Journal of Statistical Software

November 2018, Volume 87, Issue 4. doi: 10.18637/jss.v087.i04

Database-Inspired Optimizations for Statistical Analysis

Hannes Mühleisen
Centrum Wiskunde & Informatica

Alexander Bertram
BeDataDriven

Maarten-Jan Kallen
BeDataDriven

Abstract

Computing complex statistics on large amounts of data is no longer a corner case, but a daily challenge. However, current tools such as GNU R were not built to efficiently handle large data sets. We propose to vastly improve the execution of R scripts by interpreting them as a declaration of intent rather than an imperative order set in stone. This allows us to apply optimization techniques from the columnar data management research field. We have implemented several of these optimizers in **Renjin**, an open-source execution environment for R scripts targeted at the Java virtual machine. The demonstration of our approach using a series of micro-benchmarks and experiments on complex survey analysis show orders-of-magnitude improvements in analysis cost.

Keywords: automatic optimization.

1. Introduction

Computing complex statistics on large amounts of data is no longer a corner case, but a daily challenge. Existing tools such as R (R Core Team 2018) strain to handle large data sets efficiently, and achieving acceptable speed often requires statisticians to invest precious research time explicitly parallelizing their analyses, or rewriting analysis code to lower-level languages such as C/C++ or Fortran.

Outside of the statistical computing community, however, the field of *relational query optimization* has a long history solving challenges of comparable scale. From their very roots in System R (Selinger, Astrahan, Chamberlin, Lorie, and Price 1994), all relational data management systems have contained a query optimizer. The optimizer is a crucial component, the cost of running a query can diverge several orders of magnitude depending on the order and choice of operators. A well-known technique to implement these optimizers is through a

collection of optimization rules (Freytag 1987; Pirahesh, Hellerstein, and Hasan 1992). Other optimizations common to relational data management systems such as automatic intra-query parallelization are more crucial for performance than ever due to the leveling off in processor clock speeds and the resulting necessary move to multi-core architectures and applications.

To date, such techniques have enjoyed limited application to more complex statistical analyses written in imperative languages such as R, C, Fortran, or Python. In contrast to declarative query languages such as SQL or pure functional languages such as Haskell, the imperative first-this-than-that style can make it easier for researchers to write complicated analyses, but harder for interpreters to apply global optimizations automatically.

Although these limitations certainly apply to imperative programs in general, there are reasons to suspect that these techniques might still be useful to data analysis programs in particular. First, the run time of data analysis programs, we argue, tends to be dominated by computations free of side effects, punctuated only periodically by input or output operations whose execution sequence is constrained.

In addition, a significant difference between the S language and other scripting languages such as Perl or Python is R's *vectorized* data model. All variables in R are vectors of (almost) arbitrary length, and most operators will affect all elements of a vector. From a high-level view, an R program is thus a collection of functions operating on data vectors.

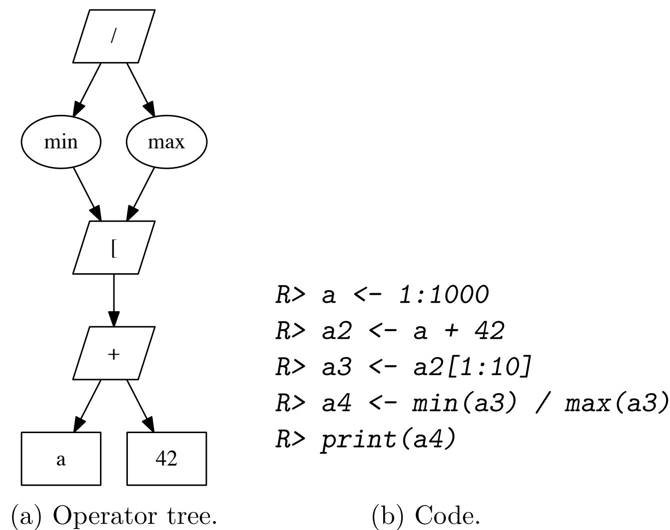
This unique property allows us to approach program execution from a *radically* different standpoint. Instead of the imperative, code-centric view of the original GNU R interpreter, we propose to *interpret R programs as a declarative, data-centric specification of intent*, similar to relational queries. This paradigm shift allows an execution engine to perform advanced reasoning for example on execution order or computational cost. This is useful because the analytical workloads the R environment was created for fundamentally differ from workloads that characterize general-purpose programming.

In this paper, we investigate the question what benefits optimization techniques common to data management systems could bring to statistical data analysis. We do so by integrating various optimization techniques into **Renjin**, an alternative R execution engine. Our central hypothesis is although R scripts appear to be similar to imperative programs like C or Fortran, they can be cleverly reinterpreted through partial evaluation to be closer to a declarative data analysis program like the ones expressed in SQL. We showcase our optimizations on both micro-benchmarks and a real-world use case, namely a large-scale survey analysis on American Community Survey (ACS) data.

The remainder of this paper is organized as follows: Section 2 introduces the **Renjin** execution environment for R scripts. Section 3 describes optimization techniques common to columnar data management systems. Section 4 describes our experiment methodology. Section 5 describes a series of micro-benchmarks aimed at showing the effects of individual optimization rules. Section 6 describes our experiments with a complex survey analysis use case. Finally, Sections 7 and 8 conclude the paper together with an overview of related work.

2. Deferred computation in Renjin

Renjin (BeDataDriven 2016) is an implementation of the R language and environment built on the Java virtual machine (JVM), whose goal is to simplify analysis of large data sets and facilitate integration with enterprise systems such as databases and application servers. **Renjin**



(a) Operator tree. (b) Code.

Figure 1: Deferred computation example.

supports several levels of indirection in data access and function dispatch, allowing uniform access to data regardless of its location, be it cloud storage, a relational database, a flat file or main memory. This allows **Renjin** to run unmodified R code on data from arbitrary sources without first loading the data into main memory. This includes many contributed packages and some C and Fortran code, which is automatically translated to JVM bytecode by **Renjin**'s toolchain. This additional layer of abstraction was a major motivation for undertaking a new interpreter; the original GNU R code base is tightly coupled to operations on in-memory arrays.

Renjin supports deferring side-effect free ("pure") computation. Instead of returning the actual result of a computation, a placeholder object is returned. This placeholder looks and behaves exactly like the actual computation result, but will only calculate results if forced to. A major advantage is that values that are not read are never computed. Furthermore, where GNU R is forced to immediately allocate space in virtual memory to hold the (potentially very large) intermediate results, **Renjin** does not need to allocate any space. This is particularly interesting, for example, for computations that generate large matrices such as the cross product. In **Renjin**, the entries of said matrix would only be calculated when read. Summary operations such as computing the minimum of a vector are also deferred to create parallelization opportunities. Deferring already deferred computation leads to an incrementally constructed directed acyclic graph (DAG) describing the computation.

Listing 1b shows an illustrative example for performance benefits through computation deferral: The computation (`+ 42`) on the whole vector `a` is pointless, because only the first 10 values are used in subsequent computation. Furthermore, the `min` and `max` computation are independent of each other and can run in parallel. Note that the `print` command is the one that ultimately forces computation of the entire computation graph. The entire deferral graph is shown in Figure 1a. Here, as well as in the other plots of deferral graphs, squares represent in-memory arrays, parallelograms represent placeholders, and ovals are summary operators.

