Tajo: A Distributed Data Warehouse System on Large Clusters

Hyunsik Choi, Jihoon Son, Haemi Yang, Hyoseok Ryu, Byungnam Lim, Soohyung Kim, Yon Dohn Chung

Department of Computer Science & Engineering, Korea University
Seoul, South Korea
{hyunsikchoi, jihoonson, haemiyang, hyoseok, byungnam, firek, ydchung}@korea.ac.kr

***Abstract*— The increasing volumes of relational data let us find an alternative to cope with them. Recently, several hybrid approaches (e.g., HadoopDB and Hive) between parallel databases and Hadoop have been introduced to the database community. Although these hybrid approaches have gained wide popularity, they cannot avoid the choice of suboptimal execution strategies. We believe that this problem is caused by the inherent limits of their architectures.**

In this demo, we present Tajo, a relational, distributed data warehouse system on shared-nothing clusters. It uses Hadoop Distributed File System (HDFS) as the storage layer and has its own query execution engine that we have developed instead of the MapReduce framework. A Tajo cluster consists of one master node and a number of workers across cluster nodes. The master is mainly responsible for query planning and the coordinator for workers. The master divides a query into small tasks and disseminates them to workers. Each worker has a local query engine that executes a directed acyclic graph of physical operators. A DAG of operators can take two or more input sources and be pipelined within the local query engine. In addition, Tajo can control distributed data flow more flexible than that of MapReduce and supports indexing techniques. By combining these features, Tajo can employ more optimized and efficient query processing, including the existing methods that have been studied in the traditional database research areas.

To give a deep understanding of the Tajo architecture and behavior during query processing, the demonstration will allow users to submit TPC-H queries to 32 Tajo cluster nodes. The web-based user interface will show (1) how the submitted queries are planned, (2) how the query are distributed across nodes, (3) the cluster and node status, and (4) the detail of relations and their physical information. Also, we provide the performance evaluation of Tajo compared with Hive.

I. Introduction

The big data era has come. Hadoop MapReduce [1], [2] has been regarded as the de-facto standard for processing large-scale data sets on shared-nothing clusters. The increasing volumes of relational data let us find an alternative to cope with them. For ad-hoc queries, several hybrid approaches [3], [4] that integrate Hadoop and parallel databases have been introduced to the database community.

Although these hybrid approaches have gained wide popularity, they cannot avoid the choice of suboptimal execution strategies. We believe that this problem is caused by the inherent limits of their architectures:

- Single Source Input: The join is the core operation in relational data warehouses, which integrates heterogeneous data sets from multiple sources. However, since MapReduce supports only a single input source, the join is performed by partitioning all input relations through the one map-reduce job. This approach is suboptimal.
- Fixed Data Flow: MapReduce has only three phases: *map*, *shuffle*, and *reduce*. In the shuffle phase, the output data of mappers are sorted, hashed, and transferred to the reducer according to the keys. The reducer again sorts the received data to group the data set associated with the same key. The join and aggregation are performed in this shuffle (repartition) method. There is no gap into which more optimized method is implanted. The hybrid systems based on MapReduce have the inherent problem that inevitably chooses suboptimal execution strategies.
- Separate Storage: Some hybrid approaches [3] using the database layer require separate data storages for data distribution and processing. In detail, HadoopDB employs a map-reduce job to partition and distribute the input relations on HDFS. The partitioned data must be loaded into the database layer for performance benefits. It takes a long time for data load and causes intensive I/O cost impeding other running workloads in the cluster.

In this demo, we present *Tajo*, a relational, distributed data warehouse system. Tajo runs on shared-nothing clusters. It uses Hadoop Distributed File System (HDFS) as the storage layer and has its own query execution engine instead of the MapReduce framework. A Tajo cluster consists of one master and a number of workers. The master is primarily responsible for the query planning and coordinates workers.

