Foundations and Trends® in
Databases
Vol. 1, No. 1 (2007) 1–140

DOI: 10.1561/1900000001

now
the essence of knowledge

Adaptive Query Processing

Amol Deshpande[1], Zachary Ives[2] and Vijayshankar Raman[3]

[1] *University of Maryland, USA, amol@cs.umd.edu*
[2] *University of Pennsylvania, USA, zives@cis.upenn.edu*
[3] *IBM Almaden, USA, ravijay@us.ibm.com*

Abstract

As the data management field has diversified to consider settings in which queries are increasingly complex, statistics are less available, or data is stored remotely, there has been an acknowledgment that the traditional optimize-then-execute paradigm is insufficient. This has led to a plethora of new techniques, generally placed under the common banner of *adaptive query processing*, that focus on using runtime feedback to modify query processing in a way that provides better response time or more efficient CPU utilization.

In this survey paper, we identify many of the common issues, themes, and approaches that pervade this work, and the settings in which each piece of work is most appropriate. Our goal with this paper is to be a "value-add" over the existing papers on the material, providing not only a brief overview of each technique, but also a basic framework for understanding the field of adaptive query processing in general. We focus primarily on *intra-query* adaptivity of long-running, but not full-fledged streaming, queries. We conclude with a discussion of open research problems that are of high importance.

1

Introduction

One of the fundamental breakthroughs of Codd's relational data model [33] was the identification of how to take a *declarative*, logic-based formulation of a query and convert it into an algebraic query evaluation tree. As described in every database textbook, this enabled physical data independence and promised many benefits: the database administrator and the DBMS optimizer became free to choose among many different storage formats and execution plans to answer a declarative query. The challenge, since then, has been how to deliver on these promises — regardless of where or how the data is laid out, how complex the query is, and how unpredictable the operating environment is.

This challenge has spurred 30 years of query processing research. Cost-based query optimization, pioneered by Selinger *et al.* [102] in System R and refined by generations of database researchers and developers, has been tremendously effective in addressing the needs of relational DBMS query processing: one can get excellent performance for queries over data with few correlations, executed in a relatively stable environment, given sufficient statistical information.

However, when even one of these characteristics is not present, the System R-style optimize-then-execute model begins to break down: as

noted in [69], optimizer error begins to build up at a rate exponential in the size of the query. As the database field has broadened to consider more general *data management*, including querying autonomous remote data sources, supporting continuous queries over data streams, encoding and retrieving XML data, supporting OLAP and data mining operations, and combining text search with structured query capabilities, the weaknesses of the traditional optimization model have begun to show themselves.

In response, there has been a surge of interest in a broad array of techniques termed *adaptive query processing* (AQP). AQP addresses the problems of missing statistics, unexpected correlations, unpredictable costs, and dynamic data by using *feedback* to *tune* execution. It is one of the cornerstones of so-called *autonomic* database management systems, although it also generalizes to many other contexts, particularly at the intersection of database query processing and the Web.

The spectrum of adaptive query processing techniques has been quite broad: they may span multiple query executions or adapt within the execution of a single query; they may affect the query plan being executed or the scheduling of operations within the plan; they have been developed for improving performance of local DBMS queries (e.g., [75, 87, 112]), for processing distributed and streaming data (e.g., [6, 72, 88, 92, 101]), and for performing distributed query execution (e.g., [115]).

This survey is an attempt to cover the fundamental issues, techniques, costs, and benefits of adaptive query processing. We begin with a broad overview of the field, identifying the dimensions of adaptive techniques. Then we focus our analysis on the spectrum of approaches available to adapt query execution at runtime — primarily in a non-streaming context. Where possible, we focus on simplifying and abstracting the key concepts of each technique, rather than reproducing the full details available in the papers; we consider generalizations of the specific published implementations. Our goal is to identify the strengths and limitations of the different techniques, demonstrate when they are most useful, and suggest possible avenues of future research.

In the rest of the section, we present a brief overview of query processing in relational database systems (Section 1.1) and elaborate on

the reasons behind the push toward adaptivity (Section 1.2); we then present a road map for the rest of the survey (Section 1.3), and briefly discuss the related surveys of interest (Section 1.4).

1.1 Query Processing in Relational Database Systems

The conventional method of processing a query in a relational DBMS is to parse the SQL statement and produce a relational calculus-like logical representation of the query, and then to invoke the query optimizer, which generates a *query plan*. The query plan is fed into an execution engine that directly executes it, typically with little or no runtime decision-making (Figure 1.1).

The query plan can be thought of as a tree of unary and binary relational algebra operators, where each operator is annotated with specific details about the algorithm to use (e.g., nested loops join versus hash join) and how to allocate resources (e.g., memory). In many cases the query plan also includes low-level "physical" operations like sorting, network shipping, etc. that do not affect the logical representation of the data.

Certain query processors consider only restricted types of queries, rather than full-blown SQL. A common example of this is select-project-join or SPJ queries: an SPJ query essentially represents a single SQL `SELECT-FROM-WHERE` block with no aggregation or subqueries.

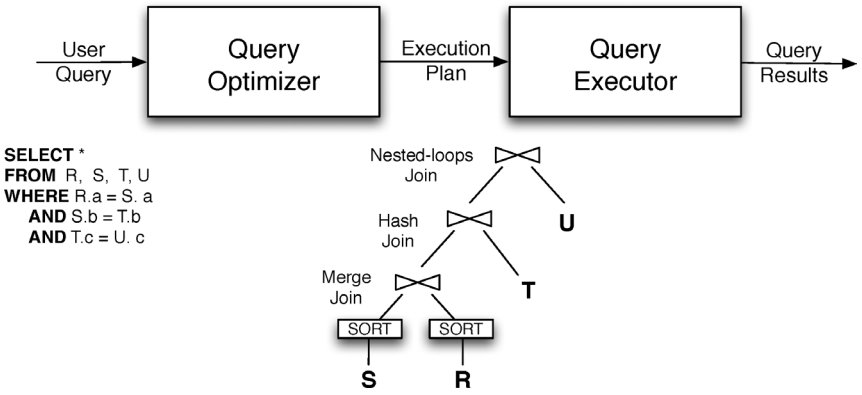

Fig. 1.1 Query processing in database systems.

An even further restriction is *conjunctive* queries, which are SPJ queries that only have conjunctive predicates in the `WHERE` clause; these can be represented as single rules in the Datalog language.

The model of query processing established with the System R project [102], which is still followed today, is to divide query processing into three major stages.

Statistics generation is done offline (typically using the `RUNSTATS` or `UPDATE STATISTICS` command) on the tables in the database. The system profiles the relation instances, collecting information about cardinalities and numbers of unique attribute values, and often generating histograms over certain fields.

The second stage, which is normally done at runtime,[1] is *query optimization*. The optimization stage is very similar to traditional compilation; in fact, in some systems, it generates directly executable code. Optimization uses a combination of *cost estimation*, where the running times of query subexpressions are estimated (based on known performance characteristics of algorithm implementations, calibration parameters for hardware speed, and the statistics generated for the relations), pruning heuristics (which are necessary to reduce the overall search space), and exhaustive enumeration. For relatively simple queries with good statistics, the plans produced by a query optimizer can be quite good, although as discussed previously, this is less true in more complex settings.

The final stage, *query execution*, is handled by an engine analogous to a virtual machine or interpreter for the compiled query plan. There are several important aspects of query execution that are of note. The first is that in general it is desirable to *pipeline* computation, such that each operator processes a tuple at a time from its sub-operators, and also propagates a single tuple to its parent for processing. This leads to better response time in terms of initial answers, and often higher throughput as delays are masked. However, not all operators are naturally amenable to pipelining (e.g., operators like sorting and grouping often must process entire table before they can determine

[1] Except for certain embedded SQL queries, which may be pre-optimized or optimized once for multiple possible input bindings.

what tuple to output next). Also, complex query plans may require too many resources to be fully pipelined. In these settings, the optimizer must break the plan into multiple segments, *materializing* (storing) intermediate results at the end of each stage and using that as an input to the next stage.

Second, the issue of *scheduling* computation in a query plan has many performance implications. Traditional query processing makes the assumption that an individual operator implementation (e.g., a nested loops join) should be able to control how CPU cycles are allocated to its child operators. This is achieved through a so-called *iterator* [53] architecture: each operator has `open`, `close`, and `getNextTuple` methods. The query engine first invokes the query plan root node's `open` method, which in turn `open`s its children, and the process repeats recursively down the plan. Then `getNextTuple` is called on the root node. Depending on the operator implementation, it will make calls to its children's `getNextTuple` methods until it can return a tuple to its parent. The process completes until no more tuples are available, and then the engine `close`s the query plan.

An alternate approach, so called *data-driven* or *dataflow* scheduling [121], is used in many parallel database systems. Here, in order to allow for concurrent computation across many machines, the data producers — not the consumers — control the scheduling. Each operator takes data from an input queue, processes it, and sends it to an output queue. Scheduling is determined by the rates at which the queues are filled and emptied. In this survey, we will discuss a number of adaptive techniques that in essence use a hybrid of the iterator and data-driven approaches.

1.2 Motivations for AQP

Over the years, many refinements have been made to the basic query processing technology discussed above. Since CPUs are more powerful today and query workloads are much more diverse, query optimizers perform a more comprehensive search of the space of query plans with joins, relying less on pruning heuristics. Selectivity estimation techniques have become more accurate and consider skewed distributions

(and to a limited extent, attribute correlations). However, the System R-style approach has begun to run into its fundamental limits in recent years, primarily due to the emergence of new application domains in which database query processing is being applied. In particular, triggers for this breakdown include the following:

- **Unreliable cardinality estimates:** The cost estimation process depends critically on estimates of the cardinality of various query subexpressions. Despite significant work on building better statistics structures and data collection schemes, many real-world settings have either inaccurate or missing statistics. (In some circumstances, as with remote data sources, statistics may be difficult or impossible to obtain.) Even when base-table statistics are perfect, correlations between predicates can cause intermediate result cardinality estimates to be off by several orders of magnitude [69, 112].
- **Queries with parameter markers:** SQL is not a pleasant language for end users, so most database queries are issued by a user clicking on a form. The SQL for such queries invariably contains parameter markers (for form input), and the precomputed query plans for such queries can be substantially worse than optimal for some values of the parameters.
- **Dynamically changing data, runtime, and workload characteristics:** In many environments, especially data streams [23, 88, 92], queries might be long-running, and the data characteristics and hence the optimal query plans might change during the execution of the query. The runtime costs can also change dramatically, especially in wide-area environments. Similarly, fluctuating query workloads can result in variations in the resources available to execute a query (e.g., memory), making it necessary to adapt.
- **Complex queries involving many tables:** Query optimizers typically switch to a heuristic approach when queries become too complex to be optimized using the dynamic programming approach. Such queries are naturally more prone

to estimation errors [69], and the use of heuristics exacerbates the problem.

- **Interactive querying:** The optimize-then-execute model does not mesh well with an interactive environment where a user might want to cancel or refine a query after a few seconds of execution: the metric changes too quickly for optimization to pay off [61]. Also, pipelined execution and early-result scheduling, even in the presence of slow data sources, becomes paramount.
- **Need for aggressive sharing:** Though there has been much work in multi-query optimization, so far no definitive solutions have emerged in this area. Traditional databases make do with almost no inter-query state sharing because their usage pattern is made up of a small number of queries against large databases. However, sharing the data as well as the computation is critical in environments such as data streams, which feature a very large number of (typically simple) queries over a small set of data streams [28, 86].

There have been two responses to the challenges posed above. The first, a very pragmatic response by application vendors, has been to build domain-specific optimization capabilities *outside* the DBMS and override its local optimizer. Many commercial DBMSs allow users to specify "hints" on what access methods and join orders to use, via SQL or catalog tables. Recently, SAP has built an application-level query processor that runs only a very limited set of plans (essentially, only table scans), but at very high efficiency [82]. While this achieves SAP's target of satisfying its users, it runs counter to the database community's goals of developing high-performance, general-purpose processors for declarative queries.

Our interest in this survey is on the second development, which has been the focus of the academic and commercial DBMS research community: the design and construction of what have come to be known as *adaptive* (or autonomic) query processing systems, that use runtime feedback to adapt query processing.

1.3 Road Map

We begin with a brief introduction to query optimization in relational database systems (Section 2). We then discuss some of the foundations of AQP, namely, three new operators, and several unifying concepts that we use throughout the survey to illustrate the AQP techniques, to analyze them, and to differentiate between them (Section 3).

We begin our discussion of adaptive query processing by considering a simple class of queries called selection ordering queries (Section 4). The discussion of adaptation techniques for join queries is divided into three parts, roughly based on the space of the query execution plans they explore. We begin with a discussion of techniques for adapting pipelined query execution (Sections 6 and 7), and cover non-pipelined query execution in Section 8. We conclude the survey with a discussion of some of the most important research challenges in adaptive query processing (Section 9).

1.4 Related Work

A number of surveys on query processing are related to this paper. We assume basic familiarity with many of the ideas of Graefe's survey on query execution techniques [53]. Kossmann's survey on distributed query processing [79] also provides useful context for the discussion, as do Ioannidis and Chaudhuri's surveys on query optimization [24, 68]. Babu and Bizarro [8] also present a survey of AQP from a different means of classification from our own (whether the scheme is plan-based, routing-based, or continuous query-based).

2

Background: Conventional Optimization Techniques

Before beginning to discuss adaptive query processing, we review some of the key concepts in single-pass, non-adaptive query optimization in relational databases (Section 2.1). Our goal is not to review general-purpose query optimization of full-blown SQL queries (we refer the reader to the surveys by Ioannidis [68] and Chaudhuri [24] for a broader overview of the topic). Instead, we aim to lay down the foundations for discussing AQP techniques in the later sections. We then briefly discuss the impact of cost estimates on the optimization process, and present strategies that have been proposed to make plan selection robust to erroneous estimates (Section 2.2).

2.1 Query Optimization

While general query optimization may consider `GROUP BY` [26] and multi-block SQL queries [58, 78, 89, 105], the heart of cost-based optimization lies in *selection ordering* and *join enumeration*, upon which we focus in this section. We begin with presenting the plan spaces that are explored for these problems, and the static planning algorithms commonly used. In some specific cases, more efficient and simpler planning

algorithms can be used, and we discuss these as appropriate later in the survey.

2.1.1 Selection Ordering: Plan Space

Selection ordering refers to the problem of determining the order in which to apply a given set of commutative selection predicates (filters) to all the tuples of a relation, so as to find the tuples that satisfy all the predicates. Figure 2.1 shows an example selection query over a *persons* relation and several query plans for that query.

Let the *query* to be evaluated be a *conjunction*[1] of n commutative predicates, $S_1, \ldots, S_n$, to be applied to the tuples from a relation R. We denote the *cost* of S_i by c_i, and the probability that a tuple satisfies the predicate S_i by $p(S_i)$, or p_i. Similarly, we denote the probability that a tuple satisfies predicates $S_{j_1}, \ldots, S_{j_k}$ by $p(S_{j_1} \wedge \cdots \wedge S_{j_k})$. If a tuple is

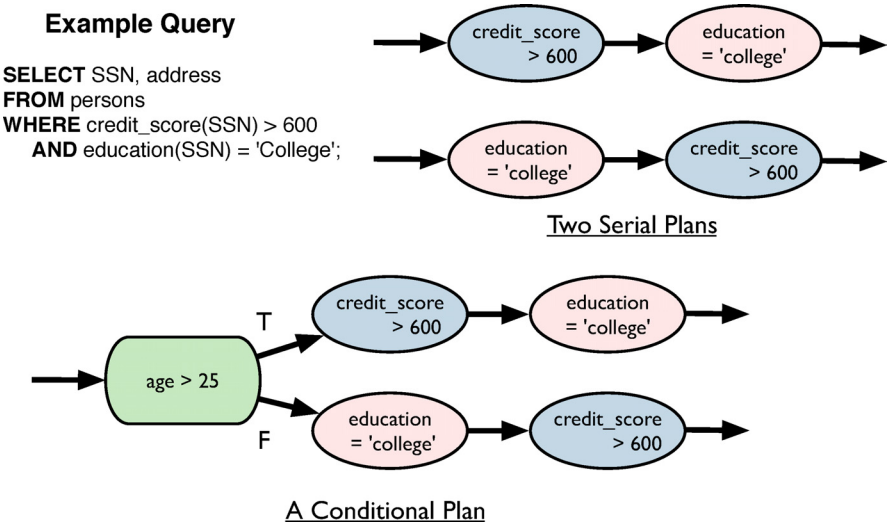

Fig. 2.1 An example query with two expensive *user-defined predicates*, two serial plans for it, and one conditional plan that uses *age* to decide which of the two expensive predicates to apply first.

[1] Queries with disjunctions are often treated as equivalent to a union of multiple queries in conjunctive normal form, although more efficient schemes exist for evaluating such plans with "bypass" operators [32] or multi-query optimization strategies [86, 91]. We do not consider such techniques in this survey and refer the reader to those works.

known to have already satisfied a set of predicates $\{S_{j_1}, \ldots, S_{j_k}\}$, then we denote the *conditional* probability that the tuple also satisfies S_i by $p(S_i|S_{j_1}, \ldots, S_{j_k})$. Note that if the predicates are independent of each other (an assumption typically made), then $p(S_i|S_{j_1}, \ldots, S_{j_k}) = p(S_i)$.

Serial Plans: The natural class of *execution plans* to consider for evaluating such queries is the class of *serial orders* that specify a single order in which the predicates should be applied to the tuples of the relation (Figure 2.1). Given a serial order, $S_{\pi_1}, \ldots, S_{\pi_n}$, where $\pi_1, \ldots, \pi_n$ denotes a permutation of $1, \ldots, n$, the *expected cost per tuple* for that order can be written as:

$$c_{\pi_1} + p(S_{\pi_1})c_{\pi_2} + \cdots + p(S_{\pi_1} \cdots S_{\pi_{n-2}} \wedge S_{\pi_{n-1}})c_{\pi_n}.$$

Or, if the predicates are independent of each other:

$$c_{\pi_1} + p(S_{\pi_1})c_{\pi_2} + \cdots + p(S_{\pi_1}) \times \cdots \times p(S_{\pi_{n-1}})c_{\pi_n}.$$

Conditional Plans: Conditional plans [17, 39] generalize serial plans by allowing different predicate evaluation orders to be used for different tuples based on the values of certain attributes. This class of plans can be especially beneficial when the attributes are highly correlated with each other, and when there is a large disparity in the evaluation costs of the predicates.

Figure 2.1 shows an example conditional plan that uses an inexpensive predicate on *age*, which is strongly correlated with both the query predicates, to decide which of the two predicates to apply first. Specifically, for tuples with age > 25, the predicate on *credit_score* is likely to be more selective than the predicate on *education*, and hence should be applied first, whereas the opposite would be true for tuples with age < 25.

2.1.2 Selection Ordering: Static Planning

The static planning algorithms for selection ordering are quite simple if the predicates are independent of one another; however, the planning problem quickly becomes NP-Hard in presence of correlated predicates.

Independent Predicates: If the predicates are independent of one another, as is commonly assumed by most query optimizers, the *optimal* serial order can be found in $O(n\log(n))$ time by simply sorting the predicates in the increasing order of $c_i/(1-p_i)$ [60, 67, 80]. Under independence assumptions, conditional plans offer no benefit over serial plans.

The expression $c_i/(1-p_i)$ is commonly referred to as the *rank* of the predicate, and this algorithm is hence called *rank ordering*.

Correlated Predicates: The complexity of the planning problem, in this case, depends on the way the correlations are represented. Babu *et al.* [10] show that the problem of finding the optimal order for a *given dataset D* (similarly, for a given *random sample* of the underlying relation) is NP-Hard. However, to our knowledge, it is not known whether the problem is tractable if the correlations are represented in some other format, e.g., using probabilistic graphical models [38, 49].

Assuming that the conditional probabilities for the predicates can somehow be computed using the correlation information, the following greedy algorithm (called *Greedy* henceforth) can be used [10]:

Algorithm 2.1 The Greedy algorithm for correlated selection ordering.
Input: A set of predicates, $S_i, i = 1, \ldots, n$; a procedure to compute *conditional probabilities*.
Output: A serial plan, $S_{\pi_1}, \ldots, S_{\pi_n}$, for applying the predicates to the tuples.

1. Choose S_{π_1} to be the predicate S_i with the lowest $c_i/(1-p(S_i))$ among all predicates.
2. Choose S_{π_2} to be the predicate S_j with the lowest $c_j/(1-p(S_j|S_{\pi_1}))$ among *remaining* predicates; $p(S_j|S_{\pi_1})$ denotes the conditional probability of S_j being true, given that S_{π_1} is true.
3. Choose S_{π_3} to be the predicate S_k with the lowest $c_k/(1-p(S_k|S_{\pi_1}, S_{\pi_2}))$ among *remaining* predicates.
4. Continue until no operators left.

For instance, if the correlation information is given in the form of a dataset (or a random sample) D, then the conditional probabilities can be computed by scanning D repeatedly as required. This algorithm can be shown to approximate the optimal solution within a constant factor of 4 [10]; for reasonable numbers of predicates, the bound is even smaller (e.g., for $n = 20$, the bound is only 2.35).

Finding an optimal conditional plan for a given dataset D can be shown to be NP-Hard as well [39] (using a straightforward reduction from *binary decision tree construction problem*). Several heuristic algorithms are presented in [39], but to our knowledge no approximation algorithms are known for this problem.

2.1.3 Multi-way Join Queries: Plan Space

In picking a query plan for a multi-way select-project-join (SPJ) query, an optimizer has to make many choices, the most important being: access methods, join order, join algorithms, and pipelining.

Access Methods: The optimizer needs to pick an access method for each table in the query. Typically there are many choices, including a direct table scan, a scan over an index (also called "index-only access" or "vertical partitioning" in the literature), an index-based lookup on some predicate over that table, or an access from a materialized view. Some stream or wide-area data sources may support only one access method: a continual stream of tuples entering the query processor. Others may allow binding parameters to be passed akin to index probes or the two-way semijoin [36].

Join Order: The access methods provide source tuples to the query plan. To join these tables, the optimizer then chooses a *join order*.

Definition 2.1. A *join order* is a tree, with the access methods as leaves; each internal node represents a join operation over its inputs.

Optimizers usually avoid performing relational cross-products, so they typically consider trees where each join (internal node) can apply some join predicates (over the node's descendants) specified

in the query. Traditionally, these internal nodes have been *binary* joins, but many adaptive techniques use n-ary join operators as well (Section 3.1.3). Each join order corresponds to a way of placing parenthesis around tables in the query: it is equivalent to a parse tree for an expression in which each join's input is parenthesized. (The poor name "join order" arose because historically DBMSs first supported only "left-deep" query plans; we will continue to use this terminology in this paper.) A left-deep or "left-linear" plan is one where only the left child of a join operator may be the result of another relational algebra operation; the right child must be a base relation. See Figure 2.2(ii).

Since the join order is a tree, it may not cover all join predicates. In particular, if the join predicates in a query form a cycle, the join order can only cover a spanning tree of this cycle. So, as part of join ordering, the optimizer also picks a spanning tree over the join graph. Join predicates on edges that are eliminated (to remove cycles) are applied after performing the join, as "residual" predicates.

Figure 2.2 shows an example multi-way join query that we use as a running example, and two possible join orders for it — one using a tree of binary join operators and the other using a ternary join operator.

Join Algorithms: The next aspect of join execution is the physical join algorithm used to implement each join in the join order — nested loop join, merge join, hash join, etc. These decisions are typically made

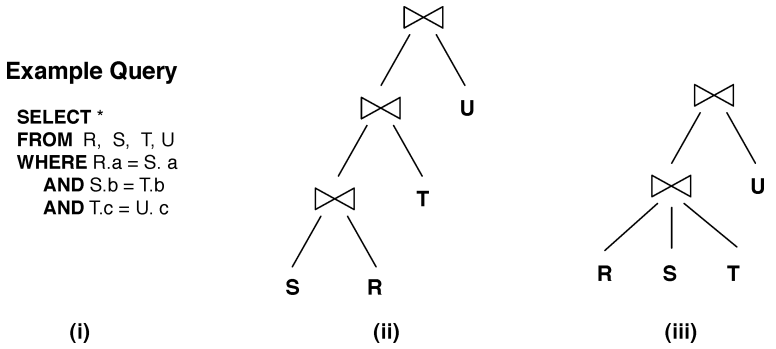

Fig. 2.2 (i) A multi-way join query that we use as the running example; (ii) a left-deep join order, in which the right child of each join must be a base relation; (iii) a join order that uses a ternary join operator.

during cost estimation, along with the join order selection, and are partly constrained by the available access methods. For instance, if the data from a relation is streaming in, pipelined join operators must be used. Similarly, an index access method is often ideally suited for use with an index nested-loops join or sort-merge join.

Pipelining vs. Materialization: One of the biggest differentiating factors between query plans for join queries is whether they contain a *blocking operator* or not. A blocking operator is one that needs to hold intermediate state in order to compute its output because it requires more than one pass over the input to create its output, e.g., a sort (which must sort the input) or a hash join (where the "build" relation is stored in a hash table and probed many times). In some cases, resource constraints might force materialization of intermediate state.

Query plans for SPJ queries can be classified into two classes:

- **Non-pipelined plans:** These contain at least one blocking operator that segments execution into stages: the blocking operator materializes its sub-results to a temporary table, which is then read by the next operator in a subsequent step. Each materialization is performed at a *materialization point*, illustrated in Figure 2.3 (i).

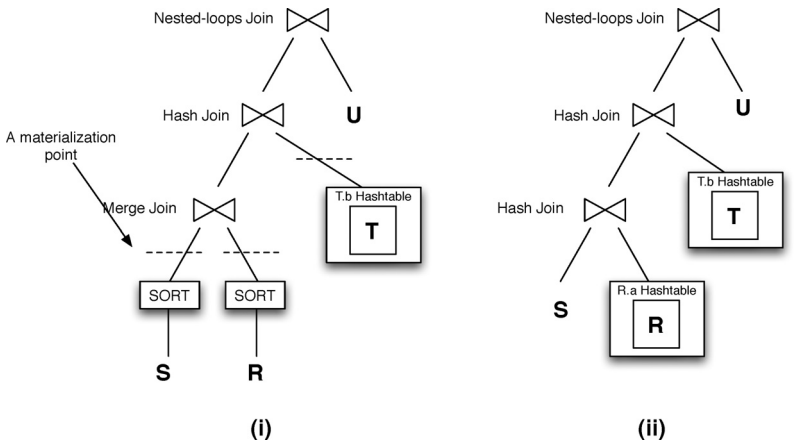

Fig. 2.3 (i) A non-pipelined plan; (ii) a pipelined plan.

- **Pipelined plans:** These plans execute all operators in the query plan in parallel, by sending the output tuples of an operator directly to the next operator in the pipeline. Figure 2.3(ii) shows an example pipelined plan that uses hash join operators. A subtlety of this plan is that it is actually only "partly pipelined": the traditional hash join performs a *build* phase, reading one of the source relations into a hash table, before it begins pipelining tuples from input to output. *Symmetric* or doubly pipelined hash join operators (Figure 2.4) are fully pipelined: when a tuple appears at either input, it is incrementally added to the corresponding hash table and probed against the opposite hash table. Symmetric operators are extensively used when quicker response time is needed, or when the inputs are streaming in over a wide-area network, as they can read tuples from whichever input is available, and they incrementally produce output based on the tuples received so far.

  We will continue to refer to plans such as the one shown in Figure 2.3(ii) as pipelined plans, as long as all the major operations in the plan are executed in a single pipeline.

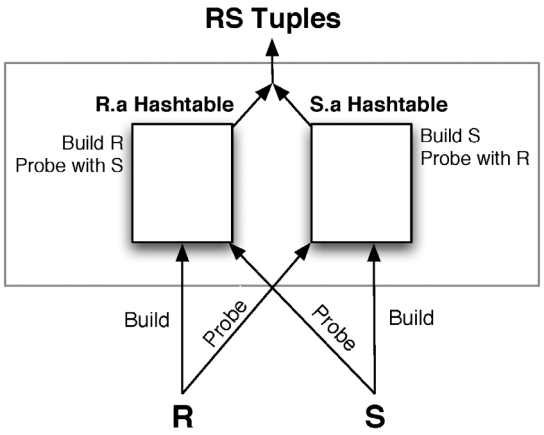

Fig. 2.4 Symmetric hash join (doubly pipelined hash join) operator builds hash tables on both inputs.

These two plan classes offer very different adaptation opportunities because of the way they manipulate state. Proposed adaptation schemes for pipelined plans usually involve changes in the tuple flow, whereas adaptation schemes in non-pipelined plans typically involve re-evaluating the rest of the plan at the materialization points.

2.1.3.1 A Variation: SPJ Queries over Data Streams

In recent years, a new class of queries has become popular: queries where the data sources are (infinite) data streams. To execute join queries over streams, *sliding windows* must be specified on the data streams in the query. As an example, if the relations R and S referred the above example query are streaming relations, then a query may be written in the CQL language [4] as follows:

select *
from R [rows 100], S [range 60 min], T, U
where R.a = S.a **and** S.b = T.b **and** S.c = U.c

This query specifies two sliding windows. At any time, the sliding window on R (a *tuple-based* window) contains the last 100 tuples of R. The window on S, a *time-based* window, contains all tuples that arrived during the last 60 min.

Because of the potentially endless nature of data streams, only fully pipelined plans can be used for executing them. In other respects, the query execution remains essentially the same, except that the earliest tuples may have to be deleted from the join operator data structures when they have gone outside the range of the window (i.e., its size or duration). For example, if the symmetric hash join operator shown in Figure 2.4 is used to execute the join between R and S, then tuples should be removed from the hash tables once they have been followed by sufficient tuples to exceed the sliding window's capacity.

2.1.4 Multi-way Join Queries: Static Planning

The wide array of choices in access methods, join orders, and join algorithms results in an enormous space of execution plans: one that is exponential in the number of tables and predicates in the query,

in the number of access methods on each table, and in the number of materialized views available. Query optimizers search in this plan space through dynamic programming, applying a number of heuristics to prune the choices and make optimization tractable. The basic method is the one proposed in the original System R paper [102]. We briefly summarize this here (for a more thorough survey of query optimization, we refer the reader to [24, 68]).

