Integrating the R Language Runtime System with a Data Stream Warehouse

Carlos Ordonez, Theodore Johnson, Simon Urbanek, Vladislav Shkapenyuk, Divesh Srivastava

AT&T Labs - Research, USA [⋆]

Abstract. Computing mathematical functions or machine learning models on data streams is difficult: a popular approach is to use the R language. Unfortunately, R has important limitations: a dynamic runtime system incompatible with a DBMS, limited by available RAM and no data management capabilities. On the other hand, SQL is well established to write queries and manage data, but it is inadequate to perform mathematical computations. With that motivation in mind, we present a system that enables analysis in R on a time window, where the DBMS continuously inserts new records and propagates updates to materialized views. We explain the low-level integration enabling fast data transfer in RAM between the DBMS query process and the R runtime. Our system enables analytic calls in both directions: (1) R calling SQL to evaluate streaming queries; transferring output streaming tables and analyzing them with R operators and functions in the R runtime, (2) SQL calling R, to exploit R mathematical operators and mathematical models, computed in a streaming fashion inside the DBMS. We discuss analytic examples, illustrating analytic calls in both directions. We experimentally show our system achieves streaming speed to transfer data.

1 Introduction

Big data analytics is notoriously difficult when there exist multiple streams and there is a need to perform advanced statistical analysis on them, beyond SQL queries. In our case, we focus on enabling statistical analytics on a data stream warehouse (DSW) [8], where the goal is to analyze multiple streams of networking data (logs). Big data analytics research is active. From a systems perspective, parallel database systems and Apache Hadoop world (HDFS, MapReduce, Spark, and so on) are currently the main competing technologies [13], to efficiently analyze big data, both based on automatic data parallelism on a shared-nothing architecture. It is well established a parallel DBMS [4] is much faster than MapReduce for analytical computations on the same hardware configuration [14]. Spark [16] has become a faster main-memory alternative substituting MapReduce, but still trailing parallel DBMSs. Our protoype follows the DBMS

[⋆] Research work conducted while the first author was a visiting research scientist with AT&T. Carlos Ordonez current affiliation: University of Houston, USA.

approach. Database systems research has introduced specialized systems based on rows, columns, arrays and streams to analyze big data. Moreover, UML has been extended with data mining [17]. However, the integration of mathematical languages, like R, with SQL, remains a challenge [6]. Currently, R is the most popular open-source platform to perform statistical analysis due to its powerful and intuitive functional language, extensive statistical library, and interpreted runtime. Unfortunately, as noted in the literature, even though there exists progress to scale R to large data sets and perform parallel processing, such as IBM Ricardo (R+MapReduce) [3], SciDB-R (R+SciDB) [15], HP Distributed R suite (R+Vertica), R remains inadequate and slow to analyze streams.

1.1 Motivation: SQL and R languages

From a language perspective, SQL remains the standard query language for database systems, but it is difficult to predict which language will be the standard for statistical analytics: we bet on R. With that motivation in mind, we present STAR (STream Analytics in R), a system to analyze stream data integrating the R and SQL languages. Unlike other R tools, STAR can directly process streaming tables, truly performing "in-database" analytics. STAR enhances R from several angles: it eliminates main memory limitations from R, it can perform data preprocessing with SQL queries, leaving mathematically complex computations as a job for R. STAR enables analytics in both directions closing the loop: (a) R programs can call SQL queries. (b) SQL queries can call R functions. In short, on one hand, users can exploit DBMS functionality (query processing, security, concurrency control and fault tolerance) and on the other hand, call R as needed for mathematical processing.

1.2 Related Work: Other Analytic Systems

The problem of integrating machine learning algorithms with a DBMS has received moderate attention [6, 12, 10]. In general, such problem is considered difficult due to a relational DBMS architecture [11], the matrix-oriented nature of statistical techniques, lack of access to the DBMS source code, and the comprehensive set of techniques already available in mathematical languages like Matlab, statistical languages like R, and numerical libraries like LAPACK. The importance of pushing statistical and data mining computations into a DBMS is recognized in [1]; this work emphasizes exporting large tables outside the DBMS is a bottleneck and it identifies SQL queries and HDFS/MapReduce (now substituted by Spark) as two complementary mechanisms to analyze large data sets. As a consequence, in modern IT environments, users generally export data sets to a statistical tool or a parallel system, iteratively build several models outside the DBMS, and finally deploy the best model back into the DBMS. Exploratory cube analysis does most of query processing inside the DBMS, but the computation of statistical models (e.g. regression, PCA, decision trees, Bayesian classifiers, neural nets), graph analytics (e.g. shortest path, connectivity, clique detection) and pattern discovery (e.g. itemsets, association rules) is more commonly performed outside a DBMS, despite the fact that DBMSs indeed offer some data mining algorithms and powerful query capabilities. A promising direction bridging DBMS and Big Data technology is to integrate query processing and MapReduce for exploratory cube queries [7].

