Bridging Two Worlds with RICE

Integrating R into the SAP In-Memory Computing Engine

Philipp Große Wolfgang Lehner Thomas Weichert Franz Färber Wen-Syan Li

SAP AG; Dietmar-Hopp-Allee 16; Walldorf, Germany
eMail: {philipp.grosse, wolfgang.lehner, thomas.weichert, franz.faerber, wen-syan.li}@sap.com

ABSTRACT

The growing need to use large amounts of data as the basis for sophisticated business analysis conflicts with the current capabilities of statistical software systems as well as the functions provided by most modern databases.

We developed two novel approaches towards a solution for this basic conflict, based on the widely-used statistical software package R and the SAP In-Memory Computing Engine (IMCE).

We thereby propose an alternative data exchange mechanism with R. Instead of using standard SQL interfaces like JDBC or ODBC we introduced SQL-SHM, a shared memory-based data exchange to incorporate R's vertical data structure. Furthermore, we extended this approach to R-Op introducing R scripts equivalent to native database operations like join or aggregation within the execution plans.

With the calculation engine, IMCE provides a framework to model logical execution plans and thereby offers a convenient way to use the full functionality of R via SQL interface. Moreover, this enables us to run R scripts in parallel without the necessity of extending the R interpreter itself.

1. INTRODUCTION

Data found in today's companies often ranges from terabytes to petabytes in size. On one hand companies gather information based on an extensive data set, such as the commercial behavior of their customers (e.g. based on loyalty cards). On the other hand they have a growing need to use this data as a basis for business analysis. The knowledge hidden in this enormous amount of data is often difficult to explore. One of the reasons for this is the fact that most database systems provide only very limited advanced analytics functionality compared to the comprehensive environment that statistical software packages offer [4].

One of those statistical software packages is the R framework [21]. It is a popular open-source initiative involving an international ecosystem of academics, statisticians, and data miners. With over 2,000 add-on packages, it provides roughly the equivalent to the "big two" commercial packages [15]: SAS and SPSS. It can therefore be used for a variety of different statistical methods, such as linear and nonlinear models, statistical tests, time series analyses, classification and clustering, providing a rich background for advanced analytics.

However most statistical software systems are not designed to handle mass data. In particular, data is usually not stored in these systems, but rather in a database, and therefore, the data has to be propagated at least once. In R, this is normally done by using a standard SQL interface such as JDBC/ODBC, or by using a CSV import. In many business use cases, the overhead for transferring the data is unacceptable, especially, if the statistical software system, like the default R runtime environment, has to wait until all data is transferred before the actual data processing can start.

Modern database systems on the other hand, provide some statistical methods as predefined functions callable via SQL or similar interfaces, but the functionality provided is usually very limited. The major difference is that statistical software packages, like R, do not only provide a number of such predefined functions, but they come with an own language and runtime environment. This fact makes it very easy to introduce new algorithms specifically designed to meet custom-tailored demands, rather than providing a generic set of callable functions.

Requirements. The number of specific algorithms using the R language in both industry and in the scientific environment is enormous. We therefore derive the following requirements for a solution of the basic conflict between the need for sophisticated analytics and the enormous data volumes involved. The solution should:

- Leverage the **parallelization** and efficient data processing capabilities provided by the database system.
- Leverage the expressiveness of the R language and allow **programming**.
- Leverage the **reuse** of already **existing functionality** in R and therefore prevent analysts from reinventing the wheel.
- **Avoid data transportation** between the database system and the statistical software package, or at least reduce communication overhead.

To the best of our knowledge there is no solution that can fully satisfy all of those requirements. Nevertheless, the goal has to be to get as close as possible.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were invited to present their results at The 37th International Conference on Very Large Data Bases, August 29th - September 3rd 2011, Seattle, Washington.
Proceedings of the VLDB Endowment, Vol. 4, No. 12
Copyright 2011 VLDB Endowment 2150-8097/11/08... $ 10.00.

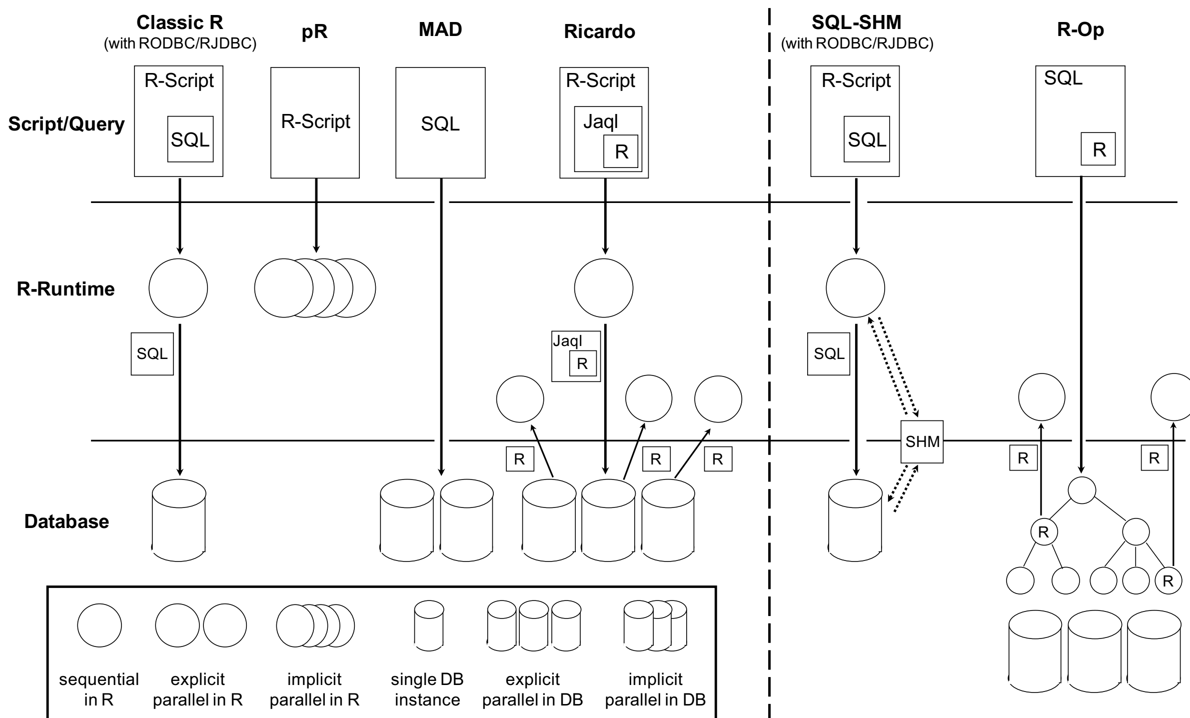


Figure 1: Comparison of different possible approaches and their parallelization.

Paper Organization. In this paper, we are going to outline different approaches towards a transparent solution. Specifically, we will first discuss a variety of existing approaches for advanced analytics on large data sets in Section 2. Followed by Section 3 where we will give a general overview over the SAP In-Memory Computing Engine (IMCE) [8, 16], on which our R integration approach is based. In this context, we argue that the growing need for advanced analytics on mass data can be seen as one of the requirements that a modern database system should take into account. In Subsections 4.1 and 4.2, we are going to outline our two novel approaches (SQL-SHM and R-Op) and in Subsection 4.3 we will discuss a use case for the integration in more detail. Finally, the evaluations follow in Section 5.