The System R paper made a number of ground-breaking observations. First, projection, and in many cases selection, can be pushed down as a heuristic. Hence, the key to optimizing an SQL query is reasoning about join order and algorithm selection.

The second key observation is that it is fairly easy to develop *cost modeling* equations for each join implementation: given page size, CPU and disk characteristics, other machine-specific information, and the cardinality information about the input relations, the cost of executing a join can be easily computed. The challenge is to estimate the cardinality of each join's output result — this forms one of the input relations to a subsequent join, and hence impacts all future cost estimation. System R only did limited reasoning about join selectivities, primarily based on hard-coded heuristics.

The third key idea — perhaps the most insightful — was that in general, the cost of joining two subexpressions is *independent* of how those subexpressions were computed. The optimal way of computing some join expression thus can be posed as the problem of considering all ways of factoring the expression into two subexpressions, optimally computing those subexpressions, and joining them to get the final expression. This naturally leads to a *dynamic programming* formulation: the System R optimizer finds all optimal ways of retrieving data from the source relations in the query, then all optimal ways of joining all pairs of relations, then all optimal ways of joining pairs with one additional relation, etc., until all combinations have been considered.

One exception to the independence property described above is the impact of sorting: a single sort operation can add significant one-time overhead, but this may be amortized across multiple merge joins. System R separately considers each so-called *interesting order* (i.e., sort

order that might be exploited when joining or grouping), as well as an "unconstrained order" in the dynamic programming table.

Only after optimizing joins (while simultaneously maximally pushing down selections and projections) did System R consider correlated subqueries and grouping — and for these, it generally did very little reasoning, and simply added a final nested loops join or group-by operator at the end.

System R limited its search in two ways to make the query optimization problem more tractable on 1970s hardware. First, it deferred reasoning about Cartesian products until all possible joins were evaluated: the assumption was that such plans were likely to be inefficient. Second, System R only considered left-deep or left-linear plans. (Left-linearness is significant because the right input to a nested loops join is conventionally the inner loop.)

Modern optimizers have extended System R's approach to plan enumeration in a number of key ways, in part because complex plans and operations became more prevalent and in part because machines became more powerful and could afford to do more optimization.

- **Plan enumeration with other operators:** Both Starburst [58] and Volcano [56] extended the cost estimation component of query optimization to explore combinations of operators beyond simply joins. For instance, group-by push-down [26] and shipment of data in a distributed setting [85] are often considered during cost estimation.
- **Top-down plan enumeration:** The original System R optimizer enumerated plans in true dynamic programming fashion, starting with single relations and building increasingly complex expressions. Some modern optimizers, following a model established in Volcano and Cascades [54], use *top-down* enumeration of plans, i.e., recursion with memoization. The primary benefit of this approach is that it enables early pruning of subexpressions that will never be used: branch-and-bound pruning, in particular, can be used to "short-circuit" the computation of subexpressions whose cost is higher than an existing, alternative expression that subsumes the existing one.

- **Cross-query-block optimization:** In contrast to System R's model, which processed each block of an SQL query separately, most modern DBMSs allow for optimizations that move predicates across blocks and in some cases even perform sophisticated rewritings, such as magic sets transformations [104]. Typically these rewritings are chosen using a combination of heuristics and cost estimation.
- **Broader search space:** A modern query optimizer often considers bushy plans (where two arbitrary sub-results may be joined, as opposed to requiring one of the inputs to be a leaf) as well as early Cartesian products. Additionally, when the query expression becomes sufficiently large, rather than attempting to do dynamic programming on all possible plans, most optimizers first partition the work using heuristics, and then run dynamic programming on each plan segment.

Of course, plan enumeration is only half of the query optimization story. In many ways, the "magic" of query optimization lies not in the plan enumeration step, but in the process of estimating a particular plan's cost.

2.2 Choosing an Effective Plan

As we previously described, the query optimizer can create a composite estimate of overall query execution cost from its knowledge of individual operators' costs, system parameters, and data distributions — ultimately enabling it to choose the minimal-estimated-cost plan from its plan enumeration space.

Naturally, one of the most essential aspects of the cost estimation process is *selectivity estimation*: given a set of input relations (which may be stored on disk or derived by the query) and an operation, the estimator predicts the cardinality of the result. This estimated cardinality then provides a prediction of the size of one of the inputs to the next subsequent operation, and so forth, until query completion.

The original System R implementation made use of very simple selectivity estimation strategies: the DBMS maintained cardinality information for each table, and selectivity estimation made use of

this information, as well as minimum and maximum values of indexed attributes and several ad hoc "magic ratios" to predict cardinalities [102].

All modern DBMSs employ more sophisticated techniques, relying on histograms created offline (via `RUNSTATS` or `UPDATE STATISTICS`) to record the distributions of selection and join key attributes. As described in [24], these histograms are not only used to estimate the results of selection predicates, but also joins. Unfortunately, because attributes are often correlated and different histograms may be difficult to "align" and intersect, significant error can build up in this process [69]. This has motivated a great deal of research into either making query plans more *robust*, pre-encoding *contingent* or *conditional* plans, or performing *inter-query adaptation* of the cost model. We proceed to discuss each of these ideas in turn.

2.2.1 Robust Query Optimization

In some cases the query optimizer has a choice between a "conservative" plan that is likely to perform reasonably well in many situations, or a more aggressive plan that works better if the cost estimate is accurate, but much worse if the estimate is slightly off. Chu *et al.* [30, 31] propose *least expected cost optimization* where the optimizer attempts to find the plan that has the lowest expected cost over the different values the parameters can take, instead of finding the lowest cost plan for the expected values of the parameters. The required probability distributions over the parameters can be computed using histograms or query workload information. This is clearly a more robust optimization goal, assuming only one plan can be chosen and the required probability distributions can be obtained. (The latter may be difficult if the workload is diverse, the data changes, or there are complex correlations.) An approach along similar lines was proposed by Babcock and Chaudhuri [7] recently, called *robust query optimization*. Here the authors provide a knob to tune the predictability of the desired plan vs. the performance by using such probability distributions.

Error-aware optimization (EAO) [119] makes use of intervals over query cost estimates, rather than specifying the estimates for single

statistical points. EAO focuses primarily on memory usage uncertainty. A later work, Rio [9], provides several features including the use of intervals. It generates linear query plans (a slight variation of the left-linear or left-deep plan, in that one of the two inputs to every join — not necessarily the right one — must be a base relation) and uses bounding boxes over the estimated cardinalities in order to find and prefer robust plans. (Rio's proactive reoptimization features are discussed in Section 8).

A different means of making plans more robust is to employ more sophisticated operators, for instance, n-way pipelined hash joins, such as MJoins [120] (discussed in Section 3.1.3) or eddies [6] (discussed in Section 3.1.2). Such operators dramatically reduce the number of plans considered by the query optimizer, although potentially at the cost of some runtime performance.

2.2.2 Parametric Query Optimization

An alternative to finding a single robust query plan is to find a small set of plans that are appropriate for different situations. Parametric query optimizers [46, 52, 70] postpone certain planning decisions to runtime, and are especially attractive in scenarios where queries are compiled once and executed repeatedly, possibly with minor parameter changes. They choose among the possible previously chosen plan alternatives at the start of execution (i.e., before the first pipeline begins) and in some cases at intermediate materialization points. The simplest form uses a set of query execution plans annotated with a set of ranges of parameters of interest; just before query execution commences, the current parameter values are used to find the appropriate plan.

Graefe *et al.* [34, 52] propose *dynamic query evaluation plans* for this purpose, where special *choose-plan* operators are used to make decisions about the plans to use based on the runtime information. Some of the choices may not be finalized until after the query has begun executing (as opposed to only at the beginning of execution); this allows the possibility of using the intermediate result sizes as parameters to make the decisions.

Ioannidis *et al.* [70] study parametric optimization for the buffer size parameter, and propose randomized algorithms for generating the parametric plans. More recently, several researchers have begun a systematic study of parametric query optimization by understanding the complexity of the problem [46, 98], and by considering specific forms of cost functions that may be easier to optimize for, generalizing from linear to piecewise linear to convex polytopes [64, 65, 94].

Despite the obvious appeal of parametric query optimization, there has not been much research in this area and the commercial adoption has been nearly non-existent. The bane of this technique is determining what plans to keep around: the space of all optimal plans is super-exponential in the number of parameters considered. More recently, an alternative idea was proposed in the progressive reoptimization work [87], where *validity ranges* are associated with query plans: if the values of the parameters are observed to be outside these ranges at the time of the execution, reoptimization is invoked. This idea is largely shared in progressive parametric query optimization (PPQO) [19], which combines the ideas of parametric query optimization with progressive reoptimization: the reoptimizer is called when error exceeds some bound (the Bounded implementation) or when there is no "near match" among the set of possible plan configurations (the Ellipse implementation).

2.2.3 Inter-Query Adaptivity: Self-tuning and Autonomic Optimizers

Several techniques have been proposed that passively observe the query execution and incorporate the knowledge learned from these previous query executions to better predict the selectivity estimates in future. Chen and Roussopoulos [27] propose *adaptive selectivity estimation*, where an attribute distribution is approximated using a curve-fitting function that is learned over time by observing the query results. *Self-tuning histograms* [1, 22] are similar in spirit, but focus on general multi-dimensional distributions that are approximated using *histograms* instead of curve-fitting. These approaches have the additional advantage of not having to directly access the data to build the statis-

tical summaries: they in effect "snoop" on the intermediate results of queries.

More recently, the LEarning Optimizer (LEO) project [112] generalizes this basic idea by monitoring arbitrary subexpressions within queries as they execute and comparing the *actual* observed selectivities with the optimizer's selectivity estimates. This information is then used to compute adjustments to the optimizer's estimates that may be used during future optimizations of similar queries. If a significant difference is observed, reoptimization may be triggered during execution; we will discuss this aspect of LEO in more detail in Section 8.

Similarly, Bruno and Chaudhuri [21] propose gathering *statistics on query expressions* (SITS) during query execution and using those during optimization of future queries. Their main focus is on deciding which of the query expressions, among a very large number of possible candidates, to maintain such statistics on.

In some sense, these techniques form a feedback loop, across the span of different queries, in the design of an adaptive database system. Many of them are also fairly easy to integrate into a commercial database system, as evidenced by the development of self-tuning wizards and autonomic optimizers in leading DBMSs.

2.3 Summary

The traditional, single-pass, optimize-then-execute strategy for query execution has served the database community quite well since the 1970s. As queries have become more complex and widespread, however, they have started to run into limitations. Schemes for robust optimization, parametric optimization, and inter-query adaptivity alleviate some of these difficulties by reducing sensitivity to errors. A significant virtue of these methods is that they impose little runtime overhead on query execution. Perhaps even more importantly, they serve as a simple upgrade path for conventional single-pass query processors.

However, there are settings in which even these techniques run into limitations: for instance, if the query workload is highly diverse, then subsequent queries may have little overlap; if the actual costs change frequently, as they may in highly distributed settings, then the recal-

ibration may not be beneficial; if the data itself changes frequently, as it may in streaming, data integration, or high-throughput settings, then the new statistics may be out of date. In these settings, we would like to immediately *react* to such changes or *adapt the current query plan*. Through the remainder of this survey, we focus on such *intra-query* adaptive query processing techniques that adapt the execution of a single query, for greater throughput, improved response time or more useful incremental results.

3

Foundations of Adaptive Query Processing

The goal of adaptive query processing is to find an execution plan and a schedule that are well-suited to runtime conditions; it does this by interleaving query execution with exploration or modification of the plan or scheduling space. At a very high level, the differences between various adaptive techniques can be explained as differences in the way they interleave. Standard, System R-style [102] query processing does full exploration first, followed by execution. Evolutionary techniques like choose nodes [34] or mid-query reoptimization [75] interleave planning and execution a few times, whereas more radical techniques like eddies [6] interleave them to the point where they are not even clearly distinguishable.

We note that the adaptive aspect is not free: given sufficient information, an offline optimization strategy could define a plan or a collection of plans (e.g., for different partitions of relations) that always matches or improves the execution times of the adaptive technique, without the overhead of exploration. However, in reality such information is seldom accurately available in advance, which is the major motivation for studying adaptivity. Different techniques incur different

amounts of overhead in order to explore the search space and to adapt an executing query plan.

In this section, we introduce some of the foundations of adaptive query processing. We begin with presenting three new operators that play a central role in several adaptive techniques that we discuss in this survey (Section 3.1). We then present a framework called the *adaptivity loop* that we use for illustrating the AQP techniques throughout this survey (Section 3.2), and discuss how the behavior of an adaptive technique may be analyzed post-mortem using traditional query plans (Section 3.3). We conclude with an analysis of several example systems using these two concepts (Section 3.4).

3.1 New Operators

Traditional database engines are optimized for disk-based I/O, which (at least on a modern storage architecture) can supply data to a DBMS at very high rates. The goal is to have very tight control loops, carefully scheduled to minimize the per-tuple overhead [53]. This type of execution, however, tends to severely restrict the possible adaptation opportunities, especially when wide-area data sources or data streams are involved. In this section, we present three new operators that address these problems by allowing for greater scheduling flexibility and more opportunities for adaptation. Such flexibility requires greater memory consumption and more execution overhead, but the result is still often superior performance.

We note that our focus here is on describing the underlying *mechanisms*; we postpone the discussion of most of the *policy* aspects to latter sections.

3.1.1 Symmetric Hash Join Operators

The traditional hash join operator is not very well suited for adaptive query processing; it must wait for the *build* relation to fully arrive before it can start processing the *probe* relation and producing results. This makes it unsuitable for handling wide-area data sources and data streams, where the inputs arrive in an interleaved fashion, and continuous result generation is desired. Further, this operator severely restricts

adaptation opportunities since the build relations must be chosen in advance of query execution and adapting these decisions can be costly.

The symmetric hash join operator [99, 121] introduced in Section 2.1.3 solves both these problems by building hash tables on both inputs (Figure 2.4); when an input tuple is read, it is stored in the appropriate hash table and probed against the opposite table, resulting in incremental output. Because the operator is symmetric, it can process data from either input, depending on availability. This operator also enables additional adaptivity since it has frequent *moments of symmetry* [6] – points at which the join order can be changed without compromising correctness or without losing work. As a result, this operator has formed the cornerstone of many AQP techniques.

The symmetric hash join operator does have a significant disadvantage in that the memory footprint is much higher since a hash table must be built on the larger input relation as well.

Several pieces of work build upon the original proposal for this operator that was developed for dataflow-based parallel query processing. Both XJoin [117] and the doubly pipelined hash join [72] adapted the operator to a multi-threaded architecture, using producer–consumer threads instead of a dataflow model, and they also included strategies for handling overflow to disk when memory capacity is exceeded. In general, they only perform runtime decisions in terms of deciding what to overflow to disk and when to process overflowed data.

Another extension, called *ripple join*, proposed for interactive query processing, adapts the order in which tuples are read from the inputs so as to rapidly improve the precision of approximated aggregates; in addition to equality join predicates, ripple joins can also be used for non-equijoins [59].

3.1.2 Eddy [6]

Avnur and Hellerstein [6] proposed the *eddy* operator for enabling fine-grained run-time control over the query plans executed by the query engine. The basic idea behind the approach they propose is to treat query execution as a process of *routing* tuples through *operators*, and to *adapt* by changing the order in which tuples are routed through the

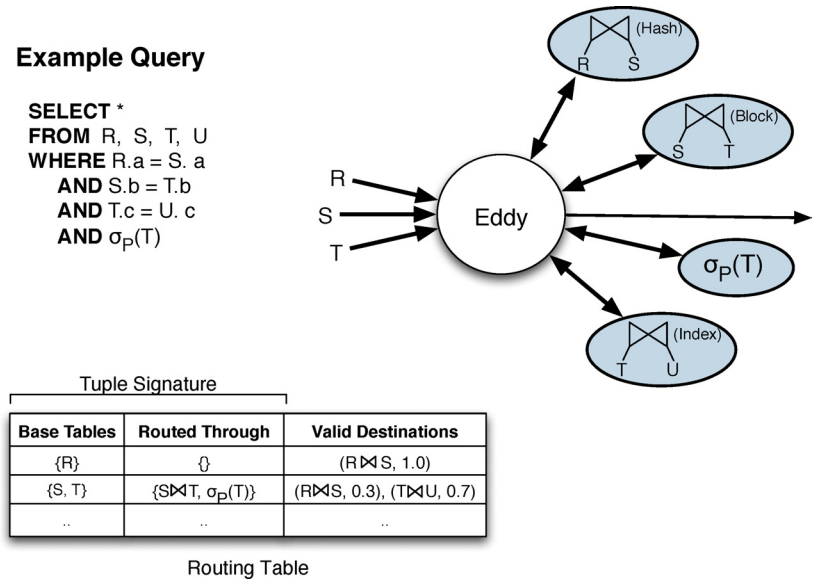

Fig. 3.1 Example of an eddy instantiated for a 4-way join query (taken from Avnur and Hellerstein [6]). A routing table can be used to record the valid routing destinations, and possibly current probabilities for choosing each destination, for different tuple signatures.

operators (thereby, in effect, changing the query plan being used for the tuple). The eddy operator, which is used as the tuple router, monitors the execution, and makes the routing decisions for the tuples.

Figure 3.1 shows how an eddy can be used to execute a 4-way join query. Along with an eddy, three join operators and one selection operator are instantiated. The eddy executes the query by routing tuples from the relations R, S, and T through these operators; a tuple that has been processed by all operators is sent to the output. The eddy can adapt to changing data or operator characteristics by simply changing the order in which the tuples are routed through these operators. Note that the operators themselves must be chosen in advance (this was somewhat relaxed by a latter approach called SteMs that we discuss in Section 6). These operator choices dictate, to a large degree, the plans among which the eddy can adapt. Pipelined operators like symmetric hash join offer the most freedom in adapting and typically also provide immediate feedback to the eddy (for determining the operator selectiv-

ities and costs). On the other hand, blocking operators like sort-merge operators are not very suitable since they do not produce output before consuming the input relations in their entirety.

Various auxiliary data structures are used to assist the eddy during the execution; broadly speaking, these serve one of two purposes:

1. Determining *Validity* of Routing Decisions: In general, arbitrary routing of tuples through the operators is not always correct. As an example, the eddy should not route R tuples to the selection operator $\sigma_P(T)$, since that operator expects and operates on T tuples. In fact, R tuples should only be routed to the $R \bowtie S$ operator.

Typically, some form of *tuple-level lineage*, associated with each tuple, is used to determine the validity of routing decisions. One option is to use the set of base relations that a tuple contains and the operators it has already been routed through, collectively called *tuple signature*, as the lineage. However, for efficient storage and lookups, compact representations of this information are typically used instead. For instance, the original eddies proposal advocated attaching two bitsets to each tuple, called *done* and *ready*, that respectively encode the information about the operators that the tuple has already been through, and the operators that the tuple can be validly routed to next [6]. We will see several other approaches for handling this later.

2. Implementation of the Routing Policy: Routing policy refers to the set of rules used by the eddy to choose a routing destination for a tuple among the possible valid destinations. This is the most critical component of an eddy and is directly responsible for the performance benefits of adaptivity. To facilitate clear distinctions between different routing policies, we will use the following unifying framework to describe a routing policy.

The routing policy data structures are classified into two parts:

— **Statistics about the query execution:** To aid in making the routing decisions, the eddy monitors certain data and operator characteristics; the specific statistics maintained depend on the routing policy. Since the eddy processes every

tuple generated during query execution, it can collect such statistics at a very fine granularity. However, the cost of maintaining detailed statistics could be high, and must be weighed against the expected adaptivity benefits of gathering such statistics.

— **Routing table:** This data structure stores the valid routing destinations for all possible tuple signatures. Figure 3.1 shows an example of such a routing table for our example query. As discussed above, more compact, bitmap-based representations may be used instead for efficient storage and lookups.

To allow for probabilistic choices (so that the eddy can *explore* other alternatives during execution), a *probability* may be associated with each destination. Deterministic policies are simulated by requiring that the routing table have exactly one destination with non-zero ($=1$) probability for each tuple signature; in that case, we further assume that destination is the first destination listed for the tuple in the routing table.

Given these data structures, the eddy follows a two-step process for routing a tuple:

— **Step 1:** The eddy uses the statistics gathered during the execution to construct or change the routing table. This may be done on a per-tuple basis, or less frequently.

— **Step 2:** The eddy uses the routing table to find the valid destinations for the tuple, and chooses one of them and routes the tuple to it. If the eddy is using probabilistic routing, this step involves a random number generation and a scan over the possible destinations (this can be reduced to $O(H(p))$ using a Huffman Tree, where $H(p)$ denotes the entropy of the probability distribution over the destinations [63]). If the eddy is using deterministic routing, this takes $O(1)$ time since we assume that the deterministic destination is listed first in the routing table.

This framework minimizes the architectural overhead of adaptivity. The policy overhead of using a routing policy depends largely on the

frequency with which the eddy executes Step 1 and the statistics that it collects; these two factors also determine how well and how quickly the eddy can adapt. This overhead can be minimized with some careful engineering (e.g., by amortizing the cost of Step 1 over many tuples [37], or by using random sampling to maintain the statistics). We will revisit this issue in more detail when we discuss specific routing policies in the later sections.

3.1.3 *n*-ary Symmetric Hash Joins/MJoins

An n-ary Symmetric Hash Join (called *MJoin* henceforth) [95, 120] generalizes the binary symmetric hash join operator to multiple inputs by treating the input relations symmetrically and by allowing the tuples from the relations to arrive in an arbitrary interleaved fashion. An MJoin has several attractive features over the alternative option of a tree of binary join operators, especially in data stream processing and adaptive query processing.

Figure 3.2 shows an example MJoin operator instantiated for a 4-way join query. MJoins build a hash index on every join attribute of every relation in the query. In the example, three hash indexes (that share the base tuples) are built on the S relation, and one hash table each is built on the other relations. For acyclic query graphs, the total number of hash tables ranges from $2(n-1)$ when every join is on a different attribute to n when all joins are on the same attribute.

An MJoin uses a lightweight *tuple router* to route the tuples from one hash table to another. The eddy operator can also be used for this purpose [95].

During the query execution, when a new tuple from a relation arrives into the system, it is first built into the hash tables on that relation, and then the hash tables corresponding to the remaining relations are *probed* in some order to find the matches for the tuple. The order in which the hash tables are probed is called the *probing sequence*. For example, when a new tuple $s \in S$ arrives into the system, the following steps are taken:

— s is built into the (three) hash indexes on the relation S.
— Let the probing sequence chosen for s be $T \rightarrow R \rightarrow U$ (Figure 3.2).

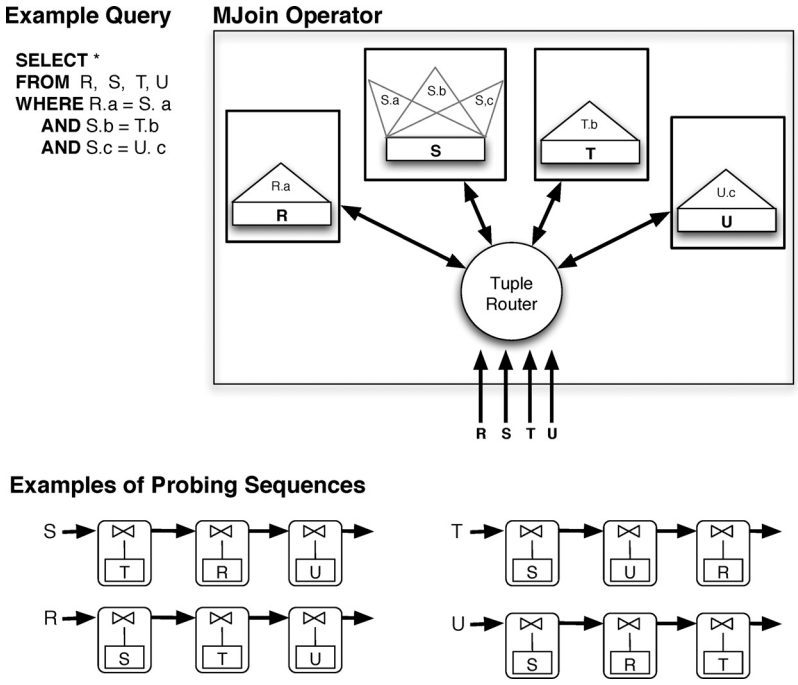

Fig. 3.2 Executing a 4-way join query using the MJoin operator. The triangles denote the in-memory hash indexes built on the relations.

— s is used to probe into the hash table on T to find matches. Intermediate tuples are constructed by concatenating s and the matches, if any.
— If any result tuples were generated during the previous step, they are used to probe into the hash tables on R and U, all in that order.

Similarly, when a new R tuple arrives, it is first built into the hash table on R. It is then probed into the hash table on S on attribute $S.a$ first, and the resulting matches are then probed into the hash tables on T and U. Note that the R tuple is not eligible to be probed into the hash tables on T or U directly, since it does not contain the join attributes corresponding to either of those two joins.

An MJoin is significantly more attractive over a tree of binary operators when processing queries involving sliding windows over data streams; when a base tuple from a relation drops out of the sliding win-

dow, only the base tuple needs to be located and removed from a hash table, since intermediate tuples are not stored in the hash tables. Further, MJoins are naturally very easy to adapt; the query plan being used can be changed by simply changing the probing sequence. For these two reasons, much work in data stream processing has focused on using an MJoin-style execution [12, 86]. However, MJoins tend not to perform as well as trees of binary join operators, especially for non-streaming data. This is mainly because they do not reuse the intermediate results; we will revisit this issue in more detail in Sections 6 and 7.

Memory Overflow: Handling memory overflow is harder with MJoins than with symmetric hash joins, especially if the probing sequences are changed during execution. Viglas *et al.* [120] study this problem and present a technique called *coordinated flushing* that aims to maximize the output rate while allowing tuples to be spilled to disk. This technique, however, can only be applied if all joins in the query are on the same attribute. More recently, Bizarro and DeWitt [18] generalize this to the non-identical join attributes case, and present a scheme that processes an input tuple maximally given the in-memory partitions of the relations; the remaining portions of the relations are joined at the end using a clean-up phase. Exploring the interactions between out-of-core execution, adaptivity, and cost metrics, remains a rich area for future research (cf. Section 9).

3.2 Adaptivity Loop

In intra-query adaptivity, regular query execution is supplemented with a control system for monitoring query processing and adapting it. Adaptive control systems are typically componentized into a four-part loop that the controller repeatedly executes (either in line or in parallel with normal tuple processing, see Figure 3.3):

Measure: An adaptive system periodically or even continuously monitors parameters that are appropriate for its goals.

Analyze: Given the measurements, an adaptive system evaluates how well it is meeting its goals, and what (if anything) is going wrong.

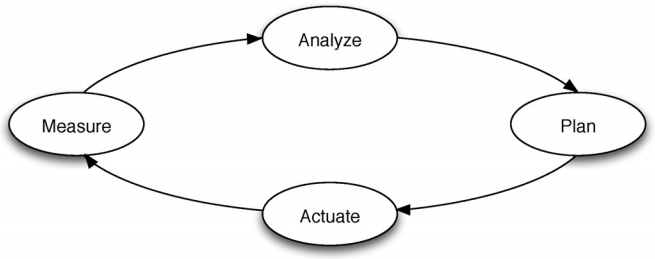

Fig. 3.3 Adaptivity loop.

Plan: Based on the analysis, an adaptive system makes certain decisions about how the system behavior should be changed.

Actuate: After the decision is made, the adaptive system executes the decision, by possibly doing extra work to manipulate the system state.

In the context of adaptive query processing, the Measure step involves monitoring data characteristics like cardinalities and distributions, and system characteristics like memory utilization and network bandwidth. Analysis is primarily done with respect to performance goals or estimates, although there have been some recent proposals for broader QOS-aware DBMSs that address goals like availability [14, 57, 66, 84, 106]. The Plan step can involve a traditional query optimizer (in, e.g., mid-query reoptimization [9, 73, 75, 87]), or be a routing policy that is performed as part of execution (in eddies [6, 40, 95]). The Actuate step corresponds to switching from one plan to another, with careful state migration to reuse work and ensure correct answers.

These above steps are simple in concept but involve difficult engineering to achieve overall efficiency and correctness. Measurement, analysis, and planning all add overhead to normal query processing, and it is often tricky to balance the ability to react quickly to newly discovered information against the ability to rapidly process tuples once a good plan has been chosen (the classic *exploration–exploitation dilemma*).

Throughout this survey, we will (where appropriate) relate the various adaptive techniques to this four-part loop, as a way of understand-

ing the key aspects of the technique and its relative trade-offs. First, we provide further detail on each of the stages.

3.2.1 Measurement

In general, all intra-query adaptation methods perform some form of monitoring, such as measuring cardinalities at key points in plan execution. As we will see in Sections 6–8, mid-query reoptimization [75, 87] does this at the end of each pipeline stage, and it may also add statistics collectors (i.e., histogram construction algorithms) at points that are judged to be key; eddies measure the selectivities and costs of the operators; corrective query processing [73][1] maintains cardinality information for all operators and their internal state (including aggregation).

Often it is desirable to explore the costs of options other than the currently executing query plan. Eddies perform exploration as a part of execution, by routing tuples through different alternative paths, serving the goals of execution and information gathering simultaneously. Antoshenkov's work on DEC Rdb [3] adopted a different, *competitive* strategy in which multiple alternative plans were redundantly run in parallel; once enough information was gathered to determine which plan appeared most promising, all other plans were terminated.

Finally, at times it is more efficient to *stratify* the search and measurement space, by executing a plan in a series of steps, and hence measuring only the active portions of query execution. Choose nodes [52] and mid-query reoptimization [75, 87] follow this strategy, interleaving measurement-plus-execution with analysis and actuation.