2 Overview of Interconnected Systems

2.1 Database System for Static Compiled SQL Queries

We first discuss the main features of the data stream warehousing (DSW) system TidalRace [8], contrasting them with features from previous systems. We start with an overview of data stream systems built at AT&T. GigaScope Tool [2] was a first system that could efficiently evaluate a constrained form of SQL on packet-level data streams with recent data as they were flowing in a network interface card. As main limitations, it was not capable of continuously storing streaming data, it could not take advantage of a parallel file system in a cluster of computers and it could not correlate (analytical querying) recent data with historical data. As time went by it became necessary to store summary historical data from streams (orders of magnitude smaller than packet-level data, but still orders of magnitude larger than transactional data) and support standard SQL including arbitrary joins (natural, outer, time band) and all kinds of aggregations (distributive, holistic). Thus the DataDepot DSW [5] was born, with novel storage on top a POSIX parallel file system, supporting standard SQL and incorporating UDFs [9]. More recently, the big data wave brought new requirements and new technology: higher stream volume (more streams with more data), intermittent streams (with traffic spikes), more efficient C++ code for queries, HDFS (instead of a POSIX file system), eventual consistency and advanced analytics beyond SQL queries. These requirements gave birth to TidalRace [8]. TidalRace [8] represents a next-generation data warehousing system specifically engineered for data management of high volume streams, building on long-term experience from the previous systems.

Storage: TidalRace uses a parallel file system (currently HDFS), where time partitions (a small time interval) are the main storage I/O unit for data streams, being stored as large blocks across nodes in the parallel cluster. The storage layout is hybrid: a row store for recent data (to insert stream records and maintain some materialized views), and a column store for large historical tables with recent and old data (to evaluate complex queries). The system provides a DDL with time-oriented extensions. Atomic data types include integers, floats, date/time, POSIX timestamps and strings. Vectors and matrices are supported internally within UDFs in C++ and special SQL access functions. The DBMS supports time-varying schemas, where columns are added or deleted from an existing table over time. This advanced feature is fundamental to keep the system running without interruption concurrently processing insertions, queries and propagating updates to materialized views.

Language: The DBMS provides standard SQL enhanced with time-oriented extensions to query streaming tables. Its SQL offers both distributive aggregations (e.g., sum() and count()) and holistic aggregations (e.g., rank, median, OLAP functions). User-defined functions (UDFs) are available as well: scalar and user-defined aggregates (especially useful for analytics), programmable in the C language. Query processing is based on compiling SQL queries into C code, instead of producing a query plan in an internal representation, which allows most optimizations at compile time. Materialized views, based on SQL queries combining filters, joins and aggregations, are a fundamental feature.

Processing: The database is refreshed by time partition, being capable of managing out-of-order arrival of record batches, intermittent streams and streams with varying speed (e.g. traffic spikes). That is, the system is robust to ingest many diverse streams traveling in a large network. The system uses MVCC (lock-free), which provides read isolation for queries when they are processed concurrently with insertions. The system provides ACID guarantees for base tables (historical tables) and database metadata (schema info, time partition tracking), and eventual consistency for views (derived tables). Therefore, queries, including those used in views, read the most up-to-date version, which is sufficient to compute queries with joins and aggregations on a time window. Query processing is multi-threaded, where threads are spawned at evaluation time by the query executable program. A key feature are materialized views, which are periodically updated when inserting records. Materialized views are computed with SQL queries combining selection filters on time partitions, time band joins and aggregations. We emphasize that every query and view should have a time range, where such time range generally selects the most recent data. The DBMS operates with a minimal time lag between data stream loading and querying (1-5 minutes) and efficiently propagates insertion of new records and removes old records to update materialized views.

In Figure 1 we show analytic applications at AT&T, where base tables are periodically and asynchronously appended by time partition, ingesting different streams, and derived tables are materialized views of compiled SQL queries. Derived tables are periodically updated with either incremental computation when feasible (i.e. joins, distributive aggregations, approximate histograms) or total recomputation of complex mathematical models during low stream traffic or low usage periods. In general, STAR pulls data from streams and pushes new records to derived tables to compute aggregations, descriptive statistics, histograms and machine learning models using the R language.