The deferral graphs described above are created at runtime. Static source code analysis is

severely limited in dynamically-typed languages. It is, for example, impossible to tell at compile-time whether the `min` symbol refers to the built-in minimum operator, or whether another user-defined function will have been bound to the `min` symbol by the time the statement is evaluated. At runtime, at the moment that `min(a3) / max(a3)` is evaluated, this identity of the functions and the types of its arguments are known. This greatly simplifies analysis of execution graphs, and the point in time between deferral graph construction and execution is when relational-style optimizers can be applied.

One of the largest differences between relational operator trees and R scripts is that operator trees are almost never written by people, but generated from, for example, SQL queries. Also, relational operators do not have side effects. The former difference requires an optimizer to be far more robust in its optimization rules, as any combinations of operators mixed with user code can appear. The latter issue is being addressed in the data management community as user-defined functions written in foreign programming languages become more widespread, e.g., in Rheinländer, Beckmann, Kunkel, Heise, Stoltmann, and Leser (2014). In our case, we only consider pure functions for inclusion in the deferral graph.

3. Column-oriented database optimization methods

Relational query optimization builds on the declarative nature of relational queries. Queries specify "what", not "how", and there are a large number of result-equivalent execution plans for each query. However, the cost at which these plans are executed can differ widely. Furthermore, there are commonly a multitude of operator implementations available for each logical operator, the textbook example is the selection between a nested-loop or hash for a join operation.

The push-down of selections is an important optimization rule. Consider the following (simplified) example from Silberschatz, Korth, and Sudarshan (2006): "Find the names of all instructors in the Music department with the titles of the courses that they teach." In relational algebra, this query is expressed as

$$\pi_{\text{name,title}}(\sigma_{\text{dept='Music'}}(\text{instructor} \bowtie \text{course}))$$

Where π is a projection, σ a selection and $\bowtie$ a join. Within the relational algebra, there is an equivalency rule that states that the selection operation can be pushed over the join operation:

$$\sigma_\theta(E_1 \bowtie E_2) \equiv (\sigma_\theta(E_1)) \bowtie E_2.$$

This only holds if the selection criterion θ considers only the attributes (fields) of one of the expressions (e.g., tables) (E_1) in the join. This is the case in our example, hence the query above can be rewritten to

$$\pi_{\text{name,title}}(\sigma_{\text{dept='Music'}}(\text{instructor}) \bowtie \text{course}).$$

This rewritten form reduces the number of tuples from the instructor relation that are subject to the join and therefore likely improves performance.

The example is a static optimization rule, one that does not need to consider runtime properties of the data in order to be applied. There are also dynamic rules which depend on some property of the data at hand (Freytag 1987; Pirahesh *et al.* 1992). For example, the

optimizer rule selecting a join implementation is dependent on at least the cardinality of the joined relations. In another subsequent step, cost-based optimizers enumerate all possible execution plans and rank them according to a cost model.

As described above, R uses vectors of values as its primitive data type. There is a direct equivalent in the relational world, column stores. Column stores decompose the physical storage of a relational table into separate columns (Manegold, Boncz, and Kersten 2000; Stonebraker *et al.* 2005). This has large advantages due to the caching strategies of modern processors, in particular for analytical query workloads. Optimization techniques developed for column stores are thus particularly applicable to R scripts. To gain an overview over query optimization in relational column stores, we have surveyed the optimizers present in **MonetDB** (Boncz, Kersten, and Manegold 2008; Idreos, Groffen, Nes, Manegold, Mullender, and Kersten 2012). Some of the most critical optimization techniques are:

- *Selection pushdown.* As described before, selection pushdown has the potential to greatly reduce the cardinality of intermediate results. In the case of a column store, pushdown is slightly complicated if the selection predicate involves two or more columns.

- *Parallel scheduling.* Complex queries consist of several independent execution paths. For example, tuples from two relations might be joined together after a series of independent selections and aggregations has been performed on them. In a multi-threaded execution environment, execution time is greatly reduced if these independent "flows" are executed in parallel. The optimizer therefore identifies independent flows and schedules them to independent threads. In addition, it minimizes the lifetime of intermediate results to reduce the main memory footprint. This is particularly critical in the column-at-a-time execution model, where operators apply to entire columns at once.

- *Function de-virtualization.* It has been shown that dispatching virtual function calls for every value in a data analysis context creates a huge overhead (Boncz *et al.* 2008). Since all the values in a single column are of the same type, looking up the appropriate function (for example, addition) for every value is unnecessary. This optimizer replaces loops of virtual function calls with a specific, bulk-optimized operator.

- *Common expression elimination.* Often, portions of relational operator trees are identical. It is of course not necessary to execute the same computation several times. The common expression elimination thus traverses the tree and replaces identical sub-trees with a reference to the first instance.

In addition, it has been shown that it is highly beneficial to cache sub-expression results between queries for analytical query workloads (Ivanova, Kersten, Nes, and Gonçalves 2009). The rationale here is that while it can generally not be assumed that analytical queries are re-run so often that it is economical to cache their results, there are often ubiquitous sub-expressions that are part of many queries. Hence, caching intermediate results, for example in a least-recently-used (LRU) cache, has the potential to greatly improve query response times at the expense of memory.

In the remainder of this paper, we will explore how these optimizations could be applied to statistical analyses expressed as R scripts.

4. Experiment methodology

To investigate what benefits relational optimization techniques can bring to R analysis scripts, we have implemented several relational optimization techniques in **Renjin**. The pushdown of selections is achieved through the lazy evaluation technique described above. Instead of materializing results of computation immediately, a "promise" is returned. In the case of subset operations, where values are not modified, a placeholder object that implements a view on the original data is returned. Parallel execution is achieved by creating a pool of worker threads. The main thread continuously iterates through the tree of deferred computation and schedules nodes and their dependencies for computation in parallel. It should be noted that **Renjin**'s promises are different from GNU R's promises because they are not allowed to have side effects, which would prohibit some optimization strategies.

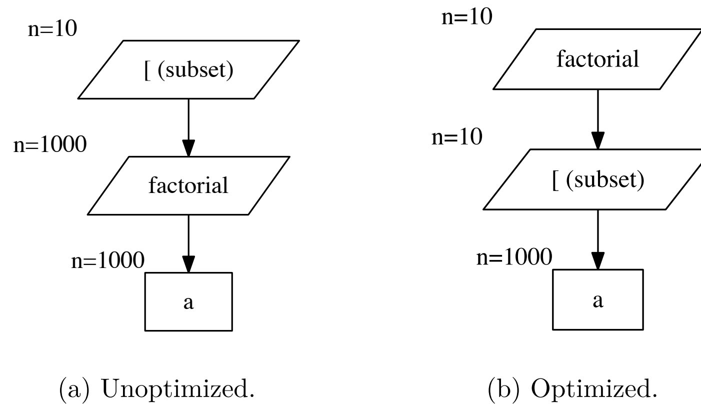
De-virtualization of data access is done through a specialization optimizer. This optimizer takes advantage of the infrastructure provided by the JVM to dynamically specialize summary operators such as `sum` or `colSums` on their argument trees. For a specific combination of operator and arguments, such as `sum(sqrt(x + y))`, where `x` is a double array, and `y` is an integer array, the specializer creates a new JVM class that computes the sum of the square root of the sum of each of the elements in `x` and `y`. The generated equivalent Java source code is as follows:

```
double compute(DoubleVector x, IntVector y) {
  double ax[] = x.toDoubleArray();
  int ay[] = y.toIntArray();
  double sum = 0;
  for (int i = 0; i < ax.length; ++i) {
    double t1 = ax[i] + ay[i];
    double t2 = Math.sqrt(t1);
    sum += t2;
  }
}
```

Currently, **Renjin** provides specializers for a limited number of summary functions (e.g., `sum` and `colSums`) that are responsible for emitting the JVM bytecode (Bruneton, Lenglet, and Coupaye 2002) needed to loop over the elements.

