GLADE: A Scalable Framework for Efficient Analytics

Florin Rusu
University of California, Merced
5200 N Lake Road
Merced, CA 95343
frusu@ucmerced.edu

Alin Dobra
University of Florida
PO Box 116120
Gainesville, FL 32611
adobra@cise.ufl.edu

ABSTRACT

In this paper we introduce GLADE, a scalable distributed framework for large scale data analytics. GLADE consists of a simple user-interface to define Generalized Linear Aggregates (GLA), the fundamental abstraction at the core of GLADE, and a distributed runtime environment that executes GLAs by using parallelism extensively.

GLAs are derived from User-Defined Aggregates (UDA), a relational database extension that allows the user to add specialized aggregates to be executed inside the query processor. GLAs extend the UDA interface with methods to `Serialize/Deserialize` the state of the aggregate required for distributed computation. As a significant departure from UDAs which can be invoked only through SQL, GLAs give the user direct access to the state of the aggregate, thus allowing for the computation of significantly more complex aggregate functions.

GLADE runtime is an execution engine optimized for the GLA computation. The runtime takes the user-defined GLA code, compiles it inside the engine, and executes it right near the data by taking advantage of parallelism both inside a single machine as well as across a cluster of computers. This results in maximum possible execution time performance (all our experimental tasks are I/O-bound) and linear scaleup.

1. INTRODUCTION

There is an increasing interest in large scale data analytics from the Web companies in the past years. This is facilitated by the availability of cheap storage that allows them to store huge amounts of user behavior data. In order to extract value out of the data, the analysts need to apply a variety of methods ranging from statistics to machine learning and beyond. SQL and relational database systems, the traditional tools for managing data, do not support directly these advanced analytical methods. Thus, multiple trends have emerged to cope with this problem.

Related work. On one side there is the group who tries to add analytical capabilities to relational databases by using the extensibility features of SQL such as User-Defined Functions (UDF) and User-Defined Aggregates (UDA) [14]. This approach was taken by a series of startup companies who transformed the open-source PostgreSQL database server into a shared-nothing parallel system with analytical functionality [5, 8].

On another side there is the Map-Reduce group [7, 1] who proposes a simplified processing model based on user code that is executed inside the engine. While the execution plan of any Map-Reduce computation is fixed (a Map phase followed by a Reduce phase), the user is allowed to inject code in each of these two phases. Although this allows for greater flexibility in terms of what code is executed when compared to SQL, the need for reusable code templates and a higher-level programming language generated a series of new SQL-like languages such as Pig Latin [11], Hive [15], and Sawzall [13] to be developed on top of Map-Reduce.

A third approach consists in embedding SQL-like operators inside an imperative programming language, thus allowing for the full expressiveness of the host language to be combined with SQL operators. DryadLINQ [9] is the representative of this approach. It uses C# as the host language and supports natively UDFs and UDAs. Essentially, a DryadLINQ program specifies the query execution plan as a graph of operators with corresponding data flowing on the edges. SCOPE [4] is implemented as a SQL-like scripting language on top of the same infrastructure (DryadLINQ and SCOPE are both Microsoft projects).

The last approach we mention here is Dremel [10], a system designed specifically for aggregate queries over nested columnar data. The Dremel query language is SQL extended with capabilities to handle nested data, while queries are executed on a multi-level tree architecture in which partial aggregates are computed at each level.

GLADE. GLADE is a system for the execution of analytical tasks that are associative-decomposable using the iterator-based interface [17]. It exposes the UDA interface consisting of four user-defined functions. The input consists of tuples extracted from a column-oriented relational store, while the output is the state of the computation. The execution model is a multi-level tree in which partial aggregation is executed at each level. The system is responsible for building and maintaining the tree and for moving data between nodes. Except these, the system executes only user code. The blend of column-oriented relations with a tree-based execution architecture allows GLADE to obtain remarkable performance for a variety of analytical tasks—billions of tuples are processed in seconds using only a dozen of commodity nodes.

