the resemblance and containment of documents. In *SEQUENCES*, 1997.

[6] A. Z. Broder. Identifying and filtering near-duplicate documents. In *COM*, pages 1–10, 2000.

[7] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the web. In *WWW'97*.

[8] A. Z. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. In *Allerton'02*.

[9] J. Cho, N. Shivakumar, and H. Garcia-Molina. Finding replicated web collections. *SIGMOD Rec.*, 2000.

[10] T. Dasu, T. Johnson, S. Muthukrishnan, and V. Shkapenyuk. Mining database structure; or, how to build a data quality browser. In *SIGMOD*, 2002.

[11] D. Fetterly, M. Manasse, and M. Najork. On the evolution of clusters of near-duplicate web pages. In *LA-WEB*, 2003.

[12] D. Fetterly, M. Manasse, M. Najork, and J. Wiener. A large-scale study of the evolution of web pages. In *WWW*, 2003.

[13] P. Kulkarni, F. Douglis, J. D. LaVoie, and J. M. Tracey. Redundancy elimination within large collections of files. In *USENIX Annual Technical Conference, General Track*, pages 59–72, 2004.

[14] J. C. Mogul, Y.-M. Chan, and T. Kelly. Design, implementation, and evaluation of duplicate transfer detection in HTTP. In *NSDI*, pages 43–56, 2004.

[15] A. Muthitacharoen, B. Chen, and D. Mazieres. A low-bandwidth network file system. In *SOSP*, 2001.

[16] R. Pagh, A. Pagh, and S. S. Rao. An optimal bloom filter replacement. In *SODA*, 2005.

[17] W. Pugh and M. Henzinger. Detecting duplicate and near-duplicate files, US Patent # 6658423.

[18] M. O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Harvard University, 1981.

[19] L. Ramaswamy, A. Iyengar, L. Liu, and F. Douglis. Automatic detection of fragments in dynamically generated web pages. In *WWW*, 2004.

JXP: Global Authority Scores in a P2P Network [*]

Josiane Xavier Parreira
Max-Planck Institute for Computer Science
Stuhlsatzenhausweg 85
66123 Saarbrücken, Germany
jparreir@mpi-sb.mpg.de

Gerhard Weikum
Max-Planck Institute for Computer Science
Stuhlsatzenhausweg 85
66123 Saarbrücken, Germany
weikum@mpi-sb.mpg.de

ABSTRACT

This document presents the *JXP* algorithm for dynamically and collaboratively computing PageRank-style authority scores of Web pages distributed in a P2P network. In the architecture that we pursue, every peer crawls and indexes Web fragments at its discretion, driven by the thematic profile or overlay neighborhood of the peer. The JXP algorithm runs at every peer, and is initialized by a local authority computation on the basis of the locally available Web fragment. Peers collaborate by periodically "meeting" with other peers in the network. Whenever two peers meet they exchange their local information and use this new information to improve their local authority scores. Even though only local computations are performed, the JXP scores approximate the global importance of pages in the entire network. The storage demand of each peer is linear in the number of Web pages and the locally stored Web fragment. Experiments show the quality and practical viability of the JXP algorithm.

1. INTRODUCTION

This paper is motivated by efforts towards building a peer-to-peer (P2P) Web search engine. P2P networks [26, 23, 22] have proven to be a scalable alternative to traditional client/server systems. However, the characteristics of such networks, namely, no central processing and high dynamics (peers constantly joining and leaving the network), pose a challenge when designing a new search engine for a P2P network. We assume that every peer has a full-fledged Web search engine and can crawl and index interesting Web fragments at its discretion, driven by thematic profiles of the user or the neighborhood in some form of "semantic" overlay network. Peers collaborate on difficult queries that cannot be satisfactorily answered with the locally available index alone (using query routing strategies [14, 8, 4], but they are autonomous in terms of their crawling strategies and what data they keep in their local indexes).

In the context of Internet search engines, link-based ranking algorithms that assign authority scores to pages, based on their "importance" on the Web, have been proven to make the search more effective [5, 18]. For instance, Google [2] uses *PageRank*, an Eigenvector-based algorithm that determines the importance of a page based on the importance of the pages that point to it. The PageRank computation is quite expensive as it involves iteratively computing the principal Eigenvector of a matrix derived from the Web link graph. An alternative but equivalent view of PageRank is that it computes stationary probabilities of a Markov chain that corresponds to a random walk on the Web. Recent work has made progress on efficiently computing PageRank scores [17, 16, 6, 12], but the high storage demand of the – sparse but nonetheless huge - underlying matrix seems to limit this kind of link analysis to a central server with very large memory. The most relevant prior work on distributed PageRank computation is [27], but this method assumes that sites compute local PageRank values based on the data that they originally host, thus strongly relying on the assumption that sites have disjoint fragments of the Web graph and are relatively reliable servers.

In this document we propose the *JXP*[1], an algorithm for dynamically computing, in a decentralized P2P manner, global authority scores when the Web graph is spread across many autonomous peers. The peers' graph fragments may overlap arbitrarily, and peers are (a priori) unaware of other peers' fragments.

The main idea of the JXP algorithm is as follows. Each peer computes the authority scores of the pages that it has in its local index, by locally running the standard PageRank algorithm. To avoid confusion with the true, global PageRank values, we refer to these local scores as the (peer-specific) *PageWeight* scores of the pages known by the peer. Note that a page may be known and indexed by multiple peers, and these may have different PageWeights for that same page. A peer gradually increases its knowledge about the rest of the network by meeting with other, randomly chosen, peers and exchanging information. To improve the initial PageWeight scores and approximating the global authority of pages, a peer uses what it learns from the other, randomly met, peers, combined with its own local information, for recomputing the PageWeight scores. Although the computations are strictly local, our goal is towards a notion

[*]Partially supported by the EU within the 6th Framework Programme under contract 001907 "Dynamically Evolving, Large Scale Information Systems" (DELIS).

Copyright is held by the author/owner. Eighth International Workshop on the Web and Databases (WebDB 2005), June 16-17, 2005, Baltimore, Maryland.

[1]JXP is an acronym for juxtaposed approximate PageRank, and also happens to be the initials of the first author.

of importance of the pages in the whole web graph. This process, in principle, runs forever, and our experiments indicate that the resulting JXP scores quickly converge to the true, global PageRank scores.

When two peers meet they temporarily form the union of their Web graph fragments; for representing the unknown part of the Web graph (which is spread across many further peers) a state-lumping technique for Markov chains is used. After recomputing PageWeights, only the resulting authority scores of each peer's pages of interest are kept. Technically, the computation involves some difficulties because of the need for proper normalization with partial knowledge of the Web graph; another complication is that the graphs of two peers may have radically different sizes and may arbitrarily overlap. It is important to emphasize that peers do not accumulate the graph fragments that they learn about when meeting other peers. So we ensure that the storage requirements are low, linear in the number of pages of interest and the local index size, and the PageRank computation is scalable, as the algorithm always runs on relative small graphs, independent of the number of peers in the network.

The locally recomputed PageWeights already reflect the importance of a page in the entire network, but different peers may have very different views, e.g., in terms of the size of their local graphs. Therefore, the JXP algorithm can optionally normalize authority scores based on *PeerWeights* that reflect the reputation and trust of peers.

The rest of the document is organized as follows. Section 2 briefly discusses related work. A quick review of the standard PageRank algorithm is presented in Section 3. In Section 4 we present the *JXP* algorithm for computing authority scores. Section 5 shows preliminary experimental results of the algorithm. Section 6 concludes this paper and presents ideas for future work.

2. RELATED WORK

Link-based authority ranking has received great attention in the literature, starting with the seminal work of Brin and Page [5] and Kleinberg [18]. Good surveys of the many improvements and variations are given in [10] and [19].

In [27] Wang and DeWitt presented a distributed search engine framework, in which the authority score of each page is computed by performing the PageRank algorithm at the Web server that is the responsible host for the page, based only on the intra-server links. They also assign authority scores for each server in the network, based on the inter-server links, and then approximate global PageRank values by combining local page authority scores and server authority values. This work bears relationships to the work by Haveliwala et al. [17] that postulates a block structure of the link matrix and exploits this structure for faster convergence of the global PageRank computation. Our idea is related to the approach of Wang and DeWitt in the sense that we also use local page scores and peer scores, but our algorithm does not require any particular distribution of the pages among the sites whereas the method by Wang and DeWitt relies on the fact that the graph fragments of the servers are disjoint. In particular, the JXP algorithm can work in scenarios where peers are completely autonomous and crawl and index Web data at their discretion, resulting in arbitrarily overlapping graph fragments.

Chen et al. [11] proposed a way of approximating the PageRank value of a page locally, by expanding a small subgraph around the page of interest, placing an estimated PageRank at the boundary nodes of the subgraph, and running the standard algorithm. In a P2P scenario, this algorithm would require the peers to query other peers about pages that have links to their local nodes, and pages that point to pages that point to local pages, and so on. This would be a significant burden for a highly dynamic P2P network. The JXP algorithm, on the other hand, requires much less interaction among peers.

Other techniques for approximating PageRank-style authority scores with partial knowledge of the global graph use state-lumping techniques from the stationary analysis of large Markov chains [20, 12]. These techniques have been developed for the purpose of incremental updates to authority scores when only small parts of the graph have changed. Dynamic computation in a P2P network is not an issue in this prior work. Another work related to this topic is the one by Broder and Lempel [6], where they have presented a graph aggregation method in which pages are partitioned into hosts and the stationary distribution is computed in a two-step approach, combining the stationary distribution inside the host and the stationary distribution inter-hosts.

A storage-efficient approach to computing authority scores is the OPIC algorithm developed by Abiteboul et al. [3]. This method avoids having the entire link graph in one site, which, albeit sparse, is very large and usually exceeds the available main memory size. It does so by randomly (or otherwise fairly) visiting Web pages in a long-running crawl process and performing a small step of the PageRank power iteration (the numeric technique for computing the principal Eigenvector) for the page and its successors upon each such visit. The bookkeeping for tracking the gradually approximated authority of all pages is carried out at a central site, the Web-warehouse server. This is not a P2P algorithm either.

In [24], Sankaralingam et al. presented a P2P algorithm in which the PageRank computation is performed at the network level, with peers constantly updating the scores of their local pages and sending these updated values through the network. Shi et al. [25] also compute PageRank at the network level, but they reduce the communication among peers by distributing the pages among the peers according to some load-sharing function. In contrast to these P2P-style approaches, our JXP algorithm performs the actual computations locally at each peer, and thus needs a much smaller number of messages.

3. REVIEW OF PAGERANK

The basic idea of PageRank is that if page p has a link to page q then the author of p is implicitly endorsing, i.e., giving some importance to page q. How much p contributes to the importance of q is proportional to the importance of p itself.

This recursive definition of importance can be described by the stationary distribution of a Markov chain that describes a random walk over the graph, where we start at an arbitrary page and in each step choose a random outgoing edge from the current page. To ensure the ergodicity of this Markov chain (i.e., the existence of stationary page-visit probabilities), additional random jumps to uniformly chosen target pages are allowed with small probability α. Formally, the PageRank of a page q is defined as:

$$PR(q) = \alpha \times 1/N + (1-\alpha) \times \sum_{p|p \to q} PR(p)/out(p) \quad (1)$$

where N is the total number of pages in the link graph, $PR(p)$ is the PageRank score of the page p, $out(p)$ is the outdegree of p, the sum ranges over all link predecessors of q, and α is the random jump probability, with $0 < \alpha < 1$ and usually set to a value like 0.15.

PageRank values are usually computed by initializing a PageRank vector with uniform values $1/N$, and then applying a power iteration method, with the previous iteration's values substituted in the right-hand side of the above equation for evaluating the left-hand side. This iteration step is repeated until sufficient convergence, i.e., until the PageRank scores of the high-authority pages of interest exhibit only minor changes.

4. THE JXP ALGORITHM

The goal of the JXP algorithm is to approximate global authority scores by performing local computations only, with low storage costs, and a moderate number of interactions among peers. JXP runs on every peer in the network, where each peer stores only its own local fragment of the global graph. The algorithm does not assume any particular assignment of pages to peers, and overlaps among the graph fragments of the peers are allowed.

The JXP algorithm has three components that are described in the subsequent subsections:

1. the local PageWeight computation based on an extended local graph,
2. the interaction with other peers, chosen at random, and
3. optional considerations on the normalization of the resulting PageWeights, from an individual peer's viewpoint, by taking into accout the PeerWeights, the relative authority or trust of the peers.

4.1 Local PageWeight Computation

For the local approximation of the global graph, as viewed from a peer with partial knowledge of the link structure, we construct an *extended local graph*. There are two different cases to consider: initial PageWeight computations by a peer that is just by itself, and PageWeight refinements when two peers meet. In the first case, we add to the local graph a special node, that we call the *world node*, representing the part of the global graph that is not stored at and not known to the peer. This is an application of the state-lumping techniques used in the analysis of large Markov models [20]. In the second case, when two peers meet we form the union of the two local graphs and extend them by a world node. We discuss the graph merging for meeting peers in Subsection 4.2. In both cases, the local PageWeight scores are then computed by running the PageRank power iteration algorithm on this extended local graph. Figure 1 depicts a peer's local graph with the additional world node.

The world node has special features, regarding its own PageWeight and how it is connected to the local graph. As it represents all the pages that are not stored at the peer, we take all the links from local pages to external pages and make them point to the world node. In the same way, as

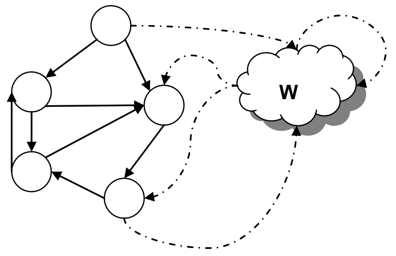


Figure 1: Extended local graph of a peer

the peer learns about external links that point to one of the local pages, we assign these links to the world node. (This is when the peer meets with another peer, see Subsection 4.2.) For a better aproximation of the total authority score mass that is received from external pages, we weigh every link from the world node based on how much of the authority score is received from the original page that owns the link. Another special feature of the world node is that it contains a self-loop link, that represents links from external pages pointing to other external pages. The PageWeight of the world node is equivalent to the sum of PageWeights of the external pages that it represents. During the local PageWeight computation the probability of a random jump to the world node is also set proportional to the number of pages it represents.

Let $Internal = \{int_1, int_2, \ldots, int_n\}$ be the set of local pages and $External = \{ext_1, ext_2, \cdots, ext_{(N-n)}\}$ the set of external pages. As peers gradually learn about external pages, we also define $Known$ as a subset from $External$ that contains pages that the peer has learned about and that have links to one of the pages in $Internal$. Then the PageWeight of the world node W and the weights of a link from the world node W to some local node $a \in Internal$ can be expressed as:

$$\begin{aligned} PageWeight(W) &= \sum_{i \in External} PageWeight(i) \\ &= 1 - \sum_{\forall j \in Internal} PageWeight(j) \end{aligned} \quad (2)$$

$$\begin{aligned} weight(W \to a) = &\frac{1}{PageWeight(W)} \\ &\times \sum_{i \in Known \;\&\; i \to a} \frac{PageWeight(i)}{Outdegree(i)} \end{aligned} \quad (3)$$

When the local PageWeight computation on the extended local graph terminates, the PageWeights of pages learned from other peers are also stored at the local peer, as they can be used for better estimation of the weights of links from the world node before the next local evaluation of PageWeights. We also estimate a PageWeight score for the pages that are still not known by the peer, based on an estimation of the total number of the pages in the graph. Following the same formulation above, let $Unknown = \{ukwn_1, ukwn_2, \ldots,$ $uknw_m\}$, where $m = N - (size(Internal) + size(Known))$.

The PageWeights of the unknown pages are defined as

$$\begin{aligned} PageWeight(ukwn_i) = (1 - (&\sum_{i \in Internal} PageWeight(i) \\ &+ \sum_{j \in Known} PageWeight(j))) \times \frac{1}{m} \end{aligned} \quad (4)$$

for all pages $ukwn_i$ in $Unknown$.

Before the execution of the local PageWeight algorithm, an initialization procedure, described in Algorithm 1, is performed. This procedure estimates the size of the global graph, creates the world node and attaches it to the local graph, sets an initial PageWeight scores ($1/N$ for all pages on the local graph, and $(N - n)/N$ for the world node), and runs the PageRank power iteration algorithm on the extended local graph, to improve the previous scores.

Algorithm 1 Initial PageWeight Computation

```
1: input: local graph $G$ and est. size of global graph $N$
2: n ← size($G$)
3: Create world node $W$
4: $PageWeight(p|p \in G) \leftarrow 1/N$
5: $PageWeight(W) \leftarrow (N - n)/N$
6: add $W$ to $G$
7: run PageRank power iteration on $G$
8: $Update(PageWeights)$
```

4.2 Peer Meetings

Algorithm 2 shows the pseudo code of the PageWeight algorithm. It starts with the peer choosing another peer in the network at random to exchange information. Once the peers have exchanged their local knowledge, it is time for them to combine both local and external information. It is important to point out that this process runs at both peers independently of each other. So we fully preserve the autonomy of peers and asynchronous nature of communication and computation in a P2P network.

Algorithm 2 The JXP Algorithm

```
1: input: local graph $G$, world node $W$, known pages $KP$
2: repeat
3:    Contact a $RandomPeer$ in the network and exchange information
4:    $extG \leftarrow$ local graph of $RandomPeer$
5:    $extW \leftarrow$ world node of $RandomPeer$
6:    $extKP \leftarrow$ list of known pages of $RandomPeer$
7:    $KP \leftarrow combinePageLists(KP, extKP)$
8:    $mergedG \leftarrow combineGraphs(G, extG)$
9:    $mergedW \leftarrow combineWorldNodes(W, extW)$
10:   add $mergedW$ to $mergedG$
11:   run PageRank power iteration on $mergedG$
12:   $Update(PageWeights)$
13:   $Update(W)$
14:   $Discard(extG, extW, extKP, mergedG, mergedW)$
```

The lists of known pages are combined by averaging the PageWeight scores of the pages from both lists. Pages that were unknown to the peer until the current iteration are added to the local list of known pages. Local graphs are combined by simply forming the union of nodes and connecting nodes by the known links between them. The PageWeight computation will always yield a correct result even if there is no link between the graphs, because of the world node and the corresponding random jump probabilities.

Combining the world nodes of the two meeting peers consists of merging their list of outgoing links, removing links that originaly come from a page that is already represented in the merged graph, and adjusting the PageWeight of the combined world node to reflect the sum of PageWeights of pages that do not belong to the combined graph. The new world node that results from this merging is then connected to the merged graph and the PageRank power iteration algorithm is again performed, yielding updated PageWeight values. The graphs are then disconnected and the local world node is recreated from the merged world node, by keeping only the links that point to a page in the local graph. The PageWeight of the world node is also recomputed. The extended graph, world node, and list of pages that belong to the other peer are then discarded. Figure 2 illustrates the process of combining and disconnecting local graphs and world nodes.

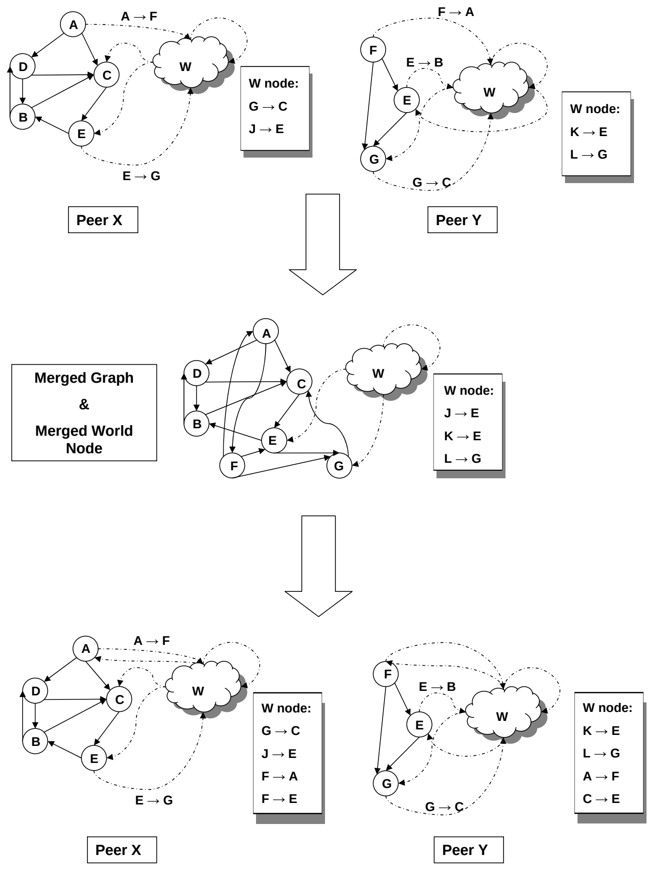


Figure 2: Illustration of the combining and disconnecting steps.

4.3 Considering PeerWeights

Two meeting peers may have fairly different characteristics in terms of their local index sizes, knowledge or awareness of the global graph characteristics (e.g., because one peer has already met many other peers, whereas the other peer just joined the P2P network), and trustworthiness or recognition by other peers. Thus, when we combine the two local graphs it could make sense to treat the two peers with different weights. The JXP algorithm can optionally take PeerWeights into account. To this end, it weighs all edges in the merged graph as follows: the weight of an edge $p \rightarrow q$ that has been known to both peers is set to $PeerWeight_1 \cdot weight_1(p \rightarrow q) + PeerWeight_2 \cdot weight_2(p \rightarrow q)$ where the subscripts 1 and 2 refer to the two meeting peers. This gives higher weight to the view of the more authoritative, knowledgable, or credible peer.

For computing PeerWeights we have a number of options. The simplest approach is to construct a peer graph with peers as nodes and edges between two peers if there is a link between two pages that are locally indexed by the two peers. This graph would be gradually constructed as peers meet over time; but given that it is orders of magnitude smaller than the Web graph, the peers' local information about this peer graph could be easily disseminated in the P2P network so that peers learn the full peer graph more quickly. An alternative, which we would actually prefer, is to compute PeerWeights on the basis of the peers' behavior and trustworthiness or recognition in the P2P community. This would serve to discriminate "good" peers from "bad", i.e., selfish or even malicious, peers. Recently, various approaches have been proposed in the literature for monitoring and tagging peers, see e.g., [15, 21, 7]. The JXP framework can easily incorporate such techniques, and we are currently investigating the implementation issues.

5. EXPERIMENTS

We evaluated the performance of the PageWeight algorithm on two different datasets: a synthetic generated dataset, and subsets of the Amazon.com dataset. The synthetic web graphs were generated using the "recursive matrix"(R-MAT) model [9], a powerful tool that can, given a few parameters, quickly generate realistic web graphs, with the power law degree distribution property. The Amazon.com dataset contains information about approximately 126,000 products (mostly books) offered by Amazon.com. The data was obtained in February 2005, through a Web Service provided by Amazon.com [1], and it is equivalent to a partial crawl of the corresponding web site. The graphs were created by considering the products as a node in the graph. For each product, pointers to similar recommended products are available in the dataset. These pointers define the edges in our graphs.

5.1 Setup

After creating the graphs, we distributed the nodes into the peers. Nodes are assigned to peers either at random, allowing some overlap among the local graphs, or simulating a crawler in each peer, starting with a random seed page and following the links and fetching the next nodes, in a breadth-first approach, up to a certain predefined depth.

The performance of the algorithm is evaluated by comparing the top-k ranking of the pages given by the JXP algorithm at each peer against the global top-k ranking of the pages given by the PageRank computation in the complete web graph. For this comparison we use Spearman's footrule distance [13], defined as $F(\sigma_1, \sigma_2) = \sum_{i=1}^{k} |\sigma_1(i) - \sigma_2(i)|$ where $\sigma_1(i)$ and $\sigma_2(i)$ are the positions of the page i in the first and second ranking. In case a page is present in one of the top-k rankings and does not appear in the second top-k ranking, its position in the latter ranking is considered to be $k + 1$. We normalized the Spearman's footrule distance, to obtain values between 0 and 1, with 0 meaning that the rankings are identical, and 1 meaning that the rankings have no pages in common.

5.2 Results

We tested our algorithm on a subset of the Amazon.com dataset containing books from the category "Computers & Internet", which contains 10,595 pages and 42,548 links. For comparison, we also tested JXP on a synthetic generated graph with 2,036 pages and 8,477 links. Results are shown in Figure 3, where the x-axis corresponds to the number of meetings a peer performed, and each line on the graph represents a different peer. For all the cases we chose the top-k level to be 100 [2].

Number of Meetings	Footrule Distance	Linear Error ($\times 10^{-5}$)	Known Pages
0	0.5773	7.82	579.4
5	0.5772	7.20	10403.3
10	0.4710	6.50	10595
20	0.3618	5.33	10595
50	0.1169	2.12	10595
90	0.0553	0.99	10595

Table 1: Average results for "Computers & Internet" data.

Besides the Spearman's footrule distance, we also measured, after each meeting, the total number of pages known by the peer, and the *linear score error*, that we defined as being the average of the absolute difference between the PageWeight score and the global PageRank score of all known pages. Table 1 shows the average of the observed values, after a certain number of meetings, on the "Computers & Internet" subset, for the case when pages are randomly distributed among peers. The table clearly shows that both Spearman's footrule distance and the linear score error quickly decrease with the number of meetings [3].

Based on these results we can conclude that, as the number of meetings among peers increases, the distance between the PageWeight scores and the PageRank scores, as well as the rankings produced by them, decreases at a high rate. Thus, PageWeight provides a good approximation to the global PageRank scores of the pages.

6. CONCLUSIONS AND FUTURE WORK

In this paper we presented the JXP algorithm for dynamically computing authority scores of pages distributed in a P2P network. The algorithm runs in a completely decentralized manner, with every peer running the algorithm independently from the other peers in the network. The algorithm requires that peers meet randomly and exchange their local graph fragments, but the overall long-running process does not require any peer to keep more information other than its own local graph fragment and the PageWeights of the pages of interest. The computations themselves are strictly local, yet we can approximate the global importance of a page in the whole graph with acceptable accuracy. Our experiments, albeit preliminary, indicate that the algorithm performs very well, converges fairly quickly, and incurs low overhead.

Future work includes more comprehensive experimentation with larger graphs and a larger number of peers. We expect the algorithm to scale up well as the number of peers increase without increasing the local data volume and local

[2]We also measured different top-k levels and obtained similar results.

[3]The same behavior was observed for the other tested graphs, but these results are omitted for space reasons.

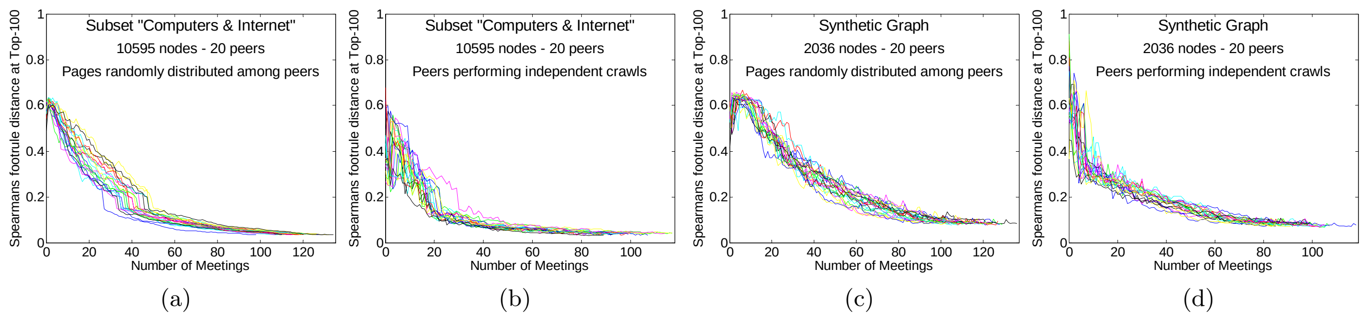


Figure 3: Preliminary results. Figures 3(a) and 3(b) show the results for the Amazon.com subset "Computers & Internet". In 3(c) and 3(d) the results for the synthetic generated web graph are shown.

computational cost. We also aim at providing a mathematical proof for the convergence of the algorithm. Moreover, we plan to extend and explore the algorithm in different scenarios; for instance, we want to test the case that a peer chooses another peer not at random but according to some criteria based on "semantic" relationships between the local interest profiles. We expect that in a semantic overlay network, the PageWeights will converge to the global PageRank scores with even fewer interactions, reducing the number of meetings needed for a good approximation.

7. REFERENCES

[1] http://www.amazon.com/gp/aws/landing.html.

[2] http://www.google.com.

[3] S. Abiteboul, M. Preda, and G. Cobena. Adaptive on-line page importance computation. In *WWW Conf.*, pages 280–290. ACM Press, 2003.

[4] M. Bender, S. Michel, G. Weikum, and C. Zimmer. Bookmark-driven query routing in peer-to-peer web search. In *Workshop on Peer-to-Peer Information Retrieval*, 2004.

[5] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *WWW Conf.*, 1998.

[6] A. Z. Broder, R. Lempel, F. Maghoul, and J. Pedersen. Efficient pagerank approximation via graph aggregation. In *WWW Conf. on Alternate track papers & posters*, pages 484–485. ACM Press, 2004.

[7] E. Buchmann and K. Böhm. Fairnet - how to counter free riding in peer-to-peer data structures., 2004.

[8] J. P. Callan, Z. Lu, and W. B. Croft. Searching distributed collections with inference networks. In *SIGIR*, pages 21–28. ACM Press, 1995.

[9] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. In *SIAM Data Mining*, 2004.

[10] S. Chakrabarti. *Mining the Web: Discovering Knowledge from Hypertext Data*. Morgan-Kauffman, 2002.

[11] Y.-Y. Chen, Q. Gan, and T. Suel. Local methods for estimating pagerank values. In *CIKM*, pages 381–389. ACM Press, 2004.

[12] S. Chien, C. Dwork, S. Kumar, and D. Sivakumar. Towards exploiting link evolution. In *Workshop on Algorithms for the Web*, 2001.

[13] R. Fagin, R. Kumar, and D. Sivakumar. Comparing top k lists. In *SIAM Discrete Algorithms*, 2003.

[14] N. Fuhr. A decision-theoretic approach to database selection in networked ir. *ACM Trans. Inf. Syst.*, 17(3):229–249, 1999.

[15] Z. Gyöngyi, H. Garcia-Molina, and J. Pedersen. Combating web spam with trustrank. In *VLDB*, pages 576–587, 2004.

[16] T. H. Haveliwala. Efficient computation of PageRank. Technical report, Stanford University, 1999.

[17] S. Kamvar, T. Haveliwala, C. Manning, and G. Golub. Exploiting the block structure of the web for computing pagerank. Technical report, Stanford University, 2003.

[18] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. In *SIAM Discrete Algorithms*, pages 668–677, 1998.

[19] A. N. Langville and C. D. Meyer. Deeper inside pagerank. *Internet Mathematics*, 1(3):335–400, 2004.

[20] A. N. Langville and C. D. Meyer. Updating pagerank with iterative aggregation. In *WWW Conf. on Alternate track papers & posters*, pages 392–393. ACM Press, 2004.

[21] N. Ntarmos and P. Triantafillou. Seal: Managing acesses and data in peer-to-peer data sharing networks. In *HDMS*, 2004.

[22] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. Technical Report TR-00-010, Berkeley, CA, 2000.

[23] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM Middleware*, pages 329–350, 2001.

[24] K. Sankaralingam, S. Sethumadhavan, and J. C. Browne. Distributed Pagerank for P2P Systems. In *HPDC*, pages 58–68, June 2003.

[25] S. Shi, J. Yu, G. Yang, and D. Wang. Distributed page ranking in structured p2p networks. In *ICPP*, 2003.

[26] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, pages 149–160. ACM Press, 2001.

[27] Y. Wang and D. J. DeWitt. Computing pagerank in a distributed internet search system. In *VLDB*, 2004.

On the role of composition in XQuery

Christoph Koch
Lehrstuhl für Informationssysteme
Universität des Saarlandes
Saarbrücken, Germany
koch@cs.uni-sb.de

ABSTRACT

Nonrecursive XQuery is known to be hard for nondeterministic exponential time. Thus it is commonly believed that any algorithm for evaluating XQuery has to require exponential amounts of working memory and doubly exponential time in the worst case. In this paper we present a property – the lack of a certain form of composition – that virtually all real-world XQueries have and that allows for query evaluation in singly exponential time and polynomial space. Still, we are able to show for an important special case – our nonrecursive XQuery fragment restricted to atomic value equality – that the composition-free language is just as expressive as the language with composition. Thus, under widely-held complexity-theoretic assumptions, the composition-free language is an exponentially less succinct version of the language with composition.

1. INTRODUCTION

XQuery is the principal data transformation query language for XML. Full XQuery is Turing-complete, but queries without recursion are guaranteed to terminate in straightforward functional implementations of the XQuery language. Recursion in XQuery is rarely used in practice (see also [15]); recursive XML transformations are usually implemented in XSLT. It was shown in [7] that nonrecursive XQuery is NEXPTIME-hard. As a consequence, it is commonly believed that any query evaluation algorithm for nonrecursive XQuery must consume doubly exponential time and exponential space for query evaluation in the worst case (cf. e.g. [6]). This is by an exponential factor worse than the complexity of relational algebra or calculus [11]. The paper [7] also introduced a clean fragment of nonrecursive XQuery called *Core XQuery* that was shown to be the expressive counterpart of nested relational algebra [5] (or similar languages such as complex value algebra without powerset [1] or monad algebra [12]) for XML.

In this paper we present a syntactic property – the lack of a certain form of composition – that virtually all real-world XQueries have and which renders *composition-free* Core XQuery just as hard as relational algebra.

By composition, informally, we refer to the use of data value construction (rather than selection using XPath) anywhere except for the construction part of an XQuery (that is, in a for-let-where-return (FLWR) construct, anywhere except for the return clause). For example, the query

```
<books_2004>
{ for $x in /bib/book where year=2004 return
     <book>
        {$x/title}
        <authors>
           { for $y in $x/author return
               <author> {$y/lastname} </author>
           }
        </authors>
     </book>
}
</books_2004>
```

is a composition-free query (so nesting queries, e.g., FLWR-statements in their return clauses, is not a problem) while

```
<books>
{ let $x := <a>{ for $w in /bib/book
                 return <b> {$w} </b> }</a>
  for $y in $x/b return $y/*
}
</books>
```

is not composition-free because it uses a let-expression that constructs a tree as an intermediate result. The equivalent query

```
<books>
{ for $y in (for $w in /bib/book
             return <b> {$w} </b>)
  return $y/*}
}
</books>
```

is still not composition-free because the "in"-expression of the outer for-loop contains a for-loop. However, there is an equivalent composition-free query,

```
<books>
   { for $w in /bib/book return $w }
</books>
```

The technical contributions of this paper are as follows.

- It is shown that composition-free Core XQuery can be evaluated in polynomial space and thus also in singly exponential time. In fact, composition-free nonrecursive XQuery is PSPACE-complete.

With composition [7]:

	with negation	without negation
deep equality	in EXPSPACE; TA[$2^{O(n)}, O(n)$]-hard	
equality on atomic values	TA[$2^{O(n)}, O(n)$]-complete	NEXPTIME-complete

Without composition:

	with negation	without negation
deep equality	PSPACE-complete	NP-complete
equality on atomic values	PSPACE-complete	NP-complete

Table 1: Summary of results on query/combined complexity of Core XQuery.

- We also show that composition-free nonrecursive XQuery without negation is NP-complete.

- Still, we are able to show for an important special case – equality is restricted to atomic values – that composition-free Core XQuery is just as expressive as Core XQuery with composition. Thus, under the usual complexity-theoretic assumptions, the composition-free language is an exponentially less succinct version of the language with composition.

An overview of the complexity results of this paper – together with the results from [7] is given in Table 1. Since the variables in composition-free XQuery range only over subtrees of the input tree, supporting deep value equality has no influence on the complexity of queries, differently from the case of Core XQuery with composition.

Nonrecursive composition-free XQuery is an important class of queries, and indeed, most practical XQueries belong to this class. (For instance, only a handful of the XML Query Use Case queries [15] employ composition.) Composition-free (Core) XQuery is also popular among implementors of limited prototype XQuery engines, e.g. [8]. Our preliminary expressiveness results show that restricting oneself to implementing composition-free Core XQuery does not cause a loss of generality, at least if equality checking is limited to atomic value equality. The expressiveness result also gives a partial explanation for why practical XQueries tend to be composition-free, as observed above. Writing more succinct queries takes an effort, and apparently does not pay off for many queries.

Note that other functional languages such as monad algebra [12] do not seem to have natural "composition-free" fragments that remain expressive.

A major motivation of this work is to define simple but relevant fragments of XQuery suitable for research prototype implementations and theoretical study (see also [4] for another attempt towards this goal). Indeed, composition-free Core XQuery may allow for special, efficient implementation techniques because all variables only range over nodes in the input tree (never over nodes from intermediate query results).

The structure of this paper is as follows. In Section 2, we introduce a clean fragment of nonrecursive XQuery, *Core XQuery*, that will be the language studied in the remainder of the paper. In Section 3 we introduce composition-free Core XQuery. In Section 4, we prove the PSPACE- and NP-completeness results for the complexity of composition-free Core XQuery. Finally, in Section 5, we prove the expressiveness result that composition-free Core XQuery captures full Core XQuery with "child" and atomic equality.

We assume basic complexity classes such as TC_0, NC_1, LOGSPACE, NP, PSPACE, and NEXPTIME known and refer to [6] for the relevant complexity-theoretic background. By TA[$2^{O(n)}, O(n)$], we denote the class of all problems solvable by alternating Turing machines in linear exponential time with a linear number of alternations (see [6] for definitions). Closure and hardness are under LOGSPACE-reductions for NP, PSPACE, and NEXPTIME and under LOGLIN-reductions (that is, LOGSPACE-reductions whose output is linear) for TA[$2^{O(n)}, O(n)$].

We use the now standard notions of data, query, and combined complexity introduced by Vardi [13].

2. CORE XQUERY

We consider the fragment of XQuery with abstract syntax

$$\begin{array}{rcl} \mathit{query} & ::= & () \mid \langle a\rangle \mathit{query}\langle/a\rangle \mid \mathit{query}\ \mathit{query} \\ & \mid & \mathit{var} \mid \mathit{var}/\mathit{axis} :: \nu \\ & \mid & \text{for } \mathit{var} \text{ in } \mathit{query} \text{ return } \mathit{query} \\ & \mid & \text{if } \mathit{cond} \text{ then } \mathit{query} \\ & \mid & (\text{let } \mathit{var} := \langle a\rangle \mathit{query}\langle/a\rangle)\ \mathit{query} \\ \mathit{cond} & ::= & \mathit{var} = \mathit{var} \mid \mathit{query} \end{array}$$

where a denotes the XML tags, *axis* the XPath axes "child" and "descendant", *var* a set of XQuery variables $\$x$, $\$x_1$, $\$x_2, \ldots, \$y, \$z, \ldots$ with a distinguished *root variable* (which is the unique free variable in the query), and ν a *node test* (either a tag name or "*"). We refer to this fragment as *Core XQuery*, or *XQ* for short.

For simplicity, we will work with pure node-labeled unranked ordered trees, and by atomic values, we will refer to leaves (or equivalently, their labels).

XQuery supports several forms of equality. We will not try to use the same syntax (=, eq, or deep_equal) as in the current standards proposal – it is not clear whether the syntax has stabilized. Throughout this paper, equality is by value (that is, by value as a tree rather than by the *yield* of strings at leaf nodes of the tree). We will write $=_{deep}$ and $=_{atomic}$ for deep and atomic equality, respectively. We will use = for statements that apply to both forms of equality.

Our only other divergence from XQuery syntax is that we assume if-expressions of the form "if ϕ then α" rather than "if ϕ then α else β". Of course, our if-expressions can be considered as a shortcut for "if ϕ then α else ()" and else-branches can be simulated using negation, "if not(ϕ) then β".

We define the semantics of an *XQ* expression α with k free variables using a function $[\![\alpha]\!]_k$ – given in Figure 1 – that takes a k-tuple of trees as input. On input tree t, query Q evaluates to $[\![Q]\!]_1(t)$. The symbol $\uplus$ in Figure 1 denotes list concatenation, l_i the i-th element of list l, $<^t_{doc}$ is the

$$
\begin{array}{rcl}
[\![()]\!]_k(\vec{e}) & := & [\,] \\
[\![\langle a\rangle\alpha\langle/a\rangle]\!]_k(\vec{e}) & := & [\langle a\rangle[\![\alpha]\!]_k(\vec{e})\langle/a\rangle] \\
[\![\alpha\ \beta]\!]_k(\vec{e}) & := & [\![\alpha]\!]_k(\vec{e}) \uplus [\![\beta]\!]_k(\vec{e}) \\
[\![\text{for } \$x_{k+1} \text{ in } \alpha & & \\
\text{return } \beta]\!]_k(\vec{e}) & := & \text{let } l = [\![\alpha]\!]_k(\vec{e}); \\
 & & \text{return } \biguplus_{1\le i\le |l|} [\![\beta]\!]_{k+1}(\vec{e}, l_i) \\
[\![(\text{let } \$x_{k+1} := \alpha)\ \beta]\!]_k(\vec{e}) & := & \text{let } l = [\![\alpha]\!]_k(\vec{e}); \\
 & & \text{return } [\![\beta]\!]_{k+1}(\vec{e}, l_1) \\
[\![\$x_i]\!]_k(t_1,\dots,t_k) & := & [t_i] \\
[\![\$x_i/\chi :: \nu]\!]_k(t_1,\dots,t_k) & := & \text{list of nodes } v \text{ of tree } t_i \text{ s.t.} \\
 & & \chi^{t_i}(root^{t_i}, v) \land \text{lab}_\nu^{t_i}(v) \\
 & & \text{in order } <_{doc}^{t_i} \\
[\![\text{if } \phi \text{ then } \alpha]\!]_k(\vec{e}) & := & \text{if } [\![\phi]\!]_k(\vec{e}) \text{ then } [\![\alpha]\!]_k(\vec{e}) \text{ else } [\,] \\
[\![\$x_i = \$x_j]\!]_k(t_1,\dots,t_k) & := & \text{if } t_i = t_j \text{ then } [\langle yes/\rangle] \text{ else } [\,]
\end{array}
$$

Figure 1: Semantics of Core XQuery.

depth-first left-to-right traversal order through tree t, χ^t is the axis relation χ on t, lab_*^t is true on all nodes of t, and lab_a^t, for a a tag name, is true on those nodes of t labeled a. All *XQ* queries evaluate to lists of nodes. However, we assume that *XQ* variables always bind to single nodes rather than lists; our fragment assures this. This semantics is (observationally) consistent with XQuery as currently undergoing standardization through the W3C [14] restricted to Core XQuery.

In our definition of the syntax of Core XQuery, we have been economical with operators introduced. Since conditions are true iff they evaluate to a nonempty collection,

$$
\begin{array}{rcl}
\text{true} & := & \langle a/\rangle \\
\phi \text{ or } \psi & := & \phi\ \psi \\
\phi \text{ and } \psi & := & \text{if } \phi \text{ then } \psi \\
\text{some \$x in } \alpha \text{ satisfies } \phi & := & \text{for \$x in } \alpha \text{ return } \phi
\end{array}
$$

Using deep equality, we can define negation,

$$\text{not } \phi := \ () =_{deep} \text{if } \phi \text{ then } \langle \text{b}/\rangle\ .$$

Conditions "every \$x in α satisfies ϕ" can be defined using "not" and "some". We will use the shortcut $\langle a/\rangle$ for $\langle a\rangle()\langle/a\rangle$. It is clear that

Proposition 2.1. *Let* $\mathbf{X}$ *be a set of operations and axes.*

- *Each* $XQ[=_{deep}, not, every, \mathbf{X}]$ *query can be translated in LOGLIN into an equivalent* $XQ[=_{deep}, \mathbf{X}]$ *query.*
- *Each* $XQ[and, or, some, \mathbf{X}]$ *query can be translated in LOGLIN into an equivalent* $XQ[\mathbf{X}]$ *query.*

Previous results on XQ

The following results on the complexity and expressive power of *XQ* have been established in [7].

Proposition 2.2 ([7]). *W.r.t. combined complexity,*

- $XQ[=_{deep}$*, all axes*$]$ *is in EXPSPACE;*
- $XQ[=_{atomic}$*, all axes, not*$]$ *is in* $TA[2^{O(n)}, O(n)]$*; and*
- $XQ[=_{atomic}$*, all axes*$]$ *is in NEXPTIME.*

Proposition 2.3 ([7]). *W.r.t. query complexity,*

- $XQ[=_{deep}$*, child*$]$ *is* $TA[2^{O(n)}, O(n)]$*-hard;*
- $XQ[=_{atomic}$*, child, not*$]$ *is* $TA[2^{O(n)}, O(n)]$*-hard; and*
- $XQ[=_{atomic}$*, child*$]$ *is NEXPTIME-hard.*

Proposition 2.4 ([7]). *W.r.t. data complexity,* $XQ[=_{deep}$*, all axes*$]$ *is*

- *LOGSPACE-complete under* NC_1*-reductions if the XML input is given as a DOM tree and*
- *in* TC_0 *if the XML input is given as a character string.*

It has been shown that many query languages for complex values that were developed earlier, such as nested relational algebra [5], complex value algebra without powerset [1], and monad algebra [12], share the same expressive power. Core XQuery is an interesting fragment of XQuery because it captures precisely this degree of expressiveness, which is commonly deemed "right" for nested, deeply structured data.

Proposition 2.5 ([7]). *XQ[=, child] captures – up to data representation issues – monad algebra on lists [12].*

That is, there are fixed mappings "V2T" (from complex values to trees), "T2V" (from trees to complex values), "M2X" (from monad algebra on lists to *XQ*), and "X2M" (from *XQ* to monad algebra on lists), s.t. for *XQ* query Q and tree T,

$$\text{X2M}(Q)(\text{T2V}(T)) = \text{T2V}(Q(T))$$

and for monad algebra query Q and complex value V,

$$\text{M2X}(Q)(\text{V2T}(V)) = \text{V2T}(Q(V)).$$

The "representation issues" are that the mappings "V2T" and "T2V" are needed. However these are independent from the mappings between the queries.

Proposition 2.5 holds for = being either atomic or deep value equality. From Proposition 2.5 it also follows that *XQ*[=, child] is a conservative extension of relational algebra up to representation issues in the spirit of [10], just like monad algebra [12].

3. COMPOSITION-FREE XQ

Composition-free Core XQuery, XQ^-[not], now is the fragment of Core XQuery obtained by the grammar

$$
\begin{array}{rcl}
\mathit{query} & ::= & () \mid \langle a\rangle \mathit{query}\langle/a\rangle \mid \mathit{query}\ \mathit{query} \\
 & \mid & \mathit{var} \mid \mathit{var}/\mathit{axis} :: \nu \\
 & \mid & \text{for } \mathit{var} \text{ in } \mathit{var}/\mathit{axis} :: \nu \text{ return } \mathit{query} \\
 & \mid & \text{if } \mathit{cond} \text{ then } \mathit{query} \\
\mathit{cond} & ::= & \mathit{var} = \mathit{var} \mid \mathit{var} = \langle a/\rangle \mid \text{true} \\
 & \mid & \text{some } \mathit{var} \text{ in } \mathit{var}/\mathit{axis} :: \nu \text{ satisfies } \mathit{cond} \\
 & \mid & \mathit{cond} \text{ and } \mathit{cond} \mid \mathit{cond} \text{ or } \mathit{cond} \\
 & \mid & \text{not } \mathit{cond}
\end{array}
$$

The keyword "every" can again be obtained from "some" and "not". Testing whether condition $\$x/\chi :: \nu$ (where χ is an axis and ν is a node test) can be matched is of course possible as "some $y in $x/\chi :: \nu$ satisfies "$y = $y". *Positive* composition-free Core XQuery XQ^- is again obtained by removing negation "not" from the language.

For our expressiveness proof below, we will use a variant of XQ^- with less syntax, i.e. in which conditions are defined using the usual query operations rather than "some", "and", and "or".

Let $XQ^\sim$ denote the XQ queries

- which do not contain "let"-expressions,
- for which for each expression "for \$x in α return β", α is of the form $\$x/\nu$, and
- which in addition support conditions $\$x = \langle a/\rangle$.

$XQ^\sim$ and XQ^- are expressively equivalent.

PROPOSITION 3.1. *$XQ^\sim = XQ^-$.*

Proof Sketch. $\Rightarrow$: For a mapping from $XQ^\sim$ to XQ^-, we define an appropriate translation function f that we use to rewrite all maximal if-conditions (i.e., conditions of if-expressions that are not subexpressions of if-expressions):

$$\begin{array}{rcl} f(\alpha\ \beta) & := & f(\alpha) \text{ or } f(\beta) \\ f(\text{for \$y in \$x}/\nu \text{ return } \alpha) & := & \text{some \$y in \$x}/\nu \\ & & \text{satisfies f}(\alpha) \\ f(\text{if } \phi \text{ then } \alpha) & := & f(\phi) \text{ and } f(\alpha) \\ f(\text{not } \phi) & := & \text{not } f(\phi) \\ f(\langle a\rangle\alpha\langle/a\rangle) & := & \text{true} \end{array}$$

On all other kinds of expressions, f is the identity.

$\Leftarrow$: For a mapping from XQ^- to $XQ^\sim$, we only need to eliminate "true", "some", "and", and "or" using their definitions from Section 2. □

EXAMPLE 3.2. It is easy to verify that the query

```
<result>
{ for $x in $root/a return
     if not(for $y in $x/b return
               if $y/c then ($y/d $y/e))
     then $x/f }
</result>
```

is $XQ^\sim$. The corresponding XQ^- query is

```
<result>
{ for $x in $root/a return
     if not(some $y in $x/b satisfies
               ($y/c and ($y/d or $y/e)))
     then $x/f }
</result>
```
□

The mappings from the proof of Proposition 3.1 can be implemented efficiently, thus our complexity results established below will hold for both XQ^- and $XQ^\sim$.

4. COMPLEXITY RESULTS FOR XQ$^-$

We now provide our complexity characterization of composition-free Core XQuery.

As announced in the introduction, the query evaluation problem for XQ^- is in polynomial space w.r.t. combined complexity.

PROPOSITION 4.1. *$XQ^-[=_{deep}, \textit{all axes}, \textit{not}]$ is in space $O(|Q| \cdot \log|t|)$, where $|Q|$ is the size of the query and $|t|$ is the size of the data tree.*

Proof Sketch. It is easy to check that by definition of the fragment, XQuery variables always range exclusively over nodes of the input tree. This can be verified by checking the invariant that each variable is introduced using a "for"-statement over a collection defined by an expression $\$x/\nu$, starting at the root node of the input tree.

Thus there is a straightforward algorithm – direct nested-loop based evaluation – for XQ^- queries that only takes memory to store a pointer into the input tree (taking space $\log|t|$) for each of the $O(|Q|)$ variables in the query. □

For the remaining results, we study decision problems and thus Boolean queries. Since valid XML query results have to consist of at least a root node, we say that a Boolean (XQ^-) query $\langle a\rangle\alpha\langle/a\rangle$ returns true iff the root node of the result tree has at least one child.

PROPOSITION 4.2. *$XQ^-[=_{atomic}, \textit{child}, \textit{not}]$ is PSPACE-hard w.r.t. query complexity.*

Proof Sketch. The problem is PSPACE-hard already with respect to query complexity (i.e., when the input tree is fixed). The proof is a minor variation of the standard proof of the PSPACE-hardness of the relational calculus (cf. [11]), and is by reduction from the Quantified Boolean Formula evaluation problem (QBF). We illustrate it with an example, which should be easy to generalize. Consider the QBF $\forall x \exists y (x \Leftrightarrow y)$, which is true. This formula can be phrased as the query

```
⟨a⟩
{ if every $x in $root/* satisfies
          (some $y in $root/* satisfies
               (not $x="t" or $y="t") and
               (not $y="t" or $x="t"))
  then ⟨yes/⟩}
⟨/a⟩
```

over the fixed data tree consisting of a root node with two children, one with string value "t" and the other with string value "f". (Of course, "every \$x in Q satisfies ϕ" is the same as "not(some \$x in Q satisfies not(ϕ))") □

While negation and universal quantification were redundant in $XQ[=_{deep}]$, and excluding them did not reduce the complexity of the language [7], it is interesting to consider the case of XQ^- without negation.

PROPOSITION 4.3. *$XQ^-[=_{deep}, \textit{all axes}]$ is in NP w.r.t. combined complexity.*

Proof Sketch. If the result of the query is to be nonempty, a node has to be written at a certain for-depth k (so the subexpression responsible for the node has up to k free XQuery variables). We can guess the value assignments of these and then check the conditions (this includes axes, node-tests, and if-conditions) along the for-loops up to depth k in polynomial time. (Note that negation only applies to conditions that contain XPath, but no XQuery.) □

PROPOSITION 4.4. *$XQ^{-}[=_{atomic}$, child] is NP-hard w.r.t. query complexity.*

Proof Sketch. This follows immediately from the NP-hardness of conjunctive (relational) queries [2], and a proof can be given e.g. by reduction from 3-Colorability: The fixed data tree consists of a root node and three children, which are labeled "red", "green", and "blue", respectively.

Given a graph $G = (V, E)$ with $V = \{v_1, \ldots, v_m\}$ and

$$E = \{\{v_{i(1,1)}, v_{i(1,2)}\}, \ldots, \{v_{i(n,1)}, v_{i(n,2)}\}\}$$

$(1 \leq i(\cdot,\cdot) \leq m)$, we construct the query

```
<result>
{ for $x_1$ in $root/* return
    ⋱
      for $x_{m-1}$ in $root/* return
        for $x_m$ in $root/* return
        if (not $x_{i(1,1)} =_{atomic} $x_{i(1,2)}$) and ... and
           (not $x_{i(n,1)} =_{atomic} $x_{i(n,2)}$) then
             <yes/>
}
</result>
```

It is easy to verify that indeed this query computes "yes" nodes precisely if G is 3-colorable. Obviously, the query can be computed from G in logarithmic space. □

5. EXPRESSIVENESS OF XQ⁻

In this final section, we show that surprisingly, for an important case (atomic equality and "child" as the only supported axis), composition-free Core XQuery is actually just as expressive as full Core XQuery. This is true even though XQ^{-} is in PSPACE and XQ is hard for TA$[2^{O(n)}, O(n)]$. Thus under commonly-held complexity theoretic assumptions, XQ is exponentially more succinct than XQ^{-}.

We use the shortcut $(\langle a\rangle\alpha\langle/a\rangle)/\chi::\nu$ for $\$x/\chi::\nu$ such that $\$x$ has been defined using "let" as $(\langle a\rangle\alpha\langle/a\rangle)$. Below, "dos" is a shortcut for the "descendant-or-self" axis; it will be redundant because $\$x/\text{dos}::\nu$ is equivalent to

$$(\text{if } \$x/\text{self}::\nu \text{ then } \$x)\ \$x//\nu.$$

LEMMA 5.1. *Let a be a label, χ an axis, ν a nodetest, and α an $XQ^{\sim}[=_{atomic}$, child, descendant, self, dos, not] expression. Then there is an $XQ^{\sim}[=_{atomic}$, child, descendant, self, dos, not] expression equivalent to $(\langle a\rangle\alpha\langle/a\rangle)/\chi::\nu$.*

Proof Sketch. Rules to rewrite each such expression

$$(\langle a\rangle\alpha\langle/a\rangle)/\chi::\nu$$

into an equivalent $XQ^{\sim}[=_{atomic}$, child, descendant, self, not] expression are easy to specify:

$$\begin{array}{rcl}
(\langle a\rangle\ \alpha\ \langle/a\rangle)/\nu & \vdash & \alpha/\text{self}::\nu \\
(\langle a\rangle\ \alpha\ \langle/a\rangle)/\text{self}::\text{b} & \vdash & () \\
(\langle a\rangle\ \alpha\ \langle/a\rangle)/\text{self}::\text{a} & \vdash & \langle a\rangle\ \alpha\ \langle/a\rangle \\
(\langle b\rangle\ \alpha\ \langle/b\rangle)/\text{self}::* & \vdash & \langle b\rangle\ \alpha\ \langle/b\rangle \\
(\langle a\rangle\ \alpha\ \langle/a\rangle)//\nu & \vdash & \alpha/\text{dos}::\nu \\
(\langle a\rangle\ \alpha\ \langle/a\rangle)/\text{dos}::* & \vdash & \langle a\rangle\ \alpha\ \langle/a\rangle\ (\alpha//*) \\
(\langle a\rangle\ \alpha\ \langle/a\rangle)/\text{dos}::a & \vdash & \langle a\rangle\ \alpha\ \langle/a\rangle\ (\alpha//a) \\
(\langle a\rangle\ \alpha\ \langle/a\rangle)/\text{dos}::b & \vdash & \alpha//b \\
()/\chi::\nu & \vdash & () \\
(\alpha\ \beta)/\chi::\nu & \vdash & (\alpha/\chi::\nu)\ (\beta/\chi::\nu) \\
(\text{for \$x in } \alpha \text{ return } \beta)/\chi::\nu & \vdash & \text{for \$x in } \alpha \\
 & & \text{return } (\beta/\chi::\nu) \\
(\text{if } \phi \text{ then } \alpha)/\chi::\nu & \vdash & \text{if } \phi \text{ then } (\alpha/\chi::\nu) \\
(\$x/\chi::\nu)/\chi'::\nu' & \vdash & \text{for } \$y \text{ in } \$x/\chi::\nu \\
 & & \text{return } \$y/\chi'::\nu'
\end{array}$$

(Note that $(\$x/\chi::\nu)/\chi'::\nu'$ in the final rule is really equivalent to the for-expression on the right-hand side of that rule, and is in general not equivalent to $\$x/\chi::\nu/\chi'::\nu'$, as the former may produce duplicates if both χ and χ' are "descendant".) □

THEOREM 5.2. *$XQ^{\sim}[=_{atomic}$, child, desc, self, not] captures the $XQ[=_{atomic}$, child, desc, self, not] queries.*

Proof Sketch. We first replace each expression of the form "(let $\$x := \langle a\rangle\alpha\langle/a\rangle$) β" by an expression $\beta' := \beta[\$x \Rightarrow \langle a\rangle\alpha\langle/a\rangle]$ obtained by substituting each occurrence of variable $\$x$ in β by $\langle a\rangle\alpha\langle/a\rangle$.

We now need to consider where such a replacement of a variable $\$x$ by an expression $\langle a\rangle\alpha\langle/a\rangle$ can occur:

1. Inside an equality $\$x =_{atomic} \alpha$ (with α either a variable or a constant $\langle b/\rangle$).

   To rewrite $\$x$ with $\langle a\rangle\alpha\langle/a\rangle$, we may assume that α is (); otherwise, we could not type $\langle a\rangle\alpha\langle/a\rangle$ to be an atomic value. Thus we obtain $\langle a/\rangle =_{atomic} \alpha$, which is $XQ^{\sim}$. Conditions $\langle a/\rangle =_{atomic} \langle a/\rangle$ and $\langle a/\rangle =_{atomic} \langle b/\rangle$ are rewritten into "true" and "not(true)", respectively.

2. Inside an expression $\$x$ or $\$x/\chi::\nu$ (either in the "in"-expression of a for-loop or as an expression constructing "output").

   Here rewriting may lead to expressions of the form $(\langle a\rangle\alpha\langle/a\rangle)/\chi::\nu$, which is not XQ syntax. We can eliminate such expressions using Lemma 5.1.

Now the query obtained is already an $XQ^{\sim}$ query if in all expressions "for $\$x$ in α return β", α is of the form $\$z$ or $\$z/\chi::\nu$. Otherwise, we apply the rewrite rules from Figure 2. This may again produce expressions $(\langle a\rangle\alpha\langle/a\rangle)/\chi::\nu$, by rule (2). We eliminate such cases again using Lemma 5.1.

It can be verified that the rewrite system thus specified indeed maps any $XQ[=_{atomic}$, child, desc, self, not] query to an equivalent $XQ^{\sim}[=_{atomic}$, child, desc, self, not] query. An example mapping to $XQ^{\sim}$ illustrating our rewrite system is given in Figure 3. □

Acknowledgments

I thank Dan Olteanu and Stefanie Scherzinger for their comments on an earlier version of the paper.

6. REFERENCES

[1] S. Abiteboul and C. Beeri. "The Power of Languages for the Manipulation of Complex Values". *VLDB J.*, **4**(4):727–794, 1995.

[2] A. K. Chandra and P. M. Merlin. "Optimal Implementation of Conjunctive Queries in Relational Data Bases". In *Conference Record of the Ninth Annual ACM Symposium on Theory of Computing (STOC'77)*, pages 77–90, Boulder, CO, USA, May 1977.

$$
\begin{array}{rcll}
\text{for \$x in () return } \alpha & \vdash & () & (1)\\
\text{for \$x in } (\langle a\rangle\ \alpha\ \langle /a\rangle) \text{ return } \beta & \vdash & \beta[\$x \Rightarrow (\langle a\rangle\ \alpha\ \langle /a\rangle)] & (2)\\
\text{for \$x in } (\alpha\ \beta) \text{ return } \gamma & \vdash & (\text{for \$x in } \alpha \text{ return } \gamma)\ (\text{for \$x in } \beta \text{ return } \gamma) & (3)\\
\text{for \$y in (for \$x in } \alpha \text{ return } \beta) \text{ return } \gamma & \vdash & \text{for \$x in } \alpha \text{ return for \$y in } \beta \text{ return } \gamma & (4)\\
\text{for \$x in (if } \phi \text{ then } \alpha) \text{ return } \beta & \vdash & \text{for \$x in } \alpha \text{ return if } \phi \text{ then } \beta & (5)\\
\text{for \$y in \$x return } \alpha & \vdash & \alpha[\$y \Rightarrow \$x] & (6)
\end{array}
$$

Figure 2: Rewrite rules for translating for-expressions to $XQ^{\sim}$.

$$
\begin{array}{rc}
(\text{let } \$x := \langle a\rangle\{ \text{ for } \$w \text{ in \$root/* return } \langle b\rangle\{\$w\}\langle /b\rangle\ \}\langle /a\rangle) \text{ for } \$y \text{ in } \$x/b \text{ return } \$y/* & \overset{elim.let}{\vdash}\\
\text{for } \$y \text{ in } (\langle a\rangle\{ \text{ for } \$w \text{ in \$root/* return } (\langle b\rangle\{\$w\}\langle /b\rangle)\ \}\langle /a\rangle)/b \text{ return } \$y/* & \overset{Lem.\ 5.1}{\vdash}\\
\text{for } \$y \text{ in (for } \$w \text{ in \$root/* return } (\langle b\rangle\{\$w\}\langle /b\rangle)) \text{ return } \$y/* & \overset{4}{\vdash}\\
\text{for } \$w \text{ in \$root/* return for } \$y \text{ in } (\langle b\rangle\{\$w\}\langle /b\rangle) \text{ return } \$y/* & \overset{2}{\vdash}\\
\text{for } \$w \text{ in \$root/* return } (\langle b\rangle\{\$w\}\langle /b\rangle)/* & \overset{Lem.\ 5.1}{\vdash}\\
\text{for } \$w \text{ in \$root/* return } \$w &
\end{array}
$$

Figure 3: Example rewriting.

[3] G. Gottlob, C. Koch, and R. Pichler. "Efficient Algorithms for Processing XPath Queries". In *Proc. VLDB 2002*, pages 95–106, Hong Kong, China, 2002.

[4] J. Hidders, J. Paredaens, R. Verkammen, and S. Demeyer. "A Light but Formal Introduction to XQuery". In *Proc. XSYM*, pages 5–20, 2004.

[5] G. Jaeschke and H.-J. Schek. "Remarks on the Algebra of Non First Normal Form Relations". In *Proc. PODS'82*, pages 124–138, 1982.

[6] D. S. Johnson. "A Catalog of Complexity Classes". In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume 1, chapter 2, pages 67–161. Elsevier Science Publishers B.V., 1990.

[7] C. Koch. "On the Complexity of Nonrecursive XQuery and Functional Query Languages on Complex Values". In *Proc. PODS'05*, 2005.

[8] C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier. "Schema-based Scheduling of Event Processors and Buffer Minimization for Queries on Structured Data Streams". In *Proc. VLDB 2004*, Toronto, Canada, 2004.

[9] A. Marian and J. Siméon. "Projecting XML Documents". In *Proc. VLDB 2003*, pages 213–224, 2003.

[10] J. Paredaens and D. Van Gucht. "Possibilities and Limitations of Using Flat Operators in Nested Algebra Expressions". In *Proc. PODS*, pages 29–38, 1988.

[11] L. J. Stockmeyer. *The Complexity of Decision Problems in Automata Theory.* PhD thesis, Dept. Electrical Engineering, MIT, Cambridge, Mass., USA, 1974.

[12] V. Tannen, P. Buneman, and L. Wong. "Naturally Embedded Query Languages". In *Proc. of the 4th International Conference on Database Theory (ICDT)*, pages 140–154, 1992.

[13] M. Y. Vardi. "The Complexity of Relational Query Languages". In *Proc. 14th Annual ACM Symposium on Theory of Computing (STOC'82)*, pages 137–146, San Francisco, CA USA, May 1982.

[14] World Wide Web Consortium. "XQuery 1.0 and XPath 2.0 Formal Semantics. W3C Working Draft (Aug. 16th 2002), 2002. `http://www.w3.org/TR/query-algebra/`.

[15] "XML Query Use Cases. W3C Working Draft 02 May 2003", 2003. http://www.w3.org/TR/xmlquery-use-cases/.

An Empirical Evaluation of Simple DTD-Conscious Compression Techniques

James Cheney
University of Edinburgh
Edinburgh, United Kingdom
jcheney@inf.ed.ac.uk

1. INTRODUCTION

The term "XML compression" has been used to describe techniques addressing several different (though related) problems, all relevant to Web data management:

1. *minimum-length coding for efficient XML document storage and transmission* [13, 5, 10, 1];
2. *compact binary formats for efficient (streaming) XML message processing and transmission* [8, 9]; and
3. *storage techniques for efficient XML database query processing* [11, 17, 3, 2].

To avoid ambiguity, in this paper, the term "XML compression" is used in the first (and, we believe, original and most accurate) sense exclusively. We will compare our proposed techniques only with other approaches that address problem (1), not problems (2) or (3).

Since XML markup often displays a high degree of redundancy, ordinary text compressors (`gzip` [7], `bzip2` [15], etc.) are frequently used for XML storage and transmission. Text compressors perform adequately for archiving XML files in many situations; however, they are blind to the underlying structure of the XML document so may miss compression opportunities. Because of this, researchers have studied, and companies have marketed, XML compression tools.

In previous work [5], we developed a streaming XML-conscious compressor `xmlppm`, and showed that it provides compression superior to other contemporary text and XML-conscious compression techniques (including XMill [13]).

The purpose of this paper is to investigate whether DTD information can be used to improve compression in `xmlppm` enough to justify the added implementation effort. We consider the *minimum-length coding problem for valid XML*: Given a data source producing XML conforming to a DTD, find the smallest possible encoding. We assume both sender and receiver have access to identical copies of the DTD.

It appears to be common sense that a DTD, XML Schema, or RELAX/NG schema should be useful in helping to compress conforming XML documents, and many of the above approaches exploit or require an XML Schema. However, we are aware of no rigorous experimental validation of DTD or schema-conscious XML compression in comparison with competitive DTD-unconscious XML compression techniques.

We choose to focus on DTDs exclusively (rather than XML Schema or RELAX/NG schemas) for several reasons: DTDs are simpler, more established, and more widely adopted; DTD parsing is built-in to most XML parsers; DTD validation is easier to implement than for the other approaches; and substantial compression improvements turn out to be possible using DTDs only. Nevertheless, XML Schema and RELAX/NG schemas can provide much more detailed information about documents, especially about their text content. We view generalizing our results to more powerful schema systems as an important future direction.

Because `xmlppm` already compresses both XML structure and text very well, and because of the complexity of the underlying PPM algorithms, it is easy to generate ideas for DTD-based compression that work well "on paper" but are either incompatible with `xmlppm` or do not improve compression relative to `xmlppm`. In this paper we describe `dtdppm`, a version of `xmlppm` that simultaneously validates and compresses XML relative to a DTD. Our main contribution is the development of four simple DTD-based optimizations that techniques that *do* work well with `xmlppm`: *ignorable whitespace stripping, symbol table reuse, element symbol prediction*, and *bitmap-based attribute list coding.* These simple techniques are validated by experiments showing substantial compression benefits for a variety of real data sources.

The structure of the rest of the paper is as follows. Section 2 reviews `xmlppm`. Section 3 presents the DTD-conscious compression techniques used in `dtdppm`. Section 4 presents experimental results, and Section 5 discusses the results. Section 6 concludes.

2. BACKGROUND

In previous work, we developed `xmlppm` [5, 4], an algorithm for XML compression based on Prediction by Partial Match (PPM) [6], one of the most advanced known text compression techniques. PPM compression builds a statistical model of the data seen so far, and uses it to generate a probability distribution predicting the next symbol; the actual symbol is transmitted using arithmetic coding relative to this distribution. In `xmlppm`, an XML file is first parsed using a SAX parser to generate a stream of SAX events. Each event is encoded using a bytecode representation called ESAX. The ESAX bytecodes are encoded using one of several "multiplexed" PPM compressors, for elements, charac-

Eighth International Workshop on the Web and Databases (WebDB 2005)
June 16–17, 2005, Baltimore, Maryland, USA

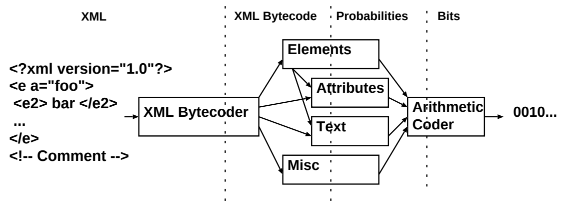


Figure 1: `xmlppm` architecture

ters, attributes, and miscellaneous symbols. The encoder and decoder states are in lockstep so that the decoder always knows which model to use for the next symbol. The architecture of `xmlppm` is shown in Figure 1.

This XML-conscious, multiplexed modeling approach offers several benefits over simply using PPM compression directly on XML text. First, `xmlppm` tokenizes element and attribute names, so less time is spent on low-level PPM compression. Second, text, element, and attribute content in XML have different statistical characteristics, so using different models for each kind of data improves compression. Third, `xmlppm` uses its knowledge of the structure of XML documents to influence the underlying statistical models and improve compression. Specifically, the element, attribute and text PPM models are given "hints" about the surrounding XML element context. (The paper [5] gives full details.)

PPM techniques are among the most advanced known text compression techniques, and many XML documents consist primarily of unstructured text, so it is not surprising that using PPM leads to improved XML compression. However, this level of performance comes at a cost. In the original reported experiments [5], the implementation of `xmlppm` was very slow (ranging from 5–40 times slower than `bzip2`). Since then, the speed of `xmlppm` has been improved considerably by incorporating Shkarin's highly optimized PPMII implementation of PPM [16]. The most recent version of `xmlppm` [4] is still about a factor of 5–10 slower than `gzip`, but is generally as fast as or faster than `bzip2`, while providing better compression.

Table 1 compares `gzip`, `bzip2`, the current implementation of `xmlppm`, and `dtdppm` on the corpus used in [5]. The experimental setup is described in Section 4. The XML column lists the size of the input files in bytes; the other columns measure compression rate in bits per input character (bpc) for each compressor. Times are in seconds. The final column % shows the percentage change in compression rate or execution time for `dtdppm` vs. `xmlppm` (that is, $((X - D)/X)$%, where X is the rate/time for `xmlppm` and D the rate/time for `dtdppm`). The results are discussed further in Section 4.1.

3. COMPRESSION TECHNIQUES

We modified the current version of `xmlppm` to read in and validate its input relative to a DTD: the resulting DTD-conscious compressor is called `dtdppm`. In `dtdppm`, the DTD validation state is available for use during compression, so DTD-specific optimizations are possible. We have implemented four DTD-based optimizations: ignorable whitespace stripping, symbol table reuse, element symbol prediction, and bitmap-based attribute list coding.

	XML (bytes)	`gzip` (bpc)	`bzip2` (bpc)	`xmlppm` (bpc)	`dtdppm` (bpc)	change %
elts	113181	0.622	0.416	0.375	0.327	13%
pcc1	51647	0.557	0.444	0.301	0.229	24%
pcc2	262669	0.320	0.183	0.145	0.123	15%
pcc3	186857	0.374	0.227	0.167	0.144	14%
play1	251898	2.160	1.549	1.449	1.453	−0.21%
play2	136841	2.141	1.630	1.473	1.464	0.62%
play3	279703	2.267	1.647	1.569	1.566	0.18%
sprot	10268	2.056	2.048	1.692	1.430	16%
stats1	669347	0.798	0.369	0.294	0.282	4.2%
stats2	616094	0.750	0.338	0.272	0.258	5.3%
tal1	734590	0.313	0.123	0.117	0.102	13%
tal2	510111	0.322	0.151	0.127	0.107	16%
tal3	251698	0.330	0.198	0.152	0.125	18%
tpc	287992	1.476	1.101	1.007	0.980	2.6%
tree	6704	1.734	1.494	1.104	0.715	35%
w3c1	220794	1.888	1.464	1.302	1.275	2.0%
w3c2	196233	1.947	1.534	1.365	1.338	2.0%
w3c3	201849	2.139	1.722	1.530	1.471	3.8%
w3c4	104938	1.804	1.521	1.333	1.280	4.0%
w3c5	247465	1.823	1.435	1.336	1.287	3.7%
weblog	2304	2.160	2.559	1.747	1.420	18%
total	5343183	1.035	0.701	0.625	0.603	3.6%
time(s)		0.61	2.38	1.89	2.50	−32%

Table 1: XMLPPM corpus

3.1 Ignorable whitespace stripping

Much of the whitespace in an XML document is irrelevant to the data being represented: for example, whitespace is often used only as a visual cue to the document's hierarchical structure. We call this whitespace *ignorable*. Our implementation attempts to drop ignorable whitespace whenever possible. PPM algorithms typically use up to 10 of the most recent characters as context to predict the next symbol. Compressing long sequences of whitespace flushes the context, so the model is unprepared for whatever comes next. Therefore, though it may seem trivial, whitespace stripping is crucial for good PPM compression performance because it helps prevent PPM models from losing track of context.

It is not generally safe to drop whitespace in the absence of a DTD, since whitespace is significant in some elements (e.g., `<xsl:text>`). However, in the presence of a DTD, whitespace can usually be safely ignored whenever it occurs inside an element whose content model does not mention `#PCDATA`. `dtdppm` tests for this whenever character data is encountered, and ignorable whitespace is dropped. When documents with ignored whitespace are decompressed, `dtdppm` optionally inserts newlines and indentation so that the resulting document will be human-readable rather than one long line.

Nevertheless, there may be documents with whitespace that is ignorable by our definition but which users wish to preserve, so whitespace stripping is optional.

3.2 Symbol table reuse

In `xmlppm`, element and attribute tags (and some other kinds of symbols) are replaced by symbol table references. However, in the absence of a DTD, the symbol table needs to be built dynamically by the encoder and decoder, so the text for each symbol is sent when it is first encountered in the document. In `dtdppm`, the DTD is available to both encoder and decoder, and it is not necessary to transmit the symbols inline. Instead, both encoder and decoder can refer to a symbol table built from the DTD.

The savings from this optimization are not dramatic, since the DTD may be much smaller then the document; however, symbol table reuse is important in a situation in which many small documents are to be compressed, especially since ordinary compression techniques are less effective for smaller documents.

3.3 Element symbol prediction

In the absence of a DTD, any element tag can, in principle, occur anywhere in the document. However, in valid XML, elements may have regular expression content models

```
<!ELEMENT foo (bar,(baz|bar)*)>
```

that constrain the children of the element. For highly structured data, frequently there is only one possible next element symbol; this can be determined by inspecting the state of the DTD validator. When this is the case, the `dtdppm` encoder omits the element bytecode since the decoder can infer the next element symbol from context. Similarly, it is possible to test whether the remaining content model is empty. In this case, the "end-element" bytecode that would ordinarily be sent is omitted because the decoder can infer it from context.

This technique may seem trivial since it does not do anything special other than to omit symbols that can be predicted from context. However, more sophisticated techniques seem to interact badly with PPM. For example, in approaches like those of Levene and Wood [12] or XComprez [10], element content sequences are encoded in a more sophisticated way that is dependent on the remaining regular expression content model. We experimented with a simple form of this approach in `dtdppm`, but found that it does not help much relative to ordinary `xmlppm` compression. This is because PPM compresses byte-aligned text, so using non-byte-aligned encodings for element symbols confuses the underlying PPM model. Nevertheless, this is definitely an area where improvement may be possible. Combining sophisticated regular expression coding techniques with PPM compression is a challenge left for future work.

3.4 Bitmap-based attribute list encoding

Attribute list declarations

```
<!ATTLIST elt att1 TYPE1 DFLT1 att2 TYPE2 DFLT2 ...>
```

are one of the most complicated features of DTDs. Attribute values can have one of several types `TYPE`:

`CDATA`	Arbitrary text
`ID`	Globally unique, can't be `FIXED`
`IDREF(S)`	Must refer to an `ID`
`NMTOKEN(S)`	Must be a name token
`ENTITY(IES)`	Must be a declared `ENTITY`
`NOTATION` $(v_1\|\cdots\|v_n)$	Enumerated, declared `NOTATION`
$(v_1\|\cdots\|v_n)$	Enumerated type

Attribute types `IDREF`, `NMTOKEN`, and `ENTITY` can be plural. Attributes can also have several default specifications `DFLT`:

`"dflt"`	Default value is `"dflt"`
`#FIXED "dflt"`	Must be present and equal `"dflt"`
`#REQUIRED`	Value must be present
`#IMPLIED`	May be absent, no default

In XML, the order of attribute-value pairs is irrelevant; thus, we may rearrange the attribute list if doing so improves compression. In `dtdppm`, we encode attribute lists by sending a (byte-aligned) bitmap indicating which attributes are present, then sending the attribute values. Default and type constraints are used to avoid sending redundant information.

The details of the encoding are as follows. First, the attribute list is scanned in order to build a bit vector. This bit vector says which attribute values are going to be sent next. `#FIXED` and `#REQUIRED` attributes are not included, since they must be present in a valid document. For `#IMPLIED` attributes, 1 indicates present, 0 absent. For attributes with a default value, 1 indicates that the value is non-default, 0 otherwise. Subsequently, the values of `#REQUIRED` attributes and other attributes whose bitmap value is 1 are transmitted. The values of attributes with enumerated types $(v_0|\cdots|v_n)$ are encoded as bytecodes $0,\ldots,n$.

For example, given

```
<!ATTLIST elt att1 CDATA #FIXED "foo"
              att2 (x|y|z) #REQUIRED
              att3 CDATA #IMPLIED
              att4 NMTOKEN "bar">
```

the encoding of the attribute list of

```
<elt att1="foo" att2="y" att4="baz">
```

is `40 01 'b' 'a' 'z' 00`. The top two bits of the first byte ($40_{16} = 01000000_2$) code the absence of `att3` and (non-default) presence of `att4`; `01` codes enumerated value `y`; and the non-default value `baz` of `att4` is transmitted as a null-terminated string.

There are many possible variations on this theme. We initially tried two simpler ideas, based on adjacency lists and vector representations of attribute lists, each of which worked well for some examples but not for others; the bitmap approach combines the advantages of the two approaches.

4. EVALUATION

The design of a corpus for testing compression techniques can be a subtle issue, because of the possibility of accidental bias towards one or another kind of data. So far, no standard corpus for XML compression (let alone DTD/schema-conscious compression) has emerged. Desirable characteristics of test data include that the data (and DTDs) be freely available online, that there are nontrivial amounts of data (whole documents instead of short examples), that the data is actually valid relative to the DTD, and finally, that the data be "realistic" (i.e., not random or arbitrary). Obviously many of these criteria are subjective. Unfortunately, it is not easy to find data sources having all these characteristics.

We have evaluated `dtdppm` on five corpora:

- The XMLPPM corpus [5]
- Short documents (NewsML)
- Medium structured application data (MusicXML)
- Medium flat datasets (UW XML repository)
- Large datasets (DBLP, Medline, PSD, XMark)

We are aware of other collections of valid XML, such as the Niagara experimental data[1] but have not had time to experiment with these other sources.

[1] `http://www.cs.wisc.edu/niagara/data.html`

Our experiments were performed on an AMD Athlon 64 3000+ (1.8Ghz clock speed) with 512MB RAM, running Red Hat Fedora Core 3. We report compression in bits per character (relative to the original XML input) and total compression time for `gzip`, `bzip2`, `xmlppm`, and `dtdppm` for each data source. (For PPM techniques decompression takes the same time as compression.) The PPM models used by `xmlppm` and `dtdppm` are order 5 models with 1MB of working memory per model (for a total of 4MB). We also benchmarked the current version of XMill[2], and found that, as in [5], it compresses no better than `bzip2` but runs up to three times faster. Because of limited space, these results are omitted.

In addition, we compared the effectiveness of the individual compression techniques, and found that no single technique was dominant. Space limits preclude a full discussion.

4.1 The XMLPPM corpus

For comparison with previous work, we evaluated the performance of `dtdppm` using the same data[3] used by [5]. This corpus contains XML files ranging from small (1KB) to large (700KB), and including both highly textual data and highly structured data. DTDs for each file were either constructed by hand or obtained online. Some errors and inaccuracies in existing DTDs were corrected.

The realism of this benchmark is debatable; its chief virtue is variety. Also, results for this benchmark may be skewed since we constructed some of the DTDs ourselves (with compression in mind), rather than using given DTDs.

Nevertheless, the results (Table 1) do indicate that DTD-conscious compression can be worthwhile for a variety of kinds of XML. In particular, small XML fragments (`sprot`, from SwissPROT; `tree`, from Penn Treebank; and `weblog`, a web log excerpt) exhibit substantial improvements of 16–35%, and large, highly-structured files (`elts`, periodic table data; `pcc1-3`, formal proofs; `tal1-3`, typed assembly language files) improve 13–24%. On the other hand, examples with a lot of text or very regular structure (`play1-3`, Shakespeare plays; `stats1-2`, baseball statistics; `tpc`, TPC benchmark data; `w3c1-5`, W3C standards) did not compress significantly better (0-5% improvement); one example compresses 0.14% worse. For these examples, `xmlppm` already compresses regular structure well and the DTDs provide no information that would help improve text compression. As with most of our examples, `dtdppm` ran slightly slower than `xmlppm`.

4.2 Short documents

NewsML[4] is an XML dialect designed for news articles from press services (e.g. Reuters). The 80KB NewsML DTD defines a NewsML document as some metadata and uses XHTML for the article content (another 56KB). We obtained the DTD and a collection of 246 example NewsML articles, ranging from 6.5–18.2KB (average size 11.2KB). The compression results for the NewsML data are summarized in Table 2. The "NewsML" line shows the compression rates over the entire corpus; the "time" line shows the total compression time.

These results suggest that NewsML documents benefit substantially from DTD-conscious compression, largely, we believe, due to symbol table reuse. Both `xmlppm` and `dtdppm` are considerably slower than `bzip2` in this case; reparsing the 136KB of DTD files accounts for roughly 55% of `dtdppm` running time. This overhead could be alleviated by specializing the compressor to the DTD.

[2] `http://www.cs.washington.edu/homes/suciu/XMILL/`

[3] available at `http://xmlppm.sourceforge.net/`

[4] `http://www.newsml.org`

	`gzip` (bpc)	`bzip2` (bpc)	`xmlppm` (bpc)	`dtdppm` (bpc)	change %
NewsML	2.292	2.241	1.982	1.484	25%
time(s)	0.38	2.19	1.54	6.11	−300%
MusicXML	0.304	0.216	0.223	0.127	43%
time(s)	0.10	1.78	0.57	0.77	−35%

Table 2: NewsML and MusicXML results

4.3 Medium structured application data

XML is becoming a widespread format for storing application data: for example, recent versions of popular office suites either store application data as XML directly, or offer the ability to export data in XML. However, standard DTDs for such data are not always available, stable, or heeded.

MusicXML[5] is an XML dialect for representing music. MusicXML documents can be translated to a sheet music PDF file of either all parts or a single part, as well as to a MIDI file that can be played directly on a synthesizer or further processed using sequencing software. We obtained the MusicXML DTD files (106KB total) and 18 example MusicXML files ranging from 8.8–230KB (101KB average), each corresponding to one or two sheets of a musical score. The compression results for MusicXML are shown in Table 2. The best compression is obtained by `dtdppm`; the average improvement is 43%. Note that plain `xmlppm` generally compresses MusicXML slightly worse than `bzip2`, but both `xmlppm` and `dtdppm` are slightly faster.

