GLADE: Big Data Analytics Made Easy

Yu Cheng
UC Merced
5200 N Lake Road
Merced, CA 95343
ycheng4@ucmerced.edu

Chengjie Qin
UC Merced
5200 N Lake Road
Merced, CA 95343
cqin3@ucmerced.edu

Florin Rusu
UC Merced
5200 N Lake Road
Merced, CA 95343
frusu@ucmerced.edu

ABSTRACT

We present GLADE, a scalable distributed system for large scale data analytics. GLADE takes analytical functions expressed through the User-Defined Aggregate (UDA) interface and executes them efficiently on the input data. The entire computation is encapsulated in a single class which requires the definition of four methods. The runtime takes the user code and executes it right near the data by taking full advantage of the parallelism available inside a single machine as well as across a cluster of computing nodes.

The demonstration has two goals. First, it presents the architecture of GLADE and how processing is done by using a series of analytical functions. Second, it compares GLADE with two different classes of systems for data analytics: a relational database (PostgreSQL) enhanced with UDAs and Map-Reduce (Hadoop). We show how the analytical functions are coded into each of these systems (for Map-Reduce, we use both Java code as well as Pig Latin) and compare their expressiveness, scalability, and running time efficiency.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*query processing, parallel databases*

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 and log 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.

The approach taken by the database community is 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). A series of companies took the open-source PostgreSQL [2] database server and transformed it into a scalable shared-nothing parallel system with analytical functionality [4]. The large Web companies took a different approach. They developed a new class of scalable systems specifically tailored to large scale data processing. The proposed Map-Reduce [5] programming paradigm has a simplified processing model based on user code that is executed inside the system. 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 [7] to be developed on top of Map-Reduce. A third approach consists in embedding SQL-like operators into an imperative programming language, thus allowing for the full expressiveness of the host language to be exploited. DryadLINQ [6] is the representative of this approach. Essentially, a DryadLINQ program specifies the query execution plan as a graph of operators with corresponding data flowing on the edges.

GLADE executes associative decomposable tasks. 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.

The main goal of the demonstration is to present the GLADE system architecture and execution model. This is done through a series of analytical functions of different complexity such as average, group-by aggregation, top-k, and k-means. As a secondary goal, we compare GLADE with two other analytics solutions: PostgreSQL enhanced with UDAs and two implementations of Map-Reduce in Hadoop [1]—native Java code and Pig Latin. We show how the analytical functions are coded into each of these systems and compare their expressiveness, scalability, and running time performance. The extensive comparison provides the au-

SIGMOD '12, May 20–24, 2012, Scottsdale, Arizona, USA.

dience a detailed image of the available analytics solutions and their capabilities. It also underlines the strengths of GLADE: simplicity in defining complex analytical functions; versatility to adapt to the execution platform with minimum configuration; and remarkable running time performance.

2. ANALYTICS QUERY PROCESSING

GLADE [9] provides an interface to define analytical functions and a specialized runtime for aggregation. The interface is based on the standard UDA interface while the runtime resembles the relational aggregate operator.

UDAs allow users to extend the functionality of a database system with specialized aggregation operators. A UDA is typically implemented as a class with a standard interface defining four methods: `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 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.

***G**eneralized **L**inear **A**ggregates* (GLA) are the main abstraction at the core of the GLADE system. GLAs represent an extension of UDAs. 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. We present the code for Average GLA as a concrete example illustrating the definition of a GLA:

```
template<class ElemType, class ResultType>
class GLA_Average:  public GLA {
private:
    unsigned long count;
    ResultType sum;
    ResultType result;
public:
    GLA_Average():  count(0), sum(0), result(0) {}
    void Accumulate(const ElemType& val) {
       count += 1;
       sum += val;
    }
    void Merge(const GLA_Average& other) {
       count += other.count;
       sum += other.sum;
    }
    void Terminate() {
       result = sum / count;
    }
    ARCHIVER_SIMPLE_DEFINITION()
};
```

The GLADE runtime environment 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.