General modifications of the deferred operator tree are done by a list of visitor-pattern optimizers. The contract with each optimizer is that every node will eventually be passed in (not necessarily in any order) and that the optimizer must return a different value if it has modified the operator tree. The list of all optimizers is then applied until none of the optimizers reported a change.

Caching and re-use of the results of sub-expressions are implemented as a visiting optimizer. The optimizer analyzes the deferred computation tree. Vectors of values are immutable in **Renjin**, thus, an aggregation of the hash codes of values together with an identifier for a specific computation, for example `sum` yields a unique hash code for this computation on specific data. References to all nodes are inserted into a LRU hash tree using their aggregated hash code as key. Once these nodes have been computed, this tree will contain a reference to the result, and subsequent optimization runs can probe the tree, see whether the result is there, and replace matching nodes with pre-computed results.



(a) Unoptimized. (b) Optimized.

Figure 2: Pushdown experiment operator tree. Computing the `factorial` on vector `a` is costly, schedule subset operation first for performance.

In addition, we have implemented an "identity removal optimizer." This visiting optimizer removes redundant arithmetical operations, for example $x + 0$, $x \cdot 1$ or x^1.

As a general approach, we will begin with micro-benchmark experiments, and study the effect of each of the optimizations mentioned above in isolation. The experiment scripts for the micro-benchmarks are mainly chosen to be small and illustrative. Then, we move to a real-world complex use case. Here, a large amount of R code from the **survey** package (Lumley 2004, 2018), written by others is optimized using our techniques. We will be able to tell whether the benefits of our optimizations still show in this realistic task.

All experiments measured wall clock time as the main performance indicator. To account for operating system and virtual machine (e.g., garbage collection) noise, all experiments were repeated five times. After each run, we have cleared the expression result cache for fairness. All experiment result plots below include error bars showing the standard error of the mean, although they are not visible in several plots because the error was too small. Where applicable, we compared the performance of **Renjin** (with and without optimization) with the performance of GNU R.

All experiments (except one, see below) were run on a standard laptop computer with a 2-core Intel Core i7 processor at 3 GHz and with 16 GB of memory. The laptop uses Mac OS X 10.10.3 as its operating system and Java version 1.8.0 as JVM. For comparison, GNU R version 3.1.3 with the built-in BLAS library was used. The source code of our patched version of **Renjin** is available[1]. All experiment code, plotting scripts and raw results are also available[2].

5. Micro-benchmark experiments

5.1. Selection push-down

In this experiment we test the benefit of deferring materialization of computation results. The following test script calculates the factorial of the large vector `a`, formally $x! = \prod_{k=1}^{x} k$:

[1] https://github.com/bedatadriven/renjin/releases/tag/relational-optimizations
[2] https://scm.cwi.nl/DA/raaql-paper-experiments

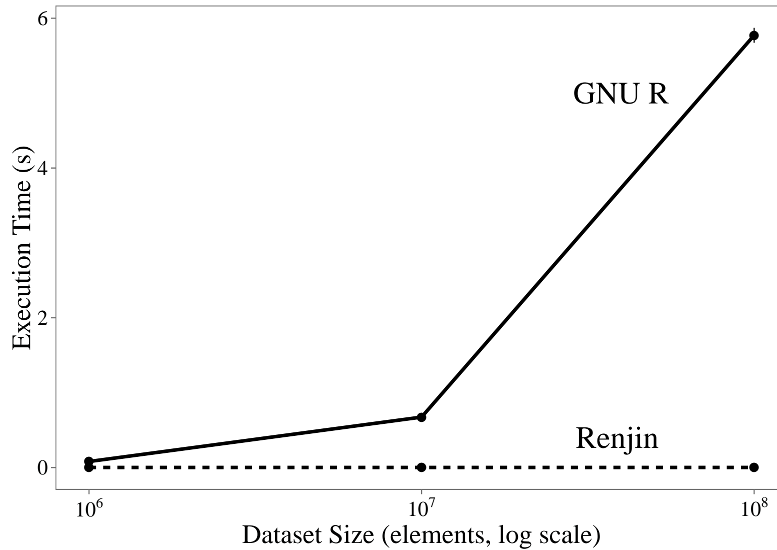


Figure 3: Selection pushdown experiment results.

```
R> b <- factorial(a)
R> b[1:10]
```

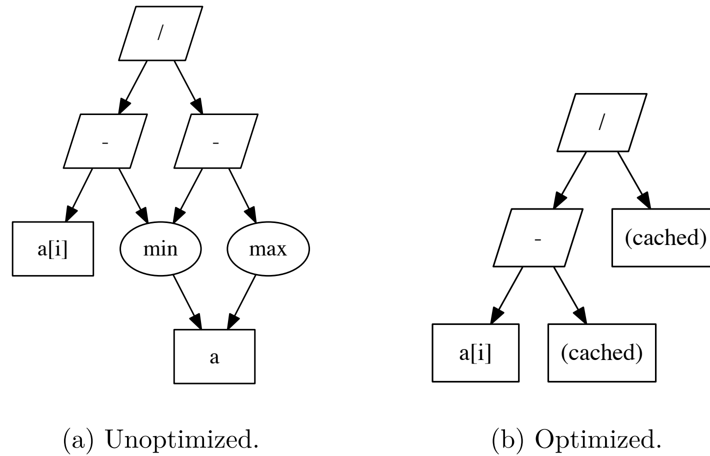
In a subsequent step, only the first ten entries are returned. The unoptimized and optimized operator trees are shown in Figure 2. We can see that logically pushing the selection of the first two values in front of the factorial reduces the amount of factorial computations to ten, regardless of the input size. As a variable, the size of vector `a` (containing random numbers in the range between 0 and 100) was varied between 10^6 and 10^8. We expect that GNU R with its immediate materialization approach will show greatly increasing execution times as the input data size increases, whereas **Renjin** will not be affected. Figure 3 shows the resulting execution times. We can see how **Renjin**'s execution time is constant and GNU R's execution time increases sharply.

