Efficient Data Management and Statistics with Zero-Copy Integration

Jonathan Lajus
ENS Cachan
France
jonathan.lajus@ens-cachan.fr

Hannes Mühleisen
CWI, Amsterdam
The Netherlands
hannes@cwi.nl

ABSTRACT

Statistical analysts have long been struggling with ever-growing data volumes. While specialized data management systems such as relational databases would be able to handle the data, statistical analysis tools are far more convenient to express complex data analyses. An integration of these two classes of systems has the potential to overcome the data management issue while at the same time keeping analysis convenient. However, one must keep a careful eye on implementation overheads such as serialization. In this paper, we propose the in-process integration of data management and analytical tools. Furthermore, we argue that a zero-copy integration is feasible due to the omnipresence of C-style arrays containing native types. We discuss the general concept and present a prototype of this integration based on the columnar relational database MonetDB and the R environment for statistical computing. We evaluate the performance of this prototype in a series of micro-benchmarks of common data management tasks.

1. INTRODUCTION

Researchers and practitioners alike have long been struggling to produce solutions for the grand challenges that appeared in the Big Data era. A single system to solve the challenges arising through large volumes of data that require complex statistical processing to find answers might even be impossible, due to the large gap in requirements: Large-scale data management is based on very efficient processing and memory bandwidth utilization, clever storage layouts and complex query optimization techniques. Unfortunately, running advanced analytical methods using the standard database interfaces is tedious to impossible. On the other hand, statistical processing packages have moved to give statisticians more and more flexibility in expressing their analyses. This however reduces optimization opportunities, since optimization of expressive programming languages is no simple task. In addition, their data handling capabilities are by far not as advanced as those in a database system.

SSDBM '14, June 30 - July 02 2014, Aalborg, Denmark.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2722-0/14/06 ...$15.00.
http://dx.doi.org/10.1145/2618243.2618265

A possible solution would be the integration of two systems that specialize on the respective tasks of data management and statistical analysis. This is our long term goal and has already been shown to be both powerful and practical [11], but a major issue remains: As the amount of data to be transferred between the data management and the statistics software grows larger, the transmission overhead grows. This is a direct consequence of the memory management techniques used on modern computing architectures, where different processes have no direct access to memory held by another process. The very popular choice of socket communication between processes requires expensive serialization and de-serialization. Another possibility is shared memory, but here, both systems need to specifically support this mode.

In this paper, we propose the integration of data management and statistical software in the same process. If the two systems run in one and the same process, data sharing can be as simple as passing pointers around. Nevertheless, the issue of different data representations within process memory by the two systems can still make a conversion process necessary. However, if both systems use the most common form of data representation, C arrays of primitive types, the conversion step might be avoided altogether. These data structures have become very popular due to the high efficiency with which they can be processed by modern hardware architectures. This allows bulk operations on large amounts of data. As an example, we show how two representatives of each class of systems, the MonetDB analytical data management system and the R environment for statistical computing can be integrated in the same process. Furthermore, both systems use C arrays of primitive types as their internal in-memory storage model. Hence, we also show how zero-copy integration can be achieved between these two systems, thereby extending the capabilities of database management systems to new application areas.

The main research question for this work was whether it is possible to integrate statistical and data management systems in a way that does not require any data conversion or copying. Such an integration would promise unparalleled performance due to the highly specific optimization opportunities on both sides; the data management side can focus on performance, while the analytical side can put emphasis on user-friendly and concise expression of statistical analyses. In this paper, we contribute a proof-of-concept integration and a micro-benchmark evaluation aimed at showing the potential of such an integration. The remainder of this paper is organized as follows: In the following Section 2, we give an extensive overview over related work. Section 3 then discusses the generic challenges for in-process data sharing. We describe our proof-of-concept integration of MonetDB and R in Section 4. Finally, we present the results of the experiments we have performed using our integration in Section 5 and conclude in Section 6.

2. RELATED WORK

The improvement of large-scale data management within statistical software packages has already enjoyed much attention. There are several distinct classes of approaches: We have identified socket-based interfaces to various databases, extensions of the statistical software itself, embeddings of differing directions, solutions for a subset of the problem and distributed data management frameworks.

The perhaps most prolific approach is to use a standardized database interface such as JDBC or ODBC to access data from the statistical environment. This is for example supported in the statistical packages R, SPSS and STATA. The basic procedure is to establish a database connection through these standardized interfaces, and then to send SQL queries to the relational database and retrieve result sets into the statistical environment. This approach has two major drawbacks: First, users have to make themselves familiar with SQL and the relational model in order to retrieve data. Second, data transferred between the process space of the database and the statistical software has to be copied multiple times and likely to be serialized and parsed in the process. As we have will see in our evaluation in Section 5, this severely limits the amount of data that can be transferred. This is especially critical as extending a SQL interface with new functionality is much to ask from a statistical analyst, and she will therefore tend to run her analysis within the statistical environment, exacerbating the need for data transfer. In the R ecosystem, database-specific connectors exist for many commercial as well as Open-Source databases, e.g. `RPostgreSQL` and `RORACLE` [5, 12].

In the class of extensions of the statistical environment itself, the general trend is to provide a faster implementation of particularly expensive data management operations. For example, the `data.table` package for R vastly improves the performance of data loading from raw CSV files, but also for join and grouping operations [6]. The extension benefits from a deep integration into R and strives for compatibility with existing data management infrastructure. This makes it possible to enhance the performance of analytical programs by adding the extension. However, the performance is achieved by re-implementing classical relational algebra operators. It is clear that the implementation of these operators has enjoyed much more attention in a system solely dependent on their performance. We will address this assumption with experiments in Section 5. Also, the dataset being processed has to fit into memory, which represents a serious constraint. A very similar approach has been followed by the `ff` package [3]. However, the compatibility with the existing data management infrastructure was dropped. Hence, analysis programs will have to be rewritten in order to benefit from the optimized data management provided by the extension. Nevertheless, the extension also provides functionality absent in the original infrastructure.