2. RELATED WORK

Given the problem that *"statistical software is geared towards deep analytics, but does not scale to large datasets, whereas DMSs scale to large datasets, but have limited analytical functionality."* [5], we identified three general approaches to tackle this deficit. In the following we will discuss each of them aligned with proposals found in the literature. Figure 1 illustrates some of them and their relationship to parallelization and communication between the systems.

1.) Enhance the statistical software environment for mass data support

The first method is to extend the statistical software package in a way that the handling of large amounts of data is improved [17]. For example, there are a number of approaches aiming to parallelize R [19, 10]. Many of them require rewriting and adaptation of existing scripts and functions, but there are also two approaches that are trying to adapt the R environment in a transparent manner. The first, called pR [18], focuses on teaching the R environment to execute loops and function calls in parallel. It uses a mixture of dynamic dependency analysis to identify tasks and loops that can be parallelized, and incremental analysis to delay the processing of conditional branches as well as dynamic loop bounds until the related variables are evaluated. Instead of parallelizing the entire R script at the granularity of individual statements, the authors adopt the master-worker paradigm in such a way that only expensive jobs (such as function calls and loops) are passed to workers, whereas simple statements and conditional statements are executed locally by the master. As shown in Figure 1, pR is the only approach offering transparent and thereby implicit parallelization of R scripts. However, it lacks database support.

Another prominent attempt is called RIOT [24, 23]. It focuses on avoiding intermediate results and making R more I/O efficient by introducing a new expression algebra to R. The basic idea is to use an expression DAG on single R operations and to use it to do database-style optimizations with a series of transformation rules.

Both approaches show very promising results in terms of performance improvements. However, there is—as already discussed—a very general problem concerning any solution that focuses solely on the statistical software side: the need for data transfer from the database to the statistical software system and potentially vice versa.

2.) Enrich the database systems with advanced analytics functionality

Instead of moving data to the processing-centric R infrastructure, another possible approach is to extend the functionality of the database system. In its basic form, we can distinguish between three possible ways. The first and most extreme approach is to specifically redesign the database system to meet the specific requirements of statistical com-

puting, like in SciDB [20]. Obviously, this approach goes far beyond enriching an existing database system.

The second path is the deep integration of individual data-mining and machine-learning algorithms into the code base of the database system itself (e.g. Legler *et al.* [14]). This kind of integration clearly has the advantage of being able to use internal components inside the database system and therefore, to take maximum advantage of the physical data layout and/or parallel execution capabilities. The downside of this deep integration is that it can usually only be done for single algorithms one by one, because it is a very labor-intensive task. Furthermore, the data analyst is bound to the algorithms and parameters that were provided by the database developers; the flexibility to extend or modify parts of the algorithm is therefore very limited.

The third possible path is to use a database query language—such as SQL—to express linear algebra functions or even higher-level algorithms as proposed in MAD [4], and therefore trying to get a database system to act like a statistical software environment. This approach clearly has the advantage of being flexible: the analyst is able to develop algorithms independently on top of the database and nevertheless is able to profit from parallel database execution, as depicted in Figure 1. The general problem with this query approach is the fact that SQL, or any other database query language, is simply not designed to express statistical computations. SQL follows a declarative logic, whereas statistical computing requires imperative and functional programming logic. As stated in RIOT [23] *"SQL is too low-level for representing many linear algebra operations; optimizing at this level is much less effective than if we know the high-level semantics of these operations"*.

3.) Improve cooperation between the database and the statistical software system

An example of the third possible approach is given by Ricardo [5], where it is argued that the two systems (DBMS and statistical software package) should be kept separate and focus on what they are best at. Following the conclusions from Chu *et al.* [3] the authors argue that many data-mining and machine-learning approaches can be split into two parts: a smaller data part where the actual semantic is executed and therefore needs statistical software support, and a second part, which operates on mass data and should be handled in a database system. The authors therefore propose that the analysts should do an extensive study of their algorithms and corresponding implementations to identify those different parts within their given problem and to split their logic in a MapReduce way [6]. The part concerning the mass data shall be expressed as a Jaql query, which can be executed by Hadoop in parallel. The benefit of this approach, as illustrated in Figure 1, is that after extensive investigation of the problem, the MapReduce paradigm helps to fully utilize the parallel Hadoop framework. Even though this idea is very intriguing, the basic assumption that only a few parameters have to be transferred to the statistical software environment does not hold true in general. In many cases—e.g. if a trained model with linear complexity to evaluate is to be applied on a big dataset for classification—the amount of data that has to be passed from the database system to the statistical software side is not to be neglected, which implies that the overall execution time is strongly influenced by it. Even though most of the implementation details of Ricardo can be hidden from end users, a sophisticated understanding of Jaql as well as of Hadoop and the MapReduce paradigm is mandatory in order to refactor algorithms this way.

None of the discussed approaches is fully satisfying. In particular, the combination of avoiding data transfer on the one hand and having a transparent solution for a parallel execution on the other hand is never fulfilled.

3. THE SAP IN-MEMORY COMPUTING ENGINE

In order to outline the general idea of bridging the gap between the data management layer on the one side and sophisticated statistical software packages on the other side, we highlight the key features of SAP's new In-Memory Computing Engine (IMCE) [8, 16]—also named the SAP HANA database. The general goal of IMCE is to provide a main-memory centric data management platform to support pure SQL for classical applications as well as a specific interaction model between SAP applications and the database system. Moreover, the system is designed to provide full transactional behavior in order to support interactive business applications. Finally, IMCE is designed with special emphasis on parallelization ranging from thread and core level up to highly parallel setups over multiple machines.

Figure 2 provides an overview of the general IMCE architecture. As already mentioned, the IMCE operates in a main-memory centric fashion, i.e. most of the data set resides in main memory in a highly compressed format following a huge variety of different compression schemes. Data objects like regular tables may either live in a column store or can be saved in a row store. The column store is typically used to hold large tables for OLAP query access patterns, i.e. scans and aggregation requests. Modifications of the database will be buffered in a delta tree and asynchronously propagated into a new compressed format (merge step). The row store typically is well suited for point access as well as database objects with a high update load, because the explicit merge step is no longer necessary. Obviously, data can be moved between the different stores to allow query expressions with database tables residing in both stores. The internal optimizer decides when to move which pieces of (intermediate) data between the engines to perform the individual database operators within the most suitable engine. The final query plan will then be executed by an engine-agnostic distributed execution framework following an abstract data flow model (see below for details). For the IMCE product, the specific engines for row and column-oriented data management are plugged into the framework – other engines are used for other SAP products, e.g. Enterprise Search.