5.2. Expression result caching

This experiment shows the benefit of expression result caching. The following test script performs statistical feature scaling on the first 100 values of a large vector, formally $X' = \frac{X - X_{\min}}{X_{\max} - X_{\min}}$:

```
R> for (i in 1:100) {
+    print((a[i] - min(a)) / (max(a) - min(a)))
+  }
```

Since the `print` statement will force computation, the test script leads to 100 separate deferred computation trees being built. However, since the vector `a` does not change within the loop, the results `min(a)` and `max(a) - min(a)` can be cached. Figure 4 shows the deferred operator tree for all loop iterations except the first one. We can see that the optimized version will only compute a difference and a quotient of scalar values due to the caching. Note that this computation contains an opportunity for parallel processing. However, we limited **Renjin** to



(a) Unoptimized. (b) Optimized.

Figure 4: Expression result caching experiment operator trees.

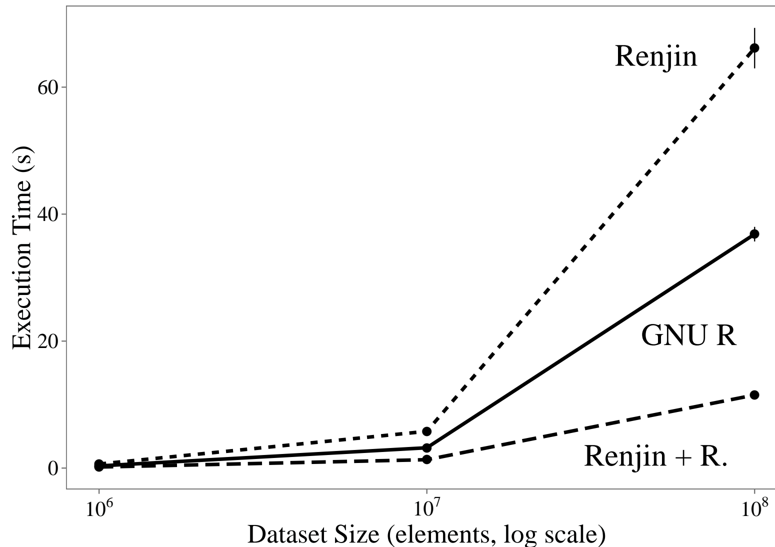


Figure 5: Expression result caching experiment results.

a single thread for fairness. Again, we varied the size of vector `a` as a variable between 10^6 and 10^8. We compare GNU R, and **Renjin** with and without the expression result caching optimization. GNU R's bytecode optimizer can greatly improve performance of interpreted loops in same cases. However, the real cost in this example is the repeated execution of the `min` and `max` functions, which the bytecode optimizer does not affect. We expect the caching optimization to greatly improve performance, and GNU R to suffer from repeated computation of redundant values.

Figure 5 shows the results of this experiment. First, we can see how **Renjin** without our optimization ("Renjin") performed considerably slower than GNU R. We attribute this difference to the predictable performance gap between tight C loops on arrays to virtual function calls in the JVM. We can however also see how **Renjin** with the result caching optimizer ("Renjin + R.") clearly outperforms GNU R as expected.

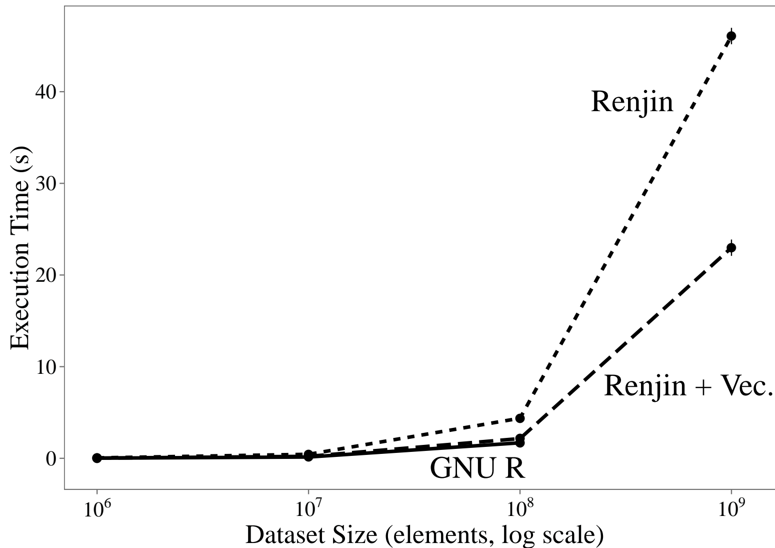


Figure 6: Vectorized operators experiment results.

5.3. Vectorized/specialized operators

As described above, the overhead created through virtualized (e.g., type-agnostic) function calls is considerable for large vectors. In this experiment, we test our optimization of using specialized code for certain operations. The following code shows a simple computation on two vectors:

```
R> sum(sqrt(a + 1) - b * a)
```

In this experiment, we varied the number of values in `a` and `b` between 10^6 and 10^9. We compare GNU R, and **Renjin** with and without the vectorized, direct-access operators. We expect the specializations to improve performance, but no significant performance benefit against GNU R, as we again have limited execution to a single thread for fairness.

Figure 6 shows the result of this experiment. We can see how the version of **Renjin** with vectorized operators ("Renjin + Vec.") clearly outperforms **Renjin** without this optimization by a factor of about 2. GNU R was unable to compute the function on the largest dataset within reasonable time (300s) due to memory exhaustion. This highlights another advantage of specialized operators.

5.4. Parallel execution

Renjin is able to schedule independent subtrees of computation to several worker threads. This experiment is testing the scalability of computation using this scheduling method. Contrary to the previous experiments, this test was run on a server with two 8-core Intel Xeon E5 processors at 2 GHz and with 256 GB of main memory. The following listing shows code calculating column-wise means of a large matrix `a` with 10 columns:

```
R> m <- matrix(nrow = ncol(a))
R> for (i in 1:ncol(a)) {
```

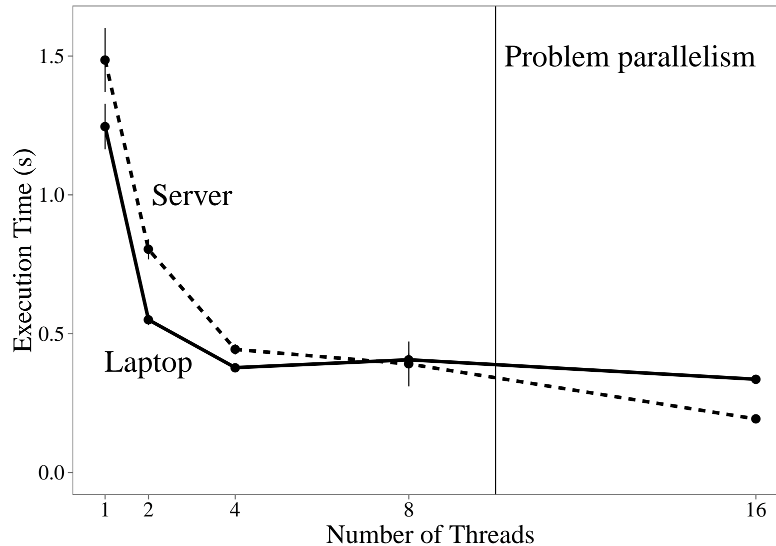


Figure 7: Parallel scheduling experiment results.

```
+     m[i, ] <- mean(a[, i])
+   }
```

In this experiment, we varied the number of rows in `a`. We also varied the number of threads **Renjin** is allowed to use and run the experiment on two different systems. Since every loop iteration is independent, we expect the parallel scheduler to greatly improve performance.