We have already argued for the in-process embedding of data management and the statistics environment. This approach has two possible directions, one that embeds the statistics environment within the database, so that analytical functions become available from the database (SQL) interface. The `REmbeddedPostgres` is an example of this group, where the R environment was embedded into the PostgreSQL database, such that functions operating on single records, aggregates and triggers can be implemented using R and its contributed libraries [10]. A more recent example is the embedding of R into the Oracle commercial database [8]. We however propose the inverse embedding direction here, where the database is embedded into the statistics environment. The most widely used representative of this class is `RSQLite`, which is also subject of our evaluation. SQLite is a compact implementation of a relational database that is designed to be embedded in other programs. Hence, embedding it in R faced fewer challenges than those described in Section 4. All those approaches share a common limitation: The data retrieved from the database tables has to be converted from the database-internal format into the R format. This requires at least one copying and processing step. Contrary, this overhead is completely unnecessary in the approach we propose. The most comprehensive integration presented so far is that of the SAP HANA database and R. Here, R scripts are executed as part of the database query execution plan [7]. Also, shared memory is used to transfer data and query results between the database and R. While not requiring serialization, conversion and copying still has to take place. In a distributed environment, R has also been embedded into MySQL with an additional CORBA-based coordination layer for coordination with a user session [4].

Also related are database-supported solutions for specific computational issues in statistical computations. Large-scale calculations on arrays or matrices are an example for such a specific issue. `RIOT` is a proposed approach to achieve transparent IO efficiency which also maps an R problem into the SQL realm [18, 19]. The authors also discuss the fundamental issues behind achieving the sought-after IO transparency. Large-scale array processing from within the statistical environment through third-party data management systems has also been described for distributed processing, for example in the `Presto` system, where functions can be executed in parallel over a single logical array [17]. Interestingly, they also note the importance of the zero-copy mechanism for efficient data sharing between the statistical and the data management system. However, having built their data management from scratch, they were able to avoid some of the pitfalls we have encountered in Section 4.

3. SAME-PROCESS DATA SHARING

From a conceptual point of view, the integration of two independent systems within a single process is simple: Multithreading facilities provided by the operating systems allow a large number of independent execution contexts, hence systems that previously were running in a single process can be moved into independent executions threads within a single process. Access to shared resources such as the network system, the file system etc. is still managed by calls to the operating system, which also provides synchronization and isolation between the threads. In practice however, several challenges do exist:

First of all, the placement of two systems in the same address space usually requires recompilation, unless special relocatable object files are available. Even if library versions are available, they are unlikely to export all the necessary operations. This means that the source code of both systems has to be available. Then, a new binary is produced by compiling both systems together. However, it is far from certain that first, both systems use a compatible programming environment and second, their programming code actually can be compiled together. For example, two separate C programs could both contain an `error()` function to handle program errors. If no special care is taken, the existence of the symbol in both code bases precludes compilation. There are static source code analysis methods to catch some of these name clashes (given sane coding practices), but integration problems will still appear in the linking phase. Overall, the effort to make the two systems compile and link together is highly dependent on the implementation of both systems.

Once both systems are successfully linked and started, the actual passing of data has to be considered. The method of choice would be the calling of a function in the data management system by the analytics software to request data. Once the data is available, the data management system in turn calls a method in the analytics package to notify it of the data being available. Alternatively, the data management system can be initialized, the query run, and shut down again from the context of the calling routine in the analytics package. If the desired data is only available within a function call, and not as a result, it would be possible to modify the code in such a way that the data management solution would be forced to wait on a semaphore until the data is no longer needed. The form in which data is passed is by reference. This means that rather than putting the entire data on the stack as a function parameter, the memory address (which is valid for both systems, since they run in the same address space) is passed. This allows the analytics system to simply de-reference the address pointer and access the data there. Another issue remains however. Low-level memory allocation requires the explicit `free()`-ing of memory that is not used any more. Yet, the data management system has no way of knowing whether the analytical system still requires access to the shared data. Hence, the task of freeing the memory that contains the shared data has to be taken over by the analytical system or by way of an additional notification mechanism in the data management part.

Figure 1 shows the described communication pattern. Within the continuous space of virtual memory, both the statistical tool as well as the database are loaded. The statistical tool initiates the interaction by sending a SQL query by means of function call to the database (1). The database then starts processing the query and produces a result set in a compatible binary format (2). Finally (and crucially), the data is not returned in serialized form of or by copying, but instead by only passing a pointer (here: `0x10000000`) to the starting address of the query response within memory to the statistics application (3). The statistics application can now directly access the data in memory. Of course, the figure oversimplifies the process, as every result set column would have their independent starting address, all of which would have to be passed over. In addition, the receiving program needs to know the number of elements in the arrays, and the type of each entry in bytes in order to read the data. From the type, the length of each entry can be determined.

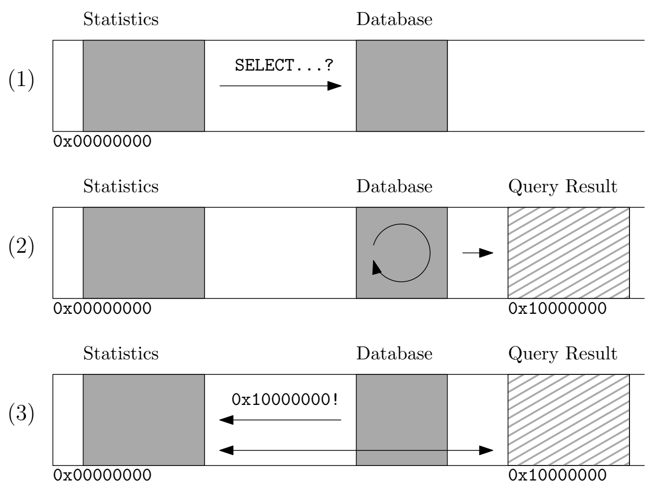


Figure 1: Communication Flow within Process Space

Of course, by running both systems on a single machine, otherwise common issues such as differing endianness are avoided. However, different programing languages can use different binary encoding for their values. Nevertheless, there is a (small) set of data types with a almost universally agreed-on encoding. Fortunately, those are also the ones with most relevance to statistical analyses: integers of various lengths (`short`,`int`,`long`), and standardized floating point numbers (`float`,`double`) [2, 1]. The compatibility in encoding even between programming environments is due to CPU support for these data types, such that operations on them can be calculated by the CPU in a single instruction. Hence, regardless of the system used, there is a good chance that the basic data is encoded in one of these types. The cited standards also include conventions for special values, for example infinity or the special value `NaN` (“Not a Number”) for divisions by zero.

