SQL/MapReduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions

Eric Friedman
Aster Data Systems
eric@asterdata.com

Peter Pawlowski
Aster Data Systems
peter@asterdata.com

John Cieslewicz
Aster Data Systems
johnc@asterdata.com

ABSTRACT

A user-defined function (UDF) is a powerful database feature that allows users to customize database functionality. Though useful, present UDFs have numerous limitations, including *install-time* specification of input and output schema and poor ability to parallelize execution. We present a new approach to implementing a UDF, which we call SQL/MapReduce (SQL/MR), that overcomes many of these limitations. We leverage ideas from the MapReduce programming paradigm to provide users with a straightforward API through which they can implement a UDF in the language of their choice. Moreover, our approach allows maximum flexibility as the output schema of the UDF is specified by the function itself at *query plan-time*. This means that a SQL/MR function is polymorphic. It can process arbitrary input because its behavior as well as output schema are dynamically determined by information available at query plan-time, such as the function's input schema and arbitrary user-provided parameters. This also increases reusability as the same SQL/MR function can be used on inputs with many different schemas or with different user-specified parameters.

In this paper we describe the motivation for this new approach to UDFs as well as the implementation within Aster Data Systems' nCluster database. We demonstrate that in the context of massively parallel, shared-nothing database systems, this model of computation facilitates highly scalable computation within the database. We also include examples of new applications that take advantage of this novel UDF framework.

1. INTRODUCTION

The analysis of increasingly large amounts of data is central to many enterprises' day-to-day operations and revenue generation. Today, even small enterprises are collecting terabytes of data. Analyzing that data in an effective, efficient manner can be key to their success.

Relational databases present SQL as a declarative interface to manipulate data. Relational query processing within relational databases often falls short of this task. Analysts feel that SQL is too limiting for the types of queries they want to write that would extract value from the data, while others who are less familiar with declarative SQL want to query the data using procedural languages that they are more proficient in. Finally, relational database implementations have imperfect query optimizers that sometimes make poor choices and do not encapsulate domain-specific optimization choices. On big data, these imperfect choices are often very costly, causing queries to fail (e.g., run out of temporary space) or to continue to run for long periods of time, consuming valuable resources.

To address these issues, many relational databases support User-Defined Functions (UDFs) in which a developer can implement tasks using a procedural language. Unfortunately, the traditional UDF framework has been designed for a single database instance, with parallelism added as an afterthought, if at all. This represents an increasingly significant shortcoming, since growing data sizes demand a parallel approach to data processing and management across hundreds of database servers.

In this paper, we introduce SQL/MapReduce (SQL/MR) as a new UDF framework that is inherently parallel, designed to facilitate parallel computation of procedural functions across hundreds of servers working together as a single relational database. We present an efficient implementation of the SQL/MR framework in a massively-parallel relational database based on our experience of providing SQL/MR as part of the Aster Data Systems nCluster shared-nothing relational database. We present examples of applications that have been made possible by SQL/MR, and we include experimental results that demonstrate the efficiency gains from SQL/MR over pure SQL.

The MapReduce programming framework by Dean and Ghemawat [7] enables parallel procedural computation across hundreds of servers. The framework is designed to work on commodity hardware and emphasizes fault tolerance, allowing tasks to be computed even if certain function invocations have failed. The framework assumes a distributed file system in which data is managed in files, and the framework manages parallelization of computation on that data.

The power of a MapReduce programming framework is amplified within the context of a massively-parallel SQL database. The SQL/MR combination is extremely powerful—it leverages SQL to perform relational operations efficiently, leaving non-relational tasks and domain-specific optimizations to procedural functions; it ensures consistency of com-

VLDB '09 August 24-28, 2009, Lyon, France.

putations by guaranteeing that functions see a consistent state of data; it enables a cost-based query optimizer to make execution decisions that rely on data statistics instead of "create-time" intuition; and it enables non-developers to assemble their own queries using higher-level BI tools.

We designed the SQL/MR framework to be a next generation UDF framework in which functions are (1) self-describing and dynamically polymorphic – this means that SQL/MR function input schemas are determined implicitly at query time and output schemas are determined programmatically by the function itself at query time, (2) inherently parallelizable – whether across multi-core machines or massively-parallel clusters, (3) composable because we define their input and output behavior to be equivalent to a SQL subquery (and hence a relation), and (4) equivalent to subqueries, ensuring that the system can apply normal, relational cost-based static optimization and dynamic reoptimization techniques.

Our implementation of SQL/MR enables functions to (1) manage their own memory and file structures, and (2) easily include third-party libraries that can be used to reduce implementation effort, while ensuring that (3) function processes are contained within a sandbox, significantly reducing the likelihood that a run-away function can damage the system. Our model is compatible with a host of programing languages, including managed languages (Java, C#), native languages (C, C++), and scripting languages (Python, Ruby).

Because of the above features, SQL/MR functions can be implemented as true library functions that operate on arbitrary input where their specific behavior is determined at query time based on the context in which they are used. This allows rich functions to be developed by experts and then reused by others on diverse workflows without changing any code. This framework and implementation make Aster *n*Cluster an application-friendly database.

The rest of the paper is organized as follows. We present related work in Section 2. The specifics of SQL/MR syntax and implementation is presented in Sections 3 and 4. In Section 5, we demonstrate some examples of SQL/MR functions. Experimental results demonstrating the scalability and performance of SQL/MR functions are presented in Section 6. We conclude in Section 7.

1.1 SQL/MapReduce

SQL/MR allows the user to write custom-defined functions in any programming language and insert them into queries that otherwise leverage traditional SQL functionality. A SQL/MR function is defined in a manner similar to MapReduce's map and reduce functions that enable parallel data processing across a cluster, but in the case of SQL/MR, the functions operate in the context of a database.