Figure 7 shows the results of this experiment. We can see how the execution time sharply drops as more threads are introduced. We can also see how the laptop outperforms the server from 1 to 4 threads. This is due to the greater clock speed of the laptop CPU. From 8 threads onwards, the server is faster, since it has more CPUs for the computation to be distributed to. However, since the available parallelism of the problem at hand is 10, the improvement between 8 and 16 threads on the server is small.

5.5. Identity removal

In this experiment, we use an admittedly extreme example for redundant arithmetical operations on a large vector `a`:

```
R> b <- a ^ 1 * 1 + 0
R> mean(b)
```

It should be kept in mind however that the constants could be less obvious, for example being weight variables, that just happen to be 0 or 1 in this particular invocation. This experiment is testing the effect of the identity removal optimizer. Effectively, only `mean(a)` needs to be computed, as can be seen from Figure 9. We varied the size of vector `a` as a variable between 10^6 and 10^8. We compare GNU R, and **Renjin** with and without the identity removal optimization. The expectations are clear, the identity removal optimizer should lead to a considerable difference in performance. Figure 8 shows the result of this

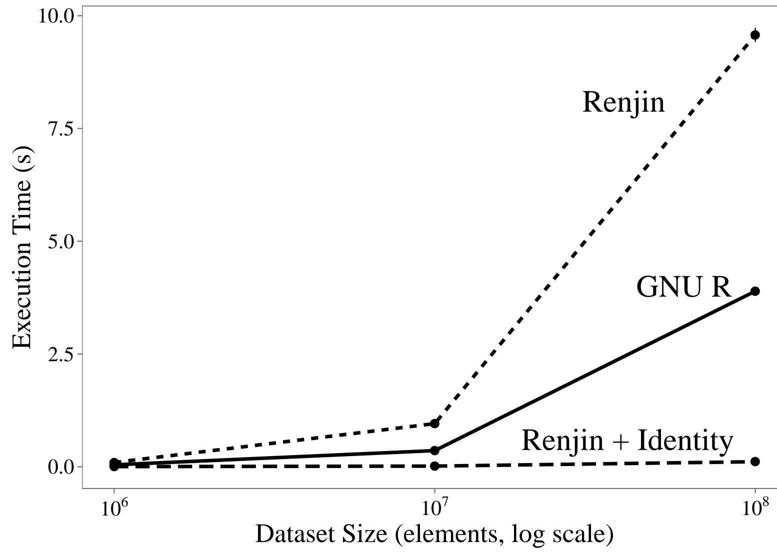
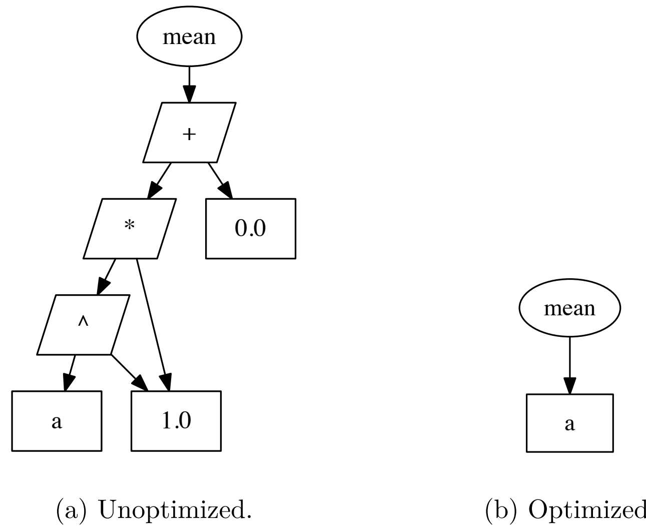


Figure 8: Identity optimizer experiment results.



(a) Unoptimized. (b) Optimized.

Figure 9: Identity optimizer experiment operator trees.

experiment. We can see results very similar to the previous experiment, with the optimized system showing the best performance.

In summary, all the optimizations we have introduced into **Renjin** have shown a beneficial effect towards analysis performance. In the next section, we will move on to a highly complex analysis on survey data.

6. Survey analysis experiments

As a realistic statistical analysis use case, we use an analysis of the American Community Survey (ACS; United States Census Bureau 2015). ACS is a yearly survey in the United

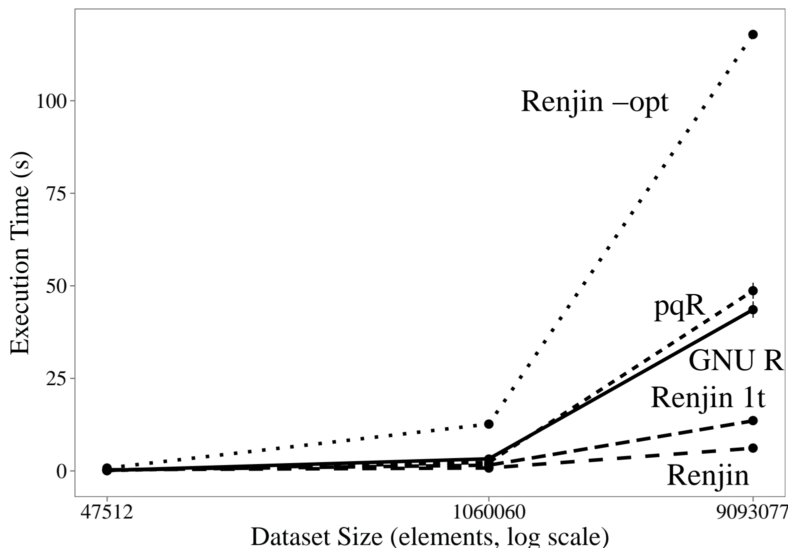


Figure 10: Survey experiment results.

States that is aimed at informing policymakers about general patterns in the population. Among others, data on income, health, education, transportation and housing are collected. However, only a tiny fraction of the population is actually surveyed. To still allow statistically valid inferences and to protect the privacy of participants, replication-based estimation is used. This means, that 80 weights are given for every observation in the dataset. These weights have to be considered to compute correct statistics. For example, the average age $\overline{\mathrm{age}}$ is calculated by multiplying the values with the overall weight column w. To obtain the standard error (SE) for this estimator, one has to repeat this calculation with all the replicate weight columns. In the case of ACS, there are 80 replicate weights w_r.