But one complication still exists: Both relational databases as well as statistical analysis software have a notion for an unknown value, or missing data (`NA`/`NULL`). The way this is encoded on an in-memory level is typically through additional values that represent the non-existing value. These are not included in the number representation standards, therefore every system has its own convention about which specific value is regarded as a missing value. Of course, it is very likely that different values are used. Hence, it would be beneficial to filter those values out on the data management side before passing the pointer. There might however be a shortcut: For numerical values, the `NA` value is usually represented by taking a magic value, for example the smallest or highest possible legal value for the C type. Since this value is hopefully defined at a single location in both systems, there could be a small modification to change this value to be compatible and recompile.

Another issue concerns more complicated data structures such as character strings or complex objects. While the representation of single strings as null-terminated character arrays is universally used, we are focusing on arrays of values. There are several alternatives to represent arrays of strings. For example, an array of pointers into separate null-terminated arrays for each entry could be used to store a string array, possible re-using memory addresses for duplicate elimination. Another option would be to use a larger string vector that contains several null-terminated strings and an offset array to access each entry. Another possibility yet would be a non-traditional string layout, e.g. using a start pointer/length structure. Many more representations are possible, and it is therefore unrealistic to expect similar in-memory representations of string vectors. Similar issues exist for other complex data types, especially as programming language borders are crossed. Fortunately, the most common statistical calculations involve variables that are either already numerical or can easily be mapped to numeric arrays. We will focus on those types in the following.

While we can now reasonably (and possibly only partially) assume compatible encoding of single values, we are ultimately interested in bulk processing of large amounts of data. Both statistical and database systems are built on a tabular data abstraction, which is equal to or closely related to the relational model as a list of named columns of various types. If a row-oriented storage layout is used, the different data types force a large amount of additional structuring bits to be added to make it clear where the row starts, where the fields start, where the fields end, and where the row ends. This can be best compared with CSV-like formats, only on a binary level. However, it has been shown, that a columnar or decomposed storage model allows far higher efficiency with regards to analytical processing [9, 16].

One of the reasons why the columnar data representation has been found superior in those use cases is the more "natural" mapping of single columns to environment-provided data structures. In particular, the concept of an indexed array of values of a single type is a basic feature of almost all programming environments, and expensive function calls are not as common. Furthermore, there is an obvious and widespread storage layout of these arrays within the conceptually continuous address space such that array elements are simply stored back-to-back without additional dividing bits [2]. Dividers are not necessary since the width of the single element type in number of bytes is known in advance. Of course, this only applies to fixed-length types. Again, these types are fortunately the most commonly used ones in statistics.

Finally, we have to discuss concurrent access to the query result and the persistence of changes to the data. If data are directly being accessed using a pointer there is no general way of prohibiting writes to these memory locations. The major issue here is that it is not clear whether the data for which the pointer is passed is part of the permanent data.In the case of main memory databases, the contents of main memory are the base data that has been loaded into the database. If for example an entire table or a subset of it is requested, there is no reason why the database should copy the data in order to create a result set. Therefore, if a pointer to the base data is given to the analytical package, modifications can potentially change (or even corrupt!) the data. This is especially true in the aforementioned case of main memory databases. Whether it is safe to allow write access to the passed data therefore completely depends on the implementation of the data management solution. A compromise to protect the database from corruption would be to rely on a low-level copy-on-write mechanism of the operating system for memory-mapped data or a raw copy of in memory. These copies may affect only small selections of slices of tables (optimized operations that give a read-only view on the table). As the database system chooses at which point it will use one allocation method or the other, a trade-off has to be found between the overhead of accessing the virtual memory and the overhead of punctual copy of views if we want to ensure the integrity of the database. Then all the write operations become local to the statistical environment. Another option is to use additional synchronization mechanisms to perform a safe modification of the data. However, this would require extensive modification of both systems. It is thus safest if a read-only access pattern is assumed for now, even there are no possibilities of enforcing this once the pointer has been passed over.

4. EMBEDDING MONETDB AND R

From the previous section, we could see that an in-process integration between a database and a statistical software package is theoretically feasible. However, we have also seen numerous potential issues, that could make zero-copy data integration impossible, because at least a single-pass conversion would be required. To continue our studies on the matter, we have chosen to attempt the described integration using two representatives of both system classes. For the data management part, we have selected MonetDB, an open source columnar relational database that is focused on analytical queries [9][1]. For the part of statistical analysis, we chose the R environment for statistical computing [14]. The advantage of both systems is their focus on the issue at hand as well as the free availability of their source code. As described above, the availability of the source code is a crucial precondition for in-process integration.

It should be noted that we have embedded MonetDB into R, not the other way around and this for two reasons. First, the changes described below are less invasive. R-core has only been slightly modified and the databases produced or read by the embedded version of MonetDB are fully compatible with the default version. Second, the use of MonetDB for data management is fully transparent to R users. The two systems communicate using simple function calls. The system is started up by first starting R. Whenever the first database operation is required, the database system is initialized, the data is loaded, the query is run, and the result set retrieved. As the R process is terminated, so is MonetDB. Of course, this will add a small initialization overhead to the execution time of the first query.