In many key ways, SQL/MR offers benefits beyond those offered by UDFs. Most importantly, SQL/MR functions are *parallel by default*. As we will discuss in Section 3, the execution model for SQL/MR functions (enabled by their MapReduce-influenced API) is inherently parallel. Queries are executed in parallel across large clusters, ensuring the system's capability scales linearly with increasing data size. We will demonstrate this in Section 6. SQL/MR functions are also *dynamically polymorphic* and *self-describing*. In practice, this means that SQL/MR function input schemas are determined implicitly at query time and output schemas

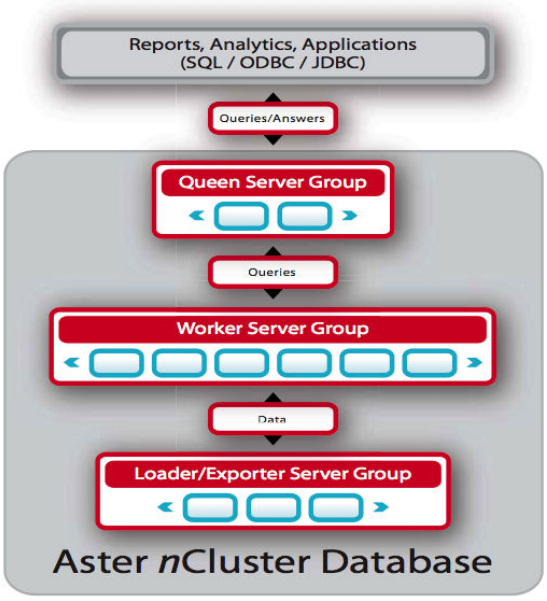

Figure 1: A schematic of the *n*Cluster database. System and query coordination are performed by *queen* nodes. Data is stored on *worker* nodes where query processing also occurs in parallel. Data loading can be accelerated with optional *loader* nodes.

are determined programmatically by the function itself at query time. Additionally, custom argument clauses may be included in the query-time invocation of the SQL/MR function, allowing further dynamic control of function behavior and schema. Because of these features, SQL/MR functions can be implemented as true library functions that operate on arbitrary input where their specific behavior is determined at query time based on the context in which they are used. This allows sophisticated analytics code to be developed by experts and then reused by others on diverse workflows without changing any code. The specifics of SQL/MR syntax and implementation are presented in Sections 3 and 4.

We will demonstrate some of the advantages of SQL/MR with the example of clickstream sessionization. More examples can be found in Sections 5 and 6.

1.2 Clickstream Sessionization

When analyzing user clicks on a website, a common analytic task is to divide a user's clicks into *sessions*. A session is a period of user activity on the website. A session is defined to include all of a user's clicks that occur within a specified range of time of one another, as defined by a timeout threshold. Figure 2 demonstrates sessionization on a table of clicks. This simple click table contains only the `timestamp` of the click and the `userid` associated with the click. In the resulting table (Fig. 2b), which is shown partitioned by `userid` for clarity, each pair of clicks for which the elapsed time between the clicks is less than 60 seconds are considered to be part of the same session.

Sessionization can be accomplished using SQL, but SQL/MR makes it easier to express and improves its performance. A SQL/MR function for sessionization, requires only one pass over the clicks table, whereas a SQL query will require an

time-stamp	userid
10:00:00	238909
00:58:24	7656
10:00:24	238909
02:30:33	7656
10:01:23	238909
10:02:40	238909

(a) Raw click data

time-stamp	userid	session
10:00:00	238909	0
10:00:24	238909	0
10:01:23	238909	0
10:02:40	238909	1
00:58:24	7656	0
02:30:33	7656	1

(b) Click data with session information

Figure 2: An example of sessionization: table (a) contains raw clicks, and (b) contains clicks grouped by `userid` and augmented with session number based on a session timeout of 60 seconds.

```
SELECT ts, userid, session
FROM sessionize (
      ON clicks
      PARTITION BY userid
      ORDER BY ts
      TIMECOLUMN ('ts')
      TIMEOUT (60)
      );
```

Figure 3: Using SQL/MR sessionize in a query.

expensive self-join. Furthermore, the SQL/MR function's dynamic polymorphism allows it to be reused to compute session information over tables of any schema. In this way, a sessionization SQL/MR function becomes a reusable library routine that any analyst can use.

We first show the use of the *sessionize* SQL/MR function in a query over the clicks table of Figure 2 and then describe the implementation of the function itself.

Figure 3 shows the use of SQL/MR sessionize in a query over the clicks table. We partition by the `userid` in order to group each user's clicks. Each partition is then ordered by the timestamp. There are two custom argument clauses, `TIMECOLUMN` and `TIMEOUT`. During initialization the `TIMECOLUMN` argument clause specifies which input attribute will be examined to determine membership in a session. The value of the `TIMEOUT` argument is also stored so that it can be used during execution to determine if a session boundary has been found. During SQL/MR initialization, the sessionize function also specifies its output to be the input schema with the addition of a `session` attribute of type integer. This behavior allows the sessionize SQL/MR function to be used with any input schema. For even greater flexibility, one could add an optional third custom argument clause, `OUTPUTALIAS`. At query time, a user could then specify the name of the new column for session information.

The implementation of the sessionize SQL/MR function is straightforward. It is implemented as a partition-function such that, when invoked, the input is partitioned by the attribute that identifies *whose* sessions we wish to identify, e.g. `userid`. Within each partition, the input must also be ordered by the attribute that determines session boundaries, e.g. `ts`. For each partition processed, a session counter is initialized to 0. In a single pass over each partition, SQL/MR sessionize compares the `TIMECOLUMN` of subsequent tuples to see if they are within the `TIMEOUT` of each other. If so, then both tuples' session number is set to the current session count, otherwise the session counter is incremented and the newest tuple's session number is assigned the new count. We show source code implementing the function when we describe the SQL/MR programming interface in Section 3.3.

2. RELATED WORK

User-defined functions and procedures are longstanding database features that enable database extensibility. Just as user-defined types (UDT) allow customization of *what* a database stores (e.g., [19]), user-defined functions allow customization of *how* a database processes data [18, 11, 6, 20, 21, 22, 4, 12]. There has been significant research related to efficiently using UDFs within database queries both in terms of optimization and execution, e.g., [6, 11, 10]. But most of this work has examined the context of a single database instance rather than parallel execution of UDFs over a shared-nothing parallel database.

There has been some work related to parallel processing of user-defined aggregates, scalar functions [14], and table operators [15]. Traditional user-defined aggregates can be executed in parallel by having the user specify local and global *finalize* functions [14]. In this way, partial aggregates are computed in parallel and then finalized globally. Parallel scalar functions that have no state are trivially parallelized. Another class of scalar functions, such as moving averages, require the maintenance of some state, but if the user explicitly specifies a type of partitioning, then the system can partition the data and perform the computation in parallel [14]. Going a step further, [15] proposes user-defined table operators that use relations as both input and output, which is similar to the input and output to SQL/MR functions. The user-defined table operators require the user to statically pick a partitioning strategy to enable parallelism as well as inform the system how the operator may be used. Our proposed SQL/MR functions do not require explicit or static choices about partitioning or use cases – that information is determined at query plan-time based on the context in which the SQL/MR function is used.

The idea of a table function is present in SQL as well, and support for user-defined table functions is present in most commercial databases (e.g., [12], [18], [16]). Oracle and SQL Server additionally support table-valued parameters. The default programming model in these systems is non-parallel, so functions are written assuming they will receive all of the input data. Some implementations allow the function to be marked for explicit parallelization. For instance, Oracle table functions have an optional `PARALLEL_ENABLE` clause at create-time that indicates that parallelization is permissible and also how input rows should be partitioned among concurrent threads. By contrast, the programming model for SQL/MR functions implies execution is parallel by default. Further, the `PARTITION BY` clause in SQL/MRthat specifies how input rows should be grouped is a semantic part of the query—rather than a function create-time option—so a function does not need to be re-created (by a DBA or end-user) simply to group input data in a different manner.