The different engines are usually located at multiple nodes within one single IMCE landscape sharing a common persistence layer. Unlike classical database systems, all data structures within the column or row store are optimized to be cache aligned, not to be block aligned for optimal use of the classical disk environments. The persistence layer is only used to create consistent snapshots for backup and subsequently allow recovery in the case of a system restart after an explicit shutdown or failure. Every node holds a local transaction manager and a local metadata repository. Both components synchronize their state with their global counter-

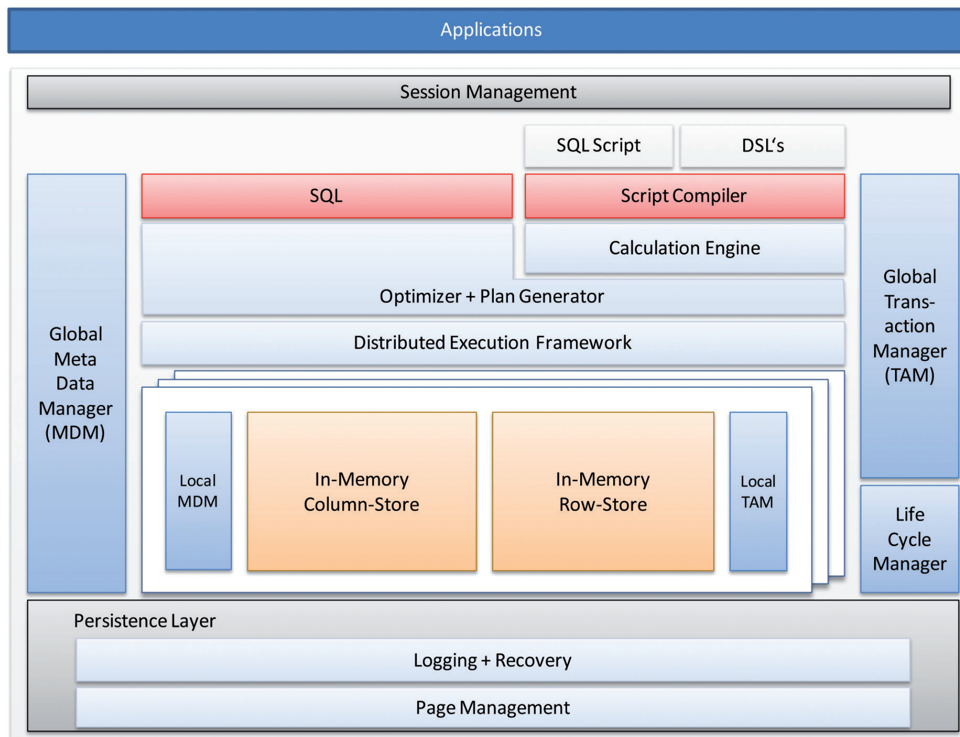


Figure 2: SAP's In-Memory Computing Engine Architecture.

parts. A data lifecycle manager orchestrates the state of different database objects. For example, database objects may be transparently moved from column to row store or vice versa. Moreover, the lifecycle manager may—instrumented by the application—move database objects (tables, partitions, etc.) from main memory to disk. The important difference in comparison to classical database architecture is the fact that the movement is driven by application semantics compared to pure reference behavior in buffer pools. Furthermore, the units of movement are much more coarse-grained compared to small pages in block-oriented database architectures.

From an application perspective, the IMCE provides multiple interfaces, offered by a session manager controlling the individual connections between the database layer and the application layer. IMCE provides a classical SQL interface allowing standard applications to exploit the underlying data management functionality. At the same time, the IMCE provides a more comprehensive interface using the calculation engine component to execute data flow graphs. Data flow graphs (calcModels) reflect an internal abstraction for multiple interfaces. For example, classical procedural SQL extensions are implemented using this technology by compiling SQL extensions to a proprietary intermediate language (script compiler); this code is then further compiled within the calculation engine to the calculation engine primitives. Following this route, multiple domain-specific languages can be supported as long as a compiler generates the IMCE-specific intermediate language.

As already mentioned, the primitives of a calcModel constitute a logical execution plan consisting of an acyclic data flow graph with nodes representing operators (plan operations) and edges reflecting the data flow (plan data). First of all, operators represent classical operations to implement the regular operations of a relational model. In addition, the IMCE supports a huge variety of special operators for directly implementing application-specific (i.e. SAP-specific) components. For example currency conversion from a business perspective is a highly complicated process and directly supported as an application-specific operator. In order to optimally support the SAP business applications, the IMCE provides a predefined set of natively implemented operators. Finally, the IMCE provides a set of non-database language runtimes as operators. For example, Python or JaveScript snippets can be directly plugged into predefined data flow graphs and obviously combined with all other operators provided by the calculation engine. As outlined in the following sections, we exploit the techniques of logical execution plans in combination with generic operators for external language runtimes as the backbone for our R integration.

A specific calcModel or logical execution plan—once submitted to IMCE in an SQL-style syntax—can be accessed in the same way as a database view, making the calcModel a kind of parameterized view. A query consuming a calcModel invokes the database plan execution to process a plan that is derived from the logical dataflow description provided by the calcModel and the individual tables and attributes provided by the query. If the calcModel contains independent data flow paths, the derived execution plan implicitly contains inter-operator parallel execution.

4. THE R INTEGRATION

Starting from the observation that a (fast) data exchange is mandatory for a database system to take advantage of advanced analytics functionality provided by R, our first step was to focus on the interface between R and the IMCE database system. Experimenting with the different interfaces supported by R, we quickly realized that the implemen-

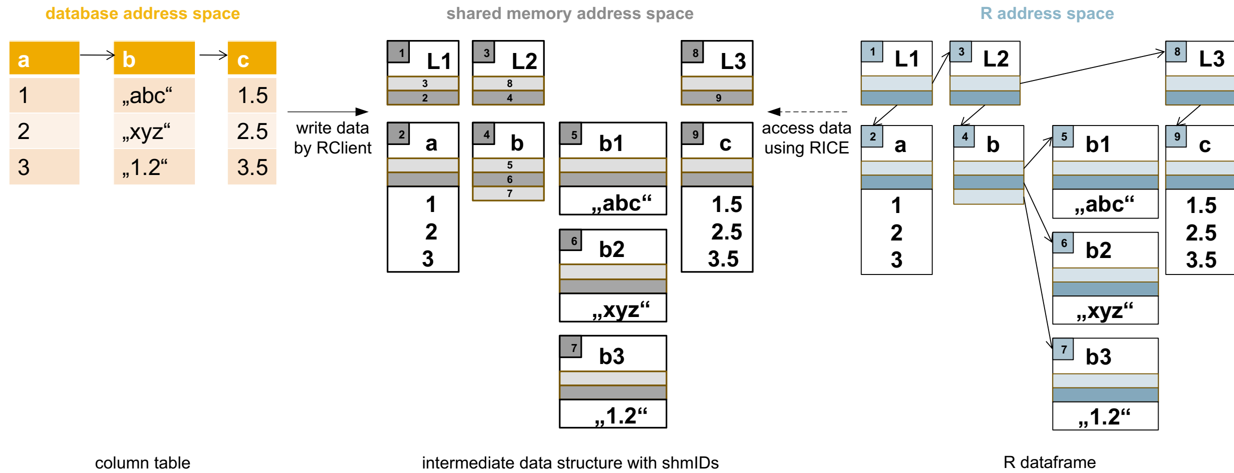


Figure 3: R-like data structure used for data transfer between IMCE and R.

tations provided by external packages to interact with standard database interfaces (like JDBC or ODBC) are not particularly tuned for large datasets. By investigating the reasons we identified a general—and therefore conceptional—problem of any standard SQL interface used for data exchange from within the R environment.

The problem is that R uses a vector based—vertical—data structure [2], whereas all standard SQL interfaces provide query results as tuple based—horizontal—result sets. Providing query results in this way usually makes perfect sense, since the horizontal data organization allows to stream the dataset in chunks. However, in the case of an application with vertical data structures, this streaming mechanism does not work, because the query result has to be fully[1] transferred into an intermediate buffer before the application can start to create the internal vertical data structure. In the case of R, the corresponding data structure for a table is the dataframe, which can be seen as a list of vertical vectors. Conceptionally, the R dataframe is similar to what the database community calls a column table.

Having the data already stored in a column store, like in the SAP In-Memory Computing Engine, it does not seem to be reasonable to use a tuple-based transfer mechanism to transfer the data from column table to "column table".