2. GENERALIZED LINEAR AGGREGATES

In this section we introduce ***G**eneralized **L**inear **A**ggregates* (GLA), the main abstraction at the core of the GLADE framework. GLAs are based on the notion of user-defined aggregates (UDA) [16] which allow users to extend the functionality of a database system with specialized aggregation operators. While the existing database literature [16, 6] focuses mostly on how to embed UDAs into SQL, we depart from this viewpoint and build a system that treats GLAs as the main component. Our approach is related in spirit with the Map-Reduce paradigm [7] which provides the user exactly two functions (Map and Reduce) and a runtime environment (Hadoop [1], for example). Similarly, GLADE, the system we build to support GLAs, provides the user with the interface to define GLAs (the UDA interface extended with methods for serialization) and a specialized runtime environment for aggregation. In the following, we take a closer look at UDAs and Map-Reduce and then we focus on GLAs. We consider two important aspects: how to define GLAs and how to build a dedicated runtime environment for aggregation.

2.1 User-Defined Aggregates

UDAs represent a mechanism to extend the functionality of a database system with application-specific aggregate operators, e.g., data mining [16], similar in nature to user-defined data types (UDT) and user-defined functions (UDF). A UDA is typically implemented as a class with a standard interface defining four methods [2, 6]: `Init`, `Accumulate`, `Merge`, and `Terminate`. These methods operate on the `state` of the aggregate which is also part of the class. While the interface is standard, the user has complete freedom when defining the `state` and implementing the methods. The execution engine (runtime) computes the aggregate by scanning the input relation and calling the interface methods as follows. `Init` is called to initialize the state before the actual computation starts. `Accumulate` takes as input a tuple from the input relation and updates the state of the aggregate according to the user-defined code. `Terminate` is called after all the tuples are processed in order to finalize the computation of the aggregate. `Merge` is not part of the original specification [16] and is intended for use when the input relation is partitioned and multiple UDAs are used to compute the aggregate (one for each partition). It takes as parameter a UDA and it merges its state into the state of the current UDA. In the end, all the UDAs are merged into a single one upon which `Terminate` is called.

While the UDA interface is general and in principle it is possible to compute any aggregate, calling UDAs from SQL imposes significant restrictions on the use of UDAs. Essentially, UDAs, as well as all other aggregates, can return at most a tuple as their result (`Terminate` is called only once). While this is sufficient for general aggregates, it is far too restrictive for complex aggregates (see [16] where a temporary table named *RETURN* is used to store multiple result tuples). As we will see in Section 2.3, GLAs avoid this problem by providing users direct access to the state of the UDA.

2.2 Map-Reduce

The Map-Reduce paradigm [7] simplifies massive data processing on large clusters to jobs consisting of two phases. In the *Map* phase, a user-defined function is applied sequentially to all the tuples (key-value pairs) of the input data set with the result of generating an intermediate set of key-value pairs. In the *Reduce* phase, a second user-defined function is applied to all the values corresponding to an intermediate key in order to aggregate them and generate an output key-value pair. Essentially, the user is required to define two functions, *Map* and *Reduce*, corresponding to the two phases of the computation. The runtime system guarantees that all the values corresponding to an intermediate key are grouped together and passed to the same *Reduce* function. Since the *Map* functions can run in parallel on partitioned data and the *Reduce* functions can run in parallel for different values of the intermediate key, algorithms that can be mapped to this framework also incur high degrees of parallelism. While this is the case for a substantial range of algorithms related to distributed indexing, we believe that aggregate computation does not fit well in the Map-Reduce paradigm. Our opinion is supported by the many languages developed on top of Map-Reduce to incorporate SQL-style aggregation (e.g., Pig Latin [11], Sawzall [13]).

2.3 GLA

The main contribution we propose in this paper is the concept of GLA which combines the UDA interface with a runtime environment similar in spirit with that of Map-Reduce. From a user perspective, a GLA defines any aggregate computation in terms of the UDA interface. While UDAs can be accessed only through SQL though, a GLA provides the user with direct access to the state of the aggregate (this is perfectly reasonable since the user provides the GLA code in the first place). For this to be possible, the GLA needs to be transferred from the runtime address space to the user-application memory space. Thus, the UDA interface needs to be extended with methods to `Serialize/Deserialize` the state. This is not a problem since the user defines the state of the aggregate and has full knowledge on what data is needed to recover the state.

The GLA runtime environment operates on similar principles to Map-Reduce and supports a maximum degree of parallelism. It takes as input the GLA definition and calls the user-defined interface methods. The execution though is targeted for aggregate computation with multiple instances of the GLA being created, one for each data partition. The GLA instances `Accumulate` the data in the partitions in parallel, then `Merge` is called along an aggregation tree to put all the partial states together (again in parallel) before `Terminate` computes the final aggregate state.