3.2.2 Analysis

The analysis step focuses on determining how well execution is proceeding — relative to original estimates or to the estimated or measured costs of alternative strategies. There are two major caveats to this phase. First, the only way to know precise costs of alternative strategies is through competitive execution, which is generally expensive. Thus most adaptive strategies employ sampling or cost modeling

[1] Originally called *convergent* query processing.

to evaluate the alternatives. Second, and perhaps more fundamentally, all adaptive strategies analyze (some portion of) *past* performance and use that to predict *future* performance. Clearly, in theory an adversarial workload can be created that does not follow this property; however, all evidence suggests that in practice, such a situation is highly unlikely.

Most adaptive techniques make use of some form of plan cost modeling in the analysis phase, comparing current execution to what was originally expected or what is estimated to be possible. Some approaches, such as mid-query reoptimization and corrective query processing, use the query optimizer's cost modeler to perform analysis. Eddies and their descendants [86, 95] generally do not perform full cost modeling (with the exception of [40]) — instead they rely on local routing heuristics to achieve a similar effect.

3.2.3 Planning

Planning is often closely interlinked with analysis, since it is quite common that the same process that reveals a plan to be performing poorly will also suggest a new plan. Mid-query reoptimization and its variants (Section 8) compute plans in stages — using analysis from the current stage to produce a new plan for the next stage, and supplementing it with the appropriate measurement operators. Corrective query processing (Section 7) incrementally collects information as it computes a query, and it uses this information to estimate the best plan for the remaining input data.

In some places, changing the query plan requires additional "repairs": Query scrambling, for instance, may change the order of execution of a query plan, and in some cases *plan synthesis* [118] is required (see Section 8.3). Similarly, corrective query processing requires a computation to join among intermediate results that were created in *different* plans; this is often done in a "cleanup" or "stitch-up" phase.

Eddies and their descendants do not plan in the same fashion as the other strategies: they use queue length, cost, and selectivity estimation to determine a "next step" in routing a tuple from one plan to another. SteMs [95] and STAIRs [40] use different planning heuristics to manage the intermediate state, in essence performing the same actions as the

stitch-up phase described above, but without postponing them to the end of query execution.

3.2.4 Actuation

Actuation, the process of changing a query plan, is a mechanism whose cost depends on how flexible plan execution needs to be. Additionally, when changing a plan, some previous work may be sacrificed, accumulated execution state in the operators may not be reused easily and may need to be recomputed.

In the simplest of cases, where query plans can only be changed after a pipeline finishes (as with mid-query reoptimization [75] and choose nodes [52]), actuation is essentially free, since it is inexpensive to reconfigure operators that have not yet begun execution. However, even with this restriction, it is possible that prior work might have to be discarded: consider a scenario where sub-optimality is detected after the build phase of a hash join, and the new plan chooses not to use the hash table [18]. Kabra and DeWitt [75] explicitly disallowed such actuations. Several other techniques, e.g., query scrambling [2], POP [87], Rio [9], consider the availability of such state while re-planning (Section 8).

In contrast, schemes that support the changing of query plans in the middle of pipelined execution must be more careful. The main concern here regards the state that gets accumulated inside the operators during execution; if the query plan is changed, we must make sure that the internal state is consistent with the new query plan. The adaptive techniques proposed in literature have taken different approaches to solving this problem. Some AQP techniques will only consider switching to a new query plan if it is consistent with the previously computed internal state. For example, the original proposal for the eddies technique [6] does not allow the access methods chosen at the beginning to be changed during execution. Although this restricts the adaptation opportunities somewhat, these techniques can still adapt among a fairly large class of query plans. Another alternative is to switch at certain consistent points indicated by *punctuation* [116] markers (e.g., the end of a window or the change in a grouping value). Other tech-

niques use specialized query operators to minimize such restrictions (e.g., MJoins [120], SteMs [95]). SteMs, in fact, use sophisticated duplicate elimination logic that allows them to adapt among a much larger plan space than most other adaptive techniques.

The other approach taken by some systems is to *modify* the internal state to make it consistent with the new query plan. Zhu *et al.* [124] call this process *dynamic plan migration* and propose and evaluate several techniques for doing this while minimizing the impact on the query execution. Similarly the STAIR operator [40] exposes the internal state using carefully designed APIs and allows the query executor to change the internal state, for ensuring correctness or for improving the performance. Corrective query processing [73] can combine state from different operators that represent the same logical subexpression; conversely, punctuation [42, 83] and other techniques like filter conditions [123] can be used to perform pruning on state. Recent work [74] has even shown that *adaptive information passing* can use intermediate state from *other* operators in an executing query plan to prune state that does not contribute to query answers.

We organize our discussion of multi-join query adaptation based on how the various AQP techniques handle intra-operator state (Sections 4.4), and we revisit this issue in more detail then.

3.3 Post-mortem Analysis of Adaptive Techniques

As opposed to traditional query processors that use a single plan to process all tuples of the input relations, adaptive techniques may use different plans for different input tuples. Many of the techniques, especially routing-based techniques like eddies, do not appear to use a "plan" at all. This makes it hard to analyze or reason about the behavior of these systems. As an example, the commonly used "explain" feature (used to inspect the execution plan used by the DBMS for a query) would be nearly impossible to support in such systems.

Throughout this survey, along with the adaptivity loop, we will also discuss how to do a retrospective analysis of an adaptive query execution, after it has finished running. The goal of such "post-mortem" analysis is to understand how the result tuples were generated and, if

possible, to express the query execution in terms of traditional query plans or relational algebra expressions.

Generally speaking, we see two types of behavior:

Single Traditional Plan: Several adaptive techniques, although they adapt during execution, use a single traditional plan for all tuples of the relations, with the only difference being that the plan is not chosen in advance. The mid-query reoptimization techniques fall in this category.

Horizontal Partitioning: Many adaptive techniques can be seen as exploiting a larger plan space than used by traditional query optimizer; plans in this plan space exploit *horizontal partitioning* (also called *horizontal decomposition* [20]) by splitting the input relations into disjoint partitions, and executing different traditional plans for different partitions. For instance, given a query, $R \bowtie S \bowtie T$, a plan that uses horizontal partitioning may partition S into S_1 and S_2, leaving R and T intact. Since joins are distributive over unions, we have that:

$$R \bowtie S \bowtie T = (R \bowtie S_1 \bowtie T) \cup (R \bowtie S_2 \bowtie T)$$

If S_1 and S_2 exhibit different data characteristics (join selectivities), then using different plans for the two subqueries would be beneficial over forcing the same plan on both subqueries. Note that, in general, sharing of data structures (e.g., hash tables) and common subexpressions across different subqueries would be essential for these benefits to start showing.

Behavior of many adaptive techniques like eddies, corrective query processing, etc., can be captured using this plan space. Horizontal partitioning is also commonly used in parallel databases where the data has to be distributed between the constituent machines, and has also been shown to be useful in faster execution of star queries in data warehouses [15]. The concept of horizontal partitioning is also related to the ideas of conditional planning (Section 2.1.1), content-based routing [17], and selectivity-based partitioning [93]; all of these techniques exploit attribute correlations to horizontally partition the data so that different partitions of a relation can be processed using different plans.

The behavior of many adaptive techniques that we discuss in this survey will be reasonably straightforward to analyze, and in such cases we will not explicitly do a post-mortem analysis. However, for more complex schemes, such as those based on routing, such analysis will enhance our understanding.

3.4 Adaptivity Loop and Post-mortem in Some Example Systems

To get a feel for these two concepts in action, we now briefly analyze some well known systems through this lens.

3.4.1 System R

As discussed in Section 2.1.4, the System R query processor proposed the optimize-then-execute paradigm still prevalent today. The adaptivity in this system was limited to inter-query adaptation.

Measurement: System R measures the cardinalities of the relations and some other simple statistics on a periodic basis (by offline scanning of the database tables).

Analysis & Planning: The statistics are analyzed during planning at "compile" time; as discussed in Section 1.1, System R uses a cost-based optimizer to make the planning decisions.

Actuation: Finally, actuation is straightforward in this system. When the query is to be executed, operators are instantiated according to the plan chosen. Since System R does not perform intra-query adaptation, once these decisions are made, they are not changed.

Post-mortem: System R uses a single plan, chosen at the beginning of execution, for all tuples of the input relations.

3.4.2 Ingres

The query processor of Ingres, one of the earliest relational database systems [113], is highly adaptive. It did not have the notion of a *query*

execution plan; instead it chose how to process tuples on a tuple-by-tuple basis.

More specifically, to *join* the data from n tables, $R_1, \ldots, R_n$, the query processor begins by evaluating the predicates on the relations and materializing the results into *hashed temps*, i.e., temporary tables hashed on appropriately chosen attributes of the relations. This is done by utilizing a special *one variable query processor (OVQP)* that forms the innermost component of the query processor. OVQP, as its name suggests, optimizes and processes queries over a single relation in a cost-based manner, taking into account the existing indexes and temporary hash tables on the relation.

After that is done, the query processor begins by choosing a tuple from the smallest table, say R_1. The values from this tuple are substituted into the query (as constants) and the resulting (new) query is recursively evaluated in a similar fashion. This process continues until a query over a single relation is obtained, which is then evaluated using OVQP. After all results for the first tuple of R_1 have been generated, the query processor repeats the process with the second tuple of R_1 and so on. Since the decision at each step of recursion is made based on the sizes of the materialized tables, different plans may be used for different tuples, and in general, this process will not lead to a single join order being used to execute the query.

From the adaptivity perspective, we note the following things.

Measurement: The key measurements are the sizes of the materialized tables that get created during the execution.

Analysis & Planning: The analysis and planning are done at the beginning of each recursive call (after a call to OVQP). The query processor analyzes the query, and the sizes of the materialized tables and decides the relation to be substituted next.

Actuation: Finally, the actuation is done by substituting the values of a tuple as constants in the current query to construct a new (smaller) query. (Note that this combines actuation with query execution itself.)

As we can see, the Ingres query processor interleaves the four components of the adaptivity loop to a great extent, and is highly adaptive

by our definition of adaptivity. Disregarding the cost of the materializations, this execution has many similarities to execution using a left-deep pipelined query plan that uses only hash joins (Section 6.1) and also to execution using the MJoin operator.

Doing the post-mortem analysis of a query execution in Ingres is quite tricky, mainly because Ingres does not use traditional query operators (such as hash joins or sort-merge joins) and does not appear to follow the steps a traditional query plan would follow. For instance, when a tuple $r \in R_1$ is chosen for substitution, in effect, a set of probes are made into the hash tables on the rest of the relations. However, these are made separately and not one after another as a query plan would do. The hash tables into which the probes are made might change from tuple to tuple as well (except for the tuples in R_1).

Post-mortem: Except in some very specific cases, the steps followed during the execution of a query in Ingres cannot be written down as traditional query plans (even if the notion of horizontal partitioning is employed).

3.4.3 Eddies

In essence, eddies unify the four components of the adaptivity loop into a single unit and allow arbitrarily interleaving between them, leaving analysis and actuation to the routing policy. We briefly discuss some of the tradeoffs in the process here, and defer a more detailed discussion to when we present the routing policies.

Measurement: Since every tuple generated in the system passes through the eddy, the eddy can monitor the operator and data characteristics at a very fine granularity. Though the policies proposed in literature differ in what is observed and what statistics are collected, most of them monitor the characteristics continuously during the execution.

Analysis & Planning: Referring back to the routing decision process (Section 3.1.2), analysis and planning are done in Step 1, with the frequency decided by the routing policy. Policies like lottery scheduling [6] do the planning for every tuple. Due to the high overhead of

planning, several latter policies did this periodically [37, 40]; it could also be done in response to certain events instead (as the A-Greedy technique [10] does for selection ordering — see Section 4.1).

Actuation: The process of actuation and its cost depend largely on the operators being used by the eddy, and also the plan space that the eddy explores (Section 3.2.4). For selection-ordering queries executed using stateless pipelined filters, the cost of actuation is negligible. For multi-way join queries, the actuation cost can vary from negligible (if n-ary hash join operator is used) to prohibitively high (if state manipulation is required [40]).

Although query execution with eddies appears arbitrary and ad hoc, depending on the nature of the operators used, the execution can usually be captured using the notion of horizontal partitioning. In essence, the tuples of the source relations are horizontally partitioned based on the routing decisions that were made for them.

Post-mortem: The behavior of an eddy can usually be captured using traditional query plans and horizontal partitioning, although this depends on the rest of the operators used during execution.

We will revisit both the adaptivity loop and post-mortem analysis for eddies when we discuss specific applications of eddies later in the survey.

3.5 Scope of the Remainder of the Survey

In the remainder of this survey, we will discuss the details of a number of different techniques, attempting to tie the themes and issues together according to the dimensions outlined in this section. We have elected to focus on depth rather than breadth in this paper. Many of the techniques we discuss in the next few sections operate on a fairly restricted class of queries: single-block select-project-join queries, with joins on equality predicates. The techniques of Sections 7 and 8 apply more generally. Most of these techniques also assume that query execution is happening at a central server; however, the relations may reside on

a local disk, may be fetched from a remote data source using a variety of access methods, or may be streaming into the system.

We relate each technique to the measure/analyze/plan/actuate loop, and at the end of each section we include a brief recap of how all of the discussed techniques fit into this loop. For several of the techniques, we also discuss how a post-mortem analysis of the query execution may be done, to get more insights into the behavior of those techniques.

4

Adaptive Selection Ordering

We begin our discussion of adaptive query processing techniques by considering a restricted query processing problem, namely selection ordering for queries on a single table. As discussed in Section 2, selection ordering refers to the problem of determining the order in which to apply a given set of commutative selection predicates (filters) to all the tuples of a relation, so as to find the tuples that satisfy all the predicates. Selection ordering is one of the central optimization problems encountered during query processing, and has received renewed attention in the recent years in the context of environments like the web [25, 35, 43, 51, 110, 111], continuous high-speed data streams [6, 10], and sensor networks [39]. These environments present significantly different challenges and cost structures than traditional centralized database systems. Selection ordering problems have also been studied in other areas such as fault detection and machine learning (see e.g., Shayman *et al.* [108] and Kaplan *et al.* [76]), under names such as learning with attribute costs [76], minimum-sum set cover [45], and satisficing search [109].

More importantly, a large class of execution plans for multi-way join queries have behavior that is similar to selection ordering plans

(Section 6); this equivalence will not only aid us in understanding complex multi-way join queries, but will also form the kernel around which many of the adaptive techniques are designed. In fact, the Ingres system makes little differentiation between selection ordering and join ordering. Through the technique of decomposition [122] (Section 3.4.2), it maps join processing into a series of tuple lookups, variable bindings, and selection predicates. Thus join really becomes a matter of determining an order for binding tuples and evaluating selections. Additionally, the problem of selection ordering is quite well-understood, with several analytical and theoretical results known for it. As we will see, this is in large part due to the "stateless" nature of selection ordering queries.

We begin this section by presenting an adaptive technique called *A-Greedy* [10] that was proposed for evaluating selection ordering queries over data streams (Section 4.1). We then consider adapting using the eddy operator discussed in the previous section, and discuss several routing policies proposed for adapting selection ordering queries (Section 4.2). We conclude with a brief discussion of several extensions of the selection ordering problem to parallel and distributed scenarios (Section 4.3).

4.1 Adaptive Greedy

The *adaptive greedy* (A-Greedy) [10] algorithm is based on the *Greedy* algorithm presented in Section 2.1.2 (Algorithm 2.1). It continuously monitors the selectivities of the query predicates using a random sample over the recent past, and ensures that the order used by the query processor is the same as the one that would have been chosen by the Greedy algorithm.

Let the query being evaluated be a conjunction of n commutative predicates, $S_1, \ldots, S_n$, over relation R, and let c_i and $p(S_i)$ denote the cost and the selectivity of S_i, respectively. Figure 4.1 shows a highly simplified overview of the A-Greedy technique for evaluating this query. There are three main components:

Query Executor: The executor simply evaluates the query over a newly arrived tuple of the data stream according to the *current* execu-

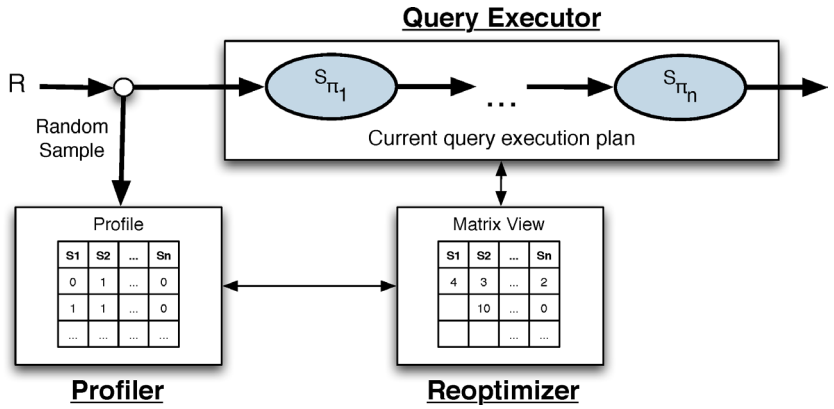

Fig. 4.1 A highly simplified overview of the *A-Greedy* technique.

tion plan as chosen by the reoptimizer. Like the Greedy algorithm, A-Greedy also uses a serial order of the predicates (Section 2.1.1) for this purpose. We denote the current serial order being used by $S_{\pi_1} \rightarrow \cdots \rightarrow S_{\pi_n}$, where $\pi_1, \ldots, \pi_n$ denotes a permutation of $1, \ldots, n$.

Profiler: This component maintains a *sliding window profile* over R using a random sample of the tuples that did not satisfy at least one predicate. The profile is typically maintained over only the tuples seen in the recent past (using a sliding window over the data stream), and contains the (boolean) results of the evaluations of all query predicates over the selected tuples. When a new tuple arrives, the profiler randomly decides whether to include the tuple into the profile; if the tuple is chosen to be included, the profiler evaluates all query predicates against the tuple, and adds the results to the profile. The profiler also removes *old* tuples from the profile as they fall out of the sliding window. Additionally, the profiler computes the expected evaluation costs, $c_i, i = 1, \ldots, n$, of the predicates using the profile tuples.

Reoptimizer: The reoptimizer ensures that the plan currently being executed is the one that would have been chosen by the *Greedy* algorithm over the profile tuples. The key insight here is to observe that

$S_{\pi_1} \rightarrow \cdots \rightarrow S_{\pi_n}$ would have been the order chosen by the Greedy algorithm if the following property holds:

Definition 4.1. Greedy Invariant: $S_{\pi_1}, \ldots, S_{\pi_n}$ satisfies the greedy invariant if, for all i:

$$\frac{c_{\pi_i}}{1 - p(S_{\pi_i}|S_{\pi_1}, \ldots, S_{\pi_{i-1}})} \leq \alpha \frac{c_{\pi_j}}{1 - p(S_{\pi_j}|S_{\pi_1}, \ldots, S_{\pi_{i-1}})} \quad \forall\ j\ >\ i$$

where $\alpha \leq 1$ is the *thrashing-avoidance parameter* (discussed below).

Intuitively, if this was not true for indexes π_i and π_j, then the next operator chosen after $\pi_1, \ldots, \pi_{i-1}$ would have been S_{π_j}, and not S_{π_i} (see Algorithm 2.1). The reoptimizer continuously monitors this invariant over the profile tuples; if it discovers that the invariant is violated at position i, it reoptimizes the evaluation order after S_{π_i} by invoking the Greedy algorithm over the profile tuples.

The reoptimizer uses a data structure called *matrix-view, V* (Figure 4.1), for monitoring the invariant:

Definition 4.2. The *matrix-view* is an upper triangular matrix such that $V[i,j],\ i \leq j$, contains the number of profile tuples that satisfied $S_{\pi_1}, \ldots, S_{\pi_{i-1}}$, but did not satisfy S_{π_j}.

This matrix-view is updated every time the profile is updated. It is easy to see that $V[i,j]$ is proportional to $(1 - p(S_{\pi_j}|S_{\pi_1}, \ldots, S_{\pi_{i-1}}))$. Along with the expected average costs of executing the predicates, as computed by the profiler, V allows the reoptimizer to quickly check if the greedy invariant is violated.

The α parameter ensures protection against thrashing; if $\alpha = 1$ and if the equation above is almost true for a pair of selections, the reoptimizer may be invoked repeatedly for correcting minor violations involving those two selections. Choosing an $\alpha < 1$ can help in avoiding such behavior (at the expense of increasing the approximation ratio to $4/\alpha$).

A-Greedy, like most AQP techniques, makes an implicit assumption that the statistical properties of the data do not undergo sudden and drastic changes. More specifically, it assumes that the data properties

in the recent past (as defined by the *length* of the sliding window) are predictive of the data properties in the near future. In a latter paper, Munagala *et al.* [90] show how this assumption may be relaxed, and present an online algorithm with a competitive ratio of $O(\log(n))$.

We can examine A-Greedy using the adaptivity loop (Section 3.2):

Measurement: A-Greedy continuously measures the properties of the operators by explicitly evaluating all the predicates over a random sample of the tuples (even if the tuples do not satisfy all predicates), and summarizing the results of this evaluation in the form of the matrix-view. The cost of this includes the predicate evaluation cost itself, as well as the cost of updating the matrix-view. A single update to the profile could result in updates to up to $\frac{n^2}{4}$ entries in the matrix-view.

Analysis: The analysis is also done continuously by the reoptimizer, which looks for violations of the greedy invariant using the matrix-view. This is an $O(n)$ operation involving up to n comparisons.

Planning: If a violation is detected in the analysis phase, the *Greedy* algorithm is used to construct a new execution plan. As discussed in Section 2.1.2, this can require scanning the profile tuples $O(n)$ times.

Actuation: The stateless nature of selection operators makes plan switch itself trivial. After a new serial order is chosen, the tuples arriving henceforth are simply processed using the new order.

Babu *et al.* [10] present several heuristic optimizations over this basic scheme to reduce the overheads of this process; due to space constraints, we omit a detailed discussion of these in this paper.

Post-mortem analysis of selection ordering queries can be done in a fairly straightforward manner using horizontal partitioning: the tuples are grouped into partitions based on the order in which the predicates were applied to them.

Post-mortem: The query execution using the A-Greedy technique can be expressed as a horizontal partitioning of the input relation by order of arrival, with each partition being executed using a serial order.

The A-Greedy technique is unique in that it explicitly takes predicate correlations into account, and analytical bounds are known for its performance. No other technique that we discuss has these properties. A-Greedy has also been used for adapting several classes of multi-way join queries as we will see later in the survey.

4.2 Adaptation using Eddies

The eddy operator [6], discussed in Section 3.1.2, can be used in a fairly straightforward manner to adapt a selection ordering query (Figure 4.2). To execute the query $\sigma_{S_1 \wedge \cdots \wedge S_n}(R)$, one eddy operator and n selection operators are instantiated (one for each selection predicate). The eddy executes the query by routing tuples through the operators. When an operator receives a tuple from the eddy, it applies the corresponding predicate to the tuple; if the predicate is satisfied, the operator returns the tuple to the eddy, otherwise it "drops" the tuple. Since selection operators are commutative, the eddy only needs to use one bitmap per tuple to determine the validity of routing decisions. This bitmap, called *done*, encodes the information about which operators the tuple has already been routed through. All operators which have the *done* bit set to *false* are valid routing destinations for the tuple. If all bits in the bitmap are set to *true*, the eddy outputs the tuple.

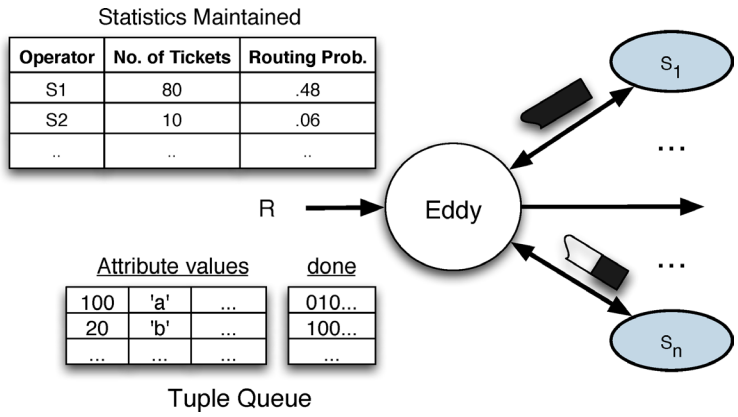

Fig. 4.2 Executing a selection query using an eddy using the Lottery Scheduling policy. Because of *back-pressure*, even though S_1 has a high ticket count, the next tuple in the queue at the eddy will not be routed to it.

Next we discuss several routing policies for adapting selection ordering queries.

4.2.1 Routing Policies

There is clearly a trade-off between the statistics that an eddy collects, the extent of its exploration of the plan space, and its ability to adapt. In this section, we will try to make this trade-off explicit by presenting several different routing policies for selection ordering queries. We will present these in the order of increasing complexity, rather than adhering to a chronological order from the literature. Table 4.1 compares these routing policies along several axes using the adaptivity loop.[1]

1. Deterministic Routing with Batching [37]: This deterministic routing policy uses *batching* to reduce the routing overhead.

Table 4.1 Comparing the techniques and the routing policies discussed in this section using the adaptivity loop. We omit the *actuation* aspect since the cost of actuation is negligible for all of these.

Deterministic [37]
Measurement: Selectivities by observing operator behavior (*tuples-in/tuples-out*); costs monitored explicitly. *Analysis and Planning:* Periodically re-plan using *rank ordering*.
A-Greedy [10]
Measurement: Conditional selectivities by random sampling; costs monitored explicitly. *Analysis:* Detect violations of the *greedy invariant*. *Planning:* Re-plan using the Greedy algorithm.
Lottery scheduling [6]
Measurement: Monitor *ticket counts* and *queue lengths*. *Analysis and planning:* Choose the route per-tuple based on those.
Content-based routing [17]
Measurement: For each operator, monitor conditional selectivities for the best classifier attribute. *Analysis and planning:* Choose the route per-tuple based on values of classifier attributes and selectivities (exploits *conditional plans*).

[1] See [19] for an experimental comparison of these routing policies.

— **Statistics maintained:** The eddy explicitly monitors the selectivity of each operator by counting the number of tuples routed toward the operator, and the number of tuples returned by it. This is sufficient to determine the operator selectivities as long as the predicates are independent of each other (an assumption that was made in that work). The costs of the operators are monitored explicitly as well using the system clock.

— **Routing policy:** The proposed routing policy makes decisions for *batches* of tuples at a time to reduce the routing policy overhead. Briefly, the eddy invokes a *reoptimizer* every K tuples, where K is a system parameter called the *batching factor*. The reoptimizer uses the statistics maintained over the past K tuples to find the optimal ordering of the predicates using the rank ordering algorithm (Section 2.1.2). The resulting plan is encoded into a routing table, and the next K tuples are routed according to that plan. This delineation between Planning and Actuation results in negligible routing overhead for reasonable batching factors ($K = 100$) [37].

2. Routing Policy based on A-Greedy: As observed by Babu *et al.* [10], the A-Greedy algorithm discussed in the previous section can be used to do the routing with a few minor changes to the basic eddy mechanism. The eddy would maintain the *profile* and the *view matrix*, and detect and correct the violations of the greedy invariant. As with the above policy, the eddy would route all the tuples using a single serial order; when a violation is detected, the eddy would invoke the reoptimizer to change the order being used. Note that, with A-Greedy, the tuples chosen for profiling must be routed through all the operators, regardless of whether they satisfy all predicates. One way to handle this would be to associate two additional bits with each tuple; one of those would be used to flag the tuples chosen for profiling (which should not be dropped by an operator), and the other would be used by an operator to record whether the tuple was falsified by the corresponding predicate.

3. Lottery Scheduling: Lottery scheduling was the initial routing policy proposed by Avnur and Hellerstein [6]. This policy was developed in a system called *River* [5] that uses a different *thread* per operator, and routes tuples between operators using *queues*. This is cleverly used to determine the operator costs.

— **Statistics maintained:** The eddy assigns *tickets* to each operator; an operator is credited with a ticket when the eddy routes a tuple to the operator, and the operator is penalized a ticket if it returns a tuple to the eddy. Assuming fair-share CPU scheduling and predicate independence, the number of tickets for an operator is roughly proportional to $(1 - s)$, where s is the selectivity of the operator.

— **Routing policy:** The eddy holds a *lottery* among the eligible operators when it has to route a tuple (either a new tuple or a tuple sent back by an operator); the chance of an operator winning the lottery is proportional to its ticket count. In other words, the eddy does probabilistic routing using normalized ticket counts as weights. Along with the operators that the tuple has already been through, the operators whose *input queues* are full are also considered ineligible to receive the tuple. The latter condition, called *backpressure*, allows the eddy to indirectly consider the operator costs when making routing decisions; the intuition being that operators with full input queues are likely to have high relative per-tuple execution cost (Figure 4.2).

Though the per-tuple overhead of this policy might seem prohibitive, the cost can be significantly reduced with some careful engineering [37].

4. Content-based Routing: Unlike the above policies, content-based routing (CBR) [17] explores and utilizes the space of conditional plans (Section 2.1.1) as follows:

— **Statistics maintained:** For each operator, CBR identifies the tuple attribute that is most strongly correlated with the

selectivity of the operator. For example, consider a selection operator over an *employee* table that applies the predicate $salary > \$100000$ to the tuples. The selectivity of this operator is likely to be strongly correlated with the attribute age (e.g., tuples with age < 20 are unlikely to satisfy the predicate). Such attributes are called *classifier* attributes. Using a random sample of the tuples seen in the recent past, CBR identifies the best classifier attribute for each operator and also maintains the operator selectivities for different values of that attribute.

— **Routing policy:** For the tuple under consideration, CBR finds the operator selectivities for that tuple by taking into consideration the values of the classifier attributes. It then routes the tuple probabilistically; the probability of routing the tuple to an operator is inversely proportional to its selectivity.

Implementing this policy requires many additional data structures over the above policies; furthermore the increase in the number of parameters necessitates higher sampling rates to learn the parameters. Hence the overhead of this technique is expected to be high. However when the operator costs are high (e.g., when using web indexes [43, 51]) and the goal is to minimize the invocations of the operators, this routing policy would prove to be beneficial.