4.1 The SQL-shortcut

To avoid the previously discussed overhead of standard SQL interfaces we introduce a shared memory (SHM) based data exchange mechanism between R and the SAP In-Memory Computing Engine (IMCE). The general idea, as shown in Figure 4, is to retrieve the data from the IMCE by sending a regular SQL command via a standard database interface (①), but instead of retrieving the result via this interface, we only return a reference to a shared memory (③). Meanwhile, we place the data into shared memory (②) in a format that can easily be used on the R side to construct the dataframe directly without having to copy the data again into its address space (④).

The SQL shared memory solution (SQL-SHM) consists mainly of four parts. The first part is a built-in database function, which allows us to bypass the normal result set return mechanism. This function is called from R by using the *RJDBC* or the *RODBC* package with a regular SQL command passed to the SQL interface of the database. The second part is the so-called RClient, which is particularly responsible for the transformation of the column table query result into the intermediate data structure. Together with the third component, the RClient writes the intermediate data structure into the shared memory. The third component is the Shared Memory Manager (SHM Manager), which is responsible for allocating and freeing shared memory. The SHM Manager is used to allocate big shared memory segments in advance and to orchestrate the nested shared memory blocks, which are required to store the intermediate R-like data structure. The SHM Manager is also used to retrieve the intermediate data structure on the R side and is therefore also part of the fourth component. The fourth component is the external R package *RICE*. This package extends the R environment in a way that allows us to register a dataframe in the R environment, even though the data of the dataframe is in fact still situated in shared memory.

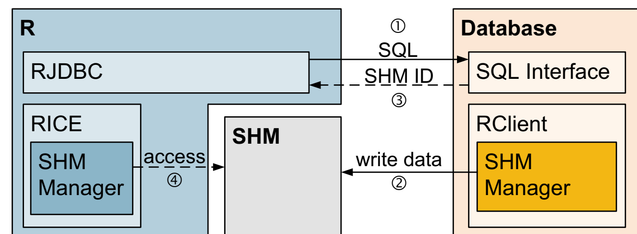


Figure 4: SQL-SHM architecture.

Figure 3 illustrates the intermediate R-like data structure we use to transfer a column table from IMCE to a dataframe in the R environment. The figure shows three different data structure representations of a column table. The first is a column table on the database side, represented as a single linked list of columns. The RClient transforms this data structure into the second representation shown in Figure 3. The intermediate data structure located in the shared memory address space is derived from the target data structure needed on the R side to represent a dataframe. Each block depicts a segment placed in shared memory. Instead of the pointers used on the R side, this intermediate data structure contains shared memory IDs (depicted in Figure 3 as

[1]An R vector is created with a fixed size, which can only be known once all tuples of the given column are transferred.

numbers), which will later be replaced on the R side during the registering process of the dataframe. The called built-in database function therefore only needs to return a reference to the first element—in our example in Figure 3, this would be a reference to the SHM segment L1—in order for the *RICE* package to be able to register the whole dataframe object in the R environment. During this process the SHM Manager on the R side is needed to retrieve the shared memory segments associated with the respective shmIDs.

The fact that our intermediate data structure is very close to the target R data structure allows us to avoid a data copy on the R side. However, the R data structure is not specifically tuned for data transportation. While there is no issue with integer or double columns, which fit as a whole into a single shared memory segment, this is different for string columns. Due to the variable byte size of strings, R represents a character vector (string column) as a list of pointers to the respective strings. For our intermediate shared memory data structure this implies that a string column of the length 3 allocates 4 shared memory segments, as seen in Figure 3. Therefore, the SHM Manager organizes string columns in nested shared memory blocks. To reduce the number of shared memory segments needed to a minimum we introduced a dictionary mechanism.

The SQL-SHM solution is designed in a way that no modifications of the R kernel itself are needed—and all of the shared memory functionality is introduced by our external R package *RICE* wrapping the functionality as ordinary R functions, like `getDataFrame`. The getDataFrame function, which triggers the described data exchange, takes three arguments:

1. An SQL connection, which has to be created using either the *RJDBC* or the *RODBC* package.
2. An SQL select statement.
3. The target name of the newly created dataframe[2].

Since (for technical reasons) the R objects located in shared memory have to be hidden from the normal R garbage collector, it is necessary to explicitly free them from shared memory, if they are not needed anymore. For this purpose, we provide a function called `cleanupObject`. If the function call is omitted, the shared memory associated with the specific dataframe will only be freed once the R runtime has been stopped.

Analogous to the R function `getDataFrame`, we also provide a function called `writeDataFrame`, which triggers the backward data transportation from the R environment to the database. In this case, the *RICE* package writes the dataframe to shared memory and the RClient takes over from there to store the data on the database side.

Script 1 shows an example R script using those SQL-SHM functions to retrieve a table from IMCE. In the first part of the script (lines 1–3) the R external packages *RICE*, *RJDBC* and *kernlab* are loaded into the R environment. For the data transfer only the first two packages are needed, whereas the third provides additional support vector machine (SVM) functionality. The second part of the R script (lines 4–9) sets up the JDBC connection using the functions provided by the *RJDBC* package. In the third part of the script (lines 10–13), this connection is used to retrieve a dataframe using JDBC and our SQL-SHM solution. Both `jdbcTab`

[2]The `getDataFrame` function itself is void to circumvent an additional copy implied by R's copy by value.

```
 1. library(RICE)
 2. library(RJDBC)
 3. library(kernlab)

 4. ## setup JDBC connection 'ch'
 5. drvName = "com.imce.sql.Driver"
 6. drvPath = "/usr/sap/NDB/HDB01/exe/imprsjdbc.jar"
 7. jdbcDriver = JDBC(drvName, drvPath, "'")
 8. con = "jdbc:imce:localhost:30115"
 9. ch = dbConnect(jdbcDriver, con, "userXY", "pw123")

10. ## get table via JDBC and via SQL-SHM
11. sql = "SELECT CLASS, ATT1, ATT2, ATT3 FROM TABLE"
12. jdbcTab = dbGetQuery(ch, sql)   ## use RJDBC
13. getDataFrame(ch, sql, "shmTab") ## use SQL-SHM

14. ## use dataframe 'shmTab'
15. ## for support vector classification
16. model = ksvm(CLASS ~ ., shmTab[1:100,])
17. pm = predict(model, shmTab[101:200,-1])
18. tab = table(pm, shmTab[101:200,1])
19. sum(diag(tab))/sum(tab)

20. ## free the shared memory from 'shmTab' dataframe
21. cleanupObject("shmTab")
```