Some systems provide support for polymorphic (context-dependent) function output schemas. This is more flexible than the typical approaches to UDFs that specify the function input and output schema statically at create time. For instance, Oracle has a generic data type called `ANYDATASET`

that can be used at function creation time to defer a decision on a particular data type; at query plan-time, the function will be asked to describe the type. This idea appears also in Microsoft's SCOPE data processing system [5], in particular to support extraction of structured data from flat files. SQL/MR functions fully embrace and extend this approach: they avoid the need for create-time configuration of a function, allow polymorphism of the input schema, and also enable the optional use of custom argument clauses (more on these in Section 3) to provide additional query plan-time parameters to the function. These query-time customization features allow SQL/MR functions to operate over a wide range of inputs and behave more like general purpose library functions than conventional UDFs.

Recently, interest in distributed parallel data processing frameworks has increased. Examples include Google's MapReduce [7], Microsoft's Dryad [13], and the open source Hadoop project [1]. These frameworks are powerful tools for parallel data processing because users need only to implement well-defined procedural methods. The framework then handles the parallel execution of those methods on data distributed over a large cluster of servers. A key advantage of these systems is that developers write simple procedural methods that are then applied in parallel using a well-defined data partitioning and aggregation procedure. A disadvantage of these frameworks is that developers must often write code to accomplish tasks that could easily have been expressed in SQL or another query language. In particular, code reuse for *ad hoc* queries is limited as there is no higher-level language than the procedural code.

Higher level systems for MapReduce-like infrastructures have been proposed, including Pig [17], Hive [2], and SCOPE [5]. Both combine the high-level declarative nature of SQL while also exposing the lower level procedural, parallel capabilities of a MapReduce framework. While Hive and SCOPE seek for SQL compatibility or at least familiarity, to integrate with MapReduce code, these systems introduce significant new syntax to normal SQL; for instance, in addition to the usual `SELECT`, SCOPE adds `PROCESS`, `REDUCE`, and `COMBINE`. By contrast, SQL/MR introduces a small amount of new syntax and semantics by representing parallel functions as a table. Overall, these languages represent good improvements to MapReduce, by introducing a form of declarative query language. We have taken the complementary approach of enhancing a massively-parallel, SQL-compliant database with a MapReduce-like programming model. This approach enables SQL/MR functions to leverage the structure of data that is inherent in relational databases via schemas, and enables optimization by cost-based query optimizers that leverage relational algebra and statistics for query-rewriting.

3. SYNTAX AND FUNCTIONALITY

In this section we present the syntax of invoking our SQL/MR functions from within a standard SQL query (Section 3.1), the execution model provided by SQL/MR functions (Section 3.2), and the API provided for implementing SQL/MR functions (Section 3.3). We also discuss the installation of SQL/MR functions (Section 3.4) and the use of other files during SQL/MR execution (Section 3.5).

3.1 Query Syntax

The syntax for using a SQL/MR function is shown in Figure 4. The SQL/MR function invocation appears in the SQL `FROM` clause and consists of the function name followed by a parenthetically enclosed set of clauses. The first, and only strictly required clause, is the `ON` clause, which specifies the input to this invocation of the SQL/MR function. The `ON` clause must contain a valid query. A table reference is also valid, but can really be thought of as syntactic sugar for a query that selects all columns from the specified table. When a query is used, it must be contained within parentheses just as a subquery appearing in the `FROM` clause must be parenthesized. It is important to note that the input schema to the SQL/MR function is specified implicitly at query plan-time in the form of the output schema for the query used in the `ON` clause.

```
SELECT ...
FROM functionname(
   ON table-or-query
   [PARTITION BY expr, ...]
   [ORDER BY expr, ...]
   [clausename(arg, ...) ...]
   )
...
```

Figure 4: SQL/MR function query syntax.

3.1.1 Partitioning

The next clause in the SQL/MR invocation is `PARTITION BY`, which specifies a comma-separated list of expressions used to partition the input to the SQL/MR function. These expressions may reference any attributes in the schema of the query or table reference specified by the `ON` clause. Section 3.3 will describe the role of the `PARTITION BY` clause in greater detail.

3.1.2 Sorting

The `ORDER BY` clause follows the `PARTITION BY` clause and specifies a sort order for the input to the SQL/MR function. The `ORDER BY` clause is only valid if a `PARTITION BY` clause has also been used. The `ORDER BY` clause may reference any attributes in the schema of the query or table reference contained in the `ON` clause and accepts a comma-separated list of any expressions that are valid in a standard SQL `ORDER BY` clause. The data within each unique partition specified by the `PARTITION BY` clause will be sorted independently using the sort order specified in the `ORDER BY` clause.

3.1.3 Custom Argument Clauses

Following the `ORDER BY` clause, the user may add any number of custom argument clauses. The form of a custom argument clause is the clause name followed by a parenthesized list of comma-separated literal arguments. The SQL/MR function will receive a key-value map of these clause names and arguments when it is initialized. The use of custom argument clauses allows query-time customization of SQL/MR functionality and is one way in which SQL/MR enables dynamic polymorphism.

3.1.4 Usage as a Relation

The result of a SQL/MR function is a relation; therefore, that result may participate in a query in exactly the same way as any other valid table reference or subquery that can

```
SELECT ...
FROM sqlmr1(
     ON sqlmr2(
        ON some_table
        PARTITION BY ...
        )
     PARTITION BY ...
     ORDER BY ...
     );
```

Figure 5: Nesting of SQL/MR functions.

also appear in the `FROM` clause of a query. A SQL/MR function need not be the only expression in the `FROM` clause. For instance, the results of two SQL/MR functions may be joined to each other or to a table or subquery. Furthermore, because a SQL/MR function results is a table and a SQL/MR function takes a table as input, SQL/MR functions may be nested directly as shown in Figure 5.

3.2 Execution Model

The execution model provided by SQL/MR functions is a generalization of MapReduce [7]. To use terms from MapReduce, a SQL/MR function can be either a *mapper* or a *reducer*, which we call a *row function* or *partition function*, respectively. SQL/MR functions may implement both interfaces if both modes of operation make sense for the function. Because of the integration of SQL/MR with SQL, it is trivial to chain any combination of map and reduce SQL/MR functions together as shown in Figure 5. To compare with MapReduce, SQL/MR allows an arbitrary number and ordering of map and reduce functions interspersed within a SQL query, whereas MapReduce allows only one map followed by one reduce.