In MonetDB, relational tables are stored as a set of "Binary Association Tables" (BATs). Each table thus consists of a set of BATs. Furthermore, database operators such as selections, projections, joins etc are operating on BATs and also produce BATs as output. Each BAT consists of two (potentially large) arrays, referred to as "head" and "tail".

[1]MonetDB is developed at the CWI Database Architectures group, where this research was also performed.

The head contains the row indices of the values stored in the BAT, and the tail the actual values of varying type. If the head array is continuous, it is omitted in favor of a starting offset in order to save memory. In addition, BATs have a header structure that contains meta data about the data stored, such as length, type etc. Within the database, BATs are passed around using a pointer to the header structure, which in turn contains pointers to the head (if applicable) and to the tail. Figure 2 shows this data structure: The meta data is held in independent memory areas that contain pointers to the actual data in simple arrays. The database also keeps track of the number of references to a single BAT, and removes them from memory if that count drops to zero. Of course, client programs expect a table as a response to a SQL query. Hence, in the last step of query processing, the resulting BATs are recombined to produce a relational table by joining the columns together on their row indices. Then, the result is serialized and sent to the client. MonetDB also contains a pluggable optimizer architecture, where different algorithms can be enabled to adapt the query execution plan.

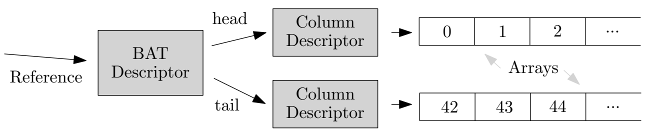


Figure 2: MonetDB Binary Association Table (BAT)

On the other side, R uses a universal data type called "Symbolic Expression" (SEXP) for all values [15]. There are a number of subtypes that hold specific kinds of data. Here, we are only interested in the vector types, for example INTSXP, REALSXP to hold integer and floating point values, respectively. Due to the vectorized nature of R, no concept of single values exists, these are represented as vectors of length one. Every vector-typed SEXP is a collection of values, which is coincidentally stored in an array of primitive types. Each SEXP contains a header structure which describes the type of the data in the value. In the case of an INTSXP, the header is immediately followed by a primitive value array as shown in Figure 3.

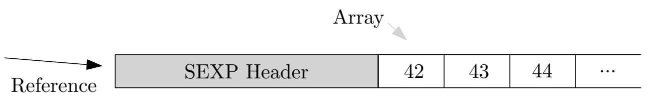


Figure 3: R Symbolic Expression (INTSXP)

It is this similarity in storage layout that makes a zero-copy integration possible. R also supports an "external pointer" data type, which at first glance would be perfect to store the BAT pointer "leaked" from MonetDB. However, built-in R operations cannot operate on this type of objects, so the preferred solution was to "dress up" a MonetDB BAT as an R SEXP. This way, we can move data into R and use both built-in and contributed analysis methods on it, without the analysis methods noticing that they actually operate on a BAT. Otherwise, at least a single copy/conversion step would be required. To perform this underhanded trick, we have extended R's memory management system with a way to construct a SEXP from a given vector without reallocating.

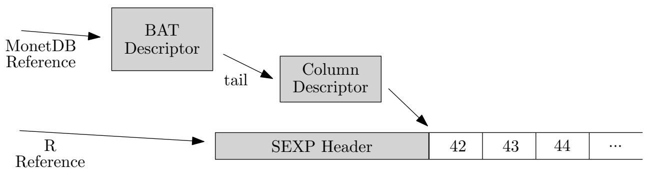


Figure 4: Dual-Role BAT/SEXP

We have added a small "optimizer" to MonetDB, which changes every query execution plan insofar that it removes the final joining of the result columns together as well as the serialization of the resulting table onto the client socket. Instead, a new function inside the R realm is called. This function gets the reference to the result BAT as well as the SQL name and type of the result column passed as parameters. From this point, we can hand over to a in-place allocation function. A major difficulty here was that R expects the SEXP header directly preceding the data array. Since it is impossible to allocate space in front of a pointer without reallocating and copying, we have also modified MonetDB's memory management to allocate space to fit the SEXP header in front of every MonetDB tail array. These changes were done in such a way that they do not interfere with the "normal" database operations. The exact size of the SEXP header is dependent on the hardware architecture, but can be calculated by `sizeof(SEXPREC_ALIGN)`. For example, on a Intel 64 Bit platform, the header requires 40 bytes of memory. Therefore, we offset the pointer into this area by the space the SEXP header requires, and let MonetDB continue. This way, as R gets the pointer, it can safely decrement it by the size of the SEXP header and write the header in front of the data array. After this "redressing" of the BAT is finished, the resulting SEXP is indistinguishable (except for the attached finalizer) from native R data vectors. This state with the native array being both part of a BAT as well as a SEXP is shown in Figure 4. Now, all R-internal operations on data vectors such as filtering, calculations etc. are supported. Furthermore, the vast amount of contributed R packages that operates on vectors can also be used. As discussed in the previous section, it is also possible to handle `NA` values by changing the "magic" values for the various numerical types to the same number.

While the data is now available as an ordinary R SEXP containing for example integer data, freeing the memory after R is finished processing the data is another concern. As mentioned, R has a garbage collector that will destroy non-referenced objects and free the associated memory region. However, we could not use this finalizer, as a BAT consists of more than the tail array. R does support natively custom finalizer functions for objects, but unfortunately only for the objects of the aforementioned "external pointer" types. Hence, we had to make a slight change in the R memory manager. This change only prevents the garbage collector from executing a low-level `free` on specially marked SEXP. Furthermore, we have found a method of attaching a custom finalizer to this object by referencing an external pointer object (mentioned above) as an attribute of the dual-use BAT/-SEXP. Once an object is garbage-collected, its attributes will lose their reference and will also be garbage-collected, which will lead to the call of the custom finalizer in the next collection iteration. This process allows a safe destruction of the SEXP created (without reallocation) using our extension in R's memory management system mentioned above. In our case, the finalizer just decrements the MonetDB reference counter for the affected BAT and thus leads to the BAT being destroyed (if required). Again, these changes do not affect other parts of R, not even the collection phase of the garbage collector, or third-party extension packages at all.