Like the A-Greedy technique, post-mortem analysis of the execution of a selection ordering query using an eddy is fairly straightforward.

Post-mortem: The query execution using an eddy can be expressed as a horizontal partitioning of the input relation with each partition being executed using a serial order (even if the CBR policy is used, each tuple is still executed using a serial order). Depending on the routing policy being used, the partitioning may or may not be by the order of arrival.

4.3 Parallel and Distributed Scenarios

In recent years, there has been a growing interest in the parallel and distributed versions of adaptive selection ordering. This has been fueled both by an increased interest in parallel databases and Grid-like or peer-to-peer environments, as well as the advent of web-based structured information sources such as IMDB and Amazon [50, 111]. The key difference between these environments and the centralized environments is that "response time" becomes the metric of interest rather than "total work." As has been noted before in the parallel databases literature, this can change the nature of the optimization problem drastically, making it significantly harder in most cases [47]. So far, most of this work has addressed static versions of this problem, with the exception of the first paper discussed below.

Tian and DeWitt [115] considered the problem of designing tuple routing strategies for eddies in a distributed setting, where the operators reside on different nodes and the goal is to maximize the average response time or the maximum throughput. To avoid communication with a centralized eddy, the operators make the routing decisions in a distributed fashion as well. Tian and DeWitt present an analytical formulation as well as several practical routing strategies based on monitoring the selectivities and the costs of the operators, and also the queue lengths and the ticket counts.

Condon *et al.* [35] studied a similar extension of the selection ordering problem to the parallel scenario, where selection operators are assumed to be assigned to separate processors and the goal is to maximize the throughout of the system. It is easy to see that processing all tuples of the relation using a single order does not utilize the full processing capacity of the system, and multiple orders must simultaneously be used to achieve that. Condon *et al.* provide algorithms to find the optimal solution in this case, under the assumption that the predicates are independent of each other.

Srivastava *et al.* [111] consider this problem in the setting of web-based information sources (*web services*) and show how the problem of executing a multi-way join query over web-based sources reduces

to a *precedence-constrained* version of the selection ordering problem. In essence, each call to a web service can be thought of as executing an (expensive) selection operation over that tuple. They provide several algorithms to find the optimal (single) order in which to invoke the web services, even in presence of arbitrary precedence constraints.

Etzioni *et al.* [43] considered a somewhat related problem of information gathering on the Internet. The *information access* problem they pose is as follows: given a collection n information sources, each of which has a known time delay, dollar cost, and a probability of providing the needed piece of information, find an optimal schedule for querying the information sources. One option is to query the information sources in sequence, stopping when one source provides the information. Though most frugal, this option is also the most time-consuming. The other extreme option is to send requests in parallel to all the sources. This option has the lowest response time, but the (dollar) cost might be prohibitive. To reduce this problem to selection ordering, the notion of success here can be seen as equivalent to a predicate *falsifying* the tuple. In other words, a source providing the information required has same effect as a predicate not being true; both of these result in a stop to further processing.

4.4 Summary

Selection ordering is a much simpler problem than optimizing complex multi-way join queries, and this simplicity not only enables design of efficient algorithms for solving this problem, but also makes the problem more amenable to formal analysis. We discussed two adaptive query processing techniques for this problem: A-Greedy and eddies. The A-Greedy algorithm has two notable features: first, it takes into account the correlations in the data, and second, approximation guarantees can be provided on its performance. The second technique we saw, eddies, is more of an architectural mechanism enabling adaptivity that is not tied to any specific decision algorithm; we will see this again when we discuss eddies for general multi-way join queries.

As we will see in the next two sections, several problems that arise in adaptive execution of multi-way join queries can be reduced to selection ordering, and the algorithms we discussed in this section can be used unchanged in those settings. This is not particularly surprising; the similarities between selection ordering and multi-way join queries were noted by Ibaraki and Kameda [67] who exploited these similarities to design an efficient query optimization heuristic.

5

Adaptive Join Processing: Overview

Adaptation schemes for join queries are significantly more complicated to design and analyze compared to those for selection ordering for several reasons. First, the execution plan space is much larger and more complex than the plan space for selection ordering; this complexity arises both from the large number of possible join orders and from the variety of kinds of join operators themselves (binary vs. n-ary, blocking vs. pipelined etc.).

But more importantly, most of the join operators are "stateful": they have internal state that depends on the tuples processed by the operator. Hash join operators, for example, build hash tables on one (or possibly both) of their input relations. Due to this internal state, the execution environment itself plays a much larger role in determining the characteristics of query execution. In particular, whether the data is being read from a local disk, is arriving in the form of a data stream, or is being read asynchronously over the wide area network, can drastically change the nature and trade-offs of query execution.

To tackle this complexity, the research community has developed a diverse set of techniques designed for specific execution environments or

specific join operators, or in some cases, both. These adaptation techniques apply in different underlying plan spaces, though the particular subspace is rarely made explicit and is often hard to tease apart from the adaptation technique. We present these techniques in three parts, roughly based on the space of the execution plans they explore:

- **History-Independent Pipelined Execution (Section 6):** Our first space of plans consists of pipelines of non-blocking join and selection operators, with one further restriction: the state built in the operators during execution is largely independent of the adaptation choices made by the query processor. This space includes a fairly large class of traditional pipelined query plans, as well as many data stream query processors. The history-independence property allows us to reduce the query execution to selection ordering (with some caveats as discussed in Section 6.1), and the techniques discussed in the previous section can be used for adaptation among such plans.
- **History-Dependent Pipelined Execution (Section 7):** The second space is similar to the first in using pipelined operators only, but the operators may internalize state that depends on the routing choices made by the query processor. This internalized state (also called "burden of routing history" [40]) makes it hard to change from one query plan to another: an adaptive algorithm must carefully reason about the built-up state and ensure that while switching plans, no output tuples will be lost and no false duplicates will be produced.
- **Non-pipelined Execution (Section 8):** Finally, we cover plans with blocking operators like sort. Blocking operators provide natural stopping points to re-evaluate query execution decisions. Adaptation techniques proposed for non-pipelined plans have mostly focused on switching plans at these stopping points, and thus have a restricted set of possible adaptations. This simplifies the task of reasoning about the execution, and the adaptation technique can often invoke

traditional query optimizers to choose not only the initial plan but also the subsequent plans.

Note that most database systems typically use query execution plans with a mixture of blocking operators and pipelines; these can be adapted independently using different AQP techniques [18].

6

Adaptive Join Processing: History-Independent Pipelined Execution

The simplest-to-understand techniques for adaptive join processing apply to query execution where the past choices made by the query processor have no bearing on the next choice it has to make. We begin our discussion of such history-independent pipelined execution with the case of left-deep pipelined plans with a single *driver* (Section 6.1). We then consider query execution where multiple drivers are permitted, and discuss adaptation when using MJoins or unary operators called SteMs (Section 6.2).[1] Finally, we present a technique called *A-Caching* that uses intermediate result caches to alleviate one of the performance concerns with history-independent execution (Section 6.3). We conclude with a brief discussion of the advantages and disadvantages of history-independent schemes and a comparison of some of them using the adaptivity loop (Section 6.4).

6.1 Pipelined Plans with a Single Driver Relation

The first class of execution plans we consider is the space of left-deep pipelined plans where one relation is designated as the *driver* relation,

[1] When multiple drivers are permitted, query execution using binary join operators is not history-independent, and will be discussed in the next section.

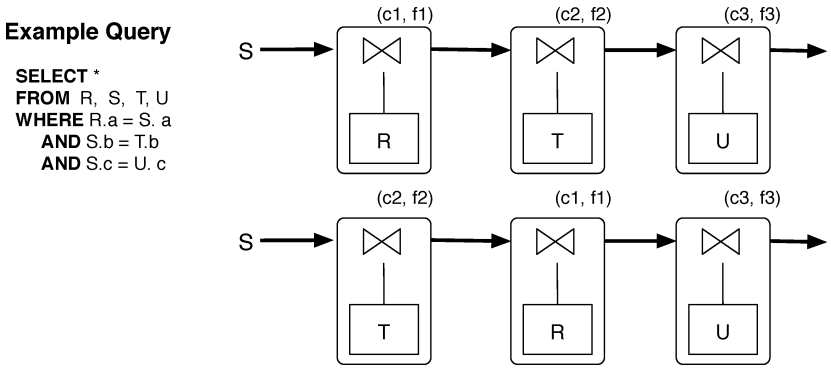

Fig. 6.1 Two possible orderings of the driven relations. c_i and f_i denote the costs and fanouts of the join operators, respectively.

and its tuples are joined with the other (*driven*) relations one-by-one (Figure 6.1). The joins are done as nested-loops joins or index joins or hash joins, depending on the access method used for the driven relations. If the driven relation is scanned, the join is a nested-loops join; if it is looked up via an existing index, the join is an index join; if it is looked up via a dynamically constructed hash index, it is a hash join. Figure 6.1 shows two example pipelined plans for a 4-relation query, with the driver relation S.

This type of execution is history-independent in the following sense. For a fixed driving tuple s, the behavior of the join operators (e.g., their join selectivities with respect to s) is independent of when s arrives, whether it is the first tuple from S, the last tuple, or any other. In other words, the joining with s is a side-effect free operation that does not alter the internal state of the join operators (the only state change is the cursor needed to generate join matches, which is transient).

We begin by discussing static optimization in this plan space and show how the problem of join ordering can be reduced to the selection ordering problem.

6.1.1 Static Planning

The space of pipelined left-deep plans is characterized by three dimensions, along which we could optimize and adapt: the choice of the driver

relation, the choice of the access method for performing each join, and the ordering of the joins.

6.1.1.1 Choosing Driver Relation and Access Methods

For an n relation query, there are at most n possible choices for the driver relation (some sources may not support scan access methods and thus may not be usable as driver relations). n is typically small enough that an optimizer can search exhaustively through the possible driver relations.

For each choice of driver relation, the join order can be chosen in a manner similar to selection ordering as we describe below. But choosing access methods is much harder. When there are multiple access methods on each driven relation, the plan space explodes combinatorially. The right choice of access methods depends on the selectivity (scans are preferred over indexes at higher selectivities), clustering, memory availability, and many other factors. Since the join costs depend on the access methods chosen, the choice of access methods cannot be made independently of the choice of join order. Hence static query optimization requires an exhaustive search (Section 2.1.4) through the combination of all access methods and join orders.

6.1.1.2 Join Ordering: Reduction to Selection Ordering

Assuming that the driver relation and access method choices have been made, the problem of ordering the driven relations bears many similarities to selection ordering. Several heuristic polynomial time query optimizers have been proposed that exploit this resemblance [67, 80].

Continuing with our example above, let $c_i, f_i, 1 \leq i \leq 3$ denote the probe costs and the *fanouts* of the join operators (number of output tuples per input tuple from the driving table) on R, T, and U, respectively. The execution cost of the two plans shown in Figure 6.1 can then be written as

$$Plan\ R \rightarrow T \rightarrow U: \quad Build\ Cost + |S| \times (c_1 + f_1 c_2 + f_1 f_2 c_3),$$

$$Plan\ T \rightarrow R \rightarrow U: \quad Build\ Cost + |S| \times (c_2 + f_2 c_1 + f_1 f_2 c_3),$$

where *Build Cost* is the (constant) cost of building the indexes on R, T, U (if needed). As we can see, the plan cost functions have the same form as the plan cost functions for selection ordering, and thus we can extend the techniques designed for solving selection ordering queries (e.g., rank ordering [67, 80]), to solve these queries as well. There are, however, several differences in the two problems that must be taken into account.

- Join fanouts may be larger than 1 (as opposed to selectivities that are always ≤ 1). However, the rank ordering algorithm can still be used to find the optimal order.
- The resulting selection ordering problem might contain precedence constraints. For example, if R is the driver relation, its tuples must be joined with S first before joining with T or U. This is because R tuples do not contain the join attributes needed to join with T or U. The algorithms for the non-precedence constrained case can be extended in a fairly straightforward manner to handle such cases [80].
- The join costs may not be the same for different driving tuples. For example, if the hash join algorithm is being used, and if the hash table does not fit into memory, then the probe cost depends on whether the corresponding partition is in memory or not [18, 117]. Blindly using a pipelined plan is not a good idea in such cases, and alternatives like XJoins [117], which employ sophisticated scheduling logic to handle large tables, should be considered instead.

  The techniques we discuss in this section largely assume a uniform cost structure with respect to the driving tuples, though the techniques explicitly adapt to costs changing over time.
- Finally, depending on the execution model and implementation of operators, subtle *cache effects* may come into play that cause this reduction to selection ordering to be suboptimal.

  Consider the first execution plan shown in Figure 6.1. Let f_1 be 10, and consider a tuple $s_1 \in S$ that produces 10 tuples

after the join with R, $s_1r_1, \ldots, s_1r_{10}$. According to the execution plan, these tuples should next be used to find matches in T. However, note that the matches with T depend only on the S component of these tuples (i.e., on s_1) and same matches (or none) will be returned from the probe into T for all of these 10 tuples. These redundant probe costs can be avoided, either by caching probe results [95], or by probing into the join operators independently, and doing a cross-product of the probe results to obtain the final results [10, 11, 18].

We briefly elaborate on the second approach. In the above example, s_1 will be used to probe into R, T, and U separately to find matching tuples; let the probes return $\{r_1, \ldots, r_{10}\}$, $\{t_1, \ldots, t_5\}$, and $\{u_1\}$, respectively. The join results are then obtained by taking a cross-product of $\{s_1\}$, and these three sets of tuples (to produce 50 result tuples). The approach can be generalized to queries with non-star query graphs in a fairly straightforward manner.

A simple reduction to selection ordering cannot easily capture such cache effects, though an alternative reduction can be used in certain special cases (e.g., star queries). We refer the reader to [41] for a more detailed discussion of this issue.

6.1.2 Adapting the Query Execution

In this model, we can adapt along any of the three dimensions of the plan space.

Adapting the Choice of Driver Table or Access Methods: Changing the driver table or the access method during the query execution typically requires complex duplicate handling procedures that can be computationally expensive. Switching access methods during execution can have high initial costs and also requires extensive changes to the query execution framework. For these reasons, there has been little work on adapting the choice of driver table or the access method during query execution. We discuss some techniques in Section 6.2.

For access method adaptation, one adaptation scheme proposed for the DEC Rdb system [3] was *competition*: run multiple access methods (AMs) in parallel for a while, monitor the selectivities and costs, and then pick one AM on each driven table. There are two limitations to this scheme. First, it introduces competition among AMs on each table individually, but does not explore the AM combinations. Second, after the competition phase completes, finalizing on an AM involves duplicate elimination, which can be a significant overhead for the rest of the query.

Adapting the Join Order: For a fixed choice of driver table and AMs, pipelined query plans are identical to selection ordering (modulo the cache effects). Thus the adaptive algorithms discussed in Section 4 can be used unchanged, aside from modifications for obeying the precedence constraints, to adapt the join order. For instance, Babu *et al.* [10] discuss how their A-Greedy algorithm (Section 4.1) can be extended to handle this case. Eddies can also be used to adapt the join order.

The similarities to selection ordering also make it easy to analyze the behavior of these techniques in retrospect, as long as the driver is not changed mid-execution.

Post-mortem: Adaptive history-independent query execution for a fixed driver can be expressed as a horizontal partitioning of the driver relation, with each partition being executed using a different left-deep pipelined plan.

6.2 Pipelined Plans with Multiple Drivers

The single driver table approach described above seems simple, but in fact choosing the driver in advance of query execution imposes serious restrictions. Perhaps most importantly, it requires all but the driver relation to be available in entirety before the execution can begin; this may lead to unacceptable delays in wide-area environments, and may be impossible when processing data streams.

The driver table is also often one of the highest-impact choices an optimizer makes. The plan execution time depends linearly on the driver table size, whereas it usually depends only sublinearly on other

table sizes (due to static or on-the-fly indexes). Choosing the right driver up-front is tricky because the optimizer has to compare not just the candidate driver table sizes but also the sizes of their join fanouts with other tables. Additionally, having a single driver forces the join algorithm to be a nested-loop or index join. Sort-merge and symmetric hash join need multiple drivers. A Grace or hybrid hash join can be viewed as having a single driver, but swapping the inner and outer requires changing the driver.

In this section, we will see several adaptation techniques that alleviate these problems, but still result in history-independent query execution. We start with discussing how to adapt when using a n-ary symmetric hash join operator (MJoin). We then discuss the issue of driver choice adaptation in more detail, and present a unary operator called *SteM*, which can be used along with an eddy to not only adapt driver tables, but also other optimizer choices like access methods. We conclude with a discussion of post-mortem analysis of these techniques, and the performance concerns with them.

6.2.1 n-ary Symmetric Hash Joins/MJoins [120]

Recall that an n-ary symmetric hash join operator builds hash tables on each relation in the query on each join attribute, and executes the query by routing the input tuples appropriately through these hash tables (Section 3.1.3). This execution is history-independent because the internal state inside the hash tables is determined solely by the source tuples that have arrived so far, and does not depend on the choices made by the router. The decision logic in the operator is encapsulated in the *probing sequences*, which specify the order in which to probe the hash tables for tuples of each driver table.

Choosing the Initial Probing Sequences: Choosing the probing sequence for a driver relation is identical to choosing the join order for pipelined plans with a single driver, as long as the hash tables fit in memory (Section 6.1.1.2). Hence, the join ordering algorithms discussed in Section 6.1 can be used unchanged for choosing the probing sequences (for each driver relation separately). Viglas *et al.* [120] propose using the rank ordering algorithm (Section 2.1.2) for this

purpose. The Greedy algorithm (Section 2.1.2) can also be used for this purpose if the correlations across the join operators are known or monitored.

Adapting the Probing Sequences: Similarly, we can use the adaptive techniques presented in Section 4 for adapting the probing sequences. Each of the probing sequences must be adapted separately, though some statistics can be shared between these. Furthermore, as discussed in Section 6.1.1.2, the probing sequences may have to obey precedence constraints depending on the query. Babu *et al.* [10], in the context of the STREAM project, use the A-Greedy algorithm (Section 4.1) for this purpose.

6.2.2 Driver Choice Adaptation

MJoins as proposed in [120] deal with streaming data sources, where the choice of driver is not under the control of the query processor. The ability to control and adapt the driver tables being used can however be immensely useful, and can be exploited for a variety of purposes:

- *To react to stalls* in asynchronous data sources by switching to other drivers (this is similar to query scrambling, which we discuss in Section 8).
- *To react to an unexpectedly large driver table* by switching to another driver. For example, consider a join involving two inputs R and S. If the input sizes are very different, an optimizer choosing a hash join has to be careful about which table it chooses as the build side, because that changes the memory consumption, I/O, number of probes, and thereby elapsed time as well. A simple adaptive strategy that avoids this risk is to repeatedly alternate between two drivers until one of them returns EOF, and then switch to a regular asymmetric hash join. Observe that this strategy incurs a competitive cost of at most twice the best in-hindsight join order.
- *To react to changing user requirements:* An important contrast between a symmetric hash join and an asymmetric one (e.g., Grace hash join) is a tradeoff between interactivity and

completion time: symmetric hash join gives pipelined results, but consumes more memory and is less efficient. The query processor can *switch from the former to the latter* as the query progresses, by alternating drivers initially and gradually converging on one, so that users get interactive results early but the query still completes quickly [95].

- *To switch drivers based on improved selectivity estimation:* Another reason to switch driver tables from R to S is if the query processor can use a selective predicate on S to do index lookups into R. The selectivity of the predicate on S may not be known up front, but can be estimated after a hash table on S has been built. A recent system called SHARP [18] describes such a technique to adapt choice of driver tables (and access methods at the same time).

6.2.3 State Modules (SteMs)

Observe that the last two adaptations of the previous section change the join algorithm during execution, from a symmetric join with multiple drivers running concurrently to an asymmetric one. Raman *et al.* [95] call this process *hybridization*. They go further, and propose a new unary operator called a SteM (a State Module), to be used with an eddy, that allows greater adaptation flexibility than an MJoin, and in particular allows flexible adaptation of join algorithms, access methods, and the join spanning tree. We give a brief overview of SteMs in this section.

Recall the MJoin logic for processing a driver tuple r from table R:

— r is built into hash indexes on R
— r is probed into the hash indexes on other tables.

The essential idea of SteMs is to relax this logic threefold:

— relax the atomicity and ordering of the build-probe operations
— allow probes into not just the hash indexes (called SteMs below) that are being built as part of this query, but also into other *pre-built* indexes (called AMs below)

— allow these lookups into pre-built indexes to supply driver tuples, in addition to table scans or streams

To realize these relaxations, we need two new kinds of operators.

An *Access Module (AM)* encapsulates a single index over a data source. An AM on table R accepts *probe tuples* p and outputs *matches* – $\{r \in R \mid (r,p)$ satisfies query predicates$\}$. Notice that unlike usual indexes, AMs *do not concatenate matches with probes*; such concatenation will be performed only by SteMs. As a result, we can treat the original table scan or stream on R as an AM that is outputting tuples in response to an initializing probe with a virtual seed tuple that matches all tuples in R.

A *State Module (SteM)* is a data structure built on-the-fly during the query execution that contains homogeneous tuples (i.e., having the same schema). SteMs support insert (build), search (probe), and optionally delete (eviction) operations. A SteM accepts *build tuples* and *probe tuples*. Build tuples are just added to the tuples in the SteM. Upon receiving a probe tuple p, a SteM returns matches, concatenated with p, plus the probe tuple p itself. For example, a hash table built over a base table as part of query execution is a SteM.

6.2.3.1 Query Execution and Hybridization

To execute a query, the query "optimizer" simply instantiates an eddy, a SteM on each base table, and an AM on every available table access method (scan and index). The query optimizer chooses up front neither a spanning tree over the join graph, nor a join order, nor any join algorithm, nor access method. All of these are determined by the way the eddy chooses to route tuples, as we describe next.

MJoins using SteMs: Observe that SteMs can be used to perform an MJoin: we simply break the MJoin box in Figure 3.2 and expose the hash-table boxes inside as three SteMs. Figure 6.2 illustrates the tuple exchange that happens between the eddy and the SteM (we use a 3-table query to keep the figure concise). But SteMs can do more than MJoins, because the tuples do not have to be routed using fixed build-then-probe logic. Instead, [95] introduce a notion of *routing constraints*,

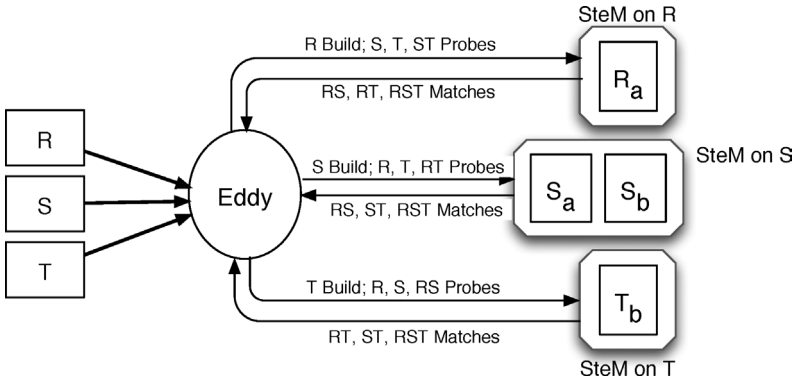

Fig. 6.2 SteMs used to do an n-ary SHJ for $R \bowtie S \bowtie T$.

that specify not a particular n-ary join algorithm but rather a characterization of the space of correct n-ary join algorithms. For example, a slight variation on this dataflow results in an index join, as we see next.

Asynchronous Index Joins using SteMs: A simple way to perform an index join is to replace one of the scan AMs in Figure 6.2, say that on R, with an index AM, as shown in Figure 6.3. Now, R tuples enter the dataflow only in response to probes from S or T. Notice that the index AM_R returns non-concatenated matches (i.e., tuples with only the R fields). These R matches will concatenate with the corresponding

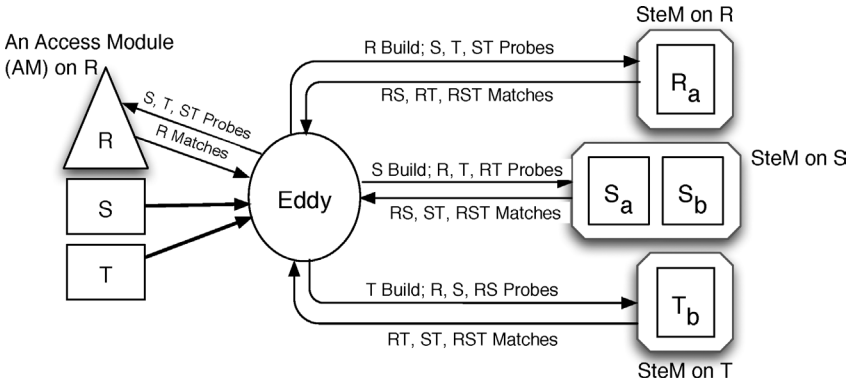

Fig. 6.3 SteMs used to do an Asynchronous index join for $R \bowtie S \bowtie T$.

probes by a hash-join logic: the matches will either probe SteMs on S, T, or be built into the SteM on R and then be probed by S and T tuples. In the literature, this is called an asynchronous index join [51], and is especially useful on queries over asynchronous data sources such as deep web sources.

Competitive Access Methods and Hybridization of Join Algorithms: We saw a few examples of join algorithm hybridization in the last section, where an n-ary SHJ switched from one algorithm to another. SteMs allow further hybridization involving the access methods themselves. Observe that Figures 6.2 and 6.3 are almost identical, except for the AM on R. Consider what happens if we add both kinds of AMs (index and scan) on R to the same dataflow. Now, the eddy will route some of the S and T tuples to the SteM on R, and others to the index on R. Essentially, it is running an MJoin and an asynchronous index join concurrently. After monitoring costs and selectivities for a while, the eddy could choose an AM which works best, say the scan AM. This means the eddy has picked a hash join after initially experimenting with an index join. If during query execution the hash table SteM on R runs out of memory, then the eddy can start routing S and T tuples to the index AM on R, i.e., adapt to an index join. But the existing tuples in the SteM on R still remain, so that work is not wasted.

Raman *et al.* [95] demonstrate several examples of such hybridization and their benefits on deep Web data sources.

6.2.3.2 Routing Constraints

The flexibility that SteMs provide comes with a price. Not all routings lead to correct results, because of two kinds of problems. First, running multiple access methods concurrently can result in incorrect numbers of duplicate output tuples. Second, asynchronous routing can result in missing results. Raman *et al.* [95] present detailed routing constraints that avoid these problems, and prove that any routing satisfying these constraints results in a correct, hybridized, join algorithm.

The SteM routing constraints are too involved to explain in this survey. But it is important to note that the routing constraints are described in terms of the routing decisions themselves, and not in

terms of the state built in the operators. This allows the correctness of the join algorithm to be enforced by an external router like an eddy, without breaking the encapsulation offered by the separate operators.

6.2.4 Post-mortem Analysis

The execution of a query using MJoins or SteMs can be captured using the notion of horizontal partitioning (Section 3.3). Both these techniques implicitly divide the source relations into partitions, with the number of partitions dictated by both the interleaving of tuples from different relations and the routing decisions made for the tuples. Roughly speaking, a contiguous block of tuples (by order of arrival) from a source relation, with all of them routed identically through the operators, forms a partition of that relation. Thus, if there is high interleaving of tuples or if the routing decisions are changed on a per-tuple basis, the number of partitions might be very high.

As an example, let the tuple arrival order for our example 3-relation query, $R \bowtie S \bowtie T$, be $r_1, r_2, r_3, s_1, s_2, t_1, t_2, t_3, r_4, t_4$, and let us assume that all tuples of a relation are routed identically. In that case, an MJoin operator will implicitly partition R into 2 partitions, $R_1 = \{r_1, r_2, r_3\}$ and $R_2 = \{r_4\}$, T into 2 partitions $T_1 = \{t_1, t_2, t_3\}$ and $T_2 = \{t_4\}$, whereas S will form a single partition containing s_1 and s_2 (Figure 6.4).

The four resulting subqueries, $R_1 \bowtie S \bowtie T_1$, $R_2 \bowtie S \bowtie T_1$, $R_1 \bowtie S \bowtie T_2$, and $R_2 \bowtie S \bowtie T_2$, will then be executed using left-deep pipelined plans. The first subquery will be executed as $(T_1 \bowtie S) \bowtie R_1$, with T serving as the driver relation (since its partition arrived last among the partitions involved in the subquery), whereas the second subquery will be executed as $(R_2 \bowtie S) \bowtie T_1$, with R serving as the driver relation. The remaining two subqueries will be executed simultaneously as $(T_2 \bowtie S) \bowtie (R_1 \cup R_2)$, with T serving as the driver relation.

Note that, the eddy also executed $S \bowtie R_1$ (when S arrived); however, this join was wasted since the intermediate results (if any) are not stored or reused.

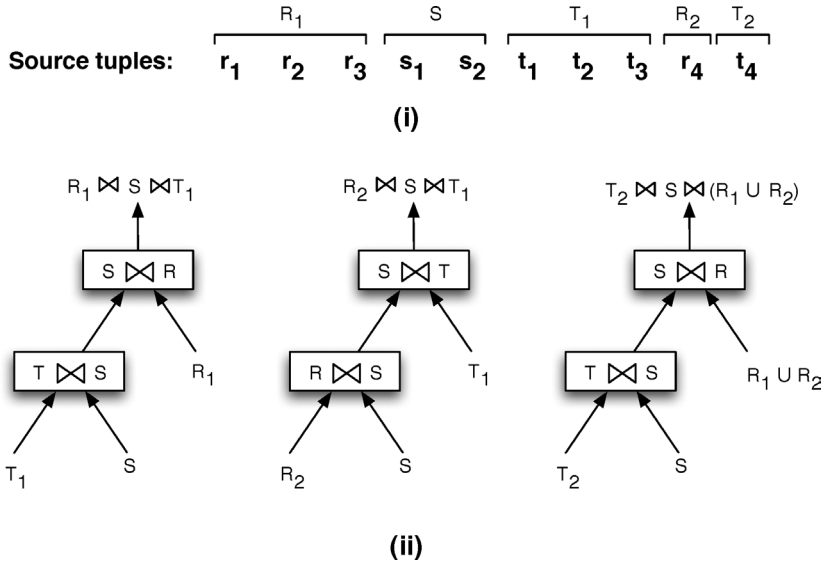

Fig. 6.4 (i) MJoins or SteMs implicit partition the data by order of arrival and routing decisions; (ii) subqueries are evaluated using left-deep pipelined plans.