2.2 R Dynamic Runtime for Interpreted Functional Programs

Storage: R provides atomic data types and data structures like most programming languages. Atomic data types have a 1-1 correspondence to data types in the C language and include integer, real, string and timestamp, where only strings have different storage from C. For data structures R provides vectors, matrices, data frames, and lists. Vectors represent a collection of elements of the same atomic data type, especially real and integer. Vectors are stored as one

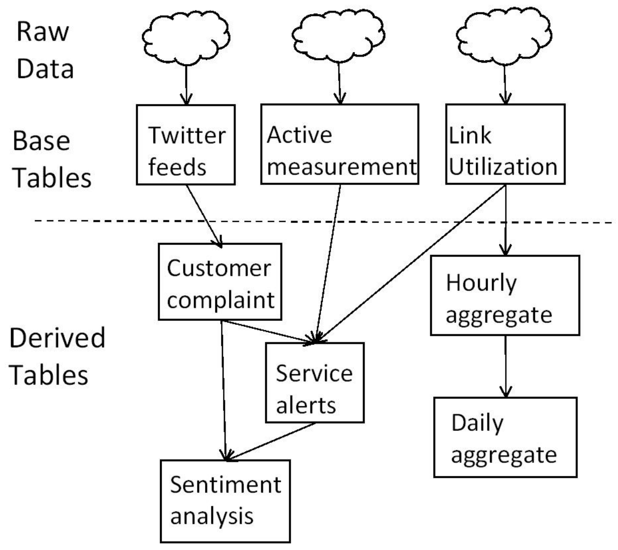

Fig. 1. TidalRace data stream warehousing system end-user applications.

contiguous block dynamically allocated. Matrices are 2-dimensional array of real numbers, also stored as one block in column major order dynamically allocated. Data frames are a list of columns of diverse data types, where each column is a C array dynamically allocated. Lists are the most general and can have elements of diverse data types, including atomic data types and even nested data structures. Therefore, it is feasible to create lists mixing lists, matrices and vectors. Vectors and matrices are easy to manipulate either as a single block or cell by cell. Data frames and lists are the most difficult to manage, easy to manipulate in R itself, but more difficult in the C language. Lists, in their most general form, are the most difficult to manipulate and transfer to the database system.

Language: By its original specification based on S, R is fundamentally functional, but it also incorporates imperative programming statements (i.e. if-then-else and loops). The data type of a variable is totally dynamic because it can change any time with a new value assignment. R also incorporates object-oriented features that enable the creation of new data types, libraries and reusable functions. To manipulate vectors, matrices and data frames R provides simple, yet powerful, subscript operators to access rows, columns and individual cells. Finally, function arguments are named which allows passing argument values in different order and providing default values. Based on our experience, the two most difficult integration aspects are correctly building data structures and setting up all function parameters before calling an R function. From a performance perspective, the most difficult programming aspect is following a functional style over an imperative style (i.e. C++) or declarative style (i.e. SQL), which requires a different algorithmic approach.

Processing: The R language runtime combines a script-based interactive shell and a dynamic interpreter. Matrix and data frame operators are evaluated in the C language and certain linear algebra matrix operators and numerical methods are evaluated by the LAPACK library. The most common form to read and write files is via plain text file I/O (e.g. with csv files), but it is feasible to perform binary I/O with pre-defined binary formats. Processing is single-threaded, which does not exploit multicore CPUs, but which requires simpler memory management since no locks are required. When R functions are called, the R runtime creates nested variable environments, which are dynamically scoped. The R garbage collector takes care of discarding old variable environments and releasing main memory, which simplifies programming, but it may be inefficient. Since the core R runtime has not suffered fundamental changes since it was born, its main memory footprint is small, especially nowadays when computers have ample RAM. The most difficult memory management aspect is tracking when a variable is no longer accessible, but this is reasonably managed by R's garbage collector.

3 Bidirectional Analytic Processing

3.1 STAR Architecture

Our system provides a fully bidirectional programming API: an R program can call (evaluate) any SQL query and its results are seamlessly and efficiently transferred into R. Alternatively, SQL can call any R function via UDFs. Both complementary interfaces are explained below. Our integrated system architecture is shown in Figure 2.

STAR makes two strong, but reasonable, assumptions to process stream data: (1) the result table from an SQL query with a time range generally fits in RAM. (2) for those result tables from SQL queries that cannot fit in RAM they generally represent materialized views (i.e. pre-processed data streams), which SQL can handle.