The MusicXML web page claims that `gzip`-compressed MusicXML documents are only about twice as large as equivalent documents in MuseData, a custom format. Since `dtdppm` compresses MusicXML 58% better than `gzip` on average, this suggests `dtdppm` is competitive with a hand-coded binary format.

4.4 Medium flat datasets

XML is sometimes used to export, or *publish*, the data in a relational table or database, often with some added structure. The UW XML repository[6] includes several example XML data sources, many of which consist of a flat sequence of elements with identical structure. Unfortunately, many of these examples do not possess DTDs, or are not valid. We chose several medium-sized examples that do have DTDs and are valid to evaluate `dtdppm` for such data.

This situation seems to offer great promise, since the DTD tells us almost everything we need to know about the structure of the data: only a few details (such as the number of rows) need to be filled in. However, data with very regular structure already compresses very well using plain XML-conscious compression techniques, because the DTD only tells the compressor things it learns quickly for itself. As a result, the amount of improvement that can be expected for such data is limited. On the other hand, many of the medium-sized files make liberal use of whitespace for readability. As a result, some improvement to compression re-

[5] `http://www.recordare.com/xml.html`

[6] `http://www.cs.washington.edu/research/xmldatasets/`

	XML (bytes)	gzip (bpc)	bzip2 (bpc)	xmlppm (bpc)	dtdppm (bpc)	change %
321gone	24442	2.213	2.228	1.884	1.741	7.5%
cornell	30979	1.026	0.954	0.880	0.732	17%
ebay	35472	2.480	2.580	2.189	2.103	3.9%
reed	283582	0.533	0.332	0.327	0.274	16%
SigRec	478337	1.363	0.812	0.802	0.745	7.1%
ubid	20246	1.494	1.515	1.296	1.114	14%
wash	3068693	0.525	0.317	0.390	0.275	30%
yahoo	25347	1.971	1.903	1.620	1.470	9.3%
total	3967098	0.673	0.431	0.477	0.372	22%
time(s)		0.28	3.21	1.17	1.65	−41%

Table 3: Medium flat file benchmark results

sulting from whitespace stripping is to be expected.

The experimental results are shown in Table 3. In a few examples (reed, wash, course information; cornell, personnel records; ubid, auction data), dtdppm achieves a substantial improvement because of whitespace stripping, improving compression substantially (14–30%). For the other examples (321gone, ebay, yahoo, auction data; SigRec, bibliographic records) improvement was in the more modest 4–9% range. Overall compression improved 22% relative to xmlppm (mostly because of wash). For this dataset, bzip2 compressed better than xmlppm, but dtdppm performed best overall. Perhaps surprisingly, both xmlppm and dtdppm were 2–2.5 times as fast as bzip2.

4.5 Large datasets

Another increasingly common scenario is the use of XML as a format for serializing large databases. Examples include scientific databases like SwissPROT/UniPROT and the Georgetown Protein Sequence Database and bibliographic databases like DBLP and Medline. These databases are typically made available on the Web and updated at intervals ranging from daily to yearly. Because of their size, scalable and effective compression is very important.

Another large dataset example is the data generated by XMark. The XMark benchmark [14] has been proposed as a means for comparing the performance of XML databases. It consists of a DTD for auction data and a data generator which generates a random valid document of size proportional to a given "scaling factor".

In Table 4, we present the results of compressing four large datasets: xmark is an example XMark file[7], medline is one part of the PubMed database[8], psd is a file from the Georgetown Protein Sequence Database, and dblp is an XML serialization of the DBLP database. The psd and dblp examples were obtained from the UW XML repository.

The results vary. For xmark, dtdppm and xmlppm compress 7.7% worse than bzip2. This is the only example in the paper for which dtdppm is not competitive (i.e., within 1% of the best). For other examples, dtdppm's compression is competitive or best. However, xmark data may not be a realistic compression benchmark because it is randomly generated. The DTDs for these documents do not provide many opportunities to predict a unique next symbol, so the behavior of dtdppm is essentially the same as xmlppm. Also, all of these documents use whitespace only trivially (i.e., each element tag is on its own line, but there is no indenting), so

[7]http://www.xml-benchmark.org

[8]http://www.ncbi.nlm.nih.gov/

	XML (bytes)	gzip (bpc)	bzip2 (bpc)	xmlppm (bpc)	dtdppm (bpc)	change %
xmark	116MB	2.616	1.754	1.888	1.889	−0.02%
time(s)		13.4	46.3	39.1	39.5	−0.8%
medline	127MB	1.278	0.888	0.841	0.838	0.4%
time		7.5	65.1	32.3	33.0	−2.2%
psd	717MB	1.209	0.857	0.867	0.846	2.5%
time		33.9	389.7	169.3	170.0	−0.4%
dblp	103MB	1.479	0.963	0.940	0.947	−0.8%
time(s)		6.9	48.8	27.3	28.2	−3.2%

Table 4: Large benchmark results

whitespace stripping has little effect. Compression time is also similar to xmlppm in most cases, although interestingly dtdppm and xmlppm are generally 15–60% faster than bzip2.

5. DISCUSSION

There are several lessons that can be learned from our experiments. DTD-conscious compression (as embodied in dtdppm) is very effective for small messages, highly-structured documents, or documents with large amounts of formatting whitespace. For large datasets, dtdppm does not compress significantly better than xmlppm; however, both xmlppm and dtdppm compress as well as or better than bzip2 but 30-50% faster. xmlppm will probably never be as fast as gzip, but xmlppm and dtdppm compress significantly better than gzip while staying within an order of magnitude of gzip's speed.

Another observation is that DTDs can be well- or ill-suited for compression. For example, in stats2 (baseball statistics), the DTD says that each player element has a sequence of optional sub-elements ($e_1^?, \ldots, e_n^?$), but the actual data exhibits only two instances of this content model. Similarly, common content models like $(e_1|\cdots|e_n)^*$ do not provide any information that helps dtdppm. Finding ways to take advantage of such content models is an important area for future work, especially since the same kind of techniques may be useful for compressing regular expression-typed text in XML Schema.

Another problem is that XML's data model is ordered, whereas many data sources (e.g. relation fields, semistructured data trees, or BibTeX records) are conceptually unordered. DTDs cannot express unordered content models efficiently so the content model $(e_1|\cdots|e_n)^*$ is often used as an approximation. Other schema systems such as ASN.1, RELAX/NG and XML Schema do provide unordered content models, but it is not obvious how to compress with respect to such content models effectively. One possibility is to sort content in unordered content models so as to place it in a normal form, as with attribute lists; however, this transformation is non-streaming.

XML encourages a structured approach to data management, but this approach is usually followed only up to a point. A typical example of the use of low-level character data formats is the use of date strings `Mar 15 17:55` instead of XML markup

```
<date><month>Mar</month><day>15</day>
      <time><hour>17</hour><min>55</min></date>
```

The former representation is briefer and more human-readable, but xmlppm will likely compress the latter much better. XML Schema's datatypes (especially dates) may be useful for improving compression for this kind of data.

6. RELATED WORK

Liefke and Suciu's XMill [13] is probably the best known XML compressor. One interesting aspect of XMill is that it allows user-defined container specifications using XPath expressions to define containers and to specify datatype-specific compressors. This can significantly improve compression, but may require nontrivial user effort. It is possible that XMill could be made schema-conscious by automatically generating specifications from schemas.

Levene and Wood [12] propose DTD-based encodings for XML data in which the encoding is dependent on the current content model. For example, content matching $r|s$ is encoded by sending 0 if the content matches r, or 1 if it matches s, then encoding the content relative to r or s respectively. This encoding has not been implemented as far as we know; also, although Levene and Wood prove an optimality result, it rests on very strong assumptions (data must conform to a nonrecursive DTD and be generated by independent random choices). This is a step in the right direction, but more theoretical understanding is needed.

Jeuring and Hagg [10] have developed XComprez, which compresses valid XML using an encoding similar to that of Levene and Wood. They use a powerful experimental programming language called Generic Haskell in which the compressor constitutes approximately 650 lines of code (in contrast to 4300 lines of C++ code for `xmlppm` and 9000 lines for `dtdppm`). It is not yet clear whether this approach scales to large XML documents, but advanced programming tools like Generic Haskell may make it easier to rapidly prototype compression techniques prior to full-scale implementation.

SCMPPM [1] is an XMLPPM variant that uses a separate PPM model to compress the text content under each element. It achieves reported improvements of 20% over plain XMLPPM when compressing large TREC datasets. However, it has not been evaluated on other data, so this result must be taken with a grain of salt.

7. FUTURE WORK

As stated in the introduction, we view `dtdppm` as the first step in a logical progression to RELAX/NG- and XML schema-conscious compression tools. In particular, RELAX/NG seems like a logical next step because it is not much more complicated than DTD yet supports datatypes for text content. As for XML Schema, we are intrigued by the possibility of compressing text relative to arbitrary regular expressions. However, we suspect that obvious ways of doing this will not be as effective as plain PPM or `xmlppm`; instead, we intend to find a way to combine PPM and regular expression-based modeling. If such an approach can be found, we believe it will also help with element content compression.

Finally, the prototype implementation[9] has a few bugs that need to be fixed, and several worthwhile optimizations appear possible (particularly pre-compiling or specializing `dtdppm` to a DTD).

8. CONCLUSIONS

The purpose of this paper was to determine whether DTD-conscious compression techniques offer enough benefits, relative to state-or-the-art XML compression, to be worth the (nontrivial) implementation effort needed. We implemented a validating compressor, `dtdppm`, which reads in a DTD and XML document and simultaneously validates and compresses it. In addition, `dtdppm` performs several optimizations on the encoding which are only possible in the presence of a DTD. Put together, these optimizations can improve compression by up to 43% over `xmlppm`. While `dtdppm` can be very effective for small or highly-structured documents, it may not compress unstructured, mostly-text, or large documents significantly better than `xmlppm`. Nevertheless, we found `xmlppm` and `dtdppm` compress large documents as well as `bzip2` but significantly (15-60%) faster.

We believe that this is the first comprehensive assessment of a DTD-conscious XML compression tool.

[9] `http://xmlppm.sourceforge.net/dtdppm`

9. REFERENCES

[1] J. Adiego, P. de la Fuente, and G. Navarro. Merging prediction by partial matching with structural contexts model. In *Proc. 2004 IEEE Data Compression Conference (DCC'04)*, page 522, 2004.

[2] Peter Buneman, Byron Choi, Wenfei Fan, Robert Hutchison, Robert Mann, and Stratis Viglas. Vectorizing and querying large XML repositories. In *Proc. 21st Int. Conference on Data Engineering (ICDE 2005)*, 2005. To appear.

[3] Peter Buneman, Martin Grohe, and Christoph Koch. Path queries on compressed XML. In *Int. Conference on Very Large Data Bases (VLDB'03)*, pages 141–152, 2003.

[4] J. Cheney. `xmlppm`, version 0.98.2. `http://xmlppm.sourceforge.net/`.

[5] James Cheney. Compressing XML with multiplexed hierarchical models. In *Proc. 2001 IEEE Data Compression Conference (DCC 2001)*, pages 163–172. IEEE, 2001.

[6] J. G. Cleary and I. H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Trans. Comm.*, COM-32(4):396–402, 1984.

[7] J.-L. Gailly. `gzip`, version 1.2.4. `http://www.gzip.org/`.

[8] Marc Girardot and Neel Sundaresan. Millau: An encoding format for efficient representation and exchange of XML over the web. *Computer Networks*, 33(1–6):747–765, 2000.

[9] Todd J. Green, Ashish Gupta, Gerome Miklau, Makoto Onizuka, and Dan Suciu. Processing XML streams with deterministic automata and stream indexes. *Transactions on Database Systems*, 29(4), December 2004.

[10] Johan Jeuring and Paul Hagg. Generic programming for XML tools. Technical Report UU-CS-2002-023, Utrecht University, 2002.

[11] W.Y. Lam, W. Ng, P.T. Wood, and M. Levene. XCQ: XML compression and querying system. In *Proc. 12th Int. Conference on the World Wide Web (WWW 2003)*, 2003.

[12] M. Levene and P. T. Wood. XML structure compression. In *Proc. 2nd Int. Workshop on Web Dynamics*, 2002.

[13] Hartmut Liefke and Dan Suciu. XMill: An efficient compressor for XML data. In *SIGMOD '00: Proc. 2000 ACM SIGMOD international conference on management of data*, pages 153–164. ACM Press, 2000.

[14] A. R. Schmidt, F. Waas, M. L. Kersten, D. Florescu, I. Manolescu, M. J. Carey, and R. Busse. The XML benchmark project. Technical Report INS-R0103, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, The Netherlands, 2001.

[15] J. Seward. `bzip2`, version 0.9.5d. `http://sources.redhat.com/bzip2/`.

[16] Dmitry Shkarin. PPM: One step to practicality. In *Proc. 12th IEEE Data Compression Conference*, pages 202–211, 2002.

[17] P. M. Tolani and J. R. Haritsa. XGRIND: A query-friendly XML compressor. In *Proc. 18th Int. Conference on Data Engineering (ICDE'02)*, pages 225–234. IEEE, 2002.

Towards a Query Language for Multihierarchical XML: Revisiting XPath

Ionut E. Iacob[*]
Department of Computer Science
University of Kentucky
Lexington, KY, USA
eiaco0@cs.uky.edu

Alex Dekhtyar[†]
Department of Computer Science
University of Kentucky
Lexington, KY, USA
dekhtyar@cs.uky.edu

ABSTRACT

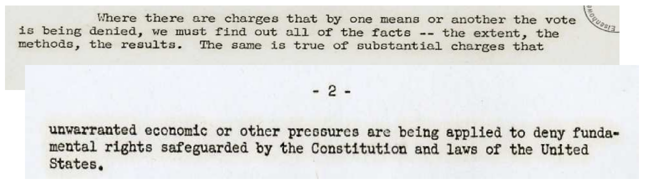
In recent years it has been argued that when XML encodings become complex, DOM trees are no longer adequate for query processing. Alternative representations of XML documents, such as multi-colored trees [7] have been proposed as a replacement for DOM trees for complex markup. In this paper we consider the use of Generalized Ordered-Descendant Directed Acyclic Graphs (GODDAGs) for the purpose of storing and querying complex document-centric XML. GODDAGs are designed to store multihierarchical XML markup over the shared PCDATA content. They support representation of overlapping markup, which otherwise cannot be represented easily in DOM. We describe how the semantics of XPath axes can be modified to define path expressions over GODDAG, and enhance it with the facilities to traverse and query overlapping markup. We provide efficient algorithms for axis evaluation over GODDAG and describe the implementation of the query processor based on our definitions and algorithms.

1. INTRODUCTION

XML has become a popular approach to storage and transfer of diverse data because of its simplicity and transparency, as well as because of wide availability of (free) tools for working with it. Availability of open-source XML-related standards allows software developers build XML-enabled applications in a straightforward manner: XML files are parsed using a combination of SAX and DOM parsers, constructed memory-resident DOM trees are accessed from applications via DOM API calls. More complex XML management tasks involve the use of XPath and/or XQuery expressions for querying the content of DOM trees and XSLT for converting the content/structure of the tree, usually, for the purpose of visualizing the data. For simple XML data, XPath/XQuery over DOM Trees provide efficient and convenient way for querying.

[*]Work supported, in part, by the NEH grant RZ-20887-02.

[†]Work supported, in part, by the NSF grant ITR-0219924 and the NSF grant ITR-0325063.



Where there are charges that by one means or another the vote is being denied, we must find out all of the facts -- the extent, the methods, the results. The same is true of substantial charges that

- 2 -

unwarranted economic or other pressures are being applied to deny fundamental rights safeguarded by the Constitution and laws of the United States.

Figure 1: A fragment of a letter detailing the proposed Civil Rights Program to the members of President Eisenhower's Cabinet.

However, straightforward approaches to organizing XML for querying might yield unsatisfactory solutions when complex markup is considered. In [7] Jagadish et al. observed that querying XML data in the presence of several hierarchies for encoding features of the same objects can be done more efficiently if alternative data structures are used in place of a set of independent DOM trees, one for each of the hierarchies. Jagadish et al. proposed a data structure called *multi-colored tree* (MCT) for storing such markup and discussed efficient query evaluation strategies.

The approach of [7] was designed with data-centric XML in mind. The multicolored tree structure is built on top of individual XML nodes. This allows hierarchies of different "colors" to share content of some of the nodes. When *document-centric* XML is considered, however, there is an additional dimension, not captured by MCTs: the *sharing* of information in the hierarchies occurs at the level of content, rather than XML elements. Indeed, typically, document-centric XML documents are built by starting with a text and introducing various markup *on top* of it. When more than one hierarchy is used to encode features of a text, often the scopes of different markup elements *overlap*. This is illustrated on the following example.

EXAMPLE 1. *Consider a fragment of [8], shown in Figure 1. We describe two markup hierarchies for this document. First hierarchy describes, using elements* `<p>` *(paragraph),* `<sentence>` *and* `<w>` *(word), the structure of the text of the document. Our second hierarchy uses elements* `<page>` *and* `<line>` *to describe the physical layout of the text. The (somewhat simplified) corresponding XML encodings of this fragment are shown in Figures 2.(a) and 2.(b). Examination of the scopes of the XML elements in these figures reveals numerous overlapping conflicts. In particular, we mention the conflict between the scope of the* `<page no="1">` *element and the content of both* `<p>` *and* `<sentence no="14">` *elements. Similarly, the* `<w>` *element around the word* `"fundamental"` *over-*

laps both `<line no="1">` *and* `<line no="2">` *elements of* `<page no="2">`*. Overall, even this simple fragment, described using just two hierarchies contains six pairs of elements with overlapping content.*

Overlap in content of elements means that the markup presented in Figure 2 cannot be stored in a single XML document/DOM tree in a straightforward manner. As they lack facilities to store overlapping markup, it also cannot be stored in a single MCT. At the same time, storing each hierarchy in a separate DOM tree is inefficient from the perspective of query processing. For example, a user query

> Find all sentences completely or partially located on page 1, which contain the word "charges"

requires navigation through both text structure and physical location markup. Similar to the cases considered in [7], executing such a query as a join is inefficient. To make matters worse, the full answer to this query must include sentence number 14, only partially located on page 1. This means that the abovementioned query is not expressible in XPath (or XQuery, for that matter) over the set of the two encodings in Figure 2[1].

In [9], Sperberg-McQueen and Huitfeldt have introduced Generalized Ordered-Descendant Directed Acyclic Graphs (GODDAGs), a data structure for storing concurrent/ multihierarchical markup. A GODDAG combines DOM trees of individual XML hierarchies together by "tying" them at the top, root level, and at the bottom, content level. In [9], Sperberg-McQueen cite the need for appropriate mechanisms for building GODDAGs and querying data stored in them. The former problem had been addressed in [6]. In this paper, we adopt GODDAG (formally described in Section 2) as the data structure for storing concurrent markup. We then proceed to: (i) define the semantics of XPath axes over multiple hierarchies in GODDAG structures (Section 3); (ii) enhance XPath syntax and semantics with constructs for capturing overlapping markup (Section 3); (iii) develop and implement algorithms for axis evaluation over GODDAG[2] and conduct a preliminary study of the efficiency of enhanced XPath over GODDAG as the means of querying multihierarchical, overlapping markup (Section 4).

This paper describes the first steps toward a query language for document-centric XML data with overlapping hierarchies. We give an extension of XPath, as a navigational language through a data structure that we consider appropriate for representing multihierarchical markup. The next step, currently under development, would be to use this XPath extension in an XQuery language extension for querying multihierarchical XML documents.

2. DATA STRUCTURE FOR OVERLAPPING HIERARCHIES

We identify three basic principles for choosing a data structure for overlapping hierarchies: (i) we want to preserve individual hierarchies inside the complete document representation, (ii) we want to easily navigate from one structure to another, and (iii) we want to capture relationships between elements in different hierarchies.

Is is a fact that complex queries are likely to be expensive ([3, 2]). In [7] it is pointed out that, even for complex hierarchies, a tree-like structure is desirable due its relative navigation simplicity.

We start by introducing *concurrent markup hierarchies* and *distributed XML documents*. A concurrent markup hierarchy (CMH) is a collection of schema definitions (DTDs, XSchemas, etc...) that share a single (root) element name, and only it[3]. Individual schemas are called *hierarchies*. Given a CMH $C = \langle T_1, \ldots, T_k \rangle$, a distributed XML document (DXD) D over C is a collection of XML documents $(d_1, \ldots, d_k)$, one for each hierarchy of C, such that all documents **have the same PCDATA content**[4] Individual documents d_i are called *components* of D. They are not expected to be valid w.r.t. their schema T_i, but must contain markup only from T_i. This separation of markup in a DXD addresses principle (i) above: each document preserves the structure of the specific encoding. Two XML documents in Figure 2 show us an example of a DXD with two document components: d_1 on top (corresponding to a "text" hierarchy), and d_2 at the bottom (corresponding to a "physical layout" hierarchy). As clear from this example, DXDs can incorporate within them overlapping markup.

Representing DXD components as individual independent DOM trees is inconvenient, as illustrated in [7]. Instead, we use a structure called *General Ordered-Descendant Directed Acyclic Graph (GODDAG)*, originally introduced by Sperberg-McQueen and Huitfeldt in [9] precisely for the purpose of storing concurrent markup. Informally, a GODDAG for a distributed XML document D can be thought of as the graph that unites the DOM trees of individual components of D, by merging the root node and the text (PCDATA). Because of possible overlap in the scopes of XML elements (text nodes) from different component documents, the underlying content of the document is stored *not* in text nodes, but in a special *new type* of node called *leaf node*. In a GODDAG, leaf nodes are children of the text nodes, and they represent a consecutive sequence of content characters that is *not broken by an XML tag from* **any** of the components of the distributed XML document. While each component of D will has its own text nodes in a GODDAG, the leaf nodes will be shared among all of them. As a consequence, **leaf nodes have multiple parents**: one in each component of D.

The GODDAG for the DXD in Figure 2 is illustrated in Figure 3. In the figure, nodes in the "text" hierarchy are on the top part, whereas nodes for the "physical layout" hierarchy are at the bottom. Leaf nodes are represented in the middle as rectangles corresponding to the PCDATA they cover. Element nodes are explicitly drawn with names and attribute values. Text nodes are symbolized by T in a circle. To easily identify the nodes, we put a unique label next to each node. Note here, for example, that the word "fundamental" is broken into two leaf nodes: L12:"funda" and L13:"mental". This allows us to represent the content of the appropriate `<w>` element (116) in the first hierarchy as {L12, L13}, while including L12 and L13 in the scope of two different `<line>` elements (29 and 211 respectively).

To define GODDAG formally, we need to introduce some notation. For an XML document d we let $root(d)$ denote the root element of d and $nodes(d)$ – the set of all nodes in DOM of d. For a node $x \in nodes(d)$ we let $string(x)$ be the PCDATA content of x (as defined in XPath [1]). We also set $start, end : nodes(d) \to \mathbb{N}$ to return the offset positions in $string(root(d))$ of *start tag* and *end tag* respectively for a node $x \in nodes(d)$. If x is a text node, then $start(x)$, $end(x)$ denote the start offset and end offset respectively. For a distributed document D we let $leaves(D)$ represent

[1]We note that it is possible to represent the desired query in XQuery by modifying the representation in Figure 2 in a number of ways, e.g., with ID/IDREF attributes, or with `<leaf>` elements representing GODDAG leafs described elsewhere in the paper.

[2]Due to the lack of space we will not present the algorithms here. The details can be found in [5].

[3]Namespaces can be used to distinguish elements from different hierarchies with the same name, but this fact is not important for the scope of this paper.

[4]The order of characters in the PCDATA content of all documents must be the same.

```
<doc id="CP56483">
...
<p>
<sentence no ="13"> <w>Where</w> <w>there</w> are
  <w>charges</w> that by one means of another the vote
  is being denied, we must find out all of the
  facts -- the extent, the methods, the results.
</sentence>
<sentence no="14">The same is true of substantial
  <w>charges</w> that unwarranted economic of other
  pressures are being applied to deny
  <w>fundamental</w> <w>rights</w> <w>safeguarded</w>
  by the Constitution and laws of the United States.
</sentence>
</p>
...</doc>
```

(a)

```
<doc id="CP56483">
...
<page no="1">
<line no="31"Where there are charges that by
      one means of another the vote</line>
<line no="32">is being denied, we must find out
      all of the facts -- the extent, the</line>
<line no="33">methods, the results. The same is true of
      substantial chargers that</line>
</page>
<page no="2">
<line no="1">unwarranted economic of other pressures are
      being applied to deny funda</line>
<line no="2">mental rights safeguarded
      by the Constitution and laws of the United</line>
<line no="3"> States.
...
</page> ...</doc>
```

(b)

Figure 2: Encoding of the fragment from Figure 1: (a) text structure, (b) physical location.

the set of all *leaf nodes* in D and we extend the domain of functions $string, start$, and end over the $leaves(D)$ set. For *leaf nodes* these functions are defined in the same way as for *text nodes*.

DEFINITION 1. *Let $D = (d_1, \dots, d_k)$ be a distributed XML document. A GODDAG of D is a directed acyclic graph (N, E) where the sets of nodes N and edges E are defined as follows:*
- $N = \cup_{i=1}^{k} nodes(d_i) \cup leaves(D)$
- $E = \cup_{i=1}^{k} \{(x, y) | x, y \in nodes(d_i) \wedge$ *x is the parent of y*$\} \cup$ $\cup_{i=1}^{k} \{(x, y) | x \in nodes(d_i)$ *is a text node*, $y \in leaves(D) \wedge$ $start(x) \leq start(y) < end(y) \leq end(x)\}$

The GODDAG data structure solves nicely the problem of navigation between CMH structures (principle (ii)): all hierarchies are connected via the common root node and common leaf nodes. The data structure also captures relationships among features in different structures. For instance, in the GODDAG in Figure 3, we can find all sentences partially or totally located on page 1: from `<page no="1">` (node 21) we navigate down and find all leaf nodes it contains (leaves `L1` to `L10`); then we navigate up in the other hierarchy and find all `<sentence>` ancestors for these leaf nodes (nodes 13 and 14). In fact, as we show below, the semantics of all standard relationships between elements from different hierarchies (ancestor, descendant, overlapping, following, preceding) *can be expressed in terms of relationship between the corresponding leaf nodes*.

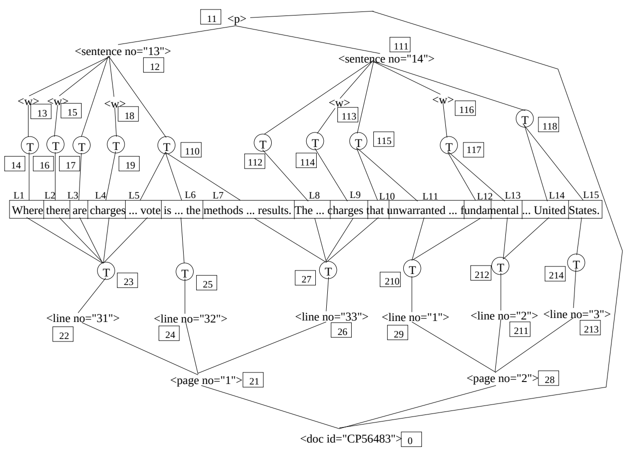


Figure 3: A GODDAG for the distributed document DXD in Figure 2

3. QUERYING DISTRIBUTED XML DOCUMENTS

XPath is a language for addressing parts of an XML document. It is intensively used as part of some XML query languages (XQuery), and can be used itself to query XML documents. In fact, in XQuery queries, XPath expressions are responsible for traversing the underlying XML document model (DOM tree) to discover requested XML nodes.

We argue in [5] that even simple queries are hard to express in XPath over representations of concurrent hierarchies that involve markup fragmentation or empty elements to overcome markup conflicts. In this section we show that when distributed XML documents are represented in GODDAG structures, we can express such queries as path expressions in straightforward ways. In addition, we show that individual components of path expressions (we concentrate on axes) have natural semantics over GODDAG, a semantics that specializes to XPath over DOM semantics when single-component documents are considered. We start by discussing how individual XPath axes can be defined in GODDAGs then we introduce formal definitions of XPath components over GODDAG.

3.1 Path Expressions Over GODDAG

Recall that XPath uses a tree of nodes model to represent an XML document. There are seven types of nodes, the *root node* (a unique node in an XML document), *element, text, attribute, namespace, processing-instruction,* and *comment* nodes. The main syntactical construction of XPath is *expression*. An *expression* operates on a *context node* and manipulates *objects* of four kinds: node-set, boolean, string, and numeric.

The instrument for addressing sets of nodes in a document is the *location path* composed of one or more *steps*. Each step consists of an *axis*, a *nodetest* and zero or more *predicates*. An axis determines the direction of traversal from the current (context) node, while nodetests and predicates filter nodes that do not match them. A *location path* syntax can be summarized as follows (comprehensive syntax is given in [1]):

```
    locationPath := step1/step2/.../stepn
    step := axis::node-test predicate*
    predicate := [expression]
```

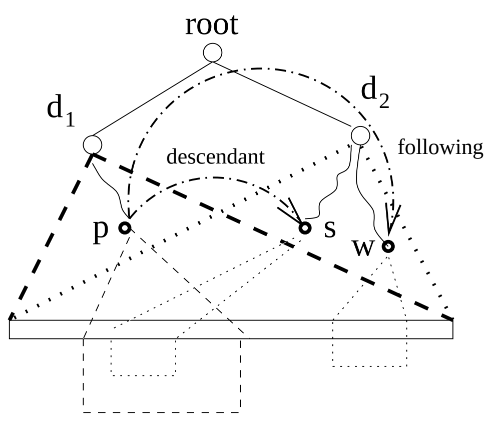


Figure 4: Descendant and following axes in GODDAG.

The main syntactical construction for a *step* evaluation is *axis*: for each node in the current *context node* set an *axis* is evaluated to a set of nodes according to the respective *axis* definition. The set of nodes from *axis* evaluation is filtered by the *node-test* (basically a node type test or a name test for *element* nodes) and *expression* result (evaluated to *true* or *false*) in the context of each node of *axis* evaluation set (*axis* plays the selection role, *node-test* and *predicate* play the filtering role). XPath uses 13 *axes* to address nodes in a document: *ancestor, ancestor-or-self, attribute, child, descendant, descendant-or-self, following, following-sibling, namespace, parent, preceding, preceding-sibling,* and *self*.

We illustrate the problem of definition of XPath axes over a GODDAG in the following examples. In the context of our example from Figures 1 and 2, consider the query "Find all sentences completely inside page 1". If `<page>` and `<sentence>` markup were in the same hierarchy, we would have expressed this query using the following XPath expression:

`//page[attribute::no="1"]/descendant::sentence`

But in our GODDAG (Figure 3), they are not. Yet, the representation mechanism should not affect our understanding of the relationship between pages and sentences. By definition of the *descendant* axis, a `<sentence>` node is a descendent of a `<page>` node if it is located in the `<page>` node's subtree. However, we can describe a descendant relationship in a different way:

> a node x is a descendant of node y iff the content range of x is *completely included* in the content range of y.

When considered over DOM trees, these two definitions are (almost) equivalent. The key difference between them is that while the former definition is DOM-specific, the latter is not. In Figure 4 we show the latter definition applied to GODDAG. Here, two components d_1 and d_2 of a DXD are shown. Node p in component d_1 has content that subsumes completely the content of node s from component d_2. By applying the definition above, we can state that s is a *descendant* of p *in the given DXD*. Similarly, because the *ancestor* relationship is the inverse of the descendant, we can use the same idea to state that p is an *ancestor* of s in the DXD.

We can use similar intuition to redefine *following* and *preceding* axes. Indeed, a node x follows a node y iff the entire PCDATA content of x is located *after* the entire PCDATA content of y in a DOM tree. This statement is, again, independent on the DOM tree structure (as opposed to the definition of the *following* axis, which relies on the document order), and therefore can be transferred to GODDAG, as illustrated in the Figure 4. Node w of component d_2 has content that lies *after* the content of node p, hence, we can state the w *follows* p and, conversely, p *precedes* w.

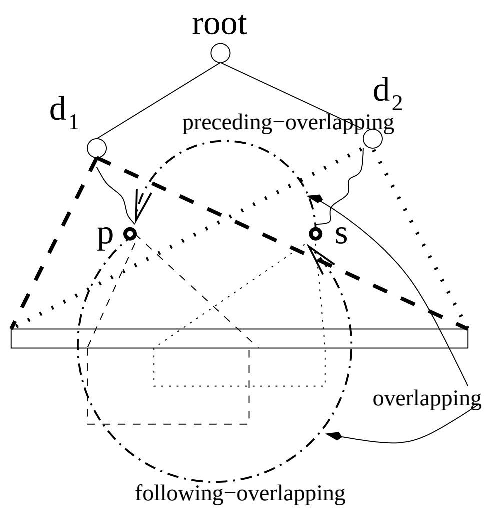


Figure 5: Axes for overlap in GODDAG.

At the same time, not all axes can be redefined in such a way, in particular, *child* and *parent* cannot transcend the boundaries of a single component in a way *descendant* and *ancestor* do. This is because unlike the notion of descendant, childhood-parenthood relations are tied to tree structures: a node x is a child of a node y iff there is an edge from y to x. Similarly, *preceding-sibling* and *following-sibling* axes rely on the existence of edges between the nodes in the DOM tree, not just on the position of the content. These axes, as well as *self* will not be extended beyond individual components of the distributed document.

One more observation can be made. Given a node x of a DOM tree, the five axes *ancestor*, *descendant*, *self*, *preceding* and *following* partition the entire DOM tree into five disjoint sets of nodes: that is, every node in the DOM tree will belong to exactly one of these axes as traversed from x. This property, however, does not hold in GODDAG: as shown in Section 2, there are GODDAG nodes with *overlapping* content (See Figure 5). Traversing the GODDAG using any of the five axes above will never yield any node that overlaps the context node in content. At the same time, as have been illustrated, queries over GODDAG require computing overlap. To accommodate for this need, we consider enhancing XPath with three new axes: *preceding-overlapping*, *following-overlapping* and *overlapping*. Intuitive meaning of these axes, as illustrated in Figure 5 is quite straightforward: x is in the result of applying *preceding-overlapping* axis to y iff x and y overlap in scope and x starts before y. In this case, y will be in the result of *following-overlapping* applied to x. The *overlapping* axis is the union of *preceding-overlapping* and *following-overlapping*.

We can now proceed to give formal definitions to XPath components over GODDAG, including the enhances apparatus to support markup overlap.

3.2 XPath over GODDAG

Let D be a distributed XML document over a concurrent XML hierarchy $C = \langle T_1, \ldots, T_k \rangle$. We define 11 new XPath axes, over the distributed document D, in the context of a node $x \in nodes(D)$: *xancestor, xdescendant, xancestor-or-self, xdescendant-or-self, xfollowing, xpreceding, following-overlapping, preceding-overlapping, overlapping, xancestor-or-overlapping*, and *xdescendant-or-overlapping*. The first six axes are versions of the corresponding XPath axes extended to GODDAG. The remaining five axes do not have analogs in XPath.

Xancestor/xdescendant axes are defined using superset/ subset relation on the content of the nodes, represented via a set of *leaf nodes* in the GODDAG. To define *xfollowing* and *xpreceding* axes, we use the relative positions of nodes in the GODDAG. However, we observe that there is no total document order over a GODDAG: overlapping markup will be incomparable.

DEFINITION 2. *The following new axes are defined:*

1. $xancestor ::= ancestor(x) \cup \{y \in nodes(D{-}doc_D(x)) | \ start(y) \leq start(x) \leq end(x) \leq end(y)\}$.
2. $xdescendant ::= descendant(x) \cup \{y \in nodes(D - doc_D(x)) | \ start(x) \leq start(y) \leq end(y) \leq end(x)\}$.
3. $xancestor{-}or{-}self ::= xancestor(x) \cup \{x\}$.
4. $xdescendant{-}or{-}self ::= xdescendant(x) \cup \{x\}$.
5. $xfollowing ::= following(x) \cup \{y \in nodes(D - doc_D(x)) | \ start(y) \geq end(x)\}$.
6. $xpreceding ::= preceding(x) \cup \{y \in nodes(D - doc_D(x)) | \ end(y) \leq start(x)\}$.
7. $following{-}overlapping ::= \{y \in nodes(D) | \ start(x) < start(y) < end(x) < end(y)\}$.
8. $preceding{-}overlapping ::= \{y \in nodes(D) | \ start(y) < start(x) < end(y) < end(x)\}$.
9. $overlapping ::= following{-}overlapping(x) \cup preceeding{-}overlapping(x)\}$.
10. $xancestor{-}or{-}overlapping ::= xancestor(x) \cup overlapping(x)$.
11. $xdescendant{-}or{-}overlapping ::= xdescendant(x) \cup overlapping(x)$.

We give some examples of the extended axes for the GODDAG shown in Figure 3. (we use node labels to identify nodes in the graph)

(A) $xdescendant(21) = \{22, 24, 26, 23, 25, 27, 14, 16, 17, 19, 110, 13, 15, 18, 12, 112, 114, 113\}$. Note 21 corresponds to the `<page no="1">` markup. The *xdescendants* of this node are all its descendants in the "physical layout" component (lines 31, 32 and 33 and corresponding text nodes) as well as the contents of sentence 13 (nodes 12,13,15,18 and the corresponding text nodes). In addition, parts of sentence 14 (node 113 and text nodes 112 and 114) also are xdescendants of 21. At the same time, sentence 14 itself is *not* an xdescendant of page 1.

(B) $following{-}overlapping(26) = \{111, 115\}$

Node 26 represents `<line no="33">` markup from page 1. The scope of its content is leaf nodes L7—L10. The scope of node 111 (`<sentence no="14">` is L8—L15: because it starts after the scope of node 26 starts and ends after the scope of node 26 ends, it belongs to the result of evaluation of the *following-overlapping* axis. Incidently, text node 115 also overlaps node 26 on the right, thus it is added to the result as well.

Remark. We note that Definition 2 allows a node x to be both an $xdescendant$ and an $xancestor$ of a node y: if $start(x) = start(y)$, $end(x) = end(y)$ and *they are in different documents*.

Proposed axes allow us to express queries to multihierarchical (distributed) documents in a straightforward manner. Consider, for example, the following queries:

(Q1): Find all sentences completely located on page 2;
(Q2): Find all words located on two lines;
(Q3): Find all sentences completely or partially located on page 1 of the document, that contain the word "charges";
(Q4): Find all occurrences of the word "Constitution" after page 1.

Table 1 shows the path expressions for these queries.

The algorithms for evaluation of the newly defined axes are given in [5].

4. EXPERIMENTAL RESULTS

We have fully implemented in Java an extension of XPath language that includes all the axes described in Definition 2. We call this processor *GOXPath*. GOXPath is a *main-memory processor*: all queries are processed over the memory-resident GODDAG structure, without addressing persistent storage. In this section we describe our preliminary study of the efficiency of this processor.

We report the results of four tests. The goals of the experiments were: (a) to compare the evaluation of extended XPath axes over documents with 2, 3, 4, 5, and 6 hierarchies; (b) to compare evaluation of axes over multihierarchical documents with different sizes (ranging from 5,000 nodes up to 500,000 nodes); (c) to compare evaluation of queries of different lengths; (d) to compare GOXPath performance with the execution of equivalent XPath queries (in terms of number of nodes manipulated) by Xalan[5] and Dom4j[6] processors. We emphasize that the goal of part (d) was not to prove that GOXPath is faster than certain XPath processors. Rather we want to show that on similar workloads GOXPath exhibits comparable performance. The tests were run on a Dell GX240 PC with 1.4Ghz Pentium 4 processor and 256 Mb main memory. The data input for the first two experiments was a distributed XML document obtained by multiplying the XML samples shown in Figure 2 enhanced with more markup when more than two hierarchies were used. The documents sizes ranged from 2MB up to 40MB on disk. Each query was evaluated four times, the average time was plotted.

In the first experiment we used 5 documents with 2, 3, 4, 5, and 6 hierarchies and of approximately 50, 55, 60, 65, and 70 thousand of nodes respectively. On these document instances we evaluated two queries, `/descendant::page/xdescendant::*`, and `/descendant::page/overlapping::*`, and the graphs of running time are shown in Figure 6 (a). The experimental results suggest that GOXPath performances are not significantly influenced by the number of hierarchies, but rather by the number of nodes that are manipulated (in the case of *xdescendant* axis the number of nodes slightly increases with the number of hierarchies, whereas for *overlapping* the number of nodes is approximately the same).

In the second experiment we studied the exaluation of extended axes for documents of different size (we used two hierarchies in this experiment). We ran two queries, `/descendant::page/ xdescendant::*` and `/descendant::page/overlapping::*`, on documents of 5 up to 5,000 thousand nodes. The experimental results in Figure 6 (b) suggest linear dependence of axis evaluation on document size [5].

In the third experiment we tested GOXPath performance on queries of different length. We used two sets of eight queries each. Each query in the first set had the prefix `/descendant::page//`, and continued with 1 up to 8 `overlapping::*` location steps. Similarly, each query in the second set had a prefix `/descendant::page/`, and continued with 1, up to 8 `xdescendant-or-self::*` location steps (note that approximately the same number of nodes were

[5]http://xml.apache.org/xalan-j/
[6]http://www.dom4j.org

Query	Path Expression
Q1	`/xdescendant::page[@no="2"]/xdescendant::sentence`
Q2	`/xdescendant::word[overlapping::line]`
Q3	`/xdescendant::page[@no="1"]/xdescendant-or-overlapping::sentence[descendant::w[string(.)="charges"]]`
Q4	`/xdescendant::page[@no="1"]/xfollowing::w[string(.)="Constitution"]`

Table 1: Using newly defined axes to express queries over multihierarchical XML documents.

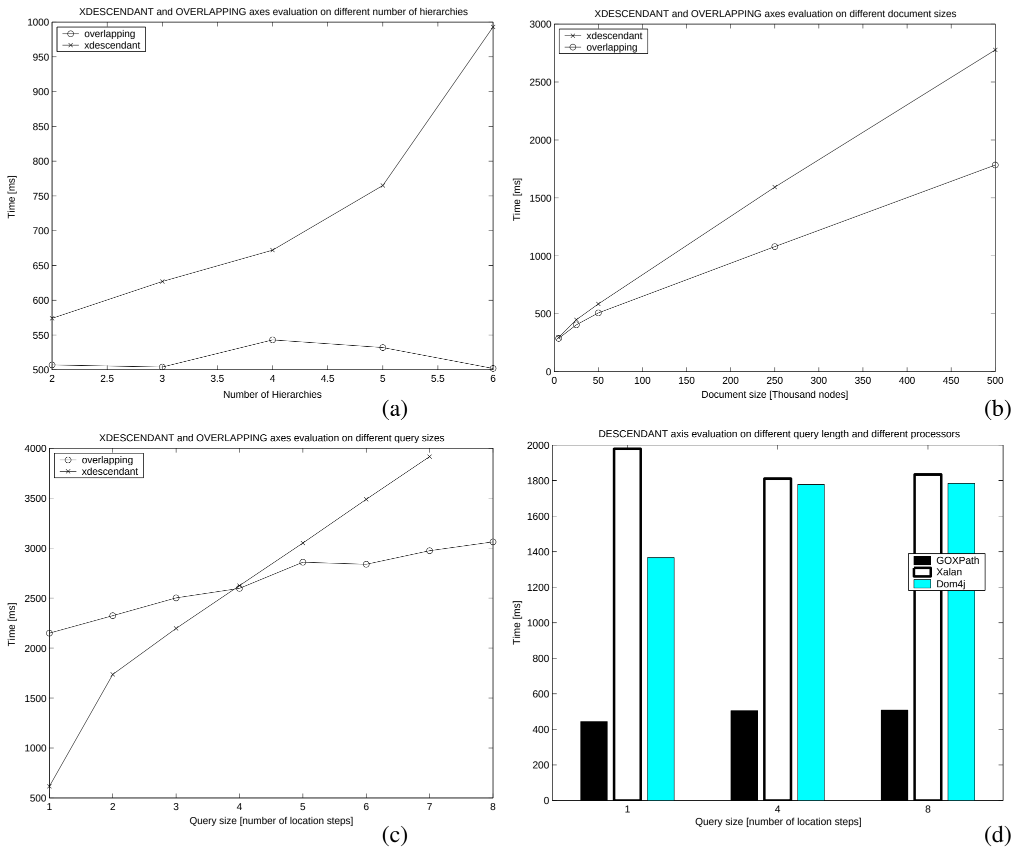


Figure 6: Experimental results.

processed at each step). The results shown in Figure 6 (c) clearly indicate linear dependence of query evaluation time on query size.

Finally, the last experiment compared the running time of GOXPath and XPath processors (Xalan and Dom4j) on workloads of comparable size (similar queries and the same number of nodes to be processed). We used documents with approximately 50 thousand nodes, two hierarchies for GOXPath, and markup fragmentation for Xalan and Dom4j. The same three queries were evaluated by each processor: `/descendant::page/descendant::*`, and the same as the preceding but ending with four, respectively eight `descendant::*` location steps. The test results are shown in Figure 6 (d) and demonstrate that GOXPath has similar performances as Xalan and Dom4j.

The experiments conducted here are preliminary and a more extensive testing is currently underway. But even these experiments show that our XPath implementation over GODDAG is efficient enough to be used in practice (and in fact, it is used as part of a larger suite of tools)[4].

5. REFERENCES

[1] XML Path Language (XPath) (Version 1.0). http://www.w3.org/TR/xpath, Nov 1999.

[2] G. Gottlob, C. Koch, and R. Pichler. The complexity of XPath query evaluation. In *Proceedings of PODS, San Diego, CA.*, pages 179–190, June 2003.

[3] G. Gottlob, C. Koch, and R. Pichler. XPath query evaluation: Improving time and space eficiency. In *Proceedings of ICDE'03, Bangalore, India.*, pages 379–390, Mar 2003.

[4] I. E. Iacob and A. Dekhtyar. A framework for processing complex document-centric XML with overlapping structures. In *ACM SIGMOD Conference*, 2005. Demo, accepted.

[5] I. E. Iacob and A. Dekhtyar. Queries over Overlapping XML Structures. Technical Report TR 438-05, U. of Kentucky, CS Dept., March 2005. http://dblab.csr.uky.edu/∼eiaco0/publications/TR438-05.pdf.

[6] I. E. Iacob, A. Dekhtyar, and K. Kaneko. Parsing Concurrent XML. In *Proceedings WIDM*, pages 23–30, November 2004.

[7] H. V. Jagadish, L. V. S. Lakshmanan, M. Scannapieco, D. Srivastava, and N. Wiwatwattana. Colorful XML: one hierarchy isn't enough. In *Proceedings SIGMOD*, pages 251–262. ACM Press, 2004.

[8] M. M. Rabb. The civil rights program - letter and statement by the attourney general. The Dwight D. Eisenhower Library, Abilene, KS, http://www.eisenhower.utexas.edu/dl/Civil_Rights_Civil_Rights_Act/CivilRightsActfiles.html, April 10 1956.

[9] C. M. Sperberg-McQueen and C. Huitfeldt. GODDAG: A Data Structure for Overlapping Hierarchies. In *DDEP/PODDP, Munich*, pages 139–160, Sept. 2000.

Indexing Schemes for Efficient Aggregate Computation over Structural Joins

Priya Mandawat
Dept of Computer Science
University of California, Riverside
pmandawa@cs.ucr.edu

Vassilis J. Tsotras [*]
Dept of Computer Science
University of California, Riverside
tsotras@cs.ucr.edu

ABSTRACT

With the increasing popularity of XML as a standard for data representation and exchange, efficient XML query processing has become a necessity. One popular approach encodes the hierarchical structure of XML data through a node numbering scheme, thus reducing typical queries to special forms (structural, path, twig) of containment joins. In this paper we consider how using an index can facilitate the computation of (exact) aggregates over structural joins. Consider for example, counting how many times `item` nodes appear under `store` nodes in a chain-store database. One straightforward solution would be to actually compute the result of the structural join (`store//item`) and count all such occurrences. However, this requires time proportional to the join computation (which with appropriate buffering is *O(n)* where n is the total number of `store` *and* `item` elements in the database). Instead, we propose the Aggregation B-tree (aB-tree), a balanced index specifically designed for computing aggregate queries such as *count, sum, min, max, average.* The efficiency of this structure arises from the manner in which it maintains *pre-computed partially aggregated* results. The overall aggregate is then computed following specific paths of the tree and accumulating partial values, resulting in an overall *O(log n)* time complexity. Since the aB-tree is an index built on the sorted element lists, it can also be used to perform efficient structural joins by skipping elements that do not participate in the join.

1. INTRODUCTION

The increasing popularity of XML as a format for data exchange and representation has naturally created a demand for efficient query processing on XML documents. The hierarchical structure of XML data is commonly captured through a node numbering scheme [8, 11, 9], thus reducing typical queries to special forms of containment joins, with structural joins (e.g., `store//item`) being the basic building block. A number of structural join algorithms that answer these queries by an exhaustive enumeration of such pairs have been developed. The algorithm proposed in [8] takes as input two ordered lists, one with the ancestor elements and one with the descendants and computes the join with one pass over the two lists (using appropriate buffering). Recently, index methods [2, 3], have also been proposed to improve the performance of structural joins by skipping elements that do not participate in the structural join result.

However, there are queries where the user may desire aggregated data instead of a simple enumeration of the result set. For example, consider the query `min(store//item/@price)` which finds the minimum price of an item over all stores. The straightforward method to address such queries would be to perform the structural join and then compute the aggregate value over the result set. This however is inefficient since computing the aggregate should not require an exhaustive enumeration of all structural join pairs. In this paper, we examine how indexing methods can be used to speed up aggregate queries over structural joins. There is recent work on estimating the size of the result set [6, 7, 1], however the objective of this paper is to address the needs of applications that require *exact* aggregate computations.

More related to our work is [4], where the XA-tree has been proposed that uses Tree-Join algorithms to handle aggregate queries over structural joins. Even though this is an improvement over the straightforward approach, it does not take advantage of the containment property in XML documents. For example, to address a query of the form `count(store//item)` the join performance depends on (i) the number of ancestor (`store`) elements that contain at least one descendant (`item`) element and (ii) the size of the query set (which for a structural join is the ancestor element list). Each such `store` element has an entry in the XA-tree and needs to be searched and *individually* aggregated i.e. this method ignores the fact that in the XML document, ancestor (store) elements may recursively contain other ancestor (store) elements, in which case, one could avoid *individually* aggregating all ancestors. Thus, the worst case complexity of this algorithm remains *O(n)*.

A commonly used numbering scheme assigns a (start, end) value pair to each element in the XML document by doing a depth-first traversal of the document (the pre-order visit assigns the start, while the post-order visit assigns the end). An important property of this encoding is (1) either the interval of a node d is completely contained in the interval of another node a (i.e. $a.start < d.start < d.end < a.end$) in which case a is an ancestor of d (2) or the intervals of two nodes are disjoint in which case they do not have an ancestor-descendant relationship. The interval of an element can thus be used to test the containment property i.e. if a node is the ancestor/descendant of another node.

In this paper, we propose computing aggregates over structural joins using the aB-tree, a balanced index built on the

[*]This research was partially supported by NSF grant IIS-0339032, UC MICRO and Lotus Interworks.

Eighth International Workshop on the Web and Databases (WebDB 2005)
June 16-17, 2005, Baltimore, Maryland

start values associated with the elements of an ancestor list. We devise an algorithm that uses the aB-tree structure and exploits the containment property of the XML document to calculate the overall aggregate by accumulating partially aggregated values only along specific paths of the tree, with a worst case *O(log n)* I/O complexity. In the aB-tree, each pair of consecutive start values is treated as an interval and stores the pre-computed partially aggregated value of all the intervals in the sub-tree rooted at it. Storing pre-computed partially aggregates values over intervals has been proposed in the SB-tree [10] for temporal aggregations. Our structure, however, is different from the SB-tree in (i) that it indexes only start values (ii) the manner in which the aggregates are computed and maintained. This is due to the fact that, in temporal aggregation, the aggregate value for an interval includes only values that span the entire interval whereas in XML aggregation it includes all the values *contained* in the interval. The contributions of the paper can be summarized as follows:

1. We propose the aB-tree, a data structure that supports efficient computation of aggregation queries.
2. We devise an algorithm for the efficient computation of aggregation functions such as count, sum, min, max and average by taking advantage of the containment property of XML documents.
3. Through an extensive experimental evaluation that compares our approach with previous ones, we demonstrate that the aB-tree significantly out-performs existing techniques.

The rest of this paper is structured as follows: In section 2 we formally define the problem while section 3 describes the aB-tree. Section 4 presents the results of our experimentation and section 5 concludes the paper.

2. PROBLEM DEFINITION

Consider two element lists A and D corresponding to the ancestor and descendant query elements respectively and an attribute whose value we denote as *val*. We define the *Aggregate Structural Join* to be a function f(A‡D,val), where '‡' denotes the parent-child relationship (i.e. A/D) or the ancestor-descendant relationship (i.e. A//D) and f is an operator performing one of the following functions:
Count: The number of A_i‡D_j pairs
Sum/Min/Max/Avg: The sum/min/max/avg of vals over all the A_i‡D_j pairs
Furthermore, we define a *Range Aggregate Structural Join*, f(A‡D,val,Q), to be an aggregate structural join reduced over some *range* Q=(s,e). For example, `count(store[state ='California']//item)` where California corresponds to a range over the whole state-space. In the special case where the range Q covers the entire XML document, the Range Aggregate Structural Join reduces to an Aggregate Structural Join.

3. THE AB-TREE

Given a list of ancestor elements A and a list of descendant elements D, the aB-tree is a B-tree built on the start values of all the ancestor elements A_i in A, that have at least one element D_j from list D as a descendant. Each pair of consecutive values in the tree is treated as an interval and each non-leaf interval in the aB-tree has an aggregate value for all the intervals below it associated with it. The leaf intervals store values for the *individual* $A_i s$. The following is a detailed description of the aB-tree index structure:

3.1 The Data Structure

- It is a balanced B-tree.
- Any node other than the root node has at least $\lceil b/2 \rceil$ and at most b element intervals.
- If an internal node N has j intervals $N.I_1$, $N.I_2$,..., $N.I_j$, then it has $j-1$ keys stored in ascending order. The j-th key, denoted $N.k_j$, represents the end of the j^{th} interval and the start of the $j+1^{th}$ interval. Each interval $N.I_j$ has a pointer $N.p_j$ to a child node. The keys in the child nodes must conform to the following *search property*: each key in the sub-tree rooted at $N.p_j$ must be strictly less than the j-th key $N.k_j$ of $N.I_j$ and each key in the sub-tree rooted at $N.p_{j+1}$ must be strictly greater than $N.k_j$. Each interval $N.I_j$ also has a value $N.v_j$ associated with it that is an aggregate over the sub-tree rooted at it.
- A leaf node is identical to an internal node except that: (i) it does not have a child pointer associated with it, and, (ii) the value $N.v_i$ associated with its i^{th} interval $N.I_i = (N.k_{i-1},\ N.k_i)$ corresponds to the aggregate value of the ancestor element with start value $N.k_{i-1}$.
- The root node is allowed to have a minimum of two intervals.

The following is a recursive interpretation for element intervals in the aB-tree nodes. Suppose node N contains a total of j element intervals. Consider the i^{th} element interval $N.I_i$. The start position of $N.I_i$ (denoted by start($N.I_i$)) is specified as follows:
(i) If $i > 1$ then start($N.I_i$) $= N.k_{i-1}$
(ii) If $i = 0$ and N has no parent node in the aB-tree, then start($N.I_i$) $= 0$
(iii) If $i = 0$ and N has a parent node N' such that $N'.p_k = N$, then start($N.I_i$) = start($N'.I_k$)
The end position of $N.I_i$ (denoted by end($N.I_i$)) is specified as follows:
(i) If $i < j$ then end($N.I_i$) $= N.k_i$
(ii) If $i = j$ and N has no parent node in the aB-tree, then end($N.I_i$) $= \infty$
(iii) If $i = j$ and N has a parent node N' such that $N'.p_k$, then end($N.I_i$) = end($N'.I_k$)

We will consider the example of the XML document tree shown in Figure 1, for the rest of this paper. Let A and D be the ancestor and descendant element tags respectively, and p be the attribute to be aggregated in our queries.

3.2 Building the aB-tree

The aB-tree on an ancestor list is created by repeatedly inserting elements from this list. For each ancestor in the list, along with the start value s we also maintain a *pre-computed* aggregate value v. The value v, for a particular ancestor, is the *count* of the descendants or the *sum/min/max* of the descendant attribute values to be aggregated, for that particular ancestor, and is computed directly from the XML document tree. Figure 2 shows the $< s, v >$ list for the A

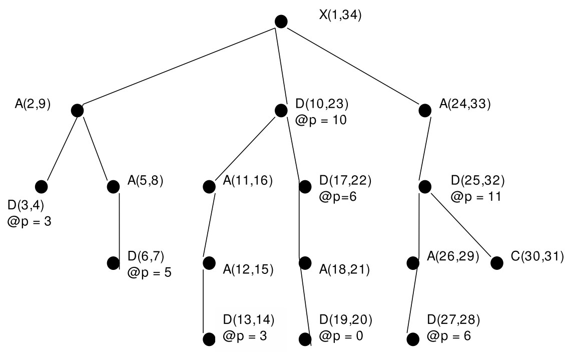


Figure 1: A sample XML document

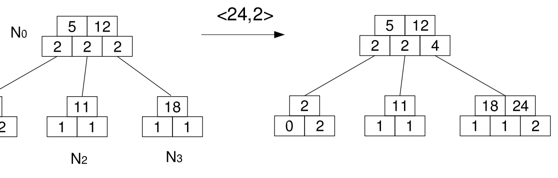
elements in the sample XML tree in Figure 1. In this list, value v corresponds to the Count aggregate, i.e., it counts the number of descendants (elements D) for the ancestor element having start value s. For example, the A element in the XML document with numbering (24,33) has two D elements as descendants, namely (25,32) and (27,28), and hence has $< s, v > = < 24, 2 >$.

The function $insert(N, < s, v >)$ below, inserts the $< s, v >$ pair for an ancestor element into the sub-tree rooted at node N and updates the aggregate values along the path of insertion to reflect the effect of the insertion. The manner in which the update is done depends on the type of aggregation being performed. We use function $agg(x, y)$ to denote the aggregation of values x, y. For example, with Sum and Count, $agg(x, y) = x + y$, while for Min, $agg(x, y) = Min(x, y)$, etc. The procedure for insertion follows:

```
Insert($N, < s, v >$)
  for each interval $N.I_i \in N$
    if ($s$ is contained in $N.I_i$)
      if ($N$ is a leaf)
        (i) The first $i-1$ intervals of $N$ stay the same;
        i.e., $N.k_j$, $N.p_j$ and $N.v_j$ remain unchanged
        for all $j < i$
        (ii) Starting from the $i+1^{th}$ interval, each interval
        is moved one position to the right; hence, $N.k_{j-1}$
        becomes $N.k_j$, $N.p_j$ becomes $N.p_{j+1}$ and $N.v_j$
        becomes $N.v_{j+1}$ for $j > i$
        (iii) $N.k_i = s$ and $N.v_i = v$
        (iv)if overflow occurs, invoke split(N) (defined later)
      else
        (i)$N.v_i$ = agg(v , $N.v_i$)
        (ii)Insert($N.p_i, < s, v >$)
      endif
    endif
  end for
```

Node N is initialized to be the tree root. For any node N, let the i^{th} interval of N contain s, i.e. $N.k_{i-1} < s < N.k_i$.

If node N is a leaf, then entry s is inserted into its correct location (i.e. the i^{th} position) in N. Insertion of s would split the i^{th} interval into two (which will become the new i^{th} and the $i + 1^{th}$ intervals). To make space for the additional interval created we move all old intervals (and their associated values and pointers) starting from the $i + 1^{th}$, to the right and insert the newly created interval. If an overflow occurs due to the additional interval, the split procedure is invoked.



A<s,v>:<2,2> <5,1> <11,1> <12,1> <18,1> <24,2> <26,1>

Figure 2: The aB-trees created using the $< s, v >$ list shown, before and after the insertion of the element $< 24, 2 >$

If node N is not a leaf node, then the partial aggregate value stored in $N.I_i$ is simply updated with v and the insert function is recursively called on its i^{th} child.

Figure 2 shows an example with the insertion of the entry $< 24, 2 >$ into the already existing tree containing all previous entries. We execute $insert(N_2, < 24, 2 >)$ on the tree and find that s=18 is contained in $N_0.I_3 = [12, \infty)$. Since N_0 is not a leaf node, we increment the aggregate value $N_0.v_3$ by 2 and invoke $insert(N_2, < 24, 2 >)$. Now since N_3 is a leaf node, $< 24, 2 >$ gets inserted in its correct location in N_3. The result is shown in the tree on the right.

3.3 Node Splitting

When a new element in inserted into the tree it is possible that the leaf node into which it is inserted is already full. This would require splitting of the leaf node into two nodes both of which must now be children of the same parent, causing a split in the parent interval. If the parent was also already full this splitting of the interval will cause it to overflow. Splitting could hence recursively propagate up the tree. The node splitting algorithm for the aB-tree is similar with the SB-tree [10], however the manner in which the aggregate values are updated when a split occurs is different due to reasons mentioned previously. Formally, suppose that an overflowing node N contains m intervals, where $m = b+1$. In the following, we define the procedure $split(N)$, which reorganizes the tree to deal with the overflow at the node N as follows:

Split(N)

1. Split N into N_1 and N_2 such that:
   (i) N_1 contains the first $\lceil m/2 \rceil$ intervals of N; that is, N_1 contains keys $N.k_1, ..., N.k_{\lceil m/2 \rceil - 1}$, aggregate values $N.v_1, ..., N.v_{\lceil m/2 \rceil}$ and if N is not a leaf, child pointers $N.p_1, ..., N.p_{\lceil m/2 \rceil}$.
   (ii) N_2 contains the remaining intervals of N; that is, N_2 contains $N.k_{\lceil m/2 \rceil + 1}, ..., N.k_m$, aggregate values $N.v_{\lceil m/2 \rceil + 1}, ..., N.v_m$ and if N is not a leaf, child pointers $N.p_{\lceil m/2 \rceil + 1}, ..., N.p_m$.

2. If N is the root node
   (i) Create a new root node N' with two intervals that point to N_1 and N_2. Set $N'.k_1 = N.k_{\lceil m/2 \rceil}$, $N'.p_1 = N_1$, $N'.v_1 = agg(N.v_1, ..., N.v_{\lceil m/2 \rceil})$ and $N'.p_2 = N_2$, $N'.v_2 = agg(N.v_{\lceil m/2 \rceil + 1}, ..., N.v_m)$
   else // N is not the root node,
   (i) suppose N has a parent node N' with $N'.p_j = N$
   (ii) Split the j^{th} interval of N' into two and have them point to N_1 and N_2. Specifically:
   a) The first $j - 1$ intervals of N' stay the same; hence, $N'.k_i$, $N'.p_i$ and $N'.v_i$ remain unchanged for all $i < j$.
   b) Starting from the $j+1^{th}$, each interval is moved one

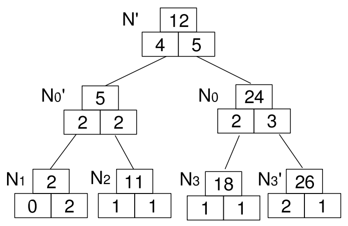


Figure 3: The aB-tree of the $< s, v >$ list in Figure 2 after the insertion of the last pair $< 26, 1 >$ which causes a split that propagates up to the root

position to the right; that is $N'.k_{i-1}$ becomes $N'.k_i$, $N'.p_i$ becomes $N'.p_{i+1}$ and $N'.v_i$ becomes $N'.v_{i+1}$, for all $i > j$.

3. Set $N'.k_j = N.k_{\lceil m/2 \rceil}$, $N'.p_j = N_1$, $N'.p_{j+1} = N_2$ and $N'.v_j = agg(N.v_1, ..., N.v_{\lceil m/2 \rceil})$, $N'.v_{j+1}= agg(N.v_{\lceil m/2 \rceil+1}, ..., N.v_m)$

4. N' overflows, call $split(N')$

where $agg(N.v_1, ..., N.v_{\lceil m/2 \rceil}) = agg(N.v_1, agg(N.v_2, agg(...agg(N.v_{\lceil m/2 \rceil -1}, N.v_{\lceil m/2 \rceil})...)))$. Figure 3 shows an example of a split. We invoke $insert(N_0, < 26, 1 >)$ on the final tree in Figure 2 which recursively calls $insert(N_3, < 26, 1 >)$. Insertion of $< 26, 1 >$ in node N_3 makes it overflow, so we split it into N_3 and N_3' and we also split the third interval of N_0 to accommodate N_3'. This causes N_0 to overflow so we further split N_0 into N_0 and N_0' and create a new root N. N_0 and N_0' become the children of this new root. Figure 3 shows the resultant tree.

Note that deletion can also be supported dynamically on the aB-tree in a way similar to the B-tree deletion with an additional chore of aggregate maintenance. However, we omit the details due to lack of space.

3.4 Range Aggregate Structural Join (RAS_Join)

In this section, we give an algorithm (RAS_Join) that describes how the aB-tree can used to compute aggregates over a range Q. The initial value of the overall aggregate v, is based on the type of aggregation.
For Sum and Count v=0
For Avg v=$< 0, 0 >$
For Min and Max v=null

```
RAS_Join($N, Q, v$)
  for each interval $N.I_i \in$ N
    if ($N.I_i \cap Q \neq \phi$)
      if ($N.I_i \subseteq Q$)
        then $v = agg(v, N.v_i)$                          (1)
      else
        if (N is not a leaf and $N.I_i \cap Q \neq \phi$)
          then RAS_Join($N.p_i, Q, v$)                    (2)
        else if (N is a leaf and $N.k_{i-1}$ is contained in $Q$)
              then $v = agg(v, N.v_i)$                    (3)
      end if
    end if
  end for
```

There are three types of intervals on each level of the aB-tree: Intervals that are completely contained in Q, intervals that intersect Q, and, intervals that are disjoint from Q. The last category is not processed since it does not participate in the aggregation.

For any node N, if the interval $N.I_i$ is completely contained within the queried interval, line 1 in the algorithm aggregates the value of that interval $N.v_i$ with the current aggregate v. Note that the algorithm does not go further down that path because the values of all the elements in the sub-tree below (that must be included in the aggregation, by the containment property) have been accounted for, by the aggregate value $N.v_i$.

Line 2 deals with an internal node N whose interval $N.I_i$ intersects Q but is not completely contained in it. Note that since Q is a continuous range, there can be only two such intervals on each level, one overlapping with the left end of Q and the other with the right end. We thus recursively invoke RAS_Join on the i^{th} child of node N to find and include children that should be part of the aggregate, i.e. children that lie in the overlapping region. As a result, the algorithm traverses down the height of the tree only along the two extreme paths. Any node outside the region bounded by these two paths need not be checked because it cannot have intervals that intersect with Q. Any node within the region bounded by these two paths need not be examined either, because all its values have been accounted for by an ancestor interval that was completely contained in Q.

Line 3 handles the specific case in which the value associated with the rightmost leaf interval of the tree intersecting with the query range is included in the aggregation. Recall that the leaf intervals have values for the individual $A_i s$ associated with them. For example, for the query count(A//D, [17,22]) on the sample document above, the ancestor [18,22] must be part of the aggregate, but would not satisfy the first two *if* conditions since the interval [18,24] in the aB-tree is neither contained [17,22] nor is a non-leaf interval. All leaf intervals other than the rightmost, that should be included in the aggregate, satisfy the condition on line 1 and need not be considered explicitly. As further example, Table 1 shows aggregate values for various sample ranges.

Range	*Count*	*Sum*
$[2, 9]$	3	13
$[1, 34]$	9	42
$[10, 23]$	3	6
$[11, 16]$	2	6
$[24, 33]$	3	23

Table 1: The aggregate values for queries count(A//D,Q) and sum(A//D,p,Q) for sample ranges Q on the tree in Figure 1

3.5 Complexity Analysis

The number of disk accesses performed by both the Insert and the RAS_Join routines, is $O(h)$ for an aB-tree of height h. Insertion, without splitting requires a single pass down the tree; an insertion that causes splitting may require, in the worst case, another pass up to the root. The RAS_Join traverses the tree down the two extreme paths intersecting the edges of the given range. Thus, the I/O complexity for both insertions and aggregate range queries is O($log_b n$) (b is the branching factor and n is the total number of elements in the tree).

The space required is proportional to the number of possible ancestor-descendant combinations i.e. O(t^2) where t is the number of element tags in the XML document. However, in practice, not all tag combinations occur as ancestor-descendant edges in a document. For the purpose of aggregate querying, one can choose to build an aB-tree on those combinations that are more frequent.

4. EXPERIMENTAL RESULTS

We implemented the aB-tree structure and performed experiments to compare its performance with the XA-tree [4]. The experiments were conducted using a Mobile Intel Pentium 1.5 Ghz Processor with 512MB RAM, running Linux. The number of I/Os is used as the performance metric indicator. We used both synthetically generated data as well as real data (the NASA dataset from University of Washington [5]). The majority of the presented results utilize the synthetic datasets because of the flexibility to manipulate parameters and thus examine their effect on the indexing methods. Due to lack of space we only present one experiment with real datasets (fig. 5); nevertheless, all real datasets displayed similar trends.

The synthetic datasets were generated in the form of ancestor and descendant element lists with controlled structural and join characteristics. For our purposes, the nesting level of the generated data is not important as we use semi-processed data with the number of $D_j s$ for each A_i being pre-computed. The sizes of the element lists vary from 1 to 20 MB. The parameter values used are specified within each experiment.

The experiments compared the performance of the (i) Range Aggregate Structural Join (RASJ) Algorithm using the aB-tree data structure and (ii) the TJ-h algorithm using the XA-tree data structure. We also compared against the straightforward approach that first computes a structural join (using an index). Our results show that for the aggregate computation the indexed structural join performs substantially worse. This agrees also with the findings of [4], where the TJ-h algorithm was compared with an indexed structural join using a XR-tree and showed that TJ-h performs significantly better. Hence for brevity, the following figures show only the comparative performance between the aB-tree and the XA-tree. We varied the following parameters for our experiments:

- **Varying the Ancestor Join Ratio (AJR):** The ancestor join ratio is the percentage of ancestor elements in the XML document that contain at least one descendant element. For the purpose of answering aggregate queries on XML documents, only those ancestors are that have at least one descendant element are inserted into the aB-tree and the XA-tree (as those that have no descendant do not contribute to the aggregate). Thus, increasing the AJR increases the number of elements in the trees and consequently the number of pages and the number of I/Os. The following parameters were used for the this experiment: QueryRange=[100,4000000], b=100, Buffer Size=500. Table 2 shows the I/O performance. The aB-tree drastically improves performance. Figure 4 depicts the AJR variation of the two methods in logarithmic scale.

$Ratio$	$No\ of\ tags$	$TJ-h$	$RASJ$
1%	10^4	208	5
10%	10^5	2022	5
30%	$3*10^5$	6076	5
50%	$5*10^5$	10110	5
80%	$8*10^5$	16158	5
100%	10^6	20202	5

Table 2: Ancestor Join Ratio vs I/Os

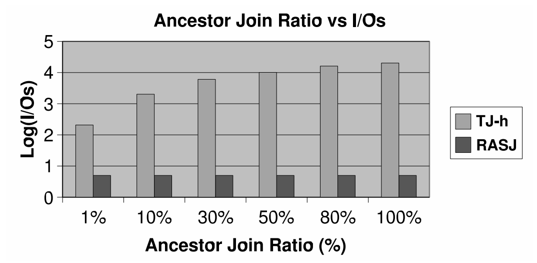


Figure 4: Variation of Ancestor Join Ratio

- **Space Comparison:** Both the aB-tree and the XA-tree use linear space ($O(n)$). In practice, the aB-tree uses less space because an interval can be represented by a single key, while in the XA-tree, an interval stores both start and end values. For the above experiments (i.e., varying the ancestor ratio), the aB-tree space varied as: 0.12, 1.23, 3.69, 6.14, 9.82, 12.23 MBs, while the corresponding values for the XA-tree were: 0.17, 1.62, 4.89, 8.14, 13.02, 16.28 MBs.

- **Varying the Branching Factor(b):** The branching factor b is the maximum number of intervals that can be held by a node in the tree. Reducing b reduces the number of ancestor elements per node (page) and hence increases the number of pages in the tree. The following parameters were used for the this experiment: QueryRange[1,4000000], Buffer Size=10, AJR =100%. Table 3 shows the results for the variation of b.

BF	$TJ-h$	$RASJ$
10	22433	9
15	14347	9
20	10550	7
25	8347	7
50	4161	5
100	2041	5
200	1013	5

Table 3: Branching Factor vs I/Os

- **Varying the Range of the Query:** The range of the query indicates the number of ancestor elements that participate in the aggregation. The following parameters were used for this experiment: b=25, Buffer Size=400, AJR=100%. The results for varying query ranges appear in Table 4.

As is seen from the experimental results with synthetic datasets, the RASJ Algorithm drastically out-performs the TJ-h join algorithm in all cases. *The fundamental reason is that the Range Aggregate Query Algorithm is only required to traverse the height of the aB-tree to perform the aggregation whereas the TJ-h algorithm needs to visit each node in the XA-tree that participates in the aggregation.* Moreover, the

$QueryRange$	$TJ-h$	$RASJ$
$[1-4,000,000]$	8346	4
$[3000-3,000,000]$	6272	7
$[200,000-2,000,000]$	4138	7
$[100,000-1,000,000]$	1864	7
$[500,000-900,000]$	833	7
$[799,000-800,000]$	9	5
$[799,990-800,000]$	2	4

Table 4: Query Range vs I/Os

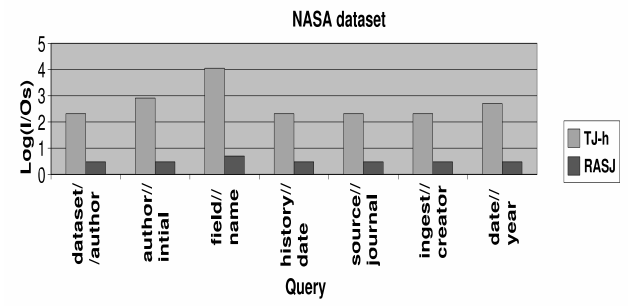


Figure 5: Various sample queries using the NASA dataset

performance of RASJ remains almost unaffected, (i) as the number of nodes in the index trees increase (either due to the increase in the ancestor join ratio or due to a decrease in the branching factor), or, (ii) as the number of ancestor elements participating in the aggregation increases (due to increase in the query range). In contrast the I/Os required by TJ-h join algorithm increases significantly in all these cases. This is because our approach depends on the height of the index tree (which increases only as a logarithm of the number of ancestors) whereas the TJ-h algorithm depends linearly on the number of ancestor nodes.

The only case where the performance of TJ-h is comparable or better than RASJ is for very small query ranges (see Table 4). This is because, for small query ranges the number of nodes to be checked by TJ-h is comparable to the height of the tree and thus the I/Os are similar. For the special case where no a//d pair exists in the given query range, TJ-h could perform better than RASJ because it may detect that no a//d pair will be found before reaching the leaf level (since the intervals in its nodes are not continuous and leave out portions that have no a//d pairs). Instead, RASJ would have to go to the leaf level for queries ranges whose start/end keys are not found in the tree. However, because RASJ is only a traversal down the height of the tree along the two extreme paths, the difference even in such infrequent cases, will not be significant, making RASJ a much more robust solution.

Finally, Figure 5 presents experimental results over a sample NASA real dataset. The dataset was 23MB with 476646 elements and a maximum depth of 8. The element lists were generated by parsing the document and using the previously described numbering scheme. We experimented with different combinations of ancestor descendant pairs. The results agree with our findings from the synthetic datasets, i.e., the RASJ performs again significantly better.

5. CONCLUSIONS AND FUTURE WORK

In this paper we examined how the evaluation of aggregate structural joins can be expedited by the use of an index. We proposed the Aggregation B-tree, which is a balanced index built on the start values of the ancestor elements, especially designed to address aggregate range queries. Each pair of consecutive start values in the aB-tree is treated as an interval and stores a pre-computed partially aggregated value that accounts for the aggregate values of the sub-tree rooted at it. We devised an algorithm (RASJ) that exploits these pre-computed partial aggregates to obtain results for range aggregate queries in *O(log n)* time, where *n* is the number of ancestor elements in the aB-tree. An experimental evaluation showed drastic improvements over the previous approaches. An advantage of our approach is that in addition to the aggregate computations, the aB-tree can also be used as an index to expedite traditional structural joins.

As future work we plan to explore how our technique can be extended to handle aggregate queries over path and twig joins.

Acknowledgements: We would like to thank Marios Hadjieleftheriou for providing us with his implementation of the SB-tree and Zografoula Vagena for her XML data generator and many helpful comments.

6. REFERENCES

[1] Z. Chen, H. Jagadish, F. Korn, N. Koudas, S. Muthukrishnan, N. R, and D. Srivastava. Counting twig matches in a tree. In *17th International Conference on Data Engineering*, April 2001.

[2] S. Chien, Z. Vagena, D. Zhang, V. Tsotras, and C. Zaniolo. Efficient structural joins on indexed xml documents. In *VLDB*, pages 263–274, 2002.

[3] H.Jiang, H.Lu, W.Wang, and B.C.Ooi. Xr-tree: Indexing xml data for efficient structural joins. In *Proceedings of ICDE*, pages 253–263, 2003.

[4] K. Liu and F. Lochovsky. Efficient computation of aggregate structural joins. In *Proceedings of the 4th International Conference on Web Information Systems Engineering*, December 2003.

[5] U. of Washington. *http://www.cs.washington.edu/research/xmldatasets/*.

[6] N. Polyzotis, M. Garofalakis, and Y. Ioannidis. Selectivity estimation for xml twigs. In *20th International Conference on Data Engineering*, Apr 2004.

[7] C. Sartiani. A framework for estimating xml query cardinality. In *WebDB*, June 2003.

[8] D. Srivastava, S. Al-Khalifa, H. Jagadish, N. Koudas, J. Patel, and Y. Wu. Structural joins: A primitive for effcient xml query pattern matching. In *ICDE Proceedings*, pages 141–152, 2002.

[9] I. Tatarinov, E. Viglas, K. Beyer, J. Shanmugasundaram, and E. Shekita. Storing and querying ordered xml using a relational database system. In *SIGMOD Conference*, June 2002.

[10] J. Yang and J. Widom. Incremental computation of temporal aggregates. In *VLDB*, pages 262–283, October 2003.

[11] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman. On supporting containment queries in relational database management systems. In *SIGMOD Conference*, May 2001.

An Evaluation and Comparison of Current Peer-to-Peer Full-Text Keyword Search Techniques[*]

Ming Zhong Justin Moore Kai Shen
Department of Computer Science
University of Rochester
Rochester, NY 14627, USA
{zhong,jmoore,kshen}@cs.rochester.edu

Amy L. Murphy
School of Informatics
University of Lugano
Lugano, CH-6904, Switzerland
amy.murphy@unisi.ch

ABSTRACT

Current peer-to-peer (p2p) full-text keyword search techniques fall into the following categories: document-based partitioning, keyword-based partitioning, hybrid indexing, and semantic search. This paper provides a performance evaluation and comparison of these p2p full-text keyword search techniques on a dataset with 3.7 million web pages and 6.8 million search queries. Our evaluation results can serve as a guide for choosing the most suitable p2p full-text keyword search technique based on given system parameters, such as network size, the number of documents, and the number of queries per second.

1. INTRODUCTION

The capability to locate desired documents using full-text keyword search is essential for large-scale p2p networks. Centralized search engines can be employed in p2p networks and provide look-up service. Although these systems may provide a high level of scalability and availability, a p2p keyword search system may be preferable due to its robustness, low maintenance cost, and data freshness.

A large number of p2p keyword search systems have been proposed, including those using document-based partitioning [10, 11], keyword-based partitioning [9, 12, 16, 21], hybrid indexing [23], and semantic search [8, 13, 24]. However, there is still no comprehensive understanding on the tradeoffs between these four types of techniques under different system environments. In this paper, we provide an evaluation and comparison of existing p2p keyword search techniques on 3.7 million web pages and 6.8 million real web queries. To further project the performance of current p2p keyword search techniques on very large datasets, we linearly scale our evaluation results to 10^9 web pages. Our results suggest that there is no absolute best choice among current p2p keyword search techniques. More importantly, our results can serve as a guide for a user to make her choice based on specific system parameters, such as network size, the number of documents, and the query throughput.

Most current performance evaluation results for p2p keyword search systems [8, 9, 13, 16, 21, 23, 24] are based on datasets with less than 530,000 web pages and 100,000 queries, which are an order of magnitude smaller than our datasets. The only exception we are aware of is Li *et al.*'s work [12], which uses 1.7 million web pages and 81,000 queries to evaluate the feasibility of keyword-based partitioning. However, they did not give specific evaluation results on the communication cost and search latency on their datasets. In addition, there is no previous performance comparison for all four types of existing p2p keyword search techniques on the same dataset.

The remainder of this paper is organized as follows. Section 2 provides an overview of our performance evaluation framework and technical background. Sections 3 to 6 evaluate each of the four types of p2p keyword search techniques and explore directions to improve the search quality or to reduce the overhead of current p2p keyword search systems. Section 7 compares current p2p keyword search techniques by scaling our simulation results to 10^9 web pages and Section 8 concludes this paper.

2. EVALUATION SETUP

In our evaluation framework, a search finds the page IDs of several (*e.g.*, 20) most relevant web pages since most users are only interested in the most relevant web pages. A complete search system may also retrieve the page URLs and digests based on the page IDs found. This step could be efficiently supported by using any distributed hash table (*e.g.*, Chord [20] or CAN [15]) and we do not examine that in our performance evaluation.

Our evaluation dataset contains 3.7 million web pages and 6.8 million web queries. The web pages are crawled based on URL listings of the Open Directory Project [6]. The queries are from a partial query log at the Ask Jeeves search engine [1] over the week of 01/06/2002–01/12/2002 and there are an average of 2.54 terms per query. The web pages are pre-processed by using the stopword list of the SMART software package [19] and removing the HTML tags. In addition, we restrict the vocabulary to be the 253,334 words that appear in our query log. After preprocessing, the average number of distinct words per page is approximately 114. In our implementation, each web page is associated with an 8-byte page ID, which is computed by using its URL as a key to the MD5 hash algorithm. Each page ID in the inverted list of term A is associated with its term frequency of A (the number of occurrences of A in the page), which is stored as a short integer (2 bytes). Thus each entry of an inverted list is 10 bytes.

We evaluate the performance of p2p keyword search techniques in terms of four metrics: *total storage consumption*, *communication cost*, *search latency*, and *search quality*. Here

[*]This work was supported in part by NSF grants CCR-0306473, ITR/IIS-0312925, and NSF CAREER Award CCF-0448413.

communication cost measures the average number of bytes that needs to be sent over the network in order to return the top 20 most relevant page IDs for a query. Search latency is the time for a p2p keyword search system to return the top 20 most relevant results. Search quality is defined as the overlapping percentage between the search results of a centralized keyword search and those of a p2p keyword search. Ideally, p2p keyword search should return exactly the same result as the centralized keyword search with moderate search latency and communication cost.

In order to estimate search latency based on the communication cost, we make the following assumptions. We assume that the latency for each link in the p2p overlay is 40 ms and the maximum Internet bandwidth consumption of a p2p keyword search system is 1 Gbps, which is approximately 0.26% of the US Internet backbone bandwidth in 2002 (381.90 Gbps as reported by TeleGeography [25]). We assume that the maximum available network bandwidth per query is 1.5 Mbps — the bandwidth of a T1 link.

We use the Vector Space Model (VSM) to rank the relevance of web pages to a query. In VSM [3], the similarity between two pages is measured by the inner product of their corresponding page vectors, which is typically computed by using variants of the TF.IDF term weighting scheme [18]. We are aware that some term weighting schemes, such as Okapi [17], are reported to have better performance than the standard term weighting scheme. However, it is *not* our goal to explore the performance of centralized keyword search systems with different term weighting schemes.

3. DOCUMENT-BASED PARTITIONING

In document-based partitioning, the web pages are divided among the nodes, and each node maintains local inverted lists of the web pages it has been assigned. A query is broadcast to all nodes, and each node returns the k most highly ranked web pages in its local inverted lists.