The SQL/MR execution model is designed for a massively parallel database and therefore strives to be parallel by default. Instances of the SQL/MR function will execute in parallel on each node in the parallel database, just as map and reduce tasks execute in parallel across a cluster in the MapReduce framework. The number of instances of the SQL/MR function per worker node is not fixed. Each instance sees a unique set of input rows, that is, each row is processed by only one instance of the SQL/MR function. The definitions of row and partition functions ensure that they can be executed in parallel in a scalable manner. Even in a single node database, the SQL/MR framework is still useful because it provides dynamically polymorphic and self-describing UDFs that may be parallelized across multiple processor cores.

We now describe row and partition functions and show how their execution models enable parallelism:

- **Row Function** Each row from the input table or query will be operated on by exactly one instance of the SQL/MR function. Semantically, each row is processed independently, allowing the execution engine to control parallelism, as described in Section 4. For each input row, the row function may emit zero or more rows. Row functions are similar to map functions in the MapReduce framework; key uses of row functions are to perform row-level transformations and processing.
- **Partition Function** Each group of rows as defined by the `PARTITION BY` clause will be operated on by exactly one instance of the SQL/MR function, and that function instance will receive the entire group of rows together. If the `ORDER BY` clause is also provided, the rows within each partition are provided to the function instance in the specified sort order. Semantically, each partition is processed independently, allowing parallelization by the execution engine at the level of a partition. For each input partition, the SQL/MR partition function may output zero or more rows. Partition functions is similar to a reduce function in MapReduce; we call it a partition function to emphasize its use for group-wise processing, as important uses do not actually reduce the size of the data set.

3.3 Programming Interface

In this section we will describe the programming interface. Using our running example of sessionization, Figure 6 shows the Java class that implements the SQL/MR sessionize function.

3.3.1 Runtime Contract

We chose the metaphor of a contract to facilitate a SQL/MR function's self-description. At plan-time, the *n*Cluster query planner fills in certain fields of a runtime contract object, such as the names and types of the input columns and the names and values of the argument clauses. This incomplete contract is then passed to the SQL/MR function's initialization routine at plan-time.

The constructor must complete the contract by filling in additional fields, such as the output schema, and then calling the `complete()` method. All instances of the SQL/MR function are required to abide by this contract, so the contract's completion should only involve deterministic inputs.

With a traditional UDF, there is a kind of contract as well: when a function is installed, the types of its input arguments and return value must be explicitly declared (in the `CREATE FUNCTION` statement). This is done by the end-user or database administrator. By contrast, with a SQL/MR function, not only is the function self-describing (requiring no configuration during installation) but the plan-time negotiation of the contract allows the function to alter its schema dynamically, adding significant flexibility to create reusable functions. We provide more examples of this in Section 5.

Help information. Because contract negotiation, and therefore output schema definition, occurs at query plan-time, it is useful to provide the writer of a query a means to discover the output schema of a particular SQL/MR invocation. We accomplish this by leveraging the self-describing nature of SQL/MR functions as well as the deterministic property of contract negotiation described above. Further, just as many command line tools have a "help" option, developers of SQL/MR functions provide help information via a help API. This includes information such as required or optional argument clauses as well as the output schema given a particular input schema and set of argument clauses.

Argument clause validation. SQL/MR automatically ensures that the query specifies argument clauses for the function that are compatible with its implementation: if an argument clause is provided but unused, or if the function

attempts to access an argument clause that has not been provided, an error message is directed to the user. For example, both of the argument clauses specified in the query shown in Figure 3 are used by the the Sessionize constructor in Figure 6. To enable optional argument clauses, a SQL/MR function's constructor can test for the presence of a specific argument clause.

3.3.2 Functions for Processing Data

The most basic aspects of the API are the `OperateOnSomeRows` and `OperateOnPartition` methods, which are part of the row and partition function interfaces, respectively. These methods are the mechanism of invocation for a SQL/MR function. The function is given an iterator to rows over which it is being invoked, along with an emitter object for returning rows back into the database. The `OperateOnPartition` method also includes a `PartitionDefinition` object, which provides the values of the `PARTITION BY` expressions. This is useful as the columns used to compute these values might not be in the function's input.

Figure 6 shows the implementation of the `OperateOnPartition` function for the sessionization SQL/MR function. Each output row is constructed from the entire input row plus the current session ID. Note that the output attributes are added to the output emitter in left to right order.

3.3.3 Combiner Functions

One of the optimizations in Google's MapReduce implementation [7] is support for combiner functions. Combiner functions decrease the amount of data that needs to be moved across the network by applying a function to combine rows in local partitions. Use of a combiner is a pure optimization; it does not affect the outcome of the final computation.

SQL/MR supports combining as an option in implementing a partition function. In some cases, network transfer is required to form input partitions for a partition function. If a partition function implements the optional interface for combining, the query planner may choose to invoke the combiner functionality prior to the network transfer, reducing the number of rows that need to be sent.

We consciously chose to make the combiner feature a detail of the partition function—rather than a separate kind of function—for usability reasons. From the perspective of the user writing a query with a partition function, there is no semantic difference if combining is performed. For this reason, we leave combining as an implementation detail that is considered by the SQL/MR function developer, but that is transparent to the user of the function.

3.3.4 Running Aggregates

SQL/MR defines a mechanism for computing a running SQL aggregate on data in a SQL/MR function. This allows a function to offer to its users a full set of familiar SQL aggregates with minimal effort. A function can request a new running aggregate by name and type (for example, `avg(numeric)`) and update the aggregate with new values, query the current result of the aggregate, or reset the aggregate. Aggregates over any data type may be requested dynamically, which is useful for polymorphic functions that may not be developed with particular data types in mind. Further, these aggregates match SQL semantics, which for some data types are subtle.

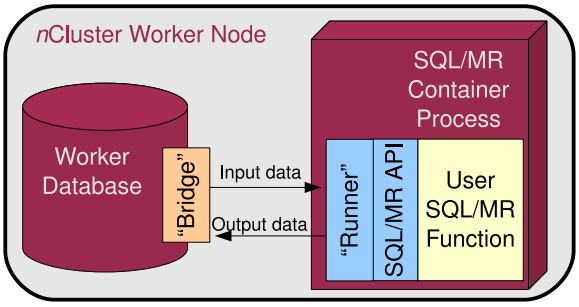

Figure 7: Diagram of the implementation of SQL/MR within the *n*Cluster database.

3.4 Installing a SQL/MR function

To use a SQL/MR function, it must be installed. We leverage the general ability to install files (described in Section 3.5) to load and manage the file containing executable code. Once installed, the system examines the file to determine that it is a function. Since functions are self-describing, no configuration or `CREATE FUNCTION` statement is required, and the SQL/MR function is immediately available for use in a query. Examining the file at install-time also reveals other static properties of the function, such as whether it is a row function or a partition function, the help information associated with the function, whether it supports combining partitions, and so on.