The key issues to integrate R with a DBMS are understanding main memory management, layout of vectors and matrices in RAM, building data frames as a set of columns, setting up R function calls, access serialization and properly configuring the operating system environment. Main memory management is significantly different in both systems. R has a garbage collector and the runtime is single threaded. R can address main memory with 64 bits, but integers for subscripts to access data structures are internally 32 bits. On the other hand, the C++ in the DBMS uses a flat 64 bit memory space also with a single thread per compiled query, but no garbage collector. Therefore, each system works as a separate OS process with its own memory space. In addition, since both systems internally have different data structure formats it is necessary to transfer and cast atomic values between them. A fundamental difference with other systems,

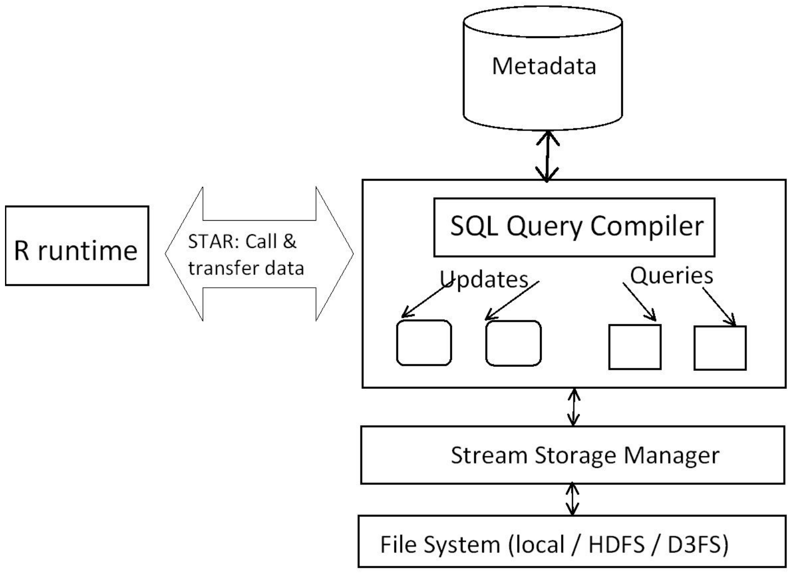

Fig. 2. STAR architecture: R $\Leftrightarrow$ DBMS.

integrating R and a parallel data system, is that building data structures and transferring them is done only in main memory, copying atomic values as byte sequences in most cases, moving memory blocks from one system to the other and avoiding creating files. In other words, we developed a direct binding between R and DBMS runtimes, bypassing slow network communication protocols.

3.2 Mapping Data Types and Assembling Data Structures

STAR exchanges data between R and the DBMS with a careful mapping between atomic values. Data structures like vectors, matrices, data frames and tables are built from atomic values. The atomic mapping is defined as follows: A real in R and the DBMS are both internally a C double in their interpreter and SQL compiler, respectively. Integers are more complicated because in C they are tied to the CPU register size, but R has an older code base; 32 bit integers are directly mapped between both systems, but C++ 64 bit integers are problematic because an integer in R internally still uses a 32 bit integer. Therefore, a 64 bit integer in C++ must be mapped to a real in R (a double in C), which avoids potential overflows. Conversely, when transferring from R to the DSW integers are mapped to 32 bit integers. Strings in the DSW are managed storing their length to avoid scanning for the C null terminator, whereas in R strings are vectors of characters. That is, neither system uses standard C or C++ strings. Therefore, strings require more involved manipulation, but since their length is pre-computed and stored they can be easily moved as byte sequences. Finally, POSIX timestamps are integers on the DBMS side, but real (C double) on the R side, which requires type casting going into either system.

Data structures include vectors, matrices and data frames on the R side and tables (including materialized views) in SQL. To achieve maximum efficiency, transferring is always done moving atomic values between systems as byte sequences: string parsing is avoided. We make sure a data frame only contains atomic values (i.e. complying with 1NF), thereby enabling transferring data into an SQL table. Lists in R violate a database first normal form. Therefore, they cannot be transferred into the DBMS, but they can be pre-processed converting them into a collection of data frames. Transferring in the opposite direction, an SQL table is straightforward to convert into an R data frame since the latter is a more general data structure. Converting an SQL table into an R matrix requires considering a sparse or dense matrix storage and how subscripts are represented in SQL. Finally, vectors/matrices in C++ are a mechanism to efficiently transfer and serialize data from the UDF to vectors/matrices in R (which have different storage and require memory protection), but not to perform statistical analysis. That is, they are transient data structures.