In our evaluation, the web pages are randomly distributed among the overlay nodes. Assuming the availability of an overlay multicast management protocol [2], the query broadcast and result aggregation are done through a pre-constructed broadcast/aggregation tree with depth $\log n$ for a network with n nodes. Only the top 20 most highly ranked pages from each node are considered. Each node in the aggregation tree merges its local query result with the results from its children and returns the top 20 pages to its parent. Thus the size of the results returned from each node is constant.

According to the VSM page ranking algorithm, the computation of term weights requires some global statistics (*e.g.*, the popularity of terms), which can only be estimated locally based on the partition at each node. Therefore, the query results of document-based partitioning may be different from the results of the centralized search, which is based on accurate global statistics. We evaluate the quality degradation of document-based partitioning using our dataset. Figure 1(A) presents the results on networks with different sizes.

The total storage consumption of document-based partitioning is $d \cdot W \cdot i$, where d is the total number of web pages, W is the average number of distinct terms per page, and i is size of an inverted list entry. For our dataset, $d = 3720390$, $W = 114$, $i = 10$ bytes. Therefore, the total storage requirement of document-based partitioning is $3720390 \times 114 \times 10 \approx 4.24$ GB.

A message of query results (containing 20 page IDs and their relevance scores) has $20 \times 10 = 200$ bytes. Assume each message has an additional overhead of 100 bytes. Thus the total communication cost for a query is $300 \times (n-1)$ bytes, which grows linearly with the network size.

The search latency of document-based partitioning is dominated by the network broadcast and aggregation time under the assumption that the local search at each node and the merging of search results can be done efficiently (otherwise no efficient p2p keyword search could be possible at all). The network broadcast and aggregation time is $2 \times \log n \times 0.04$ seconds since the tree depth is $\log n$.

4. KEYWORD-BASED PARTITIONING

For keyword-based partitioning, each node maintains the complete inverted lists of some keywords. A query with $k \geq 1$ keywords needs to contact k nodes and requires that the inverted lists of $k-1$ keywords be sent over the network.

The baseline keyword-based partitioning randomly distributes the inverted lists of keywords over network nodes and always sends the smallest inverted list over the network when computing the intersection of inverted lists. Hence a k-word query visits k nodes sequentially in the ascending order of the inverted list sizes, which aims to minimize the network communication overhead of the set intersections.

Unlike document-based partitioning, there is *no* quality degradation when using keyword-based partitioning. The total storage consumption of the baseline keyword-based partitioning is identical to that of document-based partitioning, though some optimization techniques (*e.g.* caching, pre-computation) may lead to extra storage consumption.

In the evaluation of keyword-based partitioning, we use a Chord [20] ring to organize nodes into an overlay, where the inverted list of a term x is stored in the overlay by using x as a hash key. Given a query term A, the inverted list of A can be found within $\log n \times 0.04$ seconds in a Chord ring with n nodes since the network diameter is $\log n$ and the average latency per link is 40 ms in our settings.

For a k-keyword query, the search latency T is as follows.

$$T = T_{linkLatency} + T_{transmission} \tag{1}$$

$T_{linkLatency} \leq (k+1) \times \log n \times 0.04$ seconds is the total network link latency for a k-keyword query since a k-keyword query needs to go through $k+1$ node-to-node trips and each trip takes at most $\log n \times 0.04$ seconds in a Chord with n nodes. $T_{transmission}$, the time to send the inverted lists over the network, is $\frac{C}{B}$, where C is the communication cost of the query and B = 1.5 Mbps is the available network bandwidth per query in our evaluation settings. For very large datasets, $T_{transmission}$ becomes the dominant factor of the search latency of keyword-based partitioning. Note that we do not consider the local computation time since it is usually small and negligible compared with $T_{linkLatency}$ and $T_{transmission}$.

Figure 1(B) shows the distribution of the communication cost per query for the baseline keyword-based partitioning, where the average communication cost per query is 96.61 KB and the maximum cost is 18.65 MB. Given that the average number of terms per query is 2.54, the average search latency of the baseline keyword-based partitioning can be computed based on Equation (1):

$$\begin{aligned} T &= ((2.54+1) \times \log n \times 0.04) + \left(\frac{96.61 \times 1000 \times 8}{1.5 \times 10^6}\right) \\ &= (0.14 \times \log n) + (0.52) \text{ sec.} \end{aligned} \tag{2}$$

The communication cost of the baseline keyword-based partitioning can be reduced by the following techniques without compromising the quality: Bloom filters, pre-computation,

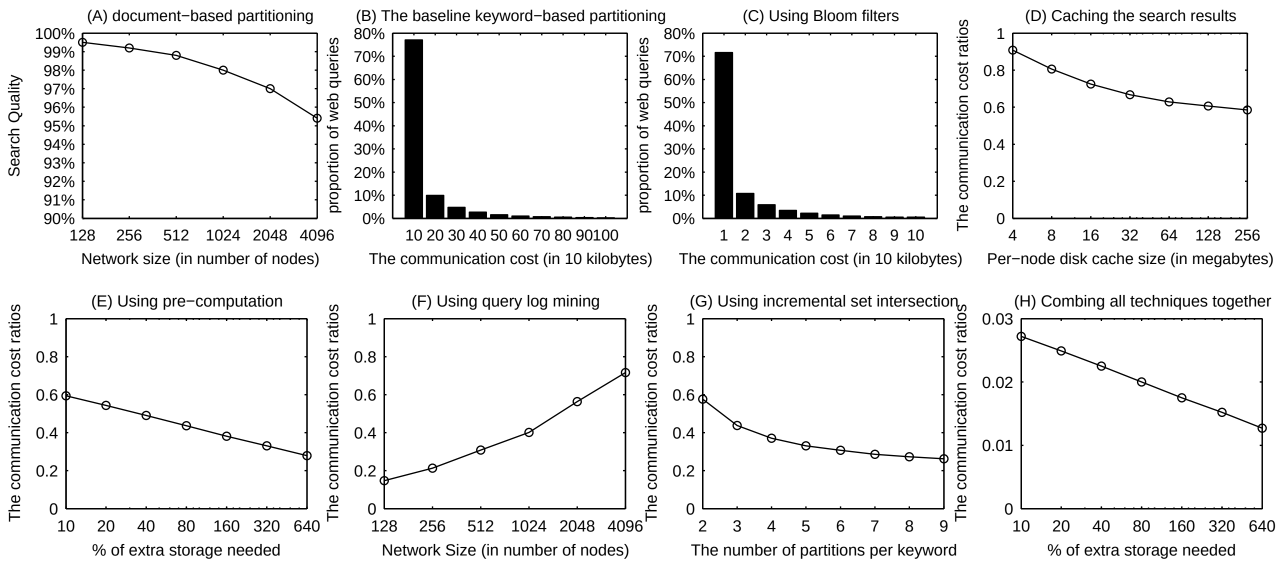


Figure 1: The evaluation results for document-based partitioning and keyword-based partitioning.

caching, query log mining, incremental set intersection, and other techniques. We discuss these techniques in the remainder of this section. During our discussion, the *communication cost ratio*, c, is defined as the average ratio of reduced communication cost to the communication cost of the baseline keyword-based partitioning. Hence the new network transmission time $T_{transmission}$ is c times that of the baseline keyword-based partitioning and $T_{linkLatency}$ remains almost unchanged. Based on Equation(2), we have the new search latency:

$$T(c) = (0.14 \times \log n) + (0.52 \times c) \text{ sec.} \tag{3}$$

4.1 Bloom Filters

Li *et al.* [12] and Reynold and Vahdat [16] suggest to use *Bloom filters* [4] as a compact representation of sets. By using Bloom filters, the communication overhead for set intersections is significantly reduced at the cost of a small probability of false positives. Given two sets A, B with $|A| < |B|$ and each element in the sets having i bytes. The number of bits, m, in a Bloom filter that minimizes the communication cost for computing $A \bigcap B$ can be determined by the following equation from Reynold and Vahdat [16].

$$m = |A| \cdot \log_{0.6185}\left(\frac{2.081}{i} \cdot \frac{|A|}{|B|}\right) \tag{4}$$

We implemented Bloom filters on our dataset based on Equation (4), where $i = 64$ for our system. Figure 1(C) shows the distribution of per-query communication cost of our implementation of Bloom filters, where the communication ratio is 0.137. Hence the search latency is reduced to $(0.14 \times \log n) + (0.137 \times 0.52)$ seconds according to Equation (3).

4.2 Caching

Previous research [12, 16] has suggested that the communication cost for set intersections can be reduced by *caching* the sets or their Bloom filters received at each node. Our experiments show that it is more helpful for each node to directly cache its search results. LFU policy is used in our cache implementation. Figure 1(D) presents the communication cost ratio of our cache implementation with different cache sizes. The new search latency can be easily calculated based on Equation (3).

4.3 Pre-Computation

Gnawali as well as others [9, 12] suggest to use *pre-computation*, which computes and stores the intersection results of the inverted lists of popular query keywords in advance. Here we pre-compute the intersections of the most frequently used keyword pairs, keyword triplets, and keyword quartets in the query log. Figure 1(E) illustrates how pre-computation saves communication cost at the expense of extra storage consumption.

4.4 Query Log Mining

We propose to use *query log mining*, which explores a better way to distribute inverted lists over the nodes than the uniformly random scheme used by the baseline keyword-based partitioning. Our query log mining clusters keywords into similar-sized groups based on their correlations. By distributing each group of keywords to a node, the intersection of the inverted lists of keywords within the same group does not incur any network communication.

We represent our query log as a weighted graph, where each node represents a query term and the weight of an edge (u, v) is the number of queries that contain both u, v. By using the chaco [5] package recursively, the graph is partitioned into groups with nearly balanced storage consumption such that the words in the same group tend to be highly correlated. A sampled group on a 4096-node network includes the following words: san ca diego francisco puerto tx austin rico antonio earthquake jose juan vallarta lucas rican luis cabo fransisco bernardino.

Figure 1(F) illustrates how our query log mining results help to reduce the communication cost ratios for networks with different number of nodes (keyword groups).

4.5 Other Techniques

Based on the assumption that users are only interested in the most relevant results of a search, *incremental set intersection* reduces the communication cost by only retrieving the top k most relevant web pages. Variants of Fagin's algorithm [7] has been used in some p2p keyword search systems [21] to achieve *incremental set intersection*.

Figure 1(G) presents the communication cost ratios when different values of the number of partitions per keyword, are used for our dataset.

As suggested by Gnawali and Li *et al.* [9, 12], other compression methods, such as compressed Bloom filters [14] and gap compression [26], may also be used to reduce the communication cost. However, these methods only lead to slight improvement in our experiment.

4.6 Combine Them Together

Figure 1(H) presents the communication cost ratios after using Bloom filters, pre-computation, caching (256 MB per node), query log mining, and incremental set intersection (with 3 partitions per keyword) together on 4096-node networks. The ratios in Figure 1(H) are larger than the product of the ratios in Figure 1(C) to 1(G) since the performances of these techniques are not completely orthogonal. Given the communication cost ratios, the search latency can be easily calculated based on Equation (3).

5. HYBRID INDEXING

Hybrid indexing [23] saves the communication cost of keyword-based partitioning by associating each page ID in an inverted list with some *metadata* of the corresponding web page.

A naive approach is to associate each page ID in an inverted list with a complete term list of the corresponding web page. This way the intersection of the inverted lists for multiple keyword search can be done locally with no communication needed. Let L be the average size of the term lists of web pages. Let l be the average size of the entries in the original inverted lists. The above naive approach requires $\frac{L}{l} + 1$ times the storage consumption of the original inverted lists, which may be as high as several hundreds and thus is prohibitive for very large datasets.

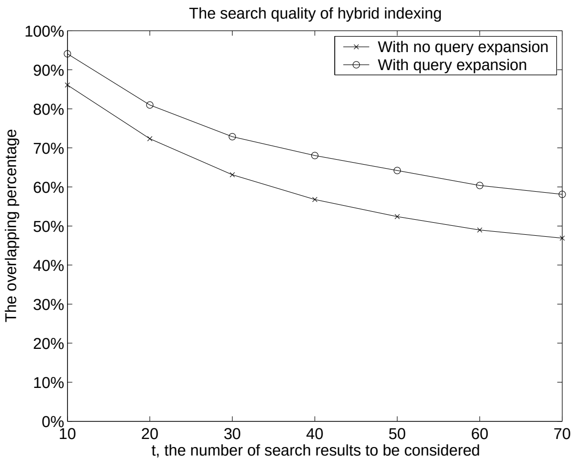


Figure 2: The search quality of hybrid indexing on our dataset. The quality is measured by the overlapping percentage between the top t search results of the hybrid indexing and those of centralized search.

The hybrid indexing approach proposed by Tang *et al.* [23] uses VSM [3] to identify the top k terms (with k highest term weights) in a document and only publishes the term list of the document to the inverted lists of these top terms. Briefly speaking, the inverted list of a term, x, only contains those page IDs that have x as their top terms. This approach may degrade the quality of the search results since if the terms of a query are not among the top terms of a document then the query cannot find this document. In their approach, classical IR algorithms and query expansion are used to improve search quality. Query expansion works by expanding the query with new relevant terms extracted from the best matched pages to the original query. For the details of query expansion, please refer to [23].

The total storage consumption of hybrid indexing is $1 + \frac{k}{W} \cdot \frac{L}{l}$ times that of the standard keyword-based partitioning. Here L is the average size of the term lists of web pages. l is the average size of the entries in an inverted list. k is the number of top terms under which a web page is published. W is the the average number of distinct terms per page. Let $k = 20$ and each entry of a term list consists of a 4-byte term ID and a 2-byte term frequency. Hence the total storage consumption of hybrid indexing on our dataset is $1 + \frac{k}{W} \cdot \frac{L}{l} = 1 + \frac{20}{114} \times \frac{6 \times 114}{10} = 13$ times that of the baseline keyword-based partitioning.

The communication cost of hybrid indexing on our dataset is 7.5 KB per query, which is independent of the dataset size.

For hybrid indexing, distributed hash tables (DHT) are necessary for storing and finding the inverted list of a term x by using x as a hash key. Let D denote the network diameter of the underlying DHT. If query expansion is not used, then a hybrid indexing search contacts the inverted list of a query term (or all inverted lists of query terms in parallel) and retrieve the search results. Hence the search latency of hybrid indexing is $2 \times D \times 0.04$ seconds. If query expansion is used, then a hybrid indexing search consists of two searches: one for the original query and the other for the expanded query. Hence the search latency is $4 \times D \times 0.04$ seconds if query expansion is used.

Figure 2 studies the search quality of hybrid indexing, where each web page is published under its top 20 terms. As suggested by Tang *et al.* [23], the query expansion in Figure 2 is based on the 10 most relevant terms in the top 10 best matched pages to the original query.

6. SEMANTIC SEARCH

Semantic search [8, 13, 24] use classical IR algorithms (*e.g.*, Latent Semantic Indexing) to map web pages and queries to points in a semantic space with reduced dimensionality (typically between 50 and 350). For each query, semantic search returns the top a few *closest* points to the query point in the multi-dimensional semantic space, where the *closeness* between points A, B is typically measured by the dot product of vectors $\vec{A}, \vec{B}$. As a result, semantic search can be characterized as *nearest neighbor search* in multi-dimensional semantic space.

Here we evaluate the performance of pSearch, the semantic search system proposed by Tang *et al.* [24] on our dataset. In our evaluation, we use LSI matrices with dimensionality 200, which are computed by applying the SVDPACK software package [22] to a term-document matrix that consists of 38457 web pages uniformly sampled from our dataset. The LSI matrices fold web pages or queries (253334-dimensional vectors) into 200-dimensional vectors, which form a semantic space. In our evaluation, the storage consumption of a web page in the semantic space is $8 + (200 \times 4) = 808$ bytes since each 200-dimensional vector (200×4 bytes) is associated with its corresponding page ID (8 bytes).

One of the key obstacles to semantic search is the mis-

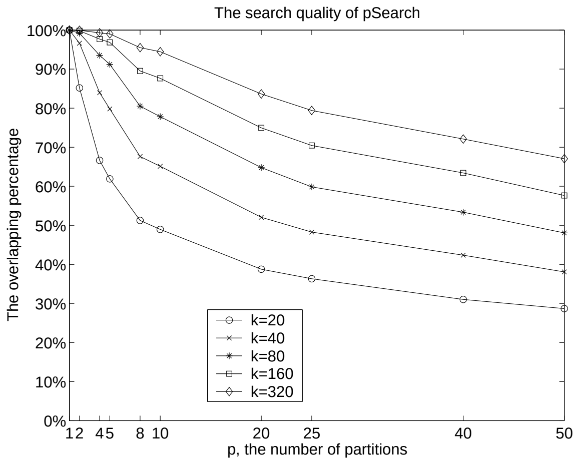


Figure 3: The quality of pSearch on our dataset. The quality is measured by the overlapping percentage between the top 20 search results of pSearch and the top 20 results of centralized semantic search.

match between the dimensionality of the semantic space (50 to 350) and the effective dimensionality of the p2p overlay hash space, which is at most $2.3 \ln n$ for a CAN network with n nodes [24]. The dimension reduction technique used in pSearch is called *rolling-index*, which partitions a semantic space into p subspaces (each has m dimensions). Hence $p \times m$ is equal to the dimensionality of the semantic space, which is 200 in our evaluation. Another constraint is that m has to be less than or equal to the effective dimensionality of the underlying CAN network. Otherwise a m-dimensional subspace cannot be efficiently implemented on the CAN network hash space. Given that the maximum effective dimensionality of a CAN network with n nodes is $\lfloor 2.3 \ln n \rfloor$ [24], we have $m \leq \lfloor 2.3 \ln n \rfloor$ and $p \geq \lceil \frac{200}{2.3 \ln n} \rceil$.

In rolling-indexing, each point is stored at p places (one for each semantic subspace) in the CAN hash space. For each query, p parallel searches are performed (one for each subspace) and each of them returns the top k points (and their page IDs) found in its corresponding semantic subspace. The query-initiating node merges the results and only returns the top 20 page IDs as the final result.

As we can see from Figure 3, the search quality increases when k (the number of the retrieved points from each semantic subspace) grows. The search quality increases when p (the number of semantic subspaces) gets smaller since a small p means that each subspace has high dimensionality and their closest points to the query are more likely to be among the global closest points. However, p has a lower bound of $\lceil \frac{200}{2.3 \ln n} \rceil$ as we explained before. Specifically, p must be at least 10 for a 6000-node CAN, which leads to 87.64% search quality if 160 points are retrieved from each semantic subspace.

Let i denote the number of bytes needed for storing a point in each semantic subspace. Let d denote the total number of documents in the system. The total storage consumption of pSearch is $d \cdot p \cdot i$. If $p = 10$ (for 6000-node CAN networks), then the total storage requirement of pSearch on our dataset is $3720390 \times 10 \times 808 \approx 30.06\,\text{GB}$, which is 7.09 times the storage requirement of document-based partitioning or the baseline keyword-based partitioning.

The communication cost of pSearch is $p \cdot k \cdot i$ bytes, which is independent of the dataset size. If we choose $p = 10$ and $k =$ 160 (lead to 87.64% search quality), then the communication cost per query is $10 \times 160 \times 808 \approx 1.29\,\text{MB}$.

According to Equation (1), the search latency of pSearch on a CAN network with n nodes is

$$T = T_{linkLatency} + T_{transmission} = (2 \times n^{\frac{1}{d}} \times 0.04) + \frac{p \cdot k \cdot i \cdot 8}{1.5 \times 10^6}\, sec.$$

where d is $\lfloor 2.3 \ln n \rfloor$, $p = \lceil \frac{200}{2.3 \ln n} \rceil$, and $i = 808$. Note that k, the number of the retrieved points from each semantic subspace, is decided based on the desired search quality since the quality increases as k grows.

7. PERFORMANCE COMPARISON

In order to project the performance of current p2p keyword search techniques on very large datasets, we scale our evaluation results to 10^9 web pages as shown in Table 1.

Table 2 summarizes the advantages and constraints of the four types of p2p full-text keyword search techniques that we considered in this paper.

Document-based partitioning is desirable for a large set of documents since its communication cost is independent of the dataset size and its storage consumption is small compared with other p2p keyword search techniques. However, document-based partitioning requires that the network size and the total number of queries per second must be small. For example, if we assume that each node can handle up to 100 queries per second and the assumptions in Section 2 hold, then document-based partitioning can support up to 4167 nodes and 100 queries per second in total. Generally speaking, the communication cost of document-based partitioning grows linearly with the network size. The number of queries received by each node per second is exactly the number of queries going into the whole system each second since document-based partitioning broadcasts each query to every node. Hence the query throughput of document-based partitioning is bounded by the query throughput of each node.

Keyword-based partitioning (optimized as described in Section 4) is the only known p2p keyword search technique with no quality degradation. Keyword-based partitioning is suitable for large-sized networks since the communication cost of keyword-based partitioning is independent of the network size. However, the communication cost of keyword-based partitioning grows linearly with the number of documents in the system. Hence a user should choose keyword-based partitioning when she prefers no quality degradation or when the total number of documents in the system is not too large. Specifically, if we require that the total network bandwidth consumption is bounded by 1 Gbps and the average search latency is less than 10 seconds, then keyword-based partitioning can support up to 10^8 web pages and 1000 queries per second in total.

Hybrid indexing has small communication cost per query (7.5 KB), which is independent of the total number of documents and network size. This small per-query communication cost is also very helpful when a large number of queries go into the system each second. However, these advantages are achieved at the cost of 10%–50% quality degradation and significant extra storage consumption (13 times under our settings). When quality degradation and extra storage consumption are acceptable, hybrid indexing is a good choice.

Semantic search favors large-sized networks because large-scale networks have high effective dimensionalities and thus lead to small dimension mismatch between the semantic space and the overlay hash space. For instance, for a 6000-

Techniques	Total storage	C, comm. cost per query	Latency	Quality
Document-based partitioning	1139.67 GB	$0.3 \times (n-1)$ KB	$2 \times \log n \times 0.04$ seconds	75% to 95% (varies with n)
Keyword-based partitioning (160% extra precomputation storage and 256 MB per-node cache are used)	2963.10 GB	905.82 KB to 1221.78 KB (varies with n)	$(0.14 \times \log n) + 4.82$ seconds to $(0.14 \times \log n) + 6.52$ seconds	100.00%
Hybrid Indexing (with query expansion and the top 20 results in consideration)	14815.70 GB	7.5 KB (independent of n)	$4 \times \log n \times 0.04$ seconds	86.05%
Semantic Search ($n = 6000, k = 160$)	8080.26 GB	1290 KB	7.59 seconds	87.64%

Table 1: The scaled performance of p2p full-text keyword search techniques on a n-node network with 10^9 pages.

Techniques	Advantages	Constraints
Document-based partitioning	1. Suitable for a large number of documents 2. Relatively small storage consumption	1. Requires small network size 2. Requires small number of queries per second in total 3. Has moderate quality degradation
Keyword-based partitioning	1. No quality degradation 2. Suitable for large-sized networks	1. The communication cost grows linearly with the total number of documents 2. Relatively high communication cost 3. Requires moderate extra storage consumption compared with document-based partitioning
Hybrid Indexing	1. Suitable for large-sized networks 2. Suitable for a large number of documents 3. Suitable for a large number of queries per second	1. Has quality degradation 2. Requires significant extra storage consumption than document-based partitioning
Semantic Search	1. Favors large-sized networks 2. Can do concept-based search	1. Requires moderate extra storage than document-based partitioning 2. Has moderate quality degradation 3. Relatively high communication cost 4. Its underlying IR techniques, *e.g.*, LSI, may have scalability problems

Table 2: The advantages and constraints of current p2p full-text keyword search techniques.

node network, semantic search has 87.64% quality with 1.29 MB communication cost per query and 8080 GB total storage consumption. When the network size grows to 36 million nodes, semantic search can achieve 91.22% quality with 322.50 KB communication cost per query and 4040 GB total storage consumption. In addition, semantic search can find those web pages with similar concepts to the query terms, though they may not have exactly the same term. For example, semantic search can retrieve documents containing the term "automobiles" for queries containing the term "cars". In summary, semantic search is suitable for large-scale networks or concept-based queries.

8. CONCLUSION

This paper provides a performance evaluation and comparison of current p2p full-text keyword search techniques on a dataset of 3.7 million web pages and 6.8 million queries. Our dataset is an order of magnitude larger than the datasets employed in most previous studies (up to 528,543 web pages and 100,000 queries). To further project the performance of current p2p keyword search techniques on very large datasets, we linearly scale our evaluation results to 10^9 web pages. Our evaluation results can serve as a guide for a user to choose p2p keyword search techniques based on specific system parameters, such as network size, the number of documents, and the query throughput.

9. REFERENCES

[1] Ask Jeeves Search. http://www.ask.com.

[2] S. Banerjee, B. Bhattacharjee, and C. Kommareddy. Scalable Application Layer Multicast. In *Proc. of ACM SIGCOMM*, pages 205–217, 2002.

[3] M. Berry, Z. Drmac, and E. R. Jessup. Matrices, Vector Spaces, and Information Retrieval. *SIAM Review*, 41(2):335–362, 1999.

[4] B. H. Bloom. Space/time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, 1970.

[5] http://www.cs.sandia.gov/ bahendr/chaco.html.

[6] The Open Directory Project. http://www.dmoz.com.

[7] R. Fagin, A. Lotem, and M. Naor. Optimal Aggregation Algorithms for Middleware. In *Proc. of ACM Symp. on Principles of Database Systems*, 2001.

[8] P. Ganesan, B. Yang, and H. Garcia-Molina. One Torus to Rule Them All: Multidimensional Queries in P2P Systems. In *Proc. of WebDB'04*, 2004.

[9] O. D. Gnawali. A Keyword-Set Search System for Peer-to-Peer Networks. Master's thesis, Dept. of Computer Science, Massachusetts Institute of Technology, June 2002.

[10] Gnutella. http://www.gnutella.com.

[11] KaZaA. http://kazaa.com.

[12] J. Li, T. Loo, J. Hellerstein, F. Kaashoek, D. Karger, and R. Morris. On the Feasibility of Peer-to-Peer Web Indexing and Search. In *Proc. of IPTPS'03*, 2003.

[13] M. Li, W. Lee, and A. Sivasubramaniam. Semantic Small World: An Overlay Network for Peer-to-Peer Search. In *Proc. of IEEE ICNP*, 2004.

[14] M. Mitzenmacher. Compressed Bloom Filters. In *Proc. of 20th ACM PODC*, 2001.

[15] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Topologically-Aware Overlay Construction and Server Selection. In *Proc. of IEEE INFOCOM*, New York, NY, June 2002.

[16] P. Reynold and A. Vahdat. Efficient Peer-to-Peer Keyword Searching. In *Proc. of Middleware'03*, 2003.

[17] S. E. Robertson, S. Walker, S. Jones, M. HancockBeaulieu, and M. Gatford. Okapi at TREC-3. In *TREC-3*, 1994.

[18] G. Salton and C. Buckley. Term-weighting Approaches in Automatic Text Retrieval. *Information Processing and Management*, 24(5):513–523, 1988.

[19] SMART. ftp://ftp.cs.cornell.edu/pub/smart.

[20] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proc. of ACM SIGCOMM*, pages 149–160, San Diego, CA, Aug. 2001.

[21] T. Suel, C. Mathur, J. Wu, J. Zhang, A. Delis, M. Kharrazi, X. Long, and K. Shanmugasunderam. ODISSEA: A Peer-to-Peer Architecture for Scalable Web Search And Information Retrieval. In *Proc. of WebDB'03*, 2003.

[22] SVDPACK. http://www.netlib.org/svdpack/index.html.

[23] C. Tang, S. Dwarkadas, and Z. Xu. Hybrid Global-Local Indexing for Efficient Peer-to-Peer Information Retrieval. In *Proc. of the First USENIX/ACM NSDI*, San Franscisco, CA, Mar. 2004.

[24] C. Tang, Z. Xu, and S. Dwarkadas. Peer-to-Peer Information Retrieval Using Self-Organizing Semantic Overlay Networks. In *Proc. of ACM SIGCOMM*, 2003.

[25] Global Internet Geography 2003. TeleGeography, Inc.

[26] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images.* 1999.

Efficient Engines for Keyword Proximity Search

Benny Kimelfeld
The Selim and Rachel Benin School of Engineering and Computer Science
The Hebrew University of Jerusalem
Edmond J. Safra Campus
Jerusalem 91904, Israel
bennyk@cs.huji.ac.il

Yehoshua Sagiv
The Selim and Rachel Benin School of Engineering and Computer Science
The Hebrew University of Jerusalem
Edmond J. Safra Campus
Jerusalem 91904, Israel
sagiv@cs.huji.ac.il

ABSTRACT

This paper presents a formal framework for investigating keyword proximity search. Within this framework, three variants of keyword proximity search are defined. For each variant, there are algorithms for enumerating all the results in an arbitrary order, in the exact order and in an approximate order. The algorithms for enumerating in the exact order make the inevitable assumption that the size of the query (i.e., the number of keywords) is fixed, but the other algorithms do not make this assumption. All the algorithms are provably efficient, that is, run with polynomial delay. The algorithms for enumerating in an approximate order are provably correct for a natural notion of approximation that is defined in this paper.

1. INTRODUCTION

The World-Wide Web is a catalyst for the amalgamation of two data-extraction paradigms: Information retrieval and database querying. An early work that combined the two paradigms is proximity search in databases [10]. More recently, several approaches (e.g., [1, 3, 5, 6, 13, 14, 15, 18]) have been proposed for combining querying and keyword searching when extracting information from either relational or XML data. A key concept underlying all these techniques is *reduced subtrees.* Formally, a *keyword search* is posed as a set of keywords K. A result of a keyword search is a subtree T of the given data graph G, such that T is *reduced* with respect to the given set of keywords K; that is, T contains K, but no proper subtree of T contains K.

DBXplorer [1], BANKS [3] and DISCOVER [14] are three systems that support keyword search in relational databases. In these systems, a database is (or can be) viewed as a data graph G that has tuples and keywords as nodes. Two tuples are connected by an edge if they can be joined using a foreign key; a tuple t and a keyword k are connected if t contains k. Thus, a result of a keyword search is a subtree of G that is reduced with respect to K. Ranking of results is based on the notion of *keyword proximity;* that is, a smaller reduced subtree has a higher rank.

The above systems use two different approaches for producing reduced subtrees. DBXplorer and DISCOVER are similar in the sense that they find all reduced subtrees by constructing join expressions (called *candidate networks* in DISCOVER and *join trees* in DBXplorer) and then evaluating those expressions according to their size, starting with the smallest expression. Thus, the results are produced in the order of increasing size.

BANKS uses a different approach—a graph-search algorithm (called *backward expanding search*) is applied directly to the data graph. In this approach, weights can be assigned to the nodes and edges of the data graph.

Neither approach is optimal. Generating all join expressions is quick if the schema is small and simple. In general, however, there could be many join expressions that yield empty results for the database at hand.

The approach of BANKS is a heuristics for generating all reduced subtrees. It may generate duplicate results and backtracking is required to eliminate them. Furthermore, the actual order of the generated results is not the exact order determined by the weights. No formal statement is made in [3] regarding by how much the actual order can deviate from the required order.

In [13], the approach of DISCOVER is extended with information-retrieval techniques for ranking. XKeyword [15] is a tool that generates, from a given XML document, descriptive fragments containing the given keywords. These fragments are obtained from reduced subtrees of the document graph. Generating the reduced subtrees is based on an adaptation of the method that is used in DISCOVER.

The work of [5] investigates the problem of finding semantic relationships among nodes of XML documents and gives efficient solutions for tree documents. XSearch [6] combines the approach of [5] with information-retrieval techniques. In [18], it is shown that an efficient algorithm for generating reduced subtrees is needed for finding semantically related nodes of graph documents (i.e., XML documents that may have ID references).

In [21], it is argued that a result of a Web search should be more than a single page, since frequently the keywords specified by the user are spread over several pages. Thus, they use heuristics for generating reduced subtrees (of the Web graph) in increasing size. But the actual order is suboptimal (i.e., not the exact order) and the time complexity is, in the worst case, exponential in the size of the diameter of the graph that embeds the generated subtrees.

In summary, algorithms for enumerating reduced subtrees are needed in different types of search engines. Existing systems have implemented heuristics that may perform well in practice. From a theoretical point of view, however, there

Eighth International Workshop on the Web and Databases (WebDB 2005), June 16-17, 2005, Baltimore, Maryland.

are two open problems. The first is efficiency. More precisely, is it possible to enumerate reduced subtrees efficiently either in an arbitrary order or in the exact order? In the case of the exact order, a known NP-complete [9] problem implies that enumeration cannot be done efficiently if the set of keywords K has an unbounded size. However, there is still the possibility of finding algorithms that are more efficient than the above heuristics if K has a fixed size. The second open problem is approximations. What is a suitable notion of enumerating in an approximate order and can it be done efficiently?

In this paper, we describe a formal framework and give complexity results for the problem of enumerating reduced subtrees. The formal framework clearly identifies three variants of this problem. One variant deals with directed subtrees and two deal with undirected subtrees. Note, for example, that BANKS generates directed subtrees whereas DBXplorer and DISCOVER generate undirected subtrees.

For each of the three variants, the following results hold. First, reduced subtrees can be enumerated efficiently if no order is imposed. Second, if the set of keywords K has a fixed size, then reduced subtrees can be enumerated efficiently in increasing size (or weight). Third, we define a formal notion of enumerating reduced subtrees in an approximate order and show that approximate enumeration can be done efficiently, even if the size of K is not fixed.

We derived the last two results by discovering intricate relationships that hold between enumerations of reduced subtrees and Steiner-tree optimization problems. The latter have been investigated extensively over many years and there is a wealth of results about them. We found that these results can be used to develop efficient algorithms for enumerating reduced subtrees.

To summarize, we show that there are efficient algorithms for several variants of keyword proximity search, where results are either ranked, unranked or approximately ranked. Our algorithms are both provably efficient and provably approximate. In comparison, earlier work (e.g., [1, 3, 13, 14, 15, 21]) used heuristics for developing such algorithms.

This paper is organized as follows. Section 2 defines basic concepts and notations. Section 3 describes the formal framework. The main results are described Sections 4. We conclude in Section 5.

2. DATA GRAPHS

A *data graph* G consists of a set $\mathcal{V}(G)$ of *nodes* and a set $\mathcal{E}(G)$ of *edges*. There are two types of nodes: *structural* nodes and *keywords*. A node is either structural or a keyword, but not both. We use $\mathcal{S}(G)$ to denote the set of structural nodes of G, and $\mathcal{K}(G)$ to denote the set of keywords of G. Unless explicitly stated otherwise, edges are directed, i.e., an edge is a pair (n_1, n_2) of nodes. Keywords have only incoming edges, while structural nodes may have both incoming and outgoing edges. Hence, no edge can connect two keywords. These restrictions mean that $\mathcal{E}(G) \subseteq \mathcal{S}(G) \times \mathcal{V}(G)$. The edges of a data graph G may have *weights*. The weight function w_G assigns a positive weight $w_G(e)$ to every edge $e \in \mathcal{E}(G)$. The weight of the data graph G, denoted $w(G)$, is the sum of the weights of all the edges of G (i.e., $w(G) = \sum_{e \in \mathcal{E}(G)} w_G(e)$).

A data graph is *rooted* if it contains some node r, such that all the nodes of G are reachable from r through a directed path. The node r is called a *root* of G. (Note that a rooted data-graph may have several roots.) A data graph is *connected* if its underlying undirected graph is connected.

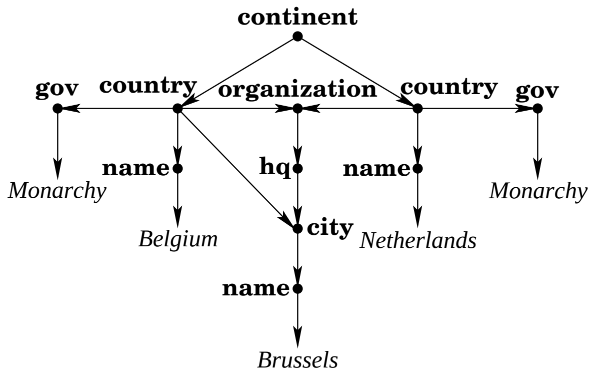


Figure 1: A data graph G_1.

As an example, consider the data graph G_1 depicted in Figure 1. (G_1 is a subgraph of the Mondial[1] XML database.) In this graph, filled circles represent structural nodes and keywords are written in italic font. Note that the keyword *Monarchy* appears twice in this figure; in the actual data graph, however, the keyword *Monarchy* is represented by a single node that has two incoming edges. Also note that the structural nodes of G_1 have *labels*, but in this paper we do not use them. The data graph G_1 is rooted and the node labeled with **continent** is the only root.

We use two types of data *trees*. A *rooted tree* is a rooted data graph, such that there is only one root and for every node u, there is a unique path from the root to u. An *undirected tree* is a connected data graph that contains no cycles, even when ignoring the directions of the edges.

We say that G' is a *subgraph* of the data graph G, denoted $G' \subseteq G$, if $\mathcal{V}(G') \subseteq \mathcal{V}(G)$ and $\mathcal{E}(G') \subseteq \mathcal{E}(G)$. Rooted and undirected *subtrees* are special cases of subgraphs.

3. KEYWORD PROXIMITY SEARCH

3.1 The Framework

In this section, we describe a framework for *keyword proximity search*. A *query* is simply a finite set of keywords K. The result of applying the query K to a data graph G consists of subgraphs of G that contain all the keywords of K. However, such a subgraph is not necessarily a meaningful answer, since an answer should typically have some additional properties. One important property is connectivity. In some cases, e.g., [1, 13, 14, 15, 18, 21], connectivity is defined in an undirected manner, while in other cases, e.g., [3, 18], connectivity is defined in a directed manner. We consider both types of connectivity and use the following definition.

DEFINITION 1 (K-SUBGRAPHS). *Let K be a query. A* rooted K-subgraph *of a data graph G is a rooted subgraph F, such that $K \subseteq \mathcal{K}(F)$. Similarly, an* undirected K-subgraph *of G is a connected subgraph F, such that $K \subseteq \mathcal{K}(F)$.*

A K-subgraph may include information that is irrelevant to the given query. For example, the original data graph G is usually a K-subgraph, but it is rarely a meaningful

[1]http://www.dbis.informatik.uni-goettingen.de/Mondial/

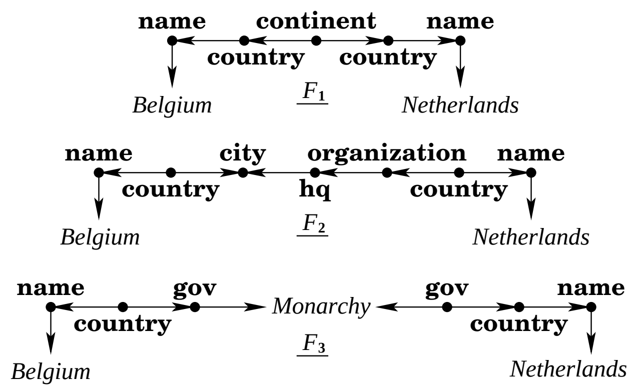


Figure 2: Fragments of G_1.

answer. Therefore, meaningful answers should be *reduced* K-subgraphs, as defined next.

DEFINITION 2 (REDUCED K-SUBGRAPHS). *Let G be a data graph. A rooted (respectively, undirected) K-subgraph F is* reduced *if no proper subgraph of F is also a rooted (respectively, undirected) K-subgraph of G.*

We use the following terminology. A reduced rooted K-subgraph is called a *rooted K-fragment.* Similarly, a reduced undirected K-subgraph is called an *undirected K-fragment.*

As an example, consider the graph G_1 of Figure 1 and the subtrees F_1, F_2 and F_3 of G_1 depicted in Figure 2. Let K be the query $\{Belgium, Netherlands\}$. The subtrees F_1, F_2 and F_3 are all K-fragments. The fragment F_1 is rooted (its root is the node labeled with **continent**). The fragments F_2 and F_3 are undirected, but not rooted. Note that in F_3 the keyword *Monarchy* is needed to maintain connectivity.

Some systems (e.g., [1, 13, 14, 15, 21]) use undirected connectivity, but require all keywords to appear in the leaves (i.e., F_3 is deemed an inappropriate result). Therefore, we introduce a third type of fragments. Formally, an undirected K-fragment F is a *strong K-fragment* if the subgraph of F that is induced by $\mathcal{S}(F)$ (i.e., the structural nodes) is connected. Note that F_2 is a strong fragment, but F_3 is not.

The following proposition shows that K-fragments are trees with some additional properties. Note that the statement about rooted K-fragments follows from the fact that, in data graphs, keywords have only incoming edges.

PROPOSITION 1. *For a data graph G:*

- *If F is a rooted K-fragment, then F is a rooted tree, such that K is the set of all keywords of F, i.e., $K = \mathcal{K}(F)$, and K is also the set of all the leaves of F.*
- *If F is an undirected K-fragment, then F is an undirected tree, such that $K \subseteq \mathcal{K}(F)$ and K includes all the leaves of F.*
- *If F is a strong K-fragment, then F is an undirected tree, such that $K = \mathcal{K}(F)$ and K is also the set of all the leaves of F.*

Note that a rooted K-fragment is also a strong K-fragment and a strong K-fragment is also an undirected K-fragment.

The three types of K-fragments lead to three types of *keyword proximity search.* First, *keyword rooted-proximity search* (KRPS) has the goal of generating all the rooted K-fragments for a given data graph G and a query K. Second, *keyword undirected-proximity search* (KUPS) has the goal of generating all the undirected K-fragments. Third, *keyword strong-proximity search* (KSPS) has the goal of generating all the strong K-fragments. The output of these searches can be exponential in the size of the input (i.e., the data graph G and the query K).

It has already been observed (e.g., [3, 21]) that the above search problems generalize Steiner-tree optimization problems. A *Steiner tree* is a minimal-weight subtree, of a given graph G, that contains a given set of nodes K. KRPS generalizes the problem of finding a directed Steiner tree. KUPS generalizes the problem of finding an undirected Steiner tree. KSPS generalizes the problem of finding a *group* Steiner tree, which is a Steiner tree that contains at least one node from each set S_i, where $S_1, ..., S_k$ is a given collection of sets of nodes. In Section 4, we show that the connection between algorithms for keyword proximity search and algorithms for Steiner-tree optimization problems is, in fact, much closer than has been previously realized.

The above framework can model a wide range of keyword-search paradigms. It includes the various types of keyword search in relational databases and XML [1, 3, 13, 14, 15], the "information unit" Web search of [21] and the semantic search of [5, 6, 18]. Moreover, it is applicable to both document-centric and data-centric environments. For example, when modeling an XML document, the text below an element can be represented by creating a keyword node for each word. The weight of the edge leading to a keyword node can be determined by various information-retrieval factors, such as *tf/idf.* Note that when applying our framework to XML, a search result can be a reduced subtree that is not necessarily a complete subtree of the given XML document, as opposed to the approach taken in INEX [8] and XRANK [11].

3.2 Search Engines

In this paper, an algorithm for a keyword proximity search is called a *search engine.* Specifically, we consider *KRPS engines, KUPS engines* and *KSPS engines.*

In this section, we describe how to measure the *quality* of a search engine. We investigate two aspects of quality. The first aspect deals with efficiency of time and space. The second aspect takes into consideration the ranking order of the results.

3.2.1 Efficiency of Search Engines

Polynomial-time complexity is not a suitable yardstick of efficiency when analyzing a search engine, since the output size could be exponential in the input size. In [17], several definitions of efficiency for enumeration algorithms are discussed. The weakest definition is *polynomial total time,* that is, the running time is polynomial in the combined size of the input and the output. Typically, however, a search engine is required to generate results as soon as possible, since users usually expect to get the results incrementally (i.e., in pages) rather than to get them all at once, at the end of the computation. Therefore, a more suitable (and stronger) definition of time complexity is *polynomial delay,* that is, the time between generating two successive results is polynomial in the input size.

For characterizing space efficiency, we only measure the space used to store intermediate results (i.e., the amount of space needed to write the output is ignored). Preferably, a search engine should only use *polynomial space,* that is, polynomial in the input size. *Linearly incremental polynomial space* means that the space needed for generating the first i results is bounded by i times a polynomial in the input size. Note that if an algorithm runs with polynomial delay, then it uses at most linearly incremental polynomial space.

In the next section, we will show that the KRPS, KUPS and KSPS problems have search engines that run with polynomial delay and use polynomial space, even if the size of the query is not fixed. However, these problems become harder if the results have to be produced according to a ranking order.

3.2.2 Ordering the Results

Presenting search results in a ranking order is highly desirable. It may even be crucial when many results are likely to exist. We take the common approach (e.g., [14, 3, 15, 1]) that the ranking of results, in a keyword proximity search, should be inversely proportional to their weight; in particular, the minimal-weight result should be first.

Consider a search engine E. For a given query K and a data graph G, we use $E_1(G,K),\dots,E_n(G,K)$ to denote the actual order of the K-fragments as they are produced by E. We use $O_1(G,K),\dots,O_n(G,K)$ to denote the same K-fragments, but according to the ranking order, that is, the order of increasing weight.

A search engine E is *optimal* if it enumerates the search results by increasing weight. In other words, E is optimal if for all queries K and data graphs G, it holds that $w(E_i(G,K)) \le w(E_j(G,K))$ whenever $i \le j$.

In general, it is intractable to achieve both polynomial delay and optimality. Therefore, we propose the notion of a search engine that produces results in an *approximate order.* We say that a search engine is C*-competitive* if the weight of its ith result is at most C times the weight of the ith result of an optimal search engine. Formally, the search engine E is C-competitive if for all queries K and data graphs G, it holds that $w(E_i(G,K)) \le Cw(O_i(G,K))$ for all $1 \le i \le n$. Note that C may be a function of G and K.

Finally, we say that a search engine is C*-optimal* if whenever one result precedes another result, then the ratio of the first to the second is no more than C. Formally, a search engine is C-optimal if for all queries K and data graphs G, it holds that $w(E_i(G,K)) \le Cw(E_j(G,K))$ for all $1 \le i \le j \le n$.

A search engine can be C-competitive without being C-optimal. The converse, however, cannot happen.

PROPOSITION 2. *A C-optimal search engine is also C-competitive.*

4. SEARCH ENGINES

In this section, we describe our results about search engines. Due to space limitations, full details of the algorithms cannot be given here and will be described in future papers.

4.1 Efficient Search Engines

If we allow search engines to produce results in an arbitrary order, then all three types of keyword proximity search can be done with polynomial delay while using only polynomial space. This result is stated in the next theorem.

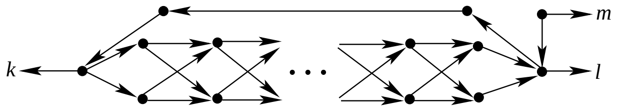


Figure 3: A data graph G_2.

Note that a straightforward corollary of this result is that there are KRPS, KUPS and KSPS engines that are optimal and run in polynomial total time (but not with polynomial delay).

THEOREM 1. *There are three engines, for KRPS, KUPS and KSPS, respectively, that run with polynomial delay and use polynomial space.*

The hard part of the above theorem is the KRPS engine. Not only is it difficult to come up with an efficient algorithm, it is also subtle to analyze the complexity of that algorithm.

The KUPS and the KSPS engines are obtained by recursive enumerations of K-fragments as follows. Given a query K, we choose a keyword $k \in K$ and recursively enumerate all the undirected (respectively, strong) $(K\setminus\{k\})$-fragments. To achieve polynomial delay (rather than just polynomial total time), each of the generated $(K \setminus \{k\})$-fragments is completed to a K-fragment while the recursion is going on. This approach, however, cannot be used in the KRPS engine. The reason is that some graphs have only a few rooted K-fragments but exponentially many rooted K'-fragments for some subset $K' \subset K$. For example, consider the directed graph G_2 depicted in Figure 3. G_2 contains exactly one rooted $\{k,l,m\}$-fragment, but it has exponentially many (in the size of G_2) rooted $\{k,l\}$-fragments. Hence, the desired complexity will not be achieved by the recursion described above, and a more elaborate recursion is needed.

4.2 Efficient and Optimal Search Engines

The three variants of keyword proximity search are unlikely to have optimal search engines that run with polynomial delay, unless P=NP. This follows from the fact that it is NP-complete [9] just to determine whether there is a Steiner tree that has a size of at most m. Hence, even the minimal K-fragment (i.e., the first result) is hard to find. However, if K has a fixed size, finding a minimal K-fragment is solvable in polynomial time. (This was proved in [7] for the undirected case, and that approach can be extended to the directed and the strong cases.)

In practice, queries are small and are assumed to have a fixed size. Under this assumption, the question is whether the tractability of finding a minimal K-fragment implies the existence of a search engine that is both efficient and optimal. The positive answer is given by the following theorem.

THEOREM 2. *If queries are of fixed size, then there are KRPS, KUPS and KSPS optimal search engines that run with polynomial delay.*

The proof of this theorem is rather intricate. We briefly describe the optimal KRPS engine. The other two engines are created using a similar approach, but the details are largely different. The starting point is the general technique of Lawler [20] (that extended that of Yen [23]) for reducing enumeration problems to optimization problems.

A straightforward application of Lawler's technique, in the directed case, yields a hard optimization problem (i.e., finding a minimal rooted K-fragment that includes a given set of edges). We developed a special encoding of data graphs that yielded a tractable version of this optimization problem (i.e., only special sets of edges have to be considered). We solved this tractable version by reducing it to the problem of finding rooted Steiner trees. (For KUPS and KSPS the reductions are to the undirected Steiner-tree problem and to the group Steiner-tree problem, respectively.)

For finding Steiner trees, we used the algorithm of [9] as a basis. However, that algorithm finds only undirected Steiner trees and hence is inappropriate for KRPS and KSPS engines. We were able to modify this algorithm for solving the directed Steiner-tree problem and the group Steiner-tree problem. We implemented the algorithm and tested its performance on the Mondial XML database. We found that this algorithm has an inherent inefficiency, since it computes and stores the shortest path between every pair of nodes in the data graph. (Specifically, doing so for the the Mondial XML database required 3.5GB of memory.) Thus, we had to modify the algorithm so that it will not compute shortest paths for all pairs of nodes. In doing so, we reduced the space requirement and improved the running time from $\mathcal{O}(|\mathcal{E}(G)||\mathcal{V}(G)|)$ to $\mathcal{O}(|\mathcal{E}(G)|\log|\mathcal{V}(G)|)$.

4.2.1 A KRPS Engine for DAGs

For the special case of acyclic data graphs, we can obtain an optimal KRPS engine with superior performance compared to the one described above. This is achieved by adapting the approach of [16] (which is not related to Steiner trees). In the previous KRPS engine, every delay is polynomial, but the degree depends on the size of the query (which is assumed to be fixed). The KRPS engine that we developed for acyclic data graphs requires an initialization step that is exponential in the size of the query (and polynomial in the size of the data graph). However, after the initialization is over, the delay between successive results is linear in the size of both the query and the data graph. Consequently, this engine can also be used for larger queries, if a slow initialization is allowed. This result is summarized in the following theorem.

THEOREM 3. *For acyclic data graphs, KRPS has an optimal search engine that runs with $\mathcal{O}(m)$ delay following an $\mathcal{O}(mn^k)$ initialization step, where k is the size of the query, and n and m are the number of nodes and edges in the data graph, respectively.*

4.3 Approximately Optimal Search Engines

As mentioned earlier, it is unlikely that there are optimal search engines that run with polynomial delay, as far as queries of unbounded size are concerned. We have already shown that if the polynomial-delay requirement is relaxed to polynomial total time, then it is possible to obtain optimal search engines. If, however, one insists on a polynomial delay for queries of unbounded size, then the best that can be expected is an enumeration of search results in an approximate order.

Obviously, approximating keyword proximity searches is at least as hard as approximating Steiner trees. More precisely, if there is a C-optimal search engine that runs with polynomial delay, then there is a C-approximation of the corresponding Steiner-tree. Surprisingly, the converse is also true, as formulated in the following theorem.

THEOREM 4. *Suppose that the directed (respectively, undirected, group) Steiner-tree problem has a C-approximation algorithm. Then KRPS (respectively, KUPS, KSPS) has a $(C+1)$-optimal engine that runs with polynomial delay.*

The proof of this theorem shows that approximation algorithms for the three Steiner-tree problems can be used to construct $(C+1)$-optimal KRPS, KUPS and KSPS engines that run with polynomial delay. The details are beyond the scope of this paper.

4.4 Performance Analysis

Several complexity results for search engines are implied by the above theorems and they are given in Table 1. The top part describes both KRPS and KSPS engines, and the bottom part describes KUPS engines. The first column gives the approximation ratio and the second column gives the time delay between successive search results. In this table, n and m denote the number of nodes and the number of edges, respectively, in the data graph. The size of the query (i.e., number of keywords) is k. The number of nodes in the ith search result is n_i. The approximation ratio is either a constant or determined by the positive integer j that is supplied by the user; in the third line of the top part, it also depends on k. Note that Table 1 gives only some of the results that could be obtained from the vast literature on approximation algorithms for Steiner trees.

The first line in both parts (where the approximation ratio is ∞) is for the algorithms of Theorem 1 that enumerate in an arbitrary order. The second line is for the algorithms of Theorem 2 that enumerate in the exact order (hence, the approximation ratio is 1). The rest of the lines are for algorithms that enumerate in an approximate order, as implied by Theorem 4. The complexity result in the third line of the top part follows from [4]. The complexity results in the last three lines of the bottom part follow from [19], [24] and [22], respectively.

We can also use the results of [2] and [12] to obtain a randomized $O(\log^2|k|\log\Delta\log\log\Delta)$-optimal KSPS engine that runs with polynomial delay, where Δ is the diameter of the data graph. Note that it is often reasonable to assume that graphs have small diameters, as suggested by the "small-world phenomenon."

Note that for KUPS, a constant approximation ratio is realizable. While the 3-optimal KUPS engine is relatively fast, the delay substantially grows when approaching the approximation ratio $2+\frac{\ln 3}{2}$ (≈ 2.55).

From [12], it follows that it is unlikely to find either a KRPS or a KSPS engine that runs with polynomial delay and has an approximation ratio that is better than a polylogarithmic function of the query size. This shows that the undirected version of keyword proximity search is inherently easier than either the directed or the strong version.

5. CONCLUSION

The framework of this paper comprises three types of keyword proximity search. For each type, there are algorithms for generating results in an arbitrary order, in the exact order (for queries of fixed size) and in an approximate order. Earlier papers (with the exception of [18]) considered just one of the three types (e.g., KRPS in [3] and KSPS in [1, 14, 21]) and used heuristics for generating search results.

KRPS/KSPS Engines

Approximation Ratio	Delay
∞	$\mathcal{O}\left(mk(n_i + n_{i+1})\right)$
1	$\mathcal{O}\ \ n_i(3^k n + 2^k m \log n$
$j(j-1)k^{\frac{1}{j}} + 1$	$\mathcal{O}(n_i n^j k^{2j})$

KUPS Engines

Approximation Ratio	Delay
∞	$\mathcal{O}\left(mk(n_i + n_{i+1})\right)$
1	$\mathcal{O}\ \ n_i(3^k n + 2^k m \log n$
3	$\mathcal{O}\left(n_i(n \log n + m)\right)$
$\frac{17}{6}$	$\mathcal{O}\ \ n_i n^3$
$1+\ \ 1+\frac{1}{\lfloor \log_2 j \rfloor + 1}\ \ \ \ 1+\frac{\ln 3}{2}$	$\mathcal{O}(n^j)$

Table 1: Time complexities of the search engines.

In comparison, our algorithms are both provably efficient (i.e., run with polynomial delay) and provably approximate (i.e., C-optimal).

One benefit of our approach is the ability to tune a search engine for a specific combination of efficiency and approximation, using the wealth of results about Steiner trees. Another benefit is the range of recall-precision tradeoffs given by the three types of keyword proximity search.

Our results can be easily extended in several ways. First, our algorithms can support the type of visualization that is done in XKeyword [15], where nodes are extended to include their neighborhoods. Second, we can also include labels in the search, either by treating labels just as keywords or by using labels as meta data, following the approach of [5, 6, 18]. Third, the KRPS, KUPS and KSPS optimal engines as well as the KRPS approximate engines can be extended so that nodes (and not just edges) may have weights, as done in BANKS [3]. It is an open problem whether the same can be done for the KUPS and KSPS approximate engines.

In future work, we plan to implement our framework and enhance its efficiency by developing indexing techniques.

6. ACKNOWLEDGMENTS

The authors thank Yair Bartal for helpful discussions. This work was supported by the Israel Science Foundation (Grant No. 96/01).

7. REFERENCES

[1] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: enabling keyword search over relational databases. In *SIGMOD Conference*, page 627, 2002.

[2] Y. Bartal and M. Mendel. Multi-embedding and path approximation of metric spaces. In *SODA*, pages 424–433, 2003.

[3] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using BANKS. In *ICDE*, pages 431–440, 2002.

[4] M. Charikar, C. Chekuri, T. Y. Cheung, Z. Dai, A. Goel, S. Guha, and M. Li. Approximation algorithms for directed steiner problems. In *SODA*, pages 192–200, 1998.

[5] S. Cohen, Y. Kanza, and Y. Sagiv. Generating relations from XML documents. In *ICDT*, pages 285–299, 2003.

[6] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv. XSEarch: A semantic search engine for XML. In *VLDB*, pages 45–56, 2003.

[7] S.E. Dreyfus and R.A. Wagner. The Steiner problem in graphs. *Networks*, 1:195–207, 1972.

[8] Norbert Fuhr, Mounia Lalmas, and Saadia Malik, editors. *INitiative for the Evaluation of XML Retrieval (INEX). Proceedings of the Second INEX Workshop. Dagstuhl, Germany, December 15–17, 2003*, March 2004.

[9] M. R. Garey, R. L. Graham, and D. S. Johnson. The complexity of computing Steiner minimal trees. *Siam*, 32:835–859, 1977.

[10] R. Goldman, N. Shivakumar, S. Venkatasubramanian, and H. Garcia-Molina. Proximity search in databases. In *VLDB*, pages 26–37, 1998.

[11] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked keyword search over XML documents. In *SIGMOD*, pages 16–27, 2003.

[12] E. Halperin and R. Krauthgamer. Polylogarithmic inapproximability. In *STOC*, pages 585–594, 2003.

[13] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient IR-style keyword search over relational databases. In *HDMS*, 2003.

[14] V. Hristidis and Y. Papakonstantinou. DISCOVER: Keyword search in relational databases. In *VLDB*, pages 670–681, 2002.

[15] V. Hristidis, Y. Papakonstantinou, and A. Balmin. Keyword proximity search on XML graphs. In *ICDE*, pages 367–378, 2003.

[16] V. M. Jiménez and A. Marzal. Computing the K shortest paths: A new algorithm and an experimental comparison. In *Algorithm Engineering*, pages 15–29, 1999.

[17] D.S. Johnson, M. Yannakakis, and C.H. Papadimitriou. On generating all maximal independent sets. *Information Processing Letters*, 27:119–123, 1988.

[18] B. Kimelfeld. Interconnection semantics for XML. Master's thesis, Hebrew University, 2004.

[19] L. Kou, G. Markowsky, and L. Berman. A fast algorithm for Steiner trees. *Acta Inf.*, 15:141–145, 1981.

[20] E. L. Lawler. A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem. *Management Science*, 18:401–405, 1972.

[21] Wen-Syan Li, K. Selçuk Candan, Quoc Vu, and Divyakant Agrawal. Retrieving and organizing web pages by "information unit". In *WWW*, pages 230–244, 2001.

[22] G. Robins and A. Zelikovsky. Improved Steiner tree approximation in graphs. In *SODA*, pages 770–779, 2000.

[23] J. Y. Yen. Finding the k shortest loopless paths in a network. *Management Science*, 17:712–716, 1971.

[24] A. Zelikovsky. An 11/6-approximation algorithm for the network steiner problem. *Algorithmica*, 9(5):463–470, 1993.

Freshness-Aware Scheduling of Continuous Queries in the Dynamic Web [*]

Mohamed A. Sharaf, Alexandros Labrinidis, Panos K. Chrysanthis, Kirk Pruhs
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260, USA
{msharaf, labrinid, panos, kirk}@cs.pitt.edu

ABSTRACT

The dynamics of the Web and the demand for new, active services are imposing new requirements on Web servers. One such new service is the processing of continuous queries whose output data stream can be used to support the personalization of individual user's web pages. In this paper, we are proposing a new scheduling policy for continuous queries with the objective of maximizing the freshness of the output data stream and hence the QoD of such new services. The proposed Freshness-Aware Scheduling of Multiple Continuous Queries (FAS-MCQ) policy decides the execution order of continuous queries based on each query's properties (i.e., cost and selectivity) as well the properties of the input update streams (i.e., variability of updates). Our experimental results have shown that FAS-MCQ can increase freshness by up to 50% compared to existing scheduling policies used in Web servers.

1. INTRODUCTION

Web databases and HTML/XML documents scattered all over the World Wide Web provide immeasurable amount of information which is continuously growing and updated. To keep up with the Web dynamics, a search engine frequently crawls the web looking for updates. Then, it propagates the stream of updates to its internal databases and indexes.

The problem of propagating updates gets more complicated when the Web server provides users with the service of registering *continuous queries*. A continuous query is a standing query whose execution is triggered every time a new update arrives [18]. For example, a user might register a query to monitor news related to the *NFL*. Thus, as new sports articles arrive to the server, all the *NFL* related ones have to be propagated to that user. As such, the arrival of new updates triggers the execution of a set of corresponding queries, since portions of the new updates may be relevant to the query. The output of such a frequent execution of a continuous query is what we call an *output data stream* (see Figure 1).

An output data stream can be used, for example to continuously update a user's personalized Web page where a user logs on and monitors updates as they arrive. It can also be used to send email notifications to the user when new results are available [6, 17].

As the amount of updates on the input data streams increases and the number of registered queries becomes high, advanced query processing techniques are needed to efficiently synchronize the results of the continuous queries with the available updates. That is particularly important when the search engine deploys a continuous monitoring scheme instead of traditional crawlers [16].

Efficient *scheduling* of updates is one such query processing technique, which successfully improves the *Quality of Data (QoD)* provided by interactive systems. In this paper, we are focusing on scheduling continuous queries for improving QoD in the interactive dynamic Web. QoD can be measured in different ways, one of which is *freshness*. The objective of our work is to improve the freshness of the continuous data streams resulting from continuous query execution as opposed to the freshness of the underlying databases [7, 8], derived views [10] or caches [15]. In this respect, our work can be regarded as complementary to the current work on the processing of continuous queries, which considers only Quality of Service metrics like response time and throughput (e.g., [6, 17, 2, 4, 1]).

Specifically, the contribution of this paper is proposing a policy for *Freshness-Aware Scheduling of Multiple Continuous Queries (FAS-MCQ)*. FAS-MCQ has the following salient features:

1. It exploits the variability of the processing costs of different continuous queries registered at the Web server.
2. It utilizes the divergence in the arrival patterns and frequencies of updates streamed from different remote data sources.
3. It considers the impact of *selectivity* on the freshness of a Web output data stream.

To illustrate the last point on the impact of selectivity, let us assume a continuous query which is used to project the number of trades on a certain stock if its price exceeds \$60. Further, assume that there is a 50% chance that this stock's price exceeds \$60. With the arrival of a new update, if the new price is greater than \$60 then a new update is added to the continuous output data stream. Otherwise, the update is discarded and nothing is added to the output data stream. So, in this particular example, the arrival of a new update renders the continuous output data stream stale with probability 50%. FAS-MCQ exploits the probability of staleness in order to maximize the overall QoD.

As our experimental results have shown, FAS-MCQ can increase freshness by up to 50% compared to existing scheduling policies used in Web servers. FAS-MCQ achieves this improvement by deciding the execution order of continuous queries based on individual query properties (i.e., cost and selectivity) as well as properties of the update streams (i.e., variability of updates).

The rest of this paper is organized as follows. Section 2 provides the system model. In Section 3, we define our freshness-based QoD

[*]This work is supported in part by NSF ITR Medium Award (ANI-0325353). The first author is supported in part by the Andrew Mellon Predoctoral Fellowship.

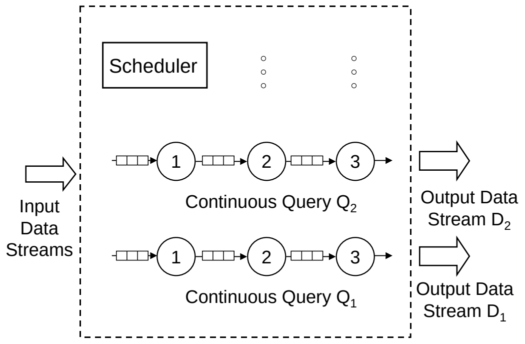


Figure 1: A Web server hosting multiple continuous queries

metrics. Our proposed policy for improving freshness is presented in Section 4. Section 5 describes our simulation testbed, whereas Section 6 discusses our experiments and results. Section 7 surveys related work. We conclude in Section 8.

2. SYSTEM MODEL

We assume a Web server where users register multiple continuous queries over multiple input data streams (as shown in Figure 1). Data streams consist of updates of remote data sources that are either continuously pushed to the Web server or frequently pulled by the Web server through crawlers. Each update u_i is associated with a *timestamp* t_i. This timestamp is either assigned by the data source or by the Web server. In the former case, the timestamp reflects the time when the update took place, whereas in the latter case, it represents the arrival time of the update at the Web server.

In this work, we assume single-stream queries where each query is defined over a single data stream. However, data streams can be shared by multiple queries, in which case each query will operate on its own copy of the data stream. Queries can also be shared among multiple users, in which case the results will be shared among them. Improving the QoD in the context of multi-stream queries as well shared queries or operators is part of our future work.

A single-stream query plan can be conceptualized as a data flow diagram [3, 1] (Figure 1): a sequence of nodes and edges, where the nodes are operators that process data and the edges represent the flow of data from one operator to another. A query Q starts at a *leaf* node and ends at a *root* node (O_r). An edge from operator O_1 to operator O_2 means that the output of operator O_1 is an input to operator O_2. Additionally, each operator has its own input queue where data is buffered for processing.

As a new update arrives at a query Q, it passes through the sequence of operators of Q. An update is processed until it either produces an output or until it is discarded by some predicate in the query. An update produces an output only when it satisfies all the predicates in the query.

In a query, each operator O_x is associated with two values:

- *processing cost* (c_x), and
- *selectivity* or *productivity* (s_x).

Recall that in traditional database systems, an operator with selectivity s_x produces s_x tuples after processing one tuple for c_x time units. s_x is typically less than or equal to 1 for operators like filters. Selectivity expresses the behavior or power of a filter. Additionally, for a query Q_i, we define three parameters

1. *total cost* (C_i),
2. *total selectivity* or *total productivity* (S_i), and
3. *average cost* (C_i^{avg}).

Specifically, for a query Q_i that is composed of a single stream of operators $< O_1, O_2, O_3, ..., O_r >$, C_i, S_i and C_i^{avg} are defined as follows:

$$C_i = c_1 + c_2 + ... + c_r$$

$$S_i = s_1 \times s_2 \times ... \times s_r$$

$$C_i^{avg} = c_1 + c_2 \times s_1 + c_3 \times s_2 \times s_1 + ... + c_r \times s_{r-1} \times ... \times s_1$$

The average cost is computed as follows. An update starts going through the chain of operators with O_1, which has a cost of c_1. With a "probability" of s_1 (equal to the selectivity of operator O_1) the update will not be filtered out, and as such continue on to the next operator, O_2, which has a cost of c_2. Moving along, with a "probability" of s_2 the update will not be filtered out, and as such continue on to the next operator, O_3, which has a cost of c_3. Up until now, on average, the cost will be $C^{avg} = c_1 + c_2 \times s_1 + c_3 \times s_2 \times s_1$. This is generalized in the formula for C_i^{avg} above as in [19].

In the rest of the paper, we use lower-case symbols to refer to operators' parameters and upper-case ones for queries' parameters.

3. FRESHNESS OF WEB DATA STREAMS

In this section, we describe our proposed metric for measuring the quality of output Web data streams. Our metric is based on the *freshness* of data and is similar to the ones previously used in [7, 10, 15, 8, 11]. However, it is adapted to consider the nature of continuous queries and input/output Web data streams.

3.1 Average Freshness for Single Streams

In our system, the output of each continuous query Q is a data stream D. The arrival of new updates at the input queue of Q might lead to appending a new tuple to D. Specifically, let us assume that at time t the length of D is $\mid D_t \mid$ and there is a single update at the input queue; also with timestamp t. Further, assume that Q finishes processing that update at time t'. If the tuple satisfies all the query's predicates, then $\mid D_{t'} \mid = \mid D \mid +1$, otherwise, $\mid D_{t'} \mid = \mid D \mid$. In the former case, the output data stream D is considered *stale* during the interval $[t, t']$ as the new update occurred at time t and it took until time t' to append the update to the output data stream. In the latter case, D is considered *fresh* during the interval $[t, t']$ because the arrival of a new update has been discarded by Q. Obviously, if there is no pending update at the input queue of D, then D would also be considered *fresh*.

Formally, to define freshness, we consider each output data stream D as an object and $F(D, t)$ is the freshness of object D at time t which is defined as follows:

$$F(D,t) = \begin{cases} 1 & \text{if } \forall u \in I_t, \sigma(u) \text{ is false} \\ 0 & \text{if } \exists u \in I_t, \sigma(u) \text{ is true} \end{cases} \qquad (1)$$

where I_t is the set of input queues in Q at time t and $\sigma(u)$ is the result of applying Q's predicates on update u.

To measure the freshness of a data stream D over an entire discrete observation period from time t_x to time t_y, we have that:

$$F(D) = \frac{1}{t_y - t_x} \sum_{i=t_x}^{t_y} F(D,t) \qquad (2)$$

3.2 Average Freshness for Multiple Streams

Having measured the average freshness for single streams, we proceed to compute the average freshness over all the M data streams maintained by the Web server. If the freshness for each stream, D_i, is given by $F(D_i)$ using Equation 2, then the average freshness over all data streams will be:

$$F = \frac{1}{M} \sum_{i=1}^{M} F(D_i) \tag{3}$$

3.3 Fairness in Freshness

Ideally, all data streams in the system should experience perfect freshness. However, this is not always achievable. Especially when the Web server is loaded, we can have data streams with freshness that is less than perfect, because of a "back-log" of updates that cannot be processed in time [10]. In such a case, it is desirable to maximize the average freshness in addition to minimizing the variance in freshness among different data streams. Minimizing the variance reflects the system's *fairness* in handling different continuous queries.

In this paper, we are measuring fairness as in [14]. Specifically, we compute the average freshness of each output Web data stream. Then, we measure fairness as *the standard deviation of freshness* measured for each data stream. A high value for the standard deviation indicates that some classes of data streams received unfair service compared to others. That is, they were stale for a longer intervals compared to other data streams. A low value for the standard deviation indicates that the difference in service (freshness) among different data streams is negligible, and, as such, the Web server handled all streams in a fair manner.

4. FRESHNESS-AWARE SCHEDULING OF MULTIPLE CONTINUOUS QUERIES

In this section we describe our proposed policy for *Freshness-Aware Scheduling of Multiple Continuous Queries (FAS-MCQ)*. Current work on scheduling the execution of multiple continuous queries focuses on QoS metrics [2, 4, 1] and exploits *selectivity* to improve the provided QoS. Previous work on synchronizing database updates exploited the *amount (frequency)* of updates to improve the provided QoD [7, 15, 8]. In contrast, our proposal, *FAS-MCQ*, exploits both selectivity and amount of updates to improve the QoD, i.e., freshness, of output Web data streams.

4.1 Scheduling without Selectivity

Assume two queries Q_1 and Q_2, with output Web data streams D_1 and D_2. Each query is composed of a set of operators, each operator has a certain cost, and the selectivity of each operator is one. Hence, we can calculate for each query Q_i its total cost C_i as shown in Section 2. Moreover, assume that there are N_1 and N_2 pending updates for queries Q_1 and Q_2 respectively. Finally, assume that the current wait time for the update at the head of Q_1's queue is W_1, similarly, the current wait time for the update at the head of Q_2's queue is W_2.

Next, we compare two policies X and Y. Under policy X, query Q_1 is executed before query Q_2, whereas under policy Y, query Q_2 is executed before query Q_1.

Under policy X, where query Q_1 is executed before query Q_2, the total loss in freshness, L_X, (i.e., the period of time where Q_1 and Q_2 are stale) can be computed as follows:

$$L_X = L_{X,1} + L_{X,2} \tag{4}$$

where $L_{X,1}$ and $L_{X,2}$ are the staleness periods experienced by Q_1 and Q_2 respectively.

Since Q_1 will remain stale until all its pending updates are processed, then $L_{X,1}$ is computed as follows:

$$L_{X,1} = W_1 + (N_1 C_1)$$

where W_1 is the current loss in freshness and $(N_1 \times C_1)$ is the time required until applying all the pending updates.

Similarly, $L_{X,2}$ is computed as follows:

$$L_{X,2} = (W_2 + N_1 C_1) + (N_2 C_2)$$

where W_2 is the current loss in freshness plus the extra amount of time $(N_1 \times C_1)$ where Q_2 will be waiting for Q_1 to finish execution.

By substitution in Equation 4, we get

$$L_X = W_1 + (N_1 C_1) + (W_2 + N_1 C_1) + (N_2 C_2) \tag{5}$$

Similarly, under policy Y in which Q_2 is scheduled before Q_1, we have that the total loss in freshness, L_Y will be:

$$L_Y = (W_1 + N_2 C_2) + (N_1 C_1) + W_2 + (N_2 C_2) \tag{6}$$

In order for L_X to be less than L_Y, the following inequality must be satisfied:

$$N_1 C_1 < N_2 C_2 \tag{7}$$

The left-hand side of Inequality 7 shows the total loss in freshness incurred by Q_2 when Q_1 is executed first. Similarly, the right-hand side shows the total loss in freshness incurred by Q_1 when Q_2 is executed first. Hence, the inequality implies that between the two alternative execution orders, we select the one that minimizes the total loss in freshness.

4.2 Scheduling with Selectivity

Assume the same setting as in the previous section. However, assume that the productivity of each query Q_i is S_i which is computed as in Section 2. The objective when scheduling with selectivity is the same as before: we want to minimize the total loss in freshness. Recall from Inequality 7 that the objective of minimizing the total loss is equivalent to selecting for execution the query that minimizes the loss in freshness incurred by the other query. In the presence of selectivity, we will apply the same concept.

We first compute for each output data stream D_i its *staleness probability* (P_i) given the current status of the input data stream. This is equivalent to computing the probability that at least one of the pending updates will satisfy Q_i's predicates. Hence, $P_i = 1-(1-S_i)^{N_i}$, where $(1-S_i)^{N_i}$ is the probability that all pending updates do not satisfy Q_i's predicates.

Now, if Q_2 is executed before Q_1, then the loss in freshness incurred by Q_1 only due to the impact of processing Q_2 first is computed as:

$$L_{Q_1} = P_1 \times N_2 \times C_2^{avg}$$

where $N_2 \times C_2^{avg}$ is the expected time that Q_1 will be waiting for Q_2 to finish execution and P_1 is the probability that D_1 is stale in the first place. For example, in the extreme case of $S_1 = 0$, if Q_2 is executed before Q_1, it will not increase the staleness of D_1 since all the updates will not satisfy Q_1. However, at $S_1 = 1$, if Q_2 is executed before Q_1, then the staleness of D_1 will increase by $N_2 \times C_2^{avg}$ with probability one.

Similarly, if Q_1 is executed before Q_2, then the loss in freshness incurred by Q_2 only due to processing Q_1 first is computed as:

$$L_{Q_2} = P_2 \times N_1 \times C_1^{avg}$$

In order for L_{Q_2} to be less than L_{Q_1}, then the following inequality must be satisfied:

$$\frac{N_1 C_1^{avg}}{P_1} < \frac{N_2 C_2^{avg}}{P_2} \tag{8}$$

Thus, in our proposed policy, each query Q_i is assigned a priority value V_i which is the product of its staleness probability and the reciprocal of the product of its expected cost and the number of its pending updates. Formally,

$$V_i = \frac{1-(1-S_i)^{N_i}}{N_i C_i^{avg}} \tag{9}$$

4.3 The FAS-MCQ Scheduler

The FAS-MCQ scheduler selects for execution the query with the highest priority value at each *scheduling point.* A scheduling point is reached when: (1) a query finishes processing an input update, or (2) when a new update arrives at the system.

In the second case, the scheduler has to decide whether to resume executing the current query or preempt it. A query is preempted if a new update has arrived at a query with priority higher than the one currently executing. Thus, we need to recompute the priority of the currently executing query based on the position of the processed update along the query operators. For example, if the processed update is at the input queue of some operator O_x along the query, then the current priority of the query is computed as:

$$\frac{1-(1-S_x)}{C_x^{avg}}$$

where S_x and C_x^{avg} are the expected productivity and expected cost of the segment of operators starting at O_x all the way to the root. If O_x has been processing the tuple for δ_x time units, then the current priority is computed as above by replacing c_x with $c_x - \delta_x$.

4.4 Discussion

It should be noted that under our policy, the priority of a query increases as the processing of an update advances. For instance, let us assume that a query has just been selected for execution. At that moment, the priority of the query is equal to the priority of its leaf node or leaf operator. After the leaf finishes processing the update, the priority of the next operator, say O_x, is computed as shown earlier. Intuitively, S_x and C_x^{avg} are greater than S and C^{avg} of the leaf operator because the remaining processing cost decreases and the expected productivity might increase too. Additionally, N_x is equal to one and our priority function monotonically decreases with the increase in N. Thus, overall, the priority of O_x is higher than that of the leaf node. Similarly, the priority of each operator in the query is higher than the priority of the operator preceding it. As such, a query Q_i is never preempted unless a new update arrives and that new update triggers the execution of a query with a higher priority than Q_i.

Also note that under our priority function (Equation 9), *FAS-MCQ* behaves as follows:

1. If all queries have the same number of pending tuples and the same selectivity, then FAS-MCQ selects for execution the query with the lowest cost.

2. If all queries have the same cost and the same selectivity, then FAS-MCQ selects for execution the query with less pending tuples.

3. If all queries have the same cost and the same number of pending tuples, then FAS-MCQ selects for execution the query with high staleness probability.

In case (1), *FAS-MCQ* behaves like the *Shortest Remaining Processing Time* policy. In case (2), *FAS-MCQ* gives lower priority to the query with high frequency of updates. The intuition is that when the frequency of updates is high, it will take a long time to establish the freshness of the output Web data stream. This will block other queries from executing and will increase the staleness of their output Web data streams. In case (3), *FAS-MCQ* gives lower priority to queries with low selectivity as there is a low probability that the pending updates will "survive" the filtering of the query operators and thus be appended to the output Web data stream.

5. EVALUATION TESTBED

We have conducted several experiments to compare the performance of our proposed scheduling policy and its sensitivity to different parameters. Specifically, we compared the performance of our proposed *FAS-MCQ* policy to a two-level scheduling scheme from Aurora where Round Robin is used to schedule queries and pipelining is used to process updates within the query. Collectively, we refer to the Aurora scheme in our experiments as *RR*. In addition, we considered a FCFS policy where updates are processed according to their arrival times. Finally, we adapted the Shortest Remaining Processing Time (*SRPT*) policy, where the priority of a query is the reciprocal of its total cost (i.e., $1/C$). The SRPT policy has been shown to work very well for scheduling requests at a Web server when the performance metric is response time [9].

Queries: We simulated a Web server that hosts 250 registered continuous queries. The structure of the query is adapted from [5, 13] where each query consists of three operators: two predicates and one projection. All operators that belong to the same query have the same cost, which is uniformly selected from three possible classes of costs. The cost of an operator in class i is equal to: 2^i time units, where i is 0, 1, or 2.

Selectivities: In any query, the selectivity of the projection is set to 1, while the two predicates have the same value for selectivity, which is uniformly selected from the range [0.1, 1.0].

Streams: The number of input data streams is set to 10 and the length of each stream is set to 10K tuples. Initially, we generate the updates for each stream according to a Poisson distribution, with its mean inter-arrival time set according to the simulated system utilization (or load). For a utilization of 1.0, the inter-arrival time is equal to the exact time required for executing the queries in the system, whereas for lower utilizations, the mean inter-arrival time is increased proportionally. To generate a back-log of updates [10], we have a parameter B which controls the number of *bursty* streams. A bursty stream is created by adapting the initially generated Poisson stream using two parameters: *burst probability (p)* and *burst length (l).* Specifically, we traverse the Poisson stream and at each entry/update we toss a coin, if the tossing result is less than the p, then the arrival time A_b of that update is the beginning of a new burst. Then, the arrival times of each of the next l updates are adjusted so that the new arrival time, A'_i, of an update u_i is set to $(A_i - A_b) * p$, where A_i is the arrival time computed originally under the Poisson distribution. We have conducted several experiments with different settings of the p, l and B parameters. Due to lack of space, we will present the simulation results where p is equal to 0.5, l is equal to 50 updates and B is in the range [0, 10] with the default being 5.

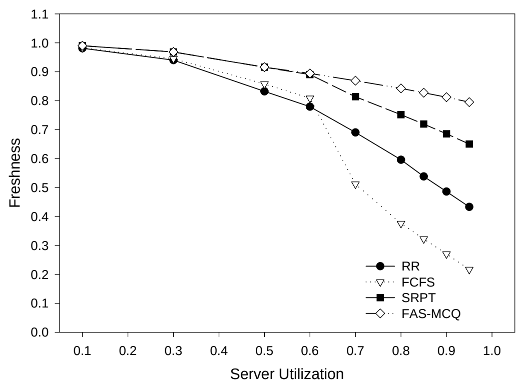


Figure 2: freshness vs. load (selectivity=1.0)

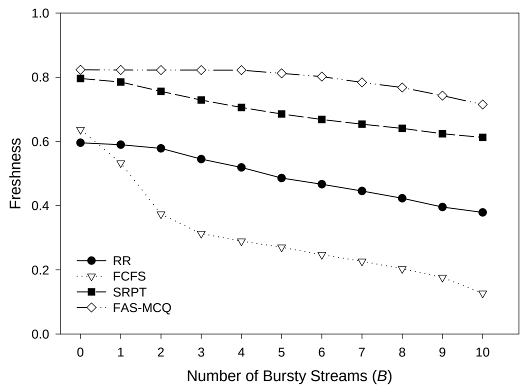


Figure 3: freshness vs. number of bursty streams

6. EXPERIMENTS

6.1 Impact of Utilization

In this experiment, the selectivity for all operators is set to 1, whereas the processing costs are variable and are generated as described earlier. Figure 2 depicts the average total freshness over all output Web data streams as the load at the Web server increases. In this experiment 5 out of the 10 input data streams are bursty. The figure shows that, in general, the freshness of the output Web data streams decreases with increasing load. It also shows that the FAS-MCQ policy provides the highest freshness all the time. The freshness provided by SRPT is equal to that of FAS-MCQ for utilizations up to 0.5. After that point, with increasing utilization, queues start building up. That is when FAS-MCQ gives higher priority to queries with shorter queues and low processing cost in order to maximize the overall freshness of data, thus outperforming SRPT. At 95% utilization, FAS-MCQ has 22% higher freshness than SRPT. If we report QoD as staleness (i.e., the opposite of freshness [15]), then FAS-MCQ is 41% better than SRPT, with just a 20% overall average staleness.

6.2 Impact of Bursts

The setting for this experiment is the same as the previous one. However, the utilization at all points is set to the default value of 90%. In Figure 3, we plot the average total freshness as the number of input data streams that are bursty increases. At a value of 0, all the arrivals follow a Poisson distribution with no bursts, whereas at 10, all input data streams are bursty as described in Section 5.

Figure 3 shows how the total average freshness decreases when the number of bursty data streams increases. It also shows that FAS-MCQ provides the highest freshness compared to the other policies. Notice the relation between FAS-MCQ and SRPT: as the number of bursty streams increases, the difference in freshness provided by FAS-MCQ compared to SRPT increases up until there are 5 bursty streams. At that point, FAS-MCQ has 20% higher freshness than SRPT. At the same time, FAS-MCQ has 1.8 the freshness of the RR policy and 3.6 the freshness of the FCFS policy.

After there are 7 bursty input streams, the performance of the FAS-MCQ and SRPT policies get closer. The explanation is that at a lower number of bursty streams, FAS-MCQ has a better chance to find a query with a short queue of pending updates to schedule for execution. As the number of bursty streams increases, the chance of finding such a query decreases, and as such, SPRT is performing reasonably well. At 10 bursty streams, FAS-MCQ has only 16% higher freshness than SRPT.