Post-mortem: The execution of a query using an MJoin operator or using an eddy with SteMs, can be seen as a horizontal partitioning of the input relations by order of arrival, with the subqueries being executed using left-deep pipelined plans.

6.2.5 Discussion

Although these strategies are easier to reason about and to design policies for, they tend to suffer from suboptimal performance compared to using a tree of binary joins or using an eddy with binary join operators (Section 7.2). The main reason is that these strategies do not reuse any intermediate tuples that may have been produced during query execution. This leads to two problems:

- **Re-computation of intermediate tuples:** Since intermediate tuples generated during the execution are not stored for future use, they have to be recomputed each time they

are needed. In the above example, the first subquery produces an intermediate result $(T_1 \bowtie S)$, which could be used for evaluating the second subquery, $R_2 \bowtie S \bowtie T_1$. However, since that intermediate result is not materialized, the second subquery must be computed from scratch.

- **Constrained plan choices:** Equally importantly, the query plans that can be executed for any new tuple are significantly constrained. In our example query, any new R tuple, r, that comes in at time τ must join with S_τ (the tuples for S that have already arrived) first, and then with T_τ. This effectively restricts the query plans that the router can use for this tuple to be $(r \bowtie S_\tau) \bowtie T_\tau$, even if this plan is known to be sub-optimal. Using an alternative plan, e.g., $r \bowtie (S_\tau \bowtie T_\tau)$, requires the ability to materialize the intermediate results.

6.3 Adaptive Caching (A-Caching)

Adaptive caching [11] was proposed to solve some of the performance problems with MJoins by enabling the router to explicitly *cache* intermediate result tuples. It thus allows exploring and utilizing the spectrum between the two extreme approaches for evaluating multi-way join queries: (a) using an MJoin operator, and (b) using a tree of binary join operators. The trade-off between these two approaches was also considered, in a static setting, by Viglas *et al.* [120].

The execution model here is not entirely history-independent since the decisions about which intermediate results to cache affect the execution state. However these intermediate result caches can be considered as *soft state*; they are used opportunistically and can be thrown away at any point without any correctness implications. This is not true of the techniques that we discuss in the next section.

The adaptive caching approach uses the MJoin operator as the basic operator in the system. It, however, differs from the MJoins approach as follows:

- A-caching uses the *A-Greedy* approach (Section 4.1) to adapt the probing sequences used for execution. This is done independently for each streaming relation.

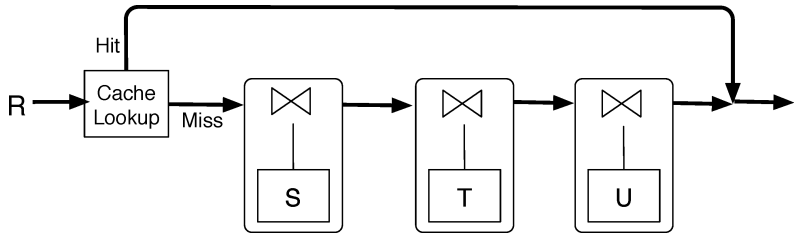

Fig. 6.5 Adaptive caching.

- Once the probing sequences are chosen, A-caching may decide to add intermediate result caches in the middle of the pipeline.

Consider the example query in Figure 6.1, and let the current choice for the probing sequence for R tuples be $S \rightarrow T \rightarrow U$ (Figure 6.5). In other words, a new R tuple, say r, is first joined with S, and then with T and finally with U. If two consecutive (or close by) tuples r_1 and r_2 have identical values of the attribute a (the join attribute for the join with S), then the resulting $S \bowtie T \bowtie U$ matches for the probes will be identical. However, the MJoins approach is unable to take advantage of such similarities, and will re-execute the joins.

The key additional construct of the A-Caching approach is an intermediate result cache. An intermediate result cache, $C^X_{\mathbf{Y}}$ is defined by: (a) X, the relation for which it is maintained, and (b) $\mathbf{Y}$, the set of operators in the probing sequence for which it stores the cached results. For example, the cache for storing the $S \bowtie T \bowtie U$ matches for the R tuples will be denoted by $C^R_{\bowtie_a S, \bowtie_b T, \bowtie_c U}$. The cached entries are of the form (u, v) where u denotes the value of the join attribute (attribute $R.a$ in this case), and v denotes the result tuples that would be generated by probing into the corresponding operators. A cache might not contain results for all possible values of the join attribute, but if it does contain at least one result for a specific value of the join attribute, it must contain all matching results for that value. In other words, if there is an entry $(a_1, (STU)_1)$ in the cache, then $(STU)_1$ must be equal to $\sigma_{S.a=a_1}(S \bowtie T \bowtie U)$.

Figure 6.5 shows how a cache lookup is introduced in the pipeline of R tuples for the above query. Given such a cache, when a tuple arrives at that point in the pipeline, the cache is consulted first to determine if the results are already cached, and if they are, the probes into S, T, and U can be avoided. On the other hand, if there is a miss, the probing continues as before.

Cache Update: There are two ways a cache might be updated. First, the results found by probing into the operators when there is a cache miss can be inserted back into the cache. Second, when new T and U tuples arrive (or existing T or U tuples need to be deleted because of sliding windows on those relations), the cache needs to be updated to satisfy the constraint described above. This latter step can be computationally expensive and hence the choice of which caches to maintain must be made carefully.

Choosing Caches Adaptively: Babu *et al.* [11] advocate a three-phase approach to the problem of adaptively executing a multi-way join query.

- The A-Greedy algorithm for adapting the probing sequences is at the top of the loop, and is oblivious to the existence of caches. It makes its decisions based solely on the operator selectivities.
- Given the probing sequence, caches are selected adaptively to optimize the performance.
- Memory is allocated adaptively to the caches selected for materialization.

Post-mortem: A post-mortem analysis of A-Caching is somewhat trickier than MJoins because the caches may be created or destroyed arbitrarily. Generally speaking, the use of caches will increase the sharing between the execution of different subqueries. For instance, let S_1, T_1, and U_1 denote the tuples of relations S, T, and U that have already arrived, and let r be a new tuple from relation R. Without any caches, r will be processed using a left-deep plan $(((r \bowtie S_1) \bowtie T_1) \bowtie U_1)$. However, if the cache $C^R_{\bowtie_a S, \bowtie_b T, \bowtie_c U}$ was being maintained (Figure 6.5), then

this tuple would be processed as $r \bowtie (S_1 \bowtie T_1 \bowtie U_1)$ instead (assuming the cache is complete).

Post-mortem: The execution of a query using A-Caching can be seen as a horizontal partitioning of the input relations by order of arrival, with the subqueries being executed using either simple left-deep pipelined plans or complex bushy plans depending on the caches that are maintained.

6.4 Summary

In this section, we saw several techniques for adaptation of multi-way join queries when the execution is pipelined and history-independent. Table 6.1 compares two — StreaMON and SteMs, using the adaptivity loop framework. StreaMON is a specific adaptive system that chooses the probing sequence using the A-Greedy algorithm of Section 4.1, while SteMs enable an adaptation framework that allow a rich space of routing policies.

The biggest advantage of history-independent execution is that the plan space is much easier to analyze and to design algorithms for. Consider a case when a total of N tuples have been processed by the system, and let M denote the number of choices that the executor faced.

Table 6.1 Comparing some representative history-independent adaptive query processing schemes using the adaptivity loop.

MJoins with A-Greedy (StreaMON [13])
Measurement: Conditional selectivities using random sampling.
Analysis and planning: Detect violations of the *greedy invariant*; re-plan using the Greedy algorithm (Section 4.1).
Actuation: By changing the probing sequences.
SteMs [95]
Measurement: Selectivities and fanouts of lookups into AMs, SteMs.
Analysis and Planning: Can use any routing policy subject to the routing constraints, e.g., A-Greedy to adapt probing sequences or various heuristics described in [95] to adapt driver.
Actuation: By routing tuples using an eddy, subject to routing constraints.

For adaptive techniques like eddies and other routing-based approaches, M could grow much faster than N. If the execution state depended on the choices made by the executor, then, even if all the choices were binary (which is unlikely), the number of different possible execution states could be as high as 2^M. On the other hand, if the state depended only on the tuples arrived into the system, there is only one possible execution state that is predictable given the input tuples. This not only insulates the future choices made by the query processor from the past choices, but also makes it easy to design routing algorithms. We will revisit this issue in the next section when we consider execution models that are not history-independent.

Second, *actuation*, switching between plans, is easier since the execution state at any point is independent of the plan being used till that point. Actuation is trickier for AQP techniques that do not maintain this property (Section 7.3).

Finally the execution state, being well-defined, is easier to manipulate. This is especially important in data streams environment where tuples may have to be deleted from the execution state.

However, as discussed in Section 6.2.5, these strategies tend to suffer from suboptimal performance when there are multiple drivers [40]. The primary reason behind this is that these strategies, with the exception of A-Caching, do not reuse intermediate results produced during execution. In the next section, we present several other techniques that materialize and reuse the intermediate tuples produced during execution, and discuss the tradeoffs between these two approaches.

7

Adaptive Join Processing: History-Dependent Pipelined Execution

We now turn our focus to the space of pipelined query plans where the state built up inside the join operators during query execution depends on the optimization or adaptation choices made by the query processor. This includes the class of traditional query plans where a tree of fully pipelined (symmetric) binary join operators is used to execute a query with multiple driver relations; the state accumulated inside the operators depends on the join order being used to execute the query.

We begin our discussion with *corrective query processing*, which uses a conventional (binary) query plan at any time, but may use multiple plans over the entire execution (Section 7.1). We then revisit the eddies architecture, and consider adaptation when binary pipelined join operators are used with eddies (Section 7.2). In Sections 7.3 and 7.4, we present the STAIR operator and the CAPE adaptive query processor, respectively, both of which allow explicit state manipulation during query processing. Finally, we compare these schemes using the adaptivity loop, and conclude with a discussion of pros and cons of history-dependent schemes (Section 7.5).

7.1 Corrective Query Processing

The *corrective query processing* [73][1] approach (which we abbreviate as CQP) exploits cost-based optimization to "steer" a query engine that devotes the majority of its resources to efficient exploitation of data and production of results. Most exploration of alternatives is done through cost estimation rather than explicit execution. The basic corrective query processing model attempts to (1) separate decisions relating to scheduling from those regarding computation cost, and (2) support a broader range of queries than most adaptive techniques, including those with aggregation and nested subqueries.

CQP considers the adaptive query processing approach to be one of *horizontal partitioning*. It begins with an initial query plan that looks much like one in a conventional DBMS, except that it relies more heavily on pipelined operators. Figure 7.1 shows the execution of a three relation query, $F \bowtie T \bowtie C$, with a group by operator aggregating at the

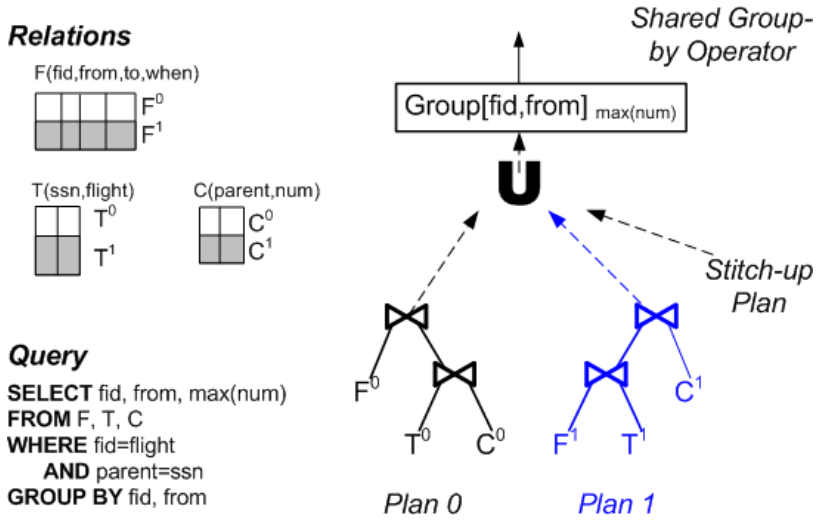

Fig. 7.1 An aggregation/join query as a combination of two plans plus a final stitch-up plan (see Section 7.1.2).

[1] Originally referred to as "convergent query processing" in [71], but renamed to more accurately convey the fact that one cannot guarantee convergence in an adaptive system, given an arbitrary adversary.

end. The CQP engine chooses an initial plan, and begins executing this first plan, requesting data from remote sources in the form of (finite) sequential streams. As execution progresses, the CQP engine monitors cost and cardinality and performs re-estimation to determine whether the initial plan is performing adequately. If not, the plan can be reoptimized based on extrapolation from the data encountered to this point. If a more promising plan is discovered, the current plan's input is suspended and it completes processing of the partition of data it has seen so far (represented by F^0, T^0, C^0). Now, the new plan begins execution in a new *phase* (Plan 1 in the figure), over the remaining data from the source streams (F^1, T^1, C^1). The plan replacement process repeats as many times as the optimizer finds more promising alternative plans, until all source streams have been consumed. Finally, a *stitch-up* phase takes the intermediate results computed in all previous phases and combines them to return the remaining results. (We note that the stitch-up phase is not strictly required by the CQP approach, as cross-phase computation could be integrated into any prior phase — but the initial work in [73] chose this implementation.) Stitch-up plan generation is based on the algebraic properties of the query plan and we discuss it in more detail in Section 7.1.2.

We can examine CQP through the lens of Section 3.2 as follows.

Measurement: CQP performs measurement by having all query operators expose their internal state, including the number of tuples processed, number of tuples returned, and number of tuples in intermediate state. As with mid-query reoptimization [75], it also supports specialized information gathering operators: in addition to the histogram operators proposed in that work, CQP considers tuple-order and unique-value detectors [73].

Analysis: The analysis stage of corrective query processing greatly resembles that of a conventional query optimizer's cost modeler, with a few notable exceptions. First, the cost estimate is for processing the *remainder* of source data, plus the required overhead for switching plans. This is done by extrapolating how many tuples are remaining at each source, and assuming that selectivities will remain constant over the remainder of the plan. (As with any adaptive scheme, this heuristic can be mis-

led by variations in the data. However, CQP has been experimentally shown to be relatively insensitive even to Zipf-skewed data. The main issue is that the arrival order must be relatively correlated with the skew.)

A second consideration is that in practice, different join attributes are often correlated rather than independent: hence, the selectivity estimation from the currently executing query plan ($T \bowtie C$, $F \bowtie TC$ in Figure 7.1) is not necessarily informative about the selectivities used by other query plans (e.g., $F \bowtie T$). The CQP work partly compensates for this by modeling the selectivities at the logical subexpression level, rather than at the operator level: thus, there is a single selectivity estimate for each logical expression independent of its evaluation plan (e.g., $F \bowtie T \bowtie C$ rather than $F \bowtie (T \bowtie C)$ and $(F \bowtie T) \bowtie C$). CQP adopts several heuristics, based on averaging the selectivities of known "similar" expressions, and avoiding expressions with join predicates that are known to have cross-product-like behavior. The general goal is to be conservative in choosing alternative plans.

Planning: Query planning is done by a top-down dynamic programming optimizer, which supports nested subqueries and aggregation as well as traditional single-SQL-block queries. The optimizer remains resident even during query execution and is periodically invoked as costs are monitored. It produces bushy plans that maximize pipelining, for three reasons: (1) the end goal is to support interactive applications, (2) it relies on their scheduling flexibility to adjust to delays, and (3) pipelined plans facilitate incremental actual cost and cardinality information gathering.

As is suggested in 7.1, a query in CQP is executed as the union of a series of plans, one per phase. (For sorted or aggregate queries, a final sort or aggregate operator may be performed on the output of this union. See Section 7.1.2.) Each of the per-phase plans computes the answers over a different horizontal partition of the data. The stitch-up plan then performs cross-phase joins.

Actuation: Actuation for CQP is done using a query engine that includes pipelined implementations not only of the hash join, but also *windowed* sort and group-by operators. The windowed sort is essentially a

fixed-length priority queue that does "best effort" reordering of tuples, such that if there are only local permutations, these will be sorted in mid-stream. The windowed group-by operator (called an *adjustable-window pre-aggregation* in [73]) divides the input into a series of non-overlapping windows: a local (partial) group-by operation is performed over each window; if the group-by reduces the number of tuples, its window size is increased (up to a maximum value specified by the optimizer), otherwise it is decreased until it is shrunk to a operate on a single tuple at a time. Both of these windowed operators are useful in that they perform at least partial work (and gather information) on a part of the input stream, while still preserving pipelined behavior (and thus CQP's ability to gather information and change the plan). Of course, a final merge-sort or post-processing group-by will need to be performed on the output at the end.

One other aspect worth noting is that the CQP approach scales to workloads that are larger than memory [71]: in fact, there is a natural parallel between the horizontal partitioning done on the data by phased execution and the overflow resolution schemes used by hash join algorithms. We now discuss some of the major design points of the CQP approach.

7.1.1 Separating Scheduling from Cost

A key consideration emphasized in [73] is that joins, as stateful operators, may dramatically amplify the effects of "bad" scheduling decisions. A collection of tuples representing an intermediate result must be joined with *every* future tuple seen by a join operator, and the result may be the creation of additional state for the parent operator, etc. Thus, query processing has two main cost-related aspects: *scheduling* tuple processing, which determines the order in which plan operators get CPU cycles, and which can affect query response time but not overall plan cost; and *operator ordering* with respect to the computation, which affects the size of intermediate results and thus overall plan cost.[2]

[2] A third aspect, *tuple ordering*, which might prioritize some tuples over others [97], allows for asymmetric treatment of output results, but is not considered here.

The eddy scheduling scheme combines both of these factors in its tuple routing; in contrast, CQP separates them under the belief that aggressively scheduling a less-desirable plan ordering is generally undesirable even in the presence of intermittent delays.

CQP relies on pipelined hash joins to perform scheduling, in a way that masks most I/O delays. Its focus is solely on cost-based plan selection, ensuring the plan is changed at consistent points, and any necessary cross-phase stitch-up computations.

7.1.2 Post-mortem Analysis

As a scheme for adaptively inserting horizontal partitioning into an executing query plan, CQP offers a great deal of flexibility. It exploits relational algebra properties relating to distribution of selection, projection, join, and other operators over union.

Selection and projection distribute trivially. The more interesting algebraic equivalences are for join and aggregation.

Join: We can take any join expression over m relations, each divided into n partitions, and write it as

$$R_1 \bowtie \cdots \bowtie R_m = \bigcup_{1 \le c_1 \le n, \ldots, 1 \le c_m \le n} (R_1^{c_1} \bowtie \cdots \bowtie R_m^{c_m}),$$

where $R_j^{c_j}$ represents some subset of relation R_j. This is equivalent to the series of join expressions between subsets that have matching superscripts:

$$R_1^i \bowtie \cdots \bowtie R_m^i, \quad 1 \le i \le n$$

Plus the union of all remaining combinations:

$$\{t | t \in (R_1^{c_1} \bowtie \cdots \bowtie R_m^{c_m}),\ 1 \le c_i \le n,\ \neg(c_1 = \cdots = c_m)\}$$

Note that this two-part version of the overall expression exactly corresponds to the example of Figure 7.1. The two phases in the figure correspond to our join expressions with matching superscripts.

The stitch-up phase performs the final union of combinations: in general, this plan must consider all possible joins across partitions that were not covered by the earlier phases. The work of [73] proposes a

specialization of the join algorithm to make this more efficient: the *stitch-up join* takes the intermediate state generated from each phase, as well as a specification of which sub-results have already been generated by prior phases. The stitch-up join then joins the remaining combinations, also annotating each tuple with which partitions were incorporated. The process, while somewhat complex, ensures that prior work is not repeated and that no duplicates are produced.

Aggregation: From the traditional query optimizer literature, it is well known that sometimes it is more efficient to *pre-aggregate* tuples before they are fed into a join operator, thus reducing the cost of the join [26]. Typically a final group by operator is required at the end; this operator performs final merging of the partially aggregated tuples.

CQP's *adjustable-window pre-aggregation operator* achieves this in a flexible way. The most common aggregate operators — sum, max, min, and count — distribute over union. Average can also be obtained, by performing a sum and count at each intermediate step, and then dividing them only at the final aggregation step. The pre-aggregation operator is free to operate over windows that perform horizontal partitions over its input, and to adjust the window size as conditions necessitate.

Other Common Operators: Naturally, union distributes over another union, so it is trivial to incorporate unions into CQP query plans. Outerjoins are somewhat more complex: here, CQP must divide between tuples that successfully join and those that do not. For non-stitch-up phases, CQP will use a conventional join, propagating any answers that match. Only during the stitch-up phase, when it has access to all data in the source relations, can it actually perform the outerjoin, returning any missing join results as well as tuples that do not successfully join. Existential quantification can also be accommodated in the CQP model, although depending on the constraints on the data, it may require eliminating duplicates introduced by segmenting plan execution into phases.

Limitations: The CQP model is quite flexible in supporting a broad range of operators. However, as discussed in [71], there are certain

operations that do not fit very well into an adaptive, stream-based processing model. Most non-monotonic operators (except for outerjoin and aggregation with average, as discussed previously) offer little possibility of incremental computation, except under query execution models that explicitly support revocation, as with [107]. Current CQP implementations do not utilize such a model, and hence they do not consider universal quantification, relational difference, or the NOT EXISTS predicate.

Post-mortem: Corrective query processing internally models its behavior as horizontal partitioning, and thus a query execution using CQP can be viewed as a horizontal partitioning of data by order of arrival.

7.1.3 A Generalization: Complementary Joins

The work of [73] proposes an alternative use of horizontal data partitioning, where the query plans are static but the data is routed to each in a dynamic fashion. The general goal is to exploit situations in which data is "mostly sorted" along dimensions where a merge-join would be advantageous over a hash join.

The key idea is to separate each join into two *complementary* implementations, one a merge join and one a hash join, preceded by a "router" that determines which join gets each subsequent tuple. See Figure 7.2, the two join implementations share state, dividing into four hash tables (two for each relation, designated h(R) and h(S) in the figure), each with the same number of buckets. Data from input relation R is routed to one of the joins based on whether it conforms to the ordering of the merge join. If a tuple that arrives is not in the proper sequence with its chronological predecessor, it is joined within the pipelined hash join in standard fashion. If it is ordered, the merge join consumes and joins it, and then stores it in the merge join's local hash table for R. Data from relation S is processed similarly. Once all data is consumed, a mini-version of stitch-up is performed: hash table R from the pipelined hash join is combined with hash table S in the merge join, and vice-versa.

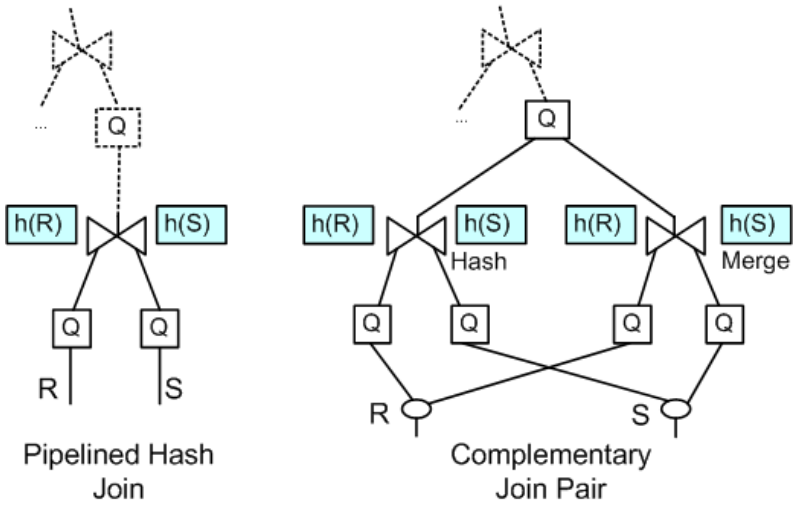

Fig. 7.2 Internals of pipelined hash join vs. complementary join pair; "Q"s represent queues between threads.

7.1.4 Open Problems

The CQP work leaves open a number of avenues for future exploration. The first question is precisely when the division of computation into normal and stitch-up phases is preferable: the stitch-up computations could be incorporated directly into each subsequent plan phase, rather than a separate one. The authors' initial experiments found that the overhead of bookkeeping in this setting often was significant, and hence they separated the cross-phase computation from the main path. However, there may be cases in which the added flexibility (and information gain) is preferable. A second question is whether supplementing the basic CQP technique with a scheme for doing exploration, in the manner of eddies, might be beneficial. Finally, the existing work does not consider how to adapt a fully distributed query plan: the assumption was that query processing is local, although the data is not.

7.2 Eddies with Binary Join Operators

We now return to eddies and see how they can be used to adapt plans involving pipelined binary joins. As discussed in Sections 4.2 and 6.2, an

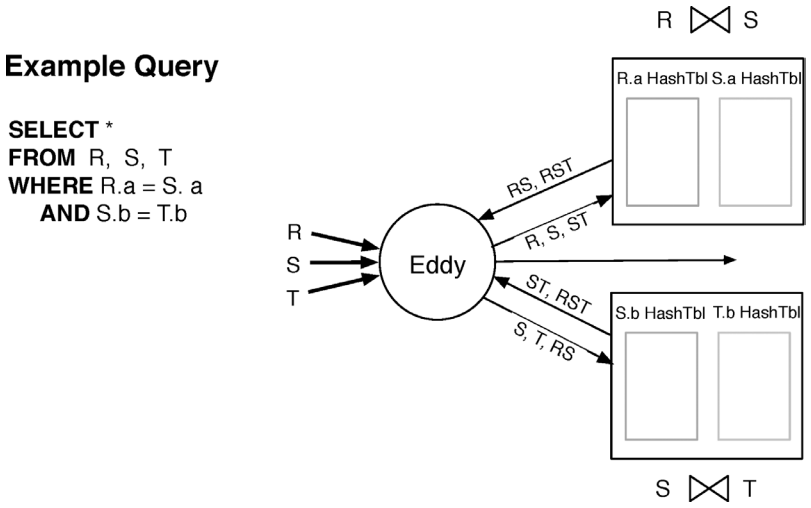

Fig. 7.3 An eddy instantiated for the query $R \bowtie_a S \bowtie_b T$. Valid options for routing are labeled on the edges.

eddy is a tuple router that sits at the center of a dataflow observing the data and operator characteristics, and affects plan changes by changing the way tuples are routed through the operators.

Figure 7.3 shows the eddy instantiated for a three-relation join query $R \bowtie_a S \bowtie_b T$ (we will use this query as the running example for this section). For this query, along with the eddy operator, two doubly pipelined (symmetric) hash join operators are instantiated.

The query is executed by routing the input tuples through these two operators. The *valid* routing options for various types of tuples (shown on the data flow edges) are as follows:

— R tuples can only be routed to the $R \bowtie_a S$ operator, and T tuples can only be routed to the $S \bowtie_b T$.
— S tuples, on the other hand, can be routed to either of the two join operators. In this simple example, this is the only flexibility the eddy has in routing.
— The intermediate RS tuples, if any generated, can only be routed to $S \bowtie_b T$, whereas ST tuples can only be routed to $R \bowtie_a S$.
— Finally, any RST tuples can only be routed to the output.

Determining Validity of Routing Decisions: As we saw above, determining whether a tuple can be routed to an operator is trickier when join operators are involved. Generally speaking, this is done by maintaining some form of *lineage* with each tuple that contains the information about what the tuple consists of, and which operators the tuple has been routed through.

- Avnur *et al.* [6] maintain the lineage in the form of *ready* and *done* bits (cf. Section 3.1.2) that are associated with each tuple. The *done* bits indicate which operators the tuple has already visited, whereas the *ready* bits indicate the valid routing destinations for the tuple. The operators are in charge of setting these bits for a tuple before it is returned to the eddy.
- For efficiency reasons, a latter implementation of eddies in PostgreSQL [37, 40] used a routing table for this purpose. The lineage of the tuple was encoded as an integer that was treated as a bitmap (similar to the *done* bitmap). A routing table, say *r-table*, initialized once at the beginning of query execution, maintained the valid routing destinations for tuples with lineage x at *r-table*$[x]$, thereby allowing the eddy to efficiently find valid routing destinations for any tuple. The size of the routing table, however, is exponential in the number of operators and hence the approach is not suitable for queries with a large number of operators.

7.2.1 Routing Policies

There has been very little work on routing policies for eddies with binary join operators. Since all tuples routed to a join operator are built into one of the hash tables, the join operators accumulate state during execution, making it hard to reason about the operator selectivities. Moreover, this state depends on the routing choices made by the eddy. Figure 7.4 (i) shows the join state at the end of execution for our example query, where S_1 and S_2 denote the sets of tuples that got routed toward the operators $R \bowtie_a S$ and $S \bowtie_b T$, respectively. Since such join state affects the output rates of the operators directly, the

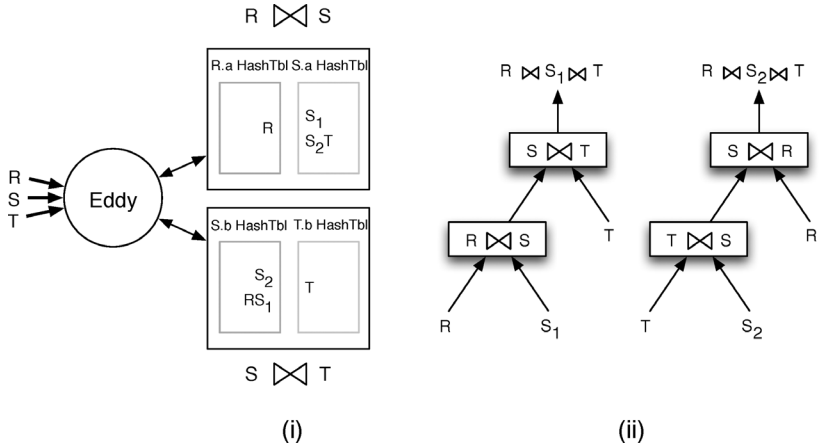

Fig. 7.4 (i) Distribution of tuples at the end of the execution for the query $R \bowtie S \bowtie T$; S_1 and S_2 denote the partitions of S that were routed to $R \bowtie S$ and $S \bowtie T$ operators, respectively. (ii) The two query execution plans executed by the eddy.

routing decisions cannot, in general, be made without taking the state into account. For example, the selectivity of the $R \bowtie_a S$ operator for the R tuples depends on the partitioning of S into S_1 and S_2, and not just on S. This state accumulation (also called *burden of routing history* [40]) makes it a challenge to design and analyze routing policies for eddies.