3.3 Data Transfer

We start by considering storage layout and processing in both systems. From the DBMS side we aim to process streams which come in row form by default. On the R side, there are two main data structures: data frames and matrices (vectors being a particular case). Notice that in general, matrices come as output from a machine learning algorithm, transforming the input data set into matrix form if needed. Data frames are organized in RAM by column, which means that a single column is stored in contiguous space. The second consideration is size: we assume tables coming from the DBMS cannot fit in main memory. The row to column conversion and limited main memory leads to block-based algorithms, which transform SQL table rows into column-oriented chunks (blocks) of a data frame. Given the bidirectional transfer, one algorithm transfers data from the DBMS to the R runtime and the second one from the R runtime to the DBMS. In order to handle such bidirectional data transfer efficiently we developed a packed (binary, space-efficient) record binary file format, which allows mixing integers, reals, date/time and variable length strings into variable length records. Data values are transferred as transferred back and forth as long byte sequences. That is, there is no parsing to maximize throughput. This packed record binary file is the basic mechanism to transfer data in both directions. Our fastest data transfer algorithms are programmed in the C language and they are exposed as R functions, to provide intuitive extensibility and interoperability (R), and maximum processing speed (C).

We consider two alternatives to program data transfer between both systems: (1) programmed in the R language; (2) programmed in the C language. In alternative (1) data mapping and building data structures is done entirely with existing R data structures, operators and functions. That is, nothing is programmed in the C language and therefore it is elegant and intuitive for analysts to customize. The caveat if alternative (1) is low speed. On the other hand, in alternative (2) we develop new R functions whose source code is programmed in the C language. That is, these functions directly access R internal data structures, especially data frames (somewhat similar to a table) and matrices (multidimensional array). Such data structures are converted to tabular form, as explained below. In alternative (2) function calls are intutitive to the analyst, but the source code is difficult to extend and customize. We emphasize that both alternatives execute within the R runtime in the R language native data structures, with dynamic memory.

Each alternative has different application scenarios. Alternative (1) (R language) is necessary to export machine learning models, which are composed of vectors, matrices and associated statistical metrics. That is, machine learning models cannot be directly converted to relational tables in SQL. Instead, the DBMS can store them as binary objects that can be accessed calling external C functions (i.e. cannot be called in SQL). Notice machine learning models are small: they take little space in main memory. In addition, the R language can provide a flexible way to transform small data sets exploiting R mathematical operators and functions. On the other hand, our system offers R data transfer functions programmed in the C language (alternative (2)). Evidently, the C language is more efficient, especially for large data sets and streams. However, C data transfer functions are tailored to data frames. Therefore, they cannot compute models with R language statements and therefore such C functions are not adequate to transfer models from R to SQL. In summary, data sets are transferred in both directions between the R runtime and the DBMS but machine learning models are transferred only in one direction (from R to the DBMS).

To round up this section, we provide a brief complexity analysis. For a data set having p attributes and n rows the time complexity of data transfer algorithms is linear $O(pn)$ because each table row (SQL) or data frame row (R) is accessed once and each value is touched once as well. In the specific case that the data set is a matrix with d dimensions (all p attributes are real numbers) and n points time complexity is $O(dn)$. Space complexity is the same, but we emphasize the DBMS and R run in different processes: they do not share memory. Therefore, data transfer is required and space consumption doubles. Finally, models consist of a collection of matrices whose size is generally $O(d)$ or $O(d^2)$. That is, they are much smaller than $O(dn)$.

3.4 Calls in Both Directions

We proceed to explain processing in more technical depth.

R calling SQL (evaluate SQL query)

Since R has a flexible script-based runtime and SQL queries in the DBMS are compiled to an executable C program it is not necessary to develop specialized

C code to call an SQL query: the SQL query is simply called with a system command call. Transferring data from the evaluated SQL query to R is achieved via the packed record binary format that is converted to data frame format (column-oriented) and then incrementally transferred to a data frame in RAM (via Unix pipes). This format resembles a big network packet, with a header specifying fields and their sizes, followed by a payload with the sequence of packed records. Notice that since in the DBMS strings have highly variable length then records also have variable length. Therefore, conversion and transfer record by record is mandatory (instead of block by block), but it is efficiently done in RAM, always moving byte sequences. When the output SQL table does not fit in RAM the data set can be processed in a block-by-block fashion in R; the drawback is that most existing R functions assume the entire data set is used as input and therefore they must be reprogrammed. When the algorithm behind the R function is incremental the respective R function is called on each block. This is a common case for transforming columns of a data set with mathematical functions, computing models with gradient descent or when partial data summaries can be merged at the end. Otherwise, when there is no incremental algorithm multiple models must be compared or merged, which is more difficult to do. A data frame containing only real numbers can then be converted to matrix. That is, it is feasible to call most R functions using a data frame or a matrix as input. Further math processing happens in R and in general R mathematical results (models, a set of matrices, diagnostic statistics) are locally stored in R. However, if the output of an R program is a data frame, preferably with a timestamp attribute, it can be converted to our packed binary format and then loaded back into the DBMS.