3. SYSTEM ARCHITECTURE

GLADE (**GLA D**istributed **E**ngine) is derived from DataPath [3], a highly efficient relational multi-query processing database system. 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. 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. 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. 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. Once the storage manager starts to produce chunks for the new query, they are routed to waypoints based on the query execution plan. 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. When the work unit finishes a task, it is returned to the pool and the chunk is routed to the next waypoint.

In order to support GLAs in DataPath, we create a new type of waypoint. 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 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. 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. When all the chunks are processed, the list of GLA states has to be merged. The merging of

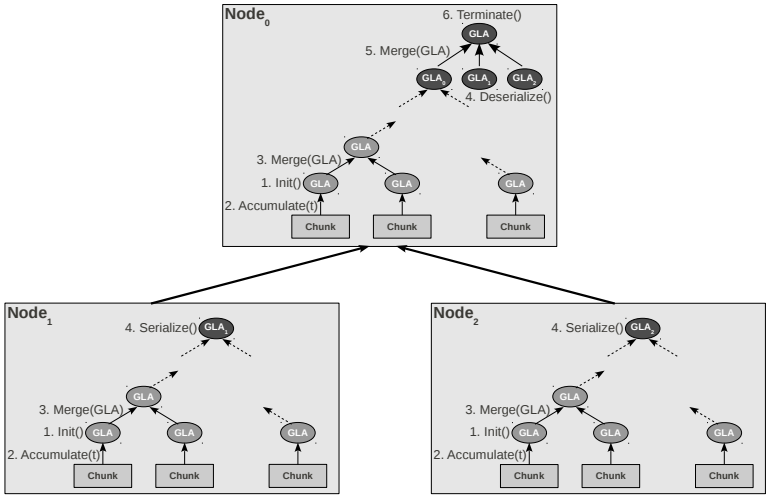

Figure 1: GLADE architecture and execution for an aggregation tree composed of three workers.

two GLAs is done by a task that calls the `Merge` method. In the end, `Terminate` is called on the last state inside another task submitted to a work unit.

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). 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. The coordinator creates an aggregation tree connecting all the workers. The tree is used for in-network aggregation of the GLAs. The execution plan, the code, and the aggregation tree information are broadcasted to all the workers. 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. 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. 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. 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.

4. THE DEMONSTRATION

In this section we introduce the data and the tasks we plan to use in our demonstration. Then we present some preliminary experimental results for one of the tasks – group by aggregation – that show the remarkable performance of GLADE—a factor of 30 decrease in running time compared to Hadoop. We end the section with a detailed presentation of the demo scenario.