While the idea on which the GLAs are founded might seem intuitive, GLAs can express a large class of aggregates that are hard to express both in SQL enhanced with UDAs as well as in Map-Reduce. We present multiple examples of GLAs to show their expressiveness in the following.

Average. The `state` consists of `sum` and `count`.
`Init`: `sum = 0.0`, `count = 0`.
`Accumulate(Tuple t)`: `sum += t.x`, `count++`.
`Merge(AverageGLA o)`: `sum += o.sum`, `count += o.count`.
`Terminate` returns `sum/count`.

Group By. Consider a relation R(A_1, A_2) and the SQL statement: `SELECT A₁, SUM(A₂) FROM R GROUP BY A₁`. We define the `state` of this GLA to be a hash table having as key attribute A_1 and as value the sum. `Init` creates an empty hash table. `Accumulate` first tries to find an entry with the same key in the hash table, case in which adds the value of A_2 to the corresponding sum. If the key is not found, a new entry is created. Essentially, `Accumulate` is nothing more than a hash-based group-by algorithm. `Merge(other)` merges all the groups in `other` into the current state by scanning the hash in `other` element by element. Merging consists in adding up the sum terms. The result is given to the user so it can use a specialized `Terminate`.

Top-K. Given a set of tuples, we need to find the k *best* tuples. In this case, `state` is a min-heap with k entries. `Accumulate(t)` compares `t` with the worst tuples in the heap and, if better, removes the worst and inserts `t` (the heap is reorganized). `Merge(other)` iterates through the tuples in the heap of `other` and calls `Accumulate` for each of them. `Terminate` gives the user the final min-heap that contains the top-k items and allows extraction in order.

K-Means. In K-Means, we want to determine k centers such that the sum of the distances between each value and the closest center is minimized. K-Means is an iterative algorithm that starts with some pre-defined centers, assigns each input data point to the closest center, and then recomputes the center of each group from the corresponding points. This process is repeated until the assignment producing the minimal distance is found. Although not immediately clear, K-Means has a straightforward representation in the GLA framework. A GLA is defined for each iteration of the algorithm. `state` consists of the k pre-defined centers – centers are shared as read-only objects among all GLA instances – and k average ((sum, count) pair) GLAs, one for each center. `Init` remembers the centers (passed as arguments). `Accumulate` finds the closest center and updates the corresponding average GLA (call `Accumulate` on it). `Merge` combines the states of the corresponding centers, while `Terminate` computes the new centers by calling `Terminate` on each average GLA. In order to properly follow the K-Means algorithm, the computed centers are passed to the `Init` function of the GLA corresponding to the next iteration. In summary, the K-Means GLA consists of k Average GLAs, each corresponding to a different partition of the original data.

3. SYSTEM ARCHITECTURE

In this section, we present GLADE (**G**eneralized **L**inear **A**ggregate **D**istributed **E**ngine), the system we have designed in order to implement the GLA framework. GLADE is a relational execution engine derived from DataPath [3], a highly efficient multi-query processing database system. Since DataPath is a shared-memory system, the first step is to implement GLA in such a server environment. The second step is to extend DataPath GLA from a server to a shared-nothing cluster architecture (GLADE) which offers higher scalability. Given the properties of GLA, this extension requires only architectural changes to transform a single-node processing system to a multi-node distributed system. The resulting system is an architecture-independent execution engine with both high scalability and performance (see the experimental results in Section 4) across a large class of aggregation problems.

3.1 The DataPath Database System

DataPath is a relational multi-query database system capable of providing single-query performance even when a large number of (different) queries are run in parallel. The DataPath design brings multiple novel ideas, the most important being to share the data across all stages of query execution. In order to maximize performance, once data are loaded in memory, the data movement is minimized everywhere in the system.

DataPath has a complex storage manager in charge of data partitioned (striped) across a large number of disks. The data of a relation is first horizontally partitioned into fixed size *chunks*, large enough to provide high sequential scan performance. The chunks are evenly distributed across all the disks in the system by the storage manager. This allows data to be read in parallel from multiple disks, thus providing high I/O throughput. Inside a chunk, data is vertically partitioned into columns, each of them stored in a separate set of pages. This minimizes the amount of data read from the disk to only the columns required by the running queries.