We will briefly discuss two routing policies for this case (both of which are extensions of policies discussed in Section 4.2.1).

- **Lottery scheduling:** The lottery scheduling policy, as described in Section 4.2.1, is agnostic to the nature of the operators, and can be used unchanged for this case. However because of the join state accumulation issues, this policy may not perform as well as for the selection ordering case [6].
- **Deterministic routing with batching [37, 40]:** This routing policy uses the routing table (discussed above) and batching to reduce the routing overheads.
  - **Statistics maintained:** The eddy explicitly monitors (a) the selectivities of the predicates, (b) the sizes of the relations, and (c) the domain sizes of the

join attributes. The latter two, under assumption of independence and uniformity, can be used to determine the join selectivities.

— **Routing policy:** The routing decisions are made for batches of tuples at a time to reduce the overhead. A *reoptimizer* is invoked every K tuples (where K is called *batching factor*). The reoptimizer uses the join selectivities to choose the best operator to route to for each possible tuple *signature*, and encodes these decisions in the routing table (by simply moving the corresponding operator to the front of the list). The next K tuples are all routed according to this plan. Experimental results in [40] suggest a batching factor of the order of 1000 resulted in acceptable routing overhead, while allowing the eddy enough flexibility in routing.

Designing effective routing policies for the case of eddies with binary join operators remains an important open problem in this area. Next we attempt to provide some insights into this problem by analyzing eddies, and discussing the issue of state accumulation in more detail. We then briefly describe an operator called STAIR that was proposed to handle some of the state accumulation issues.

7.2.2 Post-mortem Analysis

As with MJoins, eddies with SteMs, and CQP, execution using an eddy and binary join operators can also be captured using the notion of horizontal partitioning, with one critical difference: the partitioning is determined solely by the routes chosen for the tuples, and does not depend on the order of arrival. For instance, if all tuples of each relation are routed identically, the relations will not be partitioned and the eddy will execute a single traditional query plan over the entire input; on the other hand, using an MJoin or SteMs will typically result in different query plans being executed for different tuples based on the order of arrival (cf. Section 6.2.4).

We will illustrate how to do the post-mortem analysis through two examples.

Example 7.1. Figure 7.4 shows an example execution of our running example query $(R \bowtie_a S \bowtie_b T)$. In this case, S tuples are the only ones for which the eddy has a choice. Let S_1 denote the tuples that were routed to $R \bowtie_a S$ operator, and let S_2 denote the tuples routed to $S \bowtie_b T$, operator. From the data distribution in the hash tables, we can infer the plans that were used to execute the two subqueries (Figure 7.4 (ii)):

— $R \bowtie S_1 \bowtie T$: was executed as $(R \bowtie S_1) \bowtie T$ (We use this notation to mean that the R tuples were first joined with S_1 tuples and the resulting RS_1 tuples were then joined with T tuples).
— $R \bowtie S_2 \bowtie T$: was executed as $R \bowtie (S_2 \bowtie T)$.

Example 7.2. Next we will consider a more complex example that reveals certain other interesting phenomena. Consider the query $R \bowtie_a S \bowtie_b T \bowtie_c U$ (Figure 7.5). Let S_1 and S_2 denote the S tuples routed to $R \bowtie S$ and $S \bowtie T$, respectively. Similarly, let T_1 and T_2 be the T tuples routed to $S \bowtie T$ and $T \bowtie U$ operators. Also let the ST tuples generated during the execution (if any) be routed to the $R \bowtie S$ operator first.

Figure 7.5 shows the distribution of the tuples in the join operators after execution of the query. As we can see from this figure, the four subqueries and the corresponding execution plans in this case are:

— $R \bowtie S_1 \bowtie T_1 \bowtie U$: Executed using $((R \bowtie S_1) \bowtie T_1) \bowtie U$.
— $R \bowtie S_1 \bowtie T_2 \bowtie U$: Executed using $(R \bowtie S_1) \bowtie (T_2 \bowtie U)$.
— $R \bowtie S_2 \bowtie T_1 \bowtie U$: Executed using $(R \bowtie (S_2 \bowtie T_1)) \bowtie U$.
— $R \bowtie S_2 \bowtie T_2 \bowtie U$: Executed using $R \bowtie (S_2 \bowtie (T_2 \bowtie U))$.

Note that if the ST tuples were themselves split among the two possible destinations ($R \bowtie S$ and $T \bowtie U$), S and T must be further

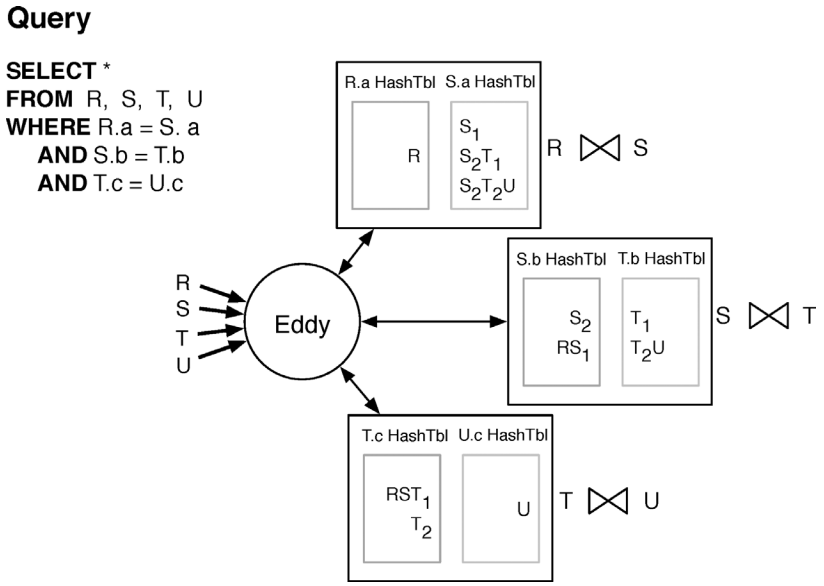

Fig. 7.5 An example distribution of tuples for query $R \bowtie_a S \bowtie_b T \bowtie_c U$ at the end of query execution.

partitioned to capture that execution. In the worst case, each S tuple and each T tuple may form a partition by itself.

As we can see, the routing destinations for tuples alone determine the plans executed by the eddy, and the order of arrival is not directly relevant. However, the routing destinations themselves would typically depend on the order of arrival of tuples.

Note that the eddy cannot simulate an arbitrary partitioning of the relations. For example, for the query $R \bowtie_a S \bowtie_b T$ (Figure 7.3), consider a partitioning where R is split into R_1 and R_2 and the two partitions are processed using plans $(R_1 \bowtie_a S) \bowtie_b T$ and $R_2 \bowtie_a (S \bowtie_b T)$, respectively. This cannot be simulated by an eddy. If S is routed toward the $R \bowtie_a S$ operator, it must join with all of R; there is no way for S to join with just a subset of the R tuples. The STAIR operator that we discuss in the next section can be used to achieve such plans.

Post-mortem: Execution of a query using an eddy and binary join operators can be viewed as a horizontal partitioning of the data (with par-

titions consisting of tuples that were routed identically). The use of binary joins establishes a number of constraints on the ways these partitions can be created and combined.

7.2.3 Burden of Routing History

The most attractive feature of the eddies framework is that the routing choices can be made on a per-tuple basis. However the above analysis shows that an eddy may only have a limited ability to choose the plans that any given tuple may participate in. Furthermore, given the complex interactions between the routing choices made for different tuples, it may not be possible to predict how the decision made by the eddy will affect the execution of subsequent tuples. This is a direct artifact of the state that gets accumulated inside the join operators, which we call the *burden of routing history*.

Perhaps the most serious problem with this state accumulation is that the adaptation opportunities an eddy may have in future may be significantly constrained because of its routing history. To illustrate this point, we review an example from the original eddies paper [6]:

Example 7.3. Consider the query $R \bowtie_a S \bowtie_b T$, using two doubly pipelined hash join operators (Figure 7.3). At the beginning of query processing, the data source for R is stalled, and no R tuples arrive. Hence the $R \bowtie_a S$ operator never produces a match, which makes it an attractive destination for routing S tuples: it efficiently eliminates tuples from processing, reducing work for the query engine. The result is that the eddy emulates a static query plan of the form $(R \bowtie S) \bowtie T$. Some time later, R tuples arrive in great quantity and it becomes apparent that the best plan would have been $(S \bowtie T) \bowtie R$. The eddy can switch the routing policy so that subsequent S tuples are routed to $S \bowtie_b T$ first. Unfortunately, this change is "too little too late": all the previously seen S tuples are still stored in the internal state of the $R \bowtie_a S$ operator. As R tuples arrive, they *must* join with these S tuples before the S tuples are joined with T tuples. As a result, the eddy effectively continues to emulate the suboptimal plan $(R \bowtie S) \bowtie T$, even after its routing decision for S has changed.

The burden of routing history severely impacts an eddy's ability to adapt in several other ways as well [40].

- **Cyclic queries — inability to adapt spanning trees:** Cyclic queries, where the join graphs have cycles, are quite common in many environments. For example, in a streaming environment, many relations may be joined on the same attribute resulting in cyclic query graphs [120]. When executing a query using binary join operators, the eddy must choose a spanning tree of the join graph a priori and then use it throughout the execution.
- **Handling sliding window queries:** Executing sliding window queries in data streams requires the ability to *delete* tuples from the execution state. Because of the intermediate tuples that get stored inside the join operators, this is harder to do in this case than the alternative approaches.
- **Restricted Pre-computation in Presence of Delays:** If the data from a remote data source is delayed, it might be attractive to perform other useful work while waiting for that data to arrive. For instance, *partial results* might be of interest in interactive environments [96]. Another option is to aggressively join all the data that has already arrived, even if that requires use of a suboptimal plans (query scrambling [2, 118]). The eddy may be unable to do this because the state is captured inside the join operators. In fact, the inability to produce partial results aggressively was one of the main reasons the SteM operator was proposed (Section 6.2.3).

7.3 Eddies with STAIRs

The STAIR operator [40] removes the limitations discussed in the previous section by exposing the state stored inside the operators to the eddy and by providing the eddy with primitives to manipulate this state. In this section, we will briefly discuss this operator and how it can be used to lift the burden of history through state manipulation.

7.3.1 STAIR Operator

A STAIR operator encapsulates the state typically stored inside the join operators. For simplicity, we will assume that an intermediate tuple, t, generated during query processing is stored as a list of pointers to the source tuples that were joined together to generate it. We will use $schema(t)$ to denote the set of relations whose tuples t contains. Formally, a STAIR on relation R and attribute a, denoted by $\mathcal{R}.a$, contains either tuples from relation R or intermediate tuples that contain a tuple from R ($R \in schema(t)$). $\mathcal{R}.a$ supports the following two basic operations:

(i) insert($\mathcal{R}.a$, t), $R \in schema(t)$: Given a tuple **t** that contains a tuple from relation R, store the tuple inside the STAIR.

(ii) probe($\mathcal{R}.a$, *val*): Given a value ***val*** from the domain of the attribute $R.a$, return all tuples r stored inside $\mathcal{R}.a$ such that $r.a =$ ***val***.

Figure 7.6(i) shows the STAIRs that would be instantiated for executing our example 3-relation query. In essence, each join operator is replaced with two STAIRs that interact with the eddy directly. These two STAIRs are called **duals** of each other. Note that even if both joins were on the same attribute, we would have two STAIRs on relation S and attribute a ($= b$). These two STAIRs are treated as separate operators since they participate in different joins.

The query execution using STAIRs is similar to query execution using join operators. Instead of routing a tuple to a join operator, the eddy itself performs an *insert* on one STAIR, and a *probe* into its dual. In fact, the following property is always obeyed during query execution:

Definition 7.1. ***Dual Routing Property:*** *Whenever a tuple is routed to a STAIR for probing into it, it must be simultaneously inserted into the dual STAIR.*

This property is obeyed implicitly by the symmetric hash join and MJoin operators as well, and can be relaxed by associating timestamps with the tuples in a fashion similar as described in [95].

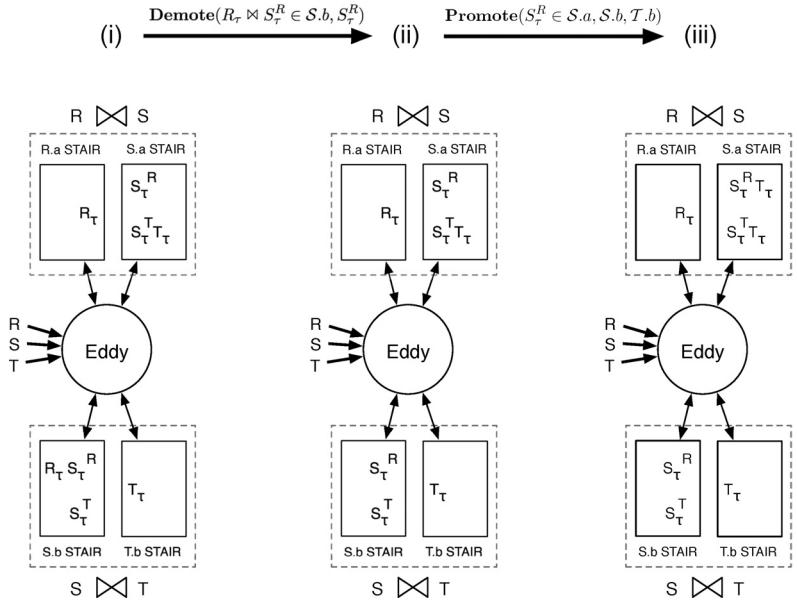

Fig. 7.6 (i) Query execution state at time τ when using an eddy and STAIRs — R_τ denotes the tuples of relation R that have been processed by time τ; (ii) The execution state after Demote($R_\tau \bowtie S^R_\tau \in \mathcal{S}.b, S^R_\tau$); (iii) The execution state after Promote($S^R_\tau \in \mathcal{S}.a, \mathcal{S}.b, \mathcal{T}.b$).

7.3.2 State Management Primitives

Other than the two basic operations described above, STAIRs also support two state management primitives that allow the eddy to manipulate the join state inside the STAIRs.

(i) Demote($t \in \mathcal{R}.a$, $t' = \pi_Y(t)$), $R \in Y \subset schema(t)$: Intuitively, the *demotion* operation involves reducing an intermediate tuple stored in a STAIR to a *sub*-tuple of that tuple, by removing some of the source tuples it contains. This operation can be thought of as *undoing* some work that was done earlier during execution. Given that the preconditions are satisfied, the operation simply replaces t by t' in $\mathcal{R}.a$.

(ii) Promote($t \in \mathcal{R}.a$, $\mathcal{S}.b$, $\mathcal{T}.b$), $S \in schema(t), T \notin schema(t)$: The *promotion* operation replaces a tuple t in a STAIR $\mathcal{R}.a$, with *super*-tuples of that tuple that are generated using another join in the query, in this case, $S \bowtie_b T$. The operation removes t from $\mathcal{R}.a$ and replaces it with the set of tuples $\{t\} \bowtie_b \mathcal{T}.b$, which are found by probing into

$\mathcal{T}.b$ using t (the pre-conditions ensure that this is a valid operation). During the process t is also inserted into $\mathcal{S}.b$. Note that the join result may be empty, in which case no tuples are inserted back into $\mathcal{R}.a$.

To amortize the costs of these operations, STAIRs also support extensions where the operations are performed on a batch of tuples at a time, instead of a single tuple at a time. Figure 7.6(ii) shows an example of a Demote operation where $R_\tau S_\tau^R$ in STAIR $\mathcal{S}.b$ is demoted to S_τ^R. Similarly, Figure 7.6(iii) shows an example of a Promote operation where S_τ^R ($\in \mathcal{S}.a$) is promoted using $S \bowtie_b T$; as a result S_τ^R is replaced by $S_\tau^R T_\tau$ in $\mathcal{S}.a$.

Theorem 7.1. [40] An eddy with STAIRs always produces correct query results in spite of arbitrary applications of the promotion and demotion operations.

Both these state management operations, as described above, can result in a state configuration that allows spurious duplicate results to be generated in future. Such duplicates may be acceptable in some scenarios, but can also optionally be removed. We refer the reader to [40] for details on how to prevent the duplicates from being generated.

7.3.3 Lifting the Burden of History using STAIRs

The STAIR state management primitives provide the eddy with the ability to manipulate state, and in effect, reverse any bad decisions that it might have made in past. We will illustrate this through an example.

Figure 7.6(i) shows the state maintained inside the join operators at time τ for our example query. Let us say that, at this time, we have better knowledge of the future and we know that routing S_τ^R toward $R \bowtie S$ was a mistake, and will lead to sub-optimal query execution in future (because $R \bowtie S_\tau^R$ has high selectivity). This prior routing decision can be reversed as follows (Figure 7.6):

— Demote the $S_\tau^R \bowtie R_\tau$ tuples in $\mathcal{S}.b$ to S_τ^R.
— Promote the S_τ^R tuples from $\mathcal{S}.a$ to $S_\tau^R \bowtie T_\tau$ using the STAIRs $\mathcal{S}.b$ and $\mathcal{T}.b$.

Figure 7.6 (iii) shows the result of these operations. As we can see, the state now reflects what it would have been if S_τ^R had previously been routed to the $\mathcal{S}.b$ and $\mathcal{T}.b$ STAIRs, instead of $\mathcal{R}.a$ and $\mathcal{S}.a$ STAIRs. As a result, future R tuples will not be forced to join with S_τ^R.

The process of moving state from one STAIR to another is referred to as *state migration*.

7.3.4 State Migration Policies

While providing the eddy with more flexibility, STAIRs introduce yet another set of policy issues for eddies, namely, when to perform state migration. Deshpande and Hellerstein [40] propose a greedy policy that "follows" the routing policy by keeping the state inside the operators consistent with the "current" plan being used by the eddy. In other words, when the eddy changes the execution plan being used, the internal join state is migrated to make it appear as if the eddy had been using the new plan from the beginning of execution. To avoid thrashing and also unnecessary migrations at the end of the query, the state is migrated in stages over a period of time instead of all at once.

7.3.5 Post-mortem Analysis

Execution using STAIRs can be analyzed using horizontal partitioning in a fashion similar to using only binary join operators as we saw in the previous section. We will illustrate this through an example. Consider the example shown above where at time τ: (1) the routing policy was changed so that the eddy starts routing all new S tuples to $S \bowtie T$, and (2) a state migration was performed as shown in Figure 7.6. Further, let this be the only adaptation that was done. Denoting by R_∞ all tuples of R that arrived by the end of the execution, the "plans" executed by the eddy can be written down as:

- $(R_\tau \bowtie S_\tau^R) \bowtie T_\tau$
- $R_\tau \bowtie (S_\tau^T \bowtie T_\tau)$
- $S_\tau^R \bowtie T_\tau$
- $R_\infty \bowtie (S_\infty \bowtie T_\infty) - R_\tau \bowtie S_\tau \bowtie T_\tau$

The first two rows are the plans executed till time τ, and the third row shows the work done during the state migration step; the last row specifies the work done after time τ (we abuse the notation somewhat to indicate that the results already produced by time τ are not regenerated).

Post-mortem: As with an eddy with binary joins, an eddy with STAIRs can be viewed as working on horizontal partitions of the data, where the data items are partitioned based on how they are routed. A state migration (using STAIR primitives) can be seen as a redundant join of some of the partitions using an alternate join order.

7.4 Dynamic Plan Migration in CAPE

The CAPE stream system [124], which permits plan changes during execution, also supports mechanisms for migrating state to make it consistent with the new query plan to be used. The CAPE query processor is not routing-based, and instead executes queries using a tree of binary join operators. Because of this, it requires that the execution state be consistent with the query plan being used in a much stricter sense. To be more precise, when executing a query using an eddy and STAIRs (or symmetric hash join operators), the hash tables may contain tuples of different types (cf. STAIR $\mathcal{S}.a$ in Figure 7.6(i)). This is not permitted in the CAPE system, and hence when the query plan is changed mid-execution, state migration *must* be done to ensure correctness.

Zhu *et al.* [124] propose and evaluate several different policies for state migration in CAPE. The *moving state strategy* is somewhat similar to the policy described above: it pauses the execution, migrates state from the old operators to the new operators, and then resumes execution using the new query plan. The *parallel tracks strategy* is specific to queries over data streams; it runs both the new and the old plans simultaneously for a while, and throws away the old plan when the tuples inside the operators of the old plan are not required for executing the rest of the query (e.g., when they fall out of the sliding windows). Note that the state migration primitives supported by STAIRs can be used

to simulate either of these strategies, and hence these policies could also be used when executing a query using STAIRs.

Post-mortem: As with STAIRS, CAPE can be viewed as executing different query plans over horizontal partitions of data (with partitions based on order of arrival). The moving state strategy is similar to STAIRs — it may combine some partitions twice using different plans; the parallel tracks strategy may process some new input tuples using two plans in parallel, but does not do redundant query processing.

7.5 Summary

In this section, we discussed a variety of schemes for adaptive query processing that use trees of binary join operators for query execution. Table 7.1 recaps some of these techniques from the perspective of the adaptivity loop. The main challenge with using binary joins for execution is dealing with and reasoning about the state that gets

Table 7.1 Comparing some of the techniques discussed in this section using the adaptivity loop.

CQP [73]
Measurement: Operator cardinalities (hence join selectivities); sort orders; incremental histograms.
Analysis and planning: Periodic or trigger-based re-planning using an integrated query reoptimizer.
Actuation: By replacing the query plan operators other than the root and leaves. Requires a stitch-up plan at the end.
Eddies with binary join operators/STAIRs [40]
Measurement: Join selectivities monitored explicitly.
Analysis and planning: Periodic re-planning using a query optimizer.
Actuation: By changing the routing tables used by the eddy. State migration (if required) using the STAIR primitives.
CAPE [124]
Measurement: System parameters (selectivities, stream rates etc).
Analysis and planning: Reoptimization when the parameters change.
Actuation: Explicit migration of state from old to new operators.

accumulated inside the join data structures. CQP, which is a general architecture for adapting full query plans, reuses state across query plans in an opportunistic manner but does not explicitly "migrate" state across dissimilar query plans. CAPE performs explicit state migration when switching plans, whereas the STAIR operator, proposed for use with an eddy, provides state management primitives that permit the eddy to handle the state more flexibly.

The complexity of reasoning about the join state raises serious concerns about using binary join operators for query evaluation. The MJoin and SteM operators that we saw in the previous section do not suffer from these issues. However, in spite of these problems, the approaches discussed in this section typically outperform the history-independent approaches, except possibly in streaming environments. This is because the history-independent approaches throw away the intermediate results computed during execution and may have to recompute those repeatedly. Experimental results in the PostgreSQL implementation of eddies confirm this behavior [40]. Latter work in the StreaMON project also considered *adaptive caching* of intermediate tuples for the same reason [11] (Section 6.3). In streaming environments, this cost may be offset partly by the significantly lower cost of tuple deletions, and also by the lower reuse of intermediate tuples (because of sliding windows).

Two systems use hybrid schemes that combine these approaches. Viglas *et al.* [120] present techniques for choosing plans that simultaneously use binary join and MJoin operators. Second, the adaptive caching work (Section 6.3) explicitly caches intermediate results to solve this problem. In our opinion, exploring the tradeoffs between these two approaches remains one of the most important open research challenges in adaptive query processing.

8

Adaptive Join Processing: Non-pipelined Execution

We have seen several adaptation techniques for pipelined plans that operate by changing the order in which tuples are pipelined (routed) through plan operators. We now turn to plans with non-pipelined (blocking) operators, the dominant style of plan considered by most DBMSs today. The techniques we describe adapt by changing the query plan at the materialization points caused by such blocking operators. They treat the plan as being partitioned into multiple pieces, and optimize and execute each piece separately.

We discuss three techniques. Plan staging (Section 8.1) is a method widely used by DBMS applications to get more predictable query performance. Mid-query reoptimization (Section 8.2) is an adaptation method that has become quite popular in recent years and been used in several query processing systems, including systems not originally designed with a focus on adaptation. These two methods form the principal response of the commercial DBMS community to the problem of poorly chosen query plans. The third technique, query scrambling (Section 8.3), is an adaptation method used in query processing over wide-area data sources to deal with bursty data arrival rates. We conclude with a discussion of post-mortem analysis and adaptivity loop for these techniques (Section 8.4).

8.1 Plan Staging

Materialization points are points in a plan where an intermediate result relation is created in entirety before proceeding with further operators of the plan. Materialization is common in query plans, and arises in several places, such as the sort operator (used for sort-merge join or group-by, or for clustering the outer of lookups into a clustered index), the build side of the hash join operator, and explicit materialization operators (used, for example, to cache results of common subqueries).

A key property of materialization points is that they split the plan tree into independent subgraphs, whose outputs are expressible as relational-algebraic expressions of their inputs. Figure 8.1 illustrates a plan with several materialization points for an example query.

Executing such a plan is equivalent to submitting separate SQL queries one after another:

P1 $\leftarrow$ sort $(Sales \bowtie \sigma_{comment\ not\ like\ \text{“}\%delayed\%\text{”}}(Shipping))$

P2 $\leftarrow$ sort(Store)

P3 $\leftarrow$ sort(Region)

P4 $\leftarrow P2 \bowtie P3$

P5 $\leftarrow P1 \bowtie Parts \bowtie P4$

We call this explicit use of separate queries *plan staging*. This staged execution provides a simple measure of resiliency to poor optimizer estimates: by materializing the intermediate results as separate

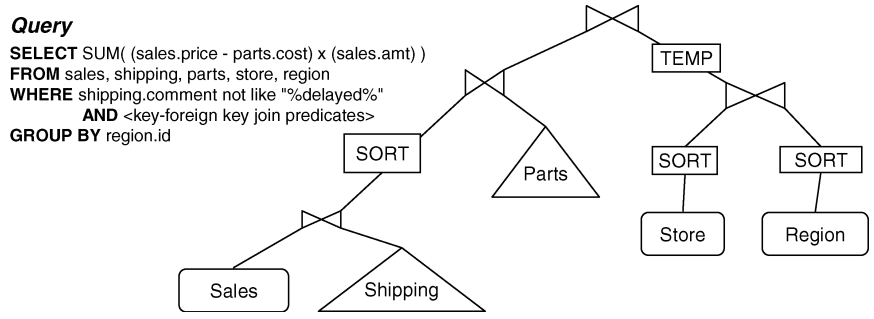

Fig. 8.1 A plan with several materialization points.

tables, the optimizer has an opportunity to compute their true cardinalities and use these to choose the plans for subsequent stages. This method is used by many application programs.[1] It results in an interleaving of optimization and execution: first optimize and run one stage to completion, and then, using its result as input, optimize and run the next stage, and so on. Since the optimization and execution steps are interleaved, the optimization of each stage can use statistics (cardinality, histograms, etc.) computed on the outputs of the previous stages. In some cases, the query optimizer will in fact add special information gathering operators in order to have more detailed statistics.

It is instructive to compare this staged execution with parametric optimization and choose-plan operators of Section 2.2.2. In parametric optimization, the optimizer picks a number of candidate plans, and the query processor waits until it reaches a choose-plan operator to decide which plan to run. By this time, values of run-time parameters and possibly some intermediate result sizes are known, so the choose-plan operator can make a more informed choice than the query optimizer could have. In contrast, plan staging does not require candidate plans to be chosen up-front; instead the optimizer is re-invoked at each stage. Recent work on switchable plans bridges the gap between these two styles (Section 8.2.5).

8.2 Mid-Query Reoptimization

In 1998, Kabra and DeWitt [75] introduced a generalization of such staged execution called *mid-query reoptimization*, where the application program does not have to get involved. This work has since been adopted and generalized in several other systems, in the forms of progressive optimization [87] and proactive reoptimization [9]. Mid-query reoptimization is best understood in terms of the following loop:

- As part of optimization, instrument the query plan with *checkpoints*;

[1] This does put the application program in the business of guiding query plans, which goes against the relational philosophy.

- During execution, if the actual cardinality[2] of tuples flowing through a checkpoint is very different from estimated: *reoptimize* the remainder of the query to switch to a new plan
- Repeat

A *checkpoint* is a unary plan operator that monitors statistics (typically cardinality) of tuples flowing through it. If the monitored value is close to the optimizer's estimate, the checkpoint acts like a no-op. If not, the checkpoint terminates the current query execution and triggers another round of optimization and execution, commonly called *reoptimization*.