6.3 Impact of Selectivity

In this experiment, the cost for all operators is set to 1 time unit. However, the selectivity is chosen uniformly from the range [0.0, 1.0]. Figure 4 depicts how the freshness decreases with increasing load at the Web server. The figure also shows that FAS-MCQ still provides the highest freshness, as it considers the probability that an update will affect the freshness of the corresponding data stream. That is opposite to SRPT which will give a higher priority to a query with low selectivity since a low selectivity will provide a low value for C^{avg}. Hence, SRPT will spend time executing queries that will only append fewer updates to their corresponding output data streams.

In this experiment, RR behaves better than SRPT at high utilizations. At a 95% utilization, FAS-MCQ gives 50% higher freshness than RR and 63% higher than SRPT.

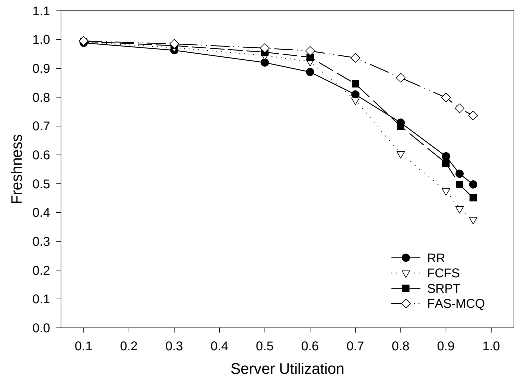


Figure 4: freshness vs. load (variable selectivity)

Figure 5 shows the standard deviation of freshness for the same experiment setting. The figure shows that for all policies, the deviation increases with increasing load where some output data streams are stale for longer times compared to other data streams. However, FAS-MCQ provides the lowest standard deviation for most values of utilization. As the utilization approaches 1 (i.e., when the Web server is about to reach its capacity), the fairness provided by FAS-MCQ gets closer to that of FCFS. Thus, FAS-MCQ is at least as fair as FCFS, even at very high utilizations.

However, the FCFS policy behaves poorly if we look beyond fairness and into the average total freshness: as shown in Figure 4, FAS-MCQ provides 96% higher average freshness compared to FCFS, despite having the same fairness.

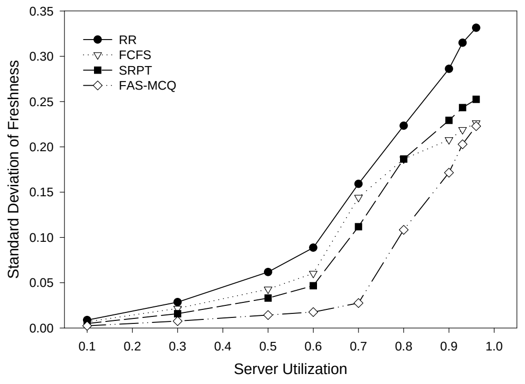


Figure 5: standard deviation of freshness

7. RELATED WORK

The work in [7, 8] provides policies for crawling the Web in order to refresh a local database. The authors make the observation that a data item that is updated more often should be synchronized less often. In this paper, we utilize the same observation, however, [7, 8] assumes that updates follow a mathematical model, whereas we make our decision based on the current status of the Web server queues (i.e., the number of pending updates). The same observation has been exploited in [15] for refreshing distributed caches and in [12] for multi-casting updates.

The work in [10] studies the problem of propagating the updates to derived views. It proposes a scheduling policy for applying the updates that considers the divergence in the computation costs of different views. Similarly, our proposed *FAS-MCQ* considers the different processing costs of the registered multiple continuous queries. Moreover, *FAS-MCQ* generalizes the work in [10] by considering updates that are streamed from multiple data sources as opposed to a single data source.

Improving the QoS of multiple continuous queries has been the focus of many research efforts. For example, multi-query optimization has been exploited in [6] to improve the system throughput in an Internet environment and in [13] for improving the throughput of a data stream management system. Multi-query scheduling has been exploited by Aurora to achieve better response time or to satisfy application-specified QoS requirements [2]. The work in [1] employs a scheduler for minimizing the memory utilization. To the best of our knowledge, none of the above work provided techniques for improving the QoD provided by continuous queries.

8. CONCLUSIONS

Motivated by the need to support active Web services which involved the processing of update streams by continuous queries, in this paper we studied the different aspects that affect the QoD of these services. In particular, we focused on the freshness of the output data stream and identified that both the properties of queries, i.e., cost and selectivity, as well as the properties of the input update streams, i.e., variability of updates, have a significant impact on freshness. For this reason, we have proposed and experimentally evaluated a new scheduling policy for continuous queries that exploits all of these aspects to maximize the freshness of the output data stream. Our proposed Freshness-Aware Scheduling of Multiple Continuous Queries (FAS-MCQ) policy can increase freshness by up to 50% compared to existing scheduling policies used in Web servers. Our next step is to study the problem when MCQ plans include shared operators as well as join operators.

Acknowledgments: We would like to thank the anonymous reviewers for their thoughtful and constructive comments.

9. REFERENCES

[1] B. Babcock, S. Babu, M. Datar, and R. Motwani. Chain: Operator scheduling for memory minimization in data stream systems. In *SIGMOD*, 2003.
[2] D. Carney, U. Cetintemel, A. Rasin, S. Zdonik, M. Cherniack, and M. Stonebraker. Operator scheduling in a data stream manager. In *VLDB*, 2003.
[3] D. Carney, U. Getintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams: A new class of data management applications. In *VLDB*, 2002.
[4] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, V. R. S. Madden, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR*, 2003.
[5] J. Chen, D. J. DeWitt, and J. F. Naughton. Design and evaluation of alternative selection placement strategies in optimizing continuous queries. In *ICDE*, 2002.
[6] J. Chen, D. J. DeWitt, F. Tian, and Y. .Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *SIGMOD*, 2000.
[7] J. Cho and H. Garcia-Molina. Synchronizing a database to improve freshness. In *SIGMOD*, 2000.
[8] J. Cho and H. Garcia-Molina. Effective page refresh policies for web crawlers. *ACM Transactions on Database Systems*, 28(4):390–426, 2003.
[9] M. Harchol-Balter, B. Schroeder, N. Bansal, and M. Agarwal. Size based scheduling to improve web performance. *Transactions on Computer Systems*, 21(2):207–233, 2003.
[10] A. Labrinidis and N. Roussopoulos. Update propagation strategies for improving the quality of data on the web. In *VLDB*, 2001.
[11] A. Labrinidis and N. Roussopoulos. Exploring the tradeoff between performance and data freshness in database-driven web servers. *VLDB J.*, 13(3):240–255, 2004.
[12] W. Lam and H. Garcia-Molina. Multicasting a changing repository. In *ICDE*, 2003.
[13] S. Madden, M. A. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *SIGMOD*, 2002.
[14] M. Mehta and D. J. DeWitt. Dynamic memory allocation for multiple-query workloads. In *VLDB*, 1993.
[15] C. Olston and J. Widom. Best-effort cache synchronization with source cooperation. In *SIGMOD*, 2002.
[16] S. Pandey, K. Ramamritham, and S. Chakrabarti. Monitoring the dynamic web to respond to continuous queries. In *WWW*, 2003.
[17] J. Shanmugasundaram, K. Tufte, D. J. DeWitt, J. F. Naughton, and D. Maier. Architecting a network query engine for producing partial results. In *WebDB*, 2002.
[18] D. B. Terry, D. Goldberg, D. Nichols, and B. M. Oki. Continuous queries over append-only databases. In *SIGMOD*, 1992.
[19] T. Urhan and M. J. Franklin. Dynamic pipeline scheduling for improving interactive query performance. *VLDB*, 2001.

Vague Content and Structure (VCAS) Retrieval for Document-centric XML Collections

Shaorong Liu, Wesley W. Chu and Ruzan Shahinian
UCLA Computer Science Department, Los Angeles, CA, USA 90095
{sliu, wwc, ruzan}@cs.ucla.edu

ABSTRACT

Querying document-centric XML collections with structure conditions improves retrieval precisions. The structures of such XML collections, however, are often too complex for users to fully grasp. Thus, for queries regarding such collections, it is more appropriate to retrieve answers that approximately match the structure and content conditions in these queries, a process also known as vague content and structure (VCAS) retrieval. Most existing XML engines, however, only support content-only (CO) retrieval and/or strict content and structure (SCAS) retrieval. To remedy these shortcomings, we propose an approach for VCAS retrieval using existing XML engines. Our approach first decomposes a VCAS query into a SCAS sub-query and a CO sub-query, then uses existing XML engines to retrieve SCAS results and CO results for the decomposed sub-queries, and finally combines results from both retrievals to produce approximate results for the original query. Further, to improve retrieval precision, we propose two similarity metrics to adjust the scores of CO retrieval results by their relevancies to the path condition for the original query target. We evaluate our VCAS retrieval approach through extensive experiments with the INEX 04 XML collection and VCAS query sets. The experimental results demonstrate the effectiveness of our VCAS retrieval approach.

1. INTRODUCTION

The increasing use of the eXtensible Markup Language (XML) in scientific data repositories, digital libraries and web applications has increased the need for effective retrieving of information from these XML repositories. The *IN*itiative for the *E*valuation of *X*ML retrieval (INEX) [1], for example, was established in April 2002 and has prompted researchers worldwide to promote the evaluation of effective XML retrieval.

XML information can be retrieved by means of either content-only (CO) or content-and-structure (CAS) queries. CO queries, similar to keyword searches in text retrieval, contain only content related conditions. CAS queries contain both content and structure conditions, in which users specify not only what a result should be about (via content conditions) but also what that result is (via structural constraints). Thus, CAS queries are more expressive and have better retrieval precision as demonstrated in past research [10, 11, 13]. Specifying exact structural constraints in queries for document-centric XML collections, however, is not an easy task. Such collections are usually marked up with a large variety of tags. For example, there are about 170 different tags in the INEX document collection. Thus, it is often difficult for users to completely grasp the structure properties of such collections and specify the exact structural constraints in queries. Therefore, for queries regarding such collections, it is more appropriate to retrieve answers that approximately match the structure and content conditions in these queries, a process also known as vague content and structure (VCAS) retrieval. For example, suppose a user is looking for article sections about "internet security." The VCAS retrieval may return article paragraphs about "internet security" to the user, even though they do not strictly satisfy the query's structural constraint (i.e., article sections).

Most existing XML engines, however, only support content only retrieval and/or strict content and structure (SCAS) retrieval. In SCAS retrieval, a query's content conditions can be loosely interpreted, but the query target's structural constraint must be processed strictly. A query target is a special node in the query's structure conditions, whose matching elements in XML collections are returned as results. For example, suppose a user is interested in article sections about "internet security." The SCAS retrieval will not return article paragraphs to the user even though they are relevant to "internet security." Thus, compared to the SCAS retrieval, the new feature in the VCAS retrieval is the approximate processing of a query target's structural constraint. This introduces two challenges to VCAS retrieval: 1) how to extend existing XML engines to derive results that approximately satisfy a query target's structure condition; and 2) how to measure the relevancy of a result to a query target's structural constraint.

Many existing approaches to XML VCAS retrieval can be classified into two categories: 1) content-only approaches (e.g., [12]); and 2) relaxation-based approaches [1, 2]. The former approaches transform a VCAS query into a CO query by ignoring structural constraints; and such approaches are simple because XML engines can be directly used for the VCAS retrieval without any extensions. Such approaches, however, lose the precision benefits that can be derived from XML structures. The latter approaches relax a query's structural constraints and then retrieve the SCAS results for the relaxed queries, which are approximate answers to the original query. Such approaches are systematic and efficient, but they may miss relevant answers due to its strict structural relaxation semantics.

To remedy these problems, in this paper, we propose a general approach that extends existing XML engines for CO and SCAS retrieval to support effective VCAS retrieval. Our approach combines the simplicity advantage provided by CO retrieval and the precision advantage rendered by SCAS retrieval. Our retrieval process consists of three steps:

- **Decomposition**. We decompose a VCAS query into a CO sub-query and a SCAS sub-query such that both sub-queries can be processed by existing XML engines.

Copyright is held by the author/owner.
Eighth International Workshop on the Web and Database (WebDB 2005), June 16-17, 2005, Baltimore, Maryland.

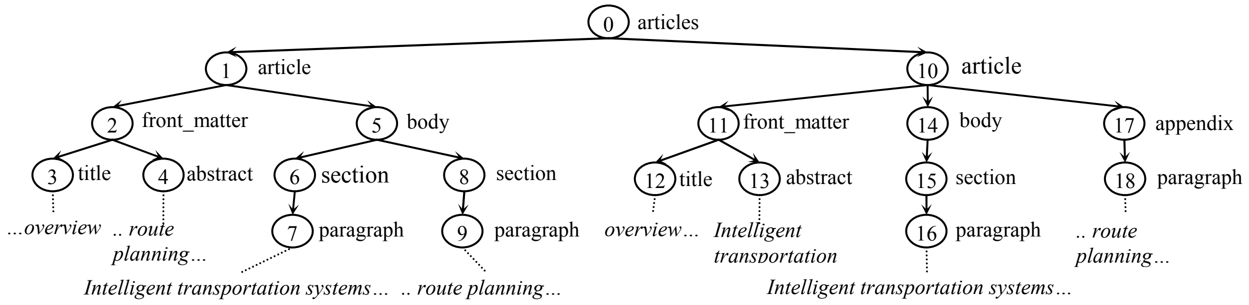


Figure 1: A tree representation of sample XML document collections.

- **Retrieval**. We use existing XML engines to retrieve CO and SCAS results for the two sub-queries.
- **Combination**. Results from the SCAS retrieval are answer to one part of the original query and results from the CO retrieval are approximate answers to the remaining part of the original query. Thus, results from both retrievals can be combined to produce approximate answers to the original query.

To improve retrieval precision, we adjust the score of a CO sub-query result by the relevancy of the result to the path condition for the query target, which is measured by target path similarity. We propose two metrics to compute the target path similarity.

To empirically evaluate the effectiveness of the proposed VCAS retrieval approach, we conduct extensive experiments on the INEX 04 document collection with all the 33 queries in the VCAS task. We use the INEX 04 VCAS relevance assessments as the "gold standard" to evaluate our experimental results.

The rest of the paper is organized as follows. Section 2 introduces the XML data model, query language and VCAS retrieval task. In Section 3, we present our XML VCAS retrieval approach and the similarity metrics. We describe our experimental studies in Section 4. Section 5 overviews related works and Section 6 concludes the paper.

2. BACKGROUND

2.1 XML Data Model

We model an XML document as an ordered, labeled tree where each element (attribute) is represented as a node and each element-to-sub-element (element-to-attribute) relationship is represented as an edge between the corresponding nodes. We represent each node as a triple (*id*, *label*, <*text*>), where *id* uniquely identifies the node, *label* is the name of the corresponding element or attribute, and *text* is the corresponding element's textual content or attribute's value. *Text* is optional because not every element contains textual content. We consider an attribute as a special sub-element of an element and a reference IDREF as a special type of value.

For example, Figure 1 shows a tree representation of a sample XML document collection. Each circle represents a node with the node *id* inside the circle and *label* beside the circle. To distinguish text nodes from element (attribute) nodes, the *text* of a node is linked to the node with a dotted line.

We now introduce the definition for *label path*, which is useful for describing the group representation of an XML tree in Section 3. A *label path* for a node *v* in an XML tree is a sequence of slash-separated labels of the nodes on the path from the root node to *v*. For example, node 6 in Figure 1 can be reached from the root node through the path: node 0 -> 1 -> 5 -> 6. Thus, the label path for node 6 is: `/articles/article/body/section`.

2.2 Query Language

We use a content-oriented XPath-like query language called Narrowed Extended XPath I (NEXI) [14], which is introduced by INEX. NEXI is based on a subset of XPath path expressions [1] with an extension of *about* functions. The syntax of NEXI is:

$$\texttt{path}_1\texttt{[abouts}_1\texttt{]//…//path}_n\texttt{[abouts}_n\texttt{]}$$

where each `path` is a sequence of nodes connected by either parent-to-child ("/") or ancestor-to-descendant ("//") axes; each `abouts` is a Boolean combination of `about` functions.

An *about* function, in the format of *about*`(path, cont)`, requires that a certain context (i.e., `path`) should be relevant to a specific content description (i.e., `cont`). Given an *about* function α, we use `α.path` and `α.cont` to represent its `path` and `cont` parameters respectively. *About* functions have non-Boolean semantics and thus they are the bases for result ranking.

With the introduction of the NEXI query format, now let us look at a sample query in the NEXI format. For example, suppose a user is searching for information on 'route planning' in articles that give an overview of intelligent transportation systems. Since 'route planning' is only one aspect of an intelligent transportation system, the user limits the search on 'route planning' to document components, such as `section`. Thus she formulates her information needs in the following NEXI query Q_1.

```
Q1:  //article[about(.//title, overview) and
about(., intelligent transportation system)]
//body///section[about(., route planning)]
```

With the description of the NEXI query format, we now introduce some notations and terminologies, which are useful for describing our VCAS retrieval methodology in Section 3.

Given a NEXI query `Q` in the format of $\texttt{path}_1\texttt{[abouts}_1\texttt{]}$ $\texttt{//…//path}_n\texttt{[abouts}_n\texttt{]}$, we call the last node on $\texttt{path}_n$, whose matches are returned as results, the *query target*. For example, in Q_1, node `section` is the query target. Further, we define *target content condition*, denoted as $C_t(Q)$, to be the union of the content descriptions in $\texttt{abouts}_n$. For instance, 'route planning' is Q_1's target content condition. Finally, we call $\texttt{path}_1$, …, $\texttt{path}_{n-1}$ the *support paths* and $\texttt{path}_n$ the *target path*. We represent the target path in a query `Q` as $P_t(Q)$. Support paths and the target path provide different structural hints to a search engine: support paths indicate where to search and a target path suggests what to return. For example, in Q_1, `//article` is the support path and `//body//section` is the target path.

2.3 VCAS Retrieval

Specifying structure conditions in a CAS query is not an easy task, in particular for document-centric XML collections with a large variety of tag names. When users specify structural constraints in queries, they often have only a limited knowledge of the structure properties of such collections. In such cases, if we process a query's structure conditions strictly, we may miss results that are not in rigid conformance with the structural constraints, but are highly relevant to users' information needs. Thus, XML vague content and structure (VCAS) retrieval is introduced. The goal of VCAS retrieval is to help users with limited structural knowledge make the maximum utilization of XML structures for more precise retrieval. In VCAS retrieval, both the structure and content conditions can be processed approximately. Thus, the relevancy of a result is judged based on whether it satisfies a user's information needs, but not on whether it strictly conforms to the structural constraints of the query. For example, for the sample query Q_1, a user may judge XML nodes 4, 9 and 18 in Figure 1 to be relevant, although these nodes do not exactly match the structure conditions in Q_1.

3. PROCESSING VCAS QUERIES

In this section, we present a general approach to XML VCAS retrieval, which consists of three steps: decomposition, retrieval and combinations.

3.1 Decomposition

Given a VCAS query `Q`, in principle, both its support paths and target path can be approximately processed. In this paper, we assume that users are strict in their search contexts but flexible in returning answers. Therefore, we process support paths strictly and the target path approximately.

With this assumption, our decomposition strategy is to decompose a VCAS query `Q` into two sub-queries: a CO sub-query, Q^{co}, consisting of the target content condition and a SCAS sub-query, Q^{scas}, consisting of support paths and all the `about` functions associated with these paths. Thus, we can use existing XML engines to perform CO and SCAS retrievals on the decomposed sub-queries respectively to collect XML nodes that approximately satisfy the target path and that strictly conform to the support paths. The following illustrates our decomposition process:

Q: $path_1[abouts_1]//…//path_n[abouts_n]$

Q^{co}: `//*[about(.,`$C_t(Q)$`)]`, where $C_t(Q)$ is the target content condition in Q.

Q^{scas}: $path_1[abouts_1]//…//\ path_{n-1}[abouts_{n-1}]$

For example, the sample query Q_1 in Section 2.2 is decomposed into the following two sub-queries:

Q_1^{co}: `//*[about(., route planning)]`

Q_1^{scas}: `//article[about(.//title, overview) and about(., intelligent transportation system)]`

Q_1^{co} searches for all the XML nodes relevant to 'route planning'; and Q_1^{scas} searches for `article` nodes relevant to 'intelligent transportation system' with a descendant node `title` about 'overview'.

3.2 Retrieval

After the query decomposition step, we use an existing XML IR engine to process the CO sub-query using CO retrieval and the SCAS sub-query using SCAS retrieval. We use our XML IR engine [8] to perform both retrievals. Our VCAS retrieval approach, however, can be used by any XML IR engine. In the following, we first overview our ranking model, and then describe how we apply this model to rank the CO and SCAS retrieval results.

3.2.1 Ranking model

The ranking model used in our XML IR engine is called the extended vector space model. This mode measures the relevancy of an XML node v to an `about` function α, where v satisfies the path condition in α. The model consists of two components: *weighted term frequency* (tf_w) and *inverse element frequency* (*ief*).

Weighted term frequency. Given a term t and an XML node v, suppose there are m different descendant nodes of v, say v_1', v_2', …, v_m', that contain term t in their texts. Let p_i ($1 \leq i \leq m$) be the path from node v to node v_i' and $w(p_i)$ be the weight of path p_i, then the weighted term frequency of term t in node v, denoted as $tf_w(v, t)$, is:

$$tf_W(v,t) = \sum_{i=1}^{m} tf(v_i',t) * w(p_i) \tag{1}$$

That is, the weighted term frequency of a term t in an XML node v is the sum of the frequencies of t in the text of v_i' adjusted by the weight of the path from v to v_i. The weight of a path is the product of the weights of all the nodes on the path, where the weight of a node is user configurable.

Inverse element frequency. The inverse element frequency of a term t in an `about` function α, denoted as $ief(t, \alpha)$, is:

$$ief(t,\alpha) = \log \frac{N_1}{N_2} \tag{2}$$

where N_1 is the number of XML nodes that satisfy the path condition in the `about` function α, i.e., α`.path`; and N_2 is the number of XML nodes that satisfy α`.path` and contain t in texts.

Relevancy score function. The relevancy score of an XML node v to an `about` function α, denoted as $score(v, \alpha)$, is the sum of all the query terms' weighted frequencies in node v adjusted by their corresponding inverse element frequencies. That is,

$$score(v,\alpha) = \sum_{t \in \alpha.cont} tf_W(v,t) * ief(t,\alpha) \tag{3}$$

The extended vector space model is effective in measuring the relevancy scores of XML nodes to `about` functions in SCAS queries [8]. Relevant nodes to such `about` functions, however, usually are of relatively similar sizes because these nodes must satisfy the path conditions of the `about` functions. For example, all the relevant nodes to the `about` function `about(//title, overview)` are `title` nodes. This, however, may not be the case for the `about` function in a CO sub-query Q^{co}. The path condition of the `about` function in Q^{co} is a wildcard, which is so general that all XML nodes are exact matches to the path condition. Thus, nodes relevant to the `about` function in Q^{co} are of varying sizes. The larger a node, the less specific it is to an `about` function. Thus, to compute the relevancy of an XML node v to an `about` function α either in a CO or a SCAS sub-query, we modify the score function in (3) to:

$$score(v,\alpha) = \sum_{t \in \alpha.cont} \frac{tf_W(v,t) * ief(t,\alpha)}{\log_2 wsize(v)} \tag{4}$$

where *wsize(v)* is the weighted size of a node *v*. Given an XML node *v*, suppose *v* has *r* different child nodes $v_1, v_2, .., v_r$. Let size(*v*) be the number of terms in the text in node *v*, then *wsize(v)* is recursively defined as follows:

$$wsize(v) = size(v) + \sum_{i=1}^{r} (wsize(v_i) * w(v_i)) \qquad (5)$$

That is, the weighted size of a node *v* is the text size of node *v* plus the sum of the weighted size of its child node v_i adjusted by their corresponding weights.

3.2.2 CO retrieval

An XML node *v* is relevant to a CO sub-query Q^{co} if either the text of *v* or that of any descendant node of *v* satisfies the content condition in Q^{co}. For example, for the CO sub-query Q_1^{co}, the text of nodes 4, 9 and 18 satisfy the content condition, i.e., `route planning`. Thus, nodes 4, 9 and 18 as well as their ancestor nodes (i.e., nodes 1, 2, 5, 8, 10 and17) are relevant to Q_1^{co}.

A CO sub-query Q^{co} contains only one *`about`* function. Thus, the relevancy score of an XML node *v* to Q^{co}, denoted as *score (v, Q^{co})*, is the relevancy score of *v* to the `about` function in Q^{co}, which can be calculated using (4).

3.2.3 SCAS retrieval

An XML node *v* is relevant to a SCAS sub-query Q^{scas} if it strictly conforms to the structure conditions in Q^{scas} and approximately satisfies the content conditions in Q^{scas}. For example, nodes 1 and 10 in Figure 1 are relevant to Q^{scas}. This is because both nodes strictly conform to the structure conditions: both are `article` nodes with a descendant node `title`. For example, node 1 has a descendant node `title` (i.e., node 3). Also both `article` nodes are about 'intelligent transportation system' and both `title` nodes are on 'overview'.

During query processing, if an XML node *v* is a match to a query node with an *`about`* function α, then the relevancy score of *v* to α is calculated using (4). The relevancy score of a SCAS result *v* to Q^{scas}, denoted as score(*v*, Q^{scas}), is the sum of all the relevancy scores of the corresponding nodes to the *`about`* functions in Q^{scas}. For example, there are two *`about`* functions in Q_1^{scas}:

```
α1: about(//article, intelligent transportation
system)
```

```
α2: about(//article//title, overview)
```

The relevancy score of a SCAS result, say node 1 in Figure 1, to Q_1^{scas} is the relevancy score of node 1 to α_1 plus the relevancy score of node 3 to α_2.

3.3 Combination

After the retrieval step, we have two lists of results: one list of results from the CO retrieval, R^{co}, and another list of results from the SCAS retrieval, R^{scas}. Each result is a pair of (*v*, s), where *v* is an XML node and *s* is the score indicating the relevancy of *v* to a sub-query. For example, for the sample query Q_1, we have two result lists, R_1^{co} and R_1^{scas}, one for each of its sub-queries. R_1^{co} = {(v_4, s_4), (v_9, s_9), (v_{18}, s_{18}), (v_1, s_1), (v_2, s_2), (v_5, s_5), (v_8, s_8), (v_{10}, s_{10}), (v_{17}, s_{17})} and R_1^{scas} = {(v_1, s_1), (v_{10}, s_{10})}, where v_i denotes node i in Figure 1 and s_i is the score for v_i.

Results from the SCAS retrieval are answers to one part of the original query and results from the CO retrieval are approximate answers to the remaining part of the original query. Thus, results from both retrievals can be combined to produce approximate answers to the original query. To do so, we focus on results from the CO retrieval because they are the nodes "matching" the original query's target. For each CO result v_{co}, let v_{scas} be a SCAS result such that v_{co} and v_{scas} are in the same document, then the relevancy of v_{co} to a query Q, denoted as score(v_{co}, Q), is:

$$\text{score}(v_{co}, Q) = f(v_{co}, p_t(Q)) * \text{score}(v_{co}, Q^{co}) + \text{score}(v_{scas}, Q^{scas}) \qquad (6)$$

where f(v_{co}, P_t(Q)) is a *target path similarity* with a value between 0 and 1 that measures how well an XML node v_{co} satisfies the target path in Q, i.e., P_t(Q).

For example, for the sample query Q_1, node 4 in Figure 1 is a result for its CO sub-query Q_1^{co}. Node 1 in Figure 1 is a result for the SCAS sub-query Q_1^{co}, which is in the same document as Node 1. Thus, the relevancy of node 4 (i.e., v_4) to Q_1 can be computed using (6). That is, score(v_4, Q_1) = f(v_4, P_t(Q))*score (v_4, Q_1^{co}) + score(v_1, Q_1^{scas}) = f(v_4, $P_t(Q_1)$)*s_4 + s_1, where s_1 and s_4 are computed using (4).

The target path similarity, f(v_{co}, P_t(Q)), is the key in the combination step. If the label path of an XML node v_{co} is an exact match to P_t(Q), then f(v_{co}, P_t(Q)) =1. It's often the case that the label path of a CO retrieval result v_{co} may not be an exact match to a query target path. In such cases, we compute the target path similarity for a CO retrieval result v_{co} to be the maximum similarity between v_{co} and an XML target node v_t where v_t is an exact match to P_t(Q), denoted as sim(v_{co}, v_t). That is,

$$f(v_{co}, P_t(Q)) = \max\{\text{sim}(v_{co}, v_t) \mid v_t \text{ is an exact match to } P_t(Q)\} \qquad (7)$$

For example, the target path similarity for node 4 (i.e., v_4) is the maximum of sim(v_4 , v_6) and sim(v_4 , v_8) since both nodes v_6 and v_8 match Q_1's target path exactly.

For a given query Q and an XML data tree *D*, there are usually many nodes in *D* whose label paths match the target path in Q exactly. For example, there are about 65470 different nodes in the INEX collection that exactly match the target path in Q_1. Thus, to reduce computations, we cluster XML nodes in *D* with the same label paths into groups similar to DataGuides[7]. For example, Figure 2 is a group representation of the XML data tree in Figure 1. Each rectangle represents a group with its identifier and label next to the rectangle. The numbers inside each rectangle are the identifiers of the nodes in Figure 1.

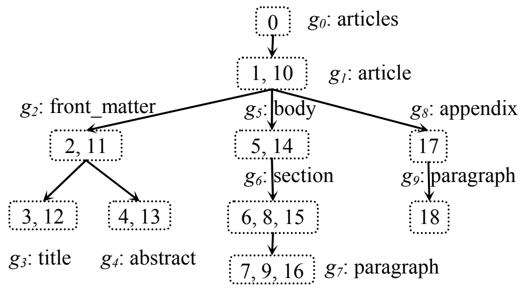


Figure 2: A group representation of the XML tree in Figure 1.

In such a group representation, each group represents a unique label path in *D*. Thus, we can reduce the computations of (7) by measuring the target path similarity of a node v_{co} to be the maximum similarity between the group which v_{co} belongs to, g_{co}, and a *target group* g_t , i.e., a group whose label path is an exact match to P_t(Q). That is,

$$f(v_{co}, P_t(Q)) = \max\{\text{sim}(g_{co}, g_t) \mid g_t \text{ is an exact match to } P_t(Q)\} \quad (8)$$

For example, the target path similarity of node 4 (i.e., v_4) is sim(g_4, g_6) since node v_4 is inside group g_4, and all the nodes that are exact matches to $P_t(Q)$, i.e. node 6 and 8, are in group g_6.

In the following, we introduce two methods to compute group similarities by considering groups' path and content aspects.

3.3.1 Path similarity

The similarity between a group g_{co} and a target group g_t, sim(g_{co}, g_t), can be computed based on the similarity between their corresponding label paths. Let $p_{g_{co}}$ and p_{g_t} be the label path of group g_{co} and g_t respectively. The greater number of common prefix nodes these two paths share, the more similar the two groups are. Thus, sim(g_{co}, g_t) is:

$$\text{sim}(g_{co}, g_t) = \frac{|p_{g_{co}} \cap p_{g_t}|}{|p_{g_{co}}| + |p_{g_t}| - |p_{g_{co}} \cap p_{g_t}|} \quad (9)$$

where $|p_{g_{co}} \cap p_{g_t}|$ represents the number of common prefixing nodes between $p_{g_{co}}$ and p_{g_t}; $|p_{g_{co}}|$ and $|p_{g_t}|$ denote the number of nodes on the paths $p_{g_{co}}$ and p_{g_t}. The denominator in (9) is used for the normalization purpose such that sim(g_{co}, g_t) = 1 when $g_{co} = g_t$.

3.3.2 Content similarity

For document-centric XML collections, the path similarity may not be very accurate in estimating group similarity. For example, given three paths p_1: `/article/body/section/title`, p_2: `/article/body/section` and p_3: `/article/body/section/paragraph`, p_1 is as similar to p_2 as p_3 to p_2 according to (9). If a user is looking for a `section` regarding specific content, then according to (9), a `title` will have the same target path similarity as a `paragraph`. Compared to a `title`, a `paragraph`, however, is a better approximation for a `section`. This is because the content of a `paragraph` is much closer to that of a `section` than the content of a `title` to that of a `section`.

This motivates us to measure the similarity between two groups based on their corresponding content. We describe the content of a group g_i via a N-vector $\overrightarrow{g_i} = (tf_{i1}, tf_{i2}, \ldots, tf_{iN})$, where N is the total number of distinct terms in an XML collection and tf_{ik} ($1 \le k \le N$) represents the frequency of term tf_{ik} in group g_i. With this vector representation of a group's content, the content similarity between two groups, g_{co} and g_t, can be estimated via the cosine of their corresponding content vectors:

$$\text{sim}(g_{co}, g_t) = \frac{\overrightarrow{g_{co}} \circ \overrightarrow{g_t}}{\sqrt{\overrightarrow{g_{co}} \circ \overrightarrow{g_{co}}}\sqrt{\overrightarrow{g_t} \circ \overrightarrow{g_t}}} \quad (10)$$

For example, using (10), we find that the similarity between a `section` group and a section's `title` group in the INEX document collection is 0.4196, while the similarity between the `section` group and a `section`'s `paragraph` group is 0.991.

4. EXPERIMENTAL STUDIES

4.1 Experimental Dataset

We use the INEX 04 dataset and all the 33 VCAS queries to evaluate the effectiveness of our VCAS retrieval methodology. The INEX 04 dataset, around 500MByte in size, consists of over 12,000 computer science articles from 21 IEEE Computer Society journals. The documents are marked with about 170 different tags. A document contains 1532 elements on average and an element has an average depth of 6.9.

4.2 Test Runs

The following four runs are used to study the effectiveness of our VCAS retrieval methodology. All the experiments use the same node weight configurations: uniform weights. That is, $w(v) = 1$ for any node v in the dataset.

- *CO run*. In this run, we ignore the structure conditions in a query and use the query's content conditions to perform CO retrieval. This run is used as the baseline for testing the effectiveness of our VCAS retrieval methodology.
- *VCAS-1 run*. In this run, we perform the VCAS retrieval with f=1 for all results. The run is used as a base line to compare the effectiveness of the path similarity and content similarity metric.
- *VCAS-path run*. In this run, we perform the VCAS retrieval using the path similarity in (8) as the target path similarity.
- *VCAS-cont run*. In this run, we perform the VCAS retrieval using the content similarity in (9) as the target path similarity.

4.3 Result Evaluation and Analysis

To evaluate the relevancy of an XML document component to a query topic, the relevance assessment working group in INEX has proposed a two-dimension relevancy metric (*exhaustiveness*, *specificity*). *Exhaustiveness* measures the extent to which the document component discusses the topic of request and *specificity* measures the extent to which the document component focuses on the topic of request. This two-dimension metric is then quantized to a single relevancy value between 0 and 1. In this paper, we use two of the most frequently used quantization methods: *strict* and *generalized*. A relevancy value is either 0 or 1 with a strict quantization; while it could be 0, 0.25, 0.5, 0.75 or 1 with a generalized quantization.

In our experiments, we use the INEX relevance assessment set version 3.0 and compute each run's mean average precision (MAP) using INEX on-line evaluation tools. Table 1 presents mean average precisions over all of the 33 query topics using both strict and generalized quantization methods. The corresponding ranks compared to all the 51 official submissions returned by other INEX participating systems are also included.

Table 1: Results over all the 33 VCAS topics in INEX 04.

Run	Strict		Generalized	
	MAP	Rank	MAP	Rank
CO	0.064	11	0.0716	7
VCAS-1	0.0844 (+31.88%)	5	0.0878 (+22.63%)	5
VCAS-path	0.0886 (+38.44%)	4	0.0887 (+23.88%)	5
VCAS-semanticst	0.0946 (+47.81%)	4	0.094 (+31.28%)	5

From Table 1, we note that our VCAS retrieval approach significantly outperforms the CO approach. The *VCAS-1* run outperforms the *CO* run by 31.88% using the strict quantization metric. This is because the CO approach ignores XML structures for simplicity but loses the precision benefit provided by XML structures. Further, by comparing the *VCAS-1* run with the *VCAS-path* and *VCAS-cont* runs, we note that similarity measures further improve our VCAS retrieval precisions. Also, the content similarity provides more precision improvement than the path similarity for the INEX VCAS retrieval task. We note that the mean average precisions of our VCAS retrieval approach are relatively high compared to all the 51 official INEX submissions. For example, the mean average precision of the *VCAS-cont* run ranks top 4 (5) using the strict (generalized) quantization method. We have also observed similar results using other quantization methods.

5. RELATED WORKS

There is a large body of work on XML information retrieval (e.g.,[3-6, 8-13]), most of which focuses on effective XML CO retrieval and SCAS retrieval. For example, Sigurbjörnsson et al propose a general methodology for processing content-oriented XPath queries [11]. The key difference between [11] and our methodology is that: [11] focuses on extending IR engines designed for CO retrieval to support SCAS retrieval; while our methodology extends XML engines designed for CO and SCAS retrievals to support VCAS retrieval.

XML VCAS retrieval is a new task in INEX 04. Many teams within the INEX initiative conducted VCAS retrievals by ignoring the query structure conditions (e.g., [12]). In [9], S. Geva proposed a VCAS retrieval approach by decomposing a query into multiple sub-queries, where each sub-query contains one structure filter and one content filter. An XML element is a result for a sub-query if it satisfies the content filter, but does not necessarily have to satisfy the structure filter. Results from different sub-queries are merged and sorted by the number of filters they satisfy. This approach is simple and effective. Our work differs [9] in two aspects: the query decomposition strategies are different; and two similarity metrics are proposed to measure the relevancy of a VCAS result to a query target path for improving retrieval precision. No such measure is used in [9].

Query relaxation is also related with XML VCAS retrieval. S. Amer-Yahia et al have some seminal studies on XML query relaxation in [1, 2]. They model a XML query as a tree and relax node and/or edge constraints on the query tree to derive approximate answers. Algorithms have been proposed to efficiently derive top-k approximate answers. Our work differs from [1, 2] in that while they focus more on the efficiency aspect, we focus on the effectiveness (i.e., retrieval precision) aspect.

6. CONCLUSION

In this paper, we propose an approach for processing XML vague content and structure (VCAS) retrieval. A content and structure (CAS) query consists of two parts, i.e., support and target, where each part contains both path and content conditions. To derive approximate answers to a query, we decompose a query into two sub-queries: one sub-query consisting of support path and content conditions (a SCAS sub-query) and another sub-query consisting of the target content condition (a CO sub-query). We then process the SCAS sub-query by SCAS retrieval and the CO sub-query by CO retrieval. Results from both retrievals are combined to produce approximate results to the original query. To improve retrieval precision, we adjust the score of a CO retrieval result by the relevancy of the result to the target path condition of the original query, which is measured by target path similarity. We propose a path similarity and a content similarity metric to compute the target path similarity. We evaluate our VCAS retrieval approach and the similarity metrics through extensive experiments on the INEX 04 dataset and all the 33 VCAS queries. Our experimental results demonstrated that: 1) our VCAS retrieval approach, by taking advantage of XML structures, significantly outperforms the content-only approach; and 2) the path and content similarity metrics are effective in estimating the relevance of CO sub-query results to a query target path constraint. Therefore, they can be used to further improve the accuracy of the ranking of the retrieved results.

REFERENCES

[1] S. Amer-Yahia, S. Cho, and D. Srivasava. Tree pattern relaxation. In *EDBT*, 2002.

[2] S. Amer-Yahia, L. V. S. Lakshmanan, and S. Pandit. FleXPath: Flexible Structure and Full-Text Querying for XML. In *SIGMOD*, 2004.

[3] R. Baeza-Yates, N. Fuhr, and Y. Maarek. Second Edition of the XML and IR Workshop. In *SIGIR Forum*, 2002.

[4] D. Carmel, Y. S. Maarek, M. Mandelbrod, Y. Mass, and A. Soffer. Searching XML Documents via XML Fragments. In *SIGIR*, 2003.

[5] D. Carmel, A. Soffer, and Y. Maarek. XML and Information Retrieval. Workshop Report. In *SIGIR Forum*, Fall 2000.

[6] N. Fuhr, M. Lalmas, S. Malik, and Z. Szlavik (eds.) INitiative for the Evaluation of XML Retrieval (INEX). *Proceedings of the Third INEX Workshop*, 2004.

[7] R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *VLD*B, 1997.

[8] S. Liu, Q. Zou, and W. W. Chu. Configurable Indexing and Ranking for XML Information Retrieval. In *SIGIR*, 2004.

[9] S. Geva. GPX – Gardens Point XML Information Retrieval at INEX 2004. In [6].

[10] T. Schlieder and H. Meuss H. Querying and Ranking XML Documents. In *Journal of American Society for Information Science and Technology*, Volume 53 (6) pp. 489-503, 2002.

[11] B. Sigurbjörnsson, J. Kamps, and M. de Rijke. Processing Content-Oriented XPath Queries. In *CIKM*, 2004.

[12] B. Sigurbjörnsson, J. Kamps, and M. de Rijke. The University of Amsterdam at INEX 04. In [6]

[13] A. Trotman. Searching structured documents. *Information Processing and Management*, 40:619-632, 2004.

[14] A. Trotman and B. Sigurbjörnsson. Narrowed Extended XPath I (NEXI). In [6].

[15] INitiative for the Evaluation of XML Retrieval. http://qmir.dcs.qmul.ac.uk/INEX.

[16] XPath. http://www.w3.org/TR/xpath

On the Expressive Power of Node Construction in XQuery

Wim Le Page Jan Hidders Philippe Michiels[*] Jan Paredaens Roel Vercammen[*]
University of Antwerp
Middelheimlaan 1
B-2020 Antwerp, Belgium

{wim.lepage, jan.hidders, philippe.michiels, jan.paredaens, roel.vercammen}@ua.ac.be

ABSTRACT

In the relational model it has been shown that the flat relational algebra has the same expressive power as the nested relational algebra, as far as queries over flat relations and with flat results are concerned [6]. Hence, for each query that uses the nested relational model and that, with a flat table as input always has a flat table as output, there exists an equivalent flat query that only uses the flat relational model. In analogy, we study a related flat-flat problem for XQuery: for each expression containing operations that construct new nodes and whose XML result contains only original nodes, there exists an equivalent "flat" expression in XQuery that does not construct new nodes.

Categories and Subject Descriptors

H.2 [**Information Systems**]: Database Management; H.2.3 [**Database Management**]: Languages—*Query languages*

Keywords

XQuery,XML,Expressive power

1. INTRODUCTION

As XQuery [1] is becoming the standard language for querying XML documents, it is important to study the properties of this powerful query language. In XQuery, a query can have a result containing nodes not occurring in the input. These new nodes are constructed during the evaluation of the expression. Nevertheless, it is still possible that only original nodes occur in the final result. We call such expressions *node-conservative*. For example, the query in Example 1.1 creates new nodes not occuring in the result. In this example we perform a join and a projection of two XML documents in XQuery.

In this paper we show that for each deterministic node-conservative expression there exists an expression without node construction that essentially always returns the same store and result sequence. For example, the query in Example 1.1 can be rewritten to the query shown in Example 1.2. In this work we will show how to generate automatically equivalent constructor-free expressions for node-conservative expressions. This result gives an indication of the expressive power of the node construction. Furthermore it can be interesting for query optimization, since optimizing node construction can be hard. For example, in [3] a translation fom a subset if XQuery to SQL is given, where the construction of new elements yields SQL statements with special numbering operations which are relatively hard to optimize.

Example 1.1 Node-Conservative Expression

The following XQuery expression

```
let $jointtable :=
        element {"table"}{
            for $b1 in doc("table.xml")/table/row
            for $b2 in doc("table2.xml")/table/row
            where $b1/a = $b2/a
            return element{"row"}{$b1/*,$b2/*} }
return
    for $b in $jointtable/row/b return string($b)
```

has the result sequence "one","two"when given the input documents table.xml and table2.xml which look as follows

```
<table>                             <table>
<row><a>1</a><b>one</b></row>       <row><a>1</a><c>red</c></row>
<row><a>2</a><b>two</b></row>       <row><a>2</a><c>blue</c></row>
<row><a>3</a><b>three</b></row>     </table>
</table>
```

Example 1.2 Constructor-Free Expression

The following XQuery expression

```
for $b1 in doc("table.xml")/table/row
for $b2 in doc("table2.xml")/table/row
where $b1/a = $b2/a
return
   for $b in ($b1/*, $b2/*)/b return string($b)
```

is equivalent to the query of Example 1.1 and does not contain node constructors.

The work in this paper was inspired by simular results for the nested relational algebra [6, 7]. In [6] it is shown that each nested algebra expression that has a flat relation as output when applied to a flat relation, is equivalent to a flat algebra expression. In [7] a very direct proof is given of this fact using a simulation technique. Other work studied the effect of adding object creation to query languages on the expressive power of these languages. For example, in [2] the effect of object identity on the power of query

[*]Philippe Michiels and Roel Vercammen are supported by IWT – Institute for the Encouragement of Innovation by Science and Technology Flanders, grant numbers 31016 and 31581.

languages is studied and a notion of determinate transformations is introduced as a generalization of the standard domain-preserving transformations. However, obvious extensions of complete database programming languages with object creation are not complete for determinate transformations. In [8] this mismatch is solved by introducing the notion of constructive transformations, a special kind of determinate transformations which are precisely the transformations that can be expressed by these obvious extensions.

This paper is structured as follows. In Section 2 we discuss LiXQuery, which we will use as a formal model for XQuery and for proving our theorems. Section 3 contains the theorem and proof for the elimination of node construction in expressions that do not contain newly constructed nodes in their results. Finally, the conclusion of this work is presented in Section 4.

2. LIXQUERY

We use LiXQuery [4, 5] as a basis for studying the expressive power of node construction in XQuery. LiXQuery is a sublanguage of XQuery that has a semantics that is consistent with that of XQuery, has the same expressive power as XQuery and has a compact and well defined syntax and semantics. The LiXQuery language was designed with the audience of researchers investigating the expressive power of XQuery in mind. The XQuery features that are omitted in LiXQuery are only those that are not essential from a theoretical perspective. We claim that the results that we show for LiXQuery also hold for XQuery.

LiXQuery has only a few built-in functions and no primitive data-types, order by clause, namespaces, comments, programming instructions and entities. Furthermore it ignores typing and only provides `descendant-or-self` and `child` as navigational axes. The other navigational axes can be simulated using these 2 axes. Although the features that LiXQuery lacks, are important for practical purposes, they are not relevant to our problem. Note that LiXQuery does support recursive functions, positional predicates and atomic values, which are essential in our approach.

We define LQE as the set of LiXQuery expressions. In LiXQuery, *expressions* are evaluated against an *XML store* and an *evaluation environment*. The XML store contains the fragments that are created as intermediate results, as well as the entire web. The store that only contains the entire web is called the *initial XML store*. The evaluation environment essentially contains mapping information for function names, variable names and the context item (including context position in the context sequence and the context sequence size). Formally, the XML store is a 6-tuple[1] $St = (V, E, \ll, \nu, \sigma, \delta)$ where: V is the set of available nodes; (V, E) forms an acyclic directed graph to represent the tree-structures; $\ll$ defines a total order over the nodes in V; ν labels element and attribute nodes with their node name; σ labels the attribute and text nodes with their string value; δ is a partial function that uniquely associates with an URI or a file name, a document node.

[1]This tuple is the same as in [5] except that the sibling order $<$ is replaced by the document order $\ll$.

The environment in LiXQuery is denoted by a tuple $Env = (a, b, v, x, k, m)$ where a is a partial function that maps a function name to its formal argument; b is a partial function that maps a function name to the body of the function; v is a partial function that maps variable names to their values; x is an item of St and indicates the context item or x is undefined; k is an integer denoting the position of the context item in the context sequence or k is undefined; m is an integer denoting the size of the context sequence, or m is undefined.

The result of an expression evaluated against an XML store and environment is a (possibly expanded) XML store (*result store*) and a sequence of one or more items over the result store (*result sequence*). Items in the result sequence can either be atomic values or nodes. The semantics of a LiXQuery expression is defined by statements of the form $St, Env \vdash e \Rightarrow St', v$, which state that when e is evaluated against a store St and an environment Env then St' is the result store and v is the result sequence over St. We derive such statement by using inference rules, which are given in [5].

We denote the empty sequence by $\langle\rangle$, non-empty sequences by, for example, $\langle 1, 2, 3\rangle$ and the concatenation of two sequences l_1 and l_2 by $l_1 \circ l_2$. Last but not least, each node has a unique identity. It is important to note that atomic values do not have an identity.

3. ELIMINATING NODE CONSTRUCTION

We will show that some XQuery expressions that contain node constructors can be simulated by another XQuery expression that does not use node construction.

3.1 Node Conservative Expressions

Clearly an expression cannot be simulated by an expression without constructors if it returns newly created nodes, so we introduce the notion of *node conservative expressions*.

Definition 1. A *node-conservative expression* (NCE) is an expression $e \in LQE$ such that for all stores St and environments Env it holds that if $St, Env \vdash e \Rightarrow St', v$ then all nodes in v are nodes in St.

In Example 1.1 we considered a join and a projection of two XML documents in LiXQuery. This expression is an example of a NCE.

Another restriction we make is that we only consider deterministic expressions. Node creation is a source of non-determinism in LiXQuery (and XQuery) because the fragment that is created by a constructor is placed at an arbitrary position in document order between the already existing trees in the store. Since node construction is the only source of non-determinism in LiXQuery, it is clear that we cannot simulate that there are many possible results without it. This is however not a fundamental feature of XQuery so we ignore non-deterministic expressions.

Definition 2. An expression $e \in LQE$ is said to be *deterministic* if for every store St and environment Env it holds that if $St, Env \vdash e \Rightarrow St', v$ and $St, Env \vdash e \Rightarrow St'', w$ then $v = w$.

Note that this is a very strict definition of determinism which, in fact, only allows node-conservative expressions. We could have allowed multiple results that were equivalent up to isomorphism over the nodes, but this would make things unnecessarily complex.

Next to restricting the types of expressions we consider we also allow a simulation to differ in its semantics from the the original in two ways. The first is that a simulation may have a defined result where the original does not. Note that we still require that whenever an expression has a defined result then the simulation has the same defined result, but not necessarily the reverse. We conjecture that the theorem also holds when we also require the reverse but proving this would add a lot of overhead to this paper without adding much extra insight in the expressive power of node construction.

The second way in which the semantics of a simulation differs from that of the original is that resulting stores only have to be the same up to garbage collection, i.e., after removing the trees that are not reachable by the δ function (the `fn:doc()` function) or contain nodes from the result sequence. If we denote the store that results from garbage collection on a store St and a result sequence v as $[St]_v$ then this leads to the following definition:

Definition 3. Given two expression $e, e' \in LQE$ we say that e' is a *simulation of* e if for all stores St and environments Env with undefined x, k and m it holds that if $St, Env \vdash e \Rightarrow St', v$ then there exists a store St'' such that $St, Env \vdash e' \Rightarrow St'', v$ and $[St'']_v = [St']_v$.

We use this definition for the following theorem, which is the main result of this paper:

THEOREM 1. *For every deterministic node-conservative*[2] *expression $e \in LQE$ there exists a simulation $e' \in LQE$ that does not contain constructors.*

3.2 Outline of the Simulation

Our goal is to transform an expression into a semi-equivalent expression which does not use node constructions. We are going to eliminate the construction by simulating it. To simulate construction we will need to simulate the store, because it is there that the information concerning the newly constructed nodes will reside. In short, the simulation performs the following steps:

1. We use a few special variables in the environment to encode a part of the store. This part will contain the newly created nodes but also parts of the old store that are retrieved with the `doc()` function;
2. Whenever a `doc()` call occurs in the original expression, the simulation will add the encoding of the document tree to the simulated store on the condition that it is not already there;
3. Accessing nodes in the store is simulated by accessing the encoded store;
4. Nodes are simulated by node identifiers which are numbers that refer to the encoded nodes in the store;
5. In order to be able to distinguish encoded atomic values from node identifiers within sequences, we let the normal atomic values be preceded by a `0` and the node identifiers by a `1`. Note that this means that in the simulation, a sequence will be twice as long and every item that was at position i will now be at position $2i$;
6. Finally, the simulation replaces the node identifiers with the corresponding nodes from the store. If the original expression is indeed a deterministic node conservative expression, the result – and thus also the result of the simulation – will contain no newly constructed nodes. Consequently, this last step is always possible if the original expression is node conservative.

[2]Since every deterministic expression is also node-conservative we can strictly speaking drop the second requirement.

The transformation of an expression to a constructor-free expression that simulates it, is expressed by a transformation function. A transformation function is a function $\epsilon : LQE \rightarrow LQE$. The commuting diagram in Figure 1 illustrates what should hold for such a transformation function ϵ for it to be correct. We show this by induction on the subexpressions e'' of an expression e.

$$\begin{tikzcd}
(St, Env) \arrow[r, "\tau"] \arrow[d, "e''"'] & (\widehat{St}, \widehat{Env}) \arrow[d, "\epsilon(e'')"'] \\
(St', v) \arrow[r, "\tau'"] & (\widehat{St}, \widehat{v})
\end{tikzcd}$$

Figure 1: This diagram depicts the relations between the several components in the translation.

On the left-hand side we see that starting from a store St and an environment Env, the evaluation of the expression e'', which may add new nodes to St, will result in a new store $St' \supseteq St$ and a result v. On the right-hand side we see that starting from a store $\widehat{St}$ and an environment $\widehat{Env}$, the evaluation of the transformed constructor-free expression $\epsilon(e'')$, which will not add new nodes to $\widehat{St}$, will result in the same store $\widehat{St}$ and a result $\widehat{v}$.

At the top of the diagram we see the encoding τ which encodes a store $St_{\widehat{Env}} \subseteq St$ into sequences of atomic values that are bound to special variables in the environment $\widehat{Env}$. Moreover, τ replaces the values of all variables in Env with sequences of atomic values and the bodies of all functions are transformed by ϵ to constructor-free expressions. At the bottom of the diagram we see the encoding τ' which encodes a store $St_{\widehat{v}} \subseteq St'$ and the value v as a sequence of atomic values $\widehat{v}$.

When we use this schema to show by induction that we can correctly translate an expression e to a constructor-free expression $\epsilon(e)$ it will hold for the evaluation of the subexpression e'' that $\widehat{St}$ is the store against which e is evaluated. Moreover, if during the evaluation of e nodes where created before the evaluation of e'' then (1) these nodes have been added to St and (2) in the evaluation of

$\epsilon(e)$ they were added to the encoded store in $\widehat{Env}$. So it will hold that $St = \widehat{St} \cup St_{\widehat{Env}}$. Obviously it has to be shown by induction that this remains true after the evaluation of e'' so it has to be shown that $St' = \widehat{St} \cup St_{\widehat{v}}$. An overview of all these relationships between the involved stores is illustrated in Figure 2.

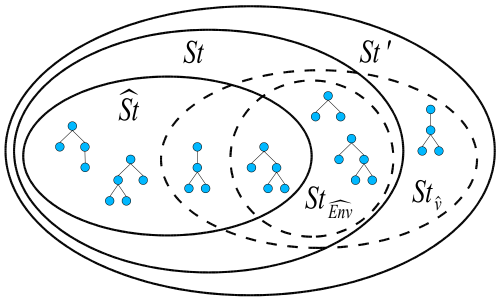


Figure 2: The stores $\widehat{St}$, $St_{\widehat{Env}}$, St, $St_{\widehat{v}}$ and St'

3.3 Encoding the Store and Environment

Before we describe how to translate LiXQuery expressions into their constructor-less simulations, we first have to look into the encodings of the store and environment based on their formal semantics.

We first describe how to encode a store in sequences of atomic values. We will define this given an injective function $id : V \to \mathbb{N}$ that provides the unique node identifier for each node and which will be used to represent the nodes in the encoding.

Definition 4. Given an XML store $St = (V, E, \ll, \nu, \sigma, \delta)$ and an injective function $id : V \to \mathbb{N}$ then we call a tuple of XML values $(\hat{V}, \hat{E}, \hat{\delta})$ a *store encoding of St under id* if

– $\hat{V} = \langle id(v_1), t_1, n_1, s_1\rangle \circ \ldots \circ \langle id(v_k), t_k, n_k, s_k\rangle$ where (1) $\{v_1, \ldots, v_k\} = V$, (2) $v_1 \ll \ldots \ll v_k$, (3) t_i equals `"text"`, `"doc"`, `"attr"` or `"elem"` if v_i is a text node, a document node, an attribute node or an element node, respectively, (4) n_k is $\nu(v_k)$ if it is defined and `""` otherwise, and (5) s_k is $\sigma(v_k)$ if it is defined and `""` otherwise,

– $\hat{E} = \langle id(v_1), id(v'_1)\rangle \circ \ldots \circ \langle id(v_m), id(v'_m)\rangle$ where $\{(v_1, v'_1), \ldots, (v_m, v'_m)\} = E$,

– $\hat{\delta} = \langle s_1, id(v_1)\rangle \circ \ldots \circ \langle s_p, id(v_p)\rangle$ where $\delta = \{(s_1, v_1), \ldots, (s_p, v_p)\}$.

Note that a store encoding is not uniquely determined given St and id because we can choose the order in $\hat{E}$ and $\hat{\delta}$.

We have to encode sequences of atomic values and nodes as sequences of atomic values. When we directly replace each node v with $id(v)$ we cannot always tell if a number represents itself or encodes a node identifer. Therefore we let atomic values that encode themselves be preceded by `0` and atomic values that are node identifiers be preceded by `1`. For illustration consider the examples in Example 3.1.

Example 3.1 Encoded values

Given a function $id = \{(v_1, 5), (v_2, 3)\}$:

value	*value encoding*
$\langle 5\rangle$	$\langle$`0`, `5`$\rangle$
$\langle v_1\rangle$	$\langle$`1`, `5`$\rangle$
$\langle 5, v_1,$ `"string"`$, v_2\rangle$	$\langle$`0`, `5`, `1`, `5`, `0`, `"string"`, `1`, `3`$\rangle$

Definition 5. Given an XML value $v = \langle x_1, \ldots, x_k\rangle$ over a store $St = (V, E, \ll, \nu, \sigma, \delta)$ and an injective function $id : V \to \mathbb{N}$, we call an XML value $\tilde{v}$ the *value encoding of v under id* if $\tilde{v} = \langle m_1, \hat{x}_1\rangle \circ \ldots \circ \langle m_k, \hat{x}_k\rangle$ where $m_i = 1$ and $\hat{x}_i = id(x_k)$ if x_k is a node and $m_i = 0$ and $\hat{x}_i = x_i$ otherwise.

Note that the encoding of value v is written as $\tilde{v}$ and not as $\hat{v}$ to distinguish it from the $\hat{v}$ in the commuting diagram in Figure 1 which encodes both a store and a value.

We now proceed with formalizing the the τ relationship in Figure 1. Recall that the relations in this diagram hold by induction on the subexpressions e'' of a simulated expression e. The resulting store $\widehat{St}$ is the store against which e is evaluated, because all nodes that are created by e'' are in $\epsilon(e'')$ encoded in $\widehat{Env}$. We will refer to the part of St encoded in $\widehat{Env}$ as $St_{\widehat{Env}}$. Since $St_{\widehat{Env}}$ describes the part of St that is retrieved or created by preceding evaluations it holds that $St = \widehat{St} \cup St_{\widehat{Env}}$ where $\widehat{St} \cap St_{\widehat{Env}}$ contains the documents that were retrieved with the `doc()` function before e'' was evaluated (see Figure 2).

Definition 6. Given a store $St = (V, E, \ll, \nu, \sigma, \delta)$, an environment $Env = (a, b, v, x, k, m)$ over this store and a transformation function ϵ we call a pair $(\widehat{St}, \widehat{Env})$ with store $\widehat{St}$ and environment $\widehat{Env} = (\hat{a}, \hat{b}, \hat{v}, \hat{x}, \hat{k}, \hat{m})$ a *store-environment encoding of St and Env under tr* if there is a store $St_{\widehat{Env}}$ and an injective function $id : V_{\widehat{Env}} \to \mathbb{N}$ such that

– $St = \widehat{St} \cup St_{\widehat{Env}}$,

– all nodes in values of variables in Env are in $St_{\widehat{Env}}$

– $\hat{a} = a$,

– $\hat{b} = \{(s, tr(y)) | (s, y) \in b\}$,

– in $\hat{v}$ (1) all variable names s bound by v are bound to the value encoding of $v(s)$ under id, (2) the variables `tau:E`, `tau:V` and `tau:delta` contain $\hat{V}$, $\hat{E}$ and $\hat{\delta}$, respectively, where $(\hat{V}, \hat{E}, \hat{\delta})$ is the store encoding of $St_{\widehat{Env}}$ under id and (3) the variables `tau:x`, `tau:k` and `tau:m` contain value encodings of x, k and m, respectively, under id, and

– $\hat{x}$, $\hat{k}$ and $\hat{m}$ are all undefined.

In turn, we now define the τ' encoding in Figure 1. Here we refer to the part of the store that is encoded in the environment as $St_{\widehat{v}}$. Since $St_{\widehat{v}}$ describes the part of St that is retrieved or created by preceding evaluations it must hold that $St' = \widehat{St} \cup St_{\widehat{v}}$ where $\widehat{St'} \cap St_{\widehat{v}}$ contains the documents that were retrieved with the `doc()` function before or during e'' was evaluated (see Figure 2).

Definition 7. Given a store $St' = (V, E, \ll, \nu, \sigma, \delta)$ and a value v over this store then a pair $(\widehat{St}, \widehat{v})$ with a store $\widehat{St}$ and an XML value $\widehat{v}$ is called a *store-value encoding of* St *and* v if there is a store $St_{\widehat{v}}$ and an injective function $id : V_{\widehat{v}} \to \mathbb{N}$ such that (1) $St' = \widehat{St} \cup St_{\widehat{v}}$, (2) all nodes in v are in $St_{\widehat{v}}$ and (3) $\hat{v} = \langle|V|\rangle \circ \hat{V} \circ \langle|E|\rangle \circ \hat{E} \circ \langle|\delta|\rangle \circ \hat{\delta} \circ \tilde{v}$ where $(\hat{V}, \hat{E}, \hat{\delta})$ is the store encoding of $St_{\widehat{v}}$ under id, and $\tilde{v}$ is the value encoding of v under id.

Based on this input/output encoding we can give the formal meaning of the diagram in Figure 1 and define when a transformation function defines a correct simulation.

Definition 8. A transformation function ϵ is said to be a *correct transformation* if it holds for every store St and environment Env that if $St, Env \vdash e \Rightarrow St', v$ and $(\widehat{St}, \widehat{Env})$ is store-environment encoding of St and Env under tr then it holds that $\widehat{St}, \widehat{Env} \vdash \epsilon(e) \Rightarrow \widehat{St}, \widehat{v}$ where $(\widehat{St}, \widehat{v})$ is a store-value encoding of St' and v.

3.4 A Correct Transformation Function

In this section we construct a transformation function $\epsilon : LQE \to LQE$ and show that the following theorem holds.

THEOREM 2. *The transformation function ϵ is a correct transformation function.*

The result of $\epsilon(e)$ is defined by induction upon the structure of e. Because of space limitations we will only show some typical translations for some types of LiXQuery expressions. Helper functions will be defined in the `eps` namespace which is assumed to be distinct from all the used namespaces in e.

We begin with the translation of the `name()` function. Here and in the following we will assume the existence of the functions `eps:V()`, `eps:E()`, `eps:delta()` and `eps:val()` which respectively extract $\widehat{V}$, $\widehat{E}$, $\widehat{\delta}$, and $\widetilde{v}$ from a store-value encoding. For computing the store-value encoding give $\hat{V}$, $\hat{E}$, $\hat{\delta}$ and $\tilde{v}$ we assume the existence of a function `eps:stValEnc()` with formal arguments `$V, $E, $delta` and `$val`. We also introduce the shorthand

```
let $eps:V, E, delta, val := getStVal($eps:res)
```

to denote

```
 let $eps:V := eps:V($eps:res)
 let $eps:E := eps:E($eps:res)
 let $eps:delta := eps:delta($eps:res)
 let $eps:val := eps:val($eps:res)
```

The translation of the `name()` function is defined as follows:

```
ε(name(e′)) =
  let $eps:res := ε(e′)
  let $eps:V, E, delta, val := getStVal($eps:res)
  return eps:epsStValEnc($eps:V, $eps:E, $eps:delta,
           eps:nu($eps:val[2], $eps:V) )
```

The function `eps:nu()` returns the name of the specified node using the information encoded in $\widehat{V}$.

The `doc()` function loads new documents into our encoded store.

```
ε(doc(e)) = let $eps:res := ε(e)
            return eps:doc($eps:res)
```

Here the function `eps:doc()` checks if the document is already in the encoded store by comparing the URI's tot the URI's already present in $\widehat{\delta}$. If this is the case it just returns the associated simulated node id as found in $\widehat{\delta}$, else the `eps:doc()` function compares the real document node obtained with the given URI, to the real documents obtained via the URI's that are already present in $\widehat{\delta}$. If this is the case, only a new entry is added to $\widehat{\delta}$ linking the new URI to the node identifier of the encoded document. If the document is not present in $\widehat{\delta}$ the document is encoded. First a document node is added to the encoded store and with the resulting node identifier a new entry is added in $\widehat{\delta}$. Then, also using this identifier, the nodes of the document are encoded and added after this document node in $\widehat{V}$. The `eps:doc()` function finally returns a store-value encoding containing the (new) node identifier as the result sequence and the (updated) store, environment and delta.

The for-expression is the most fundamental type of expression in LiXQuery. In it's translation we assume a number x that is unique for each for-expression that has to be translated. This is used to define for every for-expression a unique function `eps:for`$_x$`()`. The parameter $vars_x$ represent all free variables in e'. Recursion is used here to simulate the iteration over a sequence where the resulting store of the previous step is passed on to the following step. The translation of the for-expression is then defined as follows.

```
ε(for $s at $s′ in e return e′) =
  let $eps:res := ε(e)
  let $eps:V,E,delta,val := getStVal($eps:res)
  return eps:for_x(1, $eps:val, $eps:V, $eps:E,
       $eps:delta, vars_x)
```

with `eps:for`$_x$`()` defined as follows:

```
declare function eps:for_x($eps:pos, $eps:seq,
           $tau:V, $tau:E, $tau:delta, vars_x) {
  let $s := $eps:seq[$eps:pos*2-1], $eps:seq[$eps:pos*2]
  let $s' := $eps:pos
  let $eps:res1 := ε(e′)
  let $eps:V1,E1,delta1,val1 := getStVal($eps:res1)
  let $eps:res2 := eps:for_x($eps:pos+1, $eps:seq,
           $eps:V1, $eps:E1, $eps:delta1, vars_x)
  let $eps:V2,E2,delta2,val2 := getStVal($eps:res2)
  return $eps:stValEnc($eps:V2, $eps:E2, $eps:delta2,
             ( $eps:val1, $eps:val2 ))
}
```

The translation of node comparison expressions is done by extracting the information of identity and position contained in the store-value encoding.

```
ε(e' is e'') =
  let $eps:res := ε(e')
  let $tau:V,E,delta,val1 := getStVal($eps:res1)
  let $eps:res2 := ε(e'')
  let $tau:V2,E2,delta2,val2 := getStVal($eps:res2)
  return $eps:stValEnc( $tau:V2, $tau:E2, $tau:delta2,
          (0, $tau:val1[2] = $tau:val2[2]))
```

The translation of a construction operator extends the encoded store which is a crucial part of the simulation. To illustrate this we give the translation the element construction.

```
ε(element {e′}{e″}) =
  let $eps:res := ε(e′)
  let $tau:V,E,delta,val1 := getStVal($eps:res)
  let $eps:res2 := ε(e″)
  let $V2,E2,delta2,val2 := getStVal($eps:res2)
  return eps:addElem(V2, E2, delta2, $tau:val1, val2)
```

with `eps:addElem()` declared as follows.

```
declare function eps:addElem($V $E, $delta,
                 $nameEnc, $chEnc) {
  let $res1 := eps:addElemNode($nameEnc[2], $V, $E)
  let $V1,E1,delta1,val1 := getStVal($res1)
  let $res2 := eps:addChl($val1, $chEnc, V1, E1 )
  let $V2,E2,delta2,val2 := getStVal($res2)
  return $eps:stValEnc( $V2, $E2, $delta, $val1)
}
```

Here the function `eps:addElemNode($name, $V, $E)` adds a new element node with name `$name` and returns a store-value encoding with the new store and the new node identifier. The function `eps:addChl($parEnc, $chEnc, $V, $E)` makes deep copies for all the nodes encoded in `chEnc`, adds these under the node encoded in `$parEnc` and returns a store-value encoding with the new store and the parent node. The function uses recursion in the same way as the translation of the for-expression, in order to be able to iterate with side-effects on the store.

3.5 Creating a Constructor-Free Expression

We now sketch how to create constructor-free semi-equivalent expressions for deterministic (node-conservative) ones, i.e., how to generate the expression e' of Theorem 1, based on $\epsilon(e)$, which is working on an encoding of (St, Env). We do so by showing how encoding (St, Env) and afterwards decoding results St', v can be done for node-conservative expressions.

The expression $\epsilon(e)$ will be evaluated against $(St, \widehat{Env})$, where $\widehat{Env}$ contains the encoded store and environment. We construct $St_{\widehat{Env}}$ in such a way that it contains exactly all trees of St for which a node occurs in the variable bindings of Env. Assuming that we can have a sequence that is the concatenation of all variable bindings in Env, we can write an expression to create a new sequence `$roots` that, starting from the former sequence, filters out the nodes, applies the `root` function to each node and finally sorts this result by document order by applying a self-axis step. Since the roots of all trees that have to be in $St_{\widehat{Env}}$ are now in document order in `$roots`, we can write another expression that creates the encoded store $St_{\widehat{Env}}$ starting from an empty encoded store, by simply traversing through the trees under the nodes in `$roots` and extending $St_{\widehat{Env}} = (\widehat{V}, \widehat{E}, \widehat{\delta})$, represented by the variables `$tau:V`, `$tau:E` and `$tau:delta` in the environment $\widehat{Env}$). If this traversal is done in depth-first, left-to-right manner, we visit all nodes of St that will be encoded in $St_{\widehat{Env}}$ in document order. Node identifiers can then be chosen in such a way that they correspond to the position in $St_{\widehat{Env}}$. Starting from Env, we can now create the encoded environment $\widehat{Env}$ by replacing all expressions in b by the simulations $\epsilon(b)$, adding the variables for the encoded store and environment to the function signatures in a, replacing all sequences in the variable bindings with their encoded sequences, and finally, adding the variables `$tau:V`,`$tau:E` and `$tau:delta` to v. Since all nodes that occur in Env are encoded in $St_{\widehat{Env}}$ and node identifiers were assigned based on the position of nodes within the forest under `$roots`, we can easily obtain the encoded sequences for the variable bindings.

The result of the evaluation of $\epsilon(e)$ is the store St and a store-value encoding $St_{\widehat{v}}$. Based on this we can create the result sequence the original expression returned if it was a node-conservative expression. In that case the encoded result sequence will only contain encoded nodes of which the real counterparts were available in the initial XML store St. Therefore we can loop over encoded items in $St_{\widehat{v}}$. Encodings of atomic values are simply replaced by the atomic values itself. For every encoded node we first determine whether it was originally in $St_{\widehat{Env}}$. This can be done by storing (during the encoding phase) all nodes and their chosen node identifiers as pairs in a variable. If the node identifier occurs in this variable then it is an original node and we can easily return the corresponding node. If the root of the encoded node is an encoded document node that is associated to a URI in the variable `$tau:delta` then we can obtain the original document root node by a simple `doc` function call, else it is a newly created node and hence this expression is not a node-conservative expression. By using the position of the encoded node relative to the encoded root node, we can determine the position of the corresponding real node in the document tree and hence we replace the encoded node by the real node in the result sequence.

4. CONCLUSION

In this paper, we showed that deterministic XQuery expressions, always yielding a result with only nodes from the input store, can be rewritten to equivalent expressions that do not contain node constructors. In further research, we plan to investigate whether a similar result can also be obtained for non-recursive XQuery. Furthermore, we intend to investigate how this result can be used to optimize queries by removing or postponing node creation operations in query evaluation plans. Finally, we want to examine whether this result can be used for rewriting `let` expressions in non-recursive XQuery without using XQuery functions. This is not trivial, since simple variable substitution would result in multiple creation of the nodes on the right-hand side of the variable assignment.

5. REFERENCES

[1] XML query (XQuery). `http://www.w3.org/XML/Query`.

[2] S. Abiteboul and P. C. Kanellakis. Object identity as a query language primitive. *Journal of the ACM*, 45:798–842, September 1998.

[3] T. Grust, S. Sakr, and J. Teubner. XQuery on SQL hosts. In *Proceedings of the 30th Int'l Conference on Very Large Databases (VLDB 2004)*, August/September 2004 2004.

[4] J. Hidders, J. Paredaens, P. Michiels, and R. Vercammen. LiXQuery: A formal foundation for XQuery research. *SIGMOD Record*, September 2005.

[5] J. Hidders, J. Paredaens, R. Vercammen, and S. Demeyer. A light but formal introduction to XQuery. In *Proceedings of the Second International XML Database Symposium (XSym 2004)*, Toronto, Canada, 2004. Springer.

[6] J. Paredaens and D. Van Gucht. Converting nested algebra expressions into flat algebra expressions. *ACM Transactions on Database Systems (TODS)*, 17:65–93, 1992.

[7] J. Van den Bussche. Simulation of the nested relational algebra by the flat relational algebra, with an application to the complexity of evaluating powerset algebra expressions. *Theoretical Computer Science*, 254:363–377, 2001.

[8] J. Van den Bussche, D. Van Gucht, M. Andries, and M. Gyssens. On the completeness of object-creating database transformation languages. *Journal of the ACM*, 44:272–319, March 1997.

Indexing for XML Siblings

SungRan Cho
L3S, University of Hannover
scho@l3s.de

ABSTRACT

Efficient querying XML documents is an increasingly important issue considering the fact that XML becomes the de facto standard for data representation and exchange over the Web, and XML data in diverse data sources and applications is growing rapidly in size. Given the importance of XPath based query access, Grust proposed R-tree index, we refer to as *whole-tree indexes* (WI). Such index, however, has a very high cost for the following-sibling and preceding-sibling axes. In this paper we develop a family of index structures, which we refer to as *split-tree indexes* (SI), to address this problem, in which (i) XML data is horizontally split by a simple, yet efficient criteria, and (ii) the split value is associated with tree labeling. While the SI is straightforward to construct, it incurs the overlap problem between bounding boxes. We resolve this problem by designing the *transformed split-tree indexes* (TSI). We also study the most promising existing method of constructing R-tree, the Hilbert tree, so that we take advantage of its benefit for XML siblings. Lastly, we experimentally demonstrate the benefits of the TSI for siblings over the WI using benchmark data sets.

1. INTRODUCTION

With the advent of XML as the de facto standard for data representation and exchange over the Web, querying XML documents has become more important. In this context, XML query evaluation engines need to be able to efficiently identify the elements along each location step in the XPath query. Several index structures for XML documents have been proposed [4, 5, 9, 10, 13, 15], in a way to efficiently querying XML documents.

As XML documents are modeled by a tree structure, a numbering scheme, labeling tree elements, allows for managing the hierarchy of XML data. For example, each element has the position, a pair of its beginning and end locations in a depth first search. In general, the numbering approach has the benefit of easily determining the ancestor-descendant relationship in a tree. In this respect, R-tree index using node's preorder and postorder, we refer to as *whole-tree indexes* (WI), has been proposed in [5]. Such index, however, does not consider issues related to the costs of the preceding-sibling and following-sibling axes.

In this paper, we discuss index techniques to reduce the cost of performing XML siblings. This also addresses an issue of what efficient packing for XML tree data is. An efficient packing method for a tree is not only to group together data elements which are close in a tree, but also to reduce dead space resulting in false positives (no data in indexed space). The packing method of the WI, taking a whole tree, may cover considerable dead space, which influences querying XML siblings. We design the *split-tree index* (SI) to address the problem, in which (i) an XML tree is horizontally split by a simple, but efficient criteria, and (ii) the split value is associated with tree labeling. The SI uses standard R-tree index lookup algorithms to match elements along XPath location steps.

Since data trees are indexed separately in the SI, bounding boxes representing data elements may overlap, which impacts the performance. We resolve this problem by developing the *transformed split-tree index* (TSI), in which all elements are transformed into new dimensions. To take advantage of the semantics of the index structure, we develop index lookup for XPath axes in the TSI.

We next consider the most promising existing method of constructing R-tree for XML siblings, the Hilbert tree, which preserves data locality well in dimensions. While the WI, SI, and TSI use the preorder to cluster node elements, the Hilbert R-tree uses the Hilbert ordering generated from node's positional numbers in a tree. It is shown in our experiment that the Hilbert R-tree has an even page access across all XPath axes.

Finally, we perform an experimental comparison between the WI, SI, TSI, and Hilbert R-tree. Using the XMark benchmark data set, we demonstrate that the index lookup costs in the TSI is superior to the WI for siblings and comparable to other axes in the WI.

This paper is organized as follows. Section 2 presents some background material along with the structure of our indexes (WI, SI, and TSI) and motivates our indexes (SI and TSI). In Section 3, we present the split criteria of a tree as the base of building the SI. In Section 4, we design a new index TSI by enhancing SI and presents the index lookup for the TSI. Section 5

Eighth International Workshop on the Web and Databases (WebDB 2005), June 16-17, 2005, Baltimore, Maryland.

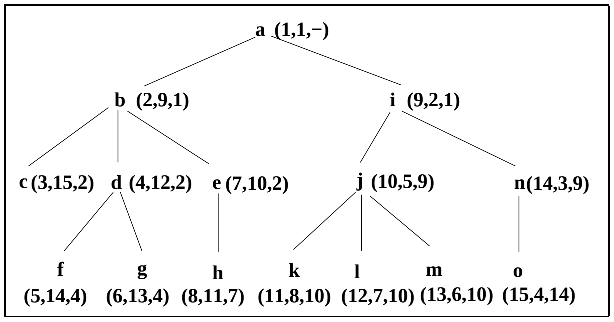


Figure 1: XML data tree encoded with Ln, Rn, PLn

describes the Hilbert R-tree for XML tree. The benefits of SI, TSI, and Hilbert indexes over WI are evaluated experimentally in Section 6. We present related work in Section 7 and conclude in Section 8.

2. BACKGROUND

We assume familiarity with XML and XPath expressions. Here, we review the encodings used on XML nodes to facilitate matching of XPath axes.

2.1 Whole-Tree Indexes: WI

A preorder and a postorder numbering of nodes (see, e.g., [9, 5]) in an (ordered, tree-structured) XML document suffice to reconstruct the XML document unambiguously.

In this paper, we use an encoding scheme, Ln and Rn, for nodes in XML documents that has the same effect as preorder and postorder. Ln is the rank at which the node is encountered in a *left to right* depth first search (DFS) of the XML data tree, and Rn is the rank at which the node is encountered in a *right to left* DFS. In order to handle level sensitive matching, such as child and parent axes (matching nodes one level apart), and following-sibling and preceding-sibling axes (matching nodes with the same parent), the parent node's Ln, written as PLn is associated with each node. Thus each XML element node is labeled with three numbers: Ln, Rn, and PLn. These numbers become coordinates in multi-dimensions. Figure 1 is an example of XML data tree where each node is encoded with Ln, Rn, and PLn.

The *whole-tree index* (WI) is obtained by the R-tree index on the (Ln, Rn, PLn) dimensions, in which XML data is loaded using Ln ordering.

2.2 Matching XPath Location Steps on WI

XML nodes are packed in a *bounding box* which denotes the set of leaf pages rooted at a non-leaf index page entry. Non-leaf pages in the index structure maintain the low and high ranges, represented in (B_Ln_{low}, B_Rn_{low}, B_PLn_{low}, B_Ln_{high}, B_Rn_{high}, B_PLn_{high}). Given a query node $Q_{Ln,Rn,PLn}$ and a non-leaf index page, let us review the conditions, for matching XPath location steps along eight XPath axes (i.e., descendant, child, ancestor, parent, preceding, following, preceding-sibling, and following-sibling).

- **descendant:** ($B\_Ln_{high} > Q_{Ln}$) & ($B\_Rn_{high} > Q_{Rn}$).
- **child:** ($B\_Ln_{high} > Q_{Ln}$) & ($B\_Rn_{high} > Q_{Rn}$) & ($B\_PLn_{low} \leq Q_{Ln} \leq B\_PLn_{high}$).
- **ancestor:** ($B\_Ln_{low} < Q_{Ln}$) & ($Rn_{low} < Q_{Rn}$).
- **parent:** ($B\_Ln_{low} < Q_{Ln}$) & ($Rn_{low} < Q_{Rn}$) & ($B\_Ln_{low} \leq Q_{PLn} \leq B\_Ln_{high}$).
- **preceding:** ($B\_Ln_{low} < Q_{Ln}$) & ($B\_Rn_{high} > Q_{Rn}$).
- **following:** ($B\_Ln_{high} > Q_{Ln}$) & ($B\_Rn_{low} < Q_{Rn}$).
- **preceding-sibling:** ($B\_Ln_{low} < Q_{Ln}$) & ($B\_Rn_{high} > Q_{Rn}$) & ($B\_PLn_{low} \leq Q_{PLn} \leq B\_PLn_{high}$).
- **following-sibling:** ($B\_Ln_{high} > Q_{Ln}$) & ($B\_Rn_{low} < Q_{Rn}$) & ($B\_PLn_{low} \leq Q_{PLn} \leq B\_PLn_{high}$).

Axes	Leaf	Axes	Leaf
ancestor	3.15	descendant	1.16
parent	1	child	1.11
preceding	846	preceding-sibling	7.56
following	847	following-sibling	166.9

Table 1: Leaf page accesses

2.3 Motivation

We first ran the XMark benchmark dataset (see http://monetdb.cwi.nl/xml/) with 10 MBytes dataset and a page capacity of 100 nodes to observe the number of page accesses for sibling axes in the WI. We counted the number of leaf page accesses of the index tree needed to find the results while exploring XPath axes. The leaf page access results for XPath location steps are given in Table 1.

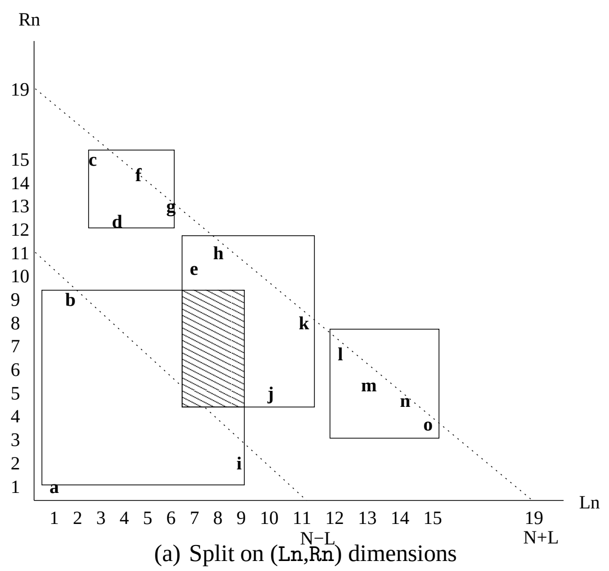
As observed in Table 1, the WI has a high cost of the following-sibling and preceding-sibling, on the average 166 page accesses for the following-sibling and 7 page accesses for the preceding-sibling. However the ancestor, parent, child, and descendant axes rather do well (on the average between 1 and 3 page accesses). For preceding and following axes, a node has (on the average) half of the document preceding and following it, respectively. The experiment results motivate us to propose a new indexing technique that can reduce the cost of XML siblings. Next we would like to discuss a question of why the costs of the preceding-sibling and following-sibling axes are not symmetric. One reason for this is that false positives (due to R-tree bounding boxes) of following-sibling axis, is much higher than those in other axes.