Tajo system has three main points. (1) Each worker has a local query engine that executes a directed acyclic graph (DAG) of physical operators. A DAG of operators can take two or more input sources and be pipelined within the local query engine. Each worker builds and executes flexible query plans that can employ the existing query evaluation techniques [5], [6] that have been studied in the database community. (2) Tajo can make use of various data repartition methods specialized for specific queries. For example, when joining two relations which are already sorted on the join key, Tajo needs to repartition only one relation to workers in which the corresponding part of another relation resides. (3) In Tajo, a physical plan that a worker executes is generated in runtime according to the available resources (e.g., memory) of workers. Workers can execute different physical execution plans in the

same phase. It enables Tajo to maximize the utilization of the resources and work well in heterogeneous environments.

This demonstration focuses on the architecture of Tajo and how it works on shared-nothing clusters. To this end, we will carry out TPC-H benchmarks on 32 cluster nodes. Tajo also provides a web-based user interface. Users can execute SQL-like queries on clusters through this interface. Users can employ the demonstration interface to (1) get a deep understanding of Tajo's behavior during execution, (2) get the plans of submitted query, (3) ask how the query are distributed across nodes, (4) ask the cluster and node status, and (5) ask the tables' detail and physical information.

II. System Architecture

Tajo has the following design principles: (1) performance, (2) fault tolerance, (3) heterogeneity and (4) high-level language. In this section, we describe the architecture and the design choice in order to satisfy the design principles.

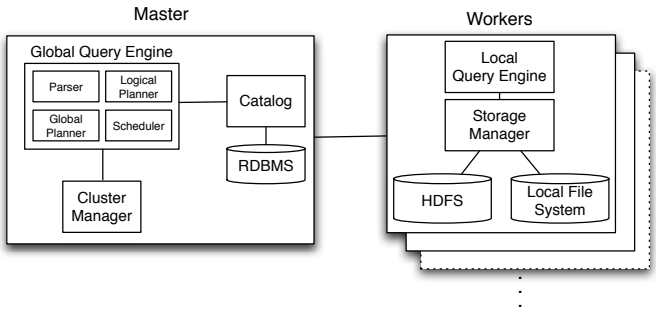

Fig. 1. System architecture of Tajo

In overall, a Tajo cluster consists of one master node and a number of workers as shown in Fig. 1. They are connected to one another via high-speed network. Tajo uses HDFS as the storage layer. HDFS consists of one *name node* (master) and a number of *data nodes*. A Tajo worker and HDFS data node run on the same physical machine.

A. Storage Layer

Tajo uses *Hadoop Distributed File System* (HDFS) as the basic storage layer. HDFS is highly scalable and fault-tolerant, and provides high I/O bandwidth. The data sets in HDFS are automatically distributed across a number of cluster nodes. Since HDFS is designed for batch processing, it is very suited to DW/OLAP workloads at which Tajo aims.

Tajo takes input data sets from HDFS and outputs the results into HDFS. A local query engine of each worker directly scans data sets on HDFS. In many cases where large-scale data sets are managed, HDFS is used as the core storage at which data sets are collected. Therefore, the fact that ad-hoc queries are available without data loading is a significant point.

Tajo also uses the local file system provided by the underlying operating system. The storage manager provides abstract scanner interfaces. Physical operators can process data sets on either HDFS or local file system. According to a query plan, Tajo can achieve the query performance by storing intermediate data or temporary table into the local file system.

B. Tajo System

1) Master: The master is primarily responsible for planning queries and coordinating the activities of workers. The master includes four components, *cluster manager*, *catalog*, *global query engine*, and *history manager*. They communicate one another.

The catalog maintains various metadata, such as tables, schemas, partitions, functions, indices, and statistics. Since the metadata are frequently accessed by the global query engine, they are stored in a conventional RDBMS (e.g., PostgreSql) for low latency access. The catalog also keeps the physical information.