Figure 8.2 illustrates the mid-query reoptimization analogue of the staged plan of Figure 8.1. The initial query plan was to join with Parts using an (index) nested-loop join (NLJN). This choice depends heavily on the cardinality estimate for the sort result. In this query, this depends on the not-like predicate, for which it is notoriously hard to estimate cardinalities. If the cardinality is small, NLJN is good, but it rapidly loses to hash or merge join as the cardinality increases. To guard against this risk, the plan has a checkpoint just before the join with Parts, which checks the output cardinality of the SORT. If the

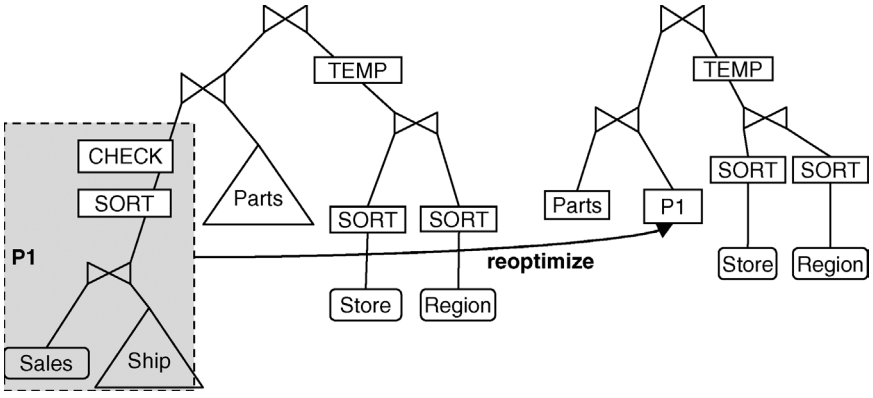

Fig. 8.2 Reoptimizing a plan at a lazy checkpoint.

[2] Any other statistic over these tuples, such as a frequency distribution, can be substituted for cardinality.

checkpoint succeeds, the query continues with the NLJN; if it is violated, reoptimization is triggered and causes a new plan with a hash join to be picked (with inner and outer reversed).

There are three main challenges in doing such a scheme:

— where should we place checkpoints
— what is the best way to switch to a new plan
— how far off should the actual cardinality be from the estimated cardinality for reoptimization to be triggered

We discuss these in turn.

8.2.1 Flavors of Checkpoints

Markl *et al.* [87] study different kinds of checkpoint operators that vary in where they are placed in the query plan. The simplest kind of checkpoints, like those of Figure 8.2, are positioned above materialization points. They are called *lazy* checkpoints. In the usual iterator formulation, the open() of a lazy checkpoint first calls open() on its input, thereby fully evaluating it and materializing the results. Then it checks the input cardinalities. Unless reoptimization is needed, the next() of a lazy checkpoint simply calls next() on its input. Reoptimization can be done with no wasted work in most cases, because it is just like plan staging. (Work is, of course, wasted if the optimizer chooses not to reuse the evaluated inputs, as we explain later.)

The drawback with lazy checkpoints is that they occur too infrequently: some plans may be mostly or fully pipelined and even when checkpoints exist, they may occur too late in query execution. Turning to the example of Figure 8.2 again, if the optimizer has underestimated $|P1|$ significantly, the query processor could waste huge amounts of CPU and memory in scanning and sorting P1 before the checkpoint is even reached.

Eager checkpoints are an alternative that avoid some of these drawbacks. In these checkpoints, the cardinalities are monitored by a counter during the calls to next(), thus the checking is done in a pipelined fashion. So they can be placed anywhere in the query plan, as in the lower left of Figure 8.3. The advantage of the eager checkpoint in this

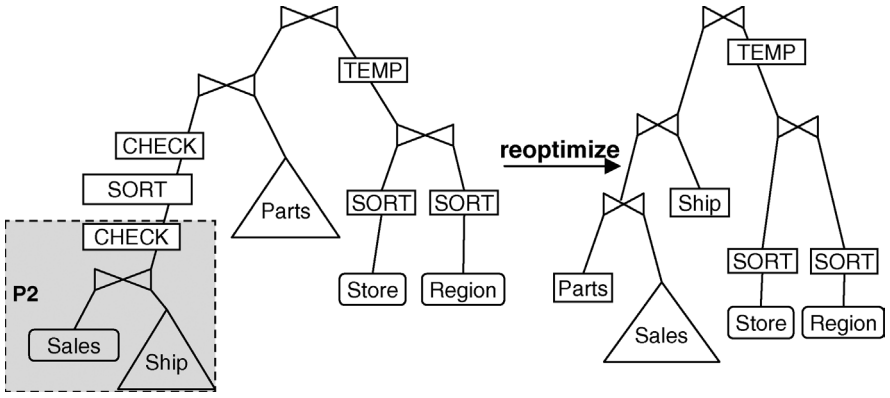

Fig. 8.3 Reoptimizing a plan at an eager checkpoint.

example is that it is triggered even while P2 is being computed and sorted. When the checkpoint is triggered, the query stops the expensive scan and sort of Sales, and instead switches to a plan that uses an index on Sales. The cardinality of tuples flowing through E increases over time, and it must extrapolate to estimate the final "actual" cardinality.

Eager checkpoints always result in wasted work upon reoptimization because the tuples that were processed through the checkpoint until reoptimization do not constitute a well-defined relational expression, and have to be thrown away. In Figure 8.3, the reoptimized execution cannot express precisely which subset of P1 has been sorted, and so cannot use the subset. This waste of any intermediate result that is not fully computed is an essential characteristic of mid-query reoptimization, and distinguishes it from the horizontal partitioning discussed in the previous two sections. Nevertheless, eager checkpoints are useful as "insurance" in places where optimizer estimates have very high degree of uncertainty, e.g., after several possibly correlated predicates.

There are two other proposals that complement checkpoints.

Forced Materialization: Markl *et al.* [87] introduce *extra materialization points* into query plans in order to enable lazy checkpoints. These extra materializations are useful in locations like the outer relation in a nested-loops join, where the optimizer estimates the

cardinality to be very low; if the optimizer proves to be right, the extra materialization was cheap, while if it is wrong, the extra materialization was pricey but paid for itself in avoiding the bad plan (nested-loops join instead of hash join). For example, if there had not been a sort below P1 in the original plan in Figure 8.3, it might be a good idea to introduce one anyway, guarding it with eager and lazy checkpoints, because "like" predicates are hard to estimate.

Operator Rescheduling: Many federated query processing systems [44, 114] reschedule the order of evaluation of plan operators so that subplans over remote data sources are fully evaluated first, before reoptimizing and running the portion of plan that runs at the federator node. This is because cardinalities of operations over federated data sources are especially hard to estimate, and the optimizer's initial estimates are often wild guesses. Such rescheduling is also applicable in non-federated plans. For example, if there was uncertainty in the estimates of $|Store|$ and $|Region|$, the query processor could evaluate the sorts on them first before proceeding with the rest of the plan.

8.2.2 Switching to a New Plan

When a checkpoint is triggered, it is an indication to the query processor to switch to a new query plan by re-invoking the optimizer. There are many tricky issues in actuating this switch efficiently:

Reusing Intermediate Results: To avoid wasted work, the optimizer must try to reuse intermediate results computed during the query execution up to the checkpoint. Returning to the lazy checkpoint of Figure 8.2, suppose that the top-most join was a hash join and that the build side was Store-Region. When the checkpoint is triggered, there are four intermediate results available, on Sales-Shipping, Store, Region, and Store-Region, with the last three overlapping each other. So the simple staged execution model of Section 8.1 does not apply. A more general solution is to cast these intermediate results as materialized views and let the optimizer decide how best to use (or ignore) them during reoptimization [87].

Getting a Better Plan: The purpose of checkpoints is to switch to a better plan after considering the difference between optimizer estimated cardinalities and the actual cardinalities. But reoptimization almost always presents the optimizer with cardinality estimates over *overlapping* relational expressions. For example, an optimizer might have cardinality estimates over three base-tables $|\sigma_a(R)|$, $|\sigma_b(S)|$, and $|\sigma_c(T)|$ from its catalog. Reoptimization might produce actual values for $|\sigma_{a\wedge b}(R \bowtie S)|$ and $|\sigma_{a\wedge b\wedge c}(R \bowtie S \bowtie T)|$, which are different from what is predicted by independence assumptions. Now, if the optimizer estimates $|\sigma_{b\wedge c}(S \bowtie T)|$ using the base table selectivities and plugs in $|\sigma_{a\wedge b}(R \bowtie S)|$ from the actual value, these two estimates are inconsistent. Most current query optimizers use such inconsistent cardinality estimation, and this can even lead to plan regression upon reoptimization (i.e., reoptimization results in a slower plan), as pointed out in [44, 81]. Instead, the optimizer needs to use the maximum entropy principle to consistently estimate cardinalities based on all available information [81].

Side-effects: Query execution can at times have side-effects. Pipelined results can be returned to the user. Query results can also be used to perform updates (for example, in queries used to maintain materialized views). It is important when reoptimizing to ensure that side-effects are applied exactly once. Side-effects from pipelined results are one of the major difficulties in doing reoptimization in pipelined plans. Markl *et al.* [87] suggest doing an anti-join to eliminate false duplicates, but this can be prohibitively expensive. If a query plan contains update operations, only lazy checkpoints can be placed above it, and upon reoptimization the optimizer *must* reuse the results up to these checkpoints as-is, to avoid applying the update twice.

8.2.3 Threshold to Invoke Reoptimization

Reoptimization always has a price. At a minimum, there is the cost of re-running the optimizer and of restarting the query execution. Often, intermediate results are not reused perfectly after reoptimization. So it is important to reoptimize only if we are reasonably sure that a much faster plan will result. This relates to the difference between estimated

and actual cardinalities at the checkpoint — if the difference is large, it is more likely that current plan is substantially worse than another, and reoptimization has a high payoff. But how should the threshold be set. One way to make a precise decision of when to reoptimize is the notion of validity ranges:

Definition 8.1. The *validity range* of an edge in a query plan is a range of cardinalities for tuples flowing along that edge within which the current query plan is optimal, or near-optimal.

Validity ranges can be used in many ways:

- To decide when to reoptimize: if the cardinality of tuples flowing through a checkpoint falls outside the validity range of its input edge, the checkpoint triggers reoptimization.
- To guide where to place checkpoints: if the validity range on an edge is very small (or has high risk of not being satisfied, in an optimizer that considers probabilities such as [7, 9]), that range is likely to be violated, and an eager check point or a lazy checkpoint via forced materialization may be called for.
- To identify risky plans: the presence of many narrow validity ranges suggest that the chosen plan has high risk and might be worth avoiding.

In [9], validity ranges are generalized to be the range of acceptable cardinalities for not just a single plan, but a set of plans among which the optimizer can easily switch at runtime without wasting work. They call such ranges *bounding boxes*.

8.2.4 Computation of Validity Ranges

Validity ranges are computed during query optimization, when the initial plan is chosen. But they are not a characteristic of this initial plan, but rather of the entire space of possible plans. For example, the validity range on a scan over a base table depends not just on the operator immediately above that scan, but on all the possible plans for the

query — a change in selectivity on one edge can have repercussions on the join order several operators above it. So computing them exactly might lead to an exponential blow-up in the optimization cost.

Instead Markl *et al.* [87] describe a method to compute approximate validity ranges such that the chosen plan is optimal with respect to other plans that share the same join tree.

By default, the optimizer sets all validity ranges to be $(-\infty, \infty)$. Later, say that the optimizer is comparing the two plans in Figure 8.4, during its dynamic programming phase. The plans differ only in the join operator used, and the cost of both operators is a function of the same input cardinalities: $\text{cost}(P1) = f_1(|P|, |Q|)$ and $\text{cost}(P2) = f_2(|P|, |Q|)$. At the current estimates p_e, q_e for $|P|$, $|Q|$, say that P1 is cheaper and the optimizer is pruning away P2. Then the optimizer computes a bounding rectangle $(p_{\min}, p_{\max})$ and $(q_{\min}, q_{\max})$, within which $\text{cost}(P1) - \text{cost}(P2) < 0$.[3] Accordingly, the optimizer updates the validity ranges on the inner and outer edges: $VR(p) \leftarrow VR(p) \cap (p_{\min}, p_{\max})$ and $VR(q) \leftarrow VR(q) \cap (q_{\min}, q_{\max})$. This intersection process may be repeated multiple times if multiple operators are compared to operator P1 — the final validity range is the one in which P1 beats all of these operators.

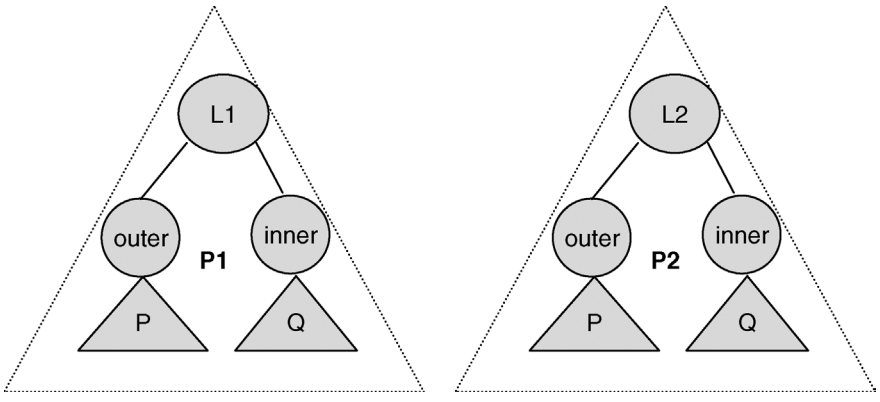

Fig. 8.4 Computation of validity ranges.

[3] Generalizing this for near-optimal plans is straightforward: the optimizer computes the rectangle corresponding to $\text{cost}(P1) - \alpha\text{cost}(P2) < \beta$ for suitable constants α and β.

This bounding rectangle is computed either through simple linear search or through standard numerical root finding techniques.

The advantage of this method is that it only adds a constant factor overhead to the optimization process: each time the optimizer prunes one plan in favor of another, it needs to calculate the cost difference at a few points around estimated cardinality, in order to compute the bounding rectangle.

But the disadvantage is that it only computes validity ranges with respect to plans that share the same join tree as the finally chosen plan (changes of join operator, and inner-outer reversals for a single join operator, are supported). As a result, validity ranges are conservative in the following sense. Even within the validity range, a plan may become suboptimal with respect to alternative plans that use different join orders. That is, validity ranges err on the side of fewer reoptimizations.

8.2.5 Bounding Boxes and Switchable Plans

A recent paper proposes an alternative to validity ranges called *bounding boxes* [9]. The idea is to proactively expect and plan for reoptimization during the initial optimization itself. Rather than choosing a single initial plan with validity ranges on its edges, they modify the optimizer to choose, for each uncertain cardinality estimate, a bounding box which is likely to straddle the actual cardinality. Corresponding to each bounding box, they choose a set of "switchable" plans up front — ones among which it is easy to switch without wasting work, such as P1 and P2 in Figure 8.4.

One challenge with this scheme is the computation of bounding boxes. To ensure that the bounding box will straddle actual cardinality, the optimizer needs good estimates of the probability distribution of cardinalities. Some papers have suggested that histogram quantile widths can be used to infer the error possibility, but this only applies to cardinalities of base tables. Cardinality mis-estimation arises more frequently and severely from missing statistics, correlations, like predicates, and so on, and there has been little work on studying the distribution of errors in these situations. The work of Babu *et al.* [9] seeks to limit the impact of cardinality mis-estimates by focusing on lin-

ear query plans, where at least one input to every join has a known cardinality.

8.3 Query Scrambling

An alternative form of plan staging has been developed in the context of query processing over wide-area data sources. Besides cardinalities, a major runtime variable in such queries is delays and burstiness in data arrival. For example, the plan of Figure 8.5 may be perfect when the base tables can supply data as soon as query execution begins. But if some table, say Sales, were on a remote data source, its tuples might arrive in a delayed or bursty fashion. Assuming a standard iterator-style query execution, a delay in Sales will stall the whole plan.

Query scrambling [118] is a method which reacts to such delays by rescheduling the order of evaluation of query plan operators, and occasionally changing the query plan by introducing new operators. Whenever a plan is stalled due to a delayed source, query scrambling considers two alternatives for continuing progress on the query:

- *Rescheduling of existing operators*: The simplest solution is to retain the current plan and react to a delay in one part of the plan by scheduling other, non-delayed sub-trees for execution. For example, Store and Region could be joined

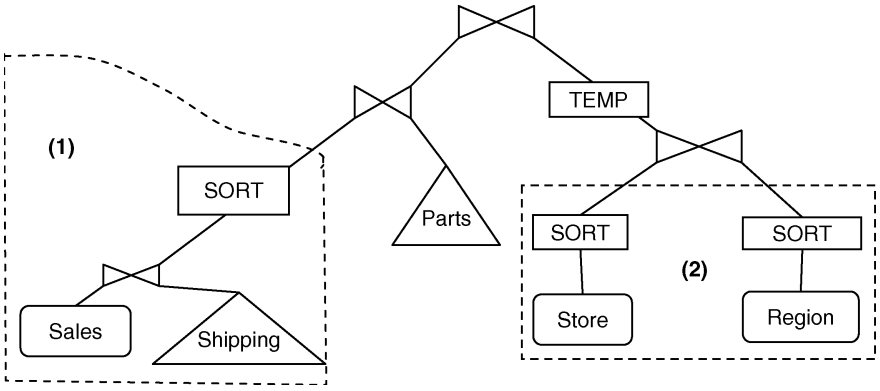

Fig. 8.5 Query scrambling example. When a delay in arrival of sales causes plan portion (1) to stall, we can run plan portion (2) until the delay is resolved.

while Sales is delayed, as indicated in Figure 8.5. The result (e.g, Store ⋈ Region) has to be materialized until the delayed sources becomes available and the original plan can resume. This materialization can have a significant cost, so query scrambling invokes a query optimizer to choose sub-trees to schedule that have (relatively) high savings in response time for low materialization overheads.

- *Synthesis of new operators*: In some cases, none of the operators in the current plan can be scheduled, because of delays or dependencies. In such situations, query scrambling introduces and executes *new* operators, which join tables that are not directly joined in the original plan (these tables could be intermediate results from previous joins). This process results in the query being evaluated in a piecemeal fashion, as in plan staging.

Urhan *et al.* [118] study several strategies for synthesizing new plan operators during query scrambling. The straightforward approach, called *Pair*, synthesizes and runs the join that will finish fastest. This process is repeated as long as the delayed data has not arrived. When the data arrives, *Pair* invokes the optimizer to construct a single query plan that best uses all the synthesized joins.

The other approach is to use a response-time based optimizer, to generate a complete alternative plan that will finish the query fastest, given that some source is currently delayed. The challenge with this approach is that the duration of the delay is hard to predict. Urhan *et al.* [118] analyze two policies for estimating this delay: one which chooses a very large delay, and one which chooses a small initial delay and repeatedly increases it if the data does not arrive in time. Experimentally, [118] observe that scrambling using a response time optimizer can hide delays quite well, for delays that are smaller than the normal query response time.

8.4 Summary and Post-mortem Analysis

Plan staging and its variants are possibly the most widely used adaptation techniques along with selection ordering. They have been used in

several independent systems and are generally believed to require less "surgery" to a conventional DBMS than other techniques. Table 8.1 compares some of these techniques using the adaptivity loop.

In a post-mortem analysis, the runtime behavior of these techniques is almost identical to a traditional optimize-then-execute query processor: they also execute a single query plan over the entire input relations. The only difference is that this query plan is not fixed in advance but can change at pre-defined materialization points (for plan staging), at checkpoints (for mid-query reoptimization), or when there are delays (for query scrambling). Additionally, in some cases, these techniques might do some extra work in generating an intermediate result that is thrown away when the query is reoptimized.

Post-mortem: Plan staging, mid-query reoptimization, and query scrambling use a single plan for processing all tuples of the input relations. This plan is chosen dynamically at query run-time in response to events such as unexpected cardinalities or source delays.

Table 8.1 Comparing some of the techniques discussed in this section using the adaptivity loop.

Plan staging
Measurement: Sizes of intermediate results. *Analysis and planning:* None. *Actuation:* Resubmit next stage of query to the DBMS.
Mid-query reoptimization [9, 75, 87]
Measurement: Cardinalities or other table statistics computed at checkpoints. *Analysis:* Detect violations of validity ranges. *Planning:* Resubmit a changed query [75], or re-invoke optimizer on original query, exposing intermediate results as materialized views [87], or switch to a pre-chosen alternative plan [9]. *Actuation:* Instantiate operators according to the new plan and start executing them.
Query scrambling [118]
Measurement: Delays in data arrival. *Analysis and planning:* Choose a join that can be run during the delay, or re-invoke the optimizer to synthesize new operators. *Actuation:* Schedule an operator that can be executed given the availability of data sources.

A significant appeal of these techniques is that they impose almost no overhead for monitoring or analysis during the main-line query execution. Unless reoptimization is triggered, each checkpoint functions as a single counter, whereas most of the previous techniques monitor multiple selectivities and costs.

However, the applicability of plan staging is primarily to plans with materialization points. Pipelined plans could also be reoptimized, but almost all reoptimization will involve wasted work. Another open issue is reoptimization in parallel (shared nothing) query plans, where a checkpoint triggering reoptimization needs to achieve global consensus that the reoptimization is worthwhile and intermediate result reuse is hard because not all nodes may have progressed to the same point.

9

Summary and Open Questions

Declarative query processing is a fundamental value proposition of a DBMS. Traditionally query processors provided this feature by first optimizing a query and then executing it. Over the years, this model has broken down due to serious problems in estimating cardinalities, increasing query complexity, and new DBMS environments. In this survey, we have seen a variety of techniques that attempt to alleviate these problems. In this section, we briefly recap the major trade-offs and constraints, identify the common mechanisms and methodologies, and pose a number of challenge problems.

9.1 Trade-Offs and Constraints

As we have seen throughout this survey, adaptive query processing can take many forms. In general, a particular adaptive solution must find a particular point among the spectrum of trade-offs among a variety of different dimensions. We briefly review some of the major issues and the approaches taken.

9.1.1 Precomputation vs. Robustness vs. Runtime Feedback

As we have seen in Section 2, commercial DBMSs employ the model of cost-based optimization developed for System R, which relies on collection of information prior to processing a query in order to choose a good plan. As the limitations of this model have become evident, one approach has been to choose *robust* plans, which may not be optimal according to the cost model but may be more resilient to unanticipated variations in the data, or to provide several *parametric* or *contingent* plans that are appropriate for different settings. Alternatively, one can re-calibrate the cost model (e.g., capturing correlations) based on the results of each query. All of these models assume that some cost information is available, and that it is useful in making some rough predictions about actual performance.

Intra-query adaptivity, upon which we focused in this survey, is at the opposite extreme, and adapts query plans being used mid-execution. Section 4 describes in detail a number of techniques for reordering (possibly expensive) selection operations. Sections 6–8 examine techniques that additionally consider the presence of joins and other operators, which are state-preserving and much more difficult to reason about. Some of these strategies focus on adapting plans in the middle of pipelined execution, whereas others focus on changing the plan at the end of a blocking operation. Each has certain benefits: the former are more responsive, whereas the latter have less potential overhead. In general, the blocking-based strategies have seen more adoption in commercial systems, whereas the pipelined strategies have been widely adopted in distributed and stream query processing settings.

9.1.2 Data Access Restrictions

Some of the techniques designed for recalibrating cost estimates (Section 2) can employ sampling and other techniques that exploit random access to data. Likewise, parametric query optimization and some of its variants assume that certain information about the source data (e.g., its actual cardinalities) is known.

In contrast, many other intra-query adaptive query processing techniques (particularly those of Sections 6 and 7) assume that access to the data is restricted to single-pass sequential access, in some arbitrary order, potentially with only a rough estimate of cardinality. Many of these techniques are specifically directed at distributed remote data sources with limited capabilities, where random access is unavailable and subsequent requests may result in data inconsistency.

9.1.3 Exploration vs. Exploitation

Adaptive query processing obtains feedback about real operating conditions and costs; however, it simultaneously has a goal of producing answers to the query. It is important to note that devoting resources to *exploration* — obtaining information about thus-unknown costs — may help in finding a better query processing strategy, but in the short term it detracts from *exploitation* — producing answers with the best current plan. These two aspects must be carefully balanced, depending on the information available at the time and the variability in the environment. These two factors also interact with the plan search space and any restrictions on when a plan may be changed (both discussed below).

In general, with the prominent exceptions of competitive execution and eddies (Section 3) and their descendants, which invest resources on exploration, most adaptive query processing strategies conservatively focus on exploitation unless they detect a better alternative. Little work has been done on considering when to actively probe or how to do better modeling.

9.1.4 Optimizer Search Space

The next major question is the space of alternative plans that is to be explored: how many options must the adaptive strategy consider? In the simplest cases (Section 8), the search space closely parallels that of traditional optimization; in other cases (e.g., eddies), each incoming tuple may be processed in what is essentially a unique query plan, due to horizontal partitioning. See the post-mortems in Sections 6–8 for more detail.

Of course, a greater search space provides more possible means of improving performance, but potentially at the cost of greater search overhead. One of the least-understood aspects of the adaptive query processing problem is the topology of the search space (particularly with respect to changing pipelined plans, as in Sections 4–7) and how it might be most effectively exploited.

9.1.5 Plan Flexibility vs. Runtime Overhead

Related to the above issue, there is also a question of the space of possible actions that can be taken to adapt execution: in other words, what restrictions have been placed on changing a plan.

In modern query engines, pipelined query execution is heavily optimized, minimizing the amount of copying and computation done at each step. Modifying the engine to support dynamic changes to the plan is not free: it often requires additional copying or buffering, and perhaps the creation of additional state. We have seen a number of techniques that are highly restricted in when they can change a plan (Section 8) as well as ones that are extraordinarily flexible (Sections 6 and 7). The ultimate conclusion is that there is no single ideal adaptive set of techniques; performance will vary depending on how much flexibility is required in a given application.

9.1.6 State Management and Reuse

The final trade-off that we wish to discuss lies in the management of state (intermediate results) in multi-pass (e.g., join and aggregation) operators. The minimum amount of state that must be maintained includes those source tuples that ultimately contribute to the answers.

However, nearly every query processing strategy (adaptive or not) maintains more state than this, for a number of reasons: (1) sometimes we cannot determine whether a source tuple will contribute to an answer until the query completes; (2) retention of intermediate results may speed up performance (e.g., it is often faster to compute $A \bowtie B$, save the sub-result, and then join that with C, rather than to do a ternary join among $A \bowtie B \bowtie C$). On the other hand, a "bad" query plan may produce intermediate results that *increase* the cost of com-

putation, e.g., because $A \bowtie B$ is very large, even though $(A \bowtie C) \bowtie B$ is not.

Thus a challenge is to determine whether to compute intermediate results, when to reuse them, and when to discard them. A major difference between mid-query reoptimization [75] and progressive optimization [87] (Section 8) is the constraints they put on state reuse. Likewise, the new operators that have been proposed for use with eddies — SteMs (Section 6.2.3) and STAIRs (Section 7.3) — adopt different strategies for state management; whereas SteMs only maintain state for base inputs, STAIRs can also maintain (and even migrate) state for partial results. Another scheme that focuses heavily on intermediate state management is corrective query processing (Section 7.1).

It is worth remarking that the state problem is often dramatically reduced in the emerging realm of data stream management systems, where relations within a join are often windowed, bounding their size in ways that enable them to be treated as filters [10]. Additionally, in many settings "punctuation" markers can be used to indicate when data may be flushed from the buffer [116].

9.2 Adaptive Mechanisms

As we have seen, the complete space of adaptive query processing techniques is quite broad and varied. However, there are a number of fundamental techniques that emerge in a number of cases.

Horizontal partitioning, running different plans on different portions of the data, is used in n-way symmetric hash joins, corrective query processing, eddies, content-based routing, and conditional planning. In some cases the partitioning is explicit (most notably in corrective query processing, content-based routing, and conditional planning); in other cases it is implicit in the functioning of the operator.

Query execution by tuple routing is a highly flexible mechanism for adaptation, used in selection ordering, eddies, MJoins, SteMs, STAIRs, and a variety of other techniques.

Plan partitioning, where the optimization and execution steps are interleaved many times and execution progresses in stages, allows the plan-

ning to be done by a traditional query optimizer. The principal limitation of this approach is that it is applicable only at some well-defined points during query execution. The principal benefit is that there is little query execution overhead to adaptivity. The methods of Section 8 all use plan partitioning in some way.

Runtime binding decisions, where certain plan choices are deferred until runtime, allow the execution engine to choose among several alternative plans (including potentially re-invoking the optimizer). Examples include choose nodes, proactive reoptimization, SHARP, and to some extent the techniques of Section 8.

In-operator adaptive logic puts scheduling and other decisions into individual query operators, rather than relying on the optimizer to make decisions. This reduces the overall optimizer search space, and it also allows the query engine to rapidly respond to delays, varying selectivities, and so on. Examples include symmetric hash joins, eddies, and MJoins.

9.3 Conclusions and Challenge Problems

In many ways, adaptive query processing is still in its early stages: the focus has largely been on engineering adaptive techniques for specific contexts, rather than on developing a formal model of adaptivity. In this survey, we attempted to provide an overview of the current state of adaptive query processing, to help identify some of the key features and trade-offs involved in the field, and, we hope, to lay out the problem space in a way that will aid others in truly understanding it. We are convinced that adaptivity is "the way to go" in terms of query processing, especially as the database query paradigm continues to expand to new domains. We end with a series of "challenge problems" that we feel are especially fascinating.

9.3.1 Understanding the Adaptive Plan Space

Many of the adaptive techniques we have seen, particularly those that do tuple routing (e.g., eddies, MJoins) completely change the plan space of query processing: instead of a single plan for the lifetime of the query,

we instead see a sequence of different plans over different tuples, and sometimes the operators are n-ary joins or hybridized join algorithms. To this point, our understanding of the "topology" of the traditional query plan space is fairly limited, and it seems vital to understand how adaptivity and hybrid operators change the problem. Moreover, only a limited amount of work (e.g., corrective query processing) has focused on how to perform adaptive execution of complex nested SQL and other queries, so we do not have a full understanding of how these complex operators affect the adaptive plan space.

9.3.2 Developing Optimal Policies

Most of the techniques we have discussed are new *mechanisms* for adaptation — they address the "actuate" step. Their focus is on changing aspects of a query plan during execution without wasting much work. However, there has been considerably less progress on adaptation *policies* — the "monitor," "plan," and "analyze" steps. Most existing work either follows the tuple routing strategy or attempts to calibrate a traditional cost model. Replanning generally occurs when costs exceed some sort of validity range. None of these strategies fully considers optimal or near-optimal strategies for combining existing knowledge, selective probes, and runtime feedback. One field that could possibly provide useful in this regard is machine learning, in particular the literature on multi-armed bandit problems [16] and on optimal online learning [62, 77]. The competitive analysis framework and analytical tools common in that community might provide useful input on how to route tuples, how often to monitor, and so forth.