The DataPath execution engine has at its core two components: waypoints and work units. A *waypoint* manages a particular type of computation, e.g., selection, join, etc. The code executed by each waypoint is configured at runtime based on the running queries. In DataPath, all the queries are compiled and loaded in the execution engine at runtime rather than being interpreted. This provides a significant performance improvement at the cost of some negligible compilation time. A waypoint is not executing any query processing job by itself. It only delegates a particular task to a work unit. A *work unit* is a thread from a thread pool (there are a fixed number of work units in the system) that can be configured to execute tasks. At a high level, the way the entire query execution process happens is as follows:

1. When a new query arrives in the system, the code to be executed is first generated, then compiled and loaded into the execution engine. Essentially, the waypoints are configured with the code to execute for the new query.
2. Once the storage manager starts to produce chunks for the new query, they are routed to waypoints based on the query execution plan.
3. If there are available work units in the system, a chunk and the task selected by its current waypoint are sent to a work unit for execution.
4. When the work unit finishes a task, it is returned to the pool and the chunk is routed to the next waypoint.

3.2 GLADE

The first step we take in the design and implementation of GLADE is to integrate generalized linear aggregates (GLA) in DataPath. For this, we create a new type of waypoint called GLA Waypoint that is in charge of GLAs. A GLA Waypoint is not aware of the exact type of GLA it is executing since all it has to do is to relay chunks and tasks to work units that execute the actual work. To accomplish this, a GLA Waypoint is configured with the tasks to execute through the process of code generation and loading based on the actual query. Once this is done, the GLA Waypoint

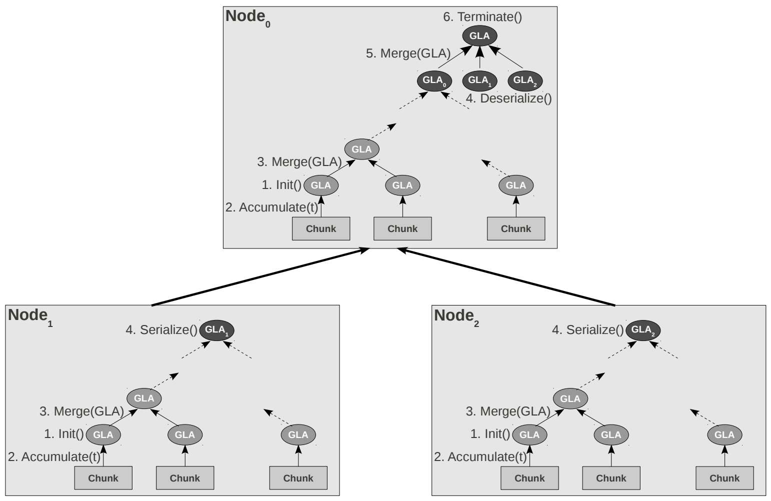

Figure 1: GLADE architecture and execution for an aggregation tree composed of three workers.

can start processing chunks to compute the GLA. A GLA Waypoint needs to store though the state of the GLA it is computing. More precisely, a list of states is stored to allow multiple chunks to be processed in parallel. With these, the computation of a GLA is as follows:

1. When a chunk needs to be processed, the GLA Waypoint extracts a GLA state from the list and passes it together with the chunk to a work unit. The task executed by the work unit is to call `Accumulate` for each tuple such that the GLA is updated with all the tuples in the chunk. If no GLA state is passed with the task, a new GLA is created and initialized (`Init`) inside the task, such that a GLA is always sent back to the GLA Waypoint.
2. When all the chunks are processed, the list of GLA states has to be merged. Notice that the maximum number of GLA states that can be created is bounded by the number of work units in the system. The merging of two GLA states is done by another task that calls `Merge` on the two states.
3. In the end, `Terminate` is called on the last state inside another task submitted to a work unit.

The reason behind designing GLADE for a shared-nothing architecture is the simplicity of the GLA framework which supports distribution natively. Intuitively, the same way we generate GLA states from chunks in a shared-memory system, we can think of generating GLAs at each processing node in a distributed environment. Merging the states together in a final GLA is almost identical in both scenarios. The only difference is that in a distributed system we first need to bring the GLAs on the same node, thus we need to be able to move GLAs between nodes. The `Serialize/Deserialize` methods are in charge of transferring GLAs. When writing these methods, the user needs to identify what data is required to recover the state of the GLA rather than to focus on the communication details, task done automatically by GLADE.