3. SPLIT-TREE INDEXES: SI

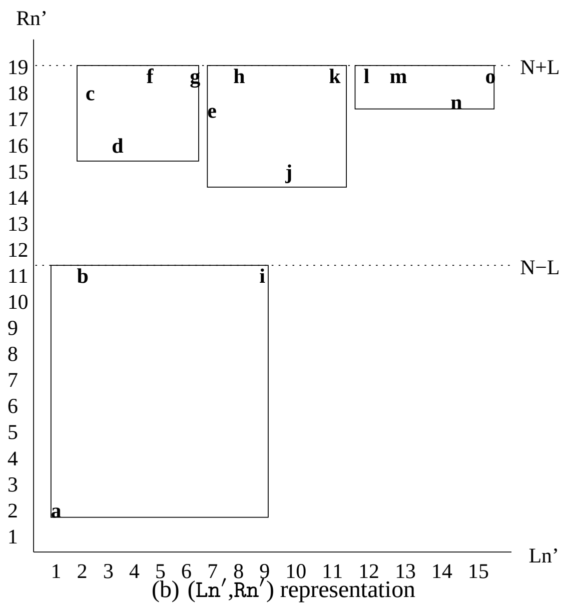
In this section, we discuss a split strategy for XML data tree as the base of obtaining the *split-tree indexes* (SI).

3.1 An Horizontal Split Strategy

An XML data tree is split using node's Ln and Rn. A key of the split strategy is α where α can be the total number of nodes in a tree, the maximum level of a tree, or etc. Since the value of α is still an open issue, we don't specify the value of α in this paper. Instead we will study the costs of XPath



(a) Split on (Ln,Rn) dimensions



(b) (Ln′,Rn′) representation

Figure 2: (Ln,Rn) and (Ln′,Rn′) dimensions

axes with varying α (experiment results provide more details in Section 6).

XML data nodes on the $(\mathtt{Ln}, \mathtt{Rn}, \mathtt{PLn})$ dimensions are divided on the center of the line $\mathtt{Rn} = -\mathtt{Ln} + \alpha$, and in effect the data tree is split horizontally. For example, Figure 2 (a) is a 2-dimensional representation of the data tree of Figure 1, where x and y axes represent Ln and Rn, respectively. It shows that the dimensions are divided by the line $\mathtt{Rn} = -\mathtt{Ln} + (\mathtt{N} - \mathtt{L})$, where N is the total number of nodes and L is the maximum level in the tree. The property of the split regions in this example is that all leaf nodes and some intermediate nodes in the data tree lie above the line $\mathtt{Rn} = -\mathtt{Ln} + (\mathtt{N} - \mathtt{L})$, which might be dense (we refer to as an *upper region*), and some intermediate nodes lie under the line, which might be sparse (we refer to as a *lower region*).

3.2 Designing SI

The *split-tree index* (SI) is constructed by indexing XML data nodes in the upper and lower regions separately. The separate packing reduces long thin boundary boxes, produced by WI, that may contain dead space (space which is indexed but does not have data). As for index lookup, the SI uses the same WI lookup algorithms (see Section 2.2) to identify the desired elements along an XPath axis from a specified element.

Since we pack each region separately, we obtain the overlap between bounding boxes at each region of the tree. For example, Figure 2 (a) where the pack capacity is four nodes, shows an overlap area highlighted in a shaded rectangle. Due to overlap, multiple paths from the root downwards on the SI may need to be traversed, which results in increasing page accesses. We address this problem by designing the TSI next.

4. TRANSFORMED SI: TSI

In this section we design a family of transformed split-tree indexes (TSI) to avoid possible overlap in the SI.

4.1 Coordinate Transformation

New coordinates of nodes are determined by their origins. A node element n with coordinates $(\mathtt{Ln}, \mathtt{Rn}, \mathtt{PLn})$ is transformed into n'=$(\mathtt{Ln}', \mathtt{Rn}', \mathtt{PLn}')$, such that

$$\mathtt{Ln}' = \mathtt{Ln}$$
$$\mathtt{Rn}' = \mathtt{Ln} + \mathtt{Rn}$$
$$\mathtt{PLn}' = \mathtt{PLn}$$

The original coordinates are extended with in the Rn direction with respect to Ln. Thus the new dimensions, $(\mathtt{Ln}', \mathtt{Rn}', \mathtt{PLn}')$, are at most $N \times L \times N$ larger than the original dimensions, where N is the total number of nodes and L is the maximum level in the tree. More importantly this result does not break the hierarchy of a tree represented in multi-dimensions. Figure 2 (b) shows the tree on $(\mathtt{Ln}', \mathtt{Rn}')$ dimensions transformed from the data tree of Figure 2 (a). In the transformed dimensions, without allowing overlap, TSI is constructed in manner of building SI whose packing is based on Ln ordering.

4.2 Matching XPath Location Steps on TSI

We discuss the conditions for matching XPath location steps along XPath axes on $(\mathtt{Ln}',\mathtt{Rn}',\mathtt{PLn}')$ dimensions. First, we present a comparison of XPath axes crossed on between (Ln,Rn) and $(\mathtt{Ln}',\mathtt{Rn}')$ dimensions, which is shown in Figure 3. This suggests the new conditions for XPath axes in the TSI. Given a query node $Q_{\mathtt{Ln}',\mathtt{Rn}',\mathtt{PLn}'}$ and a non-leaf index page represented in $(\mathtt{B\_Ln}'_{\mathtt{low}},\mathtt{B\_Rn}'_{\mathtt{low}},\mathtt{B\_PLn}'_{\mathtt{low}}, \mathtt{B\_Ln}'_{\mathtt{high}},\mathtt{B\_Rn}'_{\mathtt{high}}, \mathtt{B\_PLn}'_{\mathtt{high}})$, we develop the conditions for XPath location steps along eight XPath axes (more details are provided in Appendix).

- **descendant:** $(\mathtt{B\_Rn}'_{\mathtt{high}} - \mathtt{B\_Ln}'_{\mathtt{low}} > \mathtt{Q}_{\mathtt{Rn}'} - \mathtt{Q}_{\mathtt{Ln}'})$ & $(\mathtt{B\_Ln}'_{\mathtt{high}} > \mathtt{Q}_{\mathtt{Ln}'})$ & $(\mathtt{B\_Rn}'_{\mathtt{high}} > \mathtt{Q}_{\mathtt{Rn}'})$.

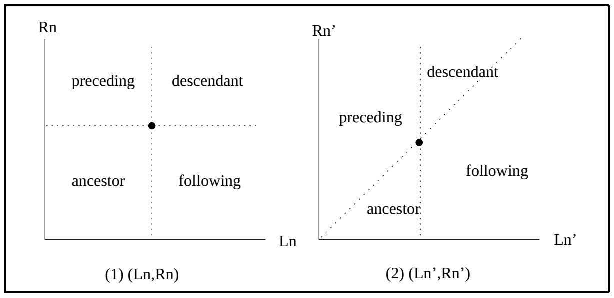


Figure 3: XPath conditions on ($\mathtt{Ln}$,$\mathtt{Rn}$) and ($\mathtt{Ln}'$,$\mathtt{Rn}'$)

- **child:** $(\mathtt{B\_Rn}'_{\mathtt{high}} - \mathtt{B\_Ln}'_{\mathtt{low}} > \mathtt{Q}_{\mathtt{Rn}'} - \mathtt{Q}_{\mathtt{Ln}'})$ & $(\mathtt{B\_Ln}'_{\mathtt{high}} > \mathtt{Q}_{\mathtt{Ln}'})$ & $(\mathtt{B\_Rn}'_{\mathtt{high}} > \mathtt{Q}_{\mathtt{Rn}'})$ & $(\mathtt{B\_PLn}'_{\mathtt{low}} \leq \mathtt{Q}_{\mathtt{Ln}'} \leq \mathtt{B\_PLn}'_{\mathtt{high}})$.
- **ancestor:** $(\mathtt{B\_Rn}'_{\mathtt{low}} - \mathtt{B\_Ln}'_{\mathtt{high}} < \mathtt{Q}_{\mathtt{Rn}'} - \mathtt{Q}_{\mathtt{Ln}'})$ & $(\mathtt{B\_Ln}'_{\mathtt{low}} < \mathtt{Q}_{\mathtt{Ln}'})$ & $(\mathtt{B\_Rn}'_{\mathtt{low}} < \mathtt{Q}_{\mathtt{Rn}'})$.
- **parent:** $(\mathtt{B\_Rn}'_{\mathtt{low}} - \mathtt{B\_Ln}'_{\mathtt{high}} < \mathtt{Q}_{\mathtt{Rn}'} - \mathtt{Q}_{\mathtt{Ln}'})$ & $(\mathtt{B\_Ln}'_{\mathtt{low}} < \mathtt{Q}_{\mathtt{Ln}'})$ & $(\mathtt{B\_Rn}'_{\mathtt{low}} < \mathtt{Q}_{\mathtt{Rn}'})$ & $(\mathtt{B\_Ln}'_{\mathtt{low}} \leq \mathtt{Q}_{\mathtt{PLn}'} \leq \mathtt{B\_Ln}'_{\mathtt{high}})$.
- **preceding:** $(\mathtt{B\_Rn}'_{\mathtt{high}} - \mathtt{B\_Ln}'_{\mathtt{low}} > \mathtt{Q}_{\mathtt{Rn}'} - \mathtt{Q}_{\mathtt{Ln}'})$ & $(\mathtt{B\_Ln}'_{\mathtt{low}} < \mathtt{Q}_{\mathtt{Ln}'})$.
- **following:** $(\mathtt{B\_Rn}'_{\mathtt{low}} - \mathtt{B\_Ln}'_{\mathtt{high}} < \mathtt{Q}_{\mathtt{Rn}'} - \mathtt{Q}_{\mathtt{Ln}'})$ & $(\mathtt{B\_Ln}'_{\mathtt{high}} > \mathtt{Q}_{\mathtt{Ln}'})$.
- **preceding-sibling:** $(\mathtt{B\_Rn}'_{\mathtt{high}} - \mathtt{B\_Ln}'_{\mathtt{low}} > \mathtt{Q}_{\mathtt{Rn}'} - \mathtt{Q}_{\mathtt{Ln}'})$ & $(\mathtt{B\_Ln}'_{\mathtt{low}} < \mathtt{Q}_{\mathtt{Ln}'})$ & $(\mathtt{B\_PLn}'_{\mathtt{low}} \leq \mathtt{Q}_{\mathtt{PLn}'} \leq \mathtt{B\_PLn}'_{\mathtt{high}})$.
- **following-sibling:** $(\mathtt{B\_Rn}'_{\mathtt{low}} - \mathtt{B\_Ln}'_{\mathtt{high}} < \mathtt{Q}_{\mathtt{Rn}'} - \mathtt{Q}_{\mathtt{Ln}'})$ & $(\mathtt{B\_Ln}'_{\mathtt{high}} > \mathtt{Q}_{\mathtt{Ln}'})$ & $(\mathtt{B\_PLn}'_{\mathtt{low}} \leq \mathtt{Q}_{\mathtt{PLn}'} \leq \mathtt{B\_PLn}'_{\mathtt{high}})$.

5. USING HILBERT R-TREE

In this section, we present the Hilbert R-tree for XML siblings. The Hilbert curve is a space filling curve that visits all the points in *k*-dimensional space exactly once and never crosses itself. The order has been used to arrange data elements in many applications such as image processing, CAD, and etc. The reason that the Hilbert ordering preserves spatial locality is to map points which are close together in *k*-dimensional space into points that also close together in one-dimensional space. In [7], the Hilbert R-tree is constructed in manner to index smaller regions of space so that search can be focused on the relevant regions. It thus minimizes the resulting page boundaries.

In order to build Hilbert R-tree for XML data, we first generate the Hilbert values of data nodes in ($\mathtt{Ln}, \mathtt{Rn}, \mathtt{PLn}$) dimensions and then load the data into the index using the Hilbert ordering. The same index lookup algorithm for XPath location steps, described in Section 2.2, is used. Consequently, from our experiment, the resulting costs across all XPath axes (except for the following and preceding axes) are almost even (this result is explained in next section).

α	l-region	u-region
N	16,447	152,757
N-L/2	9,439	159,765
N-L	5,613	163,591
N-2L	2,203	163,591
N-4L	472	168,732
N-8L	82	169,122

Table 2: Node counts at each region

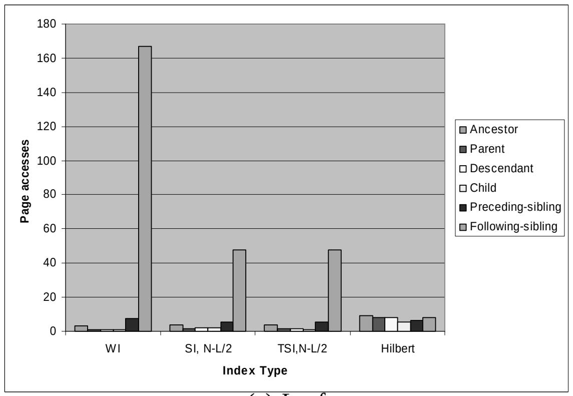
6. EXPERIMENTS

In this section, we present the result of the evaluation of our index. We conducted experiment using the XMark benchmark dataset. The size of the XMark is about 10 MBytes and the packing size is 100 nodes. In our experiment, we compare WI, SI, TSI, and Hilbert indexes.

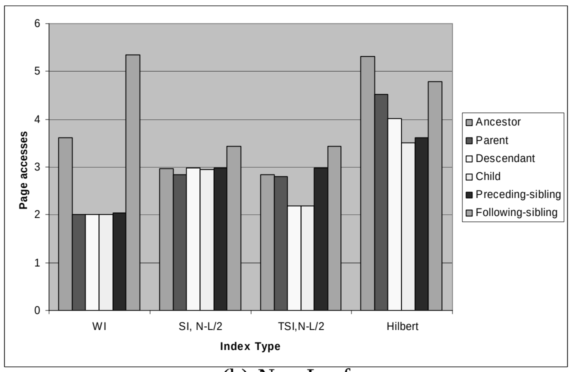
We used six XPath navigation axes for query, i.e., child, parent, ancestor, descendant, following-sibling and preceding-sibling axes. The results of following and preceding are excluded because the costs of those axes are the same on WI, SI, TSI, and Hilbert indexes. We also observed our evaluation with varying the value of α (the split value of a tree).

The leaf and non-leaf page access results for six axes over WI, SI, TSI, and Hilbert are given in Figure 5 (a) and (b), respectively, in which the value of α is chosen as $N - L/2$ for SI and TSI. From the plot (a), the SI, TSI and Hilbert indexes outperform the WI for the following-sibling and preceding-sibling axes. The number of leaf page accesses is roughly the same for the following-sibling and preceding-sibling axes between SI and TSI. When compared to WI, the following-sibling and preceding axes are about 3.5 times and 1.5 times cheaper respectively in the SI and TSI, whereas those are about 20 times and 1.1 times cheaper respectively in the Hilbert. More importantly the Hilbert has the symmetric results between the preceding-sibling and following-sibling axes due to its packing property. As for other axes, the number of page access in the SI and TSI is slightly higher than that in WI index (about 1.2 times larger for ancestor, child, descendant and 1.8 times for parent). This is expected in the split R-tree, since nodes that are close in the hierarchy, may be split and then packed separately. The Hilbert is between 3 and 8 times larger than the WI for other axes. From our results, the overlap affects the results of the descendant and child axes, but it nearly does for other axes. The TSI has a saving of 50% over the SI for the descendant and child axes. The number of non-leaf page accesses between WI and TSI is roughly the same, except for the following-sibling which is 36% cheaper in the TSI.

Next we measured the cost of executing XPath navigation axes with varying the value of α in the TSI. Associating with α, the number of nodes in the lower and upper regions is given in Table 2, where N is the total number of nodes and L is the maximum level of the tree. As one decreases the value of α, the number of leaf page accesses of the preceding-sibling and following-sibling axes increases, but that of the parent, ancestor, child, descendant axes decreases. For example, in our experiment, when α is $N - 8L$, in the TSI the ancestor, parent, descendant and child axes are up to 1.6 times smaller than when α is $N - L/2$. However, the preceding-sibling and following-



(a) Leaf



(b) Non-Leaf

Figure 4: WI, SI, TSI, and Hilbert

sibling axes are about 1.5 times and 3.5 times larger respectively with α of $N - 8L$. The result graph is shown in Figure 5 (a). The number of non-leaf page accesses is approximately the same with varying α, which is given in Figure 5 (b).

7. RELATED WORK

In this section, we primarily discuss the core problems in tree structured data. Research on indices for a tree-structured data is mainly divided into two classes: (i) a numbering based index, which assigns meaningful numbers to tree nodes as identifiers; (ii) a prefix based index for paths.

Recently index techniques using tree labeling have become a focus of research in efficiently answering XPath queries. One classical numbering based index is the level-based index (LBI) structure in [11], which decomposes the data into several levels indicating their nesting (i.e., descendant nodes are nested) and indexes the data at each level separately. Li et al. [9] proposed more flexible numbering scheme using a pair of preorder and postorder to efficiently process regular path expression queries. Wang et al. [13] developed ViST, a dynamic indexing method for XML documents, by representing both XML data and queries in structure encoded sequences. Cohen et al. [3] proposed a dynamic labeling scheme, which is useful for maintenance of index.

There are several proposals for prefix based indexing [8, 4, 16]. In [8], an identifier of an ancestor is a prefix of the identifiers of its descendants. The problem with this method is that as the length of identifiers increases, the cost increases. In [4], paths which are sequences of element tags are encoded as strings and are indexed. Deschler et al. [16] proposed MASS (a multiple axis storage structure) indexing structure that supports XPath querying and XML document updates.

8. CONCLUSION

In this paper we have considered the problem of constructing an efficient R-tree index for XPath siblings. The main idea has been to group together XML data which are close to each other in the hierarchy as well as to contain less dead space. In this context, we developed TSI, which is constructed over a horizontally split tree. We also considered an existing R-tree technique, the Hilbert tree, in order to take advantage of its clustering method for siblings. Our preliminary experiment results demonstrate the benefits of our techniques for siblings over the WI.

9. REFERENCES

[1] M. Altinel and M. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *Proc. of VLDB*, Cairo, Egypt, 53–64, 2000.

[2] J. Clark and S. DeRose. XML path language (XPath) version 1.0 w3c recommendation, Technical Report REC-xpath-19991116, World Wide Web Consortium, 1999.

[3] E. Cohen, H. Kaplan and T. Milo. Labeling dynamic XML trees, In *Proc. of PODS*, 271–281, 2002.

[4] B.F. Copper, N. Sample, M.J. Franklin, G.R. Hjaltason and M. Shadmon. A fast index for semistructured data, In *Proc. of VLDB*, Rome, Italy, 341–350, 2001.

[5] T. Grust. Accelerating XPath location steps, In *Proc. of SIGMOD*, 2002.

[6] A. Guttman. R-trees: a dynamic index structure for spatial searching, In *Proc. of SIGMOD*, 45–47, 1984.

[7] I. Kamel and C. Faloutsos, Hilbert r-tree: an improved r-tree using fractals, In *Proc. of VLDB*, Santiago, Chile, 500–509, 1994.

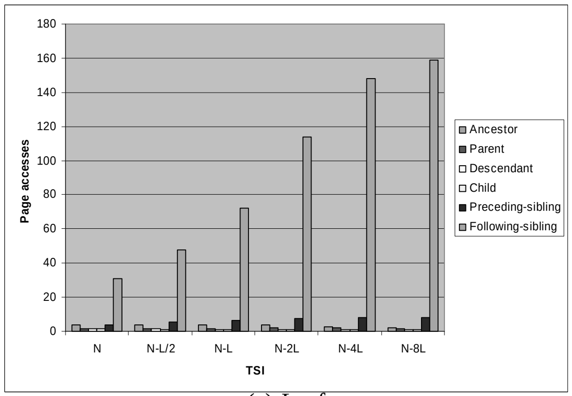
[8] W.E. Kimber. HyTime and SGML: understanding the HyTime HYQ query language, Technical Report Version 1.1 IBM Corporation, 1993.

[9] Q. Li and B. Moon. Indexing and querying XML data for regular path expressions, In *Proc. of VLDB*, Rome, Italy, 361–370, 2001.

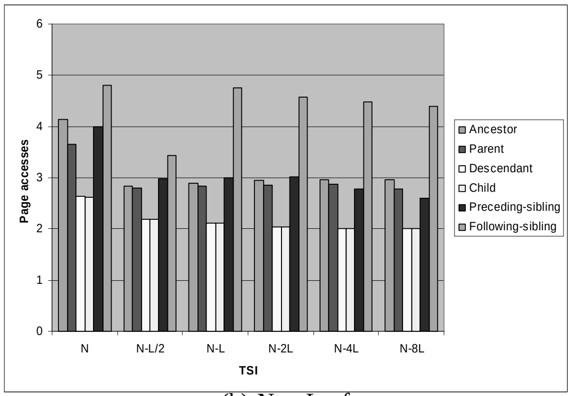
[10] T. Milo and D. Suciu. Index structure for path expressions, In *Proc. of ICDT*, Jerusalem, Israel, 271–295, 1999.

[11] K.V. Ravikanth, D. Agrawal, A.E. Abbadi, A.K. Singh and T. Smith. Indexing hierarchical data, Univ. of California, CS-Tr-9514, 1995.

[12] H. Wang and X. Meng. On the sequencing of tree structures for XML indexing, Preceding of the



(a) Leaf



(b) Non-Leaf

Figure 5: TSI

International Conference on Data Engineering, Tokyo, Japan, 2005.

[13] H. Wang, S. Park, W. Fan and P. Yu. ViST: a dynamic index method for querying XML data by tree structures, In *Proc. of SIGMOD*, San Diego, USA, 2003.

[14] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. Extensible Markup Language (XML) 1.0 second edition W3C recommendation. Technical Report REC-xml-20001006 WWW Consortium, October 2000.

[15] J. Zobel, A. Moffat and R. Sacks-Davis. An efficient indexing technique for full text database systems, In *Proc. of VLDB*, Vancouver, Canada, 352–362, 1992.

[16] K. Deschler and E. Rundensteiner. MASS: A multi-axis storage structure for large XML documents, In *Proc. of CIKM*, Louisiana, USA, 2003.

APPENDIX

The constraints with page boundary are:

$$B\_Ln'_{low} \le Ln' \le B\_Ln'_{high}, B\_Rn'_{low} \le Rn' \le B\_Rn'_{high}. \quad (1)$$

The following equations are obtained from equation 1.

$$B\_Rn'_{low} + B\_Ln'_{low} \le Rn' + Ln' \le B\_Rn'_{high} + B\_Ln'_{high},$$

$$B\_Rn'_{low} - B\_Ln'_{high} \le Rn' - Ln' \le B\_Rn'_{high} - B\_Ln'_{low}. \quad (2)$$

Condition for descendant:

- Combined with query:

$$Ln' > Q_{Ln'}, Rn' > Q_{Rn'} \equiv Ln' > Q_{Ln'}, Rn' - Ln' > Q_{Rn'} - Q_{Ln'}. \quad (3)$$

- Combined with bounding box (by Equation 1, 2, and 3):
  - $(B\_Rn'_{high} - B\_Ln'_{low} > Q_{Rn'} - Q_{Ln'})$ & $(B\_Ln'_{high} > Q_{Ln'})$.
  - $Ln' > Q_{Ln'}, Rn' - Q_{Rn'} + Q_{Ln'} > Ln' \Rightarrow Q_{Rn'} < Rn'$.
- The result condition is:

$$(B\_Rn'_{high} - B\_Ln'_{low} > Q_{Rn'} - Q_{Ln'}) \;\&\; (B\_Ln'_{high} > Q_{Ln'}) \;\&\; (B\_Rn'_{high} > Q_{Rn'}).$$

Condition for ancestor:

- Combined with query:

$$Ln' < Q_{Ln'}, Rn' < Q_{Rn'} \equiv Ln' > Q_{Ln'}, Rn' - Ln' < Q_{Rn'} - Q_{Ln'}. \quad (4)$$

- Combined with bounding box (by Equation 1, 2, and 4):
  - $(B\_Rn'_{low} - B\_Ln'_{high} < Q_{Rn'} - Q_{Ln'})$ & $(B\_Ln'_{low} < Q_{Ln'})$.
  - $Ln' < Q_{Ln'}, Rn' - Q_{Rn'} + Q_{Ln'} < Ln' \Rightarrow Q_{Rn'} > Rn'$.
- The result condition is:

$$(B\_Rn'_{low} - B\_Ln'_{high} < Q_{Rn'} - Q_{Ln'}) \;\&\; (B\_Ln'_{low} < Q_{Ln'}) \;\&\; (B\_Rn'_{low} < Q_{Rn'}).$$

Condition for preceding:

- Combined with query:

$$Ln' < Q_{Ln'}, Rn' > Q_{Rn'} \equiv Ln' < Q_{Ln'}, Rn' - Ln' > Q_{Rn'} - Q_{Ln'}. \quad (5)$$

- The result condition after combined with bounding box (by Equation 1, 2, and 4) is:

$$(B\_Rn'_{high} - B\_Ln'_{low} > Q_{Rn'} - Q_{Ln'}) \;\&\; (B\_Ln'_{low} < Q_{Ln'}).$$

Condition for following:

- Combined with query:

$$Ln' > Q_{Ln'}, Rn' < Q_{Rn'} \equiv Ln' > Q_{Ln'}, Rn' - Ln' < Q_{Rn'} - Q_{Ln'}. \quad (6)$$

- The result condition after combined with bounding box (by Equation 1, 2, and 6) is:

$$(B\_Rn'_{low} - B\_Ln'_{high} < Q_{Rn'} - Q_{Ln'}) \;\&\; (B\_Ln'_{high} > Q_{Ln'}).$$

XFrag: A Query Processing Framework for Fragmented XML Data

Sujoe Bose and Leonidas Fegaras
University of Texas at Arlington, CSE
416 Yates Street, P.O. Box 19015
Arlington, TX 76019-19015
{bose,fegaras}@cse.uta.edu

ABSTRACT

Data fragmentation offers various attractive alternatives to organizing and managing data, and presents interesting characteristics that may be exploited for efficient processing. XML, being inherently hierarchical and semi-structured, is an ideal candidate to reap the benefits offered by data fragmentation. However, fragmenting XML data and handling queries on fragmented XML are fraught with challenges: seamless XML fragmentation and processing models are required for deft handling of query execution on inter-connected and inter-related XML fragments, without the need of reconstructing the entire document in memory. Recent research has studied some of the challenges and has provided some insight on the data representation, and on the rather intuitive approaches for processing fragmented XML. In this paper, we provide a novel pipelined framework, called *XFrag*, for processing XQueries on XML fragments to achieve processing and memory efficiency. Moreover, we show that this model is suitable for low-bandwidth mobile environments by accounting for their intrinsic idiosyncrasies, without sacrificing accuracy and efficiency. We provide experimental results showing the memory savings achieved by our framework using the XMark benchmark.

1. INTRODUCTION

The widespread adoption of XML has rendered it as the language of choice for communication between collaborating systems. XML is also being studied as a data storage format and is being used by various systems for data management and query processing on native XML data [1, 2]. XQuery has become the language of choice for querying stored XML data, but recently we have seen systems, such as XSM [25], XRQL [4], and FluXQuery [3], that support XQuery processing on streamed XML data as well. XML data, being inherently hierarchical and semi-structured, poses an overwhelming overhead on critical runtime factors, such as memory requirements and processing efficiency. Given that most of queries on large XML documents are selective in nature, queries may benefit from fragmenting the XML document so that processing in parts would require less memory and processing power. There are several other reasons for fragmenting data. In a realtime sensor-based system, data is continuously generated from sensors and it disseminated in fragments as and when it occurs. Furthermore, streaming changes to data may pose less overhead by sending only fragments corresponding to the change, rather than sending the entire document with the change. Moreover, given the current shift from pull-based to push-based broadcast models, fragmentation of data provides several benefits: it is possible to prioritize data fragments so that high priority fragments of data may be scheduled ahead of the low priority ones. Also it is possible to associate quality of service parameters on the data fragments to meet delivery constraints. With the proliferation of mobile devices and with the quest for information on the move, servers disseminate data over low-bandwidth and error-prone environments. As the intermittent connectivity of mobile clients makes infeasible to deliver huge datasets, fragments may be a better choice for data delivery. Moreover it is easier to synchronize on smaller fragments because transmitting changes to data requires only sending the fragments that correspond to the change, without having to send the entire document.

The hallmark of our framework is the support for processing fragments rather than documents, especially in the presence of continuous updates to the document. This helps in optimizing the bandwidth and processing requirements by transmitting and processing only the update fragments without its entire unchanged context. Another area that benefits from a fragmented XML data model, is the inclusion of temporal extents on the XML dataset to capture the historical context of the transmitted data. In the new breed of event driven applications, as presented in XCQL [23], which require implicit temporal association, transmission in fragments provides seamless integration of temporal context within the data model and constructs for performing historical and window queries with temporal extents. In data distribution systems, fragmentation of data items is prevalent, because collaborating systems, geographically and logically dispersed, require different aspects of the data. Furthermore, system efficiency is facilitated by useful work in processing the required data and by ignoring the rest.

Unfortunately, processing fragments instead of whole XML documents is fraught with challenges. It requires not only knowledge of the locational context of fragments, which allows us to navigate from fragment to fragment during query processing, but also caching some of the fragments when necessary, since not all fragments may be available at the same time. Also, due to changes to fragments and to the intermittent connectivity of mobile clients, fragments may arrive in any order, and may be repeated or updated. In this paper, we address these challenges and provide a robust framework to process the fragments as and when they arrive without losing the overall context, resulting in lower memory footprints and faster response time.

Our previous work [18, 16] has concentrated on modeling and management of fragmented XML data and has proposed simple methods to handle the challenges present in such representation. One common way is to suspend fragments until their contained data arrives for continuing execution. This causes a serious challenge on the memory requirements, as fragments may arrive in any order. In addition, waiting for a fragment to come with complete information necessary for execution would result in blocking.

In this paper, we propose a novel pipelined execution framework for processing XQueries on streamed XML fragments. The fragments are processed as and when they arrive, and their inter-dependencies and hence their effect on the query results are resolved pro-actively. Our motivation to process the fragments as soon as possible is to conserve memory by discarding fragments that will not contribute to the result. Moreover, fragments that do not actively contribute to the result, but due to their relationship with other fragments affect the result, as in the case of fragments involved in query predicates, are kept in memory as long as necessary.

Unlike traditional applications of the pipelined processing model, query processing of fragmented data using the pipelined model of execution provides new challenges: since queries could span across fragments, we must factor the relative references between fragments while executing the query predicates and projections. Additionally, queries on XML data could operate on any level of the XML document, and hence the query predicates and projections traverse multiple fragments, which may arrive at arbitrary times in the fragmented XML stream. Also, the ability to construct new elements as part of the result XML, the presence of accumulation operators, and the out of order arrival of XML fragments, add additional challenges to the processing framework. Note that, we assume that the query clients, such as low-power hand-held devices, have limited memory and processing capacity that make it impossible to reconstruct the entire XML data before processing the queries.

The rest of the paper is organized as follows. Section 2 presents the related work in the area of XML stream query processing. Section 3 explains our framework by providing a model for XML fragments and for tag structures, which define the structural makeup of fragments. Section 4 describes in detail the pipeline model of processing fragments and the formal representation of the translation and processing framework used in XFrag. Finally, Section 6 presents experimental results from our implementation and shows the memory saving achieved in our framework.

2. RELATED WORK

Several recent efforts have focused on addressing frameworks for continuous processing of data streams [5, 8, 9, 21], however to the best of our knowledge, there is no work done in stream query processing of fragmented XML data. The Tribeca [13] data stream processing system provides language constructs to perform aggregation operations, as well as multiplexing and window constructs to perform stream synchronization, but it is restricted to relational data. Other efforts concentrating on windowed stream processing, such as StreamQuel [20], CQL [17], also address relational data only and provide SQL-like constructs to process streaming data. The COUGAR [11] system proposes the use of ADTs in object-relational database systems to model streams with associated functions to perform operations on the stream data. Several efforts have addressed the stream processing of XML data using XPath expressions [5, 7, 9]. A transducer-based XQuery processor for streaming XML data has been proposed in [25]. An alternative to transducer-based processing is a compositional XQuery processor based on SAX events, defined in [15]. An alternative fragmented XML processing model, suitable for pull-based web-service applications, is presented in Active XML [26]. In Xstream [24], the advantages of a semantics-based fragmentation of XML data for efficient transmission over a wireless medium are highlighted.

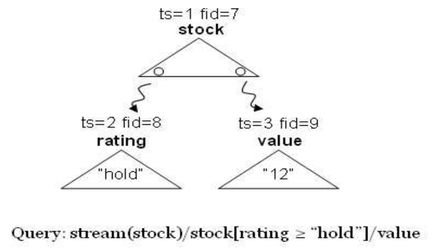


Figure 1: Sample Stock Fragments

3. OUR FRAGMENTED DATA MODEL

In our framework, the basic stream components transmitted by a server are fragments, each with its own ID. To be able to relate fragments with each other, we derive the concept of *holes* and *fillers* as detailed in our earlier work [18]. A hole represents a placeholder into which another rooted subtree (a fragment), called a filler, could be positioned to complete the tree. The filler can in turn have holes in it, which will be filled by other fillers, and so on. An example set of XML fragments is shown in figure 1.

Our framework makes use of the structural summary of XML data, called the *Tag Structure*, which defines the structure of data and provides information about fragmentation. This information is used when compiling XQuery expressions into plans that operate on the XML fragments and when deciding which fragments to keep in memory. Moreover, the Tag Structure is used in expanding wild-card path selections in queries to optimize query execution. The Tag Structure is itself structurally a valid XML fragment that conforms to the following simple recursive DTD:

```
<!DOCTYPE tagStructure [
<!ELEMENT tag (tag*)>
   <!ATTLIST tag type (filler | embedded) #REQUIRED>
   <!ATTLIST tag id CDATA   #REQUIRED>
   <!ATTLIST tag name CDATA #REQUIRED> ]>
```

A tag corresponds to an XML tagged element and is qualified by a unique id, a name (the element tagname), and

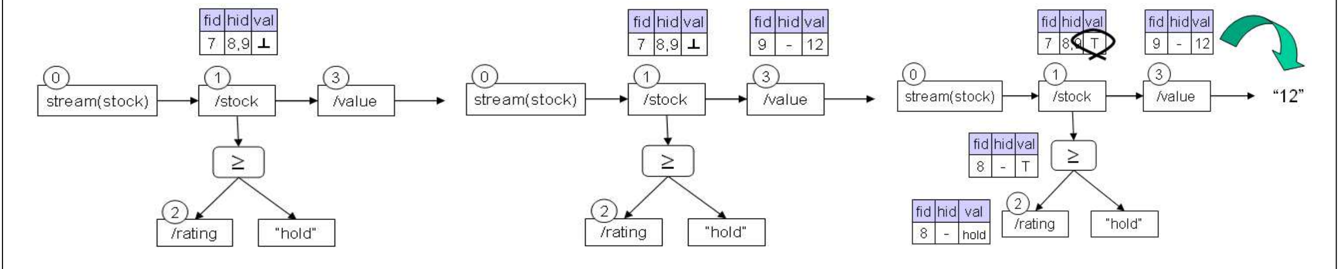


Figure 2: Processing Example

a type. A filler type implies that this element will arrive in a separate filler fragment, as opposed to the embedded type, which implies that this element is embedded within its parent element (inside the same fragment). Since the tag structure is a very important piece of information to the client for handling the input stream data, we require that it be streamed before the actual data.

4. THE XFRAG PIPELINE

Each XQuery primitive in an input query corresponds to an XFrag operator, which operates on an XML subtree at a particular level in the original XML document. For example, a path step in an XPath expression corresponds to a path operator that operates on the elements in the XML document having the same tag value and it corresponds to the same subtree level as that of the operator. Moreover, an XPath predicate expression maps to a condition operator, which may in turn reference path operators to perform predicate evaluation. As each fragment corresponds to a particular level in the original XML document, it is necessary to associate each operator with the fragments that will process. We use the Tag Structure to associate operators with the tag structure id (tsid) of the subtree that corresponds to the execution context of the operator. Each fragment is identified by the tsid of the subtree that belongs to in the original XML document, and hence each operator will process the fragment only if the tsids match. In the event that they do not match, the fragment is simply passed on to the subsequent operator in the query tree.

4.1 Fragment Relationships

As fragments in the original document may arrive in any order and query expressions may contain predicates at any level in the XML tree, it is necessary to keep track of the parent-child links between the various fragments, so that if a particular fragment does not pass the predicate evaluation at a particular level in the XML document, the corresponding descendant fragments must not be rendered as part of the output. We use the filler-id and hole-id information in the fragments to keep track of the fragment relationships. We maintain the fragment links in an association table at each operator to record the parent-child relationships seen in fragments processed by the operator. Moreover, each entry is tagged by a value of true, false, undecided (⊥), or a result fragment. While the former three values are possible in intermediate operators that do not produce a result, the latter is possible when the operator is the terminal operator in the query tree branch. Fragments corresponding to intermediate operators are discarded after recording the parent-child link relationships, thereby conserving memory. This link information corresponds to a small part of the actual data in the XML fragment, the rest of which is not relevant in producing the result.

4.2 Ancestor Inquiry

When a fragment is processed by an operator, it needs to verify if the predecessor operator has excluded its parent fragment due to either predicate failure or due to exclusion of its ancestor. For this reason, each operator maintains both a successor operator list and a pointer to the predecessor operator, using the former to hand-over fragments for processing by successor operators, and the latter to resolve fragment relationships and predicate criteria. When the predecessor inquiry is made to determine the eligibility of a particular fragment, one of four conditions may arise. (1) The parent fragment may not have arrived at the predecessor and hence there is no entry in the association table of the predecessor. In this case, the fragment is tagged with an undecided value. (2) The parent fragment had arrived at the predecessor and is tagged with an undecided value. In this case too the fragment with an undecided value is recorded. (3) The parent fragment had arrived at the predecessor and is tagged with a value of true, which implies that the parent fragment or its ancestor have passed all predicate expressions. In this case the fragment is tagged with a value of true, it is a potential candidate for output, depending on whether this operator is the result producer or is an intermediary in the query tree. (Note that the predicate evaluation follows the existential semantics of XQuery.) The last case is when the parent fragment had arrived and is tagged with a value of false. In this case, the fragment is also tagged with the value of false.

4.3 Descendant Trigger

As fragments may be waiting on operators to decide on their ancestor eligibility, they must be triggered when an ancestor condition is evaluated. Moreover, the predicate evaluation of a fragment may depend on child tags embedded in the fragment or child fragments that may arrive at a later point in time. In order to account for these dependencies, we introduce a recursive trigger invocation on the successor operators. Whenever a fragment is marked as true (or false) in a particular operator, other fragments that are waiting with an undecided value in successor operators may now be triggered to be rendered with a value of true (or false), and, subsequently, either produced (or not produced) as output. Similarly, condition operators will trigger their parent operators when a condition evaluates to true for at least one of

$$
\begin{aligned}
\mathcal{R}(\llbracket \text{stream}(url)\, path \rrbracket_\delta, \beta, ts) &\rightarrow \{(\beta', ts') \mid ts' \leftarrow \mathbf{ts}(url), \beta' \leftarrow \rho^{url}_{s,p,\omega,t}(t = ts'/@tsid, s = \mathcal{R}(\llbracket path \rrbracket_\delta, \beta', ts')\} \\
\mathcal{R}(\llbracket /A\, path \rrbracket_\delta, \beta, ts) &\rightarrow \{(\beta', ts') \mid ts' \leftarrow ts/tag[@name = \text{“}A\text{”}], \\
&\qquad \beta' \leftarrow \mu^{A}_{s,p,\omega,t}(t = (\mathsf{isfiller}(ts')?ts'/@tsid : \beta.tsid), p = \beta, \\
&\qquad\qquad s = \mathcal{R}(\llbracket path \rrbracket_\delta, \beta', ts'))\} \\
\mathcal{R}(\llbracket /@A \rrbracket_\delta, \beta, ts) &\rightarrow \{\ (\beta', ts) \mid \beta' \leftarrow \mu^{@A}_{s,p,\omega,t}(t = ts.tsid, p = \beta, s = \{\})\ \} \\
\mathcal{R}(\llbracket /*\ path \rrbracket_\delta, \beta, ts) &\rightarrow \{\ (\beta', ts') \mid ts' \leftarrow ts/tag, \beta' \leftarrow \mu^{ts'/@name}_{s,p,\omega,t}(t = (\mathsf{isfiller}(t')?t'.tsid : \beta.tsid), \\
&\qquad\qquad p = \beta, s = \mathcal{R}(\llbracket path \rrbracket_\delta, \beta', ts'))\} \\
\mathcal{R}(\llbracket //A\, path \rrbracket_\delta, \beta, ts) &\rightarrow \mathcal{R}(\llbracket /A\, path \rrbracket_\delta, \beta, ts) \cup \mathcal{R}(\llbracket /*\ //A\, path \rrbracket_\delta, \beta, ts) \\
\mathcal{R}(\llbracket /A[e]\, path \rrbracket_\delta, \beta, ts) &\rightarrow \{\ (\beta', ts') \mid ts' \leftarrow ts/tag[@name = \text{“}A\text{”}], \\
&\qquad \beta' \leftarrow \mu^{A,c}_{s,p,\omega,t}(t = (\mathsf{isfiller}(t')?t'.tsid : \beta.tsid), p = \beta, \\
&\qquad\qquad s = \mathcal{R}(\llbracket path \rrbracket_\delta, \beta', ts'), c = \mathcal{R}(\llbracket e \rrbracket_\delta, \beta', ts'))\} \\
\mathcal{R}(\llbracket e_1\ \mathsf{op}\ \ e_2 \rrbracket_\delta, \beta, ts) &\rightarrow \{\ (\beta', ts) \mid \beta' \leftarrow \sigma^{lhs,rhs,op}_{s,p,\omega,t}(t = ts/@tsid, p = \beta, s = \{\}, \\
&\qquad\qquad lhs = \mathcal{R}(\llbracket e_1 \rrbracket_\delta, \beta, ts), rhs = \mathcal{R}(\llbracket e_2 \rrbracket_\delta, \beta, ts))\ \} \\
\mathcal{R}(\llbracket \text{“}text\text{”} \rrbracket_\delta, \beta, ts) &\rightarrow \{\ (\beta', ts) \mid \beta' \leftarrow \mathcal{C}^{text}_{s,p,\omega,t}(t = ts, p = \beta, s = \{\})\ \} \\
\mathcal{R}(\llbracket \text{<}A\text{>}e\text{</}A\text{>} \rrbracket_\delta, \beta, ts) &\rightarrow \{\ (\beta', ts') \mid id \leftarrow \mathsf{genTsid}(), ts' \leftarrow\ \text{<tag name=“A” id=“}id\text{”>}ts\text{</tag>}, \\
&\qquad \beta' \leftarrow \theta^{A}_{s,p,\omega,t}(t = ts'/@tsid, p = \beta, s = \mathcal{R}(\llbracket path \rrbracket_\delta, \beta', ts'))\ \}
\end{aligned}
$$

Figure 3: XFrag Query Translation

the child tags or fragments.

4.4 XFrag Pipeline Processing Example

As an example, consider the stock stream, which produces fragments corresponding to values and ratings of stocks, with sample fragments as shown in Figure 1. The stock stream is described by the following tag structure:

```
<tag stream="stock">
   <tag id="1" name="stock" filler="true">
      <tag id="2" name="rating" filler="true"/>
      <tag id="4" name="symbol"/>
      <tag id="5" name="name"/>
      <tag id="3" name="value" filler="true"/>
   </tag> </tag>

Query 1: stream("stock.xml")
           /stock[rating >= "hold"]/value
```

The XFrag pipeline corresponding to the above query is depicted in Figure 2. When a “stock” fragment with tsid 1, filler id 7 and hole ids 8 and 9, arrives at the operator with tsid 1, the association table is updated with this information as shown in figure 2(a). Moreover, the fragment 7 is tagged with an undecided value, as the condition has not been evaluated yet for this fragment. Note that, at this point, the “stock” filler may be discarded as it is no more needed to produce the result and the hole filler association is already captured. This results in memory conservation on the fly, as we discard fragments, if they are no more needed to be retained. When the “value” fragment corresponding to the “stock” filler arrives, the operator with tsid 3 updates its association table with the value of its expression, but does not output the value, as the inquiry on the predecessor operator returns an undecided value. The “value” filler may also be discarded at that point conserving memory, as the result value, which is a subset of the fragment, is already captured in the association table. When the “rating” fragment corresponding to the “stock” filler arrives, the operator with tsid 2 updates its association table and returns the value of “hold”, as there is no condition for it to wait. The condition operator now determines that this value matches the criteria for filler id 8 and hence triggers the parent “stock” operator with the id 8 as true. The “stock” operator updates its association table for the parent filler 7 as true and triggers its successor “value” operator, which causes the value of “12” to be output.

5. XFRAG FORMAL SEMANTICS

5.1 Query Translation Function

The translation of XQuery expressions into the XFrag operator pipeline is depicted in Figure 3. The translation function $\mathcal{R}$, is a mapping from XQuery expression and the tag structure to an XFrag operator tree. Every operator is a specialization of the basic operator type β, which is characterized by a successor operator list s, a predecessor p, an association table ω, and the tag structure corresponding to the operator. The stream extraction operator ρ reads fragments from a stream, identified by url, and forwards them to the successors. Path expressions are mapped to the path projection operator μ. Wild-card and descendant path expressions are translated into a set of path projection operators by performing a wild-card projection and recursive descent on the tag structure. Predicate expressions are translated into condition operators and element construction into the construction operator θ. Since element construction adds a new tag element into the result set, the tag structure is extended with a tag equal to the element tag and a new tsid generated to identify the tag. A FLWR expression, which binds an expression to a variable, extends the environment δ, with a binding entry that relates the variable name to the XFrag operator sub-tree corresponding to the bound expression. The bindings added in the environment are referenced at the point where a variable is used in other expressions. Using the translation rules, the query used in the stock example is converted into the XFrag operator tree:

$$\rho^{stock.xml}_{s,p,\omega,t0}(\mu^{stock}_{s,p,\omega,t1}(\sigma(\geq,\mu^{rating}_{s,p,\omega,t2},\mathcal{C}^{hold}),\mu^{value}_{s,p,\omega,t3}))$$

5.2 Fragment Processing Function

The semantics of fragment handling by the various operators in XFrag is shown in Figure 4. For brevity, we have not included the semantics for all the operators, but have presented those that are used in our example. There are three basic functions defined for each operator in XFrag. The process function $\mathcal{P}$ takes a fragment and produces a set of output fragments. The inquiry function $\mathcal{I}$ takes a filler id and returns the value recorded in the association table of the operator and in any conditional expression, if present. The trigger function $\mathcal{T}$ takes a filler id and returns a set of fragments as output. The process function performs an inquiry on the association table, and, depending on the result of the inquiry and on whether it is an intermediate operator, triggers successors to output any fragments waiting to be resolved. The operators corresponding to the FLWR expressions, not presented, requires special mention. While the operator corresponding to the "FOR" expression produces result fragments as and when a fragment is available on the return clause, the "LET" expression, on the other hand, collects fragments from the return clause until all the siblings are present and then produces the result.

6. EXPERIMENTAL RESULTS

We have implemented the XFrag framework in Java and have modeled the operator types as individual classes. All operators are made to derive from the common fragment operator that provides the basic components of the XFrag pipeline operator and the supporting functions. We have ran tests using the XMark benchmark [22] on a Pentium III processor running Microsoft Windows 2000 with 512MB RAM. We have used the following 3 queries on the generated auction XML document and compared the results with the Qizx XQuery processor [10].

```
Query 1: doc("auction.xml")/site/open_auctions//
           increase
Query 2: doc("auction.xml")/site/open_auctions/
           open_auction[initial > "10"]/bidder
Query 3: doc("auction.xml")/site/open_auctions/
           open_auction/bidder[increase > "200"]
```

The memory profiling was done using the EclipseProfiler plugin for the Eclipse IDE and the results are summarized in Figure 5, using a generated auction XML document of size 23.3MB. For XFrag, the auction document was fragmented into fillers and holes, producing a file of size 27.3MB, and the resulting filler fragments were processed sequentially. Note that the running time for XFrag was about twice as much as for Qizx, however, our main focus was the memory consumption to suit processing using devices with less memory. While the Qizx XQuery processor took almost the same amount of memory to run all of the three queries, topping about 60MB of heap space usage, the XFrag framework took a maximum of 10MB of heap space. Moreover, for the first Query, it took a constant amount of memory of about 2MB, as there were no conditional expressions to be evaluated and hence fragments were output as they arrived without waiting on other related fragments. For Queries 2 and 3, the association tables were populated with the hole filler links,

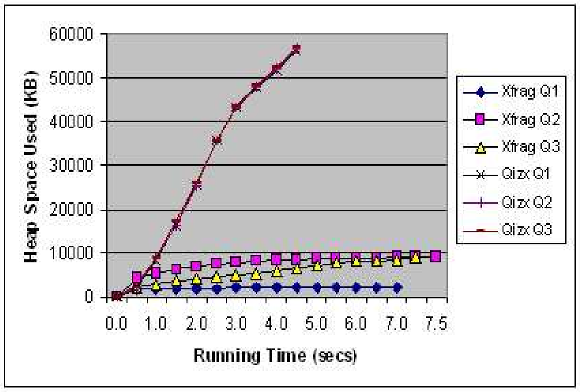


Figure 5: Comparison of Heap space usage for `XFrag` and `Qizx`

and the fragment values suspended until a matching condition signals the output to be flushed. While Query 2 had an initial increase compared to Query 3, which had a smoother increase, they both consumed the same amount of memory towards the end of the stream. Since Query 2 had to keep track of the filler-hole links from the "open_auction" fillers to the corresponding "bidder" fragments that occur later in the fragmented XML auction data, the memory consumption increased initially and then sustained when the "bidder" fragments arrived producing continuous output. However, Query 3 did not have to maintain these links as there is no condition expression present except in the "bidder" fragments only. In both Query 2 and 3, the overhead in memory consumption is due to the growth in the association table entries, however the huge advantage gained in the aggressive flushing of fragments eclipses the association table overhead.

7. CONCLUSION AND FUTURE PLANS

We have presented the XFrag framework to process fragments of XML data without having to wait for the entire XML document to be received and materialized. The fragments are processed as and when they occur and any interdependencies are pro-actively resolved, resulting in memory conservation. As future work we envision several optimization techniques that may be added to further improve on the memory usage. A possible candidate for this improvement is the association table, which may be aggressively purged to remove links that will not be needed during the course of the query processing on fragments. Moreover, to improve the running time of XFrag, instead of scheduling the 'process', 'inquire' and 'trigger' operation for each fragment, we could schedule these operations across a group of fragments, as not all fragments will result in triggering other fragments downline. However, there is a tradeoff between the scheduling frequency and the memory consumption as now more fragments may be held in memory before they can be triggered and flushed.

Acknowledgments: This work is supported in part by the National Science Foundation under the grant IIS-0307460.

8. REFERENCES

[1] C. Beeri and Y. Tzaban. SAL: An Algebra for Semistructured Data and XML. In *WebDB'99, Philadelphia, Pennsylvania*, pages 37–42, June 1999.

$$
\begin{aligned}
\mathcal{P}[\![\rho^{url}_{s,p,\omega,t}(\{\})]\!]_\delta &\rightarrow \{\, r \mid f \leftarrow \textbf{read}(url), r \leftarrow \mathcal{P}[\![s(f)]\!]_\delta \} && (R1)\\
\mathcal{P}[\![\mu^{path,c}_{s,p,\omega,t}(f)]\!]_\delta &\rightarrow \{\, r \mid f/@tsid \neq t, r \leftarrow \mathcal{P}[\![s(f)]\!]_\delta \} \bigcup \mathcal{P}[\![c(f)]\!]_\delta \bigcup \\
&\quad\; \{\, r \mid fid \leftarrow f/@tsid, fid = t, i \leftarrow \mathcal{I}[\![\mu^{path,c}_{s,p,\omega,t}(f/@id)]\!]_\delta, \\
&\qquad\quad \omega \leftarrow \omega \circ (fid, f/hole/@id, (i?\text{true} : (s = \phi ? f/path : \bot))), \\
&\qquad\quad r \leftarrow (s = \phi ? (i ? (f/path, fid) : \{\}) : (\mathcal{P}[\![s(f)]\!]_\delta \cup (i ? \mathcal{T}[\![s(fid)]\!]_\delta : \{\})))\} && (R2)\\
\mathcal{P}[\![\sigma^{lhs,rhs,op}_{s,p,\omega,t}(f)]\!]_\delta &\rightarrow \{\, r \mid f_l \leftarrow \mathcal{P}[\![lhs(f)]\!]_\delta, f_r \leftarrow \mathcal{P}[\![rhs(f)]\!]_\delta, op(f_l, f_r), \\
&\qquad\quad fid_p \leftarrow \mathcal{M}(f_l, f_r), r \leftarrow \mathcal{T}[\![p(fid_p)]\!]_\delta \} && (R3)\\
\mathcal{P}[\![\{\beta_1, \beta_2, \ldots, \beta_n\}(\{f_1, f_2, \ldots, f_m\})]\!]_\delta &\rightarrow \bigcup_{\substack{1 \le j \le m\\ 1 \le i \le n}} \mathcal{P}[\![\beta_i(f_j)]\!]_\delta && (R4)
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{I}[\![\beta_{s,p,\omega,t}(fid)]\!]_\delta &\rightarrow \vee/\{\, e.v \mid e \leftarrow p.\omega, hid \leftarrow e.hids, hid = fid \} && (R5)\\
\mathcal{I}[\![\mu^{path,c}_{s,p,\omega,t}(fid)]\!]_\delta &\rightarrow \mathcal{I}[\![\beta_{s,p,\omega,t}(fid)]\!]_\delta \wedge (c = \phi \,?\, \text{true} : \mathcal{I}[\![c(fid)]\!]_\delta) && (R6)\\
\mathcal{I}[\![\{\beta_1, \beta_2, \ldots, \beta_n\}(fid)]\!]_\delta &\rightarrow \bigvee_{1 \le i \le n} \mathcal{I}[\![\beta_i(fid)]\!]_\delta && (R7)
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{T}[\![\beta_{s,p,\omega,t}(fid)]\!]_\delta &\rightarrow \{\, (e.fid, e.v) \mid \mathcal{I}[\![\beta(fid)]\!]_\delta, s = \phi, e \leftarrow \omega, fid = e.fid \} \bigcup \\
&\quad\; \{\, \mathcal{T}[\![s(hid)]\!]_\delta \mid \mathcal{I}[\![\beta(fid)]\!]_\delta, s \neq \phi, e \leftarrow \omega, fid = e.fid, hid \leftarrow e.hids \} && (R8)\\
\mathcal{T}[\![\{\beta_1, \beta_2, \ldots, \beta_n\}(fid)]\!]_\delta &\rightarrow \bigcup_{1 \le i \le n} \mathcal{T}[\![\beta_i(fid)]\!]_\delta && (R9)
\end{aligned}
$$

Figure 4: The XFrag Processing Model

[2] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *ACM SIGMOD 2000*, pages 379–390, 2000.

[3] C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier. FluXQuery: An Optimizing XQuery Processor for Streaming XML Data. In *VLDB 2004*, pages 1309–1312.

[4] D. Florescu, et al. The BEA/XQRL Streaming XQuery Processor. In *VLDB 2003*.

[5] A. Gupta and D. Suciu. Stream Processing of XPath Queries with Predicates. In *SIGMOD 2003*, pp 419–430.

[6] Z. Ives, A. Levy, and D. Weld. Efficient Evaluation of Regular Path Expressions on Streaming XML Data. Technical Report, University of Washington, 2000, UW-CSE-2000-05-02.

[7] C. Barton, P. Charles, D. Goyal, M. Raghavachari, M. Fontoura, and V. Josifovski, “Streaming xpath processing with forward and backward axes.” in *ICDE*, 2003, pp. 455–466.

[8] D. Olteanu, T. Furche, and F. Bry. An Efficient Single-Pass Query Evaluator for XML Data Streams. In *SAC 2004*, March 2004, Nicosia, Cyprus.

[9] F. Peng and S. Chawathe. XPath Queries on Streaming Data. In *SIGMOD 2003*, pp 431-442.

[10] Qizx/Open. At `http://www.xfra.net/qizxopen`.

[11] P. Bonnet, J. Gehrke, and P. Seshadri. Towards sensor database systems. In *Proceedings of the Second International Conference on Mobile Data Management 2001*, pages 3–14.

[12] L. Golab and M. T. zsu. Issues in data stream management. In *SIGMOD Rec.*, 32(2):5–14, 2003.

[13] M. Sullivan and A. Heybey. Tribeca: A system for managing large databases of network traffic. In *the USENIX Annual Technical Conference 1998* pages 13–24.

[14] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and Issues in Data Stream Systems. In *PODS 2002*, pages 1–16.

[15] L. Fegaras. The Joy of SAX. In *XIME-P 2004*, pages 51–65. June 2004.

[16] L. Fegaras, D. Levine, S. Bose, and V. Chaluvadi. Query Processing of Streamed XML Data. In *CIKM 2002*, pages 126–133. November 2002.

[17] J. Widom. CQL: A Language for Continuous Queries over Streams and Relations. In *the 9th International Workshop on Data Base Programming Languages (DBPL)*, Potsdam, Germany, September 2003.

[18] S. Bose, L. Fegaras, D. Levine, and V. Chaluvadi. A Query Algebra for Fragmented XML Stream Data. In *the 9th International Workshop on Data Base Programming Languages (DBPL)*, Potsdam, Germany, September 2003.

[19] R. Motwani, et al. Query Processing, Approximation, and Resource Management in a Data Stream Management System. In *the 1st Biennial Conference on Innovative Data Systems Research (CIDR)*, January 2003.

[20] S. Chandrasekaran, et al. TelegraphCQ: Continuous Data flow Processing for an Uncertain World. In *Proceedings of Conference on Innovative Data Systems*, pages 269–280, 2003.

[21] D. Carney, et al. Monitoring streams—A New Class of Data Management Applications. In *VLDB 2002*, pages 215–226.

[22] A. Schmidt, F. Vaas, M. L. Kersten, M. J. Carey, I. Manolescu and R. Busse. XMark: A Benchmark for XML Data Management. In *VLDB 2002*, pages 974–985, 2002.

[23] S. Bose, L. Fegaras. Data Stream Management for Historical XML Data. In *SIGMOD 2004*, pages 239–250, June 2004.

[24] E. Wang, et al. Efficient Management of XML Contents over Wireless Environment by Xstream. In *ACM-SAC 2004*, pages 1122–1127, March 2004.

[25] B. Ludäscher, P. Mukhopadhyay, and Y. Papakonstantinou, “A transducer-based xml query processor.” in *VLDB*, 2002, pp. 227–238.

[26] S. Abiteboul, O. Benjelloun, B. Cautis, I. Manolescu, T. Milo, and N. Preda. Lazy Evaluation for Active XML. In *SIGMOD 2004*, pages 227–238, June 2004.

Analysis of User Web Traffic with A Focus on Search Activities

Feng Qiu
University of California
Los Angeles, CA 90095
fqiu@cs.ucla.edu

Zhenyu Liu
University of California
Los Angeles, CA 90095
vicliu@cs.ucla.edu

Junghoo Cho
University of California
Los Angeles, CA 90095
cho@cs.ucla.edu

ABSTRACT

Although search engines are playing an increasingly important role in users' Web access, our understanding is still limited regarding the magnitude of search-engine influence. For example, how many times do people start browsing the Web from a search engine? How much percentage of Web traffic is incurred as a result of search? To what extent does a search engine like Google extend the scope of Websites that users can reach? To study these issues, in this paper we analyze a real Web access trace collected over a period of two and half months from the UCLA Computer Science Department. Our study indicates that search engines influence about 13.6% of the users' Web traffic directly and indirectly. In addition, our study provides realistic estimates for certain key parameters used for Web modelling.

1. INTRODUCTION

Since its arrival in the early 90's, the World-Wide Web has become an integral part of our daily life. According to recent studies, people access the Web for a variety of reasons and spend increasingly more time surfing the Web. For example, [1] shows that a typical Internet user spends more than 3 hours per week online and tends to spend progressively less time in front of the TV partly due to increased "surfing" time.

This research is motivated by our desire to understand how people access the information on the Web. Even though the Web has become one of the primary sources of information, our understanding is still limited regarding how the Web is currently used and how much it influences people. In particular, we are interested in the impact of search engines on people's browsing pattern of the Web. According to recent studies [2], search engines play an increasingly important role in users' Web access, and if users heavily rely on search engines in discovering and accessing Web pages, search engines may introduce significant bias to the users' perception of the Web [3].

The main goal of this paper is to quantitatively measure the potential influence of search engines and the general access pattern of users by analyzing a real Web access trace generated from the users' daily usage. For this purpose, we have collected all HTTP packets originating from the UCLA Computer Science Department from May 15th 2004 until July 31st 2004 and analyze it to answer the following questions:

- *Search-engine impact*: How much of a user's access to the Web is "influenced" by search engines? For example, how many times do people start browsing the Web by going to a search engine and issuing a query? How many times do people start from a "random" Web site? How much do search engines expand the "scope" of Websites that users visit?
- *General user behavior*: How many different sites do people visit when they surf the Web? How much time do people spend on a single page on average? How many links do people follow before they jump to a "random" page?

The answers to the above questions will provide valuable insights on how the Web is accessed by the users. Our study will also provide realistic estimates for some of the key parameters used for Web modeling. For example, the number of clicks before a random jump is one of the core parameters used for the *random-surfer model* and PageRank computation [4].

The rest of the paper is organized as follows. In Section 2 we describe the dataset used for our analysis. In Section 3 we report our findings on the influence of search engines on the users' Web access. In Section 4 we report our other findings on the general user behavior on the Web. Related work is reviewed in Section 5 and Section 6 concludes the paper.

2. DESCRIPTION OF DATASET

In this section we first describe how we collect our HTTP access trace and discuss the necessary cleaning procedures we apply to it to eliminate "noise."

2.1 HTTP access trace

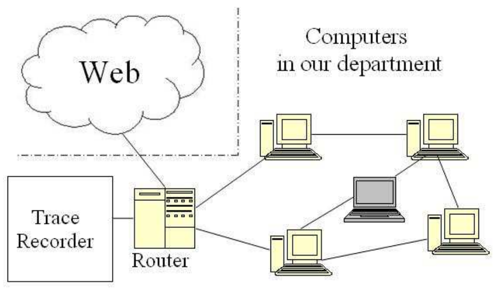


Figure 1: Network topology of UCLA CS Department

We have captured all HTTP Requests and Responses coming to/leaving from the UCLA Computer Science Department for the period of two and a half months. As we show in Figure 1, the CS department has roughly 750 machines connected through a 100Mbps LAN, which is then connected to the Internet through the department router. Since all packets that go to/come from outside machines pass this router, we can easily capture all HTTP packets

Copyright is held by the author/owner.
Eighth International Workshop on the Web and Databases (WebDB 2005), June 16-17, 2005, Baltimore, Maryland.
.

by installing a packet recorder at the router. Given the large volume of traffic, we recorded only the relevant HTTP headers (e.g., Request-URL, Referer, User-Agent, etc.) in the packets, discarding the actual content.

Statistics	Value
Collection period	May 15th – July 31st, 2004
# of local IPs	749
# of remote IPs	66,372
# of requests	2,157,887
size of our trace (in bytes)	50GB

Table 1: Statistics on our dataset

To help the reader assess the scale of our HTTP trace, we report a few statistics of our dataset in Table 1. In brief, our dataset contains 2,157,887 HTTP Requests generated by 749 machines inside of our department while they access 66,372 outside servers over a period of two and a half months.[1]

2.2 Data cleaning

The goal of this paper is to understand the user behavior on the Web. Unfortunately, a significant fraction of our HTTP trace was due to various activities that are not directly relevant to user behavior (e.g., download traffic generated by Web crawlers). In this section, we describe three main filtering criteria that we use in order to remove the "non-user" traffic from our dataset.

- *Crawler traffic*: There are a few Web crawlers running in our department for a number of research projects. The traffic from these crawlers are clearly irrelevant to user behavior, but it constituted more than half of our collected data. We filter out this crawler traffic by discarding all packets coming from/going to a few machines where the crawlers run.
- *Non-textual traffic*: Users typically consider everything within a Web page (both text and images) as a *single* Web page; they do not consider images on a page as a completely separate unit from the surrounding HTML page. However, the browser issues multiple HTTP Requests to fetch embedded images, so if we simply count the number of HTTP Requests issued by browsers, there is a mismatch between what users see (*one* Web page) and what we count (say, *five* HTTP Requests). This mismatch is particularly problematic when a Web page contains many small icons or advertising banners.

  To avoid this mismatch, we decide to limit our analysis only to text documents (e.g., HTML, PDF, PS), because most non-textual objects are embedded in an HTML page and are perceived as a part of the page. That is, we keep only the HTTP Requests that retrieve textual documents. This filtering is done by checking the *Content-Type* field of the response for each request and keeping only those whose *Content-Type* value is "text/html," "text/pdf," etc.
- *Non-browser traffic*: A number of computer programs generate HTTP Requests that do not directly reflect the users' browsing behavior. For example, a *BitTorrent* client — a distributed content dissemination system [5] — generates frequent HTTP Requests to its neighbors to check their availability and to download files. Again, since our focus is on users' Web browsing behaviors, we eliminate the traffic from these clients by checking the *User-Agent* field of the requests and retaining only those requests from well-known browsers, such as "Mozilla."

[1]The reported numbers are after we apply filtering steps described in the next section.

Other than described above, we also eliminate certain obvious noises, like requests to URLs in wrong formats. Figure 2 shows the fraction of our original trace that is filtered out by each criterion described above. The crawler filtering is most significant; more than 60% of the traffic is discarded by this criterion. After the three filtering, we are left with 5.3% of the original trace, which is 2,157,887 HTTP Requests.

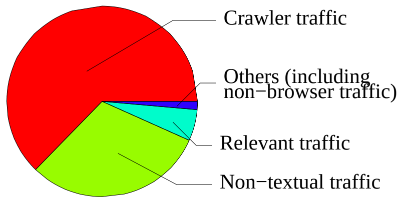


Figure 2: Fraction of discarded HTTP Requests

3. SEARCH ENGINE INFLUENCE

Based on the dataset described in the previous section, we now investigate how much search engines influence Web users. Search-engine influence can be seen from two different perspectives.

- *Help users visit more sites*: URLs of Web sites and/or pages are often hard to remember. *Bookmarks* or *Favorites* are used to maintain a user's favorite URLs, but they quickly become unmanageable as the list grows larger. Given this difficulty, users often use a search engine as an "extended bookmark"; they access pages by typing *keywords* (which are easier to remember) to a search engine instead of typing URLs. In this regard, search engines "expand" the set of pages that users can visit compared to the set of pages users have to remember or bookmark.

  How much do search engines expand the set of pages that a user visits? Is there overlap between the pages that users remember and visit directly and the ones that they visit through search engines?
- *Directing user traffic to particular sites*: Among billions of pages available on the Web, search engines direct users to a particular set of pages by picking and presenting a handful of pages in their search results given a query. Therefore, search engines "drive" a certain fraction of user traffic to the set of their selected sites. What fraction of user traffic is driven by search engines? How often do users randomly browse the Web and how often do they rely on search engines?

In order to answers the above questions, we first formalize "search-engine influence" by introducing the notion of a *referer tree*[2] in Section 3.1. We then present the statistics collected from our dataset in Section 3.2.

3.1 Influence, referer tree, and user

We assume that a user's visit to page p_2 is "influenced" by page p_1 if the user arrives at p_2 by following a link (or pressing a button) in p_1. This "link-click" information can be easily obtained from the *Referer* field in the HTTP Request headers. We illustrate the meaning of this field using a simple example.

Example 1 A user wants to visit the American Airlines homepage, but he does not remember its exact URL. To visit the page, the user first goes to the Google homepage (Figure 3(a)) by typing its URL

[2]In this paper, we use the misspelled word "referer" instead of the correct spelling "referrer" because of its usage in the standard HTTP protocol [6, 7].

www.google.com in the address bar of a browser. He then issues the query "American Airlines," for which Google returns the page in Figure 3(b). The user clicks on the first link and arrives at the American Airlines homepage (Figure 3(c)). From this homepage he further reaches other pages.

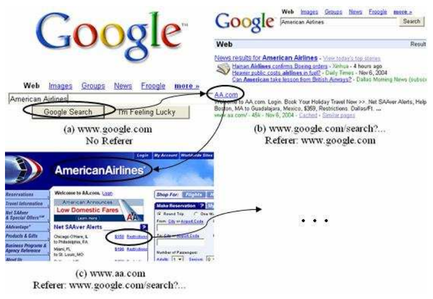


Figure 3: An example to illustrate the meaning of the *Referer* field

In this scenario, note that the user arrives at the first Google page (Figure 3(a)) directly without following a link. In this case, the *Referer* field of the corresponding HTTP Request is left empty, indicating that the user either directly typed the URL or used a bookmark entry. In contrast, the user arrives at the second and third pages (Figures 3(b) and (c)) by clicking a link or pressing a button. In these cases the *Referer* fields contain the URL of the immediately proceeding pages. For example, the *Referer* field of the second page request contains the URL of the first page, www.google.com. □

In summary, by looking at the existence and the value of the *Referer* field, we can tell whether and what links the user followed to arrive the page.

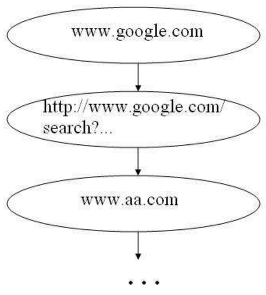


Figure 4: The referer tree for Example 1

Referer tree Using the *Referer* field information, we can construct a *referer tree*, where the nodes are the pages visited by a user and the edge from node p_1 to node p_2 means that the user followed a link from p_1 to p_2. In Figure 4 we show an example referer tree corresponding to the scenario described in Example 1. Note that the root of a referer tree represents a page visited directly by the user without following a link.

Given a referer tree, search-engine influence may be measured in one of the following ways:

- *Direct children only*: We consider that a search engine influences only the visits to the direct children of a search node (e.g., visit to www.aa.com node in Figure 4). This interpretation is reasonable in the sense that the search engine cannot control the links that its direct children present to the user.
- *All descendants*: We consider that all descendants of a search node are under search-engine influence. This interpretation is also reasonable because if the search engine did not provide the link to its direct children, the user wouldn't have arrived at any of their descendants.

In the next section, we estimate search engine influence under both interpretations.

Users In order to analyze users' Web browsing behaviors, we need to associate every HTTP Request with an individual user. In general, automatic user identification of an HTTP Request is a very complex task [8]. Fortunately, the usage pattern of our department machines allows us to use a simple heuristic for this task with reasonably high accuracy: we assume that *each IP corresponds to one user*, because all faculty members and most students have their own workstations that they primarily use for accessing the Internet.

The only concern is that some IP addresses might correspond to server machines, not workstations. On one hand, some of the servers are time shared; multiple users may simultaneously access the Web from a server, so the requests from one server represent the *aggregate behaviors* of multiple users, not the behavior of a single user. On the other hand, many servers are primarily used for computational tasks and practically no user uses them to access the Web. Therefore, if we count the requests from these servers in computing user statistics, the results may be biased.

To avoid these problems, we rely on the fact that more than 90% of user workstations run Windows or Mac operating systems, and consider only the requests from those machines when we try to measure the behavior of individual users.

3.2 Results of search-engine influence

We now report our results on search-engine influence. We notice that more than 95% of the search activities from our department goes to three major search engines: Google, Yahoo! Search and MSN Search. For this reason, we primarily focus on the influence of these three search engines in the rest of this section.

Search-engine-directed traffic In Figure 5, we show the fraction of traffic to search engine home pages (e.g, the first level nodes in Figure 4), to search engine result pages (e.g., the second level nodes in Figure 4), to their direct children (e.g., the third level nodes in Figure 4) and their descendants. Roughly, 1.0% of the user traffic goes to search engine home pages, and 5.7% are search requests. 2.1% of the user traffic goes to the direct children of search requests, with additional 4.8% to their descendants. (For this set of reported statistics, we have excluded the set of search-engine-home-page loading requests that do not lead to any further traffic, since such requests are most likely results of setting search engines as the default loading page of a Web browser.) Overall, 13.6% of user traffic is under the direct and indirect influence of search engines. Interestingly, these results imply that many of our users issue queries to search engines but do not click on links in the result pages.

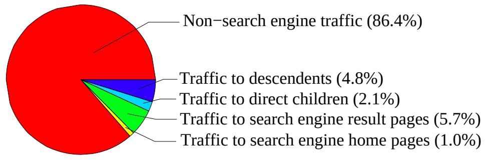


Figure 5: Search-engine traffic size

We can also assess the search-engine influence by measuring how many times people start surfing the Web from search engines. Given that the root node of a referer tree is where a user starts his surfing, we can measure this number by counting the number of search-engine-rooted referer trees. Our dataset contains a total of

380,453 referer trees, out of which 25,758 are rooted at search engines. Thus, we estimate that in about 6.8% of the time our users start surfing the Web from search engines.

Helping users visit more sites We now discuss how much search engines expand the set of sites that a particular user visits. This "site expansion" by search engines can be viewed in two ways: (1) search engine increase the *number of "starting points"* from which users can browse further (by providing new links in its search results.) (2) search engine increase the *total number of sites* that the user eventually visits. We may estimate these two effects of search engines as follows:

- *Seed-set expansion*: We refer to the set of Web sites from which a user starts his Web surfing as the *seed set* of the user. Given this definition, the *regular seed set* of a user corresponds to the root nodes of her referer trees (except when the root node is a search engine). The *search-engine seed set* corresponds to the direct children of search engine nodes.That is, the set of sites that search engines refer to, from which the user starts browsing. We can measure the seed set expansion by search engines simply by comparing these two seed sets.
- *Visit-set expansion*: We refer to the set of sites that a user eventually visits as the *visit set* of the user. The *search-engine visit set* is the set of all descendants of search-engine nodes. The *regular visit set* is all the nodes in the referer trees except the search-engine descendants. Again, by comparing these two sets, we can measure the visit set expansion by search engines.

In Figure 6, we first plot the seed set expansion effect by search engines. In the figure, the horizontal axis corresponds to time and the vertical axis shows the sizes of the regular seed set, the search-engine seed set and the overlap between them after the given time interval. For example, after six weeks, an average user has 188.2 sites in his regular seed set and 56.7 sites in the search-engine seed set, with an overlap of 15.6 sites. We observe that the relative ratio of this overlap roughly remains constant over the period of 10 weeks, which is about 8% of the regular seed set. We also observe that search engines consistently expands the size of the seed set by 22% over this period of time.

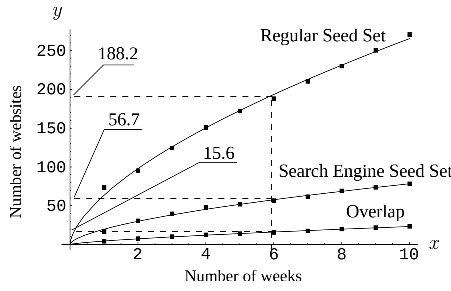


Figure 6: Search-Engine Seed Set Expansion

In Figure 7, we show a similar graph for the visit-set expansion. The meaning of the two axes of this graph is similar to the previous one. After six weeks, a user visits a total of 246.0 sites without using search engines and 72.4 sites starting from search engines, with an overlap of 23.0 sites. Similarly to the previous seed-set results, the relative size of the overlap roughly remains constant at a level of 9% of the regular visit set. Overall, search engines help an average user visit 20% more sites and the sites that users visit through search engines seem quite distinct from the sites that users visit from random surfing.

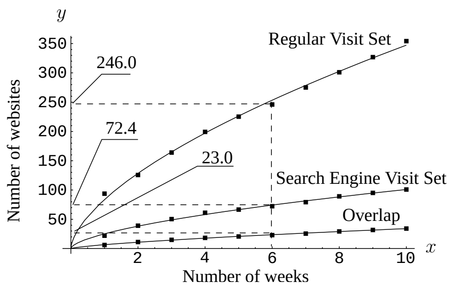


Figure 7: Search-Engine Visit Set Expansion

4. USER ACCESS STATISTICS

In this section we try to measure users' general behaviors in surfing the Web. In particular, in Section 4.1 we investigate how an average user follows hyperlinks during Web browsing. In Section 4.2 we investigate how much time people spend per page and how long they stay online in "one sitting."

4.1 Referrer tree statistics

Hyperlinks are considered a core structural component on the Web. It is generally believed that hyperlinks play a significant role in guiding people to particular Web sites. Since users' actions of following links are fully captured by referer trees, we now analyze the characteristics of the referer trees in our trace to understand the our users' clicking behavior.

In particular, we are interested in the *size*, *depth* and *branching factor* of the referer trees. The size of a referer tree measures how many pages a user visits by following links before she jumps to a new page. The depth shows how deeply a user follows hyperlinks before she stops exploring further. The branching factor indicates how many links on a page a user typically clicks on.

In Figure 8, we show the distributions of these three properties. In the graphs, the horizontal axis corresponds to the size, depth and branching factor of refer trees, respectively. The vertical axis shows the number of referer trees with the given characteristics.[3] All graphs in this section are plotted in a log-log scale.

From the graphs, we first see that all distributions closely fit power-law curves; the graphs are straight lines in the log-log scale. Also from Figure 8(a), we observe that 173,762 out of 380,453 referer trees have a single node. That is, 45% of the time, users jump to a completely new page after visiting just one page. Finally, given the mean of each distribution[4] we estimate that a typical Web user visits 5 pages by following hyperlinks, clicking on 3 links per page, but going down no more than 3 links deep.

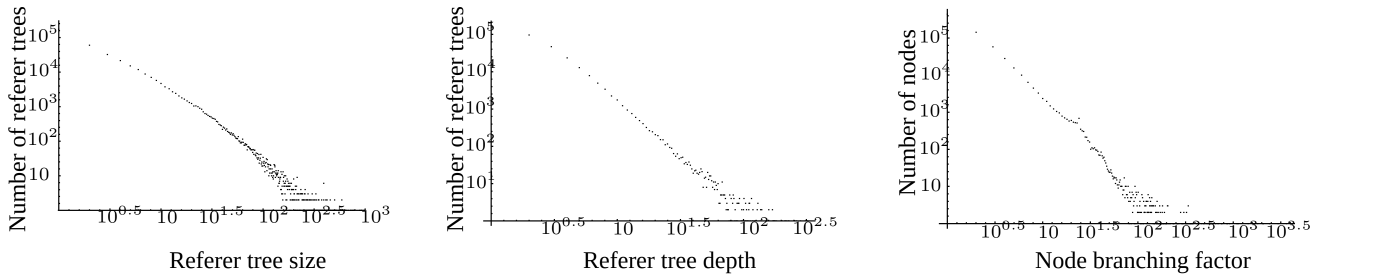
4.2 Session statistics

We now report statistics on the following characteristics of user behavior: (1) How many pages and sites do people visit once they start surfing the Web? (2) How much time do they stay online in one sitting? (3) How many times do they jump to new pages while they surf the Web? In order to answer these questions, we first introduce the notion of a *session*.

Definition of session Informally, a session refers to the set of pages that a user accesses in one "sitting." A traditional definition for the session is based on time-out. That is, after a user starts visiting Web pages, if there is a certain period of inactivity, say 10 minutes, then the current session expires and a new session starts.

[3]More precisely, because branching factors are characteristics of individual *nodes* not of *trees*, Figure 8(c) shows the number of nodes with the given branching factor.

[4]In computing the average branching factor, we exclude the leaf nodes in the tree for which users did not click any links.



(a) Size distribution (avg:5.67) (b) Depth distribution (avg: 2.52) (c) Branching factor distribution (avg: 2.95)

Figure 8: Referer tree characteristics

The main weakness of this definition is the difficulty in choosing a good time-out value. On one hand, if the time-out is set too short, the pages that a user browses in one sitting may be broken into multiple sessions, especially if the user reads a long online article. On the other hand, if the time-out is set too long, the pages that the user accesses in multiple sittings may be combined into one session. To remedy this shortcoming, we decide to extend the traditional definition using the referer-tree information.

The basic idea for our extended definition is that even if a user accesses a page after a certain period of inactivity, if the user clicks on a link on a previously accessed page to access a new page, it strongly hints that the user was actually reading the previous page. Based on this intuition, we put the accesses to page p_1 and p_2 into one session

- if they are accessed within a short time interval τ or
- if p_2 is accessed by following a link in p_1.

For example, consider Figure 9 that shows a sequence of pages accessed by a user. The relative spacing between the pages represent the time interval elapsed between the accesses. The curved arrows at the top represent that the user followed a link in the first page to the second. In this example, (p_1, p_2, p_3), (p_4, p_5), and (p_6, p_7, p_8) are put into the same sessions because they are accessed within time τ. In addition, p_3 and p_4 are put into the same session because p_4 is accessed by following a link in p_3. Overall, pages p_1 through p_5 are put into one session and pages p_6 through p_8 are put into another session.

We believe that it is safe to use a small threshold τ value under our extended definition, because as long as the users follow a link to reach from one page to another, these two pages are put into one session, even if the access interval is longer than τ. For this reason, we use a relative small value for τ, 5 minutes, for our analysis.

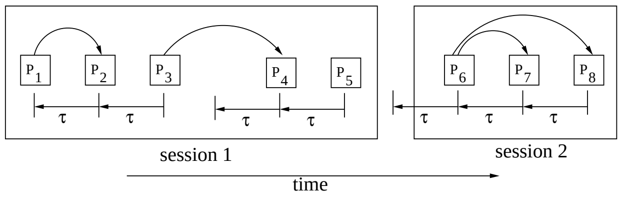


Figure 9: An example of our method of identifying sessions

Number of Web sites and pages per session Based on the session definition given above, we first report how many Web pages and sites a user visits in one session. In Figures 10 and 11 we present the distributions for Web pages and Web sites per session, respectively. The horizontal axis corresponds to the number of pages (or sites) per session, and the vertical axis shows the number of sessions that have the given number of pages (or sites). The average numbers are 21.79 for Web pages and 5.08 for Web sites, which means that a typical user visits about 22 pages in 5 Web sites in one sitting. The graph for Web pages closely fits a power-law curve, while the graph for Web sites does not exhibit a close fit.

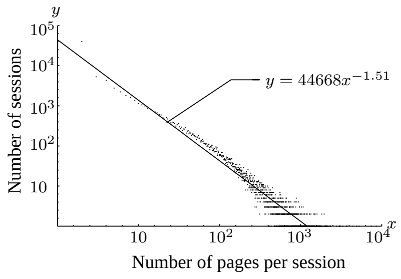


Figure 10: Number of pages per session (Avg:21.79)

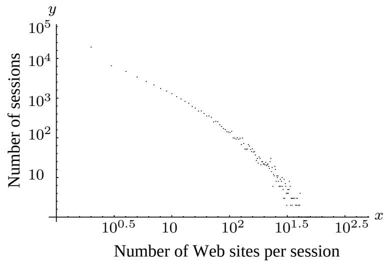


Figure 11: Number of Web sites per session (Avg:5.08)

Session length and average time per page Another interesting statistics is how much time a user spends online once she starts surfing the Web and how much time she spends reading each page. One issue in measuring these numbers is how to account for the time spent on the last page of a session. Because there is no subsequent page access, we do not know when the user stops reading the page. As a rough approximation, we assume that the time spent on the last page is equal to the average time spent on a Web page. Based this assumption, we present the session length distribution in Figure 12 and the average time per page distribution in Figure 13.

The graphs have a large number of outliers, but the general trends fit well to power-law curves. On average, a typical Web user spends about 2 hours per session and 5 minutes per page.

Number of referer trees per session Finally, we report within a session, how many times a user stops following links and jumps to a random page (by typing in a new URL or selecting a bookmark). Note that whenever the user jumps to a new page, a new referer tree is initiated. Thus we can learn how many times a user jumps to a random page in a session by counting how many referer trees the session contains. We present the number-of-referer-trees-per-session distribution in Figure 14. Again, the curve fits well to a

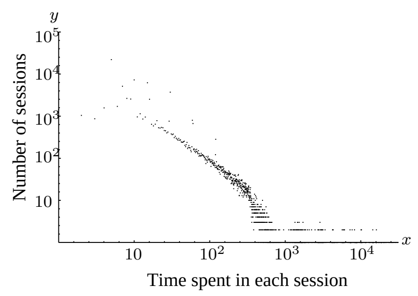


Figure 12: Session length in time (units in minutes)

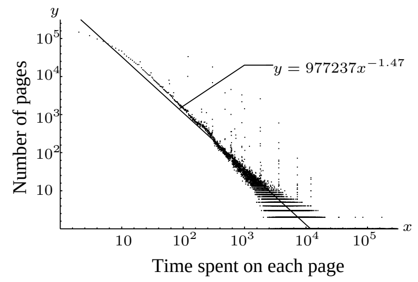


Figure 13: Time spent on each page (units in seconds)

power-law curve. The mean of the distribution is 3.83, meaning that people make about 3 random jumps per session on average.