Script 1: R script using SQL-SHM solution

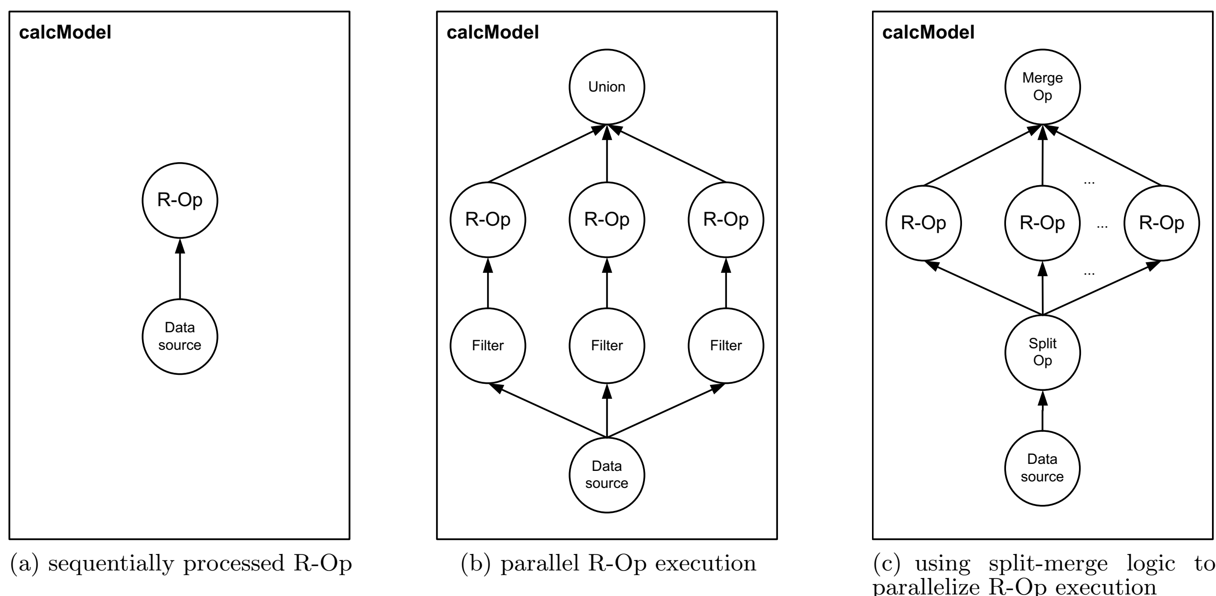
and `shmTab` contain the same contents and only differ in the way they were received. The fourth part (lines 14–19) finally uses the retrieved dataframe for further calculations. In more detail, line 16 calls the *kernlab* function `ksvm` to train a support vector machine model based on the first 100 tuples of the `shmTab` dataframe. In line 17 this SVM model is used to classify the second 100 tuples, which will then be evaluated in line 18 and printed in line 19. In the last line of the R script the `shmTab` is freed from shared memory.

4.2 R as a database operator

Although our first approach, introducing shared memory communication trigged via SQL interface (SQL-SHM), did bridge the gap between the advanced analytic framework R and the database system IMCE, it is not yet an integration of R into the database. In particular, if the result of the advanced analytic functionality is to be used as basis for further classical database operations, the SQL-SHM approach is not sufficient, since the overall control flow is situated on the R side.

In this section we are going to discuss our second approach, integrating R as a database operation (R-Op). The basic idea of the R-Op approach is to execute R scripts, equivalent to native database operations like joins or aggregations, and thereby including the R runtime as part of the database execution plans.

To realize this idea we take advantage of the calculation engine of IMCE and its capability to define data flow graphs (calcModels) describing logical database execution plans. A node in this data flow graph can be, as already introduced in Section 3, any native database operation, but also a custom operation. One of those custom operations is our newly created R operator. Like any other operator of the calcModel, the R operator consumes a number of input objects (e.g. intermediate tables retrieved from previously computed operations or other data sources like a column or row store table) and returns at least one result object. In contrast to a native database operation, a custom operator is not restricted to a static implementation and can be adjusted for each node independently. In the case of the R operator this is done by the R script, which is passed as a string argument and embedded by the operator. Figure 5(a) shows the



(a) sequentially processed R-Op (b) parallel R-Op execution (c) using split-merge logic to parallelize R-Op execution

Figure 5: Examples of calcModels containing R-operators.

minimal calcModel containing an R operator. It consists of a data source node referring to a table and the R-operator introducing the R script, which will be executed based on the data provided by the data source node.

Figure 6 illustrates the internal mechanism used to embed R during the calcModel plan execution. Whenever the plan execution reaches an R node, a separate R runtime is invoked (① and ②) using the *Rserve* package [22]. The input tables of the given node are then passed to the R process using a shared memory transfer mechanism (③ and ⑤), similar to the one we discussed in Subsection 4.1. The R script is passed via TCP/IP (④) again taking advantage of the functionality provided by the *Rserve* package. After the R runtime has completed the script execution, the resulting dataframe(s) is again placed into shared memory (⑥), and the IMCE takes care of the internal data structure conversion (⑦). Since the internal column-oriented data structure used within IMCE for intermediate results is very similar to the vector oriented R dataframe, this can be done with minimal effort.

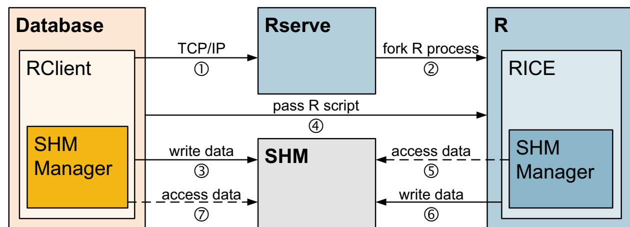


Figure 6: R-Op architecture.

One of the key benefits of having the overall control flow situated on the database side is the fact that the database execution plans are inherently parallel and therefore multiple R runtimes can be trigged to run in parallel without having to worry about parallel execution within a single runtime. For instance, if the included R script also processes data independently on subsets, we can split the data accordingly (see Figure 5(b)) and execute the R script for each subset in parallel without having to change a single line of code in the R script itself. Nevertheless, this kind of parallelization requires the calcModel to be designed accordingly. This implies that even though arbitrary R scripts can be included transparently, the surrounding calcModel needs to be modeled thoughtfully if the integration is to fully utilize the capabilities of the parallelization framework.

Instead of explicitly splitting the data into subsets by using different filter operations, like in Figure 5(b), it is also possible to use the more general and optimized split and merge operation, as shown in Figure 5(c). The split operator is a custom operation, which is specifically designed to divide data in a way that subsequent nodes can be executed in parallel. Respectively, the merge operator is a custom operator, which is designed to combine the results from operations previously executed in parallel. In a simple configuration the split operator is equivalent to a number of filters and the merge operation is equivalent to a number of union or join operations, resulting in an execution plan similar to Figure 5(b).

However, there are also other split-merge pairs possible, for instance a split-merge pair can be used to prepare and evaluate differently configured R script executions processed in parallel. More concretely, the R-operator could be used to train a number of different classification models using different settings defined by the split operator. The merge operator could be used to evaluate the different classification models choosing the one with the highest accuracy. This extended split-merge logic is therefore not restricted to data partitioning and allows the calcModel to be used for a number of *possible world* or *what-if analysis*, similar to the ideas discussed in MCDB [11].

Another closely-related advantage is the fact that integrating R into the database allows any application on top of the database to directly benefit from it. This also includes combining the R-Op and SQL-SHM approach in a way that an external R (application on top of the database) is used to query (with the getDataFrame function) a calcModel including further R runtime executions. This combined approach leverages the parallelization framework provided by the calculation engine including multiple R-Ops with the control flow on the R side.

4.3 Sample use case

With the previously described R integration in place, it is possible for the database user to take full advantage of the capabilities provided by R. This includes a broad spectrum of advanced analytic functions ready to use as well as the ability to adjust the algorithms and to configure parameters for even the most specialized demands.

To illustrate the potential provided by our R integration we are going to demonstrate how R scripts can be combined with the parallelization framework of IMCE in order to implement a highly sophisticated classification algorithm. Before we start to go into the details of the algorithm—the Cascade Support Vector Machine—and the implementation on IMCE, we will first introduce the classical Support Vector Machine (SVM) itself.