GLADE consists of two types of nodes: a coordinator and workers (as in parallel databases and distributed frameworks like Map-Reduce). The *coordinator* is in charge of scheduling the GLA computation and managing the execution across the workers. Each *worker* runs an instance of DataPath GLA enhanced with communication capabilities. When a job is received by the coordinator, the following steps are executed (Figure 1):

1. The coordinator generates the code to be executed at each waypoint in the DataPath execution plan. A single execution plan is used for all the workers.
2. The coordinator creates an aggregation tree connecting all the workers. The tree is used for in-network aggregation of the GLAs.
3. The execution plan, the code, and the aggregation tree information are broadcasted to all the workers.
4. Once the worker configures itself with the execution plan and loads the code, it starts to compute the GLA for its local data. This happens exactly in the same manner as for DataPath GLA.
5. When a worker completes the computation of the local GLA, it first communicates this to the coordinator—the coordinator uses this information to monitor how the execution evolves. If the worker is a leaf, it sends the serialized GLA to its parent in the aggregation tree immediately.
6. A non-leaf node has one more step to execute. It needs to aggregate the local GLA with the GLAs of its children. For this, it first deserializes the external GLAs and then executes another round of `Merge` functions. In the end, it sends the combined GLA to the parent.
7. The worker at the root of the aggregation tree calls the function `Terminate` before sending the final GLA to the coordinator who passes it further to the client who sent the job.

There are a few design decisions that require further discussion. We decided to use an *aggregation tree* to put the GLAs together due to the minimal communication it requires and the balanced fashion in which the aggregation is executed: all the nodes execute the same amount of work and data transmission (except, of course, for the leaf nodes). Notice though that all the workers communicate with the coordinator, but only short control messages are changed, without significant impact on the communication bandwidth. A disadvantage of using an aggregation tree comes into play when we consider the failure of nodes. If an internal node in the tree fails, all the GLAs below are lost. A simple recovery strategy for this is to save on disk (using the same `Serialize` function) the local GLA at each node. When the coordinator detects the failed node and fixes the tree, there is no work to be redone.

By implementing the GLA abstraction, GLADE manages to exploit parallelism at all levels, inside a single worker node as well as across all the workers. And everything is completely transparent to the user who needs to specify only the GLA interface: the state and six functions. The rest is done by the system automatically and very efficiently.

4. EMPIRICAL EVALUATION

In this section, we evaluate the performance of GLADE on four aggregation tasks: average, group by, top-k, and k-means. We are interested in measuring two parameters: the total execution time and the scaleup. While the *execution time* shows us how fast is GLADE in executing a large range of aggregate tasks, the *scaleup* measures how well GLADE executes larger tasks on a correspondingly larger system.

Data. We use in our experiments the HTML data generator introduced in [12]. All our queries are defined over `UserVisits`, the largest relation (155 million tuples ≈ 20GB) in the benchmark:

```
CREATE TABLE UserVisits (
   sourceIP VARCHAR(16),
   destURL VARCHAR(100),
   visitDate DATE,
   adRevenue FLOAT,
   userAgent VARCHAR(64),
   countryCode VARCHAR(3),
   languageCode VARCHAR(6),
   searchWord VARCHAR(32),
   duration INT );
```

Tasks. We run experiments to evaluate the performance of GLADE on four aggregation tasks:

- Average – computes the average time a user spends on a web page. This task measures the throughput of the I/O system (storage manager) since `Accumulate` executes only two simple arithmetic operations, thus the CPU usage is minimal.
- Group By – computes the ad revenue generated by a user across all the visited web pages. Since `Accumulate` for this GLA needs to update a hash table with 2.5 million distinct keys (groups), this task puts a significant stress on the memory hierarchy. It also tests the speed of the serialization/deserialization functions (CPU) and the communication because the GLA state is significantly large.
- Top-K – determines the users who generated the largest one hundred (top-100) ad revenues on a single visit. This task reads the same data as Group By and executes a more complicated `Accumulate` than Average, thus it is not clear apriori what is the limiting factor.
- K-Means – calculates the five most representative (5 centers) ad revenues. What we want to show with this task is the GLA composition property and how efficient it is to execute a task that incurs multiple passes over the data.