$$\overline{\mathrm{age}} = \frac{\sum(\mathrm{age} \times w)}{\sum w} \tag{1}$$

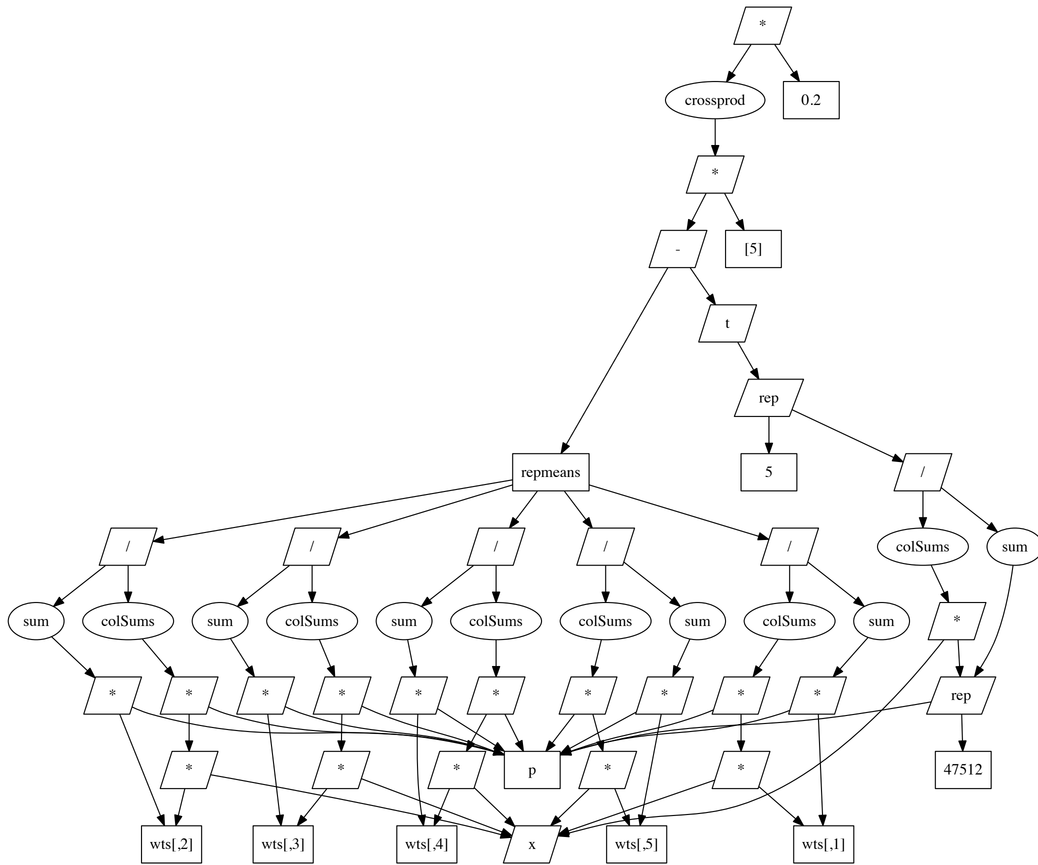
$$SE(\overline{\mathrm{age}}) = \sqrt{\frac{4}{80}\sum_{r=1}^{80}(\overline{\mathrm{age}}_r - \overline{\mathrm{age}})^2} \tag{2}$$

Here, $\overline{\mathrm{age}}_r$ is obtained by replacing w with w_r in Equation 1.

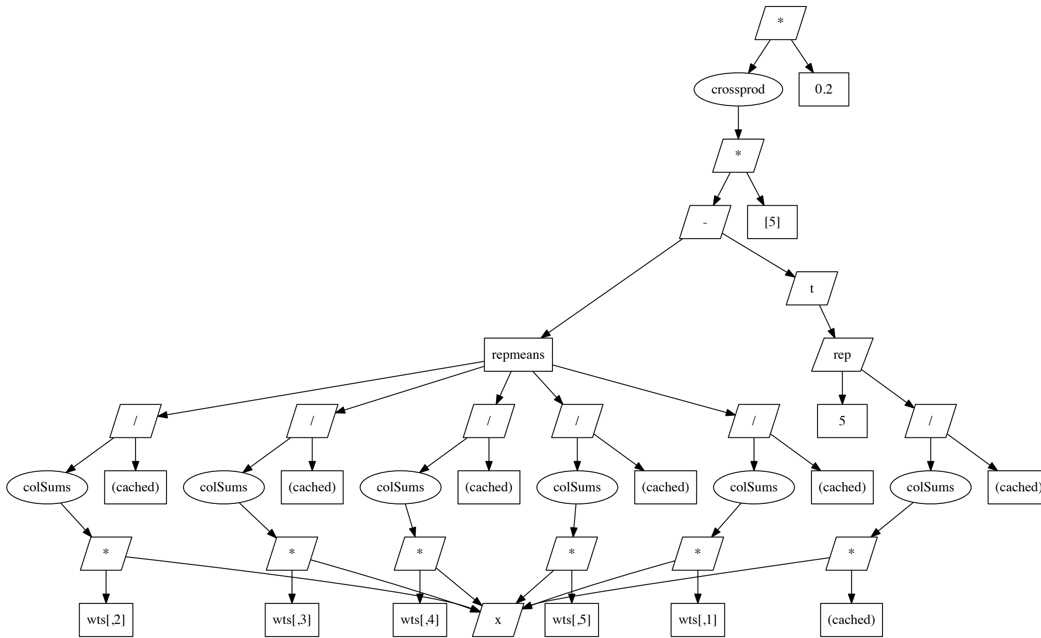
The R package **survey** for complex survey analysis is the only freely available software solution to analyze surveys like ACS. It supports surveys using replicate weights out-of-the box. The following listing shows a simple analysis of the mean age and income on the ACS data:

```
R> svydsgn <- svrepdesign(weight = ~ pwgtp, repweights = "pwgtp[1-9]",
+    scale = 4 / 80, rscales = rep(1, 80), mse = TRUE, data = svydata)
R> svymean(~ agep, svydsgn, se = TRUE)
R> svymean(~ adjinc, svydsgn, se = TRUE)
```

This short snippet expands to hundreds of lines of code through function calls. Through profiling, we found that most of the time computing these means is spent in the following code snippets:



(a) Unoptimized.



(b) Optimized.

Figure 11: Survey experiment operator trees (5 instead of 80 weights shown for readability).

```
R> for (i in 1:ncol(wts)) {
+    repmeans[i, ] <- t(colSums(wts[, i] * x * p) / sum(p * wts[, i]))
+  }
R> crossprod(sweep(theta, 2, meantheta, "-") * sqrt(rscales)) * scale
```

Here, `x` is the data column that is being analyzed, for example `agep`. Note how the `sum` computation is independent of `x`, making it a prime target for caching. Our modified version of **Renjin** is able to fully defer this computation.

The resulting deferred operator tree for the second call to `svymean` can be seen in Figure 11. We can see the effects of the identity optimizer, which removes the multiplication with `p`, which happens to be `1` in this invocation. We can also see how the independent multiplications have been cached. The 80 weight columns (of which only 5 are shown for readability) allow parallel computation. We have repeated the experiment for three different datasets, one subset with 47,512 rows, one subset with 1,060,060 rows and the entire dataset with 9,093,077 rows. We compare GNU R and our improved version of **Renjin**. For **Renjin**, we include a single-threaded run of the improved version and a version with our optimizations switched off. We also include a run on **pqR** (Neal 2015), a performance-optimized version of GNU R (see Section 7 for details). Note that we only considered the time taken to compute the survey averages, not the time it takes to construct the survey object. We expect the improved version of **Renjin** to outperform GNU R and **pqR** due to its strategic optimization.