We list the main programming steps for the R analyst: (1) execute SQL query (which must have a time range); (2) transfer the SQL result table to one R data frame with a simple R variable assignment; (3) call R function on either: (a) entire data frame (once when result table fits in RAM, common case); (b) with a block-based algorithm (iteratively, less common case). Complex R statistical results cannot imported back into the DBMS, due to R being a more general language and its functional computation model. The main reasons behind this limitation are: models are composed of matrices, vectors and associated metrics, not flat tables like SQL; the need to incorporate time ranges on every result so that they become streams as well. However, a data frame containing atomic values can be be converted to an SQL table, but this scenario makes more sense to be managed by the DBMS, as explained below.

SQL calling R (evaluate R expression or call function)

SQL is neither a flexible nor an efficient language to manipulate data structures in main memory, but it offers UDFs programmable in C++, which can be easily called in a SELECT statement. On the other hand, the most flexible mechanism to call R to perform low-level manipulation of data is to embed R code inside C (or C++) code. Since UDFs are C++ code fragments plugged into the DBMS that isolate the programmer from the internals of physical database operators we use them as the main programming mechanism to call R, bypassing files and communication protocols. Specifically, calling R from the UDF C++ code is achieved by building temporary C++ vectors and then converting the set of C++ vectors into an R matrix. Notice we do not convert SQL records to data frame format in R because we assume the R function to call takes a matrix as input, the most useful case in practice. We should mention that directly moving data from an SQL table to an R data frame in embedded R code is significantly more involved to program, but not faster than matrices. R results can be further processed in C++ inside the DBMS and potentially be imported back into a table. Only R results that are a data frame can be transferred back into some SQL table. In general, there exist materialized views which have a dependence on this temporary table. From a query processing perspective when the R result is a data frame the DSW can treat R functions as table user-defined operators, where the size of the result can be known or bounded in advance.

We summarize the main programming steps in the UDF C++ for the SQL developer: (1) Include R header files; (2) load our R library; (3) setup parameters for R function; (4) build matrix in the UDF aggregation phase (row by row, but in RAM); (5) call R function in UDF final phase, (6) write R function results back into the DBMS either as: vector or matrix (accessible via special SQL functions) or table (accessible via SQL queries) when the result is a data frame.

3.5 Examples

We discuss typical analytic examples on network data. These examples illustrate two different needs: (1) an analyst, with basic SQL knowledge, just wants to extract some relevant data from the DBMS to perform compute some machine learning model. (2) a BI person, with advanced SQL and data cubes knowledge, but basic statistical background, wants to compute some mathematical transformation on the data set that is cumbersome or difficult to do in SQL.

R calling SQL: The analyst writes several SQL queries in a script to build a data set extracted by selecting records with a time window and then aggregating columns to create variables for statistical analysis. Then this data set is ideal to be analyzed by R to get descriptive statistics like the mean or standard deviation for numeric variables and histograms for numeric or discrete variables. After the data set is well understood the analyst can exploit R to compute a predictive model such as linear regression (to predict a numeric variable) or classification (to predict a discrete variable). These tasks boil down to writing an SQL script, starting the R environment, sending the SQL script to the DBMS for evaluation, transferring the final SQL table into a data frame and then analyzing the R data frame as needed. Notice that the output of these calls cannot be sent to the DBMS since they are collection of diverse vectors, matrices, arrays and associated statistics and diagnostic metrics.

SQL calling R: In this case there is an experienced SQL user who needs to call R to exploit some complicated mathematical function. A first example is getting the covariance or correlation matrix of all variables in the data set. Many insights are derived from these matrices. Moreover, these matrices are used as input to multidimensional models like PCA. Assume the user builds the data set with SQL queries as explained above, but the user wants to store the correlation matrix in the DBMS. To accomplish this goal, the user simply needs to create a "wrapper" aggregate UDF that incrementally builds a matrix, row by row. The aggregation phase reads each record and converts it to a vector in RAM. After the matrix is built R is called in the final phase of the UDF. At the end, the correlation matrix is locally stored in the DBMS to be consumed by a C++ program using our vector/matrix library. A second example is analyzing a stream as a time series and smoothing the time series in order to visualize it or analyze it. In this case the user does want to store the smoothed time series back into the DBMS. In more detail, the user wants to call R to solve the Fast Fourier Transform to find the harmonic decomposition of the time series and identify its period. Once the period is known values are averaged with a moving time window. The net result is a time series that is easier to interpret because it has less noise and a periodic pattern has been identified. Assuming the input table has a timestamp and some numeric value an aggregate UDF builds a data frame in the aggregation phase and then it calls R in the final phase to get an output data frame. This "cleaned" time series can be transferred back into the DBMS as a streaming table.