Methodology. For each configuration, we run each experiment ten times and report the median value, more stable to extreme behavior. The difference between the mean and the median was insignificant though.

4.1 Results

We evaluate GLADE on a shared-nothing cluster consisting of 17 nodes. Each node has 4 AMD Opteron cores running at 2.4GHz, 4GB of memory, and a single disk with a maximum bandwidth of 50MB/s. The nodes are connected through a 1Gb/s (125GB/s) switch. Ubuntu 7.4 Server with kernel version 2.6.20-16 is the operating system for all the nodes. We use in our experiments a variable number of nodes, 1 to 16, each node storing a different instance of `UserVisits` (20GB). This is similar to evenly (round-robin) partitioning a 320GB `UserVisits` instance across 16 nodes. The coordinator runs on a separate node. The results for each task are depicted in Figure 2. The execution time is plotted as a function of the number of nodes.

From Figure 2a we can see that GLADE computes the average of 155 million integers residing on disk in 15 seconds. This is possible because GLADE uses vertical partitioning at the storage level and reads only the columns required by the executed query. In this case, 155M * 4B = 620MB of data are read from the disk at 50MB/s in 12.5 seconds while the rest of the time (more than 2 seconds) is spent for compiling and loading the query at runtime (this cost is paid for all the queries). In essence, this query is entirely I/O-bound while the CPU usage is minimal. When increasing the number of nodes, the execution time stays almost flat and the system achieves linear scaleup since the data that is serialized/deserialized and transfered is minimal (the state is only 8 bytes). The small increase is due to the overhead incurred by control messages, loading the code on each node, and the difference in processing across the nodes.

Similar results are obtained for K-Means (Figure 2d) which consists of a set of Average GLAs. Notice though that the impact of making multiple passes over the data, one for each iteration of the algorithm (the algorithm converges in approximately 80 iterations), is minimal on the execution time per iteration.

The results for Top-K are presented in Figure 2c. Since the amount of data read from the disk doubles, we also expect the execution time to double. The results in Figure 2c are better though because the compilation and loading time is incurred only once (2s for compilation + 2 * 12.5 = 25s for I/O for a total of 27s). Since the state of the GLA is small (800B), the time to serialize/deserialize and transfer it is negligible. This results in the almost constant execution time when the number of execution nodes increases, thus almost linear scaleup. The larger increase when compared to the Average GLA is due to the more complicated work that needs to be executed in `Merge`.

The experiments for Group By (Figure 2b) show the largest increase in execution time when the number of nodes increases. This is equivalent to poor scaleup. The reason for this lies in the large state of the GLA. There are 2.5 million distinct groups out of the 155 million tuples in `UserVisits`. These amount to more than 20MB per state. While the transfer of such a state across the network used in the experiments takes a quarter of a second and the time to serialize/deserialize it adds a similar amount, the main bottleneck for this query is the memory bandwidth. The state does not fit in cache anymore, thus a large number of probes in the hash table need to access the main memory, much slower.

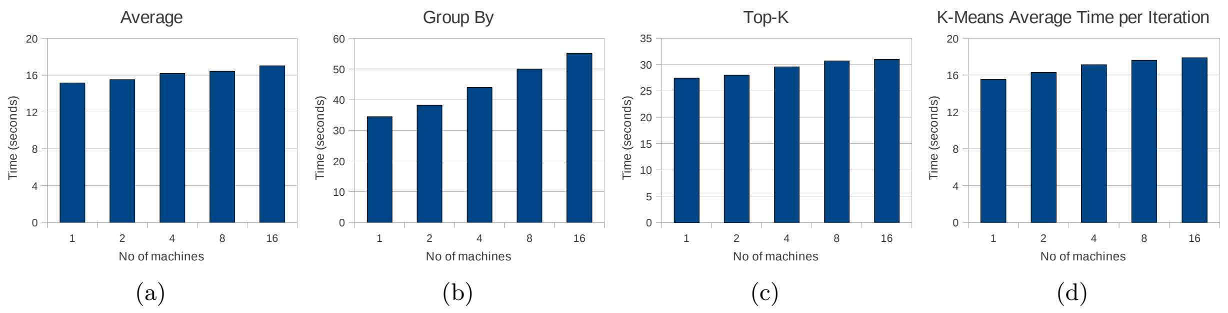

Figure 2: Execution time for a cluster with a variable number of nodes. Each node stores 20GB of data.