Figure 10 shows the results of our experiment. We can see how **Renjin** with the sum of our optimizations vastly outperforms GNU R, in particular on the large dataset (6.1 seconds vs. 43.6 seconds). Furthermore, the unoptimized run ("Renjin -opt") shows the whole effect of our optimizations. Note that the performance benefits of our optimizations demonstrated in the micro-benchmarks translate to the complex survey analysis use case. Also, **pqR**'s tactical optimizations did not produce much improvement, it performed slightly slower than GNU R with 48.7s. Finally, the single-threaded run of the optimized version of **Renjin** ("Renjin 1t") shows that multithreading only improved the timing to 13.5 seconds on the large dataset. This might be due to the `crossprod` function in the computation, for which **Renjin**'s implementation is not yet parallelized.

Overall, we conclude that we are able to demonstrate large performance benefits of the strategic optimization techniques we have described on a real-world complex data analysis use case.

7. Related work

The various inefficiencies of GNU R are well-known and various approaches have been proposed to address them. For GNU R itself, a typical approach is to create special functions that combine functionality that previously required the creation of large intermediate vectors. These are then implemented in C for improved performance. For example, the `anyNA(x)` function in GNU R is such an optimized shortcut for `any(is.na(x))`. However, these optimizations are only effective locally, and are unable to perform global optimizations such as the ones we have presented. Moreover, they shift the burden of optimization from the interpreter to the programmer.

Riposte (Talbot, DeVito, and Hanrahan 2012) also uses deferred operator graphs obtained at run-time to optimize computation. Computation is done by a so-called "vector virtual machine", where a vector object supports primitive operations such as arithmetical operations,

reductions or grouping. Compatible vector objects are reduced, for example, several arithmetic operations on the same data can be computed in a single step. A JIT compiler then turns sequences of vector objects into SIMD-optimized code with multithreaded scheduling. Evaluation shows that Riposte achieves an average speedup of 15× over 12 different workloads without explicit programmer parallelization.

FastR (Kalibera, Maj, Morandat, and Vitek 2014) is another approach to optimizing R programs. Here, two major areas of optimization are considered: Code and data specialization. Code specialization is very similar to the Java HotSpot virtual machine mode of operation. Here, generic functions in the abstract syntax tree of the R program are observed during execution. If it is found that the types of the input values are fixed, a specialized implementation is chosen instead. To guard against possible type changes, guards are added before the specialized method call to be able to fall back to the generic implementation. For data specialization, simple cases of data types such as arrays of integers or integer sequences are represented by a special type. This simplifies matching for code specialization. In benchmark experiments, a speedup of 8.5x was measured. In a very related approach, a prototype R execution engine has also been implemented on the "Truffle" VM (Würthinger *et al.* 2013; Kotthaus, Plazar, and Marwedel 2012).

However, FastR and related approaches are unable to perform the global, analysis-level methods we have presented here. For data intensive programs like those that characterize R's use cases, the specializations of parts of the abstract syntax tree are too local and the "big picture" is obscured. Also, runtime specialization based on a discrete set of rules has limitations, as the number of specializations needed quickly explodes when considering possible permutations of operators and types. **Renjin** avoids this by generating specializations dynamically by composing specialized readers and iterators using bytecode generation at runtime.

Both Riposte and FastR, however, illustrate the challenges of exploring innovative optimization strategies for a language as expressive and dynamic as R. Because neither interpreter implements a large enough subset of the R language to run real-world workloads such as the survey experiment treated here (we confirmed this for FastR) it is difficult to judge whether improvements demonstrated on micro-benchmarks, such as computing a Fibonacci sequence, translate to performance gains on real-world workloads. Also, optimizing R code from a programming language perspective might be misleading, as its core is built around data crunching, not loop evaluation. To the best of our knowledge, nobody has yet successfully applied relational-style optimizations to statistical programs through deferred evaluation graphs.

pqR (Neal 2015) ("Pretty Quick R") is a heavily patched version of GNU R for improved performance. Changes focus on reducing interpreter overhead, moving some computation to secondary threads, merging sequences of vector operations and possibly parallelizing them and general avoidance of vector allocation where possible. While very impressive, these optimizations are strictly local, within a single or nested sequence of function calls. Strategic optimization over the entire computation is not considered.

A more low level optimization approach is "dynamic page sharing" (Kotthaus, Korb, Engel, and Marwedel 2014). Here, memory allocation is modified by increasing deduplication granularity from the object to the memory page level. Furthermore, hints were added to the GNU R interpreter to tell the memory allocator that a page would be, for example, completely overwritten. This effectively bypasses signal-based checks for modification. Benchmark results show a speedup of up to 5.2x and a memory footprint reduction of up to 53.8%. This opti-

mization can be considered orthogonal to our approach, as it only considers low-level memory overhead.

Another related approach is moving data-heavy computations to specialized systems. For example, the **sqlsurvey** package (Lumley 2014) constructs SQL queries to compute complex survey analysis results. Data and queries are handed over to MonetDB (Boncz *et al.* 2008), where the described operators apply relational optimizations to the generated queries. An extension of this approach is the creation of placeholder objects that wrap a database table in the R context (Mühleisen and Lumley 2013). Then, compatible operations such as groupings are automatically moved to the database. A similar approach was later followed by the commercial product "Oracle R Enterprise" (Oracle Corporation 2016). There are also approaches to lazy evaluation of data inside R itself. For example, the **dplyr** package defers actual computation of data analysis tasks as long as possible (Wickham and Francois 2015).

Finally, it should be noted that the programming language Haskell is built on lazy evaluation and optimizing program execution graphs and has been the subject of previous work as well (Coutts, Leshchinskiy, and Stewart 2007; Bloss, Hudak, and Young 1988). However, these optimizations are language- and not data-centric.

8. Conclusions and outlook

In this paper, we have proposed to view execution of statistical analysis programs from a different standpoint, where the analysis script is interpreted not in an imperative order, but as a declaration of intent, with the details of execution left to an execution engine. We have argued that optimization techniques developed for columnar relational databases are interesting candidates to optimize execution of statistical analysis programs. We have shown how deferred execution graphs are the preferred medium to perform this optimization on rather than parse trees due to clarity of types, etc. We have extended the **Renjin** system with relational-inspired strategical and tactical optimizers that operate on deferred execution graphs.

In a series of micro-benchmarks, we have shown how selection push down, expression result caching, identity removal, specialized operators and parallel scheduling all improve analysis performance. In another experiment we have investigated the impact of all our optimization on a unmodified real-world statistical program, a complex survey analysis on the American Community Survey dataset. Our results show that our optimized version of **Renjin** is able to outperform the standard GNU R interpreter by a factor of 7. To the best of our knowledge, such a degree of optimization has not been shown for complex analysis yet. We conclude that treating statistical data-heavy analysis as declarative intent and applying relational optimization techniques to them indeed has the potential to greatly improve the performance of statistical analysis on big(ger) data. More generally, we also conclude that holistic, strategic optimizations of data analysis scripts have a clear advantage over single-statement tactical optimisations.

8.1. Future work

In this paper, we have only considered a subset of the wealth of relational optimization methods, in particular, rule-based optimization. In future work we would like to expand our optimization to cost-based optimization techniques, where alternative execution graphs are

enumerated and ranked using a cost model. Developing such a cost model in the unknown environment of statistical analysis, where semantics of operators are less clear than in a relational system is certainly interesting. Scheduling execution could potentially benefit from cache- and NUMA-aware scheduling methods. Furthermore, there are multiple cache management strategies that could be investigated for the result caching method we have presented. Finally, operations on some vectors could be further split up to increase parallelism where possible.