Function files may actually be a `.zip` archive containing a function file along with other, third-party libraries. These third-party libraries are made available to the function; for instance, in the case of Java, they are automatically included in the function's classpath. This has been useful for a variety of purposes: a linear algebra package for solving linear equations, a natural language processing library, and so on.

3.5 Installed Files and Temporary Directories

In order to facilitate the distribution of configuration files and other auxiliary file data, the system allows users to install arbitrary files in addition to function files. Installing a file replicates it at all the workers, making it available for reading by SQL/MR functions. Each function is also provided with a temporary directory, which is cleaned up after the function is finished and whose space usage is monitored during function execution.

We have found these capabilities useful in the distribution of configuration files, static data files for things like dictionaries, as well as the installation of binaries that a SQL/MR function can then execute. This last use case, in particular, demonstrates the focus on usability: it has enabled us in some cases to quickly push existing C binaries into the parallel execution environment without expending large amounts of time in turning these binaries into callable libraries with well-defined APIs.

4. SYSTEM ARCHITECTURE

In this section we first briefly present the system architecture of *n*Cluster (Section 4.1), a massively-parallel relational database system. We then describe how SQL/MR integrates into *n*Cluster (Section 4.2).

```
public class Sessionize implements PartitionFunction
{
   // Constructor (called at initialization)
   public Sessionize(RuntimeContract contract)
   {
      InputInfo inputInfo = contract.getInputInfo();

      // Determine time column
      String timeColumnName =
         contract.useArgumentClause("timecolumn").getSingleValue();
      timeColumnIndex_ = inputInfo.getColumnIndex(timeColumnName);

      // Determine timeout
      String timeoutValue =
         contract.useArgumentClause("timeout").getSingleValue();
      timeout_ = Integer.parseInt(timeoutValue);

      // Define output columns
      List<ColumnDefinition> outputColumns =
         new ArrayList<ColumnDefinition>();
      outputColumns.addAll( inputInfo.getColumns() );
      outputColumns.add(new ColumnDefinition("sessionid", SqlType.integer()));

      // Complete the contract
      contract.setOutputInfo( new OutputInfo(outputColumns) );
      contract.complete();
   }

   // Operate method (called at runtime, for each partition)
   public void operateOnPartition(
      PartitionDefinition partition,
      RowIterator inputIterator, // Iterates over all rows in the partition
      RowEmitter outputEmitter // Used to emit output rows
      )
   {
      int currentSessionId = 0;
      int lastTime = Integer.MIN_VALUE;

      // Advance through each row in partition
      while ( inputIterator.advanceToNextRow() )
      {
         // Determine if time of this click is more than timeout after the last
         int currentTime = inputIterator.getIntAt(timeColumnIndex_);
         if ( currentTime > lastTime + timeout_ )
            ++currentSessionId;
         // Emit ouput row with all input columns, plus current session id
         outputEmitter.addFromRow(inputIterator);
         outputEmitter.addInt(currentSessionId);
         outputEmitter.emitRow();
         lastTime = currentTime;
      }
   }

   // State saved at initialization, used during runtime
   private int timeColumnIndex_;
   private int timeout_;
};
```

Figure 6: Implementation of the reusable sessionize function using the SQL/MR Java API.

4.1 nCluster Overview

*n*Cluster [3] is a shared-nothing parallel database [8], optimized for data warehousing and analytic workloads. *n*Cluster manages a cluster of commodity server nodes, and is designed to scale out to hundreds of nodes and scale up to hundreds of terabytes of active data.

Query processing is managed by one or more *Queen* nodes. These nodes analyze client requests and distribute partial processing among the *Worker* nodes. Each relation in *n*Cluster is hash-partitioned across the Worker nodes to enable intra-query parallelism.

In addition to database query processing, automated manageability functionality in *n*Cluster ensures adding new machines and redistributing data is a one-click operation, and the system performs automatic fail-over, retry of queries, and restoration of replication levels after a node failure. These features are essential in a large cluster of machines, where failures of various kinds occur regularly.

4.2 SQL/MR in nCluster

The implementation of the SQL/MR framework in Aster *n*Cluster requires us to define the interactions of the SQL/MR function with the query planning and query execution frameworks of the relational database.

4.2.1 Query planning

SQL/MR functions are dynamically polymorphic, meaning that their input and output schemas depend upon the context in which they are invoked. We resolve the input and output schemas during the planning stages of the query—a task that is designated to the query planner at the Queen node.

The query planner receives a parse tree of the query. It resolves the input and output schemas of the SQL/MR functions in a bottom-up traversal of the parse tree. When a SQL/MR function is encountered in this traversal, the planner uses the already-defined schema of the input relations—along with the parsed argument clauses specified in the query for the function—to initialize the function by invoking the function's *initializer* routine. The *initializer* routine must decide the function's output columns that will be produced by the function's *runtime* routine during query execution. (In our Java API, the initializer routine corresponds to the constructor of a class implementing one of the row or partition function interfaces, while the runtime routine is the method defined by the interface.)

As described in Section 3.3.1, the metaphor for the function is one of a contract: the query planner provides some guarantees about the input and the function provides guarantees about its output, and both are promising to meet these guarantees during query execution. This negotiation allows the function to have a different schema in different usage scenarios—what we call dynamic polymorphism—while maintaining the property that the schema of a SQL query is well-defined prior to execution.

In addition to enabling dynamic polymorphism, this notion of a contract enables a rich integration with query planning. The developer of a function may be aware of certain properties of its execution. For instance, a function might emit rows in a certain order, pass through certain columns from the input to the output, be aware of statistical information about the output, and so on. The contract is a natural conduit for the function to provide this information to the query optimizer. The function can provide such information to the query planner during the invocation of its *initializer* routine during planning. Importantly from a usability perspective, SQL/MR does not require the end-user or database administrator to specify a variety of complicated `CREATE FUNCTION` clauses during function installation to inform the planner of such function properties. Instead, this information can be encoded by the function's developer and be encapsulated inside the function, which describes itself during query planning.

4.2.2 Query execution

SQL/MR functions are treated as an execution operator in the local Worker database: the rows of input to the function is provided from an iterator over the `ON` clause while their output rows are in turn provided into the next execution node up the execution tree. In the case of partitioned input, the rows are divided into groups; this may be done either by sorting or hashing the rows according to the values of the `PARTITION BY` expressions.

SQL/MR functions are executed in parallel across all nodes in *n*Cluster, as well as in parallel across several threads at each node. Since the MapReduce-based programming model is agnostic to the degree of parallelism, the system can control the level of parallelism transparently to utilize the available hardware. The SQL/MR framework simply instantiates several instances of the function, one on each thread. Input rows are distributed among the threads, and output rows are collected from all threads.