Data. We plan to use in our demonstration the HTML data generator introduced in [8]. All our queries are defined over `UserVisits`, the largest relation in the benchmark (155 million tuples ≈ 20GB for the shown results):

```
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. The following four aggregation tasks are used throughout the demonstration:

- 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.

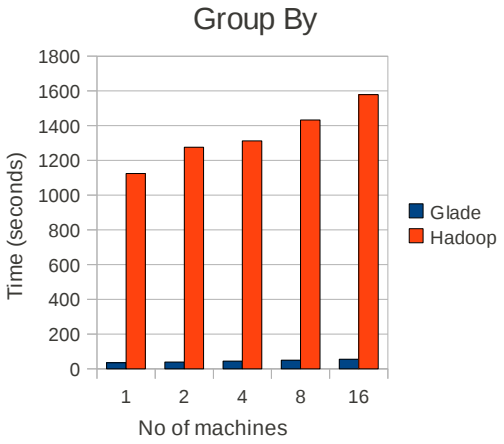

Figure 2: Direct comparison between GLADE and Hadoop.

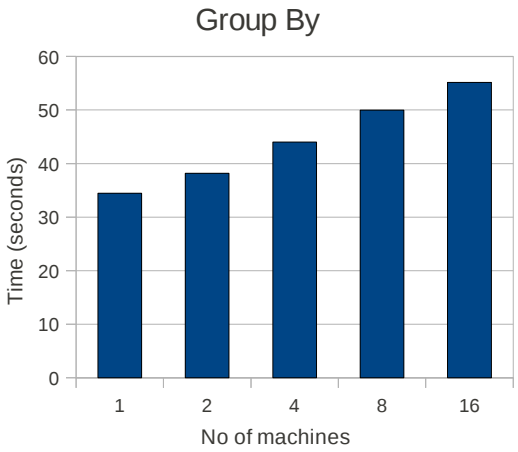

Figure 3: Group by aggregation in GLADE.

4.1 Experimental Results

Figure 2 depicts the execution time results for the group by task run on a shared-nothing cluster consisting of 17 nodes with GLADE and Hadoop. We used Hadoop version 0.20.1 with the NameNode and the JobTracker running on one machine in the cluster and the rest of the nodes configured as DataNodes and TaskTrackers, respectively. We configured Hadoop according to the best practices recommended in [1, 8]. The results in Figure 2 show the immense gap – a constant factor of 30 – between the execution time of GLADE and Hadoop that is maintained across all the configurations. To get a clear look at the scalability of the two systems, Figure 3 presents the same results only for GLADE. There are multiple reasons for the huge gap in running time between GLADE and Hadoop, some of them conceptual and some of them due to the implementation of the two systems: GLADE uses columnar storage and reads only the data required by the executed query while Hadoop reads all the data in the relation; GLADE uses only point-to-point communication between the nodes while Hadoop requires all-to-all communication; GLADE stores only the GLA state while Hadoop stores all the intermediate results for reliability; GLADE uses the vectorized processing model while Hadoop implements the tuple-at-a-time execution strategy.

4.2 Demo Scenario

SIGMOD participants who attend the GLADE demonstration get the chance to see a side-by-side comparison of the different alternatives to large scale data analytics for all the tasks presented above (we included here only the results for the group by aggregation task due to space limitation constraints). We plan to have all the tasks configured and available for the attendees to run. If so desired, the attendees can change the parameters or add modifications to a particular task. The implementation of the analytical functions, system architecture, and execution model is compared for GLADE, PostgreSQL [2], and Hadoop [1]. This provides for a complete picture of the expressiveness and performance of each approach, thus allowing the audience to get a deep understanding of the existing solutions in big data analytics. It also emphasizes the properties of GLADE and its position in the analytics landscape.

There are two scenarios we plan for the demonstration: single node and distributed cluster. For the single node scenario, an instance of `UserVisits` (20GB), the systems, and the queries reside on a local laptop. The attendees are shown how the same analytical function is expressed in each of the systems and how the functions are executed. Since GLADE takes advantage of columnar storage and uses thread-level parallelism extensively, its execution time is considerably shorter than the alternatives. The second demonstration scenario is a distributed cluster environment typical for big data analytics. The same queries are run over an 8TB `UserVisits` instance partitioned across a cluster consisting of 8 large nodes residing at UC Merced. Each node has 16 cores, 16GB of memory, and 4 disks (1TB each). A connection to the cluster is made from the local laptop and the queries are run remotely. The attendees are shown how GLADE executes the same queries in a cluster environment, hiding all the communication details from the user. While both GLADE and Hadoop scale out when increasing the number of nodes, GLADE has significantly better running time performance.

5. REFERENCES

[1] Hadoop. `http://hadoop.apache.org/`.
[2] PostgreSQL. `http://www.postgresql.org/`.
[3] S. Arumugam and al. The DataPath System: A Data-Centric Analytic Processing Engine for Large Data Warehouses. In *SIGMOD 2010*.
[4] J. Cohen and al. MAD Skills: New Analysis Practices for Big Data. In *VLDB 2009*.
[5] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI 2004*.
[6] M. Isard and Y. Yu. Distributed Data-Parallel Computing Using a High-Level Programming Language. In *SIGMOD 2009*.
[7] C. Olston and al. Pig Latin: A Not-So-Foreign Language for Data Processing. In *SIGMOD 2008*.
[8] A. Pavlo and al. A Comparison of Approaches to Large-Scale Data Analysis. In *SIGMOD 2009*.
[9] F. Rusu and A. Dobra. GLADE: A Scalable Framework for Efficient Analytics. In *LADIS 2011*.