4.3.1 Support Vector Machines

The Support Vector Machine is a frequently applied classification algorithm. The goal of classification is to train a model which can be used to distinguish different groups within the data, known as classes. The conceptual idea behind the SVM is to find hyperplanes between the data points dividing them into such classes. The SVM algorithm transforms the data points into a higher dimensional space where such hyperplanes can be found. These hyperplanes are spanned by a small subset of the training data, the so-called support vectors, which are computed by solving an optimization problem with a quadratic target function and linear constraints.

Since R already provides a wide variety of SVM implementations [12], using this functionality as part of an IMCE execution is as simple as passing an SQL query including a few lines of R code (as shown in Script 2).

```
 1. ## definition of the input and output data types
 2. CREATE TABLE TYPE SVM_IN (CLASS INTEGER, ...);
 3. CREATE TABLE TYPE SVM_OUT (CLASS INTEGER, ...);

 4. ## Creates an SQL-script function including the R script
 5. ## for the classification task. The support vectors
 6. ## are returned as dataframe/table.
 7. ## 'input1' is the reference name of the input table
 8. ## 'SVM_IN' describes the schema of the input table
 9. ## 'result' is the reference name of the returned table
10. ## 'SVM_OUT' is expected schema of the returned table
11. CREATE FUNCTION svm(IN input1 SVM_IN, OUT result SVM_OUT)
12. LANGUAGE RLANG AS '''
13.   ## loading SVM functionality
14.   library(kernlab)

15.   ## use dataframe 'input1'
16.   ## for support vector classification
17.   model = ksvm(CLASS ~ ., input1)
18.   ## retrieve support vectors
19.   sv = xmatrix(model)[[1]]

20.   ## return support vectors as dataframe 'result'
21.   result = as.data.frame(sv)
22. ''';

23. ## execute SQL-script function and retrieve result
24. CALLS svm(inputTab, resultTab);
25. SELECT * FROM resultTab;
```

Script 2: SQL script using R for SVM classification

The code shows the SQL-style syntax used by IMCE to set up a calcModel with an R node. In the first block (lines 1–3), two table types are created representing the schema of the actual input and result tables. They are used in the next line (line 11) to set up the calcModel node defining the function interface. The directive `LANGUAGE RLANG` (line 12) defines the type of the script passed to the node. The code is similar to what we discussed in Script 1 in Subsection 4.1: it computes an SVM model (lines 13–17) and sets the support vectors as the result to the return variable (lines 18–21). After defining the calcModel, it is called via the `CALLS` command (line 24): `inputTab`, containing the training dataset, and `resultTab` are physical database tables. After running the calcModel, the final result is obtained by the `SELECT` statement on the resultTab (line 25).

This example illustrates a straightforward way to leverage the SVM implementation provided by R. Although this example already takes full advantage of the R integration itself, it does not use the full potential of IMCE, because the computations will only be done sequentially, similar to Figure 5(a).

4.3.2 Cascade Support Vector Machines

The major problem of the SVM algorithm is that it underlies at least quadratic space and time requirements [1] in regard to its input data. In general, splitting an optimization problem (like the SVM), solving the parts independently and merging the results normally does not lead to the global optimum. Therefore, additional actions must be considered to omit the limitations of the conventional SVM. One approach to parallelize the algorithm was proposed by Graf *et al.* [9]: the Cascade Support Vector Machine.

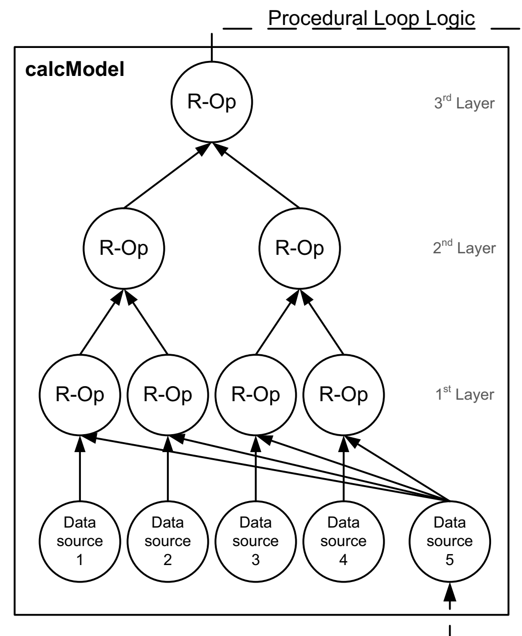


Figure 7: SVM cascade using calcModel with R.

In detail, the training data is split up into several (disjoint) subsets and the SVM algorithm runs separately on each of them. Even though it is not expected that the partial results are equal to the global optimum, it can be assumed that at least some of the support vectors of the whole problem are found in this way (if there is not a serious bias in the initial subsets). By combining the support vectors of different subsets and applying the SVM algorithm again, a good approximation to the global optimum can be found. This is based on the idea that "interior" points of the subsets are likely to be "interior" points of the whole set. The

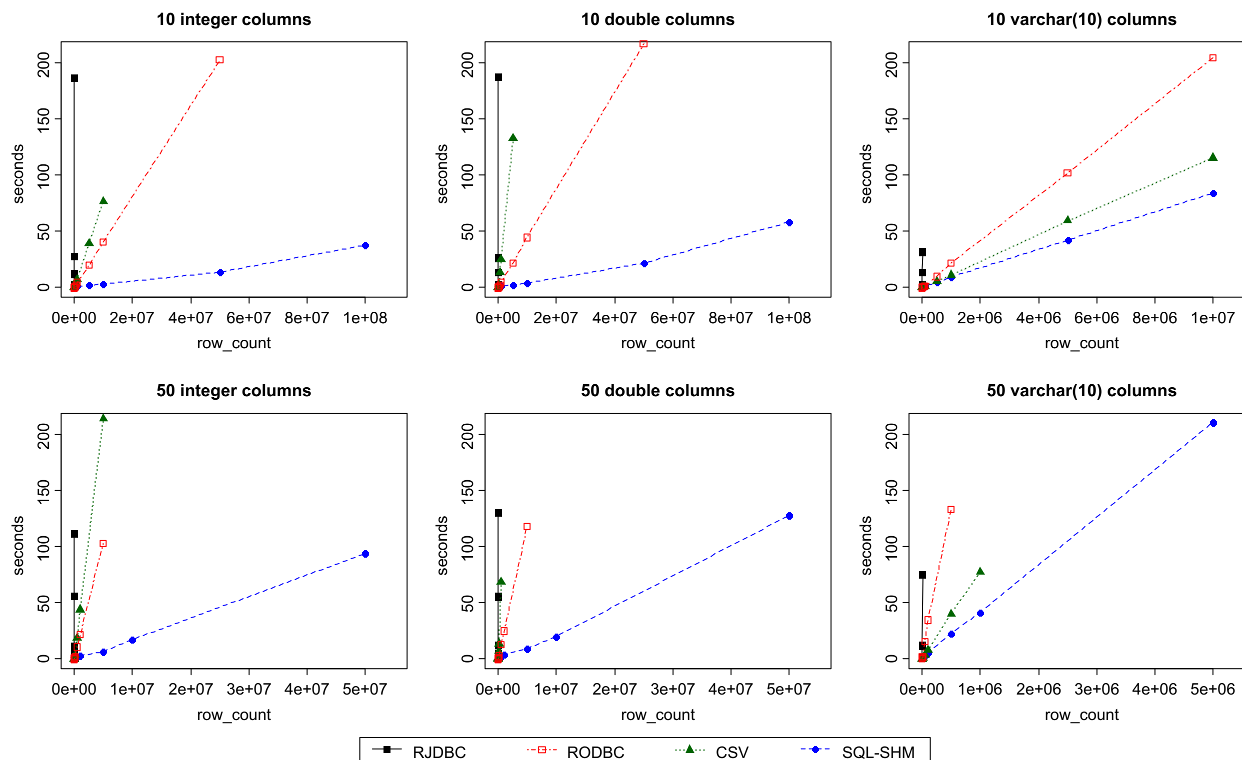


Figure 8: R interface performance for different data types and tables sizes.