4 Experimental Evaluation

We did not conduct a detailed benchmark of STAR at AT&T due to two main reasons: (1) STAR can work with any DBMS supporting SQL and materialized views. (2) TidalRace, our current DBMS, works with confidential data whose characteristics cannot be disclosed. Instead, we focus on understanding STAR's ability to analyze high-volume streams with low-end hardware (i.e. under pessimistic conditions).

4.1 Hardware and Software

Our benchmark experiments were conducted on a rack server with a Quad-core CPU running at 2.153 GHz (i.e. 4 cores), 4GB RAM and 1 TB disk. As explained above, STAR was programmed in the R and C languages, providing an API to transfer data and make function/query calls in both directions.

In order to test correctness of results we performed full bi-directional data transfers with large data sets (tens of attributes, millions of records): (1) exporting an SQL table to an R data frame and then exporting the R data frame back to the DBMS to another SQL table. (2) transferring a data frame to an SQL table and then exporting such SQL table back to R as another data frame. We did not test correctness of complex SQL queries or arbitrary R scripts computing models because we do not alter the results returned by each system. Since these tests are basically a Y/N check mark they are omitted.

Our STAR system works with any DBMS supporting SQL. The time to evaluate SQL queries will vary widely depending on the specific query, DBMS storage (e.g. row or column based), indexing data structures for sliding time windows and size of result table. On the other hand, the time to import data into a DBMS will vary widely depending on parallel processing, storage layout and file format. On the other hand, R functions to compute models (e.g. K-means, linear regression, PCA) take a few seconds working on our data set. We emphasize our packed record binary file allows processing as efficient as possible. It is understood R does not scale well to large n. Therefore, we focus on measuring time after the SQL query result is ready or before the packed binary file is imported into the DBMS.

4.2 Benchmarking Data Transfer

Table 1. Comparing languages: transfer time and throughput (10 mixed type attributes, time in secs).

n	R	C		
	bin	csv	bin	recs/sec
100k	60	1.1	0.011	9.1M
1M	604	9.5	0.096	10.0M
10M	na	98.1	0.968	9.7M

In general, network data streams have few columns (2-10), resembling normalized tables. Most stream data sets have at least one time attribute (typically a timestamp) and some measurement (count or real number). Additional attributes include geographical location, network connection information (source and/or destination), and device information (e.g. MAC, firmware version). Based on this motivation, we use a data set with 10 attributes, including a timestamp, two variable length strings and seven measurements selected from the KDD network intrusion data set obtained from the KDD Cup web site. Each record is about 60 bytes which is wide enough to trigger heavy I/O. Our goal is to compare speed and measure maximum throughput. Table 1 compares speed to transfer data from the DBMS to R, with data transfer programmed in R and data transfer programmed in tuned C code. We stopped execution at one hour. The R language is more than three orders of magnitude (1000 times) slower than C to process the packed binary file. The R language provides built-in routines to read CSV text files, programmed in the C language. In this case our packed binary file is two orders of magnitude faster (100 times). The last column in Table 1 highlights our system is capable of transferring 10M records/second between the DBMS and R, surpassing DBMS query processing speed and R mathematical speed in most cases (i.e. our system is not a bottleneck to process a large data set despite the fact it is strictly sequential and it reads/writes to secondary storage). We emphasize that any reasonably complex SQL query (mixing joins and aggregations) or R program working on a large data set with 10M records is likely to take a few seconds or minutes, even with parallel processing. Exporting a model takes only a fraction of one second (e.g. 0.1, 0.5 secs) for data sets with up to hundreds of dimensions after the model is computed (refer to Section 3.3 for an explanation on data set and matrix sizes). Therefore, we omit time measurements to evaluate specific SQL queries or to export a machine learning model (e.g. K-means clustering, PCA, linear regression, classification).