From a user's perspective, we wrap the embedded database engine into a function call with the SQL query as the parameter and a R `data.frame` structure as return value, compatible to the standardized R database interface [13]. In addition, we re-use the deep database integration layer for R we have presented in our previous work [11]. There, we have shown how R users can interact with data stored inside a relational database in an almost native way. We overloaded R-internal data management operations to modify an underlying SQL query. This SQL query was then sent to the database, the query processed, and the result set was delivered back to R, where it is converted into the R type to hold tabular data, the `data.frame`. Previously, the connection between the MonetDB and R used a TCP socket. We were able to re-use and update this work for the embedded version and the simple R code below demonstrates the ease of use of the user interface. No actual data has to be copied or modified at all, the only data modifications regard the addition of meta data and data organization structures. Still, users are able to use the embedded database without realizing that they do. This way, we gain both high performance and SQL transparency.

Listing 1: Returns a `data.frame`, result of the query: SELECT c2 FROM t1 WHERE (c1 > 42)

```
c <- dbConnect(MonetinR(),
        "/absolute/path/to/database")
mf <- monetinr.frame(c, "t1")
subset(mf, c1 > 42)$c2
```

With regards to the question of read and write sharing, MonetDB uses memory-mapped files to store both base data as well as most intermediate results. This leads to two different kinds of behaviour with regard to the persistence of changes to the data. If the query result is a part of the base data column, changes to it will be written to disk by the operating system. On the other hand, if data was calculated as part of the query execution, it will be not be made persistent. This inconsistency makes it difficult to officially support writing to the shared data objects. However, there is not much we can do to prevent writing, since we have deliberately no control over data access to maximize compatibility. Nevertheless, writing to the vector on the R side should be avoided.

5. EXPERIMENTAL RESULTS

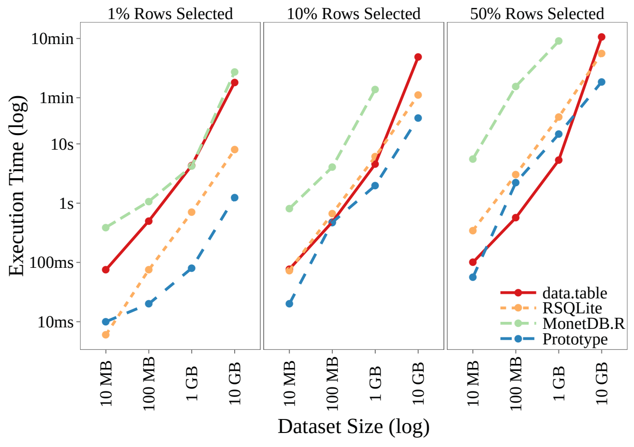
Since we were unable to find any R data access benchmark, we have chosen to perform a series of micro-benchmarks based on the most common data access tasks to evaluate the potential benefits of the same-process integration of R and MonetDB outlined above. We have chosen three competitive large-scale data management solutions for R to be compared with our proof-of-concept prototype:

- `data.table`, a R extension specifically designed for fast data access
- `MonetDB.R`, the socket-based integration of MonetDB and R presented in our previous work [11]
- `RSQLite`, an R integration of the embedded SQL database SQLite.

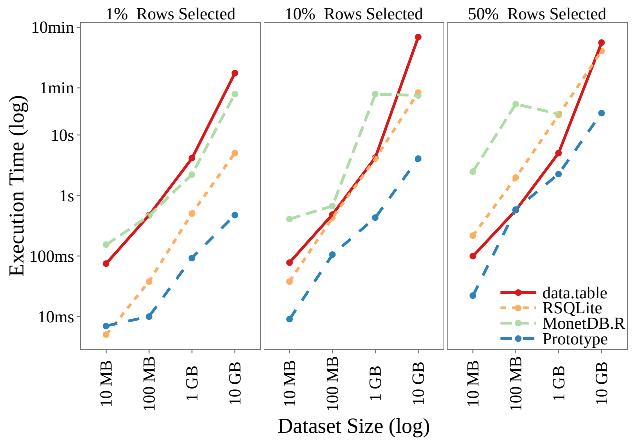
From comparing these systems, we expect to see the impact of highly optimized database systems to data access performance (especially comparing `data.table` with the relational systems) as well as the advantages of using column stores (Prototype and `MonetDB.R` against `RSQLite`) as well as the advantages of in-process integration (`MonetDB.R` against prototype).

The four implementations were compared by testing four basic data management features. First, we tested the selection of a subset of the rows from a table. Since the size of the selection result is expected to have an impact on performance, we have varied the selectivity of the selection between 1, 10 and 50 % of the data. We expect that handling large results will be very difficult for the socket-based solutions. We have also tested both selection and projection of a subset of rows. Only a subset of the columns and only a subset of the rows in the table were requested as result set. Here, column-based solutions are expected to perform better than their row-based counterparts. Another feature tested is aggregation, or grouping in database terminology. Grouping tends to become difficult as the amount of groups to be aggregated becomes large. Hence, we have also varied the group size in 1, 500 and a variable group size that was equivalent to 10 % of the rows in the data set. Here, the larger result sets will be difficult to handle for the socket-based solution, but it is unclear how the other contenders will fare. Finally, we have tested another class of operations, the join (or "merge" in R terms) of two tables by a shared attribute. Depending on the algorithm used, the join performance is heavily influenced by the relative sizes of the two tables. Hence, we have repeated this experiment using a second table with both 1 and 10 % of the size of the first table. All these operations are common to many data analysis tasks, but are also common to relational queries.