On top of this, the number of TLB page misses add the extra time when compared to the Top-K results (the same amount of data is read from the disk). Also, `Merge` needs to merge large hash tables that do not fit in cache.

Remarks. The experimental results confirm the scalability of GLADE across a different mix of tasks and for a different number of nodes. They also confirm the efficiency of the system which is I/O-bound (memory-bound) across all the tasks, even when the state of the aggregate is large and the computation done in the GLA methods is complex. Based on other published results [12] and our investigation on the system used throughout the experiments, we believe that GLADE offers top performance for the tasks at hand.

5. CONCLUSIONS

In this paper we present GLADE, a scalable distributed framework for analytical tasks. GLADE exposes the UDA iterator interface and uses a multi-level tree as execution model. Partial aggregates are computed at each level of the tree, thus the amount of data transferred between nodes is minimized. Having as input relational data in column-oriented format, GLADE returns the final state of the computation to the user. The experimental results show the remarkable performance GLADE is obtaining on a variety of analytical tasks—billions of tuples are processed in seconds across a small cluster consisting of only a dozen commodity machines. The blend of column-oriented storage, user code compiled inside the execution engine, and an architecture that takes full advantage of the thread-level parallelism inside a single machine are behind these results limited only by the physical hardware resources.

In future work, we plan to address some of the limitations of GLADE and to add new functionality to the framework. Currently, the system executes a single task and then it returns the resulting state to the user. It is not clear how to pass the state as input to a new task. Existing systems choose to either materialize the result and then to re-read it in the new task (Map-Reduce) or to define a standard interface between tasks (operators in relational databases). A related problem is passing input parameters to tasks. In the current implementation, GLADE queries correspond to Map-Reduce jobs. Similar to the languages built on top of ensembles of Map-Reduce jobs, we can think of implementing libraries of standard GLAs. In order to be reusable at large scale, these GLAs need to be templated. Thus, our goal is to be able to create programs with composable GLAs that are linked together in such a way that the resulting states flow from one to another.

6. REFERENCES

[1] Hadoop. `http://hadoop.apache.org/`. [Online; accessed July 2011].

[2] Microsoft SQL Server. `http://msdn.microsoft.com/en-us/library/ms131057.aspx`. [Online; accessed July 2011].

[3] S. Arumugam and al. The DataPath System: A Data-Centric Analytic Processing Engine for Large Data Warehouses. In *SIGMOD 2010*.

[4] R. Chaiken and al. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. In *VLDB 2008*.

[5] J. Cohen and al. MAD Skills: New Analysis Practices for Big Data. In *VLDB 2009*.

[6] S. Cohen. User-Defined Aggregate Functions: Bridging Theory and Practice. In *SIGMOD 2006*.

[7] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI 2004*.

[8] E. Friedman and al. SQL/MapReduce: A Practical Approach to Self-Describing, Polymorphic, and Parallelizable User-Defined Functions. In *VLDB 2009*.

[9] M. Isard and Y. Yu. Distributed Data-Parallel Computing Using a High-Level Programming Language. In *SIGMOD 2009*.

[10] S. Melnik and al. Dremel: Interactive Analysis of WebScale Datasets. In *VLDB 2010*.

[11] C. Olston and al. Pig Latin: A Not-So-Foreign Language for Data Processing. In *SIGMOD 2008*.

[12] A. Pavlo and al. A Comparison of Approaches to Large-Scale Data Analysis. In *SIGMOD 2009*.

[13] R. Pike and al. Interpreting the Data: Parallel Analysis with Sawzall. In *Scientific Programming Journal 2003*.

[14] L. A. Rowe and M. Stonebraker. The POSTGRES Data Model. In *VLDB 1987*.

[15] A. Thusoo and al. Hive – A Warehousing Solution Over a MapReduce Framework. In *VLDB 2009*.

[16] H. Wang and C. Zaniolo. Using SQL to Build New Aggregates and Extenders for Object-Relational Systems. In *VLDB 2000*.

[17] Y. Yu and al. Distributed Aggregation for Data-Parallel Computing: Interfaces and Implementations. In *SOSP 2009*.