For a variety of reasons, we execute the threads of a SQL/MR function in a separate process from the local database process. Executing in a separate process allows the externally-developed SQL/MR function code to be effectively sand-boxed and controlled using typical operating system mechanisms—for fault isolation, scheduling, resource limitation, forced termination, security, and so on—without relying on any particular programming language runtime environment to provide such functionality. For instance, if an end-user or database administrator decides to cancel a query that is executing a function, we simply kill the process running it. This model has been key in effectively maintaining overall system health in the presence of user code. Isolating a function in a separate process allows us to both limit the damage it can do to the system, as well as manage scheduling and resource allocation, using existing operating system primitives.

Figure 7 shows a diagram of how SQL/MR is implemented within *n*Cluster. Within the worker database is a component we call the "bridge" which manages the communication of data and other messages between the database and the out-of-process SQL/MR function. In a separate process, the counterpart to bridge, the "runner" manages communication with the worker database for the SQL/MR function. An API is built on top of the runner with which users implement SQL/MR functions. This modularization makes it relatively easy to add support for additional programming languages to the SQL/MR framework.

5. APPLICATIONS

In this section we present examples of applications that can be implemented using the SQL/MR framework. We start with a simple example that compares and contrasts SQL/MR directly with MapReduce presented in [7].

5.1 Word Count

Since the publication of [7], performing a word count has become a canonical example of MapReduce, which we use here to illustrate the power of SQL/MR. In contrast to the MapReduce example, SQL/MR allows the user to focus on the computationally interesting aspect of the problem – tokenizing the input – while leveraging the available SQL infrastructure of perform the more pedestrian grouping and counting of unique words.

We have written a general purpose SQL/MR row function called *tokenizer* that accepts a custom argument clause to specify the delimiters to use. The output of tokenizer is simply the tokens.[1] The query containing the SQL/MR invocation groups its results by the token values and computes a `COUNT(*)` aggregate. The result of mixing SQL and SQL/MR is a more succinct word count function that leverages existing database query processing infrastructure:

```
SELECT token, COUNT(*)
FROM tokenizer(
      ON input-table
      DELIMITER(' ')
      )
GROUP BY token;
```

Not only is this simpler than a pure MapReduce implementation, but it allows the query optimizer to leverage existing parallel query execution optimizations for computing an aggregate in a distributed manner.

5.2 Analysis of Unstructured Data

SQL is generally ill-suited to dealing with unstructured data. However, SQL/MR enables a user to push procedural code into the database for transforming unstructured data into a structured relation more amenable for analysis. While such transformation is possible with traditional UDFs, the dynamic polymorphism of SQL/MR functions allows such a transformation to be significantly more flexible and usable.

Consider the `parse_documents` function shown below. It is designed to encapsulate a collection of metrics to be computed about a document. A user can specify particular metrics of interest via the `METRICS` argument clause, and the function will compute these metrics. Additionally, the output schema of the function will reflect the requested metrics. Note that these metrics can be computed with a single pass through the data, but the framework allows the flexibility to specify the metrics of interest in an ad-hoc way. This is a useful and usable way to wrap a library of text analytics code for reuse by an analyst in a variety of scenarios.

```
SELECT word_count, letter_count, ...
FROM parse_documents(
    ON (SELECT document FROM documents)
    METRICS(
       'word_count',
       'letter_count',
       'most_common_word',
       ...)
    );
```

[1]As implemented, the tokenizer creates tokens from all columns with character data in the input using the specified delimiter. Non-character columns are returned as whole tokens. One could easily extend the SQL/MR tokenizer to take an additional custom argument clause that specifies the input columns to tokenize.

5.3 Parallel Load and Transformation

SQL/MR functions can also be used to provide support for both reading from external sources. Consider the use case of hundreds of retail locations that send daily sales data in comma separated files back to the home office to be loaded into nCluster. The common solution is to use an external process to load the data. In nCluster, one can perform transformations inside the cluster using a SQL/MR function that takes as input a set of urls that identify the external files to load and an argument clause that defines the expected input schema and desired output schema. After being fetched and transformed by the SQL/MR function, the data is immediately available to participate in other query processing such as immediate filtering or aggregation. If the goal is to load the external table into nCluster, using a SQL/MR function for transformation is beneficial because it now runs in parallel within nCluster, leveraging the parallel computational power of all of the worker nodes and improving performance as the loading process now runs in the same location where the data will ultimately be stored. Because of the flexibility of SQL/MR functions, arbitrary source formats can be supported simply by writing the appropriate SQL/MR function that can then be used as a library function for all subsequent reading or loading of data from an external source.

5.4 Approximate Percentile

Computing exact percentiles over a large data set can be expensive, so we leveraged the SQL/MR framework to implement an approximate percentile algorithm. This allows parallel computation of percentiles if some amount of error is acceptable. This implementation also leverages SQL/MR's dynamic polymorphism to enable computation of approximate percentiles over a wide range of numeric types.

We implemented the distributed approximate percentile algorithm described in [9] as a pair of SQL/MR functions. To apply this technique, one specifies the percentile values desired and the maximum relative error e. The relative error is defined as follows: for each value v that the algorithm estimates as being in the n-th percentile, the real percentile of v is between $n-e$ and $n+e$. At a high level, the algorithm works by computing summaries of the data on each particular node, and then merging these summaries on a single node to compute the approximate percentiles. We implemented this algorithm with an `approximate_percentile_ summary` function that is invoked over all the relevant data on a particular node, outputting a summary. The summaries are then brought together at a single node by using a `PARTITION BY 1` construct,[2]where they are merged into a final output by the `approximate_percentile_merge` function. The output schema of `approximate_percentile_merge` consists of the input schema with a `percentile` column prepended.

6. EXPERIMENTAL RESULTS

The SQL/MR framework brings a lot of expressive power to relational databases. We showed in Section 5 that queries

[2]SQL/MR functions are designed to be parallel by default. However, there exist situations in which processing data serially is required. To accommodate these cases we allow a constant to appear in the `PARTITION BY` clause. This causes all input data to be collected on one worker node and then processed serially by the specified SQL/MR function. The user is warned that the SQL/MR function will not execute in parallel.

```
SELECT percentile, ...
FROM approximate_percentile_merge(
      ON approximate_percentile_summary(
            ON source_data
            RELATIVE_ERROR(0.5)
            DATA_COLUMN('values')
            )
      PARTITION BY 1
      PERCENTILES(25, 50, 75)
      );
```

Figure 8: Approximate percentile using SQL/MR.

that are difficult or impossible to express in traditional SQL (e.g., approximate medians) can be easily expressed in the SQL/MR framework using SQL/MR functions. In this section, we extend the argument that SQL/MR queries can result in faster implementations than a pure SQL query. Our experimental results show the following results:

- The SQL/MR queries exhibit linear scaling as the degree of parallelism is increased proportional to the size of data being queried.
- The ability of SQL/MR functions to manipulate their own data structures allows them to finish tasks in one pass over data that would have required multiple joins in a pure SQL query

We ran all experiments on an nClusterof x86 servers with two dual-core 2.33 Ghz Intel Xeon processors, 4GB of RAM, and eight 72GB SAS drives configured with RAID 0.