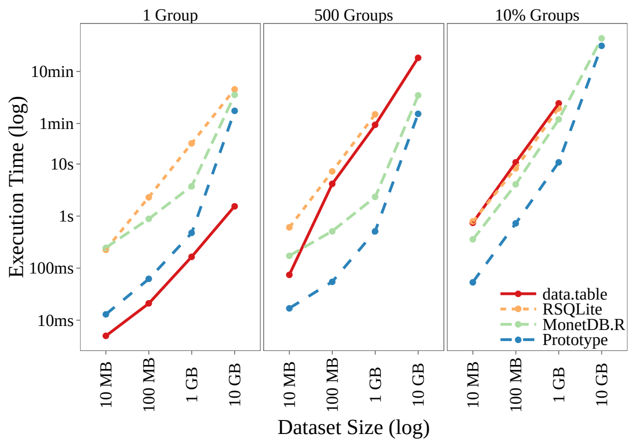
Furthermore, we have repeated all of those experiments on four data sets of increasing size, 10 MB, 100 MB, 1 GB and 10 GB. These data set sizes were chosen to show the limits of the tested systems, and of course their capability to handle large amounts of data. We suspect that "lightweight" solutions such as `RSQLite` will be unable to handle the larger data sizes particularly well. All data consisted of integer columns with random values that were auto-generated as required. For all experiments, we have collected the wall clock time that the systems under test required to complete, since this is the most obvious measure for user-facing systems. Single experiment runs were limited to one hour of execution time. All tests were run on a standard desktop computer with 16 GB of main memory and a 3.4 Ghz Intel i7 CPU. The operating system was Fedora Linux 18. The three systems we have compared our approach to are openly available. In the interest of repeatability, we have also published our slightly modified R version and the modified and reorganized



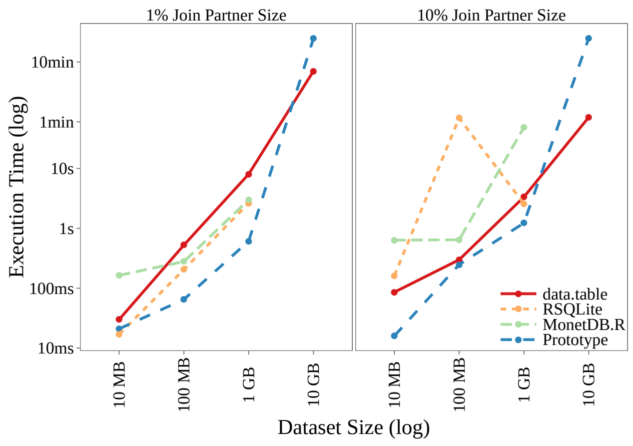
(a) Selection



(b) Selection and Projection



(c) Grouping and Aggregation



(d) Joining of two tables

Figure 5: Experimental Results

MonetDB database[2]. In the latter case, reorganization of the folder structure and the build process was necessary to fit R's package structure. However, this allows us also to compile and install the entire database the same way as any other R package.

On to the results of the evaluation. All plots in this section are structured as follows: Results for each of the different configurations are plotted in their own box, with the particular configuration noted on top. Within each box, the horizontal axis denotes the dataset size (on a logarithmic scale) and the vertical axis describes the response time (also on a logarithmic scale). For example, the configuration of the selection experiment is the percentage of rows retrieved from the data set. If the tested system took longer than the timeout of one hour, the measurement is omitted altogether. All reported timings are cold runs, that is, R and (if applicable) the external database were restarted before every test. The full results in absolute numbers are also given in Appendix A.

[2]`https://github.com/lajus`

Figure 5(a) shows the result of the first experiment, where the very basic transfer of increasing number of rows into R was tested. We can see a very consistent pattern over all data sets. Most notably, the largest result set was retrieved by our prototype in around 110 s, with every other solution taking at least two times more. However, the result set with 50 % of the data being retrieved showed `data.table`.to be faster for the first three data sets. We attribute this to the row recombination overhead in MonetDB. Also, a curious pattern repeats itself over the first three data sets, where the performance of `data.table` shows to be rather independent of the result set size. This might be due to its lazy loading approach, where the actual data file is not loaded until accessed. Since the data loading cost is constant, this might explain this behavior. When comparing the `data.table` observations to the closest running mate, `RSQLite`, we see that 10 % of the data being transferred is the threshold where `RSQLite` becomes slower than `data.table`. This can also be explained by the aforementioned lazy bulk loading, which favors large amounts of data retrieved, which in addition are then already available within the R address space and in a compatible format. Both these advantages are not present in `RSQLite`. Furthermore, we can observe timeouts for `MonetDB.R` for the two largest result sets. This comes as no surprise, when comparing and extrapolating its behavior with the previous data sets. `MonetDB.R` was just able to create and transfer the largest result set for the 1 GB data set, and the 10 % of the 10 GB dataset is twice as large. Another important result is the overall slow performance of `MonetDB.R`, which uses socket communication and serialization. The large amount of overhead can directly seen in the plot. This validates our motivation for this research, as we have suspected that the socket communication overhead might be responsible for a large amount of total response time.

A very similar picture can be seen in Figure 5(b), where the results of the selection/projection experiments are plotted. We have predicted that the column-based solutions will show an advantage here, and indeed the timings for our prototype are less than in the previous experiment. While `MonetDB.R` does in theory also profit from this the high socket overhead drowns this advantage, except for very large result sets. This comes to no surprise, as the socket overhead is high but constant, and less data will eventually show an impact. Also, we can see how the additional effort for projection increases the difference between the row- and column-based solutions, in particular our Prototype and `RSQLite`.

For the grouping experiments, the timings are plotted in Figure 5(c). The group size of one has only a single value as a result set. Hence, the socket overhead should be less. We can see exactly this effect by comparing the performance of `MonetDB.R` to the other systems. Surprisingly, we also see our Prototype being slightly slower than `data.table` for the single-value result sets. This is most likely due to the fact that the single group precludes multi-threaded execution of the grouping in MonetDB. Next, we can also observe very similar performance for `RSQLite` and `data.table`, for the larger amounts of groups in the first three data sets. This hints at a very comparable implementation of the grouping in these two systems. Still, the highly optimized code in MonetDB was able to outperform both for these experiments. For the largest data set, we can see how only MonetDB with both integration methods was able to complete the largest aggregation. Here, we can see the advantage of both a highly-optimized relational database as well as the in-process integration.