9.3.3 Effective Handling of Correlation

Correlation is a major source of cardinality estimation errors and poor plans, but is not easily solved by adaptation. Consider a query with n predicates where there is endemic positive correlation (i.e., compound selectivities larger than product of individual selectivities). An optimizer might choose an initial join order $((R \bowtie S) \bowtie T) \bowtie U$. After finding that $R \bowtie S$ is much larger than expected and realizing the correlation between R and S, the query processor might switch to a

plan than joins S and T, only to learn that they are also correlated. This process can continue as it slowly learns about all the correlation present in the system — each time, the query processor will switch to plan that joins tables not joined until then. This phenomenon is easily observed during mid-query reoptimization and is described as "fleeing from knowledge to ignorance" in [44]. In some cases that can be a very good strategy, as it focuses on exploring the unknown; however, in other cases it simply results in a continuous succession of bad plans. Again, we are hopeful that techniques from machine learning might be helpful in this area.

9.3.4 Managing Resource Sharing

Traditional query optimization is done under the assumption that a query has full use of the machine (or a slice of the machine). Of course, from the early days of DBMSs, the buffer pool has been a mechanism for supporting sharing (as discussed in, e.g., [29]), and there do exist a number of runtime techniques for sharing among filescans and even simultaneously running pipelines [48]. However, we believe that there is significant opportunity to use adaptive techniques to adjust to resource contention (e.g., between operators or between plans) and to perform an adaptive version of multi-query optimization [100, 103]. A significant challenge will be managing the interaction between adaptivity within a plan and the fact that certain subexpressions will be shared across plans.

9.3.5 Scaling to Out-of-core Execution

Most adaptive query processing techniques — at least those that focus on adapting in mid-pipeline — assume that the majority of processing will be done in memory. Some strategies (e.g., the variants on the symmetric hash join, and corrective query processing) do indeed work with larger-than-memory workloads, but they do not fully consider the impact of disk overflow on performance. For instance, the query processor might adapt several operators in a query plan once one operator overflows, in the manner of hash teams [55]. We believe there is significant exploration to be done in this space.

9.3.6 Developing Adaptation Metrics

For a long time, the metric for a DBMS was simple: run transactions at high throughput and run queries fast. Today, query execution is not an isolated activity but rather part of a larger business process. As a result, predictable behavior is vital: e.g., if a new feature speeds up 99% of queries by a factor of two and slows down 1% by a factor of two, that can be a disastrous outcome. Even robust query plans do not precisely capture this notion: they optimize for a *plan* that are likely to perform similarly over a range of queries — but they do not consider how to make the *optimizer* produce more consistent plans. A second issue is interactivity. User interaction has traditionally been ignored by DBMSs, but analysts doing ad hoc queries may wish to work directly on the data rather than on a cached copy. Should the goal for an adaptive query processor be that it dynamically respond to user needs, and support tasks like query refinement and approximate results directly in the query processor? There has been little progress in this area besides some work on online aggregation [61]. In general, it seems that we need new measures for query processing performance that take into account new aspects such as robustness and consistency, rapid refinement, reaction to new conditions, and so forth.

As we have seen from this list of open challenges, the adaptive query processing space still has a wide-open frontier. We look forward to upcoming developments with great anticipation, particularly the development of formal models and theoretical foundations.

Acknowledgments

We would like to express our thanks to our collaborators in the field of adaptive query processing, especially Sirish Chandrasekaran, Mike Franklin, Peter Haas, Alon Halevy, Joe Hellerstein, Sailesh Krishnamurthy, Sam Madden, Volker Markl, Mehul Shah, and Dan Weld. We are also grateful to our editor, Joe Hellerstein, and the anonymous reviewers of this article for their constructive criticism of the earlier drafts of the article. Many thanks also to Now publishers, and especially to James Finlay and Mike Casey. Portions of this work have been funded by NSF grants IIS-0546136, IIS-0477972, IIS-0513778 and DARPA HR0011-06-1-0016.

References

[1] A. Aboulnaga and S. Chaudhuri, "Self-tuning histograms: building histograms without looking at data," in *SIGMOD '99: Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, (New York, NY, USA), pp. 181–192, ACM Press, 1999.

[2] L. Amsaleg, M. J. Franklin, A. Tomasic, and T. Urhan, "Scrambling query plans to cope with unexpected delays," in *Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems,* Miami Beach, FL, pp. 208–219, IEEE Computer Society, December 18–20 1996.

[3] G. Antoshenkov and M. Ziauddin, "Query processing and optimization in Oracle Rdb," *The VLDB Journal*, vol. 5, no. 4, pp. 229–237, 1996.

[4] A. Arasu, S. Babu, and J. Widom, "The CQL continuous query language: Semantic foundations and query execution," *The VLDB Journal*, vol. 15, no. 2, pp. 121–142, 2006.

[5] R. H. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. E. Culler, J. M. Hellerstein, D. Patterson, and K. Yelick, "Cluster I/O with River: making the fast case common," in *IOPADS '99: Proceedings of the sixth workshop on I/O in parallel and distributed systems*, (New York, NY, USA), pp. 10–22, ACM Press, 1999.

[6] R. Avnur and J. M. Hellerstein, "Eddies: continuously adaptive query processing," in *SIGMOD '00: Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, (New York, NY, USA), pp. 261–272, ACM Press, 2000.

[7] B. Babcock and S. Chaudhuri, "Towards a robust query optimizer: a principled and practical approach," in *SIGMOD '05: Proceedings of the 2005 ACM*

SIGMOD international conference on Management of data, (New York, NY, USA), pp. 119–130, ACM Press, 2005.

[8] S. Babu and P. Bizarro, "Adaptive query processing in the looking glass," in *CIDR '05: Second Biennial Conference on Innovative Data Systems Research*, pp. 238–249, Asilomar, CA, 2005.

[9] S. Babu, P. Bizarro, and D. DeWitt, "Proactive re-optimization," in *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data,* Baltimore, MD, pp. 107–118, New York, NY: ACM Press, June 14–16 2005.

[10] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, and J. Widom, "Adaptive ordering of pipelined stream filters," in *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, (New York, NY, USA), pp. 407–418, ACM Press, 2004.

[11] S. Babu, K. Munagala, J. Widom, and R. Motwani, "Adaptive caching for continuous queries," in *ICDE '05: Proceedings of the 21st International Conference on Data Engineering (ICDE'05)*, (Washington, DC, USA), pp. 118–129, IEEE Computer Society, 2005.

[12] S. Babu and J. Widom, "Continuous queries over data streams," *SIGMOD Rec.*, vol. 30, no. 3, pp. 109–120, 2001.

[13] S. Babu and J. Widom, "StreaMon: an adaptive engine for stream query processing," in *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, (New York, NY, USA), pp. 931–932, ACM Press, 2004.

[14] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker, "Fault-tolerance in the Borealis distributed stream processing system," in *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, (New York, NY, USA), pp. 13–24, ACM Press, 2005.

[15] L. Bellatreche, K. Karlapalem, M. K. Mohania, and M. Schneider, "What can partitioning do for your data warehouses and data marts?," in *IDEAS '00: Proceedings of the 2000 International Symposium on Database Engineering & Applications*, (Washington, DC, USA), pp. 437–446, IEEE Computer Society, 2000.

[16] D. A. Berry and B. Fristedt, *Bandit Problems: Sequential Allocation of Experiments*. Springer, October 1985.

[17] P. Bizarro, S. Babu, D. DeWitt, and J. Widom, "Content-based routing: different plans for different data," in *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pp. 757–768, VLDB Endowment, 2005.

[18] P. Bizarro and D. DeWitt, "Adaptive and robust query processing with SHARP," Tech. Rep. 1562, University of Wisconsin – Madison, CS Dept., 2006.

[19] P. G. Bizarro, *Adaptive Query Processing: Dealing with Incomplete and Uncertain Statistics*. PhD thesis, University of Wisconsin – Madison, 2006.

[20] P. D. Bra and J. Paredaens, "Horizontal decompositions and their impact on query solving," *SIGMOD Rec.*, vol. 13, no. 1, pp. 46–50, 1982.

[21] N. Bruno and S. Chaudhuri, "Exploiting statistics on query expressions for optimization," in *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, (New York, NY, USA), pp. 263–274, ACM Press, 2002.

[22] N. Bruno, S. Chaudhuri, and L. Gravano, "STHoles: a multidimensional workload-aware histogram," in *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, (New York, NY, USA), pp. 211–222, ACM Press, 2001.

[23] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah, "TelegraphCQ: Continuous dataflow processing for an uncertain world," in *CIDR '03: First Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, 2003.

[24] S. Chaudhuri, "An overview of query optimization in relational systems," in *PODS '98: Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, (New York, NY, USA), pp. 34–43, ACM Press, 1998.

[25] S. Chaudhuri, U. Dayal, and T. W. Yan, "Join queries with external text sources: Execution and optimization techniques," in *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pp. 410–422, ACM Press, May 26 1995.

[26] S. Chaudhuri and K. Shim, "Including group-by in query optimization," in *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, (San Francisco, CA, USA), pp. 354–366, Morgan Kaufmann Publishers Inc., 1994.

[27] C. M. Chen and N. Roussopoulos, "Adaptive selectivity estimation using query feedback," in *SIGMOD '94: Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, (New York, NY, USA), pp. 161–172, ACM Press, 1994.

[28] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang, "NiagaraCQ: a scalable continuous query system for internet databases," in *SIGMOD '00: Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, (New York, NY, USA), pp. 379–390, ACM Press, 2000.

[29] H.-T. Chou and D. J. DeWitt, "An evaluation of buffer management strategies for relational database systems," in *VLDB '85: Proceedings of 11th International Conference on Very Large Data Bases*, pp. 127–141, Stockholm, Sweden: Morgan Kaufmann, August 21–23 1985.

[30] F. Chu, J. Halpern, and J. Gehrke, "Least expected cost query optimization: what can we expect?," in *PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, (New York, NY, USA), pp. 293–302, ACM Press, 2002.

[31] F. Chu, J. Y. Halpern, and P. Seshadri, "Least expected cost query optimization: an exercise in utility," in *PODS '99: Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, (New York, NY, USA), pp. 138–147, ACM Press, 1999.

[32] J. Claussen, A. Kemper, G. Moerkotte, K. Peithner, and M. Steinbrunn, "Optimization and evaluation of disjunctive queries," *IEEE Transactions on Knowledge and Data Engineering*, vol. 12, no. 2, pp. 238–260, 2000.

[33] E. F. Codd, "A relational model of data for large shared data banks," *Commun. ACM*, vol. 26, no. 1, pp. 64–69, 1983.

[34] R. L. Cole and G. Graefe, "Optimization of dynamic query evaluation plans," in *SIGMOD '94: Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pp. 150–160, Minneapolis, MN: ACM Press, May 24–27 1994.

[35] A. Condon, A. Deshpande, L. Hellerstein, and N. Wu, "Flow algorithms for two pipelined filter ordering problems," in *PODS '06: Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, (New York, NY, USA), pp. 193–202, ACM Press, 2006.

[36] D. Daniels, "Query compilation in a distributed database system," Tech. Rep., IBM, 1982. Research Report RJ 3423.

[37] A. Deshpande, "An initial study of overheads of eddies," *SIGMOD Rec.*, vol. 33, no. 1, pp. 44–49, 2004.

[38] A. Deshpande, M. Garofalakis, and R. Rastogi, "Independence is good: dependency-based histogram synopses for high-dimensional data," in *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, (New York, NY, USA), pp. 199–210, ACM Press, 2001.

[39] A. Deshpande, C. Guestrin, W. Hong, and S. Madden, "Exploiting correlated attributes in acquisitional query processing," in *ICDE '05: Proceedings of the 21st International Conference on Data Engineering (ICDE'05)*, (Washington, DC, USA), pp. 143–154, IEEE Computer Society, 2005.

[40] A. Deshpande and J. M. Hellerstein, "Lifting the burden of history from adaptive query processing," in *VLDB '04: Proceedings of the 30th International Conference on Very Large Data Bases,* Toronto, Canada, August 29–September 3 2004.

[41] A. Deshpande and L. Hellerstein, "Flow algorithms for parallel query optimization," Tech. Rep. CS-TR-4873, University of Maryland at College Park, 2007.

[42] L. Ding and E. A. Rundensteiner, "Evaluating window joins over punctuated streams," in *CIKM '04: Proceedings of the thirteenth ACM international conference on Information and knowledge management*, (New York, NY, USA), pp. 98–107, ACM Press, 2004.

[43] O. Etzioni, S. Hanks, T. Jiang, R. M. Karp, O. Madani, and O. Waarts, "Efficient information gathering on the internet," in *FOCS '96: Proceedings of the 37th Annual Symposium on Foundations of Computer Science*, pp. 234–243, IEEE Computer Society, 1996.

[44] S. Ewen, H. Kache, V. Markl, and V. Raman, "Progressive query optimization for federated queries," in *EDBT '06: Proceedings of the 10th International Conference on Extending Database Technology*, pp. 847–864, 2006.

[45] U. Feige, L. Lovász, and P. Tetali, "Approximating min-sum set cover," *Algorithmica*, vol. 40, no. 4, pp. 219–234, 2004.

[46] S. Ganguly, "Design and analysis of parametric query optimization algorithms," in *VLDB '98: Proceedings of the 24th International Conference on Very Large Data Bases*, pp. 228–238, Morgan Kaufmann, August 24–27 1998.

[47] S. Ganguly, W. Hasan, and R. Krishnamurthy, "Query optimization for parallel execution," in *SIGMOD '92: Proceedings of the 1992 ACM SIGMOD international conference on Management of data*, (New York, NY, USA), pp. 9–18, ACM Press, 1992.

[48] K. Gao, S. Harizopoulos, I. Pandis, V. Shkapenyuk, and A. Ailamaki, "Simultaneous pipelining in QPipe: Exploiting work sharing opportunities across queries," in *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*, (Washington, DC, USA), p. 162, IEEE Computer Society, 2006.

[49] L. Getoor, B. Taskar, and D. Koller, "Selectivity estimation using probabilistic models," in *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, (New York, NY, USA), pp. 461–472, ACM Press, 2001.

[50] R. Goldman and J. Widom, "DataGuides: Enabling query formulation and optimization in semistructured databases," in *VLDB '97: Proceedings of 23rd International Conference on Very Large Data Bases*, pp. 436–445, Athens, Greece: Morgan Kaufman, August 25–29 1997.

[51] R. Goldman and J. Widom, "WSQ/DSQ: a practical approach for combined querying of databases and the web," in *SIGMOD '00: Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, (New York, NY, USA), pp. 285–296, ACM Press, 2000.

[52] G. Graefe and K. Ward, "Dynamic query evaluation plans," in *SIGMOD '89: Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, (New York, NY, USA), pp. 358–366, ACM Press, 1989.

[53] G. Graefe, "Query evaluation techniques for large databases," *ACM Comput. Surv.*, vol. 25, no. 2, pp. 73–169, 1993.

[54] G. Graefe, "The Cascades framework for query optimization," *IEEE Data Engineering Bulletin*, vol. 18, no. 3, pp. 19–29, 1995.

[55] G. Graefe, R. Bunker, and S. Cooper, "Hash joins and hash teams in Microsoft SQL Server," in *VLDB '98: Proceedings of 24th International Conference on Very Large Data Bases*, pp. 86–97, Morgan Kaufman, August 24–27 1998.

[56] G. Graefe and W. J. McKenna, "The Volcano optimizer generator: Extensibility and efficient search," in *ICDE '93: Proceedings of the Ninth International Conference on Data Engineering,* Vienna, Austria, pp. 209–218, IEEE Computer Society, April 19–23 1993.

[57] H. Guo, P.-Å. Larson, R. Ramakrishnan, and J. Goldstein, "Relaxed currency and consistency: how to say "good enough" in SQL," in *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, (New York, NY, USA), pp. 815–826, ACM Press, 2004.

[58] L. M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh, "Extensible query processing in starburst," in *SIGMOD '89: Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, (New York, NY, USA), pp. 377–388, ACM Press, 1989.

[59] P. J. Haas and J. M. Hellerstein, "Ripple joins for online aggregation," in *SIGMOD '99: Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, (New York, NY, USA), pp. 287–298, ACM Press, 1999.

[60] J. M. Hellerstein, "Optimization techniques for queries with expensive methods," *ACM Transactions on Database Systems*, vol. 23, no. 2, pp. 113–157, 1998.

[61] J. M. Hellerstein, R. Avnur, A. Chou, C. Hidber, C. Olston, V. Raman, T. Roth, and P. J. Haas, "Interactive data analysis: The Control project," *Computer*, vol. 32, no. 8, pp. 51–59, 1999.

[62] M. Herbster and M. K. Warmuth, "Tracking the best expert," *Machine Learning*, vol. 32, pp. 151–178, August 1998.

[63] D. A. Huffman, "A method for the construction of minimum redundancy codes," in *Proc. Inst. Radio Eng.*, pp. 1098–1101, 1952.

[64] A. Hulgeri and S. Sudarshan, "Parametric query optimization for linear and piecewise linear cost functions.," in *VLDB '02: Proceedings of 28th International Conference on Very Large Data Bases, August 20-23, 2002, Hong Kong, China*, pp. 167–178, 2002.

[65] A. Hulgeri and S. Sudarshan, "AniPQO: Almost non-intrusive parametric query optimization for nonlinear cost functions.," in *VLDB '03: Proceedings of 29th International Conference on Very Large Data Bases, September 9-12, 2003, Berlin, Germany*, pp. 766–777, 2003.

[66] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik, "High-availability algorithms for distributed stream processing," in *ICDE '05: Proceedings of the 21st International Conference on Data Engineering (ICDE'05)*, (Washington, DC, USA), pp. 779–790, IEEE Computer Society, 2005.

[67] T. Ibaraki and T. Kameda, "On the optimal nesting order for computing N-relational joins," *ACM Transactions on Database Systems*, vol. 9, no. 3, pp. 482–502, 1984.

[68] Y. E. Ioannidis, "Query optimization," *ACM Computing Surveys*, vol. 28, no. 1, pp. 121–123, 1996.

[69] Y. E. Ioannidis and S. Christodoulakis, "On the propagation of errors in the size of join results," in *SIGMOD '91: Proceedings of the 1991 ACM SIGMOD international conference on Management of data*, (New York, NY, USA), pp. 268–277, ACM Press, 1991.

[70] Y. E. Ioannidis, R. T. Ng, K. Shim, and T. K. Sellis, "Parametric query optimization," *The VLDB Journal*, vol. 6, no. 2, pp. 132–151, 1997.

[71] Z. G. Ives, *Efficient Query Processing for Data Integration*. PhD thesis, University of Washington, August 2002.

[72] Z. G. Ives, D. Florescu, M. Friedman, A. Levy, and D. S. Weld, "An adaptive query execution system for data integration," in *SIGMOD '99: Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, (New York, NY, USA), pp. 299–310, ACM Press, 1999.

[73] Z. G. Ives, A. Y. Halevy, and D. S. Weld, "Adapting to source properties in processing data integration queries," in *SIGMOD '04: Proceedings of the 2004*

ACM SIGMOD international conference on Management of data, (New York, NY, USA), pp. 395–406, ACM Press, 2004.

[74] Z. G. Ives and N. E. Taylor, "Sideways information passing for push-style query processing," Tech. Rep. MS-CIS-07-14, University of Pennsylvania, 2007.

[75] N. Kabra and D. J. DeWitt, "Efficient mid-query re-optimization of sub-optimal query execution plans," in *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, (New York, NY, USA), pp. 106–117, ACM Press, 1998.

[76] H. Kaplan, E. Kushilevitz, and Y. Mansour, "Learning with attribute costs," in *STOC '05: Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, (New York, NY, USA), pp. 356–365, ACM Press, 2005.

[77] M. Kearns and S. Singh, "Near-optimal reinforcement learning in polynomial time," *Machine Learning*, vol. 49, pp. 260–268, November 2002.

[78] W. Kim, "On optimizing an SQL-like nested query," *ACM Transactions on Database Systems*, vol. 7, no. 3, pp. 443–469, 1982.

[79] D. Kossmann, "The state of the art in distributed query processing," *ACM Comput. Surv.*, vol. 32, no. 4, pp. 422–469, 2000.

[80] R. Krishnamurthy, H. Boral, and C. Zaniolo, "Optimization of nonrecursive queries," in *VLDB '86: Proceedings of the 12th International Conference on Very Large Data Bases*, (San Francisco, CA, USA), pp. 128–137, Morgan Kaufmann Publishers Inc., 1986.

[81] M. Kutsch, P. J. Haas, V. Markl, N. Megiddo, and T. M. Tran, "Integrating a maximum-entropy cardinality estimator into DB2 UDB," in *EDBT '06: Proceedings of the 10th International Conference on Extending Database Technology*, 2006.

[82] T. Legler, W. Lehner, and A. Ross, "Data mining with the SAP NetWeaver BI accelerator," in *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pp. 1059–1068, VLDB Endowment, 2006.

[83] H.-G. Li, S. Chen, J. Tatemura, D. Agrawal, K. S. Candan, and W.-P. Hsiung, "Safety guarantee of continuous join queries over punctuated data streams," in *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pp. 19–30, VLDB Endowment, 2006.

[84] W.-S. Li, V. S. Batra, V. Raman, W. Han, and I. Narang, "QoS-based data access and placement for federated systems," in *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pp. 1358–1362, VLDB Endowment, 2005.

[85] L. F. Mackert and G. M. Lohman, "R* optimizer validation and performance evaluation for distributed queries," in *VLDB '86: Proceedings of the 12th International Conference on Very Large Data Bases*, (San Francisco, CA, USA), pp. 149–159, Morgan Kaufmann Publishers Inc., 1986.

[86] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman, "Continuously adaptive continuous queries over streams," in *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pp. 49–60, ACM Press, 2002.

[87] V. Markl, V. Raman, D. Simmen, G. Lohman, H. Pirahesh, and M. Cilimdzic, "Robust query processing through progressive optimization," in *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, (New York, NY, USA), pp. 659–670, ACM Press, 2004.

[88] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma, "Query processing, resource management, and approximation in a data stream management system," in *CIDR '03: First Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, 2003.

[89] I. S. Mumick, S. J. Finkelstein, H. Pirahesh, and R. Ramakrishnan, "Magic is relevant," in *SIGMOD '90: Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, (New York, NY, USA), pp. 247–258, ACM Press, 1990.

[90] K. Munagala, S. Babu, R. Motwani, and J. Widom, "The pipelined set cover problem.," in *ICDT '05: Proceedings of the 10th International Conference, Edinburgh, UK*, pp. 83–98, 2005.

[91] K. Munagala, U. Srivastava, and J. Widom, "Optimization of continuous queries with shared expensive filters," in *PODS '07: Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, (New York, NY, USA), pp. 215–224, ACM Press, 2007.

[92] J. Naughton, D. DeWitt, D. Maier, A. Aboulnaga, J. Chen, L. Galanis, J. Kang, R. Krishnamurthy, Q. Luo, N. Prakash, R. Ramamurthy, J. Shanmugasundaram, F. Tian, K. Tufte, S. Viglas, Y. Wang, C. Zhang, B. Jackson, A. Gupta, and R. Chen, "The niagara internet query system," *IEEE Data Engineering Bulletin*, June 2001.

[93] N. Polyzotis, "Selectivity-based partitioning: A divide-and-union paradigm for effective query optimization," in *CIKM '05: Proceedings of the 14th ACM International Conference on Information and knowledge management*, pp. 720–727, New York, NY: ACM Press, 2005.

[94] V. G. V. Prasad, *Parametric Query Optimization: A Geometric Approach*. Master's thesis, IIT Kanpur, 1999.

[95] V. Raman, A. Deshpande, and J. M. Hellerstein, "Using state modules for adaptive query processing.," in *ICDE '03: Proceedings of the 19th International Conference on Data Engineering, Bangalore, India*, pp. 353–364, 2003.

[96] V. Raman and J. M. Hellerstein, "Partial results for online query processing," in *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pp. 275–286, ACM Press, 2002.

[97] V. Raman, B. Raman, and J. M. Hellerstein, "Online dynamic reordering for interactive data processing," in *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*, pp. 709–720, Edinburgh, Scotland: Morgan Kaufmann, 1999.

[98] S. V. U. M. Rao, *Parametric Query Optimization: A Non-Geometric Approach*. Master's thesis, IIT Kanpur, 1999.

[99] L. Raschid and S. Y. W. Su, "A parallel processing strategy for evaluating recursive queries," in *VLDB '86: Proceedings of the 12th International Con-*

ference on Very Large Data Bases, pp. 412–419, Morgan Kaufmann Publishers Inc., 1986.

[100] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe, "Efficient and extensible algorithms for multi query optimization," in *SIGMOD '00: Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, (New York, NY, USA), pp. 249–260, ACM Press, 2000.

[101] E. A. Rundensteiner, L. Ding, T. M. Sutherland, Y. Zhu, B. Pielech, and N. Mehta, "CAPE: Continuous query engine with heterogeneous-grained adaptivity," in *VLDB '04: Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada*, pp. 1353–1356, 2004.

[102] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access path selection in a relational database management system," in *SIGMOD '79: Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, 1979.

[103] T. K. Sellis, "Multiple-query optimization," *ACM Trans. Database Syst.*, vol. 13, no. 1, pp. 23–52, 1988.

[104] P. Seshadri, J. M. Hellerstein, H. Pirahesh, T. Y. C. Leung, R. Ramakrishnan, D. Srivastava, P. J. Stuckey, and S. Sudarshan, "Cost-based optimization for magic: Algebra and implementation," in *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pp. 435–446, ACM Press, 1996.

[105] P. Seshadri, H. Pirahesh, and T. Y. C. Leung, "Complex query decorrelation," in *ICDE '96: Proceedings of the Twelfth International Conference on Data Engineering,* New Orleans, LA, pp. 450–458, February 26–March 1 1996.

[106] M. A. Shah, J. M. Hellerstein, and E. Brewer, "Highly available, fault-tolerant, parallel dataflows," in *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, (New York, NY, USA), pp. 827–838, ACM Press, 2004.

[107] J. Shanmugasundaram, K. Tufte, D. J. DeWitt, J. F. Naughton, and D. Maier, "Architecting a network query engine for producing partial results," in *ACM SIGMOD Workshop on the Web (WebDB) 2000,* Dallas, TX, pp. 17–22, 2000.

[108] M. A. Shayman and E. Fernandez-Gaucherand, "Risk-sensitive decision-theoretic diagnosis," *IEEE Transactions on Automatic Control*, vol. 46, pp. 1166–1171, 2001.

[109] H. Simon and J. Kadane, "Optimal problem-solving search: All-or-none solutions," *Artificial Intelligence*, vol. 6, pp. 235–247, 1975.

[110] U. Srivastava, K. Munagala, and J. Widom, "Operator placement for in-network stream query processing," in *PODS '05: Proceedings of the Twenty-Fourth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pp. 250–258, 2005.

[111] U. Srivastava, K. Munagala, J. Widom, and R. Motwani, "Query optimization over web services," in *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pp. 355–366, VLDB Endowment, 2006.

[112] M. Stillger, G. Lohman, V. Markl, and M. Kandil, "LEO – DB2's LEarning Optimizer," in *VLDB '01: Proceedings of 27th International Conference on Very Large Data Bases*, Morgan Kaufmann, September 11–14 2001.

[113] M. Stonebraker, E. Wong, P. Kreps, and G. Held, "The design and implementation of Ingres," *ACM Transactions on Database Systems*, vol. 1, no. 3, pp. 189–222, 1976.

[114] M. Templeton, H. Henley, E. Maros, and D. J. V. Buer, "InterViso: Dealing with the complexity of federated database access," *The VLDB Journal*, vol. 4, no. 2, 1995.

[115] F. Tian and D. J. DeWitt, "Tuple routing strategies for distributed eddies," in *VLDB '03: Proceedings of 29th International Conference on Very Large Data Bases*, pp. 333–344, Berlin, Germany: Morgan Kaufmann, September 9–12 2003.

[116] P. A. Tucker and D. Maier, "Exploiting punctuation semantics in data streams," in *ICDE '02: Proceedings of the 18th International Conference on Data Engineering*, (Washington, DC, USA), p. 279, IEEE Computer Society, 2002.

[117] T. Urhan and M. J. Franklin, "XJoin: a reactively-scheduled pipelined join operator," *IEEE Data Engineering Bulletin*, vol. 23, no. 2, pp. 27–33, 2000.

[118] T. Urhan, M. J. Franklin, and L. Amsaleg, "Cost based query scrambling for initial delays," in *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, pp. 130–141, Seattle, WA: ACM Press, June 2–4 1998.

[119] E. Viglas and S.-J. F. Naughton, *Novel Query Optimization and Evaluation Techniques*. PhD thesis, University of Wisconsin at Madison, 2003.

[120] S. Viglas, J. F. Naughton, and J. Burger, "Maximizing the output rate of multi-way join queries over streaming information sources," in *VLDB '03: Proceedings of the 29th International Conference on Very Large Data Bases*, Berlin, Germany: Morgan Kaufmann, September 9–12 2003.

[121] A. N. Wilschut and P. M. G. Apers, "Dataflow query execution in a parallel main-memory environment," in *PDIS '91: Proceedings of the First International Conference on Parallel and Distributed Information Systems,* Fontainebleu Hilton Resort, Miami Beach, FL, pp. 68–77, IEEE Computer Society, 1991.

[122] E. Wong and K. Youssefi, "Decomposition — strategy for query processing," *ACM Transactions on Database Systems*, vol. 1, no. 3, pp. 223–241, 1976.

[123] D. Zhang, J. Li, K. Kimeli, and W. Wang, "Sliding window based multi-join algorithms over distributed data streams," in *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*, (Washington, DC, USA), p. 139, IEEE Computer Society, 2006.

[124] Y. Zhu, E. A. Rundensteiner, and G. T. Heineman, "Dynamic plan migration for continuous queries over data streams," in *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, (New York, NY, USA), pp. 431–442, ACM Press, 2004.