6.1 Clickstream Analysis

Web administrators often use clickstream logs to understand the behavior of their consumers so that they can make changes to their website structure to improve engagement metrics. For example, web advertisers often wish to know the average number of clicks between a user starting from the homepage of a particular publisher and then clicking on an advertisement. Web retailers are interested in knowing the average number of clicks between a user entering the site and purchasing an item. Web publishers are interested in knowing the average number of articles a person reads if she starts in the Politics section of the website before entering the Entertainment section of the website.

Given a relation `Clicks(user_id int, page_id int, category_id int, ts timestamp)` that stores information about a user, the page the user clicked and the time at which the user clicked that page, what is the average number of pages a user visits between visiting a page in category X and a page in category Y? We refer to the click in category X as the starting click and the click in category Y as the ending click. We generated a synthetic data set of clicks with a SQL/MR function which maps over a table of users, expanding each row into a set of clicks for that particular user. We generated 1000 clicks for each user with random values for the `ts`, `category_id`, and `page_id` columns (all chosen from a uniform distribution). There were fifty million rows per node.

To answer this question, we first wrote a pure SQL query, which is shown in Figure 9. The query works by first joining every click in category X with every click in category Y from

```
SELECT
   avg(pageview_count)
FROM
(
   SELECT
      c.user_id, matching_paths.ts1,
        count(*) - 2 as pageview_count
   FROM
      clicks c,
      (
         SELECT
            user_id, max(ts1) as ts1, ts2
         FROM
         (
            SELECT DISTINCT ON (c1.user_id, ts1)
               c1.user_id,
               c1.ts as ts1,
               c2.ts as ts2
            FROM
               clicks c1,clicks c2
            WHERE
               c1.user_id = c2.user_id AND
               c1.ts < c2.ts AND
               pagetype(c1.page_id) = 'X' AND
               pagetype(c2.page_id) = 'Y'
            ORDER BY
               c1.user_id, c1.ts, c2.ts
         ) candidate_paths
         GROUP BY user_id, ts2
      ) matching_paths
   WHERE
      c.user_id = matching_paths.user_id AND
      c.ts >= matching_paths.ts1 AND
      c.ts <= matching_paths.ts2
   GROUP BY
      c.user_id, matching_paths.ts1
) pageview_counts;
```

Figure 9: The pure SQL query used to answer the described clickstream analysis question.

the same user, provided that the Y-category click occurs later in time. This is followed by a `SELECT DISTINCT` on the joined result to leave only the ending click that happened soonest after the starting click. Next, the timestamps of each starting and ending click are projected, and the number of clicks that occur in the clickstream between these two timestamps is counted. Finally, this count is averaged across all pairs of matching start and end clicks.

Next, we wrote a SQL/MR function to answer the same question. The query that invokes this function is shown in Figure 10. We partition the input data to this function by `user_id` and order it by `ts`. This means that the function will read an ordered sequence of clicks. Additionally, the function is provided argument clauses specifying the starting page category, the ending page category, and the metrics to be computed (in this case, length). Once the input data is partitioned and sorted, this function makes a single pass through the clickstream. Each time it encounters a click on a page in the starting page category, it stores the position, and each time it encounters a click on page in the ending category, it emits the difference between the ending page's

```
SELECT avg(length)
FROM match_path(
    ON clicks
    PARTITION BY user_id
    ORDER BY ts
    START_PAGE_CATEGORY('X')
    END_PAGE_CATEGORY('Y')
    COMPUTE('length')
    );
```

Figure 10: The SQL/MR query used to answer the described clickstream analysis question.

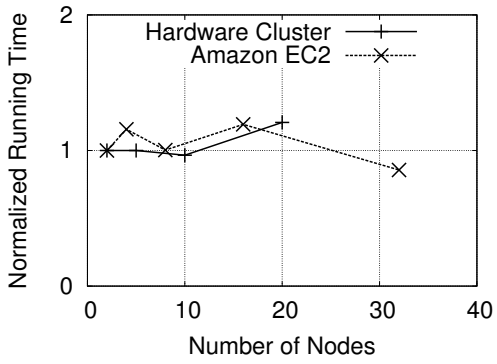

Figure 11: Scale out behavior of SQL/MR on both an hardware cluster and on a cluster deployed on Amazon EC2.

position and the starting page's position.

We ran both the above-described pure SQL and SQL/MR queries on an nCluster composed of 2, 5, 10, and 20 nodes as well as an nCluster deployed on Amazon EC2 with 2, 4, 8, 16, and 32 nodes. The amount of data per node was kept constant. Figure 11 shows the linear scaling out behavior of SQL/MR. A growth in the cluster's size matched by a proportional growth in the amount of data in the cluster yields constant query performance. Because nearly all of the computation of the path matching can be pushed down to the worker nodes, this is the behavior we expected.

We also compared the running time of the SQL/MR query to that of the pure SQL query. The SQL/MR query returned a result about nine times faster than the SQL query. Figure 12 shows a breakdown of the running time of both queries. Note that the execution of the SQL/MR is split evenly between the sorting of the input data (as defined by the `PARTITION BY` and `ORDER BY` clauses) and the actual processing of the data. The execution of the pure SQL query is dominated by the self-join and local `DISTINCT`, with the global `DISTINCT` and final join making up the remainder of the running time.

6.2 Searching Baskets of Page Views

Because a SQL/MR function can maintain its own data structures, it can perform analyses in a single pass over the data that pure SQL requires multiple passes to accomplish. To demonstrate this property, we will consider the task of finding baskets of page views that contain a specified set of pages. For this experiment we reuse the same click data

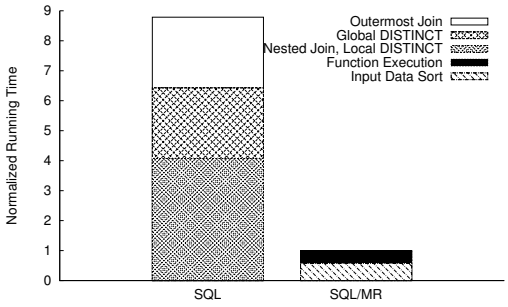

Figure 12: A comparison of the runtime breakdown of SQL and SQL/MR clickstream analysis queries.

as above on a 13 node nCluster. Each user's clicks is now considered a basket of page views. We further define one or more sets of pages, referring to each as a "search set". A user's basket is a match for this query if any one of the search sets is completely contained in the user's basket of page views. Each search set may contain any number of distinct pages. We created SQL and SQL/MR queries to answer this question. Figure 13 shows the normalized performance of searching these baskets for a search set of increasing size using SQL and SQL/MR.