Finally, Figure 5(d) shows the timings for the joining of two data sets. Joins are a core requirement and performance bottleneck for relational databases, therefore it is little surprising that our prototype outperformed all other systems for all data set but the largest. Again, the socket overhead prevented `MonetDB.R` from achieving a comparable performance. Surprisingly, `data.table` shows a higher performance for the larger join partner size. The reason for this discrepancy might be the selection of another join algorithm (e.g. moving from a hash join to a inner loop join) for the larger join partner. Again, we observe timeouts for `MonetDB.R` and `RSQLite` on the largest data set.

Overall, we can observe vast performance benefits for our prototype, especially when comparing it with its socket-based competitor, which otherwise is very similar. In particular, handling large amounts of data and corresponding results sets in particular was one of its strengths. We were surprised and impressed by the high performance shown by the `data.table` package, which was able to compete with our prototype and beat `RSQLite` on most occasions. We also confirm our expectation that the socket communication is a major hindrance when large result sets are communicated between systems. Also, the results for the more complex operations such as groups and joins showed a clear advantage for the columnar relational database.

6. CONCLUDING REMARKS

In this paper, we have tried to push the integration between statistical software packages and data management tools as far as possible. To the best of our knowledge, same-process zero-copy data sharing is as far this integration can possibly go. The potential benefits for large-scale data analyses are obvious and numerous. From our discussion of previous work, it became clear that no previous system supports this level of integration. From analyzing data storage inside virtual memory, we have argued that C-style arrays of primitive types are likely to exist on the bottom layer of many software solutions dealing with data. By coercing the data management system into sharing the memory reference to this low-level data structure and tricking the statistical software into regarding this data structure as its own, an integration of unparalleled performance would be possible, despite some remaining issues such as undefined values and write access to the shared data.

To investigate this point, we then performed a prototypical integration as described using the columnar relational database MonetDB and the R environment for statistical computing. We were able to masquerade low-level arrays of primitive values in such a way that both a MonetDB and R regarded them as their internal data structure. Our experiments have shown that this combination can vastly outperform other approaches, especially once a significant amount of data has to be transferred between systems. Only then the overheads for copying and data conversion become dominant.

This insight already hints at the next challenge: Since the shared data is indistinguishable from native objects, secondary data management operations will also be performed by the statistics package without explicit intervention. However, the integration is based on the assumption that the system best suited for a particular task should execute this task. However, this would require to pass the possibly modified data back to the database, before other queries can be run. While certainly possible, further modification of both systems would be required. Also, the class of iterative data analysis algorithms could require a very large number of round-trips between the two systems. However, we can repeat our zero-copy method for the other direction of sharing data as well. Here, we would create a temporary table and create the BAT/Column descriptors accordingly. Then, the complete round-trip between database and statistics environment is possible, and since no actual data is being copied, the overhead for each iteration is constant.

With regards to the integration of statistical software such as R into other database management systems than MonetDB, we found that there are two main strategies: First, if the low-level data representation for numerical arrays is based on C arrays of primitive numeric types, the problem is reduced to properly manage memory and practical engineering to allow both programs to run in the same address space. Contrary, if the in-memory representations of data are conceptionally different, for example as it is the case between R and any row-based database management system, each access to a vector from the statistical side has to be mapped to a function call that operates on the alien data structure. The host program then would need to implement a layer of abstraction for the low-level vector accesses. This has profound implications on performance due to the overheads created by the data conversion and function calls for every value. Furthermore, integrating such an abstraction layer would require a massive rewrite of R in particular, and break many contributed packages which rely on the memory layout and direct pointer-based access. Due to these issues, such an integration would be all but lightweight.

From a practicality standpoint, it might be more feasible to invest in a single copying/conversion step, while still running both processes in the same virtual memory address space. This way, the missing value encoding and the read/write problem could be avoided. Operating systems also already use copy-on-write semantics if processes fork themselves, but this functionality is not yet exported to user-space programs. Of course, this conversion would have to be possible in both directions, such that the analytical environment could pass data back to the database. In both cases, a classical optimization problem arises, where two systems can produce result-equivalent results, but not under the same timing parameters. The decision of where a operation should be run can be very simple if data sharing is free, but as we have seen, this might not be very practical. This then opens up a whole new area of research problems, although one where previous work from related areas, e.g. in distributed databases potentially could be reused.

Considering the applicability and possible generalization of our work, we propose to pass pointers to C-style arrays containing native types between statistical environment and data management solutions. Obviously, this is only possible if both systems use this data representation. However, due to the properties of modern hardware, this representation can avoid cache misses and function call overhead and is able to saturate modern CPUs. Due to this reasons, the representation of data in this manner has attracted much attention in recent years and it is unlikely the differences in memory access that favor columnar data processing vanish anytime soon. However, statistical environments are far less concerned with raw performance, here, user convenience and a rich programming interface are key. Therefore, it is less likely to find the array data representation in this context. That being said, R's massive gain in popularity in recent years have make it the de-facto statistical environment. As we have shown, R does use this array-style data representation. Our work therefore applies to both the integration of R into columnar data management system as well as to the more generic integration of systems that benefit from sharing large amounts of data that both use equivalent low-level data representations.

Finally, we come back to our research question of whether it is possible to integrate statistical and data management systems in a way that does not require any data conversion or copying. From the concepts and results we have presented in the paper, we can conclude that this is certainly possible, although only under some restrictions of varying seriousness. We argue that our integration was prototypical and representative of the issues that arise should such an integration be attempted.