Cascade SVM merges the support vectors two-by-two in the style of a binary tree into one single set of support vectors.

In many cases, this approach will return already good results after the first run through the cascade. However, if the global optimum has to be reached, the process must be iterated. For this reason, the result of the first run is fed back into the first layer where it is used for classification of each original training subset (with linear complexity). All incorrectly-classified vectors are united with the support vectors and the algorithm is executed repeatedly as described above. The authors [9] showed that this approach often converges into the global optimum within just 2–5 iterations.

Due to the quadratic behavior of the SVM, the time requirements are decreased enormously: in the first layer the complexity is reduced by a factor of k^2 with k number of nodes, since those nodes can be executed in parallel. In all subsequent layers the nodes only have to handle the resulting support vectors, usually consisting of small subsets of the whole training data, thus the complexity is also quite low.

The Cascade Support Vector Machine can formally be described as a data flow plan, and therefore as a calcModel. A schematic view of this model is shown in Figure 7.

Here, a database table containing the training data is split initially by a series of data sources (1–4) serving as input for the first layer of R-nodes of the cascade. The R operators take them as inputs and compute the support vectors. Each two results are united and passed to the next layer where the SVM algorithm is applied again. This process is done in every layer until the topmost node is reached.

The final set of support vectors is stored by an external logic in the database, accessible through data source node 5. This logic is necessary, since the Cascade SVM requires iterations and calcModels do not allow loops yet. When the cascade is called again and the R-operators in the first layer find a non-empty data source 5 (i.e. the support vectors are fed back into the cascade), their training data subset is classified and checked against the class labels. All vectors that are predicted incorrectly are combined with the support vectors and sent to the next layer where the Cascade SVM algorithm continues as described above.

The external logic calling the cascade terminates as soon as a user-defined criterion is fulfilled, e.g. a fixed number of iterations or a steady set of support vectors is found.

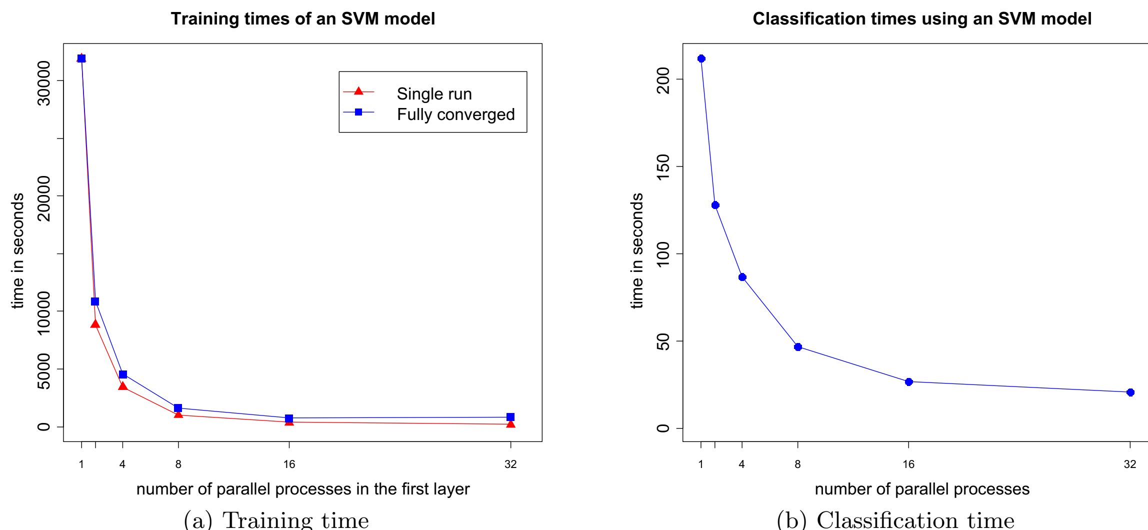
Even though the Cascade Support Vector Machine is a sophisticated algorithm, we had been able to implement it with little effort using our R-Op approach.

5. EVALUATION

In the following section we discuss our evaluations for both the SQL-SHM as well as the R-Op approach. The first part of the evaluation focuses on the shared memory performance from the perspective of our SQL-SHM solution, whereas the second part discusses the performance achieved in the Cascaded Support Vector Machine use case. The hardware used for our evaluation is an Intel(R) Xeon(R) X7560 (4 sockets) with 32 cores 2.27GHz using hyper-threading and 256 GB of main memory. We used R 2.11.1.

5.1 SQL-SHM performance

To evaluate the performance improvement achieved with our SQL-SHM solution, we measured the query time on tables of various sizes and with different datatypes. The datatypes we used were integer, double, and varchar(10), which can be mapped to the three native R datatypes integer (numeric),



(a) Training time (b) Classification time

Figure 9: Cascade SVM on IMCE evaluation.

double (numeric), and character. Figure 8 shows the query performance using the two standard SQL interfaces JDBC, ODBC and our SQL-SHM solution. Additionally, we measured an SQL-triggered CSV export/import of the same tables.

Comparing the results, it is obvious that the *RJDBC* package performance does not scale well. We can rule out that this is caused by the JDBC driver of IMCE, because the same queries triggered by other applications using the same JDBC driver show much better performance. Comparing the *RJDBC* results with the *RODBC* package performance also implies that the additional overhead introduced by the tuple-based data transfer is not the main cause either. We therefore assume the reason to be implementation specific due to the embedding of Java in the R environment.

Taking the 10 integer column table as example, it took over 186 seconds to transfer 50 thousand rows via *RJDBC*. In contrast, *RODBC* needed 203 seconds and our SQL-SHM solution only 13 seconds to transfer 50 million rows. In this particular case, the speedup between *RODBC* and our SQL-SHM solution has a factor of 15.6, but since both interfaces scale linearly the performance benefit achieved increases naturally with the amount of data.

The influence of the datatypes can well be observed for the CSV export/import. While CSV performance is always worse than *RODBC* for the integer and double tables, it is better than *RODBC* for the string tables. This can be explained by the fact that for strings there is no additional datatype conversion necessary during the import.

For the SQL-SHM solution the datatypes also have influence. In particular for string columns, additional overhead is produced using multiple shared memory segments to represent a single column.

5.2 Performance for the parallel use case

To prove the speed enhancement by parallelization using our R-Op approach, we tested the Cascade Support Vector Machine on the *KDD'99* dataset [7]. It consists of about 4.9 million data points of network connections with 41 features grouped into "good" connections and several kinds of attacks. The task was to create a classifier that is able to distinguish between those types. For all tests, we achieved similar accuracy rates as in the *KDD'99* contest.

In our first evaluation, we are going to show the possible speed-up by adding layers to the cascade. For this, we opt out two basic cases that had to be taken into account: a single run of the cascade and a fully converged classifier. The set with 4.85 million data points was trained with the sequential SVM using one single R-Node and the Cascade SVM with 2, 4, 8, 16, and 32 nodes in the first layer. For all tests we used the *kernlab* [13] implementation of the SVM similar to Script 2.