5 Conclusions

We presented a "low level connector" system to efficiently transfer data and enable calls between SQL and the R languages in both directions, thereby removing R main memory limitations, allowing SQL to perform mathemtical computations, achieving streaming speed and improving interoperability between both systems. Our system defends the idea of combining sequential processing in R with a streaming computation model, where the stream is either: a query result table coming from the DBMS or a transformed data set coming from R imported back into the DBMS. We introduced a packed binary file which allows efficient data transfer in both directions for data sets having fixed (integers, reals, date, timestamp) and variable length columns (strings). We provide functions that allow calling R functions from SQL and calling SQL queries from R. In addition, we provide external functions to transfer machine learning models from R to the DBMS, as objects. Benchmark experiments show data transfer functions programmed in C are orders of magnitude faster than functions programmed in R. However, such C functions can only convert data frames into SQL tables and vice-versa. That is, they cannot convert mathematical models, consisting of matrices, vectors and statistics, into SQL tables. On the other hand, functions programmed in the R language are slow to transfer data sets, but efficient and intuitive to export models from R to the DBMS. Despite its limitations, we believe R will remain as a major alternative to perform statistical and even more general mathematical processing on large data sets and streams. Our prototype is a step in that direction.

Our bidirectional transfer/call approach offers many research opportunities. Scaling R beyond RAM limits and exploiting parallel processing remain important research issues. Specifically, we want to develop incremental machine learning algorithms for large SQL tables that can call R mathematical operators and functions. Parallel processing in R challenging since its architecture is single-threaded, but matrix operators and numerical methods are highly parallel. At a more fundamental level we will keep studying how to transform SQL tables into matrices and vice-versa and how to exploit SQL queries on transformed data frames and matrices.

References

1. J. Cohen, B. Dolan, M. Dunlap, J. Hellerstein, and C. Welton. MAD skills: New analysis practices for big data. In *Proc. VLDB Conference*, pages 1481–1492, 2009.

2. C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: a stream database for network applications. In *Proc. ACM SIGMOD*, 2003.
3. S. Das, Y. Sismanis, K.S. Beyer, R. Gemulla, P.J. Haas, and J. McPherson. RICARDO: integrating R and hadoop. In *Proc. ACM SIGMOD Conference*, pages 987–998, 2010.
4. A. Ghazal, A. Crolotte, and R. Bhashyam. Outer join elimination in the Teradata RDBMS. In *DEXA Conference*, pages 730–740, 2004.
5. L. Golab, T. Johnson, J. Spencer Seidel, and V. Shkapenyuk. Stream warehousing with DataDepot. In *Proc. ACM SIGMOD*, pages 847–854, 2009.
6. J. Hellerstein, C. Re, F. Schoppmann, D.Z. Wang, E. Fratkin, A. Gorajek, K.S. Ng, and C. Welton. The MADlib analytics library or MAD skills, the SQL. *Proc. of VLDB*, 5(12):1700–1711, 2012.
7. D. Jemal, R. Faiz, A. Boukorca, and L. Bellatreche. MapReduce-DBMS: An integration model for big data management and optimization. In *Proc. DEXA Conference*, pages 430–439, 2015.
8. T. Johnson and V. Shkapenyuk. Data stream warehousing in Tidalrace. In *CIDR*, 2015.
9. C. Ordonez. Building statistical models and scoring with UDFs. In *Proc. ACM SIGMOD Conference*, pages 1005–1016, NY, USA, 2007. ACM Press.
10. C. Ordonez. Statistical model computation with UDFs. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 22(12):1752–1765, 2010.
11. C. Ordonez. Can we analyze big data inside a DBMS? In *Proc. ACM DOLAP Workshop*, 2013.
12. C. Ordonez and J. García-García. Vector and matrix operations programmed with UDFs in a relational DBMS. In *Proc. ACM CIKM Conference*, pages 503–512, 2006.
13. C. Ordonez and I.Y. Song. Relational versus non-relational database systems for data warehousing. In *Proc. ACM DOLAP Workshop*, 2010.
14. M. Stonebraker, D. Abadi, D.J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. MapReduce and parallel DBMSs: friends or foes? *Commun. ACM*, 53(1):64–71, 2010.
15. M. Stonebraker, P. Brown, D. Zhang, and J. Becla. SciDB: A Database Management System for Applications with Complex Analytics. *Computing in Science and Engineering*, 15(3):54–62, 2013.
16. M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *HotCloud USENIX Workshop*, 2010.
17. J. Zubcoff and J. Trujillo. Extending the UML for designing association rule mining models for data warehouses. In *Proc. DaWaK*, pages 11–21. Springer, 2005.