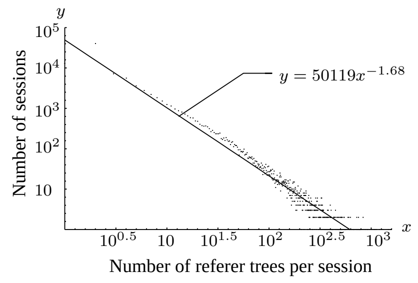


Figure 14: Number of referer trees per session (Avg:3.83)

5. RELATED WORK

Researchers have studied cognitive and behavioral aspects of user's Web search activities in the past [9, 10, 11, 12, 13, 14]. In these studies the main focus is how various subjective factors such as users' information need, knowledge, expertise and past experiences affect users' search behavior. Researchers also attempt to build cognitive or behavioral models (e.g. state transition graphs) to explain such behavior. In contrast, our study mainly focuses on *quantifying* the influence of Web search in people's daily Web access.

Our work is also related to earlier studies on how users surf the Web by following static links [15, 16, 17]. Compared to these studies we emphasize more on users' search behavior.

There has also been extensive research in general characteristic of Web queries [18, 19, 20]. A rather comprehensive review of such studies can be found in [21]. While these works mainly focus on reporting the statistics of Web queries by inspecting search engine logs, in this paper we are more concerned about the impact of search activities by studying Web search in a larger context of user's overall Web access.

6. CONCLUSION

In this paper, we tried to provide a quantitative picture on how users access the information on the Web using a 2.5-month Web trace data collected form the UCLA Computer Science Department.

We summarize some of our main findings as follows:

- We find that about 13.6% of all Web traffic is under the direct or indirect influence of search engines. In addition, search engines help users reach 20% more sites by presenting them in search results, that may be otherwise unreachable by the users.
- A typical Web user follows 5 links before she jumps to a new page, spending 5 minutes per page. In one sitting, she visits 22 pages residing on 5 Web sites.

One limitation of our study is that our observation was made on a potentially-biased user population. Therefore, some of the characteristics that we observed may not be generalizable to the entire Web user population. It will be an interesting future work to see how some of our observations may change when our quantification methods are applied to a larger and more general dataset.

7. ACKNOWLEDGEMENT

This material is based upon work supported partially by the National Science Foundation under Grant No. 0347993. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of NSF.

8. REFERENCES

[1] SIQSS: Internet and society study, detailed report. http://www.stanford.edu/group/siqss/Press_Release/internetStudy.html, 2000.

[2] Brian Morrissey. Search guiding more Web activity. http://www.clickz.com/news/article.php/2108921, 2003.

[3] J. Cho and S. Roy. Impact of Web search engines on page popularity. In *Proc. of WWW '04*, 2004.

[4] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. In *Proc. of WWW '98*, 1998.

[5] BitTorrent. http://bittorrent.com/.

[6] World Wide Web Consortium. Hypertext Transfer Protocol - HTTP/1.1. www.w3.org/Protocols/ rfc2616/rfc2616.html, 1999.

[7] World Wide Web Consortium. HTML 4.01 specifi cation. www.w3.org/TR/html4/, 1999.

[8] P. Baldi, P. Frasconi, and P. Smyth. *Modeling the Internet and the Web*, chapter Modeling and Understanding Human Behavior on the Web. WILEY, 2003.

[9] C. Hoelscher. How Internet experts search for information on the Web. In *Proc. of WebNet '98*, 1998.

[10] C. Holscher and G. Strube. Web search behavior of Internet experts and newbies. In *Proc. of WWW '99*, 1999.

[11] C.W. Choo and B. Detlor. Information seeking on the Web: An integrated model of browsing and searching. *First Monday*, 5(2), 2000.

[12] R. Navarro-Prieto, M. Scaife, and Y. Rogers. Cognitive strategies in web searching. In *Proc. of the 5th Conf. on Human Factors & the Web*, 1999.

[13] A. Broder. A taxonomy of Web search. *SIGIR Forum*, 36(2), 2002.

[14] D.E. Rose and D. Levinson. Understanding user goals in Web search. In *Proc. of WWW '04*, 2004.

[15] L.D. Catledge and J. Pitkow. Characterizing browsing strategies in the World-Wide Web. *Comp. Networks ISDN Syst.*, 27:1065–1073.

[16] L. Tauscher and S. Greenberg. Revisitation patterns in World Wide Web navigation. 1997.

[17] A. Cockburn and B. McKenzie. What do Web users do? an empirical analysis of Web use. *Int. J. Human Computer Studies*, 54:903 – 922.

[18] C. Silverstein, M. Henzinger, H. Marais, and M. Moricz. Analysis of a very large Web search engine query log. *SIGIR Forum*, 33(1):6 – 12, 1999.

[19] B.J. Jansen, A. Spink, and T Saracevic. Real life, real users, and real needs: A study and analysis of user queries on the Web. *Information Processing and Management*, 36(2):207 – 227, 2000.

[20] A. Spink, B.J. Jansen, D. Wolfram, and T. Saracevic. From E-Sex to E-Commerce: Web search changes. *IEEE Computer*, 35(3):107 – 109, 2002.

[21] B.J. Jansen and U. Pooch. A review of Web searching studies and a framework for future research. *JASIST*, 52(3):235 – 246, 2001.

Processing Top-N Queries in P2P-based Web Integration Systems with Probabilistic Guarantees

Katja Hose   Marcel Karnstedt   Kai-Uwe Sattler   Daniel Zinn
Department of Computer Science and Automation, TU Ilmenau
P.O. Box 100565, D-98684 Ilmenau, Germany

ABSTRACT

Efficient query processing in P2P-based Web integration systems poses a variety of challenges resulting from the strict decentralization and limited knowledge. As a special problem in this context we consider the evaluation of top-N queries on structured data. Due to the characteristics of large-scaled P2P systems it is nearly impossible to guarantee complete and exact query answers without exhaustive search, which usually ends in flooding the network. In this paper, we address this problem by presenting an approach relying on histogram-based routing filters. These allow for reducing the number of queried peers as well as for giving probabilistic guarantees concerning the goodness of the answer.

1. INTRODUCTION

Schema-based Peer-to-Peer (P2P) systems, also called Peer Data Management Systems (PDMS) are a natural extension of federated database systems which are studied since the early eighties. PDMS add features of the P2P paradigm (namely autonomous peers with equal rights and opportunities, self-organization as well as avoiding global knowledge) to the virtual data integration approach resulting in the following characteristics: each peer can provide its own database with its own schema, can answer queries, and is linked to a small number of neighbors via mappings representing schema correspondences.

Though, PDMS are surely not a replacement for enterprise data integration solutions, they are a promising approach for loosely-coupled scenarios at Internet-scale, such as Web integration systems. However, the expected advantages like robustness, scalability and self-organization come not for free: In a large-scale, highly dynamic P2P system it is nearly impossible to guarantee a complete and exact query answer. The reasons for this are among others possibly incomplete or incorrect mappings, data heterogeneities, incomplete information about data placement and distribution and the impracticality of an exhaustive flooding. Therefore, best effort query techniques such as similarity selection and join, nearest neighbor search and particularly top-N operators are most appropriate. By "best effort" we mean, that we do not aim for exact results or guarantees but instead try to find the best possible solution wrt. the available local knowledge. However, even if we relax exactness or completeness requirements we still need estimations or predictions about the error rate. For a top-N query this means for example that we can give a probabilistic guarantee that x percent of the retrieved objects are among the top N objects which we would get if we had asked all peers in the system.

Eighth International Workshop on the Web and Databases (WebDB 2005)
June 16-17, 2005, Baltimore, Maryland

Consider a scenario from the scientific domain where a virtual astronomical observatory is built by integrating individual peers (observatories) offering data about sky observations. We assume XML as the underlying data format and XQuery as the common query language. A typical query in this scenario would ask for astronomical objects that match a condition to a certain degree. For instance a researcher could query for objects next to a certain point in space defined by a set of coordinates or objects with an average brightness "similar" to a predefined one.Thus, we can describe the pattern of such queries as follows:

```
for $i in fn:doc("doc.xml")/path
order by rank($i/path) limit N return ...
```

where *rank* is a ranking function defining an order on the elements and **limit** restricts the result set to the first N elements returned by **order by**. The ranking function can be defined simply by exploiting the order of the attribute values or even using the euclidian distance, e.g., for coordinate values. An example from the above mentioned scenario is the following query asking for the 10 stars closest to a given sky position:

```
for $s in fn:doc("sky.xml")//objects
order by distance($s/rascension,
              $s/declination, 160, 20)
limit 10 return ...
```

In order to allow an efficient evaluation such queries should be treated using a special top-N operator $\hat{\sigma}^{N}_{rank}(E)$ where E denotes a set of elements. This operator returns the N highest ranked elements from E. The implementation of this operator as well as strategies for processing in a PDMS are the subjects of our work and are described in the following sections.

2. RELATED WORK

Several approaches and efficient strategies for top-N query processing have been developed concerning classical RDBMS. As introduced in [5, 3] one promising approach is to use existing data structures (i.e. indexes) and statistics (i.e. histograms) for mapping a top-N query into a traditional selection query. The aim is to determine a tight n-dimensional rectangle that contains all the top-N tuples for a given query but as few additional tuples as possible thereby dealing with histogram bucket boundaries instead of single tuple values. The algorithm is based on finding a minimal score that is sent along with the query as selection predicate guaranteeing at least k elements that are ranked higher. Another approach that tries to quantify the risk of restarting a query probabilistically is presented in [8], but the authors only deal with the problem of determining an optimal selection cutoff value.

TPUT [4] is an algorithm that was designed for efficiently calculating top-N aggregate queries in distributed networks. This algorithm represents an enhanced version of the *Threshold Algorithm (TA)* that was independently developed by multiple groups [9, 11]. Further variants of TA have been developed for multimedia repositories [6], distributed preference queries over web-accessible databases [12], and ranking query results of structured databases [1]. Another work [13] that is based on TA introduces a family of approximate top-N algorithms that probabilistically try to predict when it is safe to drop candidate items and to prune the index scans that are an integral part of TA. All these algorithms, however, need several round-trips in order to retrieve the final result whereas our approach tries to ask each peer only once.

The algorithms presented in [2] try to improve top-N query processing by dynamically collecting query statistics that allow for better routing when the query is issued the next time. The first time, however, all peers have to participate in processing the query while several round-trips are required in order to retrieve the final result. This often leads to situations where peer have to wait for each other.

3. EVALUATING TOP-N QUERIES

3.1 Classification

Evaluating top-N queries provides four different main approaches each describing one main principle and thus one general class of more detailed processing techniques. Without loss of generality we assume a tree view on all peers established at the query initiating peer, the so called peer or query tree.

1. Naive top-N: collect N (if possible) data elements from each available peer, sort and evaluate at initiator.
2. Partial top-N: same as naive strategy, but combine and prune results at peers passed on the way back to the initiator.
3. Globally optimized Top-N: minimize the set of queried peers before actually querying them – based on global information, e.g., using a global index structure.
4. Locally optimized Top-N: also minimize the set of queried peers, but decide at each involved peer again which peers to discard, based on locally available information.

Class 1 and 2 algorithms promise poor efficiency. The problem with strategies of class 3 is their need for global information. Global knowledge may not be achievable, because peers do not provide information about all their data. Furthermore, maintenance and building tasks are expensive and top-N queries over the attribute(s) not indexed require flooding the network or an index rebuild (e.g., re-hash). In this work we focus on class 4 approaches, based on knowledge gained from feedback of processed queries. We alternatively call this class locally pruned top-N, because at each peer again we prune the set of possible query paths using only locally available information.

3.2 Histograms and Routing Filters

In order to be able to prune query paths locally we need information about the data distribution at each peer. Approaches based on local index structures have been shown to be suitable ([7]) and we adopted them to our needs of schema and instance level information in previous works. The developed data structures do not simply index on pre-selected attributes, rather they collect information about all attributes occurring with high frequencies, thus we call them *routing filters*. Because histograms are successfully used in a wide variety of optimization and estimation techniques we decided to integrate them into the filters in order to approximate data distribution on instance level.

At each peer for each established connection to a neighbor $neigh$ one separate filter is maintained, which includes schema and instance level information for all peers reachable by querying $neigh$ (that is, all peers "behind" $neigh$). Routing filters are built and maintained using a query feedback approach, thus they are empty when a peer joins and grow as time passes by according to the received query results. If they are not limited to a certain horizon around the owning peer (e.g., by using a hop count [7]) and if we assume that no peer leaves the network, they will finally converge to global knowledge wrt. the query workload.

Histograms approximate data distributions by partitioning a sort parameter into intervals (buckets). An approximated source parameter value (we focus on frequencies) is stored for each bucket. A wide variety of histograms has been developed. We use *V-Optimal histograms*, which partition the sort parameter in such a manner that the variance of source parameter values, i.e., frequencies, within each bucket is minimized [10]. The value domain is approximated by a continuous distribution in the bucket range. Note that our algorithms do not depend on the optimality of V-Optimal histograms. The only thing they rely on is a characteristical value describing the error of the assumed frequency distribution in each bucket. V-Optimal histograms lead to good approximations.

As routing filters and types of histogram are not focus of this work, we omit details. In Figure 1 we picture routing filters exemplarily. In that figure simple histograms on attribute 'x' for two neighbors of the filter owning peer are depicted. In the next section we will show how to exploit the routing filters' information for optimizing top-N query processing and how to provide probabilistic guarantees for the result.

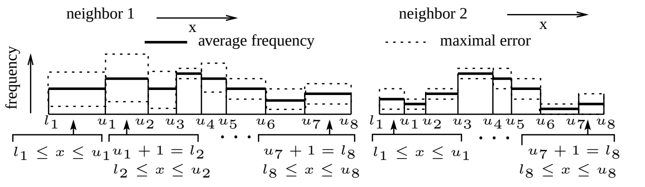


Figure 1: Histograms of routing filters on 'x' for two neighbors

3.3 Histograms and Probabilities

Routing filters combine information about schema and instance level. What is important for optimizing the evaluation of top-N queries is the approximation on instance level provided by the histograms, therefore in the following we focus on these histograms. Moreover, due to the lack of space, we concentrate on the evaluation of only nearest neighbor queries as a prominent representative of top-N queries. The proposed algorithm draws conclusions about the frequency distribution based on the average frequency and the maximum absolute error per bucket. In this subsection we will at first introduce a notation before describing the main algorithm. For simplicity we only consider discrete sort parameters and one-dimensional histograms. Continuous values can either be regarded as special case where values have to be discretized before they can be processed alike or similar algorithms can be developed that are optimized for dealing with such sort parameters. The proposed methodology as described in the following can also be used with multi-dimensional histograms. Evaluating top-N queries based on ranking functions defined over more than one attribute, we have to revert to multi-dimensional histograms. The introduced calculations and approximations hold in analogy.

Notation & Basics Given n data values, represented by function $R : \{1..N\} \rightarrow \mathbb{N}$, we can determine the frequency $F(k)$ for any sort parameter value k with $F(k) = |\{e|R(e) = k\}|$. F's domain is now being partitioned into buckets that can have different sizes.

Several statistical information is assigned to these buckets in order to approximate the original function F. Let bucket B_i be defined as $B_i = \{l_i, \ldots, u_i\}$, where l_i specifies the lower and u_i the upper boundary. For each bucket the average frequency h and the maximum absolute error e are defined as follows:

$$h_i := \frac{\sum_{k=l_i}^{u_i} F(k)}{u_i - l_i + 1} \quad , \quad e_i := \max_{k=l_i..u_i} \{|F(k) - h_i|\}$$

For the rest of this paper we will use the terms T and B_i for symbolizing intervals and buckets as well as for referring to the corresponding set of elements. Considering an arbitrary interval of interest T, $T \subseteq B_i$ and $card(T) := \sum_{t \in T} F(t)$ the following boundary can be identified for the minimum cardinality:

$$card_{min} = \max\{|T| \cdot \lceil h_i - e_i \rceil, |B_i| \cdot h_i - (|B_i \setminus T| \cdot \lfloor h_i + e_i \rfloor)\} \qquad (1)$$

A boundary for the maximum cardinality can be defined in analogy. Considering subsets T that are not necessarily limited to one bucket, similar equations can be defined. For any bucket B_j that is partly included in T, $card(T \cap B_j)$ minimum and maximum values can be determined. For any bucket B_j that is completely included in T, we can exactly determine the cardinality via $card(B_j) = |B_j| \cdot h_j$. By summing up these values $card(T)$ can be restricted by minimum and maximum values but usually not determined exactly. However, assuming a probability distribution for $F(i)$ in "cut" buckets, a probability distribution for $card(T)$ can be calculated. Based on this distribution our top-N algorithm that is defined in the following provides probabilistic guarantees.

Specification for Top-N Query Algorithm Our main algorithm takes the following input parameters: the queried value X, the top-number N, a value M_{in} ("missed"), two probabilities P_{in}^C (C for "correct") and P^{OP} (OP for "one pass"), and a distance $dist$ (not provided to, but determined by the initiating peer). It always returns an $(N{+}2)$ tuple $Res := (data^N, M, P)$[1], which complies with the following statements: With a probability of minimum P, $P \geq P_{in}^C$, the algorithm missed at most M data items – that is the globally correct top-N result based on all data in the network contains at most M data items that are not returned by our algorithm. With a minimum probability of P^{OP}, M is equal to or less than M_{in}. Thus, P^{OP} represents a parameter adjusting the probability of having to re-initiate a query. We handle P^{OP} as an internal system parameter, in our first tests we set $P^{OP} = 0.9$. When re-initiating, P^{OP} must be shifted accordingly. Our algorithm tries to minimize costs by minimizing the number of peers involved for processing the query – with respect to the (probabilistic) boundaries mentioned above.

Problem statement and the algorithm's specification are related as follows: A user initiates a query and provides a required quality guarantee that x percent of the retrieved objects are among the real top-N objects ("real" symbolizes the result we would get if all peers are queried). We reflect x to the value $M_{in} := N - x \cdot N$. Adjusting P_{in}^C provides an additional possibility to affect the accuracy of the retrieved result and therefore the performance of query processing. The algorithm guarantees by a probability of P_{in}^C that the returned result includes at least x percentage of the globally correct result. If not specified, the default value is $P_{in}^C = 1$. P_{in}^C defines the maximum acceptable uncertainness when evaluating the histograms and deciding which peers to discard.

Synopsis for Top-N Query Algorithm In general, all peers execute the following steps:

1. determine the set of neighbors that promises the lowest execution costs while adhering to the provided quality limits
2. distribute the quality limits among the peers to query
3. forward the query with adopted quality limits to the determined subset of peers
4. wait for answers, combine local data and results, set resulting guarantees, return a combined answer to the querying peer

Figure 2 summarizes these steps in an algorithmic form. The interesting details are how to determine the subset of peers to query and how to distribute the required restrictions among these neighbors.

```
Input:  $N$: number, $X$: queried value, $M_{in}$: number,
        $P_{in}^C$: probability, $P^{OP}$: probability, $dist$: number

if initiating peer then $dist$ = calc-dist($N, X, M_{in}, P^{OP}$); fi
$A, dist, P_L^C, M_L$ = calc-neighs($N, X, M_{in}, P_{in}^C, dist$);
$P_{ch}^C, M_{ch}$ = distribute-missed($P_{in}^C, M_{in}, P_L^C, M_L$);
foreach peer $a \in A$ do                    /* in parallel! */
    $R_{ch}, M_{ch}^a, P_{ch}^{C,a}$ = $a \rightarrow$process-query($N, X, M_{ch}, P_{ch}^C, P^{OP}, dist$);
done
$R_L, M_L, P_L^C$ = combine($R_L, M_L, P_L^C, R_{ch}, M_{ch}^a, P_{ch}^{C,a}$);
if initiating peer then $M_L$ = adjust-missed($R_L, M_L, dist$); fi
send-answer($R_L, M_L, P_L^C$);
```

Figure 2: Procedure *process-query*

At first, the query initiating peer determines an initial distance $dist$. In order to do so, the locally available histograms are regarded as a view on the global data distribution. Thus, the global range of the queried attribute is expected to equal the maximal range combined over all neighbor histograms and the local data. $dist$ reflects the smallest range around the queried point X in which at least $N - M_{in}$ elements are expected by a probability of P^{OP}. This interval is determined using a sub-algorithm called ***calc-dist*** and is provided to all subsequently queried peers and used by them for evaluating possible subsets of peers. This is applicable because we assume the initiating peer having gathered global knowledge in the queried attribute's histogram wrt. the established peer tree, and therefore simplifies the following explanations. Different assumptions, e.g., exploiting limited knowledge, result in having to calculate $dist$ repeatedly at each peer and come along with some minor algorithmic modifications. Because each queried peer always returning N elements (if available, all the same if these values lie in interval $[X - dist, X + dist]$ or not), the algorithm works with each $dist$, even if it is 0 or randomly chosen. As we will see, the problem is that we will hardly discard any peers when choosing it too large – in general choosing it too low will result in missing more top elements and having to decrease the returned guarantee value accordingly in order to match the algorithm's specification. Whether probability P_{in}^C holds or not when pruning query paths is only tested in interval $dist$, thus we should choose it appropriately.

In the next paragraphs we will briefly describe each of the subroutines called in Figure 2.

Procedure *calc-dist* An interval I is iteratively and symmetrically widened starting with $I = [X]$. In an iteration step $d = 0, 1, \ldots, \max\{\max_i\{u_i\} - X, X - \min_i\{l_i\}\}$, it is tested whether the probability that at least $N - M_{in}$ data items lie within the interval $I := [X - d, X + d]$ is greater than or equal to P^{OP}. If so, d is returned. Note that the algorithm always ends.

Procedure *calc-neighs* The difficult part is to find all possible subsets of neighbors at each involved peer that fulfill the algorithm's specification in respect to M_{in} and P_{in}^C. For each considered subset A' of all peers A the probability p that the peers in $A \setminus A'$ provide at most M_{in} values inside interval I is determined. If $p \geq P_{in}^C$ holds we may decide to query only the peers from A'. If the final result is completely included in I, discarding these peers does not violate the probabilistic limits. This can only be checked at the initiating peer after the final result is received. If any element

[1]Given that the domain of R includes at least N values

of the result is not included in $[X-dist-1, X+dist+1]$ we have to adjust the returned M_L in order to adhere to the specification of our algorithm, which is done calling *adjust-missed*.

Procedure *distribute-missed* The algorithm allows for missing up to M_{in} relevant data items in $dist$, thus we have to guarantee:

$$P\left(\begin{array}{l}\text{globally discarded peers provide less than or}\\ \text{equal to } M_{in} \text{ values in } [X-dist, X+dist]\end{array}\right) \geq P^C_{in} \tag{2}$$

Enforcing equation 2 and handling final result elements from outside $dist$ accordingly our algorithm works correct as specified.

As the algorithm allows each participating peer to discard some of its neighbors, we have to "distribute" the allowed overall error. If we prune a query path we discard a single set of peers A_i. Assume all globally discarded sets are $A := \{A_i | \forall i \neq j : A_i \cap A_j = \emptyset; i, j = 1, \ldots, v\}$, where v is the number of totally pruned query paths, and M_D is the set of all possible corresponding distributions of M, $M_D := \{\{M_i | M_i \geq 0 \wedge \sum_i M_i = M; i = 1, \ldots, v\}\}$. Let $P_M(A_i)$, respectively $P_{\leq M}(A_i)$, denote the probability that all peers from A_i have exactly, respectively at most, M values in $[X - dist, X + dist]$. Assuming that the missed values are distributed independently from the peers the following equation holds:

$$P_M(\bigcup_i A_i) = \sum_{M_d \in M_D} \prod_{M_i \in M_d} P_{M_i}(A_i) \tag{3}$$

With respect to a certain distribution M'_d of M_{in}, equation 4 can be used to estimate p when distributing data items and probabilities over several peers:

$$p = P_{\leq M}(\bigcup_i A_i) \geq \prod_{M_i \in M'_d} P_{\leq M_i}(A_i) \geq P^C_{in} \tag{4}$$

The right unequal sign exactly is what the algorithm guarantees. We omit the simple proof for the left unequal sign – listing the included factors and summands the principle of proof gets evident. Thus, by ensuring that the product of $P_{\leq M_i}(A_i)$ forall i is greater than or equal to P^C_{in} and that the sum of all M_i is not greater than M_{in}, we can guarantee that equation 2 holds and the result complies with our algorithm's specification. Another interesting point is which distributions of M_{in} to consider. In our basic version we distribute M_{in} uniformly over all queried peers, more sophisticated approaches could weight this distribution according to the amount of elements expected from each peer. Based on these considerations, our algorithm determines possible M_{in} and P^C_{in} parameters for forwarding the query to neighbored peers.

While we are currently passing one value for M_{in} and P^C_{in} to neighbored peers, an improved algorithm can pass a whole probability distribution of possible M_{in} values. Then it would be possible to use the exact equation 3 rather than the approximation 4.

Procedure *adjust-missed* As we have mentioned above we have to adjust the missed value that is returned if any element e of the final result does not lie inside the interval $[X-dist-1, X+dist+1]$. In case we discarded at least one peer, we could have missed one "better" (closer to X) element e' in $[X-dist-1, X+dist+1]$ for each element e. This is due to the fact that all tests whether to discard a peer or not are only applied to the interval $[X-dist, X+dist]$. Therefore, we adjust M by:
$M = \max\{M_L, |\text{result items outside } [X - dist - 1, X + dist + 1]|\}$.

Calculating Probabilities Finally, we give a brief look at how to calculate $P_{\leq M}(A_i)$, $P_{\geq M}(A_i)$ respectively. The performance and correctness of the whole processing strategy strongly depends on these calculations, as we will show in Section 4. Following from

$$P_{\leq M}(A_i) = 1 - P_{\geq M+1}(A_i)$$

we only need to implement one of these functions. Let I denote the interval of interest. First of all, we can calculate the exact values for all buckets that are completely included in I. After that, we could calculate minima and maxima of all subsets of buckets that are 'cut' via I. The overall minimum/maximum is the sum of all these minima/maxima plus the exact values determined before. The crucial question is how to determine a probability distribution for all values in between one arbitrary interval $[min, max]$. This may be handled by assuming for instance uniformly distributed or normally distributed frequencies. When assuming normally distributed values we can revert to the variance in a bucket in order to approximate $P_{\leq M}(A_i)$. For simplicity, in our first implementation we simply store the variance for each bucket. Naturally, this increases the space needed by the histograms which could be used to improve their accuracy – in order to bypass this disadvantage we could as well approximate the variance using the stored maximal error (and vise versa), but we will in general be limited to only use a rule of thumb for this purpose. Corresponding investigations are part of our ongoing work.

$P_{\leq M}(A_i)$ can now be calculated using the individual distributions for $F(k)$ for all $k \in I$ on all peers $\in A_i$ as separate random variables, which requires computing the convolution between them and therefore mostly depends on solving complex integrals. [13] gives interesting approximations and equations of how to compute the convolution for uniform, Poisson and arbitrary distributions. Due to the lack of space we for our part assume normal distributions, and thus can easily calculate an approximated probability distribution for $card(I)$. The sum of n $N(\mu_j, \sigma_j^2)$, normally distributed random variables, $N(\mu, \sigma^2) := N(\sum_n \mu_j, \sum_n \sigma_j^2)$, is normally distributed, too. We only have to truncate the resulting probability distribution in between $[min, max]$. If $F_{N(\mu,\sigma^2)}$ denotes the cumulative distribution function for a $N(\mu, \sigma^2)$ normally distributed random variable, $P_{\leq M}(A_i)$ can be approximated as:

$$P_{\leq M}(A_i) = \begin{cases} 0 & , min > M \\ 1 & , max \leq M \\ \frac{F_{N(\mu,\sigma^2)}(M) - F_{N(\mu,\sigma^2}(min)}{F_{N(\mu,\sigma^2)}(max) - F_{N(\mu,\sigma^2}(min))} & , \text{else} \end{cases} \tag{5}$$

3.4 Pruning Query Paths

The main intuition behind our method is to prune query paths based on probabilistically guaranteed assumptions in order to optimize processing costs. During the introduction of our algorithms we indicated when and where any query path is pruned, but we did not mention how costs are integrated. This is done by sorting all subsets of directly connected peers according to their costs, starting with the cheapest one (which only includes the local peer). Running through this list, the algorithm stops investigating further subsets when the first set matching the probabilistic conditions is found.

Our prototype algorithm implements a greedy philosophy: each peer tries to discard as many neighbors as possible. This early pruning results in discarding larger subtrees than pruning in later phases of processing. In the current stage of development our implementation only considers the number of queried peers as cost factor. Intuitively, these cost calculations should be extended to take into consideration, e.g., the number of messages, bandwidth, data volume, hops or whatever measure is suitable for expressing costs. This also includes lowering the amount of elements returned by each peer below N, but we have to keep in mind that this goes along with algorithmic modifications and in our current implementation ensures correctness (even when unexpected peer failures occur). A more sophisticated approach could try to deduce trade-offs between the costs for one subset and the achievable percentage of the global top-N result. In this case the system, or the user in an interactive way, may decide which trade-off to choose, rather than restrict the processing to minimal costs. This strategy reflects Quality-of-Service approaches but is not the focus of this paper.

Example Figure 3 shows how our algorithm processes a query issued at peer 1 with the following parameters: queried value $X = 5$, number of queried top items $N = 5$, the number of allowed missed values $M = 2$ (for illustration M symbolizes M_{in}). We assume a considered distance $dist = 2$ and probability $P_{in}^{C} = 1$.

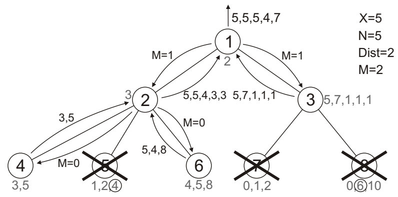


Figure 3: Example for applying algorithm

At first peer 1 forwards the query to its neighbors 2 and 3 distributing M equally among them, both peers are acting likewise but do not forward the query to peers 5, 7 and 8. 5 and 8 are pruned because peers 2 and 3 are allowed to miss one data item each. Peer 7 can be pruned because it has no data item in the interval of interest defined by X and $dist$. After having merged, ranked and pruned the received neighbor and local results, peers 2 and 3 send their results to peer 1 which acts likewise and provides the final result.

4. EVALUATION

In this section we will analyze and interpret the results that we gained running some experiments. We use a simple simulation environment that allows for using routing filters based on histograms as index structures. A bundle of extended tests will follow in future work. For each bucket we store the average frequency, the variance, and the maximum error between average and value frequencies. For simplicity but without loss of generality we assume global bucket boundaries for all histograms. Note that this is no requirement for our algorithm. Using this simple environment we are able to evaluate our algorithm by comparing the processed results and the number of queried peers. The number of resulting messages and generated data volume is not analyzed in the following due to a strong correlation with the number of queried peers.

4.1 Experiments

Our experiments will show the quality of the guarantees given by our algorithm and the performance benefit of pruning. We distinguish between (i) probabilistic guarantee and (ii) specified correctness in terms of the ratio $1 - \frac{M_{in}}{N}$ running from 0 ($M_{in} = N$) to 1 ($M_{in} = 0$), i.e., with a probabilistic guarantee of p the result misses only M_{in} result items in comparison to the global top-N result. What we will evaluate in particular includes:

- The quality of our probabilistic guarantees using histograms.
- The benefit of pruning using histograms.
- The impacts of frequency distribution and network structure.
- The influence of user parameters (P_{in}^{C}, N, M_{in}) on query results and guaranteed correctness (defined as $1 - \frac{M_{out}}{N}$).

Not in the focus of this work but part of our future work is evaluating the impact of dynamics and filter maintenance strategies.

Impact of value frequency distribution within each bucket In our first experiments we investigated the impact of the approximated distribution within each bucket, i.e., what happens if the actual frequency distribution does not comply with a normal distribution. The results shown in figures 4 (left) and 5 (left) represent the average of several test runs with $N = 100$ and $P_{in}^{C} = 0.6$ where each run queries another target value X of the same attribute. The P2P network forms a tree of 100 peers with each peer (except leaf nodes) having 4 children. Our astronomical test data is distributed arbitrarily among all peers. Both figures show that P_{out}^{C} gets higher with increasing specified correctness. This is due to the fact that with higher specified correctness the algorithm has to ask more peers. Doing so the probability of missing any data items decreases which is represented by increasing P_{out}^{C}.

Figure 4 (left) shows the results of queries where only those buckets are queried whose value frequency distributions more or less comply with a normal distribution. As we have expected, our algorithm works fine in such situations since this is exactly what it assumes for each bucket. The guaranteed correctness that the algorithm gives for its result is always lower than the *actual correctness* that we define as the ratio $\frac{|R_G \cap R_P|}{|R_P|}$, where R_G represents the global and R_P the query result. This means that the algorithm always gives right guarantees. Both lines are situated above the straight line representing the specified correctness. This means that the result is always even better than demanded by the user.

Figure 5 (left) shows what happens when querying buckets whose actual frequency distributions do not at all comply with the assumed normal distribution. Both guaranteed and actual correctness often have lower values than the specified correctness. This obviously leads to incorrect query results in terms of having missed more values of the global top-N result than specified by the user and guaranteed by the algorithm. Since our algorithm is based on assuming a normal distribution this is not astonishing. Combining this basic approach with other distributions should lead to better results which we will investigate in future work.

Influence of N In further tests we analyzed the influence of the number of queried values N on the result quality. All tests were run on the same network structure and with the same parameters as our first ones. Instead of varying the specified correctness we set the specified M_{in} value to $\frac{N}{2}$ in order to have an equal correctness ratio for all test runs. As revealed by figure 4 (right) we expected the quality to decrease with higher N, our approach works well with small N. For larger N, however, the guaranteed correctness is higher than the actual correctness. This is due to the fact that with increasing N our approach has to combine more and more random variables. Therefore, small estimation errors for each variable result in a rather large overall error depending on the size of N. Since the reason for issuing top-N queries is retrieving only a few result items out of many, this can be regarded as a minor restriction of our approach. In order to increase the performance of our algorithm for larger N we could increase the number of buckets per histogram. Due to limited space we do not investigate this aspect of the optimal number of buckets any further.

Influence of P_{in}^{C} Specifying the value of P_{in}^{C} the user tries to minimize execution costs by taking the risk of retrieving a result that misses more than M_{in} data items of the global result. Thus, the higher P_{in}^{C} (i.e., the smaller the risk) the higher should be the number of queried peers, since asking more peers minimizes the risk of missing any result items. Exactly this is shown by the tests whose results are presented in figure 4 (middle), where the only differences to the preceding tests are varying P_{in}^{C} and setting N to 100 and M_{in} to 25, i.e., setting the specified correctness to 0.75.

Influence of network structure In order to show that our approach does not only work with one special network structure, we created other network trees. The results for a tree with an arbitrary number of neighbors between 0 and 6 is presented in figure 5 (middle). Because of all parameters being set to the same values as in our first tests and the test data being distributed arbitrarily among all peers, the differences in figures 4 (left) and 5 (middle) must be due to the network structure. Since the curve progression in both figures does not differ in any remarkable manner, we can

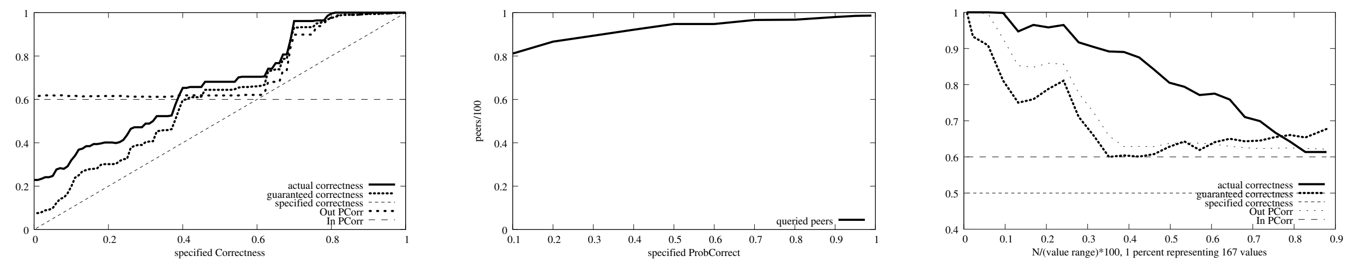


Figure 4: (left): Querying buckets with overall normal value frequency distribution, (middle): Influence of P_{in}^{C} on the number of queried peers, (right): Influence of N on guaranteed correctness

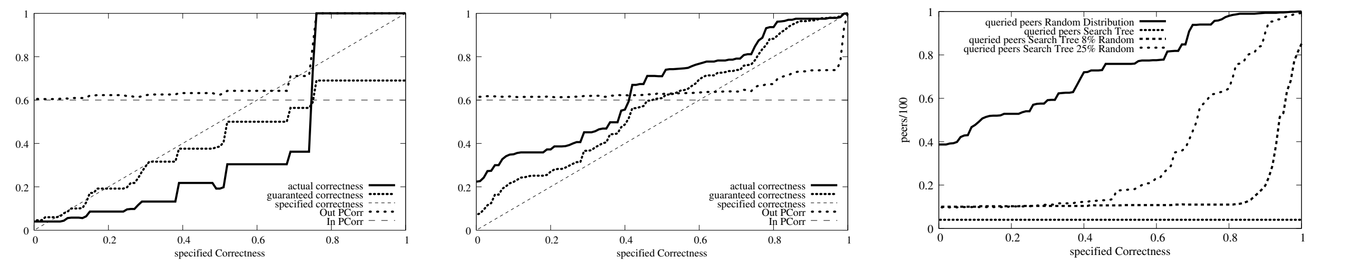


Figure 5: (left): Querying buckets with non-normal value frequency distribution, (middle): Influence of peer tree structure on guaranteed correctness, (right): Influence of P_{in}^{C} and data distribution on the number of queried peers

reason that the network structure does not influence quality or performance of our approach. Future work will consider more general network structures including cycles and dynamics.

Influence of data distribution on the number of queried peers There are two extreme cases concerning data distribution: (i) every peer has relevant data and (ii) we have a search tree where we can answer queries by asking a minimum number of peers. Figure 5 (right) shows the number of peers with varying specified correctness. The upper line originates from the same test run as depicted in figure 4 (left) where all peers had relevant data, so that with a specified correctness of 1 and $P_{in}^{C} = 1$ all peers had to be asked. The curves of all other test scenarios should lie in between the two corresponding curves. Figure 5 (right) proves this by showing the curves of two scenarios based on the search tree mentioned above but with a few per cent of randomly distributed data.

5. CONCLUSION AND OUTLOOK

Querying structured data in large-scale Web integration systems is often feasible only by applying best effort techniques such as top-N queries. In this paper, we have presented an approach for processing such queries in a schema-based P2P integration system. We have shown that we can reduce the number of asked peers by relaxing exactness requirements but are still able to guarantee a required percentage of the complete result by a certain probability.

However, the approach presented in this paper represents only a first step. There are two major directions for our ongoing work. First, so far we have considered only one-dimensional histograms and simple rank functions. Though, in principle the approach can be easily extended to multidimensional histograms there are still some open issues. We encountered additional open aspects of the proposed method, i.e., assumed frequency distribution and its impacts, which we plan to investigate more detailedly. Second, we have assumed that each participating peer already owns histograms for all of its neighbors and the queried attributes. In P2P environments we expect limited knowledge and high dynamics. This could be handled by modifying our algorithm and combining it with an incremental processing strategy and gossiping approaches for approximating global knowledge. In parallel, we will optimize the query feedback strategy and apply extended cost calculations.

6. REFERENCES

[1] S. Agrawal, S. Chaudhuri, G. Das, and A. Gionis. Automated ranking of database query results. In *CIDR*, 2003.

[2] W.-T. Balke, W. Nejdl, W. Siberski, and U. Thaden. Progressive distributed top k retrieval in peer-to-peer networks. In *ICDE'05*, 2005.

[3] N. Bruno, S. Chaudhuri, and L. Gravano. Top-k selection queries over relational databases: Mapping strategies and performance evaluation. *ACM TODS, Vol. 27, No. 2*, 2002.

[4] P. Cao and Z. Wang. Efficient top-k query calculation in distributed networks. In *PODC '04*, pages 206–215, 2004.

[5] S. Chaudhuri and L. Gravano. Evaluating top-k selection queries. In *VLDB'99*, pages 397–410, 1999.

[6] S. Chaudhuri, L. Gravano, and A. Marian. Optimizing queries over multimedia repositories. *IEEE TKDE*, 16(8):992–1009, 2004.

[7] A. Crespo and H. Garcia-Molina. Routing indices for peer-to-peer systems. In *22nd Int. Conf. on Distributed Computing Systems*, pages 23–32, July 2002.

[8] D. Donjerkovic and R. Ramakrishnan. Probabilistic optimization of top n queries. In *VLDB'99*, pages 411–422, 1999.

[9] U. Güntzer, W.-T. Balke, and W. Kiessling. Optimizing multi-feature queries for image databases. In *VLDB 2000*, pages 419–428, 2000.

[10] Y. E. Ioannidis and V. Poosala. Balancing histogram optimality and practicality for query result size estimation. In *ACM SIGMOD'95*, pages 233–244. ACM Press, 1995.

[11] A. Lotem, M. Naor, and R. Fagin. Optimal aggregation algorithms for middleware. In *PODS'01*, Mar. 03 2001.

[12] A. Marian, N. Bruno, and L. Gravano. Evaluating top-queries over web-accessible databases. *ACM TODS'04*, 29(2):319–362, 2004.

[13] M. Theobald, G. Weikum, and R. Schenkel. Top-k query evaluation with probabilistic guarantees. In *VLDB 2004*, pages 648–659, 2004.

Context-Sensitive Keyword Search and Ranking for XML

Chavdar Botev
Cornell University
cbotev@cs.cornell.edu

Jayavel Shanmugasundaram
Cornell University
jai@cs.cornell.edu

1. INTRODUCTION

Traditionally, keyword-search-based information retrieval (IR) has focused on "flat" documents, which either do not have any inherent structure or have structure that is not exploited by the IR system. Thus, even if users wanted to search over only specific sub-sets and/or sub-parts of the documents, they still had to search over the entire document collection. In contrast, many emerging XML document collections have a hierarchical structure with semantic tags, which allows users to specify the context of their search more precisely.

As an illustration, consider a large and heterogeneous XML digital library that contains content ranging from Shakespeare's plays to scientific papers. A user who is interested in learning more about Shakespeare's plays can limit the scope of her search to just the relevant plays by specifying the following XPath query: //Play[author = 'William Shakespeare']. Thus, if she were to issue a keyword search query for the words "speech" and "process", she would only get XML element results in Shakespeare's plays that contain the keywords (such as a relevant <speech> element), and would not get XML elements about (say) voice recognition systems.

In this paper, we refer to this notion of restricting the search context as context-sensitive search. Supporting context-sensitive search introduces the following two challenges. The first challenge is to *efficiently find search results in the search context without having to touch irrelevant content.* In the example above, if Shakespeare's plays constitute only 0.1% of the entire content of the digital library, an efficient context-sensitive search implementation should not process the remaining 99.9%. The second challenge is to *effectively rank keyword search queries evaluated in a search context.* For example, in the popular TF-IDF scoring method, the IDF component represents the inverse document frequency of the query keywords in the entire document collection. However, using this IDF value directly for context-sensitive search can produce very unintuitive results.

As an illustration, consider the keyword search query for the words "speech" and "process" over Shakespeare's plays in XML [20]. We obtained the top 10 results from the query using one of the TF-IDF based XML ranking algorithms [1] published in the literature. We then took a heterogeneous XML collection consisting of IEEE INEX 2003 documents [11] (containing scientific papers) and Shakespeare's plays in XML, limited the search context to Shakespeare's plays, evaluated the same keyword search query and obtained the top 10 results using the same ranking algorithm. From the user's point of view, these are semantically identical queries. However, 9 of the top 10 results in the first set of results were not present in the second set!

Eighth International Workshop on the Web and Databases (WebDB 2005),
June 16-17, 2005, Baltimore, Maryland.

This wide variation in results can be explained as follows. It turns out that "speech" is very frequent in Shakespeare's plays (low IDF) but not in the entire collection (high IDF), while "process" is relatively frequent in the entire collection (low IDF) but not in Shakespeare's plays (high IDF). Consequently, the first experiment emphasized results that contained "process", while the second emphasized results that contained "speech". From the user's point of view, the first experiment is likely to return more meaningful results because "process" – and not "speech" – is the more uncommon word in the search context (Shakespeare's plays). Thus, the results of the second experiment are heavily skewed by elements that are not even in the search context.

In general, it is desirable for context-sensitive search results to be ranked as though the user query was evaluated over the search context in isolation (in our example, we desire that the second experiment should return the same results as the first experiment). We call this *context-sensitive ranking* (a similar concept is also referred as query-sensitive scoring in [7]).

In this paper, we present the design, implementation and evaluation of a system that addresses the above issues central to context-sensitive search. We make the following contributions:

1) We present enhanced inverted list structures and query evaluation algorithms that enable the efficient evaluation of context-sensitive keyword-search queries, without having to touch content irrelevant to the search context.
2) We develop a framework that allows for efficient context-sensitive ranking. We note that, unlike [7], our goal is not to develop new ranking algorithms for context-sensitive search. Rather, our goal is to provide a framework that will enable existing ranking methods to be used for context-sensitive ranking.

We also quantitatively evaluate the performance of our inverted list data structure and context-sensitive ranking based on existing XML ranking algorithms.

2. SYSTEM MODEL AND ARCHITECTURE

2.1 Model

We represent a collection of XML documents as a forest of trees $G = (N, E)$, where N is the set of XML element nodes and E is the set of containment edges relating vertices. Node u is a *parent* of a node v if there is an edge $(u, v) \in E$. Node u is an *ancestor* of a node v if there is a sequence of one or more containment edges that lead from u to v. The predicate *contains*(v, k) is true iff the vertex v directly or indirectly (through sub-elements) contains the keyword k, where k can be an element name, attribute name, attribute value, or other textual content. The granularity of query results is at the level of XML *elements* since returning specific elements (such as <speech>) usually gives more context information than returning the entire document ([1][5][8][21]).

2.2 System Architecture

Figure 1 shows our system architecture. The user query consists of two parts: (1) the keyword search query, and (2) the search context. Given a specification of the search context, the Context Evaluator returns a set of XML element IDs I such that the descendants of I define the search context. The search context consists of all the elements in the sub-trees rooted at the elements from the specification value. The Query Engine takes in the set of IDs I and the user keyword-search query, and ranks the elements in the search context with respect to the query. In producing the ranked query results, the Query Engine uses Index Structures and a Ranking Module. The latter is extensible with respect to various ranking functions.

For ease of exposition, we assume simple disjunctive queries (i.e., queries using only the 'or' operator) in this paper. The search context for these queries is specified using XPath. We use a standard XPath evaluator leveraging the work in [22].

The focus of this paper is on the Index Structures and Query Engine components. In the following section, we address three main challenges in designing these components. First, we show how to efficiently limit the search results to only those elements that occur in the search context. Second, we design a framework that supports context-sensitive ranking, i.e., ranking as though the query was evaluated over the search context in isolation. Third, we present an efficient evaluation algorithm that integrates these two solutions.

3. INDEXING AND QUERY EVALUATION

We first define the problem of context-sensitive ranking. We then describe our index structures, our ranking framework, and its integration with the query engine.

3.1 Context-Sensitive Ranking

Consider a set of XML elements E, a ranking algorithm R, and a keyword-search query Q. We define $RankRes_{E,R,Q}$ to be the set of pairs (e, s), where $e \in E$ and s is the score (rank) of e with respect to the query Q obtained using the ranking algorithm R. Intuitively, $RankRes_{E,R,Q}$ is the ranked results that we would have obtained if E was a stand-alone collection and the query results were ranked using R. Now, consider a context-sensitive search system S that uses the ranking algorithm R and is operational over an element collection E. Consider a user query (Q, SC), where Q is the keyword-search query, and $SC \subseteq E$ is the set of elements that constitute the user-specified search context. We define $CRankRes_{S,E,SC,R,Q}$ to be the set of pairs (e, s) returned by S for the user query (Q, SC), where $e \in SC$ and s is the score of e obtained using R in the SC.

We say that a system S supports *context-sensitive ranking* with respect to a ranking algorithm R iff for every set of XML elements E and for every user query (Q, SC) (where $SC \subseteq E$),

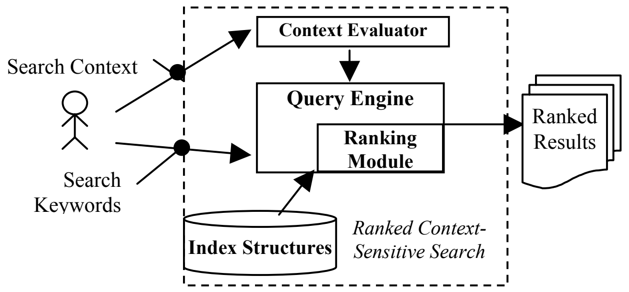


Figure 1: System Architecture

$RankRes_{SC,R,Q} = CRankRes_{S,E,SC,R,Q}$. In other words, the ranked results produced by S for a user query (Q, SC) are exactly the same as the ranked results produced in a stand-alone collection SC (using the ranking algorithm R). Thus, the system S provides users with the abstraction of working with a personalized document collection defined by the search context (SC), even though the search context may be part of a large heterogeneous collection (E) in the system. This will avoid ranking anomalies such as the one reported in the introduction.

We note that our focus is *not* on the design of new ranking algorithms R. Rather, our goal is to develop a general framework so that many *existing* ranking algorithms can be embedded in our system while still supporting context-sensitive ranking.

3.2 Index Structure

Our main goal is to enable the efficient evaluation of context-specific queries. A naïve strategy is to evaluate the user query over all the elements in the index, and check whether each element is present in the search context. This approach, however, is likely to be very inefficient when the search context is only a very small fraction of the entire collection.

We now propose an index structure that addresses the above issues by extending the inverted list index structure [18]. Two key modifications need to be made to make the inverted list applicable for context-sensitive XML keyword search. First, we need to capture the XML hierarchy in the inverted list entries so that nested elements that contain the query keywords can be returned as results; for this part, we build upon prior published work in this area [8][13]. Second, we need to structure the inverted list so that entries that do not belong to the search context can be easily skipped; this will enable the efficient evaluation of context-specific queries. We consider each in turn.

3.2.1 Capturing XML Hierarchy in Inverted Lists

A simple way to structure the inverted list is to store for each keyword the IDs of all elements that *directly or indirectly* contain the keyword. The downside of this approach is the associated space overhead. We need to store the IDs of the elements that directly contain the keyword *and the IDs of their ancestors*, because the ancestors too *indirectly* contain the keyword and should be returned as query results [8].

To address this space (and performance) overhead, Guo et al. [8] and Lee et al. [13] propose an encoding for element IDs called Dewey IDs. Each element is assigned a number that represents its relative position among its siblings in the XML document tree. The

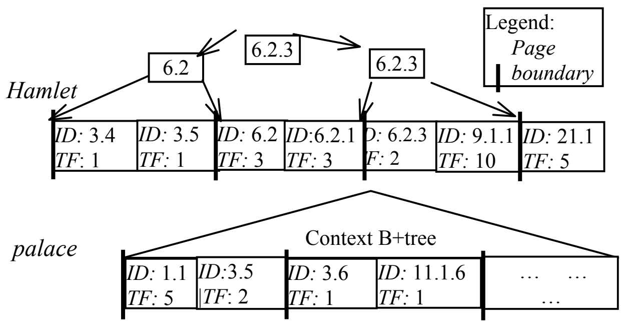


Figure 2: Inverted List Structure

path vector of the numbers from the root to an element uniquely identifies the latter. For example, the top element will be assigned Dewey ID 1; its children will be assigned 1.1, 1.2, …, etc.

Thus, ancestor-descendant relationships are implicitly captured and the inverted lists only need to store the IDs of the elements that *directly* contain the keyword. Figure 2 shows a fragment of the inverted list index using Dewey IDs for the words “Hamlet” and “palace” (ignore the page boundaries and B+-trees for now – they are used for context-sensitive search, discussed next). For each keyword, the entries contain the IDs of the XML elements that directly contain the keyword and other information such as the TF for the elements.

3.2.2 Efficiently Limiting the Search Context

To limit efficiently the search context *SC* for a query *Q*, we need to skip over the entries in the inverted list that do not belong to the search context. Also, recall from the system architecture that the Context Evaluator returns element IDs and all *descendants* of these elements constitute the search context; thus, we also need to identify the descendants efficiently.

It turns out that both of these problems can be addressed elegantly by building a B+-tree index on each inverted list based on the Dewey ID. Since the inverted lists are sorted based on the Dewey ID, the inverted lists can serve as the leaf level of the B+-tree; this avoids having to replicate the inverted list in the B+-tree, and leads to large space savings. Figure 2 shows this a sample index organization with fan-out 2.

The B+-tree index can be used to efficiently skip over irrelevant entries as follows. Assume that the Context Evaluator returns a list of Dewey IDs d_1, …, d_n that define the search context. We first start with d_1 (say 6.2) and probe the B+-tree of the relevant keyword inverted list to determine the smallest ID in the inverted list that is greater than or equal to d_1 in lexicographic order (6.2 in our example). We then start sequentially scanning the inverted list entries from that point onwards. Since all descendants of d_1 are clustered immediately after d_1 in the inverted list (as they share a common prefix), this scan will return all the descendant entries of d_1 (6.2, 6.2.1, 6.2.3). The scan will be stopped as soon as an entry is encountered whose prefix is not d_1 (and hence not a descendant of d_1). The same process is repeated for d_2, …, d_n. Note how all possible descendants of d_1 (e.g., 6.2.2,6.2.2.1, etc.) are *not* explicitly enumerated but only the descendants that appear in a relevant inverted list are retrieved.

3.3 Ranking Module

There are three important aspects that we consider in the design of the Ranking Module. First, it should support context-sensitive ranking, whereby only the elements that belong to the search context contribute to the ranks of the query results. Second, the Ranking Module should provide an extensible framework that supports a general class of ranking functions so that many existing (and possibly new) ranking schemes developed for non-context-sensitive ranking of XML results can be directly applied to the context-sensitive ranking problem. Finally, the Ranking Module should enable tight integration with the Query Engine so that context-specific ranking can be done efficiently.

In the following discussion, we will focus on TF-IDF based ranking methods since they are one of the most popular scoring methods used in IR. Furthermore, these scoring methods are well suited for presentation of the concepts of context-sensitive ranking. Indeed, the IDF value depends on the search context: it is the number of *search context elements* that contain a query keyword. Thus, such ranking methods can dynamically adapt based on whether the query keywords are frequent or rare in the search context (irrespective of whether they are frequent or rare in the entire collection). We note that the TF component depends on the content of an element and is usually independent of the context.

3.3.1 Modeling XML Ranking Functions

Given a user keyword search query k_1, …, k_n, issued over a search context *SC*, most TF-IDF based XML ranking methods [1][5][21] can be characterized as a function $\boldsymbol{R}(C_{k1}, \ldots, C_{kn}, E_{k1,e}, \ldots, E_{kn,e})$ that takes in the following parameters for each element $e \in SC$ and returns the score for *e*.

- $C_{k1}, \ldots, C_{kn}$: Each C_{ki} contains scoring information based on a query keyword k_i and the search context *SC* (e.g. the IDF for k_i)
- $E_{k1,e}, \ldots, E_{kn,e}$: Each $E_{ki,e}$ contains scoring information based on a keyword k_i the element *e*, and its descendants (e.g. TF of the keyword k_i with respect to *e*).

As an illustration, consider the XXL search engine [21] that uses the TF-IDF ranking method. The C_{kj} parameters are the element frequency values for the keyword k_j: C_{kj} = <number of elements containing k_j>/<number of elements in the search context> = $1/idf(k_j)$. The $E_{kj,e}$ parameters contain the normalized TF for the keyword k_j with respect to the element *e*: $E_{kj,e}$ = <number of occurrences of k_j in *e*>/<maximum term occurrences in *e*>. The overall-score function $\boldsymbol{R}$ combines the C_{kj} and the $E_{kj,e}$ parameters using the cosine similarity: $\boldsymbol{R}(C_{k1}, \ldots, C_{kn}, E_{k1,e}, \ldots, E_{kn,e})$ = $\sum_{j=1,..,n} \frac{(E_{kj,e}/C_{kj}) \times idf(k_j) tf(k_j, Q)}{\|e\|_2 \times \|Q\|_2}$, where $\|\cdot\|_2$ denotes the L_2 measure. The XSearch [1] and XIRQL [5] ranking methods can be similarly captured using the above framework.

3.3.2 Integration with the Index and Query Engine

For efficient query evaluation, our query processing algorithm (described in Section 3.4) relies on two fundamental principles: (1) During the evaluation of a keyword search query k_1, …, k_n, only the inverted list entries for the keywords k_1, …, k_n, that occur in the search context are accessed. No other inverted list entries are accessed, nor is the actual text content of an element accessed during query processing; (2) Query evaluation occurs in a single pass over the query keyword inverted lists (although a pre-processing pass is also used – details are in Section 3.4). Thus, once an index entry is accessed, it is not accessed again.

The above observations suggest that the ranking function parameter values should (1) be computed solely from the query keyword index entries in the search context, and (2) should be accumulated in a single pass over these index entries. We now formalize these notions. Consider the computation of a C_{kj} parameter. It can be computed using a function $\boldsymbol{FC_{kj}}$ that works like an accumulator: $\boldsymbol{FC_{kj}}$: Dom(C_{kj}) × *InvListEntry* → Dom(C_{kj}), where Dom(C_{kj}) is the domain of the values of C_{kj}. On each invocation, it takes in the current value of C_{kj} and the current index entry, and creates the new value of C_{kj}. When the last entry is processed, the result is the final value of the C_{kj} parameter. For example, consider the case of computing the IDF of the keyword k_j. The initial value of C_{kj} is 0, and each invocation of the $\boldsymbol{FC_{kj}}$ function simply increments the current value of C_{kj} by one over the number of elements in the search context. The IDF is the reciprocal of the final value of C_{kj} after processing all entries in the inverted list for k_j that occur in the search context.

Now, consider the computation of a $E_{kj,e}$ parameter. $E_{kj,e}$ is computed by the repeated application of the function $\boldsymbol{FE_{kj}}$: Dom($E_{kj,e}$) × *Node* × *Node* × *InvListEntry* → Dom($E_{kj,e}$). The function works like an accumulator and takes in the current value of $E_{kj,e}$ (first parameter), the current search context element *e* (second parameter), a descendant of *e* containing directly k_j (third parameter), and the index entry corresponding to the descendant (fourth parameter), and uses these arguments to compute the new value of the $E_{kj,e}$ parameter. This function captures the intuition that only information in the inverted list corresponding to an element's descendants is used to compute element-specific ranking information. For example, in XXL, the $\boldsymbol{FE_{kj}}$ function updates the TF of the keyword k_j with respect to the current search context element *e*.

3.4 Query Engine

We now describe how the Query Engine can efficiently support context-sensitive ranking for any ranking algorithm that can be characterized in terms of the ***R***, $\boldsymbol{FC_{kj}}$, and $\boldsymbol{FE_{kj}}$ functions (by efficient, we mean linear in the number of index entries in the search context). Our query-processing algorithm builds upon the work in [8] and extends it using a two-phase algorithm for context-sensitive search and ranking. In the first (pre-processing) phase, it computes context-sensitive information that is used for ranking (i.e., the C_{ki} or IDF values) by making a pass over the relevant parts of the query keyword inverted lists. In the second (regular) phase, it makes another pass over the same inverted list entries, and finds the top-k ranked query results.

The pre-processing phase is necessary because the regular phase cannot compute the overall rank of an element without the appropriate context-sensitive values. Thus, the regular phase cannot just keep track of the top-k results using a result heap since the score cannot be computed until the very end. Consequently, all elements should be retained and scored, which will be expensive. We note that the overhead of an extra phase is minimal since the first phase will bring all the relevant disk resident entries

```
01. procedure EvaluateQuery ($k_1$, $k_2$, …, $k_n$, $cid_1$, …, $cid_m$, N)
      // $k_1$ … $k_n$ are query keywords, $cid_1$ … $cid_m$ define search context,
      // N is the # query results  invList[$k_i$] is inverted list for $k_i$,
      // btree[$k_i$] is context B+-tree for $k_i$

      // Pre-processing phase: compute $C_{ki}$s
02. for (each $cid_j$) {
03.   for (each $k_i$) {
04.     ilPos = btree[$k_i$].probe($cid_j$); invList[$k_i$].startScan(ilPos);
05.     curEntry = invList[$k_i$].nextEntry;
06.     while ($cid_j$ is a prefix of curEntry.deweyID) {
07.       $C_{ki}$ = $FC_{ki}$($C_{ki}$, curEntry);
08.       invList[$k_i$].nextEntry;
09.} } }

      // Regular phase; compute top-N query results
10. resultHeap = empty; deweyStack = empty;
11. for (each $cid_j$) {
12.   for (each $k_i$) {
13.     ilPos = btree[$k_i$].probe($cid_j$); invList[$k_i$].startScan(ilPos);
14.     curEntry[$k_i$] = invList[$k_i$].nextEntry;
15.   }
16.   while ($\exists k_i$ such that $cid_j$ is a prefix of curEntry[$k_i$]) {
          // Get the next inverted list entry with the smallest DeweyID
17.     find $k_i$ such that curEntry[$k_i$].deweyID is the smallest deweyID;

        // Find the longest common prefix between deweyStack
        // and currentEntry.deweyId
18.     find largest lcp such that
          deweyStack[p] = curEntry[$k_i$].deweyId[p], 1 <= p <= lcp

        // Pop non-matching deweyStack entries
        // (their descendants have been fully processed)
19.     while (deweyStack.size > lcp) {
20.       sEntry = deweyStack.pop();
21.       score = R($C_{k1}$, …, $C_{km}$, sEntry.$E_{k1}$, …, sEntry.$E_{km}$);
22.       if score among top N seen so far,
            add (deweyStack.deweyID, score) to resultHeap;
23.     }

        // Push new ancestors (non-matching part of
        //currentEntry.deweyId) to deweyStack
24.     for (all i such that lcp < i <= currDeweyIdLen)
25.       { deweyStack.push(deweyStackEntry); }

        // Accumulate inverted list score information
26.     for (each deweyStack entry sEntry) {
27.       sEntry.$C_{ki}$ = $FE_{ki}$(sEntry.$C_{ki}$, sEntry.deweyID,
                            currentEntry.deweyID, currentEntry);
28. }  } // End of looping over all inverted lists

29.   Pop entries of deweyStack in context $cid_j$, and add to result heap
      (similar to lines 19-23)
30. } // End of processing $cid_j$
31. return resultHeap
```

Figure 3: Query Algorithm

into memory; thus, the second phase, which accesses the very same entries, has practically no overhead (see Section 4).

Figure 3 shows the query evaluation algorithm. The pre-processing phase (lines 02–09) works as follows. For each the search context ID cid_j, and for each query keyword k_i, it identifies the entries in the inverted list for k_i that are descendants of cid_j. It does this by probing the context B+-tree for keyword k_i using cid_j, and scanning the inverted list for k_i from that point onwards. The

Push

4	(1, 0)
3	(1, 0)
ID	**(E_1, E_2)**

(*a*)

Pop

3	(1, 0)
ID	**(E_1, E_2)**

(*b*)

Push

5	(1, 0)
3	(2, 0)
ID	**(E_1, E_2)**

(*c*)

Update

5	(1, 2)
3	(2, 2)
ID	**(E_1, E_2)**

(*d*)

Figure 4 The DeweyStack Transition

context-sensitive information C_{ki} is accumulated for each entry.

The second phase (lines 10-31) computes the top-k query results using the context-sensitive information C_{ki}. For efficiency, it maintains a stack of Dewey IDs, the DeweyStack. Using the DeweyStack, we can keep track of the score information of both the elements in the inverted lists and their *ancestors* (since the ancestors also indirectly contain the query keywords; note that this dependence is explicitly captured by the $\boldsymbol{FE_{ki}}$ function). The scoring information for the ancestors is updated while a descendant is being processed. This can be achieved using the DeweyStack because all ancestors and descendants are clustered in the B+-tree.

The algorithm for the second phase works as follows. For each cid_j, the relevant index entries for all the query keywords are scanned in parallel until the end of the current context (lines 16 - 28). The smallest ID from these lists is chosen (line 17). Based on it, the algorithm identifies the entries in the DeweyStack whose descendants have been fully processed; these entries are popped out of the stack, their ranks are computed using the function $\boldsymbol{R}$, and they are added to the result heap if they are among the top-k results so far (lines 19-23). The ancestors of the current smallest ID are then pushed onto the stack (lines 24-25), and the score information of all these ancestors is updated based on the current index entry (using function $\boldsymbol{FE_{ki}}$). This process is repeated until all of the relevant entries from the inverted list are processed.

As an illustration, consider the keyword query 'Hamlet palace' over the search context defined by the ID 3 using the index in Figure 2. The first phase computes C_{ki}'s (IDFs) by accessing the relevant entries: 3.4 and 3.5 for "Hamlet", and 3.5 and 3.6 for "palace". Then, in the second phase, the relevant entries are merged. First, the entry 3.4 is processed because it has the smallest DeweyID (Figure 4a). The stack state keeps track of the current score (TF value) for both 3.4 and all of its ancestors (3, in our case). The next entry processed is 3.5 from the first inverted list. Since the largest common prefix with the DeweyStack entry is 3, we can conclude that all descendants of 3 in the stack (3.4) do not have any further descendants in the search context. Thus, 3.4 is popped from the stack and is added to the result heap if it is one of the current top-k results (Figure 4b). Next, 3.5 from the first inverted list is pushed onto the stack (Figure 4c), and then 3.5 from the second list is used to update the TF values in the stack (Figure 4d). The algorithm then reads in 3.6, pops out 3.5, and continues in a similar manner.

4. EXPERIMENTAL RESULTS

We have implemented the system framework and algorithms described in the previous sections using C++. Using this framework, we have implemented context-sensitive versions of the following: XXL [21], XSEarch [1], and XIRQL [5]. We indexed a heterogeneous XML collection consisting of Shakespeare's plays [20], INEX IEEE articles [11], and SIGMOD Record in XML [19]. The size of the entire collection was 521MB, and the size of the inverted lists was 719MB. The space overhead for the context B+-trees to enable context-sensitive search was just 12MB. Our experiments were performed on a Pentium IV 2.2GHz processor with 1GB of RAM running Windows XP. When measuring performance, we used a cold operating system cache.

We performed two types of experiments. The first type measured the performance benefits of using context B+-trees to skip over irrelevant entries in the inverted lists. The second type of experiment measured how much context-sensitive ranking can influence the ranks of query results.

For the first set of experiments, we compared the performance of the following three implementations: (1) a baseline naïve approach that scans all the entries in the inverted lists, including those that do not belong to the search context (*Naïve*), (2) the algorithm in Section 3.4, but without using the pre-processing phase to compute the context-sensitive C_{ki} (IDF) values (*CSS*), (3) the full context-sensitive search and ranking algorithm described in Section 3.4 (*CSSR*). *CSS* only supports context-sensitive search, but does not support context-sensitive ranking. Thus, the performance difference between *CSSR* and *CSS* quantifies the performance overhead of context-sensitive ranking.

Figure 5 shows the performance results when the size of the search context (the percentage of the total number of elements that are in the search context) is varied. This suggests that context-sensitive search offers significant performance benefits (by up to a factor of 5) over *Naïve*. The latter does not skip over irrelevant entries in the inverted list. In contrast, CSS and CSSR show consistently better performance with smaller context sizes because they only have to scan the relevant portions of the inverted lists. We expect this difference to be even bigger for larger databases or more selective keywords.

Interestingly, there is practically no overhead for *CSSR* as compared to *CSS*, even though *CSSR* makes two passes over the relevant entries. The reason is that the first pass of *CSSR* brings all the relevant entries into memory; hence, the second scan has no measurable overhead. This suggests that context-sensitive ranking adds no measurable overhead.

The second experiment compared two lists of ranked results produced by the same ranking algorithm. The first list was the top-10 results when the IDF value was computed using the entire collection. The second set was the top-10 results produced when the IDF value was computed using only the search context elements (which is ideally what the user would like to see). The difference between these two lists is thus a measure of how much context-sensitive ranking is likely to change what the user sees in the top ranked results.

Table 2 shows the scaled Spearman Footrule Distance [2] as a measure of the difference for the XXL, XSEarch and XIRQL ranking methods for some search contexts and keywords where there are high variations in the IDF values. The scaled Footrule distance measure produces values in the range [0, 1], where 0 means identical results and 1 means that the ranked lists do not have common elements. As shown, some query keywords such as "process speech" have almost no common results in the two lists, while others have more common results.

In summary, the experiments show that context-sensitive ranking

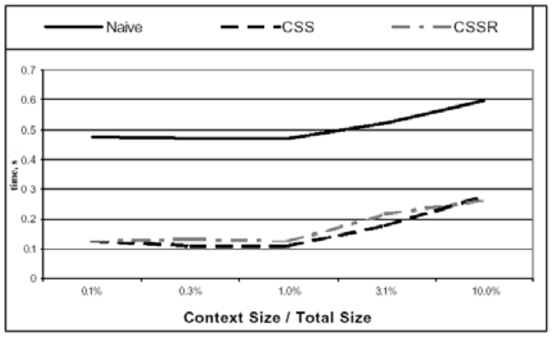


Figure 5 Context Size vs. Query Time

can significantly influence ranked results with negligible performance overhead.

5. RELATED WORK

There has been a lot of recent work on keyword search over XML. Some of these, like [3], [10], [24] and the SGML indexing techniques in [13], do not consider the issue of ranking. Various scoring methods for semi-structured document collections have been proposed [1][4][5][8][16][17][21][23]. However, unlike the present paper, none of the above addresses the issue of context-sensitive ranking and its tight integration with context-sensitive search.

Grabs and Schek [7] propose a context-sensitive scoring method for the INEX collection. Their definition of context uses predefined categories (element nodes of the same type). Our work is complementary to the above work in that we do not propose a specific scoring method but develop a general framework whereby multiple scoring methods, including that in [7], can be incorporated. Our focus is thus on developing the underlying system architecture, efficient inverted lists and query evaluation using these inverted lists, which are not considered in [7]. We also support a more flexible search context specification based on XPath without restrictions on the search context. Halverson et al. [9] and Kaushik et al. [15] discuss inverted lists with B+-trees in the context of structural joins, but do not consider context-sensitive ranking. Jacobson et al. [12] propose techniques for context-sensitive search over LDAP repositories but they focus on efficiently evaluating the context expression and not on evaluating keyword-search queries or ranking results.

6. CONCLUSIONS

We have defined the problem of context-sensitive ranking and studied its integration with context-sensitive search. We have proposed a general ranking framework whereby a large class of existing TF-IDF based ranking algorithms can be directly adapted for context-sensitive ranking. We have also proposed efficient index structures and query evaluation strategies for evaluating and ranking context-sensitive queries. In the future, we plan a user evaluation study to quantify the retrieval benefits of context-sensitive ranking.

Table 1 Effect of the Context-Sensitive Ranking

Query	Context	XXL	XSEarch	XIRQL
process speech	shakesp	0.81	0.81	1
join complexity	inex/tk	1	0.81	0.02
Sigmod opportunities	inex/tk	0.59	0.19	0.8
itemsets statistics	inex/tk	1	0.51	0.13
relational decomposition	inex/tk	1	0.32	0.16

7. REFERENCES

[1] Cohen, S., J. Mamou, Y. Kanza, Y. Sagiv. XSEARCH: A Semantic Search Engine for XML. VLDB 2003.

[2] Diaconis, P., R. L. Graham. "Spearman's Footrule as a Measure of Disarray". *J. of the Royal Society of Statistics*, series B39 (1977).

[3] Florescu, D., Kossmann, D., Manolescu, I. Integrating Keyword Search into XML Query Processing. WWW 2000.

[4] Fuhr, N., T. Rölleke. A Probabilistic Relational Algebra for the Integration of Information Retrieval and Database Systems. TOIS, 15 (1), 1997.

[5] Fuhr, N., Großjohann, K. XIRQL: A Language for Information Retrieval in XML Documents. SIGIR 2001.

[6] Goldman R., N. Shivakumar, S. Venkatasubramanian, H. Garcia-Molina. Proximity Search in Databases. VLDB 1998.

[7] Grabs, T., H.-J. Schek. "PowerDB-XML: A Platform for Data-Centric and Document-Centric XML Processing". XSym 2003, Berlin, Germany.

[8] Guo, L, F. Shao, C. Botev, J. Shanmugasundaram. XRANK: Ranked Keyword Search over XML Documents. SIGMOD 2003.

[9] Halverson, A. et al. Mixed mode XML query processing. In VLDB, 2003.

[10] Hristidis, V., Y. Papakonstantinou, A. Balmin. Keyword Proximity Search on XML Graphs. ICDE 2003

[11] INEX 2003. http://inex.is.informatik.uni-duisburg.de:2003/

[12] Jacobson, G., B. Krishnamurthy, D. Srivastava, D. Suciu. Focusing Search in Hierarchical Structures with Directory Sets. CIKM 1998.

[13] Lee, Y., S.-J. Yoo, K. Yoon, P. B. Berra. Index Structures for Structured Documents. Digital Libraries Conf., 1996.

[14] Luk, R., et al. A Survey of Search Engines for XML Documents. SIGIR Workshop on XML and IR, 2000.

[15] Kaushik, R., R. Krishnamurthy, J. Naughton, R. Ramakrishnan. On the Integration of Structure Indexes and Inverted Lists. SIGMOD 2004.

[16] Myaeng, S., D.H. Jang, M.S. Kim, and Z.C. Zhoo. A Flexible Model for Retrieval of SGML Documents. SIGIR 1998.

[17] Navarro, G., Baeza-Yates, R. Proximal Nodes: A Model to Query Document Database by Content and Structure. Information Systems, 1997.

[18] Salton, G. Automatic Text Processing: The Transformation, Analysis and Retrieval of Information by Computer. Addison Wesley, 1989.

[19] SIGMOD Record in XML. http://www.acm.org/sigmod/record/xml/XMLSigmodRecordNov2002.zip

[20] Shakespeare's Plays in XML. http://www.oasis-open.org/cover/bosakShakespeare200.html

[21] Theobald, A., Weikum, G. The Index-Based XXL Search Engine for Querying XML Data with Relevance Ranking. EDBT 2002.

[22] Yoshikawa, M., T. Amagasa, T. Shimura, S. Uemura. XRel: A Path-Based Approach to Storage and Retrieval of XML Documents Using Relational Databases. ACM TOIT 1(1), 2001.

[23] Yu, C., Qi, H., Jagadish, H. Integration of IR into an XML Database. INEX Workshop, 2002.

[24] C. Zhang, J. Naugthon, D. DeWitt, Q. Luo, G. Lohman. "On Supporting Containment Queries in Relational Database Management Systems". SIGMOD 2001

Constructing Maintainable Semantic Mappings in XQuery

Gang Qian
Dept. of Computer Science and Engineering
Southeast University, Nanjing 210096, China
qiangang@seu.edu.cn

Yisheng Dong
Dept. of Computer Science and Engineering
Southeast University, Nanjing 210096, China
ysdong@seu.edu.cn

ABSTRACT

Semantic mapping is one of the important components underlying the data sharing systems. As is known, constructing and maintaining such mappings both are necessary yet extremely hard processes. While many current works focus on seeking automatic techniques to solve such problems, the mapping itself is still left as an undecorated expression, and in practice it is still inevitable for the user to directly deal with such troublesome expressions. In this paper we address such problems by proposing a flexible and maintainable mapping model, where atomic mapping and combination operators are the main components. Conceptually, to construct global mapping for the whole target schema, we first construct the atomic mappings for each single target schema element, and then combine them using the operators. We represent such combined mappings as mapping trees, which can be incrementally constructed, and can be locally maintained. Also, we outline the main issues in combining our work with the current automatic techniques, and analyze the maintainability of the mapping tree. Though our discussion is applicable to other models, this paper limits the attention to the XML model and the XQuery language.

1. INTRODUCTION

Semantic mapping is one of the important components underlying the data sharing (e.g., data integration and data exchange) systems. For example, the mappings may be used to translate the user query over the target (mediated) schema into queries over the source schemas (e.g., [8]), or translate the data resided at different sources into the target database. To enable data sharing, the user has to first construct the semantic mappings between the target and the source schemas. Also, as the application requirements or the schemas change, the user has to maintain and modify the early constructed mappings.

As is known, constructing and maintaining such mappings both are extremely labor-intensive and error-prone processes. Trying to provide automated support, much recent literature has extensively studied the techniques like *schema matching*, *mapping discovery* and *mapping adaptation*. Given a pair of schemas, the technique of schema matching focuses on discovering semantic correspondences (*matches*) between schema elements (e.g., [11, 6, 18, 5]). Taking these matches as input, the tools like `Clio` [3] then are employed to further discover and generate the candidate semantic mappings between the schemas, e.g., in the form of a *naive* SQL or XQuery expression [10, 13]. When the schemas evolve, the technology of mapping adaptation is responsible for adjusting the mappings constructed originally and keeping them as consistent as possible [19].

Eighth International Workshop on the Web and Databases (WebDB 2005), June 16-17, 2005, Baltimore, Maryland

Despite this progress, however, it is still inevitable for the user to directly construct and maintain the mappings. In practice there are many factors that may require to modifying and maintaining the mappings. For example, mapping construction is usually a process of repeated refinement. In most case, only semantically valid and partial mappings the automated techniques can discover. To obtain the *desired* one, the user may need to further refine the discovered mappings, or completely reconstructed it in the cases beyond the intelligence of the automated techniques. A detailed motivation appears in Section 2.

As the automated techniques could not completely solve these mapping problems, we are inspired to explore other *complementary* ways to alleviate the burden on the user. Currently, schema mappings are mainly represented as (query) expressions, which are troublesome for the user to deal with. In dynamic environment like the Web, schemas and application requirements may change frequently. We believe that a maintainable mapping representation would be more suitable than the undecorated expression. Further, as large, complicated schemas become prevalent on the Web, it may be more feasible to incrementally construct the whole schema mappings, e.g., starting with simple mappings, and then gluing them to formulate the globe ones.

In terms of the above observations, in this paper we propose a flexible and maintainable mapping model, where *atomic mapping* and *combination operators* are the main components. Specifically, we limit our attention to the XML model and the mappings expressed in XQuery, though our discussion is also applicable to others. In our model, two atomic mappings (say M_1 and M_2) may be combined using the *Nest*, the *Join* or the *Merge* operator, and the resulting mapping is called *combined mapping* (say M_3). We say that the combined mapping M_3 is *maintainable*, which means that it can be combined again with others, possibly using another combination operator, and it is also possible to reset the operator of connecting M_1 and M_2, or recover M_1 and M_2 from M_3.

With our model, to construct global mapping for the whole target schema, we begin to construct the atomic mappings for each single target schema element, and then incrementally combine them using the operators. Such *flexibility* in mapping construction makes our model adapt well to complicated applications. We represent the combined mapping as a mapping tree. To maintain and modify the schema mapping, we only need to adjust the corresponding nodes of the mapping tree, while other

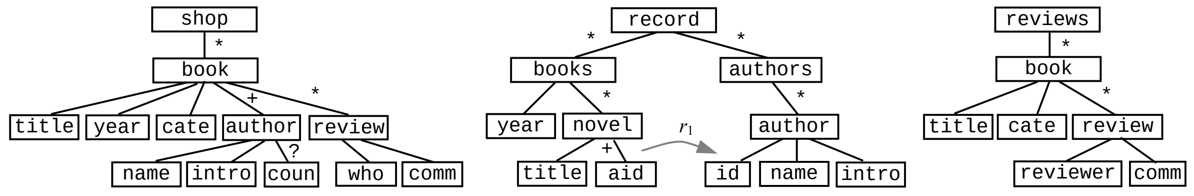


Figure 1. The target schema T (left), source schema S1 (middle) and S2 (right)

nodes and their relationships are reused. Note that our approach would not replace, but rather complement existing techniques to assist the user to manage the schema mappings. We analyze the maintainability of the mapping tree, and outline the main issues in combining our work with the current automatic techniques.

2. A MOTIVATING EXAMPLE

We start with a common example of sharing book information and illustrate the practical requirement for a flexible, maintainable mapping representation model. Suppose there is an online shop that wants to collect data from other sources. Figure 1 shows the schemas T of the shop and S1 and S2 of its two sources, which serve as our running example for discussing schema-to-schema mappings throughout the paper.

We model nested schemas as tree structures, where each tree edge denotes the structural constraint, the non-tree edge like r_1 of S1 indicates the referential constraint, and the multiplicity label associated with the tree edge represents the cardinality constraint. In the source (of) S1, books are grouped by year, and then categorized by the styles such as novel. The source S2 provides reviews of books. We suppose that the book is identified by its title. Note that it is uncertain that every novel instance of S1 must have corresponding reviews in S2.

Using XQuery, we give an example of mapping expression as follows, which relates the source schemas S1 and S2 and the target schema T, and indicates the correspondences between schema elements, e.g., the novel of S1 and the book of T.

```
<shop>
for $bs in doc("S1")//books, $n in $bs//novel
return <book>
    {$n/title, $bs/year}
    <cate>{"novel"}</cate>{
    for $a in doc("S1")//author
    where $n/aid=$a/id
    return <author>
        {$a/name, $a/intro}
    </author>}
    … … … … … …
    </book>
</shop>
```

Figure 2. An example mapping expression

In practice, there are many cases where the mappings have to be modified and maintained. First, constructing the mappings may well be a repeated refinement process, especially for complicated applications. For example, to construct the above mapping, the user at the beginning might have related the book elements of S2 and T. In another case, if the referential constraint r_1 of S1 did not hold, then the above mapping may need to be refined to define the target author instances by the aid of novel, while for those authors not stored in S1, the related target attributes like name may be filled with null values or Skolem functions. Second, when the application requirements or the schemas change, the mappings need to be maintained accordingly. For example, for some reason the shop may want to constraint the schema element review by "+". Again, the shop may want to alter to share reviews from other more economy sources.

The undecorated expression is troublesome for the user to deal with. In contrast, we propose to represent the schema mapping as a combined formulation, where *atomic mappings* and *combination operators* are the main constituents. The atomic mapping defines the local view of a single schema element. Using the combination operator, two atomic mappings can be connected, and the result is a *combined mapping*, which can be further combined with other combined or atomic mappings.

Example 2.1 For the single target elements book and title, we respectively construct the atomic mappings as follows.

$M_{\mathtt{book()}}$: **for** $n1 **in** doc("S1")//novel
return <book></book>

$M_{\mathtt{title}}$: **for** $n2 **in** doc("S1")//novel, $t1 **in** $n2/title
return $t1

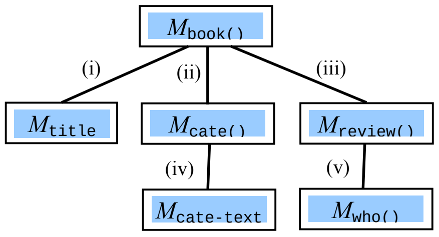
Using the *Nest* operator (see Section 3), we combine $M_{\mathtt{book()}}$ with $M_{\mathtt{title}}$ and obtain the mapping $M_{\mathtt{book(title)}}$ as follows.

```
for $n1 in doc("S1")//novel
return <book>{
  for $n2 in doc("S1")//novel, $t1 in $n2/title
  where $n1=$n2
  return $t1}
  </book>
```

The above combined mapping represents a more significant view, where the initially separate title instances returned by $M_{\mathtt{title}}$, now is structurally nested within the paper instances. Continuing to apply the operators in the same way, other instances also can be nested within the returned paper instances. □

A mapping should be semantically valid, i.e., conforming to the constraints contained in the target schema. Intuitively, ignoring its contexts (i.e., the associated constraints), we think of the single schema element as the simplest form of schema. Then the atomic mapping represents the semantic relationship between the source schemas and the simplest target schema. As the atomic mappings are combined, the separated schema elements are stitched up, and the ignored contexts are recovered. Thus, to construct the global mappings for the whole target schema, we begin to construct the atomic mappings for each single target schema element, and then combine them together by applying the operators.