SQL performance degrades as we increase the size of the largest search set. This is because self-joins are used to assemble candidate baskets of clicks for comparison with the search sets. Assembling all size n sets of page views in a user's basket requires $n-1$ self-joins on the clicks table. The most optimized SQL query we were able to write is unfortunately too large to show here due to space constraints. When the search set size is small, the SQL query outperforms SQL/MR because a query with zero or few joins is relatively easy to optimize and evaluate. The increasing number of self-joins eventually complicates both optimization and execution. In fact, we found that searching for multiple search sets, especially sets of different sizes, greatly impacted SQL performance. The SQL results shown are for the best performing SQL queries – those that match users' baskets against only one search set.

The `findset` SQL/MR query that answers the same question is shown below. The `SETID` clause specifies the basket partitions, and the `SETITEM` clause defines the attribute that is the item in the baskets. Each `SET`n clause defines one search set.

```
SELECT userid
FROM findset( ON clicks
   PARTITION BY userid
   SETID('userid')
   SETITEM('pageid')
   SET1('0','1','2')
   SET2('3','10) )
```

In contrast to SQL performance, SQL/MR performance is insensitive to both the search set size and the number of search sets because only one pass over the data is required. During this one pass, simple bookkeeping is performed to test if a user's clicks satisfy any of the candidate sets. The SQL/MR query is also easier to extend to additional search sets by simply adding new `SET`n argument clauses. This stands in contrast to the SQL query, where the addition of

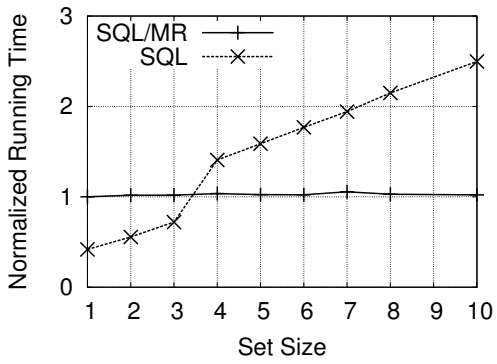

Figure 13: Running time of finding users with clicks that match given sets using SQL and SQL/MR.

a larger search set will require additional self-joins.

7. CONCLUSION

In this paper we have presented SQL/MapReduce, a new framework for user-defined functions. In this framework, functions are self-describing, polymorphic and inherently parallelizable–whether over multi-core processors or over massively parallel servers. The functions accept relations as inputs and output relations; in this respect, their behavior is identical to SQL sub-queries. This enables the functions to be composable, i.e., they can be nested and joined to other sub-queries and functions. In fact, a nested SQL query now trivially defines a data-flow path that chains together SQL sub-queries and SQL/MR functions. Since functions behave like sub-queries, we enable dynamic cost-based re-optimizers to collect statistics at run-time and change the execution order of functions and sub-queries to improve performance. The SQL/MR functions are self-describing at query-time, which allows them to choose their behavior and output schema based on the context in which they are used. This self-describing, dynamic polymorphism facilitates the creation of rich analytic libraries that can be invoked in very different contexts, thereby maximizing code reuse.

We also present an implementation of the framework in a massively-parallel shared-nothing database, Aster *n*Cluster. The implementation allows functions to manage their own memory and file structures. The database manages resources consumed by the function, ensuring that function executions are well-behaved and clean-up after themselves.

As a model by which rich functions can be pushed inside a parallel database, the SQL/MR framework makes the case for an application-friendly database.

Acknowledgements

We are thankful to the engineering team at Aster Data Systems without whom SQL/MR and this paper would not have been possible. In particular we thank Prasan Roy, Mohit Aron, Brent Chun, and Rangarajan Vasudevan. We also thank Arley Lewis for his technical editing.

8. REFERENCES

[1] Apache Software Foundation. Hadoop, March 2009. `http://hadoop.apache.org`.

[2] Apache Software Foundation. Hive, March 2009. `http://hadoop.apache.org/hive/`.

[3] Aster Data Systems. Aster *n*Cluster database. White paper, 2008. Available online: `www.asterdata.com`.

[4] M. Carey and L. Haas. Extensible database management systems. *SIGMOD Rec.*, 19(4):54–60, 1990.

[5] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: Easy and efficient parallel processing of massive data sets. In *VLDB*, pages 1265–1276, 2008.

[6] S. Chaudhuri and K. Shim. Optimization of queries with user-defined predicates. *ACM Trans. Database Syst.*, 24(2):177–228, 1999.

[7] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[8] D. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *Commun. ACM*, 35(6):85–98, 1992.

[9] M. Greenwald and S. Khanna. Space-efficient online computation of quantile summaries. In *SIGMOD Conference*, pages 58–66, 2001.

[10] J. M. Hellerstein and J. F. Naughton. Query execution techniques for caching expensive methods. In *SIGMOD*, pages 423–434, 1996.

[11] J. M. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *SIGMOD*, pages 267–276, 1993.

[12] IBM. *IBM DB2 Universal Database Application Development Guide: Programming Server Applications*, 2004. Version 8.2.

[13] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys*, pages 59–72, 2007.

[14] M. Jaedicke and B. Mitschang. On parallel processing of aggregate and scalar functions in object-relational DBMS. In *SIGMOD*, pages 379–389, 1998.

[15] M. Jaedicke and B. Mitschang. User-defined table operators: Enhancing extensibility for ORDBMS. In *VLDB*, pages 494–505, 1999.

[16] Microsoft Corporation. Table-valued user-defined functions, June 2009. `http://msdn.microsoft.com/`.

[17] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: A not-so-foreign language for data processing. In *SIGMOD*, pages 1099–1110, 2008.

[18] Oracle. *Oracle Database PL/SQL Language Reference*, 2008. Version 11g Release 1.

[19] M. Stonebraker. Inclusion of new types in relational database systems. In *ICDE*, pages 262–269, 1986.

[20] M. Stonebraker, J. Anton, and E. Hanson. Extending a database system with procedures. *ACM Trans. Database Syst.*, 12(3):350–376, 1987.

[21] M. Stonebraker and G. Kemnitz. The POSTGRES next generation database management system. *Commun. ACM*, 34(10):78–92, 1991.

[22] M. Stonebraker, L. A. Rowe, and M. Hirohama. The implementation of POSTGRES. *IEEE Trans. Knowl. Data Eng.*, 2(1):125–142, 1990.