Acknowledgments
We would like to thank Thomas Lumley of the R core team and Sjoerd Mullender from the MonetDB team for their invaluable help. Also, we thank the anonymous reviewers for their insightful comments. H. Mühleisen is supported by the *COMMIT/* project funded by NWO.

7. REFERENCES

[1] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, pages 1–70, 2008.

[2] *Intel 64 and IA-32 Architectures Software Developer's Manual*, 06 2013.

[3] D. Adler, C. Gläser, O. Nenadic, J. Oehlschlägel, and W. Zucchini. *ff: memory-efficient storage of large data on disk and fast access functions*, 2013. R package version 2.2-11.

[4] F. Chen and B. D. Ripley. Statistical computing and databases: Distributed computing near the data. In *Proceedings of the 3rd International Workshop on Distributed Statistical Computing (DSC 2003)*, 2003.

[5] J. Conway, D. Eddelbuettel, T. Nishiyama, S. K. Prayaga, and N. Tiffin. *RPostgreSQL: R interface to the PostgreSQL database system*, 2013. R package version 0.4.

[6] M. Dowle, T. Short, S. L. with contributions from A Srinivasan, and R. Saporta. *data.table: Extension of data.frame for fast indexing, fast ordered joins, fast assignment, fast grouping and list columns.*, 2013. R package version 1.8.10.

[7] P. Große, W. Lehner, T. Weichert, F. Färber, and W.-S. Li. Bridging two worlds with RICE integrating R into the SAP in-memory computing engine. *PVLDB*, 4(12):1307–1317, 2011.

[8] M. Hornick and T. Plunkett. *Using R to Unlock the Value of Big Data: Big Data Analytics with Oracle R Enterprise and Oracle R Connector for Hadoop*. McGraw-Hill Osborne Media, 2013.

[9] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten. MonetDB: Two decades of research in column-oriented database architectures. *IEEE Data Engineering Bulletin*, 35(1):40–45, 2012.

[10] D. T. Lang. Scenarios for using R within a relational database management system server. Technical report, Bell Labs, 2001.

[11] H. Mühleisen and T. Lumley. Best of both worlds: relational databases and statistics. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, SSDBM, pages 32:1–32:4, New York, NY, USA, 2013. ACM.

[12] D. Mukhin, D. A. James, and J. Luciani. *ROracle: OCI based Oracle database interface for R*, 2013. R package version 1.1-10.

[13] R. S. I. G. on Databases. *DBI: R Database Interface*, 2013. R package version 0.2-7.

[14] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, 2013.

[15] R Core Team. *R Internals*. R Foundation for Statistical Computing, 3.1.0 edition, 2014.

[16] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-store: A column-oriented DBMS. In K. Böhm, C. S. Jensen, L. M. Haas, M. L. Kersten, P.-Å. Larson, and B. C. Ooi, editors, *VLDB*, pages 553–564. ACM, 2005.

[17] S. Venkataraman, E. Bodzsar, I. Roy, A. AuYoung, and R. S. Schreiber. Presto: distributed machine learning and graph processing with sparse matrices. In Z. Hanzálek, H. Härtig, M. Castro, and M. F. Kaashoek, editors, *EuroSys*, pages 197–210. ACM, 2013.

[18] Y. Zhang, H. Herodotou, , and J. Yang. RIOT: I/O-efficient numerical computing without SQL. In *Proceedings of the 2009 Conference on Innovative Data Systems Research*, 2009.

[19] Y. Zhang and J. Yang. Optimizing I/O for big array analytics. *CoRR*, abs/1204.6081, 2012.

APPENDIX

A. FULL EXPERIMENTAL TIMINGS

All timings are given in seconds.

10 MB		`data.table`	`RSQLite`	`MonetDB.R`	`Prototype`
Selection	1%	0.07	0.01	0.15	0.01
	10%	0.08	0.04	0.41	0.01
	50%	0.10	0.22	2.48	0.02
Selection/ Projection	1%	0.07	0.01	0.15	0.01
	10%	0.08	0.04	0.41	0.01
	50%	0.10	0.22	2.48	0.02
Grouping	1	0.01	0.23	0.25	0.01
	500	0.07	0.60	0.17	0.02
	10%	0.75	0.79	0.35	0.05
Joining	1%	0.03	0.02	0.16	0.02
	10%	0.09	0.16	0.64	0.02
100 MB					
Selection	1%	0.50	0.07	1.07	0.02
	10%	0.48	0.67	4.06	0.47
	50%	0.57	3.04	93.21	2.22
Selection/ Projection	1%	0.47	0.04	0.46	0.01
	10%	0.48	0.43	0.67	0.10
	50%	0.57	1.98	32.47	0.59
Grouping	1	0.02	2.31	0.88	0.06
	500	4.17	7.23	0.52	0.06
	10%	10.78	8.33	4.08	0.73
Joining	1%	0.52	0.20	0.28	0.07
	10%	0.32	69.66	0.64	0.25
1 GB					
Selection	1%	4.33	0.70	4.19	0.08
	10%	4.55	6.07	82.90	1.97
	50%	5.29	28.30	540.60	14.53
Selection/ Projection	1%	4.12	0.51	2.24	0.09
	10%	4.24	4.04	47.08	0.43
	50%	5.06	21.44	22.21	2.28
Grouping	1	0.17	25.04	3.77	0.47
	500	56.46	90.35	2.33	0.52
	10%	145.29	117.95	72.09	10.82
Joining	1%	8.29	2.63	2.99	0.61
	10%	3.33	2.57	48.55	1.23
10 GB					
Selection	1%	109.18	7.97	164.53	1.25
	10%	291.63	66.75		27.47
	50%	636.89	337.47		111.25
Selection/ Projection	1%	105.09	5.04	47.71	0.47
	10%	414.02	50.53	45.15	4.07
	50%	334.62	245.27		23.14
Grouping	1	1.55	273.22	212.09	104.98
	500	1102.49		207.00	92.63
	10%			2595.01	1863.73
Joining	1%	980.11			1478.90
	10%	247.42			1490.24