Figure 9(a) shows the training times for a single iteration of the cascade as well as for the fully converged case in comparison to the conventional SVM algorithm with one process. Using two R operators in parallel in the first layer of the cascade achieves already a speed-up of factor 3.6 for a single run and 2.9 for a fully converged model. Extending the number of parallel processes increases this further—when using 32 nodes, the cascade is 173.5 times faster for one single run and achieves a speed-up of factor 43.5 when it is iterated until the global optimum is found.

For all our tests, the cascade converged in the second iteration on the training set, except for the case with 32 parallel nodes: here, three repetitions were neccessary. Therefore, using only 16 nodes was faster for the fully converged case. In general the speed-up is limited by the additional communication overhead between the layers and the number of needed iterations.

With the measured speed-up the SVM algorithm can be applied in real business scenarios, since the training times are reduced from several days/hours to hours/minutes. Especially if a model has to be trained more than once, e.g. because of substantial changes within the data, this method remains practical.

The second part of our evaluation was the classification speed using implicit parallelism as indicated in Figure 5(c). We implemented a split operator distributing the evaluation set into 1 to 32 parallel processes. The R operators classified 290k data points of the evaluation set and returned their computed class labels. We used a previously-calculated SVM model consisting of 5k support vectors. As seen in Figure 9(b), our setup was able to reduce the classification time from 3.5 minutes to just 21 seconds by using 32 nodes simultaneously. Therefore, through parallelization the SVM algorithm becomes practical in real time business applications,

especially those where a fast evaluation of huge amounts of data points is the most important part of the classification.

6. SUMMARY

The growing need to use large amounts of data as the basis for sophisticated business analysis conflicts with the current capabilities of statistical software systems as well as the functions provided by most modern databases.

In this paper we discussed a variety of existing approaches for advanced analytics on large data sets and introduced the two main concepts of our own novel approaches. We thereby outlined the work of an ongoing project in an industrial setup involving an international team located in Walldorf, Germany and Shanghai, China. To date, the two approaches (SQL-SHM and R-Op) are implemented and used internally and we are just about to prepare the open source publication of the R external package *RICE*.

Our first approach (SQL-SHM) significantly reduced the communication overhead between R and the SAP In-Memory Computing Engine (IMCE). Whereas our second approach (R-Op) enabled IMCE to include R scripts as part of the database execution plan and therefore allows us to use multiple R runtimes in parallel processing advanced analytic functionality. This will, as we showed in our evaluations, bridge the gap between the statistical software package and the database system and thereby enable R functionality to be transparently applied in real business scenarios.

7. ACKNOWLEDGMENTS

The authors would like to thank the SAP In-Memory Platform development team and in particular Sebastian Seifert, Christoph Weyerhäuser, Veit Spägele, Rick Liu, Caro Ge and Jianfeng Yan for their numerous contributions to the preparation of this paper and their constant work on the software development.

8. REFERENCES

[1] C. Burges. A tutorial on support vector machines for pattern recognition. *Data mining and knowledge discovery*, 2:121–167, 1998.

[2] J. M. Chambers. *Programming with Data*. Springer Verlag, 1998.

[3] C. T. Chu, S. K. Kim, Y. A. Lin, Y. Yu, G. R. Bradski, A. Y. Ng, and K. Olukotun. Map-Reduce for Machine Learning on Multicore. In *NIPS '06: Proc. of Neural Information Processing Systems*, pages 281–288. MIT Press, 2006.

[4] J. Cohen, B. Dolan, M. Dunlap, J. M. Hellerstein, and C. Welton. MAD skills: new analysis practices for big data. In *VLDB '09: Proc. of the VLDB Endowment*, volume 2, pages 1418–1492. VLDB Endowment, 2009.

[5] S. Das, Y. Sismanis, K. S. Beyer, R. Gemulla, P. J. Haas, and J. McPherson. Ricardo: Integrating R and Hadoop. In *SIGMOD '10: Proc. of the SIGMOD international conference on Management of data*, pages 987–998, New York, NY, USA, 2010.

[6] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI '04: Proc. of the conference on Symposium on Opearting Systems Design & Implementation*, page 10, Berkeley, CA, USA, 2004.

[7] A. Frank and A. Asuncion. UCI machine learning repository. http://archive.ics.uci.edu/ml/, 2010.

[8] F. Färber, B. Jäcksch, C. Lemke, P. Große, and W. Lehner. Hybride Datenbankarchitekturen am Beispiel der neuen SAP In-Memory-Technologie. *Datenbank-Spektrum*, 10:81–92, 2010.

[9] H. Graf, E. Cosatto, L. Bottou, I. Dourdanovic, and V. Vapnik. Parallel support vector machines: The Cascade SVM. *Advances in neural information processing systems*, 17:521–528, 2005.

[10] S. Guha. *Computing environment for the statistical analysis of large and complex data*. PhD thesis, Purdue University, 2010.

[11] R. Jampani, F. Xu, M. Wu, L. Perez, C. Jermaine, and P. Haas. MCDB: a monte carlo approach to managing uncertain data. In *SIGMOD '08: Proc. of the SIGMOD international conference on Management of data*, pages 687–700, 2008.

[12] A. Karatzoglou, D. Meyer, and K. Hornik. Support Vector Machines in R. *Journal of Statistical Software*, 15:1–28, 2006.

[13] A. Karatzoglou, A. Smola, K. Hornik, and A. Zeileis. kernlab – An S4 Package for Kernel Methods in R. *Journal of Statistical Software*, 11:1–20, 2004.

[14] T. Legler, W. Lehner, J. Schaffner, and J. Krüger. Robust Distributed Top-N Frequent Pattern Mining Using the SAP BW Accelerator. In *VLDB '09: Proc. of the VLDB Endowment*, volume 2, pages 1438–1449, 2009.

[15] R. A. Muenchen. *R for SAS and SPSS Users*. Springer, Berlin, 2008.

[16] H. Plattner and A. Zeier. *In-Memory Data Management: An Inflection Point for Enterprise Applications*. Springer, Berlin, 2011.

[17] Revolution Analytics. *RevoScaleR: Getting Started Guide*, July 2010.

[18] N. Samatova. pR: Introduction to Parallel R for Statistical Computing. In *CScADS '09: Proc. of Scientific Data and Analytics for Petascale Computing Workshop*, pages 505–509, 2009.

[19] M. Schmidberger, M. Morgan, D. Eddelbuettel, H. Yu, L. Tierney, and U. Mansmann. State of the art parallel computing with R. *Journal of Statistical Software*, 31:1–27, 2009.

[20] M. Stonebraker, J. Becla, D. Dewitt, K. T. Lim, D. Maier, O. Ratzesberger, and S. Zdonik. Requirements for Science Data Bases and SciDB. In *CIDR '09: Proc. of the conference on Innovative Data Systems Research*, 2009.

[21] R. D. C. Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2010.

[22] S. Urbanek. Rserve - A Fast Way to Provide R Functionality to Applications. In *DSC '03: Proc. of the International Workshop on Distributed Statistical Computing*, 2003.

[23] Y. Zhang, H. Herodotou, and J. Yang. RIOT: I/O Efficient Numerical Computing without SQL. In *CIDR '09: Proc. of the Conference on Innovative Data Systems Research*, 2009.

[24] Y. Zhang, W. Zhang, and J. Yang. I/O-Efficient Statistical Computing with RIOT. In *ICDE '10: Proc. of the IEEE International Conference on Data Engineering*, pages 1157–1160, 2010.