Acknowledgments

This work was funded by the Netherlands Organisation for Scientific Research (NWO) and BeDataDriven through their project "R as a Query Language".

References

BeDataDriven (2016). ***Renjin*** *– A JVM-Based Interpreter for the R Language for Statistical Computing*. The Hague, The Netherlands. URL http://www.renjin.org/.

Bloss A, Hudak P, Young J (1988). "Code Optimizations for Lazy Evaluation." *LISP and Symbolic Computation*, **1**(2), 147–164. doi:10.1007/bf01806169.

Boncz PA, Kersten ML, Manegold S (2008). "Breaking the Memory Wall in **MonetDB**." *Communications of the ACM*, **51**(12), 77–85. doi:10.1145/1409360.1409380.

Bruneton E, Lenglet R, Coupaye T (2002). "ASM: A Code Manipulation Tool to Implement Adaptable Systems." In *In Adaptable and Extensible Component Systems*. URL http://asm.ow2.org/current/asm-eng.pdf.

Coutts D, Leshchinskiy R, Stewart D (2007). "Stream Fusion. From Lists to Streams to Nothing at All." In *ICFP'07*.

Freytag JC (1987). "A Rule-Based View of Query Optimization." In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, SIGMOD '87, pp. 173–180. ACM, New York. doi:10.1145/38713.38735.

Idreos S, Groffen F, Nes N, Manegold S, Mullender KS, Kersten ML (2012). "**MonetDB**: Two Decades of Research in Column-Oriented Database Architectures." *IEEE Data Engineering Bulletin*, **35**(1), 40–45. URL http://sites.computer.org/debull/A12mar/monetdb.pdf.

Ivanova MG, Kersten ML, Nes NJ, Gonçalves RAP (2009). "An Architecture for Recycling Intermediates in a Column-Store." In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pp. 309–320. ACM, New York. doi:10.1145/1559845.1559879.

Kalibera T, Maj P, Morandat F, Vitek J (2014). "A Fast Abstract Syntax Tree Interpreter for R." In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '14, pp. 89–102. ACM, New York. doi:10.1145/2576195.2576205.

Kotthaus H, Korb I, Engel M, Marwedel P (2014). "Dynamic Page Sharing Optimization for the R Language." In *Proceedings of the 10th Symposium on Dynamic Languages*, DLS '14, pp. 79–90. ACM, Portland. URL http://dl.acm.org/citation.cfm?id=2661094.

Kotthaus H, Plazar S, Marwedel P (2012). "A JVM-Based Compiler Strategy for the R Language." In *useR! 2012 – The International R User Conference, Abstract Booklet*, p. 68. Nashville. URL https://www.R-project.org/conferences/useR-2012/draft-abstract-booklet-06122012.pdf.

Lumley T (2004). "Analysis of Complex Survey Samples." *Journal of Statistical Software*, **9**(8), 1–19. doi:10.18637/jss.v009.i08.

Lumley T (2014). ***sqlsurvey**: Analysis of Very Large Complex Survey Samples.* R package version 0.6-11/r41, URL https://R-Forge.R-project.org/projects/sqlsurvey/.

Lumley T (2018). ***survey**: Analysis of Complex Survey Samples.* R package version 3.33-2, URL https://CRAN.R-project.org/package=survey.

Manegold S, Boncz PA, Kersten ML (2000). "Optimizing Database Architecture for the New Bottleneck: Memory Access." *The VLDB Journal*, **9**(3), 231–246. doi:10.1007/s007780000031.

Mühleisen H, Lumley T (2013). "Best of Both Worlds: Relational Databases and Statistics." In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, SSDBM, pp. 32:1–32:4. ACM, New York. doi:10.1145/2484838.2484869.

Neal R (2015). "**pqR** News." Accessed on 2015-06-25, URL http://www.pqr-project.org/NEWS.txt.

Oracle Corporation (2016). *Oracle Advanced Analytics – Oracle R Enterprise.* Redwood City. URL http://www.oracle.com/technetwork/database/database-technologies/r/r-enterprise/overview/.

Pirahesh H, Hellerstein JM, Hasan W (1992). "Extensible/Rule Based Query Rewrite Optimization in Starburst." In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, SIGMOD '92, pp. 39–48. ACM, New York. doi:10.1145/130283.130294.

R Core Team (2018). *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

Rheinländer A, Beckmann M, Kunkel A, Heise A, Stoltmann T, Leser U (2014). "Versatile Optimization of UDF-Heavy Data Flows with Sofa." In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pp. 685–688. ACM, New York. doi:10.1145/2588555.2594517.

Selinger PG, Astrahan MM, Chamberlin DD, Lorie RA, Price TG (1994). "Access Path Selection in a Relational Database Management System." In M Stonebraker (ed.), *Readings in Database Systems*, 2nd edition, pp. 84–95. Morgan Kaufmann Publishers Inc., San Francisco. URL http://dl.acm.org/citation.cfm?id=190956.190966.

Silberschatz A, Korth H, Sudarshan S (2006). *Database Systems Concepts*. 5 edition. McGraw-Hill, Inc., New York.

Stonebraker M, Abadi DJ, Batkin A, Chen X, Cherniack M, Ferreira M, Lau E, Lin A, Madden S, O'Neil E, O'Neil P, Rasin A, Tran N, Zdonik S (2005). "C-Store: A Column-Oriented DBMS." In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pp. 553–564. VLDB Endowment. URL http://dl.acm.org/citation.cfm?id=1083592.1083658.

Talbot J, DeVito Z, Hanrahan P (2012). "**Riposte**: A Trace-Driven Compiler and Parallel VM for Vector Code in R." In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pp. 43–52. ACM, New York. doi:10.1145/2370816.2370825.

United States Census Bureau (2015). "American Community Survey." Accessed on 2015-06-24, URL http://www.census.gov/acs/www/.

Wickham H, Francois R (2015). ***dplyr**: A Grammar of Data Manipulation*. R package version 0.4.1, URL https://CRAN.R-project.org/package=dplyr.

Würthinger T, Wimmer C, Wöß A, Stadler L, Duboscq G, Humer C, Richards G, Simon D, Wolczko M (2013). "One VM to Rule Them All." In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pp. 187–204. ACM, New York. doi:10.1145/2509578.2509581.

Affiliation:

Hannes Mühleisen
Database Architectures Group
Centrum Wiskunde & Informatica
Amsterdam, The Netherlands
E-mail: hannes@cwi.nl
URL: http://hannes.muehleisen.org/

Alexander Bertram, Maarten-Jan Kallen
BeDataDriven
The Hague, The Netherlands
E-mail: alex@bedatadriven.com, mj@bedatadriven.com
URL: http://www.bedatadriven.com/

Journal of Statistical Software http://www.jstatsoft.org/
published by the Foundation for Open Access Statistics http://www.foastat.org/
November 2018, Volume 87, Issue 4 *Submitted:* 2016-03-18
doi:10.18637/jss.v087.i04 *Accepted:* 2017-08-07