The combined mapping possesses a tree structure, where the node contains a cluster of atomic mappings, and the edge denotes the applied operator. Figure 3.a shows an example of the mapping tree. We can insert other atomic mappings into the mapping tree



(i) (*Nest*, `$n1=$n2`)   (iii) (*Nest*, `$n1/title=$b1/title`)

Figure 3. The global mapping between the target and the source schemas is represented as a mapping tree.

and construct the global mapping equaling to the one shown in Figure 2. Besides such flexibility, compared to the naive mapping expression, the combined mapping is also maintainable. For example, we can update the operator type to reflect the change of the cardinality constraint. Also, we can modify locally the atomic mappings contained in the mapping tree, while other parts are remained and reused.

3. MAPPING COMBINATION

Atomic mapping. We consider the `FLWR` [17] expression and define atomic mapping as a restricted query formulation. In contrast with the usual XQuery expression, which may contain arbitrary nested queries, an atomic mapping consists of only one `FOR`, one `RETURN` and one optional `WHERE` clauses, and, specifically, has the following general form.

```
for $v1 in SP1(), $v2 in SP2($v1), ……, $vn in SPn($vn-1)
where φ
return () | constant | SPn+1($vj) | <tag></tag>
```

Here, `SP` is a simple path expression with no branching predicates, and $\texttt{SP}_1\texttt{()}$ indicates that $\texttt{SP}_1$ must start at a schema root, while $\texttt{SP}_k\texttt{(\$v}_{k-1}\texttt{)}$ denotes that $\texttt{SP}_k$ is relative to the variable $\texttt{\$v}_{k-1}$. The filter φ is a conditional expression w.r.t. the variables of the atomic mapping. The `RETURN` clause indicates that the atomic mapping may be *empty*, *constant*, *copy*, or *constructor* type.

In Example 2.1, the atomic mapping $M_{\texttt{title}}$ is copy type, and $M_{\texttt{paper()}}$ is constructor type, while the following atomic mapping $M_{\texttt{cate-text}}$ is constant type.

```
Mcate-text: for $n4 in doc(“S1”)//novel
            return “novel”
Mreview(): for $b1 in doc(“S2”)//book, $r1 in $b1/review
           return <review></review>
```

We refer to $\texttt{\$v}_k$ ($1 \le k \le n$) as the *F-variable* of the atomic mapping, and $\texttt{\$v}_n$ as the *primary F-variable* (*PFV*) and others as the *prefix F-variable*. Besides binding tuples of instances, the F-variables may also be used to filter the binding tuples, copy the source fragments, or connect with other mappings. In the latter case, as will be seen in Section 4, the prefix F-variables such as `$n2` in $M_{\texttt{title}}$ may be inserted dynamically. In other words, the user only needs to construct $M_{\texttt{title}}$ as follows.

```
for $t in doc(“S1”)//novel/title return $t
```

In combining mappings, the F-variables sharing the same name will be renamed.

Combination operator. In the following, M_1 and M_2 denote atomic mappings with the general forms. We define a few basic operators to combine M_1 and M_2. The resulting mapping is called *combined mapping*, denoted by M_3. Different from mapping (or query) *composition* [7], where one query can be answered directly using the results of another query, *combining* two mappings is a “parallel” connection, which includes joining the bound sources instances and combining the constructed target instances.

The bound sources are related by combination path, which is a comparison expression w.r.t. the F-variables of M_1 and M_2, e.g., `$n1=$n2` in Example 2.1. In Section 4 we will discuss how to discover such combination paths in combining the mappings.

The operators are responsible for structurally relating the target instances returned by M_1 and M_2. At the same time, we respectively use the *Nest*, *Join* and *Merge* operator types to capture the cardinality constraints contained in the target schema. Note that [13] shows the techniques of generating mappings in the case of referential constraints, which also apply our context and are omitted here. We use the following example to explain the intuition behind the combination operators, while a bit more rigorous formalism will be given when defining the combined mappings.

Example 3.1 As another example, we use the *Nest* operator to combine $M_{\texttt{book()}}$ with $M_{\texttt{review()}}$, and get a combined mapping as follows.

```
for $n1 in doc(“S1”)//novel
return <book>{
  for $b1 in doc(“S2”)//book, $r1 in $b1/review
  where $n1/title=$b1/title
  return <review></review>}
  </book>
```

The above mapping indicates that for each `novel`, there will be a new `book` instance returned, no matter whether there are corresponding reviews in the source `S2`. In other words, the *Nest* operator captures the outer-join relationship between the binding tuples of the combined atomic mappings. With the constraints in the schemas of Figure 1, it is valid to apply the *Nest* operator to combine $M_{\texttt{book()}}$ with $M_{\texttt{review()}}$. However, in the shop schema, when the cardinality constraint “*” of `review` is replaced with “+”, those books with no reviews should be filtered out. The *Nest* operator cannot satisfy such requirement. Instead, we use the *Join* operator to combine $M_{\texttt{book()}}$ with $M_{\texttt{review()}}$, and get a combined mapping as follows.

```
for $n1 in doc(“S1”)//novel
let $v:= for $b1 in doc(“S2”)//book, $r1 in $b1/review
        where $n1/title=$b1/title
        return <review></review>
where count($v)>0
return <book>{$v}</book>                              □
```

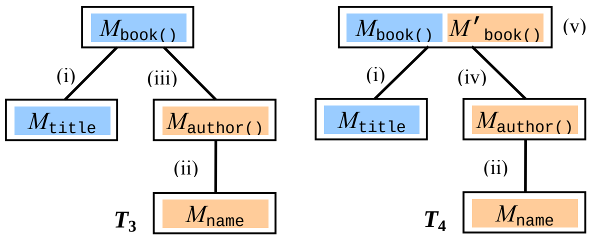
Let ψ denote the combination path. For both the *Nest* and the *Join* operators, we constrain M_1 to be constructor type. Figure 4 respectively shows the resulting combined mapping M_3, which specifies that the target instances returned by M_2 would be nested within those returned by M_1. Syntactically, for the *Nest* operator, the resulting mapping M_3 is obtained by nesting M_1 within the `RETURN` clause of M_2, while for the *Join* operator, M_3 is obtained by introducing a new `LET`-variable `$v` to bind the sequences returned by M_2, and a condition `count($v)`>0 to filter out those unsatisfied binding tuples.

```
for $v_{1,1} in SP_{1,1}(), ……, $v_{1,n} in SP_{1,n}($v_{1,n-1})
where φ_1
return <tag>
   for $v_{2,1} in SP_{2,1}(), ……, $v_{2,m} in SP_{2,m}($v_{2,m-1})
   where φ_2 and ψ
   return exp </tag>
```

```
for $v_{1,1} in SP_{1,1}(), ……, $v_{1,n} in SP_{1,n}($v_{1,n-1})
let $v:= for $v_{2,1} in SP_{2,1}(), ……, $v_{2,m} in SP_{2,m}($v_{2,m-1})
          where φ_2 and ψ
          return exp
where count($v)>0 and φ_1
return <tag>{$v}</tag>
```

```
for $v_{1,1} in SP_{1,1}(), ……, $v_{1,n} in SP_{1,n}($v_{1,n-1})
for $v_{2,1} in SP_{2,1}(), ……, $v_{2,m} in SP_{2,m}($v_{2,m-1})
where φ_1 and φ_2 and ψ
return <tag></tag>
```

Figure 4. The *Nest, Join* and *Merge* operators



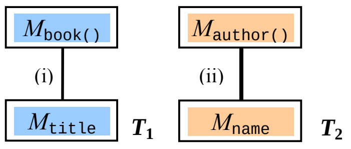
$M_{author()}$:
```
for $a1 in doc("S1")//author return <author></author>
```
$M'_{book()}$:
```
for $a3 in doc("S1")//author return <book></book>
```
(i) (*Nest*, `$n1=$n2`) (ii) (*Nest*, `$a1=$a2`) (iv) (*Nest*, `$a3=$a1`)
(iii) (*Nest*, `$n1/aid=$a1/id`) (v) (*Merge*, `$n1/aid=$a3/id`)

Figure 5. Constructing the general mapping tree

In addition, the application may need to express the "product" relationship between the binding tuples of M_1 and M_2. When the target schema is a default XML view over the relational database [15], for example, the flattened instances may be required to be returned. We use the *Merge* operator to satisfy such demands. In this case, if M_1 is constant or copy type, then M_2 must be empty type; if M_1 is constructor type, then M_2 must also be constructor type and with the same `<tag>`. For the latter Figure 4 shows the general form of the resulting combined mapping M_3, where the returned instances are merged.

Combined mapping. In defining the above operators, we present the combined mapping M_3 as an equivalent expression. Now, to define the general combined mappings, we first model M_3 as a *mapping tree*. Specifically, the atomic mapping is considered as a node. If M_1 and M_2 are combined using the *Nest* or the *Join* operator, then M_2 is a child node of M_1, and the edge is labeled with (*Op*, ψ). If they are combined using the *Merge* operator, then the nodes are united into one, which contains both M_1 and M_2, and is associated with ψ.

For example, the following mapping tree T_1 corresponds to the combined mapping $M_{book(title)}$ (see Example 2.1), while the tree T_2 means a combined mapping obtained by combining $M_{author()}$ with $M_{name()}$ using the *Nest* operator.



(i) (*Nest*, `$n1=$n2`)
(ii) (*Nest*, `$a1=$a2`)

Being a query, the atomic mapping returns a forest of data (DOM) tree. Specifically, for each binding tuple t, if the filter φ holds, then the atomic mapping will construct a data tree d. In this case we also say that $t \to d$ holds. Corresponding to the types, the data tree d may be an empty node, a text node, a copied subtree, or an element node. The *Nest* operator applied between $M_{book()}$ and M_{title} indicates that, for each binding tuple t ($t \to d$ holds) of $M_{book()}$, if there are n binding tuples t_n ($t_n \to d_n$ holds) of M_{title} satisfying the combination path (i.e., `$n1=$n2`), then the combined mapping tree T_1 will return the data tree d with d_n ($n \geq 0$) nested within its root node. In our running example, n=1.

Further, we use the *Nest* operator to combine the $M_{book()}$ of T_1 with the $M_{author()}$ of T_2, which means inserting T_2 into the root of T_1. Figure 5 shows the obtained mapping tree T_3, where the new edge "(iii)" is associated with the applied operator and combination path. In another case, we first combine $M'_{book()}$ (see Figure 5) with the $M_{author()}$ contained in T_2, and obtain a mapping tree (say T'_4). Then we use the *Merge* operator to combine the $M_{book()}$ contained in T_1 with the $M'_{book()}$ contained in T'_4, which means merging the root nodes of T_1 and T'_4. The obtained mapping tree T_4 is shown in Figure 5.

Let t_1 and t_2 respectively denote the binding tuples of $M_{book()}$ and $M'_{book()}$, and let d_1 and d_2 respectively denote the data trees returned by T_1 and T'_4. The *Merge* operator applied above indicates that, for each pair of the binding tuples (t_1, t_2), where both $t_1 \to d_1$ and $t_2 \to d_2$ hold, if the combination path holds, then the mapping tree T_4 will return a data tree which merges the root nodes of d_1 and d_2. Intuitively, the mapping tree T_4 defines flattening `author` instances.

As can be seen, in a general mapping tree, each node contains atomic mappings, which are combined using the *Merge* operator, and each edge corresponds to the *Nest* or the *Join* operator, which relates the atomic mappings contained in the parent and in the child nodes. Note that there are no the possibilities where both the *Nest* and the *Join* operators are simultaneously applied in the same edge, since the atomic mappings contained in the same node contribute to the same target instances. But now the combination path may be a conjunctive formulation.

Let a node in the mapping tree contain i atomic mappings. We can define the semantics of the general mapping tree in terms of the tuple (t_1, t_2, …, t_i), where each t_i denotes the binding tuple of the corresponding atomic mapping. We omit the details here. In terms of the rules of formulating the combined mapping M_3 as shown in Figure 4, we write the equivalent mapping expression for the mapping tree T_3 as follows.

```
for $n1 in doc("S1")//novel
return <book>{
  for $n2 in doc("S1")//novel, $t1 in $n2/title
  where $n1=$n2 return $t1}{
  for $a1 in doc("S1")//author
  where $n1/aid=$a1/id return <author>… … …</author>}
  </book>
```

Using the normalization rules such as shown in [8], the above mapping may be minimized into the expression fragment of the one in Figure 2. The ways to construct the mapping tree are multiple. Alternatively, for example, to obtain the mapping tree T_4, we can first merge T_1 with $M'_{\texttt{book()}}$, and then insert T_2. We believe that such flexibility would be popular in constructing the global mapping, especially for the complicated applications. Note that the sibling nodes are order sensitive in the mapping tree.

4. AUTOMATED SUPPORT

Besides the flexibility, the combined mapping is also maintainable. In this section we further combine our work with the current automated techniques. Specifically, we show how to generate the atomic mappings and discover the combination paths. Also, we analyze the maintainability of the combined mappings.

Given the target and the source schemas, the atomic mappings are first generated in terms of the matches produced by a tool of schema matching (e.g., `LSD` [6]). Interestingly, due to the maintainability of the combined mapping, our model does not require that all the produced matches should be desired. Next, keeping the target schema in mind, the user incrementally combine the atomic mappings using the operators, i.e., inserting and merging the mapping trees. In this process, a tool may be used to suggest the candidate combination paths.

Maintainability. As motivated in Section 2, the requirements of modifying and maintaining the mappings may result from the way of incrementally constructing the mappings, the refinement of the mappings, or the evolution of the schemas. With our mapping model, they are all reduced to maintaining the mapping trees, e.g., inserting and merging subtrees. In the mapping trees, the atomic mapping may be related through the combination paths with other atomic mappings contained in the same, parent, or child nodes. As the modifications occur in the mapping tree, these combination paths would need to be discovered or adjusted, while other parts of the mapping trees would be remained and reused.

Modifying the combination operators (*Nest* and *Join*) will not affect the combination paths that relate the atomic mappings. For example, consider the Example 3.1 and the mapping tree in Figure 3, if in the target schema the cardinality constraint “*” of `review` is replaced with “+”, then we only need to modify the operator type in the edge “(iii)” from *Nest* to *Join*.

On the other hand, for the following modifications, we find out those pairs of atomic mappings (M_1, M_2) between which the combination paths would need to be discovered or adjusted. In the following, S represents the set of (M_1, M_2), Tr corresponds to the mapping trees and r is its root node, and $atoms(n)$ denotes the set of atomic mappings contained in the node n of the mapping tree. We also use p and c_i to respectively denote the parent and the children nodes of the node n.

Inserting Tr_1 *into the node* n *of* Tr_2.

$S \leftarrow (M_1, M_2)$, where $M_1 \in atoms(r_1)$ and $M_2 \in atoms(n)$.

Merging Tr_1 *into the node* n *of* Tr_2.

$S \leftarrow (M_1, M_2)$, where $M_1 \in atoms(r_1)$ and $M_2 \in atoms(n) \cup atoms(p) \cup atoms(c_i)$.

Updating the atomic mapping M contained in the node n.

$S \leftarrow (M_1, M_2)$, where $M_1 \in \{M\}$ and $M_2 \in atoms(n) \cup atoms(p) \cup atoms(c_i) - \{M\}$.

Removing the atomic mapping M from the node n.

$S \leftarrow (M_1, M_2)$, where $M_1, M_2 \in atoms(n) \cup atoms(p) \cup atoms(c_i) - \{M\}$ and $M_1 \neq M_2$.

Consider the mapping tree T_4 shown in Figure 5. The atomic mappings $M'_{\texttt{book()}}$ and $M_{\texttt{author()}}$ contained in the tree are related by the combination path `$a3=$a1`. The modification of removing $M'_{\texttt{book()}}$ from the root of T_4 would affect the atomic mapping pairs $\{(M_{\texttt{book()}}, M_{\texttt{title}}), (M_{\texttt{book()}}, M_{\texttt{author()}})\}$, which are respectively taken to discover the candidate combination paths (discussed in a moment). Guided by these candidate paths, the user then may adjust the above path to `$n1/aid=$a1/id`.

Note that such adjustment is not additionally introduced by our mapping model. In contrast, such maintainability inherent in our model provides the opportunities for automating the process of mapping maintenance, be it caused by schema evolution, mapping refinement, or other factors. In our going work, we are developing methods to assign priorities to the discovered combination paths, and to heuristically reduce the amount of the potentially affected atomic mapping pairs. Additionally, our first experiment in the book domain shows that there are averaged 1.2 atomic mappings contained in each node of the mapping trees.

Schema matching. Schema matching [6, 18, 5] produces a set of semantic correspondences (matches) between the elements of the schemas, from which atomic mappings can be generated. For example, a 1-1 match would specify that the element `name` in `S2` match `who` in the shop schema `T`. Then, an atomic mapping $M_{\texttt{who}}$ could be generated via specifying its PFV (see section 3) by the element `name`. As to complex type (e.g., 1-n) matches, several atomic mappings may be generated, which then are combined using the operators. In practical mapping construction, it is often the case where the matches initially used to obtain the mappings are not the desired. Fortunately, as shown above, our model is able to make it local to update the corresponding atomic mappings.

Discovering combination path. Combination path ψ is used to relate the bound source instances of the atomic mappings, and can be heuristically discovered in terms of the semantic relationships between the source schema elements. As presented in [10, 13, 19], such relationships are captured by the *structural*, *user* and *logical associations*, which respectively describe a set of associated schema elements.

Consider the source schema `S1`. For example, the elements such as `novel` and `title` are in a structural association, while the elements such as `novel`, `title`, `author` and `name` are in a logical association. To relate the novel with the book reviews, the user may explicitly relate the `title` elements. Then the elements such as `novel` and `title` of `S1`, and `book` and `title` of S2 are in a user association.

Let $e1$ and $e2$ respectively denote the source schema elements specifying the PFVs of the atomic mappings M_1 and M_2. If the elements $e1$ and $e2$ are in a structural association, then ψ may be formulated in terms of their common path. Consider to combine the atomic mappings $M_{\texttt{book()}}$ and $M_{\texttt{title}}$. Their PFVs (i.e., `$n1` and `$t1`, see Example 2.1) are respectively specified by the elements `novel` and `title` of `S1`, which are in a structural association. In terms of the common element, i.e., `novel`, the path `$n1=$n2` is generated, where, as a prefix F-variable, `$n2` may be dynamically inserted into $M_{\texttt{title}}$, if it does not exist.

If $e1$ and $e2$ are in a structural but in a user association, then ψ may be formulated with the path assigned by the user. Lastly, if they are neither in a structural nor in a user, but in a logical association, then ψ may be formulated in terms of the referential path between the schema elements $e1$ and $e2$. For example, the combination path, `$n1/aid=$a1/id`, of relating $M_{\texttt{book()}}$ and $M_{\texttt{author()}}$ is generated in terms of the logical relationship between the elements `novel` and `author`. In a similar way, the prefix F-variables can also be introduced dynamically, if they are not defined in constructing the atomic mappings.

5. RELATED WORK

Schema mappings are extensively used in many modern applications such as the data integration and data exchange systems. To alleviate the burden on the user for constructing and maintaining such semantic mapping, many efforts have been made to pursue maximum automatic support, which can be classified into works on schema matching and mapping discovery. The former focuses on discovering semantic correspondences between the elements of a pair of schemas (e.g., [11, 6, 18, 5], see also [14] for a recent survey). Among these, [18, 5] discussed how to obtain complex type matches, based on the domain ontology and the multi-matcher mechanism. Under the assumption that the desired matches have been given, [10, 13] proposed to further discover candidate schema-to-schema mappings. In their approach, the matches are related using the chase technique [1, 12] to search the semantic relationships between the source or target schema elements. In terms of such semantic relationship, [19] proposed to compute the matches affected by schema evolution, and then re-employ the mapping discovery system to adjust the mappings.

In contrast, we propose a flexible and maintainable model to represent XML mappings. We think that the global mapping can be constructed in a piecemeal fashion, where, to some extent, the partial mappings resemble subgoals in datalog programs. Also, the XML-QL language [4] allow for defining partial mappings. Yet we provide a richer scheme for combining the results of the different partial mappings. In our model, mappings are considered as the first-class citizens that can be operated. Such idea also was used in [2, 9] to deal with the management of meta data. Yet the subjects these works focused on are not the mappings but the matches between the schema elements. Additionally, there have been many GUI-style tools developed to assist the user to construct the mappings (queries) in XQuery, where the queries are formulated in terms of the syntax ingredients such as the `FOR`, `LET`, `WHERE` and `RETURN` blocks [16]. Differently, our mapping model is based on the semantic relationships between mappings to be connected.

6. CONCLUSION

Semantic mappings are key for enabling a variety of data sharing scenarios. This paper described the flexible and maintainable mapping model, where the atomic mapping and the combination operator are the main components. Conceptually, to construct the global mapping for the whole target schema, we first construct the local atomic mappings for the single target schema element, and then combine them using the operators. We represented the combined mappings as the mapping trees. Then the mapping problem is reduced to the problem of constructing and maintaining the mapping trees. We analyzed the maintainability of the mapping tree, and presented how to combine our work with the current automated techniques.

7. ACKNOWLEDGEMENTS

We would like to thank the anonymous referees for their insightful comments.

8. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. Foundations of Databases. Addison-Wesley, 1995.

[2] P. A. Bernstein. Applying Model Management to Classical Meta Data Problems. In *Proc. of CIDR*, 2003.

[3] Clio. http://www.cs.toronto.edu/db/clio/

[4] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A Query Language for XML. In *proc. of WWW*, 1999.

[5] R. Dhamankar, Y. Lee, A. Doan, A. Halevy, and P. Domingos. iMAP: Discovering Complex Semantic Matches between Database Schemas. In *proc. of SIGMOD*, 2004.

[6] A. Doan, P. Domingos, and A. Halevy. Reconciling schemas of disparate data sources: A machine learning approach. In *proc. of SIGMOD*, 2001.

[7] J. Madhavan and A. Halevy. Composing mappings among data sources. In *Proc. of VLDB*, 2003.

[8] I. Manolescu, D. Florescu, and D. Kossman. Answering XML Queries on Heterogeneous Data Sources. In *proc. of VLDB*, 2001.

[9] S. Melnik, E. Rahm, P. A. Bernstein. Rondo: A Programming Platform for Generic Model Management. In *proc. of SIGMOD*, 2003.

[10] R. Miller, L. Haas, and M. Hernández. Schema Mapping as Query Discovery. In *Proc. of VLDB*, 2000.

[11] T. Milo and S. Zohar. Using schema matching to simplify heterogeneous data translation. In *proc. of VLDB*, 1998.

[12] L. Popa and T. Val. An Equational Chase for Path-Conjunctive Queries, Constraints, and Views. In *proc. of ICDT*, 1999.

[13] L. Popa, Y. Velegrakis, R Miller, M. A. Hernandez, and R. Fagin. Translating Web Data. In *Proc. of VLDB*, 2002.

[14] E. Rahm and P.A. Bernstein. A survey of approaches to automatic schema matching. The *VLDB Journal*, 10(4): 334–350, 2001.

[15] J. Shanmugasundaram, J. Kiernan, E. J. Shekita, C. Fan, and J. Funderburk. Querying XML views of relational data. In *proc. of VLDB*, 2001.

[16] Stylus Studio. http://www.stylusstudio.com

[17] XQuery. http://www.w3.org/XML/Query

[18] L. Xu and D Embley. Using domain ontologies to discover direct and indirect matches for schema elements. In *Proc. of the Semantic Integration Workshop at ISWC*, 2003.

[19] Y. Velegrakis, R. J. Miller, and L. Popa. Preserving mapping consistency under schema changes. The *VLDB Journal*, 13(3): 274-293, 2004.

The Framework of an XML Semantic Caching System

Wanhong Xu

Center for Computational Genomics
Department of Electrical Engineering and Computer Science
Case Western Reserve University, Cleveland, OH
Wanhong.Xu@case.edu

ABSTRACT

As a simple XML query language but with enough expressive power, XPath has become very popular. To expedite evaluation of XPath queries, we consider the problem of caching results of popular XPath queries to answer queries.

Existing semantic caching systems can answer queries that have been already cached, but can't combine cached results of multiple XPath queries to answer new queries. In this paper, we describe the architecture of a new semantic caching system, and mainly introduce the novel framework of this system. We show that our framework can represent cached XML data by a set of XPath queries, and the cached data can be used to answer new queries that may not be cached. We also consider incrementally maintaining the cached XML data in our system. The results suggest that our caching system is practical and has better answerability.

1. INTRODUCTION

Recently, more and more data are represented and exchanged as XML documents over Internet. XPath [9], recommended by W3C, is a simple but popular language to navigate XML documents and extract information from them, and to be used as sub-languages of other XML query languages such as XQuery [6].

There has been a lot of work to speedup evaluation of XPath queries, including, index techniques [8], structural join algorithms [2] and minimization of XPath queries [3, 15, 11]. Recently, the problem of answering queries using cached results in XML world has begun to attract more attention since it has been proven that the caching technique can improve performance significantly in traditional client-server databases, distributed databases and Web-based information systems. This problem has been discussed for query optimization [5] and semantic caching [7, 1, 12, 16].

We begin by giving some examples in the semantic caching scenario to describe motivation of studying this problem.

Eighth International Workshop on the Web and Databases (WebDB 2005), June 16-17, 2005, Baltimore, Maryland.

Motivation Examples: Consider the following XML document t stored in an XML server, which partially describes enzyme information of a biological pathway:

```
<Pathway name = "PA1">
  <Reaction name = "RE1">
    <Enzymes>
      <Protein name = "PR1" EC# ="1.0.0.1"/>
      <RNA name = "RN1"/>
    </Enzymes>
  </Reaction>
  <Reaction name = "RE2">
    <RNA name = "RN2">
  </Reaction>
</Pathway>
```

Let's assume that a client issues to the server an XPath query v :

$$//RNA$$

which retrieves all **RNA** elements. Suppose the client caches the result of v. When the client issues another XPath query p :

$$/Reaction/RNA$$

which retrieves **RNA** subelements of all **Reaction** elements. We know that the result of p is included in the cached result, i.e., the result of v. But, we can't compute p's result by using the cached result because we don't know which **RNA** elements in cached result belong to the result of p. In fact, the cached result of v can only be used to answer the same query as v. However, if we can store extra information, for example, the paths from the root of XML document t to each RNA subelement, then the cached result of v can be used to answer p. The cached result with path information, denoted as t', is given as follows:

```
<Pathway>
  <Reaction>
    <Enzymes>
      <RNA name = "RN1"/>
    </Enzymes>
  </Reaction>
  <Reaction>
    <RNA name = "RN2">
  </Reaction>
</Pathway>
```

t' is still an XML document. We can get the same result of evaluating p over t' as that of evaluating p over t. Hence, storing extra information can improve the answerability of cached results.

In this paper, we introduce a framework to represent the cached XML document, discuss how to decide that the cached information is enough to answer a query, and how to incrementally maintain the cached XML document.

2. PRELIMINARIES

2.1 Trees and Tree Patterns

Generally, an XML database consists of a set of XML documents. We model the whole XML database as an unordered rooted node-labelled tree (called **XML tree**) over an infinite alphabet Σ (A virtual root node might be introduced to connect all XML documents if necessary). In this XML tree, each internal node's label corresponds to an XML element or attribute name, and each leaf node's label corresponds to a data value. In addition, we assume that each node has a unique node identifier. An XML tree is shown in Fig. 1 (a) as an instance. We let T_Σ be the set including all possible XML trees over Σ. Formally, we have:

Definition 2.1. *An XML database is a tree $t\langle V_t, E_t, r_t\rangle$ over Σ called* ***XML tree****, where*

- *V_t is the node set and E_t is the edge set;*
- *$r_t \in V_t$ is the root of t;*
- *Each node n in V_t has a label from Σ (denoted as $n.label$) and a unique node identifier (denoted as $n.id$).*

Given an XML tree $t\langle V_t, E_t, r_t\rangle$, the size of t is defined as the cardinality of V_t, and we also say that $t'\langle V_{t'}, E_{t'}, r_{t'}\rangle$ is a **subtree** of t if $V_{t'} \subseteq V_t$ and $E_{t'} = (V_{t'} \times V_{t'}) \cap E_t$. If t' includes the root of t, t' is also called as the **rooted subtree** of t.

In this paper, we discuss a fragment of XPath queries denoted as $XP^{\{/,//,*,[]\}}$, as in [13]. This fragment consists of label tests, child axes(/), descendant axes(//), branches([]) and wildcards(*). It can be recursively represented by the following grammar:

$$xp \rightarrow l|*|xp/xp|xp//xp|xp[xp]$$

where l is a node label from Σ. As said in [13], any XPath query from $XP^{\{/,//,*,[]\}}$ can be trivially represented as a labelled tree(called **tree pattern**) with the same semantics.

Definition 2.2. *A* ***tree pattern*** *p is a tree $\langle V_p, E_p, r_p, o_p\rangle$ over $\Sigma \cup \{$'*'$\}$, where V_p is the node set and E_p is the edge set, and:*

- *Each node n in V_p has a label from $\Sigma \cup \{$'*'$\}$, denoted as $n.label$;*
- *Each edge e in E_p has a label from $\{$'/','//'$\}$, denoted as $e.label$. The edge with label '/' is called child edge, otherwise called descendent edge;*

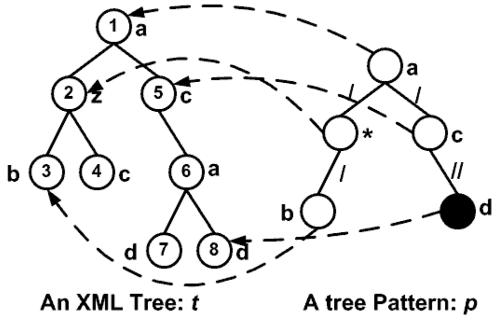


Figure 1: (a)An XML tree t; (b)A pattern p.

- *$r_p, o_p \in V_p$ are the root and output node of p respectively.*

For example, an XPath query $a[*/b]/c//d$ from $XP^{\{/,//,*,[]\}}$ is represented as a tree pattern shown in Fig. 1(b), where the dark node is the output node. Without loss of generality, we refer to tree patterns as patterns in the rest of this paper.

We now define an **embedding** (also called **pattern match**) from a pattern to an XML tree as follows:

Definition 2.3. *Given an XML tree $t\langle V_t, E_t, r_t\rangle$ and a pattern $p\langle V_p, E_p, r_p, o_p\rangle$, an* ***embedding*** *from p to t is a function $e : V_p \rightarrow V_t$, with following properties:*

- *Root preserving: $e(r_p) = r_t$;*
- *Label preserving: $\forall n \in V_p$, if $n.label \neq$ '*', $n.label = e(n).label$;*
- *Structure preserving: $\forall e = (n_1, n_2) \in E_p$, if $e.label =$ '/', $e(n_2)$ is a child of $e(n_1)$ in t; otherwise, $e(n_2)$ is a descendent of $e(n_1)$ in t.*

The embedding maps the output node o_p of p to a node n in t. We say that the node n is the result of this embedding. As an example, dashed lines between Fig. 1(a) and (b) shows an embedding, and its result is the node with id = 8. Actually, there could be more than one embedding from p to t. We define the result of p over t, denoted as $p(t)$, as the union of results of all embeddings, i.e.,

$$\cup_{e \in EB}\{e(o_p)\}$$

where EB is the set including all embeddings from p to t.

For a given XML tree t, we also consider evaluating a set of patterns $S = \{p_1, p_2, ..., p_n\}$ over t. The result, denoted as $S(t)$, is the union of the result of evaluating each p_i in S over t, formally defined as:

$$S(t) = \cup_{p_i \in S} p_i(t)$$

2.2 Containment of Tree Patterns

For any two patterns p_1 and p_2, p_1 is said to be *contained* in p_2(denoted as $p_1 \sqsubseteq p_2$) iff $\forall t \in T_\Sigma$ $p_1(t) \subseteq p_2(t)$. Similarly, we also say that a pattern p is contained in a pattern set S(denoted as $p \sqsubseteq S$) iff $\forall t \in T_\Sigma\ p(t) \subseteq S(t)$, and a pattern set S_1 is contained in a pattern set S_2(denoted as $S_1 \sqsubseteq S_2$) iff $\forall t \in T_\Sigma\ S_1(t) \subseteq S_2(t)$.

It's easy to show that $S_1 \sqsubseteq S_2$ iff $\forall p_i \in S_1\ p_i \sqsubseteq S_2$. However, it's not always true that $p \sqsubseteq S$ implies $\exists p' \in S$ s.t. $p \sqsubseteq p'$.

3. SYSTEM OVERVIEW

In this section, we describe the architecture of our caching system, which is designed to work with XML web services and improve their performance.

Generally, web services are running at web servers and they act as bridges between clients and XML servers. When web services got service requests from clients, they would issue a series of XML queries to XML servers and return the corresponding results to clients.

The caching system runs as an independent application in the web server. Its architecture is shown in Fig. 2. This system intercepts all XML queries issued by web services, and tries to answer them by using local cached XML data instead of submitting them to the XML database server. It consists of several components. Next, We describe them one by one.

The *Cache* stores cached XML data with a semantic scheme. This semantic scheme consists of a set of patterns, and describes current cached XML data. The cached data is organized as an XML tree, which is a rooted subtree of the XML tree exported by the XML database server, more details given in Section 4.1.

When received an XML query, the *Query Manager* decides whether the cached XML tree can *totally answer* this query according to current semantic scheme or not, i.e., we can get the same result of evaluating this query over the cached XML tree as that of running it on the server, further discussed in Section 4.2. The *Index* built on the semantic scheme is used to speed-up the decision. If yes, the *Query Manager* will run this query against the local cached XML tree; otherwise submit it to the server.

The *Replacement Manager* employs replacement strategies (like **LRU**) to clear out less queried or expired data and incorporate new data. Specific details about how to incrementally maintain the cached XML tree will be given in Section 4.3.

4. FORMAL FRAMEWORK

We next introduce our formal framework and discuss how to represent, query and maintain the XML data cached in our caching system.

4.1 Representing the Cached XML Data

We discuss our representation of the cached XML information in this subsection. The main idea is that the cached XML information is represented by a set of patterns S, and this representation must satisfy two requirements: one is that the cached XML data must be a rooted sub-tree of the XML tree exported by the XML database server; the other

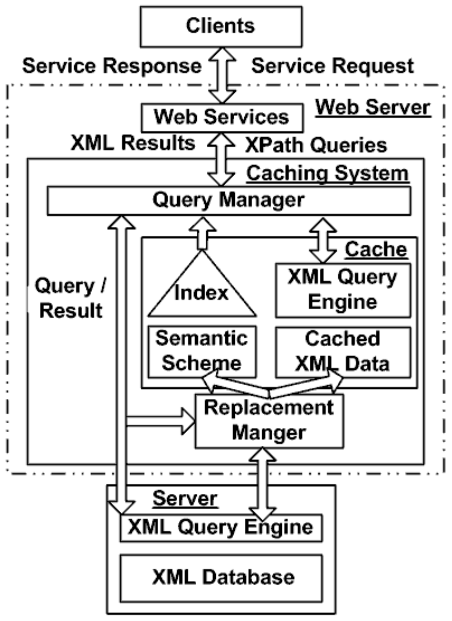


Figure 2: System Architecture

one is that the cached XML tree must ***totally answer*** any pattern in S, i.e, for any pattern $p \in S$, we can get the same result of evaluating this pattern p over the cached XML tree as that of evaluating p over the exported XML tree of the server.

We will describe this representation, satisfying the above two requirements, by defining the XML tree to be cached for a pattern set S. We formalize the representation for one pattern first, and then extend it to a pattern set later.

Before introducing the representation, we define the embedding node set of a pattern p over an exported XML tree t. This set includes all nodes in t mapped from nodes in p by an embedding between p and t. Hence, nodes in this set may be queried or accessed in the evaluation of p over t. Formally, we have:

Definition 4.1. *Given an exported XML tree t and a pattern $p\langle V_p, E_p, r_p, o_p\rangle$, an* ***embedding node set*** *of p over t, denoted as $ENS(p,t)$, is $\cup_{e \in EB}(\cup_{v_i \in V_p}\{e(v_i).id\})$ where EB is the set including all possible embeddings from p to t and e is an embedding.*

Example 4.2. *In Fig. 3, there are only two embeddings from pattern p_1 shown in (a) to an XML tree t shown in (b). The dashed lines represent one embedding, and it maps nodes in p_1 to nodes with id = 1, 2 and 8 respectively in t. The dotted lines represent the other one, and it maps nodes in p_1 to the nodes with id = 1, 3 and 8 respectively in t. So, the embedding node set for p_1 is the union of nodes mapped by these two embeddings, i.e.,* $\{1,2,8\} \cup \{1,2,3\} = \{1,2,3,8\}$.

We next generally define a materialized tree for any node set N over an exported XML tree t.

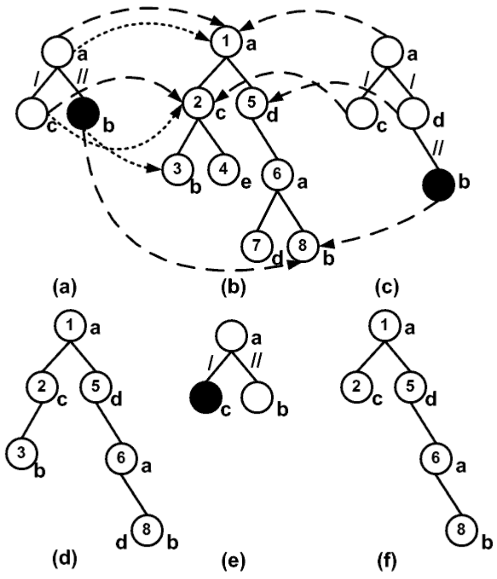


Figure 3: (a)Pattern p_1 (b)The exported XML tree t (c) Pattern p_2 (d) The XML tree t_{p_1} represented by p_1 (e)Pattern p_3 (f)The XML tree t_{p_2} represented by p_2

Definition 4.3. *Given an exported XML tree $t\langle V_t, E_t, r_t\rangle$ and a set of nodes $N \subseteq V_t$, we say that a rooted subtree $t'\langle V_{t'}, E_{t'}, r_{t'}\rangle$ of t is a **materialized tree** for N over t if $N \subseteq V_{t'}$.*

However, there are maybe more than one materialized trees for a set of nodes N over an exported XML tree t. We are only interested in those with the minimum size. We say t' is the **minimal materialized tree**(***MMT***) for N over t if t' is a materialized tree for N over t and any materialized tree t'' for N over t has larger size than t'.

Example 4.4. *In Fig. 3, let's assume that a node set $N = \{1, 2, 8\}$. Both XML trees shown in (d) and (f) respectively are materialized trees for N over the XML tree shown in (b). But, the XML tree shown in (f) is a minimal materialized tree of N, and the XML tree shown in (d) is not.*

Not only minimal materialized trees have the smallest sizes, but also they have very good properties shown in the following lemma.

Lemma 4.5. *Given an exported XML tree t and a set of tree nodes N, t', which is a minimized materialized tree for N over t, has the following properties:*

- *For any materialized tree t'' for N over t, t' is a rooted subtree of t'';*
- *t' is unique, i.e., any other minimized materialized tree is identical to t'.*

We say that the XML tree represented by a pattern p over an exported XML tree t is the minimal materialized tree for the node set $ENS(p, t)$ over t, denoted as t_p.

Next, we generalize the above definitions to a pattern set $S = \{p_1, p_2, ..., p_n\}$. Given an exported XML tree t, we say that the embedding node set for S, denoted as $ENS(S, t)$, is the union of the embedding node set for each $p_i \in S$, i.e.,

$$\cup_{p_i \in S} ENS(p_i, t)$$

and the XML tree represented by this pattern set S, denoted as t_S, is the minimal materialized tree for the node set $ENS(S, t)$ over t.

Our representation satisfies the two requirements listed in the beginning of this subsection. The first one is obvious, and the second one is guaranteed by the following theorem.

Theorem 1. *Given an exported XML tree t and a pattern set S, the XML tree t_S represented by S can totally answer any pattern p_i in S, i.e., $p_i(t) = p_i(t_S)$.*

4.2 Querying the Cached XML Tree

Given an XML tree t exported by the XML database server and a set of tree pattern S, we cache t_S in the web server. We want to use this t_S to answer patterns issued by clients. But, we need to assure that t_S can totally answer them, before evaluating them against t_S. From Theorem 1, we know that t_S can totally answer those patterns included in S. However, t_S can totally answer more patterns not only in S. This is the advantage of our framework. In this subsection, we will discuss the problem how to decide whether t_S can totally answer a pattern p(maybe not in S) or not. The basic idea is to check whether t_p (represented by p) is a rooted subtree of t_S or not. We have the following result:

Lemma 4.6. *Given an exported XML tree t, a pattern p and a pattern set S, t_S can totally answer p if t_p is a rooted subtree of t_S.*

From the above lemma, our problem is reduced to decide whether or not t_p is a rooted subtree of t_S. We can further reduce our problem to decide whether $ENS(p, t) \subseteq ENS(S, t)$ or not in our next result.

Lemma 4.7. *Given an exported XML tree $t\langle V_t, E_t, r_t\rangle$ and two node set $N(\subseteq V_t)$ and $N'(\subseteq V_t)$, the minimal materialized tree for N over t is a rooted subtree of the minimal materialized tree for N' over t if $N \subseteq N'$.*

Hence, t_p is a rooted subtree of t_S if $ENS(p, t) \subseteq ENS(S, t)$. However, there could be some patterns that t_S can totally answer but their embedding node sets are not included in $ENS(S, t)$. The following is an example.

Example 4.8. *In Fig. 3, the exported XML tree t is shown in (b), and two patterns p_1 and p_2 are shown in (a) and (c) separately. We have that $ENS(p_1, t) = \{1, 2, 3, 8\}$*

and $ENS(p_2, t) = \{1, 2, 5, 8\}$. The XML trees t_{p_1} and t_{p_2} represented by p_1 and p_2 are shown in (d) and (f) respectively. Assume we cache t_{p_1} and want to answer p_2. Although $ENS(p_2, t) \not\subseteq ENS(p_1, t)$ due to the node with id = 5 in $ENS(p_2, t)$, t_{p_2} is a rooted subtree of t_{p_1}, i.e., t_{p_1} can totally answer p_2. The reason is that if a rooted subtree of t has the node with id = 8 from t, then it will also have node with id = 5 because the node with id = 5 is an ancestor of node with id = 8 in t. Hence, the node with id = 5 is redundant for $ENS(p_2, t)$ in representing an XML tree.

Furthermore, for a given pattern p, the nodes in $ENS(p, t)$ mapped from the internal nodes of p are redundant to represent an XML tree. The following definitions and lemmas are given to deal with this case.

Definition 4.9. *Given an XML tree t and a tree pattern $p\langle V_p, E_p, r_p, o_p\rangle$, an **embedding leaf node set** $ELNS(p, t)$ is defined as $\cup_{e \in EB}(\cup_{v_i \in V_p^{leaf}}\{e(v_i).id\})$, where EB is a set including all possible embeddings from p to t and $V_p^{leaf} \subseteq V_p$ includes all leaf nodes of p.*

In above example, both embedding leaf node sets for patterns p_1 and p_2 over the exported XML tree t are $\{1, 2, 8\}$.

For an exported XML tree t, we similarly define that the embedding leaf node set for a pattern set S, denoted as $ELNS(S, t)$, is the union of the embedding leaf node set for each $p_i \in S$, i.e., $\cup_{p_i \in S} ELNS(p_i, t)$. We have the following results:

Lemma 4.10. *Let t be an exported XML tree. For a given pattern p and a pattern set S, the following hold:*

- *The minimal materialized tree for $ENS(p, t)$ over t (i.e., t_p) is identical to the minimal materialized tree for $ELNS(p, t)$ over t.*
- *The minimal materialized tree for $ENS(S, t)$ over t (i.e., t_S) is identical to the minimal materialized tree for $ELNS(S, t)$ over t.*

Following Lemma 4.6, 4.7 and 4.10, we easily have that t_p is a rooted subtree of t_S if $ELNS(p, t) \subseteq ELNS(S, t)$.

So far, we reduce the problem of deciding whether t_S can totally answer p or not to the problem that whether $ELNS(p, t) \subseteq ELNS(S, t)$ or not. We next consider how to decide $ELNS(p, t) \subseteq ELNS(S, t)$.

We denote a pattern $p\langle V_p, E_p, r_p, o_p\rangle$ as p_{o_p}, where $o_p \in V_p$ is the output node. We also can choose any node in V_p as the output node of p. For example, the pattern p with a node v_1 instead of o_p as the output node can be denoted as p_{v_1}. For a pattern p, we introduce a tree pattern set(**TPS**) including all patterns by choosing each node in p as the output node, and this pattern set can be formally defined as $\cup_{v_i \in V_p}\{p\_{v_i}\}$ and denoted as TPS_p. If we only choose leaf nodes in p as the output node to build the set, we denote it as TPS_p^{leaf}.

Example 4.11. *In Fig. 3, the pattern p_1 shown in (a) has one leaf node with label 'b' as its output node. This pattern has another leaf node with label 'c'. If we choose it as output node, we can have another pattern p_3 shown in (e). Hence, the pattern set $TPS_{p_1}^{leaf}$ for p_1 is $\{p_1, p_3\}$.*

Our next result shows that the node set $ELNS(p, t)$ is equal to the result of evaluating the pattern set TPS_p^{leaf} over t.

Lemma 4.12. *Given an XML tree t and a pattern p, $ELNS(p, t) = TPS_p^{leaf}(t)$*

For a pattern set S, we similarly define a pattern set TPS_S^{leaf} as $\cup_{p_i \in S} TPS_{p_i}^{leaf}$. From the above lemma, we can easily have $ELNS(S, t) = TPS_S^{leaf}(t)$.

By combining all above lemmas, we finally have the following conclusion:

Theorem 2. *Given an exported XML tree t, a pattern p and a pattern set S, t_S can totally answer p if $TPS_p^{leaf} \sqsubseteq TPS_S^{leaf}$.*

Hence, we reduce the problem deciding whether t_S can totally answer p or not to the containment problem between two pattern sets TPS_p^{leaf} and TPS_S^{leaf}. We will discuss the complexity of this problem and corresponding algorithms in Section 5.

4.3 Incremental Maintenance of the Cached XML Tree

When the XML data represented by some patterns in our caching system are expired or less queried by clients, we need to consider clearing out them and incorporating new data. In this subsection, we discuss how to incrementally maintain the cached XML tree when a pattern is added to or removed from the semantic scheme, which is a pattern set.

Given an XML tree t exported by the server, let's assume we already have t_S for a pattern set S. When we want to add a pattern p to S, our problem is how to get $t_{S \cup \{p\}}$ from t_S. The idea is that we acquire t_p from server first, and then **merge** t_p to t_S.

We can get t_p by pruning all nodes in t whose descendants don't include any node in $ELNS(p, t)$. The **merge** of t_p and t_S works as follows: We assume that t_S has nodes $e_1, ..., e_k$ as its children, and t_p has nodes $n_1, ..., n_l$ as its children. For each node $e_i (1 \leq i \leq k)$, if there is a node $n_j (1 \leq j \leq l)$ that has the same id as e_i, we recursively merge the subtree of t_p, which is rooted at n_j and includes all its descendants, to the subtree of t_S, which is rooted at e_i and all its descendants; otherwise, we put the subtree rooted at n_j under the root t_S as its new subtree. We have the following result about this merged tree:

Lemma 4.13. *Given a pattern p and a pattern set S, the tree merged from t_p and t_S is the minimal materialized tree for $ELNS(S \cup \{p\}, t)$, i.e., $t_{S \cup \{p\}}$.*

When we remove a pattern p from S, our problem is how to get $t_{S\backslash\{p\}}$. We only need to compute $ELNS(S\backslash\{p\}, t)$. Similar to acquire t_p, we can get $t_{S\backslash\{p\}}$ by pruning all nodes in t_S whose descendants don't include any node in $ELNS(S \setminus \{p\}, t)$.

5. COMPLEXITY

Obviously, incrementally maintaining the cached XML tree can be solved polynomially. In this section, we mainly discuss the complexity of deciding containment between two pattern sets.

The complexity of the pattern (not pattern set) containment problem has been well studied [13] and also for its three subclasses, which only use two of the three features: '//', '[]'and '*'in addition to '/'. The problem is in co-NP complete [13] for $XP^{\{/,//,[],*\}}$ and in P for its three subclasses [3, 14, 15].

[13] showed that the containment problem between one pattern and one pattern set can be reduced to that between two patterns. Hence, the containment problem between two pattern sets is still in coNP-complete. Furthermore, in case that all patterns in two pattern sets don't include '//'or '*', the pattern set containment problem can be decided in polynomial time. However, when patterns have '//', '*' but no '[]', this problem between two pattern sets is in coNP-complete even though this problem between two patterns is in P.

The only heuristic polynomial-time algorithm to decide containment between two patterns is to find a homomorphism between them. This algorithm is practical and sound for patterns, and also complete for its three subclasses. Based on this algorithm, we can also give a heuristic algorithm to decide containment between two pattern sets: for two pattern sets S_1 and S_2, this algorithm reports $S_1 \sqsubseteq S_2$ if $\forall p_1^i \in S_1\ \exists p_2^j \in S_2$ s.t. $p_1^i \sqsubseteq p_2^j$. We also use finding homomorphisms algorithm to decide $p_1^i \sqsubseteq p_2^j$.

6. RELATED WORK

Semantic Caching has been studied a lot in the relational model [4, 10]. Recently, semantic caching has attracted moderate attentions in XML world [7, 16, 5, 1, 12]. In [7], Chen et al consider using cached results of previous XQuery queries to answer new queries. In [16], Yang et al consider mining frequent tree patterns to cache their results for answering new queries. In both works, only queries, whose results have already been cached, can be answered. In [5], Balmin et al consider using pre-computed results of XPath queries also with data values, full paths, or node references to speedup processing of XPath queries. However, this work rules out the combination of results of multiple XPath queries in evaluating XPath queries. The most similar work to ours is [1, 12]. They consider prefix-selection queries that can also be represented as a tree. However, those queries don't support the important feature of XPath language: descendant axes, and also have strong constraints on structures of query trees, for example, two siblings must have different labels in a query tree.

7. CONCLUSION

This paper has introduced a novel framework for a new semantic caching system. The proposed framework offers the representation system of cached XML data, the algorithms to decide whether a new query can be totally answer by cached XML data or not, and to incrementally maintain cached XML data.

8. REFERENCES

[1] S. Abiteboul, L. Segoufin, and V. Vianu. Representing and querying xml with incomplete information. In *PODS*, 2001.

[2] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural joins: a primitive for efficient xml query pattern matching. In *ICDE*, pages 141–152, 2002.

[3] S. Amer-Yahia, S. Cho, L. V. Lakshmanan, and D. Srivastava. Minimization of tree pattern queries. In *SIGMOD*, 2001.

[4] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan. Scalable template-based query containment checking for web semantic caches. In *ICDE*, pages 493–504, 2003.

[5] A. Balmin, F. Ozcan, K. S. Beyer, R. J. Cochrane, and H. Pirahesh. A framework for using materialized xpath views in xml query processing. In *VLDB*, pages 60–71, 2004.

[6] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simeon. Xquery 1.0: An xml query language, November 2003.

[7] L. Chen and E. A. Rundensteiner. Ace-xq: A cache-aware xquery answering system. In *WebDB*, pages 31–36, 2002.

[8] Y. Chen, S. B. Davidson, and Y. Zheng. Blas: An efficient xpath processing system. In *SIGMOD*, pages 47–58, 2004.

[9] J. Clark. Xml path language (xpath).

[10] S. Dar, M. J. Franklin, and B. Jonsson. Semantic data caching and replacement. In *VLDB*, pages 330–341, 1996.

[11] S. Flesca, F. Furfaro, and E. Masciari. On the minimization of xpath queries. In *VLDB*, pages 153–164, 2003.

[12] V. Hristidis and M. Petropoulos. Semantic caching of xml databases. In *WebDB*, pages 25–30, 2002.

[13] G. Miklau and D. Suciu. Containment and equivalence for an xpath fragment. In *PODS*, pages 65–76, 2002.

[14] T. Milo and D. Suciu. Index structures for path expressions. In *ICDT*, pages 277–295, 1999.

[15] P. T. Wood. Minimizing simple xpath expressions. In *WebDB*, pages 13–18, 2001.

[16] L. H. Yang, M. L. Lee, and W. Hsu. Efficient mining of xml query patterns for caching. In *VLDB*, pages 69–80, 2003.

A Data Model and Query Language to Explore Enhanced Links and Paths in Life Science Sources

George Mihaila
IBM T.J. Watson Research Center
mihaila@us.ibm.com

Felix Naumann
Humboldt-Universität zu Berlin
naumann@informatik.hu-berlin.de

Louiqa Raschid
University of Maryland
louiqa@umiacs.umd.edu

Maria Esther Vidal
Universidad Simón Bolívar
mvidal@ldc.usb.ve

ABSTRACT

Links in life science sources capture important domain knowledge. However, current simple physical link implementations are not rich in either representation or semantics. This paper proposes the *e-link* framework and tools to assist scientists in exploring and exploiting the knowledge that should be captured in links.

1. INTRODUCTION

An abundance of Web-accessible bio-molecular data sources contain data about scientific entities, such as genes, sequences, proteins and citations. The sources have varying degrees of overlap in their content and they are richly interconnected to each other. Experiment protocols to retrieve relevant data objects (data integration queries) explore multiple sources and traverse the links and the paths (informally concatenations of links) through these sources. While such navigational queries are critical to scientific exploration, they also pose significant limitations and challenges.

The key limitation is that current physical link implementations are inherently poor with respect to both syntactic representation and semantic knowledge. We illustrate using an example. OMIM is a source that has knowledge on human genes and genetic disorders. Each entry in OMIM may have links to entries in multiple other sources. While there is significant knowledge and curation effort associated with the creation of each of these links, this knowledge is not explicitly captured in the link. All links appear to occur at the level of the OMIM entry. In a later section, we discuss many examples of specific sub-elements within the OMIM entry that are actually associated with the link; this is additional knowledge that is useful to the scientist. Similarly, suppose we consider two or more links from OMIM entries to say proteins in SWISSPROT. These links do not explicitly specify the underlying relationship that led to the creation of this link and one may assume that the links capture the same relationship. However, a scientist who examines the OMIM and SWISSPROT entries may conclude that the relationships that have been captured are quite different.

Links between entries in the sources are created for many different reasons. Biologists capture new discoveries of an experiment or study using links, whereas data curators add links to augment, to complete or to make consistent the knowledge captured among multiple sources. For example, a result reported in a paper in PUBMED may lead a curator to insert a link from a data entry in say OMIM to this citation in PUBMED. Algorithms insert links automatically when discovering similarities among two data items, e.g., to represent sequence similarity following a BLAST search. Thus, the simple unlabeled physical links that are in use today are insufficient to represent subtle and diverse relationships.

In this research, we propose the *e-link* framework and methodology and tools to assist scientists in exploring and exploiting the knowledge that should be captured in links. To do so we must accomplish the following three objectives. In this paper, we address the first two objectives.

- Develop a data model that can represent sources, data objects and the enhanced semantic links (*e-links*) between data objects.
- Develop a query language and query evaluation engine for scientists to meaningfully explore these semantically enhanced *e-links* and paths.
- Develop (machine-assisted) techniques to extract, generate and label existing links to create *e-links*.

We briefly review related research. Clearly there is much related work in knowledge representation. RDF and XML Topic Maps provide a rich conceptual framework and expressive query languages that can be applied to represent the *e-link* framework. Ontologies also provide a framework to capture the semantics of links. Our objective in this paper is to focus on the simple framework needed for *e-links* and we expect to freely exploit ideas from the richer frameworks.

The three major repositories, NCBI, DDBJ and EBI have recently made significant efforts to provide integrated access to the many entries and links between entries that exist in the sources that they manage. Examples include ELink [1], RefSeq and LocusLink at NCBI; LinkDB [2], and Integr8 [3].

These projects focus on providing unified access to the links but do not attempt to enhance the representation and the semantics of links.

There are other projects that target and enhance specific links. For example, the PDBSProtEC project [4, 12] is a resource to link PDB chains with SwissProt codes and EC numbers. We expect that there will be many such efforts to enhance specific links. Our *elink* framework is generic and can in principle be applied to any enhanced link; we would of course need to develop the machinery to interpret such enhanced links and translate to our *e-link* framework.

Recent work in [7, 11] present sophisticated query languages to explore knowledge in interconnected data sources, beyond simple navigational queries. For example, the IBM DB2 Graph Extender supports complex queries on large object graphs and can discover associations important to systems biology, e.g., across genome comparisons. These projects can be extended to accommodate the enhanced semantics of the *e-link* framework.

The paper is organized as follows: Section 2, describes a simple model for life science sources and presents examples of enhanced links. In Sec. 3, we present the data model for the *e-link* framework and in Sec. 4, we (informally) present the query language. Sec. 5 considers the semantics of queries and Sec. 6 describes the steps of query evaluation.

2. MODELING LIFE SCIENCES SOURCES

We first describe a simple model for life science sources. The model was first presented in [8, 9] where we investigated interesting metrics to characterize life science sources.

2.1 A Simple Model for Life Science Sources

Life science sources may be modeled at three levels: the physical level, the object level and the ontology level. The physical level corresponds to the actual data sources and the links that exist between them. An example of data sources and links is shown in Fig. 1.

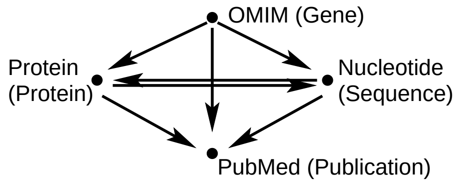


Figure 1: A Source Graph for NCBI Data Sources (and Corresponding Scientific Entities)

The sources are a subset of sources at the National Center for Biotechnology Information (NCBI) and can be accessed at `http://www.ncbi.nlm.nih.gov`. The sources are PubMed, Protein, Nucleotide, and Omim (not an NCBI source). The physical level is modeled by a directed *Source Graph*, where nodes represent data sources and edges represent a physical link between two data sources. A data object in one data source may have a link to one or more data objects in another data source, e.g., a gene associated with a disease in Omim links to multiple citations in PubMed. An *Object Graph* as shown in Fig. 2 represents the data objects of the sources and the object links between the objects. Thus, each link in the *Source Graph* corresponds to a collection of object links of the *Object Graph*, each going from a data object in one source to another object, in the same or a different source. Note that links in *Source Graph* can be bi-directional (though not always symmetric) and *Source Graph* may be cyclic.

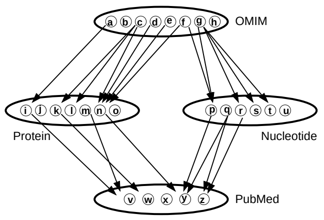


Figure 2: An Object Graph for NCBI Data Sources with Data Entries (Objects) and Links

The ontology level consists of classes (entity classes, concepts or ontology classes) that are implemented by one or more physical data sources or possibly parts of data sources of *Source Graph*. For example, the class *Citation* may be implemented by the data source PubMed. A source of *Source Graph* typically provides a unique identifier for each of the entities or objects in *Object Graph* and includes attribute values that characterize them. Table 1 provides a mapping from the logical classes to some physical data sources of some *Source Graph*.

CLASS	DATA SOURCE
Sequence (s)	NCBI Nucleotide database EMBL Nucleotide Sequence database DDBJ
Protein (p)	NCBI Protein database UniProt SwissProt
Citation (c)	NCBI PubMed NCBI Book

Table 1: A Possible Mapping from Ontology Classes to Physical Data Sources of *Source Graph*

2.2 Enhancing Links Among Data Entries

We present examples to illustrate that the the simple unlabeled physical links that are in use today are insufficient to represent diverse relationships.

Consider a SwissProt entry with a link to an Omim entry; it is illustrated in Fig. 3. In the flat structure of the SwissProt entry, this link is represented by embedding an Omim ID as a top-level attribute of the entry, and the entry may include an HTML hyperlink to the Omim entry. Such a link neither represents the sub-element of the SwissProt entry to which the link refers, nor the sub-element of the Omim entry to which the link points, nor does it represent the reason to insert this link. Biologists examining the SwissProt entry rely on their experience and can infer these link properties after a time-consuming examination. Machines and algorithms cannot perform such analysis at the necessary level of detail and precision. In this particular case, the *e-link* should not originate from the SwissProt entry; instead the "real" origin is the CC-DISEASE attribute within that entry. The *e-link* should also not represent a generic relationship; it should be labeled as a *causal* relationship, telling humans and machines that *the protein in question is known to cause the disease* pointed to by the *e-link*.

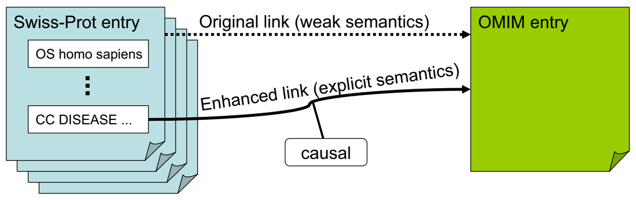

Swiss-Prot entry
OS homo sapiens
CC DISEASE ...
Original link (weak semantics)
Enhanced link (explicit semantics)
causal
OMIM entry

Figure 3: The Enhanced *e-link* from Swiss-Prot to OMIM

We note that determining the semantics and labels of *e-links* for some physical link between two sources may not be straightforward, and scientists may not always reach a consensus as to the desired semantics. Nevertheless, we believe that the significant activity related to ontologies for the life sciences, and the resulting advances in establishing controlled vocabularies to describe functionality and relationships among concepts, e.g., the GO Ontology [5] and GOA [6] will contribute towards the success of our research.

Consider the physical link from the origin source UNIPROT to the target source OMIM illustrated in Fig. 4. The physical link instances between UNIPROT and OMIM entries corresponds to two distinct *e-links* with different semantics. Both *e-links* originate in the same sub-element of UNIPROT. One *e-link* has the meaning *is causal for disease* and the target sub-element in OMIM is CLINICALFEATURES. The second *e-link* has the meaning *describes genetic defects* and the target sub-element in OMIM is MAPPING. In this example, the original physical link of the *Source Graph* is classified as two *e-links*, whose target sub-element in OMIM is different, and where the two *e-links* have different meaning.

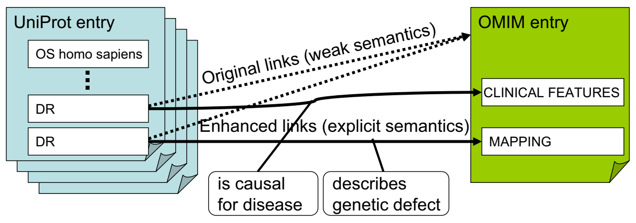


Figure 4: Enhancing a Link from UniProt to Omim to Produce Two *e-links* with Different Target Sub-Elements in Omim

Next, consider the link from the origin source UNIPROT to the target source GO in Fig. 5. This physical link captures three *e-links*, where the origin sub-element and the meaning of the three *e-links* is different. The target is the GO entry. The first *e-link* has meaning *has (sub)cellular location* and the origin sub-element in UNIPROT is SUBCELLULAR. The second *e-link* has the meaning *has molecular function* and the origin sub-element in UNIPROT is MOLFUNC and the third *e-link* has meaning *participates in biological process* and the origin sub-element in UNIPROT is BIOLPROC.

Finally, we consider the case where different physical links between different data sources appear to have the same meaning. There are six physical links in the *Source Graph*, each originating in the same sub-element of UNIPROT. The target of each link is a data entry in one of six different protein data sources, InterPro, Pfam, SMART, PROSITE, PRINTS, and TIGRFAMS; the links are illustrated in Fig. 6.

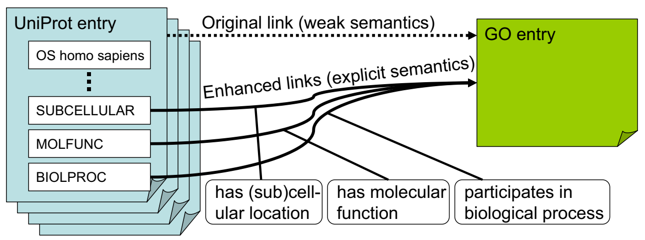

UniProt entry
OS homo sapiens
SUBCELLULAR
MOLFUNC
BIOLPROC
Original link (weak semantics)
Enhanced links (explicit semantics)
GO entry
has (sub)cellular location
has molecular function
participates in biological process

Figure 5: Enhancing a Link from UniProt to GO to Produce Three *e-links* with Different Origin Sub-Elements in UniProt

While the physical links are between different sources, they each have the meaning *contains a sequence signature*. This example of six physical links producing potentially six *e-links*, all of which are equivalent with respect to meaning is a frequent occurrence in life science sources, because the contents of sources overlap and the sources are richly interconnected. This motivates our research on a data model that can specify the equivalence of *e-links* that are of the same link type.

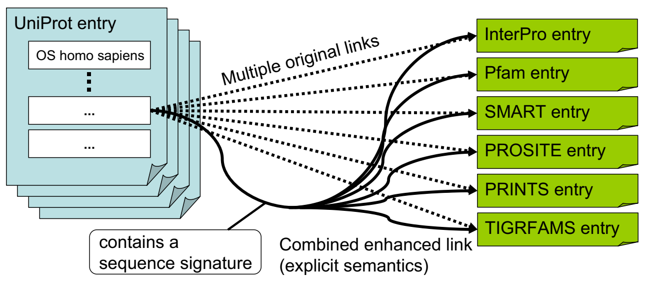


Figure 6: Enhancing a Link from UniProt to Six Protein Data Sources

3. THE DATA MODEL

In this section we formalize the life sciences graphs introduced in the previous section. The data model we consider is comprised of three graphs, which capture the various abstraction levels: the *Ontology Graph* at the logical level and the *Source Graph* and *Object Graph* at the physical level. This is an extension of our previous work on modeling life science sources [8, 9].

An *Ontology Graph* is a graph (C, LT), where C is a set of logical classes (e.g., protein, gene, citation) and LT is a set of link types between logical classes. A link type is a triple (C_1, l, C_2) where C_1 is the origin class, C_2 is the target class and l is a label from a set of link labels L. For example (citation, *describesBehaviorOf*, gene) is a link type between citations and genes. We note that at this stage we only consider a simple label to capture the semantics of a link; in future work we expect to consider the use of ontologies with their richer semantics.

A *Source Graph* is a graph (S, L_S), where S is a set of sources which store instances of logical classes and L_S is a set of source links which implement link types. Each source s stores instances of a single logical class, denoted $m_S(s)$. A source link in LS is a triple (s_1, l, s_2) such that there exists a link type $(m_S(s_1), l, m_S(s_2))$ in LT. Given a typed link (e.g., *describesBehaviorOf*) between class C_1 (e.g., citation) and class C_2 (e.g., gene), a pair of sources (S_1, S_2) where S_1 (PubMed) contains objects of the logical class C_1 and S_2 (UniGene) contains objects of target logical class C_2, can implement this typed link. Note that the set of implemented link types, e.g., (PubMed *describesBehaviorOf*, UniGene) is made public by the sources. Thus, PubMed will typically advertise that it has links of this type. If the reverse links are stored by UniGene, it too will advertise the link. We note that the links between the objects may not be symmetric.

An *Object Graph* is a graph (O, L_O), where O is a set of objects and L_O is a set of links between objects. There is also a mapping $m_O : O \to S$ defining for each object o, the source $m_O(o)$ where o is physically stored. A given object of class C_1 in source S_1 can have a link with a label l (e.g., describesBehaviorOf) to another object of class C_2 in source S_2 only if there is a source link (S_1, l, S_2) in L_S.

In addition we have a link concatenation matrix LL which specifies the meaningful concatenations of link types: if (lt_1, lt_2) is a pair in LL, then the target class of lt_1 is the same as the origin class of lt_2 and the concatenation of lt_1 with lt_2 is meaningful. We note that for now we only consider pairs of links.

We are now ready to integrate the ontology, source and object graphs into a single, unified data model.

DEFINITION 1. *The* e-link *data model is a 9-tuple* $= (C, L, LT, S, L_S, O, L_O, m_S, m_O, LL)$ *where:*

- C *is a set of classes*
- L *is a set of link labels;*
- LT *is a set of link types;*
- S *is a set of sources*
- L_S *is a set of links between sources;*
- O *is a set of objects;*
- L_O *is a set of links between objects;*
- m_S *is a mapping from* S *to* C*;*
- m_O *is a mapping from* O *to* S*;*
- LL *is a set of pairs of link types.*

4. THE QUERY LANGUAGE

We define the *e-link* query language as a regular expression over the alphabet $C \cup L$ where each class occurrence can optionally be annotated with a predicate expression. The BNF specification of the language syntax is given in Fig. 7.

The result to all queries are paths in the *Source Graph* or the *Object Graph*, where a node in the graph is also a path. Some of the uses of the query language are as follows:

- Identify sources in the *Source Graph* that implement a given class: For example, to find sources that implement class "publication ", one can submit the query Q_1 = `publication`.

- Identify sources in the *Source Graph* that implement a given link type: An example is a source that contains *proteins that are linked to an entry in the RefSeq database.* Assuming a link type (`protein`, *linkedToEntryInRefSeq*, `refseqentry`) in LT. The query Q_2 = `protein` *linkedToEntryInRefSeq* retrieves all sources that contain protein entries that are linked to an entry in the RefSeq database.

- Identify paths in the *Source Graph* using wildcards: The symbols ϵ_C and ϵ_L are wildcards matching any class and any link type respectively. For example, to retrieve all the sources linked to the PubMed source by any link type, one can write a query such as the following:
  Q_3 =`publication`[*source* = "PubMed"] ϵ_L ϵ_C;

- Identify paths in the *Source Graph* that satisfy a path regular expression: For example the query Q_4 = `publication` (*describes* | *describes describes*) retrieves all sources that contain classes connected by one or two *describes* links starting from sources containing publications.

- Identify paths in the *Object Graph* that satisfy a path regular expression that can include source/object predicates: For example the query Q_5 = `publication`[*author* = "John Smith" and *title* contains "cancer"] *describes* `protein`[*source* = "RefSeq"] would retrieve publications written by Smith whose title contains the term "cancer" and which describe RefSeq proteins.

Query	:=	"(" Query ")"
	|	Query "|" Query
	|	Query Query
	|	ϵ_C
	|	ϵ_L
	|	Term
Term	:=	ClassName Annotation
	|	LinkLabel
Annotation	:=	empty
	|	"[" Condition "]"
Condition	:=	"(" Condition ")"
	|	Condition "and" Condition
	|	Condition "or" Condition
	|	"not" Condition
	|	Field
	|	Field Op Value
Op	:=	"=" | "$\neq$" | ">" | "<"
	|	"$\geq$" | "$\leq$" | "contains"

Figure 7: The Syntax of the Query Language

5. QUERY LANGUAGE EVALUATION

Intuitively, a query can be evaluated in five steps.

Step 1. Enumerate simple path expressions matching the regular path expression Q (the predicate expressions are carried over). For example, the query Q = `publication` (*describes* | *describes describes*) is expanded as follows:

`publication` *describes*
`publication` *describes describes*

Step 2. For each such simple path expression e, insert ϵ_L in between each pair of consecutive class labels and ϵ_C in between each pair of consecutive link labels to obtain e'. If the simple path expression ends with a link label, append the ϵ_C symbol at the end. This yields the following:

`publication` *describes* ϵ_C
`publication` *describes* ϵ_C *describes* ϵ_C

Step 3. Typecheck each e': verify if each triple of consecutive symbols of the form $c_1 l_1 c2$ corresponds to an actual link type $(c_1, l_1, c_2) \in LT$ and if each triple of consecutive symbols of the form $l_1 c_1 l_2$ corresponds to a pair $(l_1, l_2) \in LL$. Also, if there are wildcards, replace ϵ_C by all possible class and ϵ_L by all possible link types that result in valid path expressions (according to the typechecking rules above).

Suppose that LT specifies that `publication` can only describe `publication`, `protein` or `gene`, and LL includes (*describes*, *describes*). This step yields the following:

`publication` *describes* `publication`
`publication` *describes* `protein`
`publication` *describes* `gene`
`publication` *describes* `publication` *describes* `publication`
`publication` *describes* `publication` *describes* `protein`
`publication` *describes* `publication` *describes* `gene`

Step 4. For each valid e' find all the actual *source path* instances matching e' in the *Source Graph* that also satisfy all the source predicates. A source path instance in the *Source Graph* is said to match a simple path expression if there exists a mapping from the set of all the sources in this path instance to the class labels in the path expression. For a particular *Source Graph*, this step could yield:

PubMed *describes* PubMed
PubMed *describes* SwissProt
PubMed *describes* UniGene
PubMed *describes* PubMed *describes* PubMed
PubMed *describes* PubMed *describes* SwissProt
PubMed *describes* PubMed *describes* UniGene

Step 5. For each source path, evaluate a query evaluation plan (for this path) against the *Object Graph*.

6. QUERY EVALUATION

We provide a sketch of the steps for query evaluation. We then provide a brief description of *SGSearch*, an algorithm to find *source paths* in the *Source Graph*. We then discuss a naive evaluation of the *source paths* against the *Object Graph* (O, L_O) by a mediator accessing utilities ESearch, EFetch and ELink currently supported by the NCBI.

6.1 Query Evaluation Stages

1. Validate the query against LT and LL. This corresponds to Steps 1, 2, and 3 of Sec. 5.
2. Find all *source paths* in the *Source Graph*, (S, L_S) that satisfy the query; this is Step 4. We describe an exhaustive search algorithm *SGSearch* next.
3. Eliminate meaningless *source paths* using LL. Scientists may further eliminate *source paths* that are not of interest to them, e.g., they do not use data from a specific source in the path. They may also rank the *source paths* based on domain specific criteria. Note that this step improves efficiency and usefulness but does not impact the semantics; it is not included in Sec. 5.
4. Evaluate each *source path* on the *Object Graph* (O, L_O), starting with the highest ranked *source path.* We describe a naive evaluation strategy that assumes the mediator has a *decision rule* to determine the link type of links in the *Object Graph.*
5. Return results to the user. This may include the *objects* and links of a path in the *Object Graph* or only the *target objects* reached along paths of the *Object Graph.*

6.2 SGSearch

SGSearch is an extension of a search algorithm presented in [9]. The extension is to consider labeled links in the *Source Graph*; the original algorithm only considered unlabeled links. *SGSearch* is based on a deterministic finite state automaton (DFA) that recognizes a regular expression (query Q_r). The algorithm performs an exhaustive breadth-first search of all paths that satisfy the query. *SGSearch* assumes that Q_r is semantically correct, i.e., the original query has been rewritten as described in Sec. 5 to include all needed wild card class labels, ϵ_C, and link labels, ϵ_L.

Suppose DFA is the automaton that recognizes the regular expression query Q_r. The DFA is represented by a set of transitions, where a transition is a 4-tuple t=(i,f,e,$Pred$), and where, i represents the initial state of t, f represents the final state of t, and e corresponds to the label of t. Note that $e \in C$ U L, i.e., e belongs to the set of class labels or link labels. *Pred* represents a predicate expression to be satisfied by a source that implements a class or by objects in a source. For simplicity we do not discuss *Pred* in this section. The state i (respectively f) may be a *start state* (respectively *end state*) of the DFA.

The exhaustive algorithm *SGSearch* comprises two phases: (a) *build path* and (b) *print path.* In phase *build path*, for each visited transition t^p=(i,f,e,$Pred$), *SGSearch* annotates each transaction with a set t^p.*currentImp* corresponding to the label e. If e is a class label or the wild card class label ϵ_C, t^p.*currentImp* includes *all* s_i in S such that $e \in m_S(s_i)$. If e is a link label, t^p.*currentImp* includes this label. Finally, if e is the wild card link label ϵ_L, then t^p.*currentImp* includes all the labels in L.

For each transition t^p=(i,f,e,$Pred$), if i is not a *start state* of the DFA, then *SGSearch* annotates each element n of t^p.*currentImp*, with a set n.*previousImp*; this is either s_i.*previousImp* or l.*previousImp*, depending on whether n is a source or link. If n is a link label l, *SGSearch* considers the transaction t^{p-1} previous to t^p, and creates a set l.*previousImp* that includes all sources s_j in t^{p-1}.*currentImp* that are adjacent to l in the *Source Graph*(S,L_S) and that satisfy *Pred.* These adjacent sources s_j must participate in a link, such as (s_j, l, s_i) in L_S. Note that if l does not have an adjacent source in t^{p-1}.*currentImp*, it is no longer considered as an element of t^p.*currentImp.*

If n is a source, then, *SGSearch* considers two transitions t^{p-1} and t^{p-2}, where t^{p-1} is previous to t^p and t^{p-2} is previous to t^{p-1}. It finds sources s_j in t^{p-2}.*currentImp* that satisfy *Pred* and, that are adjacent to s_i in L_S through a link label l that is included in t^{p-1}.*currentImp.* Similarly, if s_i does not have an adjacent source in t^{p-2}.*currentImp*, it is no longer considered as an element of t^p.*currentImp.*

In phase *print path*, the algorithm starts from the set t^{fin}.*currentImp* corresponding to the final transition t^{fin} whose final state is an *end state* of the DFA. For each s_i in this set, *SGSearch* uses the set s_i.*previousImp* to construct a path. The path terminates in one of the sources corresponding to the start transition t^1.currentImp; recall that the initial state of this transition is a *start state* of the DFA.

6.3 Naive Evaluation by a Mediator

We now describe a naive evaluation of the *source paths* produced by *SGSearch* against the *Object Graph.* For illustration, we consider the *NCBI Object Graph.* NCBI is the gatekeeper for NIH data sources. The Entrez utilities for search and retrieval from NCBI sources include ESearch, ELink and EFetch. Given a source and some search predicate, ESearch finds objects in the source that satisfy the predicate and EFetch retrieves those objects. Together, they act like the σ relational operator. Given an object identifier (o.UID) and a target source, ELink retrieves all links (o.UID pairs) starting from the given object o_i and reaching objects in the target source. We describe a naive evaluation strategy based on these utilities.

Repeat for each subpath (s_i, l, s_j) in a source path until the path terminates:

1. Invoke ESearch on the current source s_i with the search predicate *Pred.* Retrieve a set of object identifiers (UIDs) for some objects in O_i.

2. Invoke EFetch to obtain XML documents for each object $o_i \in O_i$. Determine those links with link label l.

3. Invoke ELink on all object links from o_i, with label l, and reaching an object o_j in source s_j. Create a set of objects O_j.

We assume that for each source link registered in L_S, the mediator has a decision rule to determine the link type of all outgoing object links from object o_i. We illustrate the decision rules using an example. Consider the portion of a UniProt entry in Fig. 8.

```
      ID    MEFV_HUMAN STANDARD; PRT; 781 AA.
      AC    O15553; Q96PN4; Q96PN5;
      DT    16-OCT-2001 (Rel.  40, Created)
      DT    16-OCT-2001 (Rel.  40, Last sequence update)
      DE    Pyrin (Marenostrin).
      ...   ...
→1    OS    Homo sapiens (Human).
→1    OX    NCBI_TaxID=9606;
      RN    [1]
      ...   ...
      CC    -!- DISEASE: DEFECTS IN MEFV ARE THE
            CAUSE OF FAMILIAL MEDITERRANEAN
      CC        FEVER (FMF) [MIM:249100]...
      ...   ...
→2    DR    MIM; 608107;
→3    DR    MIM; 249100;
      ...   ...
→4    FT    VARIANT 694 694 M -> I (in FMF).
      ...   ...
```

Figure 8: Portions of the UniProt entry O15553

The four lines marked with '→' correspond to four *e-links*. We describe decision rules to classify two *e-links*. For $\rightarrow^1$, the following rule (represented by a triple) will be used to determine the link type: (./OS & ./OX, *is causal for disease*, lt_i). The first item of the triple specifies that when the two sub-elements (attributes) OS and OX occur in the UniProt entry, then this is a link of type lt_i with link label *is causal for disease.* The attribute values (OS Homo sapiens (Human) and OX NCBI_TaxID=9606) correspond to the actual object link. For $\rightarrow^4$, the following rule is used: (./FT, *genetic background*, lt_j). The attribute FT determines the rule to be of type lt_j with label *genetic background.*

7. DISCUSSION

We present the *e-link* framework of a data model and query language that allows scientists to express knowledge of links and to exploit this knowledge in answering queries. We discuss the naive evaluation of these queries by a mediator.

There are clearly many challenges that must be addressed. The first task is developing machine assisted techniques to extract semantics and to provide labels for existing links. We note that there are several ongoing efforts to enhance links [2, 3, 4, 10, 12]. This task is difficult, because a link in the *Source Graph* may often have multiple semantics. The second task is exploiting existing work in ontologies in the task of associating semantics to links. We note that in our current prototype, the semantics is limited to a simple label, whereas ontologies can support richer relationships. Finally, we have to develop robust and efficient techniques for query evaluation that scale to the large distributed *Object Graph* of the life science domain.

Acknowledgements: We thank Barbara Eckman, Stephan Heymann, Zoe Lacroix, Alex Lash, Woei-Jyh Lee and Peter Rieger for their feedback. This research was supported by NSF grants IIS0222847 and IIS0430915 and by the German Research Society (DFG grant no. NA 432).

8. REFERENCES

[1] *www.ncbi.nlm.nih.gov/entrez/query/static/elink_help.html.*

[2] *www.genome.ad.jp/dbget-bin/www_linkdb.*

[3] *www.ebi.ac.uk/integr8/.*

[4] *www.bioinf.org.uk/pdbsprotec/.*

[5] *www.geneontology.org.*

[6] *www.ebi.ac.uk/GOA/.*

[7] Barbara A. Eckman, Paul Brown, A. Kershenbaum, R. Mushlin, and S. Mitchell. The IBM DB2 systems biology graph extender research prototype. *White Paper, IBM Life Sciences*, 2005.

[8] Z. Lacroix, H. Murthy, F. Naumann, and L. Raschid. Characterizing properties of paths in biological data sources. *Proceedings of the DILS Conference and Springer-Verlag Lecture Notes in Computer Science (LNCS)*, (2994):187–202, 2004.

[9] Z. Lacroix, L. Raschid, and M.E. Vidal. Efficient techniques to explore paths in life science data sources. *Proceedings of the DILS Conference and Springer-Verlag Lecture Notes in Computer Science (LNCS)*, (2994):203–211, 2004.

[10] Alex Lash, Woie-Jyu Lee, and Louiqa Raschid. A protocol to extract and generate links capturing marker semantics from pubmed to the human genome. *Under review*, 2005.

[11] Ulf Leser. A query language for biological networks. Technical Report 187, Institut fuer Informatik der Humboldt Universitaet zu Berlin, 2005.

[12] A. Martin. PDBSprotEC: A web-accessible database linking PDB chains to EC numbers via swissprot. *Bioinformatics*, 20(6):986–988, 2004.

Malleable[*] Schemas: A Preliminary Report

Xin Dong
University of Washington
Seattle, WA 98195
lunadong@cs.washington.edu

Alon Halevy
University of Washington
Seattle, WA 98195
alon@cs.washington.edu

ABSTRACT

Large-scale information integration, and in particular, search on the World Wide Web, is pushing the limits on the combination of structured data and unstructured data. By its very nature, as we combine a large number of information sources, our ability to model the domain in a completely structured way diminishes. We argue that in order to build applications that combine structured and unstructured data, there is a need for a new modeling tool. We consider the question of modeling an application domain whose data may be partially structured and partially unstructured. In particular, we are concerned with applications where the *border* between the structured and unstructured parts of the data is not well defined, not well known in advance, or may evolve over time.

We propose the concept of *malleable schemas* as a modeling tool that enables incorporating both structured and unstructured data from the very beginning, and evolving one's model as it becomes more structured. A malleable schema begins the same way as a traditional schema, but at certain points gradually becomes vague, and we use keywords to describe schema elements such as classes and properties. The important aspect of malleable schemas is that a modeler can *capture* the important aspects of the domain at modeling time without having to commit to a very strict schema. The vague parts of the schema can later evolve to have more structure, or can remain as such. Users can pose queries in which references to schema elements can be imprecise, and the query processor will consider closely related schema elements as well.

1. INTRODUCTION

There has been significant interest recently in combining techniques from data management and information retrieval (as surveyed in [4]). The underlying reason is that knowledge workers in enterprises are frequently required to analyze data that exist partially in structured databases and partially in content management systems or other repositories of unstructured data. Similarly, the WWW is a repository of both structured and unstructured sources (webforms and webpages). To support the querying needs in these applications we should be able to seamlessly query both structured and unstructured data, and consider query paradigms that involve both ranking answers and structure based (SQL-like) conditions on query answers.

Previous work in this area focused on dealing with hybrid data *after* the fact. That is, it is assumed that we already have some set of structured data and another set of unstructured data, and the goal is to manage it and query seamlessly.

This paper looks at the entire process of building an application that involves both structured and unstructured data. We ask the following basic question: how do we model data for an application that will involve both structured and unstructured data? In particular, we are concerned with the case where the *border* between the structured and unstructured parts of the data is not well defined, and may evolve over time.

When we start modeling a domain, we typically want to model it as precisely as possible by defining its structure with a schema (or possibly a more expressive modeling paradigm such as an ontology). However, in the process of modeling we may realize the following. First, we may not be able to give a precise model of the domain, either because we don't know what it is or because one does not exist. Second, we may prefer not to model the domain in such level of detail because an overly complex model may be a burden on the users. Third, there are parts of the domain we may want to leave unstructured for the time being.

To address these needs, we propose the concept of *malleable schemas* as a modeling tool that enables incorporating both structured and unstructured data from the very beginning, and evolving one's model as it becomes more structured. A malleable schema begins the same way as a traditional schema, but at certain points gradually becomes vague. The important aspect of malleable schemas is that a modeler can *capture* the important aspects of the domain at modeling time without having to commit to a very strict schema. The vague parts of the schema can later evolve to have more structure, or can remain as such. Users can pose queries in which references to schema elements can be im-

[*]`Merriam-Webster:` *Malleable* – 1: capable of being extended or shaped by beating with a hammer or by the pressure of rollers 2a: capable of being altered or controlled by outside forces or influences b: having a capacity for adaptive change

Eighth International Workshop on the Web and Databases (WebDB 2005), June 16-17, 2005, Baltimore, Maryland.

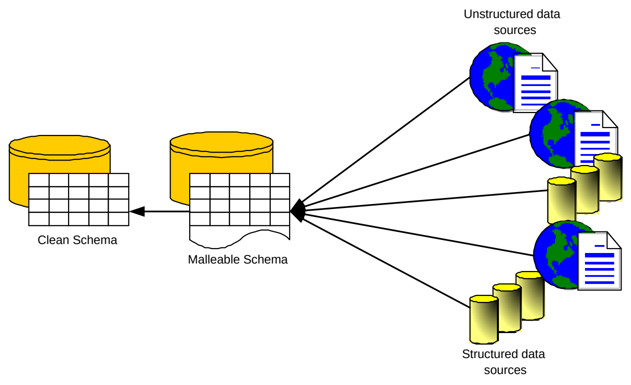


Figure 1: When someone is trying to create a schema for a domain to integrate both structured and unstructured data from a variety of data sources, malleable schemas can help her *capture* the important aspects of the domain at modeling time without having to commit to a very strict schema. The vague parts of the schema can later evolve to have more structure, or can remain as such.

precise, and the query processor will consider closely related schema elements as well. Figure 1 depicts the key idea of malleable schemas.

The concept of malleable schemas evolved from several different applications we have been considering recently: (1) personal information management (PIM) [8], where we constantly model both structured and unstructured data and the model of the domain needs to be very easy to use, (2) information integration on the web [3, 9, 18], where the diversity of information sources does not allow creating a single mediated schema to which everything cleanly maps, and (3) biomedical informatics [21], where our understanding of the domain is constantly evolving from multiple different views and sources.

Large-scale information integration remains one of the important challenges in web data management. By its very nature, as we combine a large number of information sources, our ability to model the domain in a completely structured way diminishes. We argue that the marriage of structured and unstructured data is crucial for building robust integration systems, and the modeling questions that arise are key to the success of such systems. This paper presents our initial work on malleable schemas. We motivate the concept with examples, present an initial formal model, and discuss the implementation challenges.

Related work

There has been a significant body of work on supporting keyword search in databases [15, 1, 16], result ranking [2, 13, 12], and approximate queries [20, 23, 10, 11, 5]. They all assume that the model of the data is precise, but we want to add flexibility in the queries. In contrast, our goal is to allow the model itself to be imprecise in certain ways. Probabilistic databases [24, 7] (and formalisms such as Bayesian Networks) allow imprecision about facts in the database, but the model of the domain is still a precise one.

The work closest to ours is the XXL Query Engine [22], where the queries allow for imprecise references to schema elements. The idea there is that the user will query a large *collection* of XML DTDs, and there is no unifying DTD for all of them. Malleable schemas, in contrast, offer a middle point between a collection of schemas/DTDs (or a corpus [19]) in a domain and a single crisp schema for that domain. The idea of a malleable schema is that someone *is* trying to create a schema for the domain, but in the process of doing so needs to introduce (possibly temporarily) some imprecision into the model. We expect to leverage some of the techniques in [22, 23] in our query processing engine.

Outline: We first present motivating examples for malleable schemas. Section 3 defines a bare-bones formalism that includes malleable classes and properties, and describes the semantics of querying. Section 4 describes several extensions to the basic model, and Section 5 discusses implementation issues. Section 6 concludes.

2. MOTIVATING EXAMPLES

We present two motivating examples for malleable schemas taken from our application domains: information integration on the web and personal information management. The intuition underlying malleable schemas is the following. A traditional schema is a very structured specification of the domain of interest. It assumes that you *know* the structure that you're trying to capture and that it *can* be specified. Malleable schemas are meant for contexts in which one or more of the following hold:

- There is no obvious structure for the domain, and therefore our model of the domain needs to be vague at certain places.
- The structure of the domain is not completely known at modeling time, and may become clearer as the application evolves and the user needs clarify.
- The structure is inherently evolving over time because the domain is extremely complicated and itself the subject of study (e.g, biomedical informatics). Consequently, by nature there will always be parts of the domain that are not precisely modeled.
- A complete structure of the domain would be too complicated for a user to interact with. For example, trying to model every detail of items found on one's desktop in a PIM system would be too overwhelming for a typical user, and maintaining the model would also be impractical.
- The borders between the structured and unstructured parts of the data are fuzzy, and therefore the modeling paradigm needs to support smoother transitions between the two parts.

The idea of malleable schemas is the following. A modeler starts out creating a schema of a domain intending to capture the domain as precisely as possible. However, at certain points in the modeling process, the schema can become less precise. Malleable schemas provide a mechanism by which the modeler can *capture* the imprecise aspects of the domain during the modeling phase in a well principled fashion. Malleable schemas allow the modeler to capture these aspects using keywords, but tell the system that these keywords are meant to capture elements of the schema, rather than being arbitrary keyword fields.

In the discussion below we assume a very simple data model: our domain is comprised of objects (with ID's). Objects have properties – we distinguish between relationships that relate pairs of objects and attributes that relate objects with ground values. Objects are members of classes that can form a hierarchy. We assume objects can belong to multiple classes.

Example 1. Consider building an information integration system for web sources, whose goal is to answer queries from multiple databases available on the web (e.g., querying multiple real-estate sites).

You begin modeling the domain by trying to capture the salient aspects of real-estate that appear in the sources, and such that you'll be able to pose meaningful queries on as many sites as possible. As an example, you create the class RealEstate, intended to denote real-estate objects for sale or rental. In an ideal world (which seems likely when you start building the application), there would be some obvious sub-classes of RealEstate (such as houses, condo's) that you would incorporate into the model. However, after inspecting several sites you realize that there are many more sub-classes, and the relationship between them is not clear. Furthermore, different sites organize real-estate objects in varying ways, and the concepts used in one place overlap but don't correspond directly with concepts used elsewhere. For example, you may encounter vacation rentals, short-term rental and sublets. As a consequence, you cannot create a model of real-estate such that there would be a clean mapping between your categories and those used in the sources. In short, there is no single way of identifying all the subclasses of RealEstate.

What you would like to do now is to create a set of subclasses, each described by words or a phrase (typically found as menu items on a real-estate search form on the web). The subclasses will not necessarily be disjoint from each other; in fact, there may be overlaps between the classes. Later on in the life of the application, after having seen many real-estate listings and user queries, you may decide to impose more structure on the subclasses of RealEstate.

In principle, you could do this by creating a property for the class RealEstate called RealEstateType, and have keywords or phrases be the content of that property. However, while doing so could be a way of *implementing* malleable schemas (see Section 5), it has several disadvantages from the modeling perspective because the system does not *know* that these keywords are identifying subclasses of RealEstate. Specifically, (1) you would like to refer to these subclasses in queries in the same way as you refer to other sub-classes, (2) later on you would like to evolve the schema (possibly with the help of the system) to create a more crisp class hierarchy, and (3) you may want to create subclasses for these classes as well. Hence, in a sense, you want to create a new keyword property, but you want the system to know that it is identifying subclasses of an existing class.

This type of example is extremely common in information integration applications that involve many independently developed sources. By nature these domains are complicated and there is no obvious single way to model them. Different categorizations arise because site builders have different views of the world, and often because of natural geographical differences. In addition, large-scale information integration fundamentally pushes on the limits being able to model a domain with a single structured representation. □

Example 2. The following example illustrates that the same idea can be applied to properties in the schema. Consider the domain of personal information management, where the goal is to offer users a logical view of the information on their desktops [8]. (Note that in practice this logical view is created *automatically* without any investment by the user).

Suppose you are creating a schema for information that people store on their desktops. You create a class called Project, and a property called Participant. But soon you realize that not all participants are equal, and you have various kinds of participation modes. For example, you may have a programmer on the project, a member in the initial planning phases, advice-giver, etc. You cannot anticipate all the possible participation modes nor classify them very crisply. Hence, you would like to create sub-properties of Participant so you can at least *capture* some of the information about the types of participation, and have these sub-properties described by keywords. Note that in this example, even if you could create a clear description of all the types of participation, you may not want to do so because the model will be too complex for users to understand.

This example is not possible to implement with yet another keyword attribute as we did in Example 1. Suppose you create a text property called ParticipationType. The question is then what object to attach it to. It does not suffice to attach it to the participant object because it is not a property of that object, but of the relationship of that object to the project. In principle, the keyword is expressing a relationship between two objects in the domain, and the only way to do that in the object-oriented model we are considering is with a property. Of course, even if you could express ParticipationType somehow, all the disadvantages mentioned in Example 1 still hold. □

Note that one of the early purported advantages of XML is that you can add tags (corresponding to properties and classes) at will. Even ignoring for a moment that XML has evolved to be mostly guided by schemas, XML is, again, a possible implementation avenue for malleable schemas. However, our focus is on the modeling aspects – trying to create a schema for a domain while capturing the vague aspects and evolving the schema with time.

3. FORMALIZING MALLEABLE SCHEMAS

We now describe a formal model for malleable schemas. We focus on the main constructs, and then mention several extensions in Section 4.

3.1 The data model

We frame our discussion in the context of a very simple schema formalism, close in spirit to object-oriented schemas. There have been a plethora of object-oriented modeling languages suggested in the literature. Our goal is not to argue for one or the other. Instead, we chose a set of features from these languages that are important for our discussion, and our focus is on adding malleable features to the formalism.

We model the domain using objects and properties. Each property has a domain and a range, where the domain is a set of classes, and the range is either a set of classes or a set of ground values. We distinguish between two types of properties: relationships, whose ranges are sets of classes, and attributes, whose ranges are sets of ground values. In other words, a relationship is a binary relation between a pair of objects, while an attribute is a binary relation between an object and a ground value. We denote classes by

$C_1, \dots, C_m$, and properties by $P_1, \dots, P_n$. In what follows, we refer to classes and properties collectively as *elements*.

We support class hierarchies and property hierarchies, which model the IS-A relationships. For example, Condo is a sub-class of RealEstate, and programmer is a sub-property of participant. Specifically, $C_i \sqsubseteq C_j$ denotes that C_i is a sub-class of C_j, and $P_i \sqsubseteq P_j$ denotes that P_i is a sub-property of P_j. We assume that the classes form a directed acyclic graph, as do the properties. Note that a sub-class inherits properties from its parent classes. That is, if C_1 is in the domain (resp. range) of P_1, and $C_2 \sqsubseteq C_1$, then C_2 is also in the domain (resp. range) of P_1. The domain and range of a sub-property can be sub-classes of those of its parent-properties. Specifically, if C_1 and C_2 are in the domain (resp. range) of P_1 and P_2 respectively, and $P_1 \sqsubseteq P_2$, then $C_1 \sqsubseteq C_2$.

The malleable schema elements: The malleable elements look exactly the same as the other schema elements, except for the following (mostly conceptual) differences:

- While the name of a regular class or property is typically a carefully chosen string, the names of schema elements can be keywords or phrases, and those are often obtained from external sources. Later we will extend malleable schema elements to include also regular expressions (Section 4).
- For simplicity, we restrict malleable elements to appear only on the left-hand side of $\sqsubseteq$ inclusions. We can easily extend and allow malleable elements on the right-hand side (i.e., have a sub-class element for a malleable element).
- They are marked as malleable. (This is not a requirement, but it may be important for future schema evolution.)

We refer to malleable elements as either *malleable classes* or *malleable properties*. Note that the same name can be both a malleable property and a malleable class, though they are treated as two distinct elements in the schema.

While from a formal point of view malleable schema elements are not so different from ordinary ones, the important point to emphasize is how they are used in the modeling process. The typical process of modeling a domain assumes that we are trying to come up with a very clean model, and hence choose our schema names carefully. In contrast, the malleable schema elements are meant for the cases where we cannot (maybe temporarily) model the domain cleanly, and so we capture certain aspects using keywords. Hence, by nature we may have many overlapping malleable classes or properties (possibly even identical ones called differently), and there will typically be relatively many malleable sub-classes for a class (or sub-properties of a property).

Example 3. Continuing with example 1, suppose we define the following malleable sub-classes of RealEstate: VacationRental, ShortTermRental, and Sublet. In addition, we define the following malleable sub-properties of contactPerson: agent, leaseAgent, and rentalClerk. Note that it is hard to precisely define the relationships between these malleable schema elements (for example, VocationRental, ShortTermRental and Sublet can largely overlap), but we would like to capture them in the model. Formally, we have:

- VacationRental, ShortTermRental, Sublet $\sqsubseteq$ RealEstate
- agent, leaseAgent, rentalClerk $\sqsubseteq$ contactPerson

3.2 Queries

In our discussion of queries we do not pin down a specific query language. Instead, we describe the principles of incorporating malleable schemas into a given query language. Our goal is to modify a given query language as minimally as possible.

There are two changes we make to the query language. First, wherever we can refer to a class (resp. property) we allow the query to refer to a malleable class (resp. malleable property). Second, we distinguish between *precise* references in the query and *imprecise* ones. We denote imprecise references by $\sim K$, where K is either a class or a property.

Example 4. Consider the following query that asks for short term rentals in the Tahoe area. Note that we have an imprecise reference to ShortTermRental and to leaseAgent.

Q : SELECT city, price, $\sim$ leaseAgent
FROM $\sim$ ShortTermRental
WHERE location="Tahoe"

□

A query that only makes precise references is answered in exactly the same way as it would be otherwise. That is, we treat every malleable schema element as a normal schema element.

The interesting case is when the query can make imprecise references to the malleable schema elements. Intuitively, when we have a reference $\sim K$, we want to refer to all elements in the schema that are *similar* to the element K. We do not make the definition of similarity part of the query language, since it depends on the particular context of the application. For example, the following types of similarities can be employed:

- **Term similarity:** Schema names can be compared by using some string distance such as the Levenstein measure [6], or according to some lexical references, such as Wordnet [25], or by the term usage similarity computed with some TF/IDF measure on a corpus of documents on the application domain, or using the combination of any of the above.
- **Instance similarity:** Similarity can be estimated by gleaning information from the instances in the database. For example, if the instances of Apartment and Flat tend to have very similar characteristics, we may deem them to be similar.
- **Structural similarity:** Here, the similarity of two elements can be determined by their context. We can compare the super-elements, sub-elements, and sibling-elements of two elements. For example, if two elements have very similar sub-elements, chances are higher that they are similar. Also, two sibling elements can be similar as they might overlap.
- **Schema-corpus similarity:** There have been several pieces of recent work exploring the use of schema corpora for tasks such as schema matching and mediated schema creation [14, 19]. The underlying idea in these works is to leverage statistics on large collections of schemas in order to determine similarity between attributes from disparate schemas. The same idea can be applied here, where instead of similarity

between disparate schemas we consider similarity between terms in the same malleable schema. In fact, a malleable schema can be viewed as an intermediate point in the evolution of a corpus of schemas into a traditional schema.

Note that in principle, the names of schema elements appearing in imprecise references do not even have to be in the schema. Hence, malleable schemas are attractive in cases where users are querying unfamiliar (or very complex) schemas.

Reformulating queries over malleable schemas: Given a similarity measure over malleable schema elements, the next issue is how to *expand* a query over a malleable schema to get the intended answer.

In the simplest case, query reformulation amounts to expanding to a union query. For example, if in Q the reference to leaseAgent were a precise one, then we simply need to create a union query that considers both ShortTermRental and VacationRental, assuming they were deemed to be similar sub-classes. However, this is not the end of the story. First, it may be the case that VacationRental does not have a leaseAgent property. In that case we need to pose the query so that the tuples coming from VacationRental do not have the column for leaseAgent (otherwise the query will be invalid). Second, since Q does have an imprecise reference to leaseAgent, we need to check several combinations, resulting in the following query:

```
Q′ : SELECT city, price, leaseAgent
     FROM ShortTermRental
     WHERE location="Tahoe"
     OR
     SELECT city, price, rentalClerk
     FROM ShortTermRental
     WHERE location="Tahoe"
     OR
     SELECT city, price, leaseAgent
     FROM VacationRental
     WHERE location="Tahoe"
     OR
     SELECT city, price, rentalClerk
     FROM VacationRental
     WHERE location="Tahoe"
```

Some of these subqueries may not be valid, and therefore need to be pruned. Furthermore, some of the subqueries can be combined (returning four attributes in each query block).

Finally, we note that there has been significant work on trying to rank answers of queries posed over combinations of structured and unstructured data [4]. We do not go into that issue here, and believe that it is largely orthogonal to the concept of malleable schemas.

Querying the schema: In addition to allowing queries on the instances, we allow queries on the schema (e.g., in the spirit of [17]), as the user might want to know the relationship between the schema elements to help evolve the schema. Given class C, the user can ask for C's parent-classes and sub-classes, and more importantly, for classes that are similar to C. The same queries can be posed for properties too.

4. EXTENSIONS

We now briefly mention several extensions to our basic model for malleable schemas.

Malleable property chains: This extension is a powerful generalization of malleable properties. In addition to the imprecision that can be captured with malleable properties, malleable chains can capture varying *structures* of data. For example, when we integrate information about people from multiple sources, not only do we have different properties for people, but they may be structured differently (e.g., the nesting structure of name, address, etc.). Note that in querying, the similarity among chains compares not only each of the properties in the chain, but also global aspects of the chain. Hence, for example, we may consider two chains with *different* lengths to be similar (e.g., phoneNo with contact/phone), or we may consider the concatenation of two chains to be similar to another chain(e.g., name/firstName and name/lastName with fullName).

Element names as regular expressions:[1] Often there is more structure to the set of sub-classes (or sub-properties) we want to define, and this structure can be described by regular expressions. For example, we may want to create properties *Agent to denote any kind of agent, and define *Agent $\sqsubseteq$ agent to specify that they are sub-properties of agent. In this way we may help *identify* various properties that we want to be agent-related properties.

Malleable values: We often capture aspects of objects in our model with values. For example, when modeling web sites for an integration application, we may have an attribute topic that is assigned one of several values (e.g., BusinessRelated, KidsRelated, Shopping). Formally, these can also be specified as sub-classes in the model, but it is sometimes easier to model such distinctions with values. Hence, we can also support *malleable values*. For example, suppose you created a value BusinessRelated for modeling web sites that have content related to business. However, you then realize that you are not quite sure what you precisely mean by this category. There are web sites that offer articles about business, reviews of business and products, and sites about business people. You can create a description attribute that can have these values and maybe later evolve them into categorization as well.

5. IMPLEMENTATION

We are currently implementing a prototype modeling and querying tool for malleable schemas. We are implementing it over a relational database, though most of the principles of the implementation should carry over to XML, object-relational systems or data integration systems. The details of the implementation are beyond the scope of this paper. We briefly describe its main components below.

- *Modeling*: The modeling tool enables the modeler to create a malleable schema (in terms of classes and properties). The tool also allows to query the model itself in the process of modeling. For example, when the modeler creates a sub-class, she may want to query for similar sub-classes that are already in the schema.

[1]We thank Gerhard Weikum for this idea.

- *Translation to relational schema*: We take the malleable schema and create a malleable relational schema for storing the data.
- *Query reformulation*: Given a query over the relational schema, we translate it into a set of SQL queries that can be posed over the database. The translation process obtains similarity measures between schema elements from an external module.
- *Ranking*: The ranking of the answers in the result considers two factors. First, the set of queries generated by the query reformulator is ordered by the similarity of the schema elements. Second, when we actually see the tuples in the result, we may further refine the ordering of the answers.

6. CONCLUSIONS AND FUTURE WORK

We described malleable schemas, a conceptual tool for modeling in applications that involve both structured and unstructured data. The key idea underlying malleable schemas is that the modeler should be able to capture all the aspects of the domain without having to commit to a clean schema immediately. We argue that such a capability is crucial in applications that combine data from a large number of sources since it is typically impossible to create a clean single schema from the start. In fact, malleable schemas can be viewed as an intermediate point in the evolution of a large collection of schemas into a single coherent schema for a domain. Malleable schemas raise several interesting semantic issues, as well as challenges for efficient query processing and automatically evolving a malleable schema to more structured schema.

Acknowledgments

We thank the anonymous reviewers for their insightful comments. The research was funded by NSF CAREER Grant 9985114, NSF ITR Grant 0205635, and DARPA Contract 03-000225.

7. REFERENCES

[1] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: A system for keyword-based search over relational databases. In *ICDE*, 2002.

[2] S. Agrawal, S. Chaudhuri, G. Das, and A. Gionis. Automated ranking of database query results. In *Proc. of CIDR*, 2003.

[3] K. C.-C. Chang, B. He, and Z. Zhang. Toward large scale integration: Building a metaquerier over databases on the web. In *CIDR*, 2005.

[4] S. Chaudhuri, R. Ramakrishnan, and G. Weikum. Integrating DB and IR technologies: What is the sound of one hand clapping? In *Proc. of CIDR*, 2005.

[5] W. W. Cohen. Data integration using similairty joins and a word-based information representation language. *ACM Transactions on Information Systems*, 18(3):288–321, 2000.

[6] W. W. Cohen, P. Ravikumar, and S. E. Fienberg. A comparison of string distance metrics for name-matching tasks. In *IIWEB*, pages 73–78, 2003.

[7] N. Dalvi and D. Suciu. Answering queries from statistics and probabilistic views. In *VLDB*, 2005.

[8] X. Dong and A. Halevy. A Platform for Personal Information Management and Integration. In *Proc. of CIDR*, 2005.

[9] X. Dong, J. Madhavan, and A. Halevy. Mining structures for semantics. *ACM SIGKDD Explorations Newsletter*, 6:53–60, 2004.

[10] R. Fagin. Fuzzy queries in multimedia database systems. In *PODS*, 1998.

[11] L. Gravano, P. G. Ipeirotis, H.V.Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *VLDB*, 2001.

[12] L. Guo, J. Shanmugasundaram, K. Beyer, and E. Shekita. Structured value ranking in update-intensive relational databases. In *ICDE*, 2005.

[13] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked keyword search over XML documents. In *SIGMOD*, 2003.

[14] B. He and K. C.-C. Chang. Statistical schema matching across web query interfaces. In *Proc. of SIGMOD*, 2003.

[15] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient IR-style keyword search over relational databases. In *VLDB*, 2003.

[16] V. Hristidis and Y. Papakonstantinou. DISCOVER: Keyword search in relational databases. In *VLDB*, 2002.

[17] L. V. S. Lakshmanan, F. Sadri, and S. N. Subramanian. Schemasql: An extension to sql for multidatabase interoperability. *ACM Transactions on Database Systems*, 26(4):476–519, 2001.

[18] A. Y. Levy, A. Rajaraman, and J. J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proc. of VLDB*, pages 251–262, Bombay, India, 1996.

[19] J. Madhavan, P. Bernstein, A. Doan, and A. Halevy. Corpus-basd schema matching. In *ICDE*, 2005.

[20] A. Marian, S. Amer-Yahia, N. Koudas, and D. Srivastava. Adaptive query processing of top-k queries in XML. In *ICDE*, 2005.

[21] P. Mork, A. Halevy, and P. Tarczy-Hornoch. A model for data integration systems of biomedical data applied to online genetic databases. In *AMIA*, 2001.

[22] A. Theobald and G. Weikum. Adding relevance to XML. *Lecture Notes in Computer Science*, 1997:105–124, 2000.

[23] A. Theobald and G. Weikum. The index-based XXL search engine for querying XML data with relevance ranking. In *EDT*, 2002.

[24] J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *CIDR*, 2005.

[25] Wordnet. http://www.cogsci.princeton.edu/ wn/.

Mining the inner structure of the Web graph[*]

Debora Donato
Universita di Roma,“La Sapienza”
donato@dis.uniroma1.it

Stefano Leonardi
Universita di Roma,“La Sapienza”
leon@dis.uniroma1.it

Stefano Millozzi
Universita di Roma,“La Sapienza”
millozzi@dis.uniroma1.it

Panayiotis Tsaparas
University of Helsinki
tsaparas@cs.helsinki.fi

ABSTRACT

Despite being the sum of decentralized and uncoordinated efforts by heterogeneous groups and individuals, the World Wide Web exhibits a well defined structure, characterized by several interesting properties. This structure was clearly revealed by Broder et al. [4] who presented the evocative *bow-tie* picture of the Web. Although, the bow-tie structure is a relatively clear abstraction of the macroscopic picture of the Web, it is quite uninformative with respect to the finer details of the Web graph. In this paper we mine the inner structure of the Web graph. We present a series of measurements on the Web, which offer a better understanding of the individual components of the bow-tie. In the process, we develop algorithmic techniques for performing these measurements. We discover that the scale-free properties permeate all the components of the bow-tie which exhibit the same macroscopic properties as the Web graph itself. However, close inspection reveals that their inner structure is quite distinct. We show that the Web graph does not exhibit self similarity within its components, and we propose a possible alternative picture for the Web graph, as it emerges from our experiments.

1. INTRODUCTION

In the past decade the world has witnessed the explosion of the World Wide Web from an information repository of a few millions of hyperlinked documents into a massive world-wide “organism” that serves informational, transactional, and communication needs of people all over the globe. Naturally, the Web has attracted the interest of the scientific community, and it has been the subject of intensive research work in various disciplines. One particularly interesting line of research is devoted to analyzing the structural properties of the Web, that is, understanding the structure of the *Web graph* [4, 15, 1].

The Web graph is the directed graph induced by the hyperlinks of the Web: the nodes are the (static) HTML pages, and the edges are the hyperlinks between them, directed from the page that contains the link to the target of the link. Understanding the structure and the evolution of the Web graph is a fascinating problem for the community of theoretical computer science. At the same time it has many practical implications. Knowledge of the Web structure can be used to devise better crawling strategies [17], perform clustering and classification [15], improve browsing [5]. Furthermore, it can help in improving the performance of search engines, one of the major driving forces in the development of the Web. The celebrated HITS [13] and PageRank [3] algorithms rely on the link structure of the Web to produce improved rankings of the search results. The knowledge of the macroscopic structure of the Web has been used in devising efficient algorithms for the computation of PageRank [12, 10].

The first large-scale study of the Web graph was performed by Broder et al. [4] and it revealed that the Web graph contains a giant component that consists of three distinct components of almost equal size: the CORE, made up of a single strongly connected component; the IN set, comprised by nodes that can reach the CORE but cannot be reached by it; the OUT set, consisting of nodes that can be reached by the CORE but cannot reach it. These three components form the well known *bow-tie* structure of the Web graph, shown in Figure 1[1].

The bow-tie picture describes the macroscopic structure of the

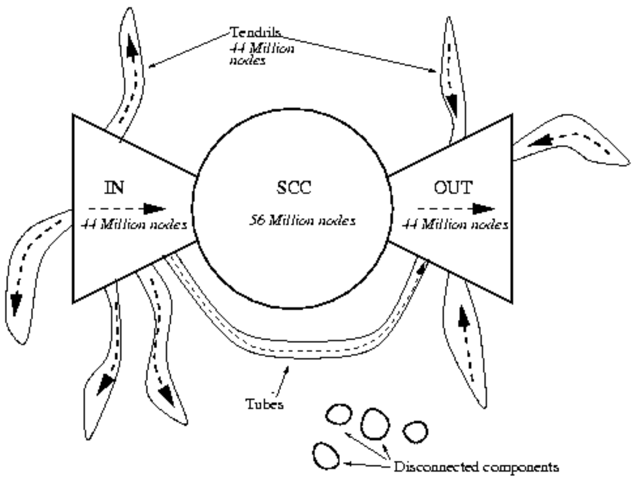


Figure 1: The bow-tie structure of the Web graph

[*]Partially supported by the EU under contract 001907 (DELIS) and 33555 (COSIN), and by the Italian MIUR under contract ALINWEB.

Eighth International Workshop on the Web and Databases (WebDB 2005), June 16-17, 2005, Baltimore, Maryland.

[1]The figure is reproduced from [4]

Web. However, very little is known about the inner structure of the components that comprise it. Broder et al. [4] pose it as an open problem to study further the structure of those components. Understanding the finer details of the Web graph is an interesting problem on its own, but it is also important in practice for improving the performance of algorithms that rely on the link structure of the Web. Furthermore, it could be useful for refining the existing stochastic models for the Web [1, 18, 14].

The study of the Web graph poses additional challenges. Typically, the Web graph consists of millions of nodes and billions of edges. Performing standard graph algorithms (such as BFS and DFS) on a graph of this size is a non-trivial task since data cannot be stored in main memory. It is therefore necessary to devise external-memory algorithms [6] that can work on massive graphs. The challenge is to customize the algorithms to the Web graph, taking advantage of the specific structure of the Web.

In this paper we study the finer structure of the Web graph, addressing the open question raised by Broder et al. [4]. We refine the bow-tie picture by providing details for its individual components. In the process we develop a suite of algorithms for handling massive graphs. Our contributions can be summarized as follows.

- We implement a number of external and semi-external memory graph theoretic algorithms for handling massive graphs, which can run on computers with limited resources. Our algorithms have the distinct feature that they exploit the structure of the Web in order to improve their performance.

- We experiment with four different crawls and we observe the same macroscopic properties previously reported in the literature: the degree distributions follow a power-law, and the graph has a bow-tie structure, although (depending on the crawler) a little different in shape.

- We study in detail the inner structure of the bow-tie graph. We perform a series of measurements on the CORE, IN and OUT components. Our measurements reveal the following surprising fact: although the individual components share the same macroscopic statistics with the whole Web graph, they have substantially different structure. We suggest a refinement of the bow-tie picture, the *daisy structure* of the Web graph, that takes our findings into account.

The rest of the paper is structured as follows. In Section 2 we review some of the basic graph theoretic definitions, and some of the previous work. In Section 3 we outline the algorithms for handling the Web graph. in Section 4 we present our experimental findings. We conclude in Section 5 with a discussion on the implications of our findings, and possible future experiments.

2. BACKGROUND

Graphs and Power Laws: We will be using various basic graph theoretic definitions and algorithms that can be found in any graph theory textbook (e.g., [7]). Here, we only remind the reader of the definitions of strongly and weakly connected components.

A set of nodes S forms a *strongly connected component (SCC)* in a directed graph, if and only if for every pair of vertices $u, v \in S$, there exists a path from u to v, *and* from v to u. A set of nodes S forms a *weakly connected component (WCC)* in a directed graph G, if and only if the set S is a connected component of the undirected graph G_u that is obtained by removing the directionality of the edges in G.

We will often talk about power-law distributions which are characteristic of the Web. A discrete random variable X follows a power law distribution if the probability of taking value i is $P[X = i] \propto 1/i^{\gamma}$, for a constant $\gamma \geq 0$. The value γ is the exponent of the power-law.

Related Work: The study of the structure of the Web graph has recently been the subject of a large body of literature. A well-documented characteristic of the Web graph is the ubiquitous presence of power law distributions. Kleinberg et al. [14] and Barabasi and Albert [1] demonstrated that the in-degree of the Web graph follows a *power-law* distribution. Later experiments by Broder et al. [4] on a crawl of 200M pages from 1999 by Altavista confirmed it as a basic property: the in-degree of a vertex is distributed according to a power-law with exponent $\gamma \approx 2.1$. The sizes of the SCC components also follow a power-law. The out-degree distribution follows an imperfect power law distribution.

Broder et. al. [4] studied also the structure of the Web graph, and presented the bow-tie picture. They decomposed the Web graph into the following components (Figure 1): the CORE, consisting of the largest SCC in the graph; the IN, consisting of nodes that can reach the CORE; the OUT, consisting of nodes that are reachable from the CORE; the TENDRILS, consisting of nodes not in the CORE that are reachable from the nodes in IN, or can reach the nodes in OUT; the DISC, consisting of the remaining nodes.

Dill et al. [9] demonstrated that the Web exhibits self-similarity when considering "Thematically Unified Clusters" (TUCs), that is, sets of pages that are brought together due to some common trait. Thus the Web graph can be viewed as the outcome of a number of similar and independent stochastic processes. Pennock et al. [18] also argue that the Web is the sum of stochastic independent processes that share a common (fractal) structure.

The findings about the structure of the Web generated a flurry of research in the field of random graphs. Given that the standard graph theoretic model of Erdös and Rèny [11] is not sufficient to capture the generation of the Web graph, various stochastic models were proposed [1, 18, 14]. Most of them address the fact that the in-degrees must follow a power-law distribution [1]. The *copying model* [14] generates graphs with multiple bipartite cliques [15].

3. ALGORITHMIC TECHNIQUES FOR HANDLING THE WEB GRAPH

This study has required the development of a complete algorithmic methodology for handling very large Web graphs. As a first step we need to identify the individual components of the Web graph. For this we need to be able to perform graph traversals. The link structure of the Web graph takes several gigabytes of disk space, making it prohibitive to use traditional graph algorithms designed to work in main memory. Therefore, we implemented algorithms that achieve remarkable performance improvements when processing data that are stored on external memory. We implemented *semi-external* algorithms, that use only a small constant amount of memory for each node of the graph, as well as *fully-external* algorithms that use an amount of main memory that is independent of the graph size.

We implemented the following algorithms.

- A semi-external graph traversal for determining vertex reachability using only 2 bits per node. The one bit is set when the node is first visited, and the other when all its neighbors have been visited (we say that the node is "completed"). The algorithm operates on the principle that the order in which the vertices are visited is not important. Starting from an initial set of nodes, it performs multiple passes over the data, each time visiting the neighbors of the non-completed nodes.

- A semi-external Breadth First Search that computes blocks of reachable nodes and splits them up in layers according to their distance from the root. In a second step, these layers are sorted to produce the standard BFS traversal of the graph.

- A semi-external Depth First Search (DFS) that needs 12 bytes plus one bit for each node in the graph. This traversal has been developed following the approach suggested by Sibeyn et al. [19].

- An algorithm for computing the largest SCC of the Web graph. The algorithm exploits the fact that the largest SCC is a sizable fraction of the Web graph. Thus, by sampling a few nodes of the graph, we can obtain a node of the largest SCC with high probability. We can then identify the nodes of the SCC using the reachability algorithm. As an end product we obtain the bow-tie regions of the Web graph, and we are able to compute all the remaining SCCs of the graph efficiently using the semi-external DFS algorithm.

A software library containing a suite of algorithms for generating and processing massive Web graphs is available online[2]. A detailed presentation of some of these algorithms and a study of their efficiency has been presented in [16]. A complete description of these algorithms is available in the extended version of this work [8].

4. EXPERIMENTS AND RESULTS

We experiment with four different crawls. The first three crawls are samples from the Italian Web (the `.it` domain), the Indochina Web (the `.vn, .kh, .la, .mm,` and `.th` domains), and the UK Web (the `.uk` domain) collected by the "Language Observatory Project" [3] and the "Istituto di Informatica e Telematica" [4] using UbiCrawler [2]. The fourth crawl is a sample of the whole Web, collected by the WebBase project at Stanford[5] in 2001. This sample contains 360 millions of nodes and 1.5 billion of edges. In order to eliminate non-significant data, we pruned the frontier nodes (i.e. the nodes with in-degree 1 and out-degree 0, on which the crawler has been arrested). The sizes of the crawls are shown in Table 1.

4.1 Macroscopic measurements

As a first step in our analysis of the Web graph, we repeat the experiments of Broder et al. [4] on the macroscopic analysis of the graph. We computed the in-degree, out-degree and SCC size distributions. As expected, the in-degrees, and the sizes of SCCs follow a power-law distribution, while the out-degree distribution follows an imperfect power-law. All our measurements are in agreement with the respective measurements of Broder et al. [4] for the AltaVista crawl. More detailed results on the various distributions for the WebBase crawl are reported in [16].

We also computed the macroscopic structure of the Web graph. We observe a bow-tie structure. The relative sizes of the components of the bow-tie are shown in Table 1, where we also present the numbers for the AltaVista crawl [4], for the purpose of comparison. The first observation is that for the Italian, Indochina, and UK crawls, the IN and TENDRILS components are almost non-existent. As a result either the CORE is overgrown (for the Italian and UK crawls), or the nodes are equally distributed between the CORE and the OUT component. For the WebBase crawl we observe that the relative size of IN (11%) is significantly smaller than that observed in the AltaVista crawl, while the OUT component (39%) is now the largest component of the bow-tie. These discrepancies with the AltaVista crawl can most likely be attributed to different crawling strategies and capabilities, rather than to the evolution of the Web. The first three crawls are relatively recent, and all crawls are generated using a small number of starting points. Unfortunately, large-scale crawls are not publicly available.

4.2 The inner structure of the bow-tie graph

We now study the fine-grained structure of the Web graph. We are interested in understanding not only the characteristics of each component individually, but also how the components relate to each other. For this purpose we label each node with the name of the component to which it belongs. This gives us five sets of nodes (CORE, IN, OUT, TENDRILS, DISC). For each such subset we obtain the induced subgraph, resulting in five different subgraphs. For example, when referring to the IN graph, we mean the graph that consists of the nodes in IN and all the edges between these nodes.

As a first step in the understanding of the individual components we compute the same macroscopic measures as for the whole Web graph. We compute the in-degree, out-degree and SCC size distributions for each of the IN, OUT, TENDRILS and DISC graphs. Figure 2 shows the plots of the distributions for each component and for the whole graph, for the case of the WebBase crawl. It is obvious that the same macroscopic laws that are observed on the whole graph are also present in the individual components.

4.2.1 The structure of the IN and OUT components

Given the fact that the in-degree, out-degree, and SCC size distributions in the IN and OUT components are the same as for the whole Web graph, it is tempting to conjecture that the Web has a *self-similar* structure. That is, the bow-tie structure repeats itself inside the IN and OUT components. Dill et al. [9] demonstrated that the web exhibits self-similarity when considering "thematically unified" sets of web pages. These subsets are structurally similar to the whole Web. Similar observations are made by Pennock et al. [18]. However, the subsets considered by these previous works are composed by nodes that may belong to any of the components of the bow-tie graph. The question we are interested in is if such self-similarity appears when considering the individual components of the bow-tie graph.

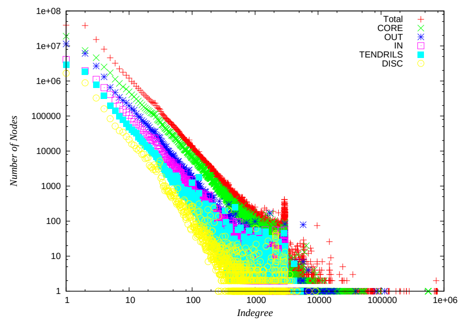
The first indication that the self-similarity conjecture is not true comes from the fact that there is no large SCC in the IN and OUT components. For the OUT component, in all crawls, the largest SCC is only a few thousands of nodes. Given that the size of the OUT component is in the order of millions, the largest SCC is staggeringly small. Furthermore, this is also the second largest SCC in the graph, which, compared the largest one (the CORE), is minuscule. We observe a similar phenomenon for the IN component. For the WebBase graph (which is the most interesting case, since the IN component is a non-trivial fraction of the graph) the largest SCC in the IN component is less than 6,000 nodes. Detailed numbers about the size of the largest SCC in the IN and OUT components are given in Table 2.

Therefore, it appears that there exists no sizable SCC in the IN and OUT components that could play the role of the CORE in a potential bow-tie. However it is still possible that there exists a giant weakly connected component (WCC) in each component. We therefore computed the WCCs of the two sets. Surprisingly we discovered that there is no giant WCC in either of the two components. In fact, there is a large number of WCCs per component and their sizes follow a power law distribution. Figure 3(a) shows

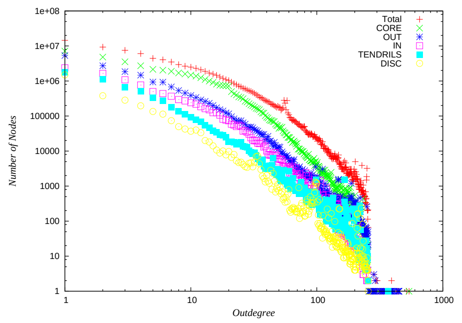
[2]http://www.dis.uniroma1.it/∼cosin/
[3]www.language-observatory.org
[4]www.itt.cnr.it
[5]http://www-diglib.stanford.edu/ testbed/doc2/WebBase/

	Italy	Indochina	UK	WebBase	AltaVista
nodes	41.3M	7.4M	18.5M	135.7M	203.5M
edges	1.15G	194.1M	298.1M	1.18G	1.46G
CORE	29.8M (72.3%)	3.8M (51.4%)	1.2M (65.3%)	44.7M (32.9%)	56.4 (27.7%)
IN	13.8K (0.03%)	48.5K (0.66%)	312.6K (1.7%)	14.4M (10.6%)	43.3 (21.3%)
OUT	11.4M (27.6%)	3.4M (45.9%)	5.9M (31.8%)	53.3M (39.3%)	43.1 (21.2%)
TENDRILS	6.4K (0.01%)	50.4K (0.66%)	139.4K (0.8%)	17.1M (12.6%)	43.8 (21.5%)
DISC	1.25K (0%)	101.1K (1.4%)	80.2K(0.4%)	6.2M (4.6%)	16.7 (8.2%)

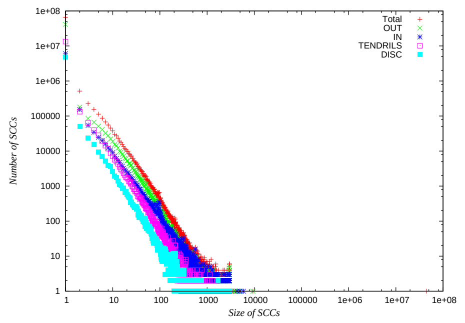
Table 1: Sizes and bow-tie components for the different crawls and the Alta Vista graph



(a) In-degree distributions



(b) Out-degree distributions



(c) SCC size distributions

Figure 2: Macroscopic measures for all components

the WCC size distribution for the WebBase graph. Statistics for all graphs are reported in Table 2. Most of the WCCs are of size one. The singleton WCCs comprise 10-22% of the IN component (with the exception of Indochina), and 20-45% of the OUT component. On the other hand, the largest WCC is never more than 30% of the component it belongs to, which is small compared to the giant WCC in the Web graph, which contains more than 90% of the nodes. For the WebBase graph, the largest WCC in the IN component consists of just 1% of the nodes, while the largest WCC in the OUT component consists of 28% of the nodes.

We also investigate how the nodes in the largest WCCs in the IN and OUT components are connected to see if they organized in a bow-tie shape. Our investigation revealed that starting from the largest SCC in the WCC, we can create a bow-tie that is no more than 15% of the WCC (for the Italian Web), and usually less than 5%. The rest belongs to the DISC component. (Note that a node that points to the tendrils coming out of IN, or is pointed to by those going into OUT, belongs to DISC, although it is still weakly connected to the graph). This suggests that the WCC consists of multiple small atrophic bow-ties that are sparsely interconnected with each other.

	Italy	Indochina	UK	WebBase
depth IN	2	11	15	8
depth OUT	26	21	25	580

Table 3: IN and OUT depth

In order to better understand how the nodes in IN and OUT are arranged with respect to the CORE, we performed the following experiment. We condensed the CORE in a single node and we performed a forward and a backward BFS. This allows us to split the nodes in the IN and OUT components in *levels* depending on their distance from the CORE. The depths of the components are shown in Table 3. In all graphs, the depths of the components are relatively small. Furthermore, most nodes are concentrated close to the CORE. Typically, about 80-90% of the nodes in the OUT component are found within the first 5 layers. For the WebBase graph, although the OUT is much deeper, with 580 levels, more than 58% of its nodes are at distance 1 from the CORE, and 93% are within distance 5. Furthermore, after level 305 there exists only a single chain of nodes that extends until level 580, making the effective depth of the OUT 305. The node distributions, level by level, for the WebBase graph are shown in Figure 3(b) and 3(c), for the IN and OUT sets respectively. The plots are in logarithmic scale.

Therefore, we conclude that the IN and OUT components are shallow and highly fragmented. They are comprised of several sparse weakly connected components of low depth. Most of their volume consists of nodes that are directly linked to the CORE.

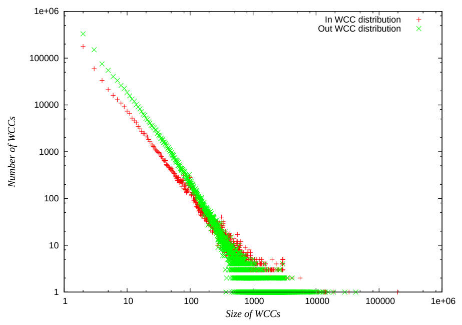
4.2.2 The structure of the CORE

As a first step in the study of the CORE graph, we examine its relation with the IN and OUT components. We define an *entry point* to the CORE to be a node that is pointed to by at least one node in the IN component, and an *exit point* to be a node that points to at least one node in the OUT component. A *bridge* is a node that is both an entry and an exit point. The number of entry and exit points is shown in Table 2. It is interesting to observe that a large fraction of the entry points act like bridges. Furthermore, with the exception of the UK crawl, the majority of the nodes in the CORE is connected to the "outside" world. In the WebBase crawl, this number is around 80% of the whole CORE, while the "deep CORE" consists of a little more than 20%.

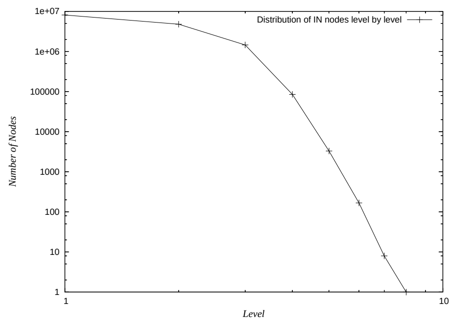
We also compute the in-degree distribution of the entry points when we restrict the source of the links to be in the IN component, and, as expected, we observe a power law. This implies that most nodes "serve" as entry points to just a few nodes in the IN component, while there exist a few nodes that serve as entry points to a large number of IN nodes. Similar distributions are obtained when we consider the out-degree distribution of the exit points, restricted

	Italy	Indochina	UK	WebBase
The IN component				
nodes in IN	13.8K (0.03%)	48.5K (0.66%)	312.6K (1.69%)	14.4M (11%)
max SCC	1,590	7,867	4,171	5,876
number of WCCs	1,633	117	62K	3.68M
max WCC	4,085 (29.5%)	13.2K (27.2%)	8,246 (2.7%)	197.5K (1.3%)
singleton WCCs	1,543 (11.15%)	63 (0.13%)	56K (17.89%)	3.2M (22.46 %)
The OUT component				
nodes in OUT	11.4M (27.6%)	3.4M (45.9%)	5.9M (31.8%)	53.3M (39%)
max SCC	19,170	39,283	26,525	9,349
number of WCCs	3.73M	729,6K	1.97M	25.4M
max WCC	1.43M (12.52%)	335.9K (9.85%)	457.4K (7.75%)	14.94M (28.01%)
singleton WCCs	3.49M (30.6%)	672K (19.71%)	1.84M (31.11%)	24.48M (45.91%)
The CORE component				
nodes in CORE	29.8M (72.3%)	3.8M (51.4%)	1.2M (65.28%)	44.7M (33%)
entry points	10.2K (0.03%)	2.3K (0.06%)	106.3K (0.88%)	2.6M (5,87%)
exit points	15.6M (52.2%)	2.3M (59.6%)	4.8M (39.8%)	29.6M (72.03%)
bridges	6.25K(0.02%)	1.5K (0.04%)	61.8K (0.51%)	2M (4.58%)
connectors	1.7M (5.71%)	164.2K (4.32%)	537.9K (4.45%)	2.96M (6.63%)
petals	325.3K (1.09%)	52.5K (1.38%)	138K (1.14%)	1.4M (3.14%)

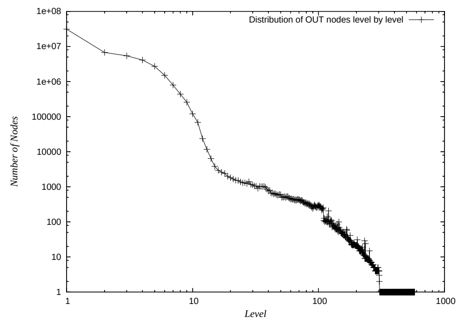
Table 2: Statistics for the IN, OUT and CORE components for each crawl



(a) Distribution of WCC sizes per component



(b) Distribution of IN nodes level by level



(c) Distribution of OUT nodes level by level

Figure 3: Characteristics of the IN and OUT components

to the OUT component.

We then study the connectivity of the CORE. We first look for nodes that are loosely connected to the CORE. We define a *connector* to be a node of the CORE that has a single in-coming and out-going link. A connector forms a *petal* if the source of the in-coming link, and the target of the out-going link are the same node. Large number of connectors would imply weak connectivity of the CORE. The number of connectors is shown in Table 2, and it is on average around 5%. Of these 20 to 45% are petals. Therefore, connectors are only a small part of the CORE.

In order to further understand the connectivity of the CORE, we test the resilience of the CORE to targeted attacks by performing the following experiment. For some k we delete all nodes from the CORE that have total degree (in-degree plus out-degree) at least k. We then compute the size of the largest SCC in the resulting graph. Table 4(a) shows how the size of the largest SCC changes as we decrease k, and we increase the number of deleted nodes for the case of the WebBase graph. Similar trends are observed in the other crawls. We observe that the threshold on the total degree must become as low as 100 in order to obtain an SCC of size less than 50% of the CORE.

We note that there is a large discrepancy between the values of the in-degrees and out-degrees in the Web graph. The highest in-degree is close to 566K, while the highest out-degree is just 536. Note that an upper-bound on the out-degree may be imposed by the crawler, if it limits the number of outgoing links of a page that it explores. Therefore, it may be the case that when deleting the nodes with high total degree, we only delete nodes with high in-degree. We experiment with a different kind of attack that removes (approximately) k nodes with the highest in-degree and k nodes with the highest out-degree. The results are shown in Table 4(b). The CORE remains resilient even against this combined attack. An interesting observation while performing this experiment was that the nodes with the highest in-degree and the nodes with the highest out-degree are quite distinct. Actually, the correlation between the in-degree and out-degree is close to zero. It appears that nodes that are strong hubs in the CORE are not also strong authorities.

There are two ways to interpret these results. The first is that there are no obvious *failure points* in the CORE, that is, strong hubs or authorities that pull the rest of the nodes together, and whose removal from the graph causes the immediate collapse of the network. In order to disconnect the CORE you need to remove nodes with sufficiently low degree. On the other hand, note that we managed to reduce the largest SCC to 35-40% of the original by removing about 1M nodes. However this is less than 1% of the total nodes. In that sense the CORE is vulnerable to targeted attacks.

deg	del	max SCC	max SCC %	SCC num
50,000	9	44.2M	98.9%	81K
21,500	39	43.7M	97.9%	175K
10,000	199	43.2M	96.6%	285K
4,000	1.1K	42.3M	94.7%	505K
1,000	55K	35.1M	78.6%	3.7M
500	120K	31M	69.6%	5.7M
100	1.03M	14.8M	34.6%	14.7M

(a) Deleting nodes with high total degree

in-deg	del	out-deg	del	total del	max SCC	max SCC %	SCC num
4,000	1.1K	233	1,154	2,263	42.2M	94.4%	595K
2,600	9.9K	185	10K	20.6K	39.8M	89.0%	1.75M
1,750	26K	158	25K	51K	37M	82.9%	3M
1,000	52K	130	54K	105K	33.7M	75.5%	4.75M
500	112K	105	108K	219K	29.4M	66.1%	7M
225	259K	82	227K	487K	23.5M	53.3%	10M
120	518K	62	499K	949K	17.8M	40.8%	13M

(b)Deleting nodes with high in-degree and out-degree

Table 4: Sensitivity of the CORE under targeted attacks

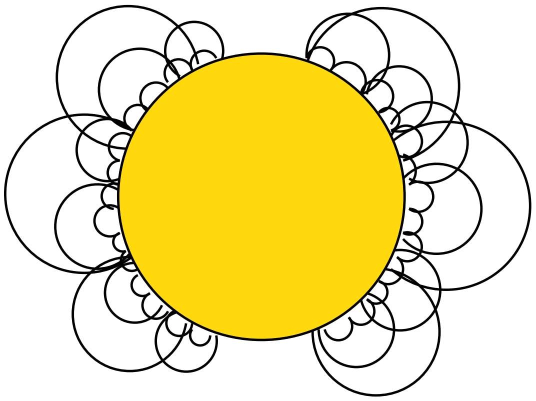


Figure 4: The daisy structure of the Web

5. DISCUSSION AND FUTURE WORK

In this paper we undertook a study of the Web graph at a finer level. We observed that the ubiquitous presence of power laws describing several properties at a macroscopic level does not necessarily imply self-similarity in the individual components of the Web graph. Indeed, the different components have quite distinct structure, with the IN and OUT being highly fragmented, while the CORE being well interconnected.

Our work suggests a refinement of the bow-tie pictorial view of the Web graph [4]. The bow-tie picture seems too coarse to describe the details of the Web. The picture that emerges from our work can better be described by the shape of a *daisy* (Figure 4): the IN and OUT regions are fragmented into large number of small and shallow *petals* (the WCCs) hanging from the central dense CORE.

It would be interesting to obtain larger, and more "realistic" crawls, and perform the same measurements to verify our hypothesis. Our current results are sensitive to the choices and limitations of the crawlers, and it is not clear if the available crawls are representative of the actual Web graph. Unfortunately, there are no publicly available crawls that have been collected with the aim of validating our hypothesis on the structure of the Web graph. We plan in the future to collect crawls with this goal in mind.

A deeper understanding of the structure of the Web graph may also have several consequences on designing more efficient crawling strategies. The fact that IN and OUT are highly fragmented may help in splitting the load between different robots without much overlapping. Moreover, the fact that most of the vertices are at few hops from the CORE may explain why breadth first search crawling is more effective than other crawling strategies [17].

Our work motivates further experiments on the Web graph. It would be interesting to devise efficient algorithms for estimating the clustering coefficient, a commonly used measure for connectivity. Furthermore, further exploration of the structure of the CORE is necessary to gain a deeper understanding. Possible measurements could include spectral properties, or clustering and community discovery. As a concluding remark, we observe that we are still very far from devising a theoretical model that is able to capture the finer connectivity properties of the Web graph.

6. REFERENCES

[1] A.L. Barabasi and A. Albert. Emergence of scaling in random networks. *Science*, (286):509–512, 1999.

[2] P. Boldi, B. Codenotti, M. Santini, and S. Vigna. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice & Experience*, 34(8):711–726, 2004.

[3] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. In *WWW*, 1998.

[4] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, S. Stata, A. Tomkins, and J. Wiener. Graph structure in the web. *Computer Networks*, 33:309–320, 2000.

[5] J. Carrìere and R. Kazman. Webquery: Searching and visualizing the web through connectivity. In *6th WWW Conference*, 1997.

[6] Y. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *SODA*, 1995.

[7] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms*. MIT Press and McGraw-Hill, 1992.

[8] S. Millozzi D. Donato, S. Leonardi and P. Tsaparas. Mining the inner structure of the web graph. Technical report, DELIS-TR-157, http://delis.upb.de/docs/, 2005.

[9] S. Dill, R. Kumar, K. McCurley, S. Rajagopalan, D. Sivakumar, and A. Tomkins. Self-similarity in the web. In *Proceedings of the 27th VLDB Conference*, 2001.

[10] N. Eiron, K. S. McCurley, and J. A. Tomlin. Ranking the web frontier. In *WWW*, 2004.

[11] P. Erdös and A. Rèny. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci.*, 5:17–61, 1960.

[12] S. Kamvar, T. Haveliwala, C. Manning, and G. Golub. Exploiting the block structure of the web for computing pagerank. Technical report, Stanford University, 2003.

[13] J. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5):604–632, 1997.

[14] J. Kleinberg, R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. The web as a graph: measurements, models and methods. In *Proc. Intl.Conf. on Combinatorics and Computing*, 1999.

[15] R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. Trawling the web for emerging cyber communities. In *WWW*, 1999.

[16] L. Laura, S. Leonardi, S. Millozzi, U. Meyer, and J.F. Sibeyn. Algorithms and experiments for the webgraph. In *European Symposium on Algorithms (ESA)*, 2002.

[17] M. Najork and J. L. Wiener. Breadth-fi rst crawling yields high-quality pages. In *WWW Conference*, 2001.

[18] D.M. Pennock, G.W. Flake, S. Lawrence, E.J. Glover, and C.L. Giles. Winners don't take all: Characterizing the competition for links on the web. *Proc. of the National Academy of Sciences*, 99(8):5207–5211, April 2002.

[19] J.F. Sibeyn, J. Abello, and U. Meyer. Heuristics for semi-external depth fi rst search on directed graphs. In *SPAA*, 2002.

Managing Multiversion Documents & Historical Databases: a Unified Solution Based on XML

Fusheng Wang
Siemens Corporate Research
fusheng.wang@siemens.com

Carlo Zaniolo   Xin Zhou   Hyun J. Moon
University of California, Los Angeles
{zaniolo, xinzhou, hjmoon}@cs.ucla.edu

ABSTRACT

XML can provide a very effective environment for the preservation of digital information whereby historical information can be easily preserved and searched through powerful historical queries. We propose a unified approach to represent multiversion XML documents and transaction-time databases in XML, and show that temporal queries can then be expressed in standard XQuery. In our demo we demonstrate the benefits of this approach on several examples, including the UCLA course catalog, W3C XLink standards, the CIA WorldFact Book, and a database of company employees. We will also demonstrate the ICAP and ArchIS system that have explored two alternative implementation architectures, one based on native XML DBMS, and the other on mapping the historical XML views back into a relational DBMS.

1. INTRODUCTION

Preservation of digital artifacts represents a critical issue for our web-based society [10, 14]. Web documents are frequently revised, and this creates the problem of how to effectively organize, search, and query multiversion documents. When presented with multiversion documents, users will want to pose queries on the evolution history of the documents and their contents, in addition to searching and retrieving specific versions of such documents. Users would also like to view and query the history of the content of a relational database in a similar fashion.

Our demo will illustrate that the technology is at hand for supporting the preservation and queries of multiversion documents, since XML and its query languages can manage and search effectively the history of relational databases and XML documents. Our approach consists in viewing their history as one integrated document, whereby each element is timestamped with its period of validity, by its `vstart` and `vend` attributes. The advantage of this approach is that it makes it possible to support powerful historical queries using standard XQuery.

Eighth International Workshop on the Web and Databases (WebDB 2005), June 16-17, 2005, Baltimore, Maryland.

2. VERSION MANAGEMENT IN XML

Our demo will cover three case studies on evolving XML documents and several examples involving the transaction-time history of relational databases.

A first case study is the UCLA course catalog that is published anew every two years. By integrating successive versions into a time-stamped history, we can express basic snapshot queries such as “Find all the graduate courses offered by the History Department in 1998”, and “When was course CS143 introduced?”. But we can also pose more complex queries, such as “Show me the growth of the courses offered by the CS department in the last 20 years (a temporal aggregate query)”, or “How many years did it take for ‘nanotechnology’ topics to migrate from graduate-level courses to undergraduate ones?”.

A second interesting example involves the successive versions of the W3C XLink standards [4] that are regularly published as new XML documents. Temporal queries bring out changes between the successive versions and also changes on meta-level information. For instance, a query that proved to have unexpected ramifications is “Where authors ever dropped from one version of XLink to the next?”.

While XLink and the UCLA course catalog were originally created as XML documents, the CIA World FactBook [3] was instead converted from HTML. This is an interesting publication that describes the economy, government, population, resources, etc., of every country in the world. A new version of the WorldFact Book is published each year; this creates an opportunity to search the political/economic history of the globe, and ask very specific queries on the evolution of particular countries and regions, using XML.

These case studies reveal that an integrated history document can lead to interesting findings and unexpected discoveries that would have been difficult to obtain from the separate versions of the document. These examples will be discussed in the course of our demo.

Historical queries represent an excellent example of the ability of digital libraries to enhance content delivery services well beyond those provided by traditional libraries. Our ICAP system also supports services, such as color-marking the changes between any two versions of the document (not just successive ones) and advanced features inspired by the version machine [9].

3. HISTORICAL QUERIES

An important advantage of our approach is that it can be applied to both XML documents and relational databases, and it requires no change in existing standards. This is significant, given that support for temporal information and historical queries proved to be difficult in standard SQL. Such difficulties led to a number of proposed temporal extensions to SQL [16]; but these have not been incorporated into commercial DBMS. However, historical information can be managed and queried in XML, without requiring extensions to the current standards, since:

- XML provides a richer data model, whose structured hierarchies can be used to support temporally grouped data models by simply adding temporal attributes to the elements. Temporally grouped representations have long been recognized to provide a very natural data model for historical information [11, 12], and
- XML provides more powerful query languages, such as XQuery, that achieves native extensibility and Turing completeness via user-defined functions [15]. Thus, the constructs needed for temporal queries can be introduced as user-defined libraries, without requiring extensions to existing standards.

In the ICAP project [1], we have defined a temporal library of XQuery functions to facilitate the formulation of historical queries and isolate the user from lower-level details, such as the internal representation of the 'now' timestamp. The complete gamut of historical queries—including snapshot and time-slicing queries, element-history queries, *since* and *until* queries—can be expressed in standard XQuery [1].

4. MULTIVERSION XML DOCUMENTS

In the ICAP project [1],

(i) we use structured diff algorithms [6, 13] to compute the validity periods of the elements in the document,
(ii) we use the output generated by the diff algorithm, to represent concisely the history of the documents with a temporally grouped data model. Then,
(iii) we use XQuery, enhanced with the library of temporal functions discussed above, to formulate temporal queries on the evolution of these documents and their contents.

For instance consider a very simple document in three successive versions:

```
<document>                <!--This is version 1 -->
    <chapter no="1">
        <title>Introduction</title>
        <section>Background</section>
        <section>Motivation</section>
    </chapter>
</document>
<document>                <!--This is version 2 -->
    <chapter no="1">
        <title>Overview</title>
        <section>Background</section>
        <section>History</section>
    </chapter>
    <chapter no="2">
        <title>Related Work</title>
        <section>XML Storage</section>
    </chapter>
</document>
<document>                <!--This is version 3-->
    <chapter no="1">
        <title>Overview</title>
        <section>Background</section>
        <section>History</section>
    </chapter>
    <chapter no="2">
        <title>Related Work</title>
        <section>XML Indexing</section>
    </chapter>
</document>
```

To store and query efficiently the history of this evolving document, we compute the differences between its successive versions, using a structure diff algorithm such as those described in [6, 13]. Then, we represent the history of the document by time-stamping and temporally grouping these deltas as shown below. We call this history-grouped document *V-Document.*

```
<document vstart="2002-01-01" vend="now">
  <chapter vstart="2002-01-01" vend="now">
      <no isAttr="yes"vstart="2002-01-01"
                    vend="now">1</no>
      <title vstart="2002-01-01"
             vend="2002-01-01">Introduction</title>
      <title vstart="2002-01-02"
                    vend="now">Overview</title>
      <section vstart="2002-01-01"
                  vend="now">Background</section>
      <section vstart="2002-01-01"
             vend="2002-01-01">Motivation</section>
      <section vstart="2002-01-02"
                    vend="now">History</section>
  </chapter>
  <chapter vstart="2002-01-02" vend="now">
      <no isAttr="yes" vstart="2002-01-02"
                              vend="now">2</no>
      <title vstart="2002-01-02"
                  vend="now">Related Work</title>
      <section vstart="2002-01-02"
             vend="2002-01-02">XML Storage</section>
      <section vstart="2002-01-03"
                 vend="now">XML Indexing</section>
  </chapter>
</document>
```

We obtain a temporally grouped representation (similar to that of SCCS [17]) that can be easily queried using XQuery.

4.1 Complex Queries

Using XQuery [5], complex temporal queries on V-Documents can be easily expressed as described next.

QUERY 1. Evolutionary queries: find the history of titles for Chapter 1:

```
for $title in
doc("V-Document.xml")/document/chapter[no="1"]/title
return $title
```

This query returns the list of the titles of chapter 1, each with the time periods in which those titles were used. Thus for the example at hand it will return:

```
<title vstart="2002-01-01"
      vend="2002-01-01">Introduction</title>
<title vstart="2002-01-02"
      vend="now">Overview</title>
```

The next query shows an example of duration query.

QUERY 2. Duration queries: find titles that didn't change for more than 2 consecutive years.

```
for $title in doc("V-Document.xml")/document/chapter/title
let $dur:=substract-dates($title/@vend, $title/@vstart)
where dayTimeDuration-greater-than($dur, "P730D")
return  $title
```

The next two examples demonstrate how the connectives **since** and **until** from first-order temporal logic can be expressed using XQuery on V-Documents:

QUERY 3. A Since B: find chapters whose "History" sections remain unchanged since the version when the title was changed to "Introduction and Overview".

```
for $ch in doc("V-Document.xml") /document/chapter
let $title := $ch/title[.="Introduction and Overview"]
let $sec := $ch/section[.="History"]
where not empty($title) and not empty($sec)
 and $sec/@vstart = $title/@vend and $sec/@vend ="now"
return  $ch
```

QUERY 4. A Until B: find all chapters whose titles have not changed until a new section "History" was added.

```
for $ch in doc("V-Document.xml")/document/chapter
let $title := $ch/title[1]
let $sec := $ch/section[.="History"]
where not empty($title) and not empty($sec)
 and  $title/@vend = $sec/@vstart
return  $ch
```

ICAP provides several temporal functions, including a function called *snapshot($node, $versionTS)* that only returns the element and its descendants where $vstart \leq$ versionTS $\leq vend$.

QUERY 5. Snapshot queries: retrieve the version of the document on 2002-01-03:

```
for $e in doc("V-Document.xml")/document
return snapshot( $e,"2002-01-03")
```

Other functions written in XQuery hide the user from the implementation details of 'now'. There is also a recursive *diff-identical($node, $version1TS, $version2TS)* function that returns the elements that have changed between $version1TS$ and $version2TS$. This function can, e.g., be used to show and/or color-code the differences between two arbitrary document versions.

QUERY 6. Change queries: retrieve the elements that have changed from 2002-01-01 to 2002-01-03:

```
for $e in doc("V-Document.xml")/document
return alldiff($e,"2002-01-01","2002-01-03")
```

5. ARCHITECTURE & PERFORMANCE

In the ICAP system, history documents are stored and managed using a native XML systems: in our current implementation we use Tamino [7] and X-Hive [8]. In all three case-study examined (i.e, UCLA course catalog, W3C XLink, and the CIA World FactBook), we are dealing with sizes that do not exceed a few megabytes, and can be effectively supported by native XML DBMS. However, the performance of this approach can be unsatisfactory when dealing with the history of database relations. The current contents of typical databases can exceed the gigabyte size—and the size of their transaction time history collected over several years can be significantly larger. These sizes are easily handled by relational database systems whose users have also learned to expect that their DBMS performs well for large data sets. For instance, users of transaction time databases are likely to require that their queries on past snapshots are not much slower than those on their current databases. Our tests [18] indicate that native XML databases do not scale up to this task, and their performance are likely to disappoint the users of relational databases. As described in [18], these performance problems can effectively be addressed by using a relational DBMS to support historical (virtual) XML views and queries on these views. In our ArchIS system, therefore, we decompose the historical views into individual tables that contain the history of each attribute in the relation. Then, we transform and execute the XQuery statements expressed against the views into equivalent SQL/XML statements against the stored tables. This approach assures a satisfactory level of performance [18].

6. TESTBED AND DEMO

Several interesting test cases will be demonstrated in our demo, including the UCLA course catalog, W3C XLink standards [4], the CIA World FactBook [3], and a database of company employees. These examples reveal that V-Documents often lead to interesting findings that cannot be easily inferred from the snapshots of the original documents. Alternative system architectures and their performance [18] will also be covered in the demo.

Acknowledgment

The authors would like to thank Bertram Ludäscher and Richard Marciano for illuminating discussions on the preservation of documents.

7. REFERENCES

[1] The ICAP Project. http://wis.cs.ucla.edu/projects/icap/.
[2] UCLA Catalog. http://www.registrar.ucla.edu/catalog/.
[3] CIA: *The World Factbook.* http://www.cia.gov/cia/publications/factbook/
[4] XML Linking Language (XLink). http://www.w3.org/TR/XLink/.
[5] XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/.
[6] Microsoft XML Diff. http://apps.gotdotnet.com/xmltools/xmldiff/.
[7] Software AG: Tamino XML Server, http://www.softwareag.com/tamino.
[8] X-Hive/DB. http://www.x-hive.com.
[9] The Versioning Machine. http://mith2.umd.edu/products/ver-mach/
[10] Library of Congress. Displays for Multiple Versions from MARC 21 and FRBR. http://www.loc.gov/marc/marc-functional-analysis/multiple-versions.html
[11] J. Clifford. *"Formal Semantics and Pragmatics for Natural Language Querying"*. Cambridge University Press, 1990.
[12] J. Clifford, A. Croker, F. Grandi,and A. Tuzhilin, *"On Temporal Grouping"*, in Proc. of the Intl. Workshop on Temporal Databases, 1995.
[13] Gregory Cobena, Serge Abiteboul, Amelie Marian, *"Detecting Changes in XML Documents"*, in ICDE 2002.
[14] A.R. Kenney, et. al., *"Preservation Risk Management for Web Resources Virtual Remote Control in Cornell's Project Prism*, *D-Lib Magazine*, Jan 2002, 8(1).
[15] S. Kepser. *"A Simple Proof for the Turing-Completeness of XSLT and XQuery"*. In *Extreme Markup Languages*, 2004.
[16] G. Ozsoyoglu and R.T. Snodgrass, *"Temporal and real-time databases: A survey"*. in TKDE, 7(4):513–532, 1995.
[17] M. J. Rochkind, *"The Source Code Control System"*, IEEE Transactions on Software Engineering, SE-1, 4, Dec. 1975, p. 364-370.
[18] F. Wang, X. Zhou and C. Zaniolo, *"Efficient XML-based Techniques for Archiving Querying and Publishing the History of Relational Databases"*, Submitted for Publication.
[19] F. Wang and C. Zaniolo. *"XBiT: An XML-based Bitemporal Data Model"*, in ER 2004.
[20] F. Wang and C. Zaniolo, *"Publishing and Querying the Histories of Archived Relational Databases in XML"*, in WISE 2003.

T-SIX: An Indexing System for XML Siblings

SungRan Cho
L3S, University of Hannover
scho@l3s.de

ABSTRACT

We present a system for efficient *indexed* querying of XML documents, enhanced with sibling operations. R-tree index proposed in [5] has a very high cost for the following-sibling and preceding-sibling axes. We develop a family of index structures, which we refer to as *transformed split-tree indexes*, to address this problem, in which (i) XML data is horizontally split by a simple, yet efficient criteria, (ii) the split value is associated with tree labeling, (iii) all data elements are transformed into new dimensions to avoid possible overlap between bounding boxes representing data elements in the split tree. The T-SIX system incorporates building transformed split-tree index for XML documents as well as query processing on all XPath axes to provide query answers.

1. INTRODUCTION

Efficient querying XML documents is an increasingly important issue considering the fact that XML becomes the de facto standard for data representation and exchange over the Web, and XML data in diverse data sources and applications is growing rapidly in size. Given the importance of XPath based query access, XML query evaluation engines need to be able to efficiently identify the elements along each location step in the XPath query. In this context, several index structures for XML documents have been proposed [4, 5, 7, 8, 10], in a way to efficiently querying XML documents.

As XML documents are modeled by a tree structure, a numbering scheme, labeling tree elements, allows for managing the hierarchy of XML data. For example, each element has the position, a pair of its beginning and end locations in a depth first search. In general, the numbering approach has the benefit of easily determining the ancestor-descendant relationship in a tree. In this respect, R-tree index using node's preorder and postorder, we refer to as *whole-tree indexes* (WI), has been proposed in [5]. Such index, however, does not consider issues related to the costs of the preceding-sibling and following-sibling axes.

In this paper, we develop index techniques to reduce the cost of XML siblings. It also addresses an issue of what efficient packing for XML tree data is. An efficient packing method for a tree is not only to group together data elements which are close in a tree, but also to reduce dead space resulting in false positives (no data in indexed space). In the WI, packing method, taking a whole tree, may cover considerable dead space, which influences querying XML siblings. We design the *transformed split-tree index* to address the problem, in which (i) an XML tree is horizontally split by the simple, but efficient criteria, (ii) the split value is associated with tree labeling, (iii) all data elements are transformed into new dimensions to avoid possible overlap between bounding boxes representing data elements in the split tree. To take advantage of the semantics of the index structure, we develop novel index lookup algorithms for XPath axes for the transformed split-tree index.

We describe T-SIX, a system for indexed querying enhanced with XML sibling operations. T-SIX incorporates tree labeling and coordinate transformation for XML documents. T-SIX implements novel index lookup algorithms for querying, providing query answers for the entire set of XPath query axes.

This demonstration is organized as follows. Section 2 shows the architecture of T-SIX. Section 3 gives the structure of our index and describes functionality and features that T-SIX encompasses.

2. SYSTEM ARCHITECTURE

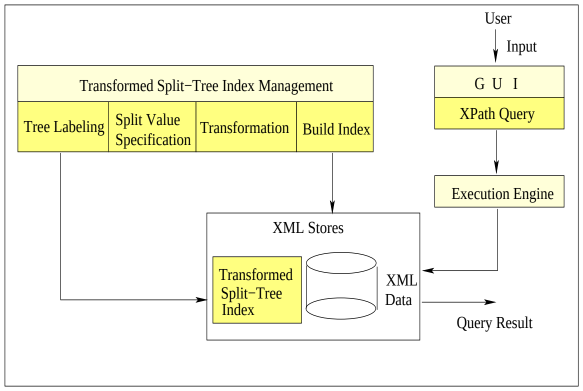


Figure 1: Architecture of T-SIX

T-SIX is a Java-based prototype. Its architecture is depicted in Figure 1. It consists of two main components: the *transformed split-tree index management* module and the *XPath querying* module. The first component implements various aspects of managing transformed split-tree index including tree labeling and coordinate transformation. It accepts the split value of a tree through a graphical interface and builds the index based on the value. The second

Eighth International Workshop on the Web and Databases (WebDB 2005), June 16-17, 2005, Baltimore, Maryland.

component implements a query processor on all XPath axes, that accepts queries and returns desired elements.

3. DEMONSTRATION OUTLINE

The system encompasses the following functionality and features that will be demonstrated: (a) facilitates browsing of different XML data sets, (b) facilitates browsing of element's original and transformed positional numbers, (c) incorporates building transformed split-tree index for XML data, (d) incorporates novel index lookup algorithms in the transformed split-tree index for XPath query processing, (e) implements a flexible and interactive graphical interface and display of queries and query results, (f) supports flexible ways to input split value information for XML documents, and (g) supports adjusting page size parameter of the index in an interactive mode.

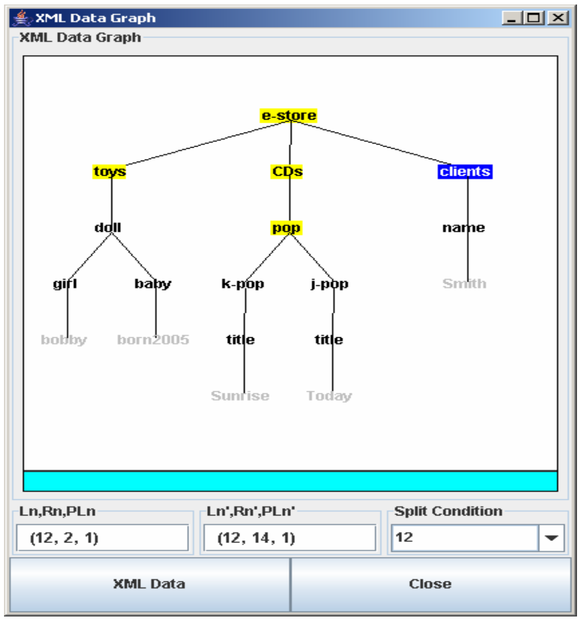
3.1 Transformed Split-Tree Index Management

Tree Labeling: We use an encoding scheme, `Ln` and `Rn`, for nodes in XML documents that has the same effect as preorder and postorder. `Ln` is the rank at which the node is encountered in a *left to right* depth first search (DFS) of the XML data tree, and `Rn` is the rank at which the node is encountered in a *right to left* DFS. In order to handle level sensitive matching, such as child and parent axes (matching nodes one level apart), and following-sibling and preceding-sibling axes (matching nodes with the same parent), the parent node's `Ln`, written as `PLn` is associated with each node. Thus each XML element node is labeled with three numbers: `Ln`, `Rn`, and `PLn`. These numbers become coordinates in multi-dimensions. Users can view element's positional information, `Ln`, `Rn`, and `PLn`, through an interface. In Figure 2, an example XML database represents an `e-store` that contains information about items and clients.

Split Value Specification: XML documents can be selected and browsed in graphical form. XML element nodes are labeled with element tags or string values; edges are either between elements or between an element and a string value. Once XML document is selected, a user can specify the split value, which is associated with element's `Ln` and `Rn`. In effect XML data tree is divided horizontally by the split value. T-SIX system provides conveniently visualize XML tree split. In Figure 2, for example, once a user submits the split value of 12, the `e-store` document is split by the value, in which `e-store`, `toys`, `CDs`, `pop` elements are cut in the XML tree and highlighted through a graphical interface. Split value information for XML documents can be dynamically modified on demand.

Transformation: While the separate packing reduces long thin boundary boxes, that may contain dead space (space which is indexed but does not have data), it causes the overlap between bounding boxes at each region of the tree. Due to overlap, multiple paths from the root downwards on the SI may need to be traversed, which results in increasing page accesses. Before building the index, T-SIX transforms coordinates of element nodes to avoid possible overlap. An element node n=($\mathtt{Ln}, \mathtt{Rn}, \mathtt{PLn}$) is transformed into n'=($\mathtt{Ln}'$, $\mathtt{Rn}'$, $\mathtt{PLn}'$), such that

$$\mathtt{Ln}' = \mathtt{Ln}$$
$$\mathtt{Rn}' = \mathtt{Ln} + \mathtt{Rn}$$
$$\mathtt{PLn}' = \mathtt{PLn}$$



Ln,Rn,PLn	Ln',Rn',PLn'	Split Condition
(12, 2, 1)	(12, 14, 1)	12

Figure 2: XML data

As a result, the original dimensions are extended with in the `Rn` direction with respect to `Ln`. Users can view the transformed coordinate information of elements.

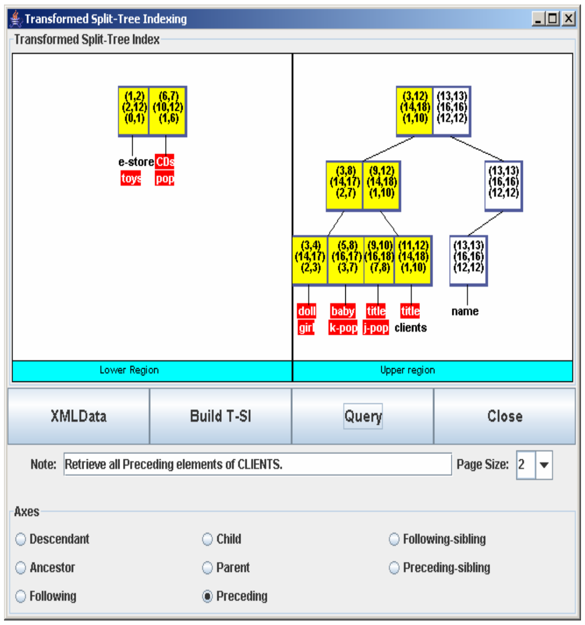
Building Index: A transformed split-tree index is constructed on new dimensions ($\mathtt{Ln}'$, $\mathtt{Rn}'$, $\mathtt{PLn}'$). T-SIX supports for loading new documents to construct the index in the hierarchical structure. Transformed split-tree index contents as well as index operations can be efficiently visualized in the system. Users can input the page capacity through a graphical interface. Figure 3, for example, shows a transformed split-tree index over the split `e-store` dataset of Figure 2 with a page capacity of 2. The leaf pages in the index contain both leaf and non-leaf XML elements, and non-leaf index pages indicate page boundaries by the smallest and the largest values occurring in the page.

3.2 XPath Querying

Once an XML document to be queried is selected, users can specify XPath queries through a flexible graphical interface. After a user query is submitted, T-SIX uses the transformed split-tree index to return all relevant XML data. T-SIX system supports all XPath axes and provides various ways to conveniently visualize query results. In Figure 3, for example, a user issues a query, "retrieve all elements preceding the `clinets` element". As a result, elements returned as well as non-leaf pages scanned in the index are highlighted in the graphical interface. The XML data tree also highlights the query results for easy understanding of XPath query operations.

4. CONCLUSION

The T-SIX system provides XPath querying enhanced with sibling operations. It supports novel transformed split-tree indexing methods to facilitate query operations in XML documents and dynamic changes of split value information on the XML data tree. It accepts XPath queries and displays query results through a graphical user interface. Furthermore, it supports dynamic visualization of



Note: Retrieve all Preceding elements of CLIENTS. Page Size: 2

Axes

- ( ) Descendant
- ( ) Ancestor
- ( ) Following
- ( ) Child
- ( ) Parent
- [✓ Preceding]
- ( ) Following-sibling
- ( ) Preceding-sibling

Figure 3: XPath querying

the index.

5. REFERENCES

[1] M. Altinel and M. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *Proc. of VLDB*, Cairo, Egypt, 53–64, 2000.

[2] J. Clark and S. DeRose. XML path language (XPath) version 1.0 w3c recommendation, Technical Report REC-xpath-19991116, World Wide Web Consortium, 1999.

[3] E. Cohen, H. Kaplan and T. Milo. Labeling dynamic XML trees, In *Proc. of PODS*, 271–281, 2002.

[4] B.F. Copper, N. Sample, M.J. Franklin, G.R. Hjaltason and M. Shadmon. A fast index for semistructured data, In *Proc. of VLDB*, Rome, Italy, 341–350, 2001.

[5] T. Grust. Accelerating XPath location steps, In *Proc. of SIGMOD*, 2002.

[6] A. Guttman. R-trees: a dynamic index structure for spatial searching, In *Proc. of SIGMOD*, 45–47, 1984.

[7] Q. Li and B. Moon. Indexing and querying XML data for regular path expressions, In *Proc. of VLDB*, Rome, Italy, 361–370, 2001.

[8] T. Milo and D. Suciu. Index structure for path expressions, In *Proc. of ICDT*, Jerusalem, Israel, 271–295, 1999.

[9] K.V. Ravikanth, D. Agrawal, A.E. Abbadi, A.K. Singh and T. Smith. Indexing hierarchical data, Univ. of California, CS-Tr-9514, 1995.

[10] H. Wang, S. Park, W. Fan and P. Yu. ViST: a dynamic index method for querying XML data by tree structures, In *Proc. of SIGMOD*, San Diego, USA, 2003.

[11] H.V. Jagadish, S. Al-Khalifa, A. Chapman, L.V.S. Lakshmanan, A. Nierman, S. Paparizos, J.M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, C. Yu, TIMBER: A native XML database, VLDB Journal 11(4): 274-291, 2002.

[12] XQuery 1.0: An XML query language, W3C Working Draft, November 2002.

[13] K. Deschler and E. Rundensteiner. MASS: A multi-axis storage structure for large XML documents, In *Proc. of CIKM*, Louisiana, USA, 2003.