The cluster manager maintains the membership of the workers and resource information of each cluster node. When an additional node joins the Tajo cluster, the cluster manager updates its membership information. All cluster nodes report their resource information (i.e., the number of available processors, memory usages, and remaining disk spaces) periodically. This information is also used for the query planning and provisioning.

When a query is submitted, the global query engine builds a global plan based on the metadata of tables and cluster information which are provided from the catalog and the cluster manager, respectively. For the distributed execution, the global query plan is fragmented into query units. These query units are assigned to workers. During the query processing, the global engine monitors statuses of the running queries for the query optimization and fault tolerance. The details of query planning and query processing will be explained in Section III.

The history manager records the metadata of the executed queries, including query statements, statistics, and logical plans. It also provides a way to search them by query identifiers.

2) Worker: A worker has a query execution engine that performs assigned query units, each of which is a basic unit of query executed by a worker. A query unit contains a logical plan and fragments. A fragment is chunk information of a input relation. During execution, a worker sends periodically the reports of the running queries and the resource status to the master. According to these reports, the master agilely reacts to unexpected failures.

III. Query Processing

We defined an SQL-like query language, called *Tajo Query Language* (TQL). It supports most of SQL DMLs, such as *select*, *from*, *where*, *join*, *group-by*, *order-by*, *union*, and *cube*. In addition, TQL supports two kinds of variables to indicate a scala value and a temporary table respectively. It also supports the assignment statement for them. By this feature, we can easily handle the intermediate data of complex queries as a scala value or table. It gives users opportunities to share the intermediate data among multiple queries. Due to the space limit, we omit the details of TQL.

A. Query Planning

As shown in Fig. 2, Tajo has multiple steps to transform a query statement to physical execution plans. When a user submits a query, the global query engine parses the query into an abstract syntax tree (AST) and compiles it into a logical plan. By using the statistics of the catalog, the query optimizer finds the best logical plan equivalent to the original logical plan. The optimizer uses a cost-based approach in order to find the optimal join order, whereas it uses a rule-based approach for other cases. Also, the query optimizer pushes down the predicates and projections into the appropriate logical operators. This optimization reduces the I/O cost as well as the network cost.

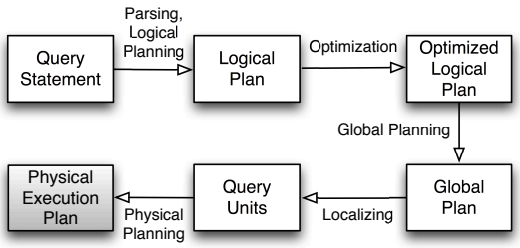

Fig. 2. Query Transformation

The optimized logical plan is transformed into a global query plan. In this step, some logical operators (i.e., group-by, sort, and join) are transformed into two phase with appropriate repartition methods. Usually, the first phase computes local data on each node, and the intermediate data are range-partitioned or hash-partitioned on the specified keys (e.g., sort keys, grouping keys). The second phase computes the partitioned data on each node. As a result, a global query plan forms of a directed acyclic graph (DAG) of subqueries, which represents a data flow. Fig. 3 shows an example of a global plan generated from TPC-H Q12.

Based on the physical information of table, a subquery is localized into a number of *query units*, each of which is a basic unit of a query executed by a worker. A query unit contains a logical plan of the subquery and fragments. A fragment is composed of an input relation path, its schema and metadata, and an offset range of the entire input relation. The fragments and the input relations of the logical plan must be one-to-one mapping association.

Then, the global query engine schedules the query units along with the DAG of the global plan. Like MapReduce, it uses the dynamic scheduling mechanism that continuously assigns each query unit to the available worker in runtime. With this approach, Tajo can easily achieve the load balancing for long time queries.

When a worker receives a query unit, it transforms the logical plan of the query unit into a physical execution plan according to its own computing resources (e.g., available memory). As a result, the query units of the same subquery can lead to different physical execution plans on different

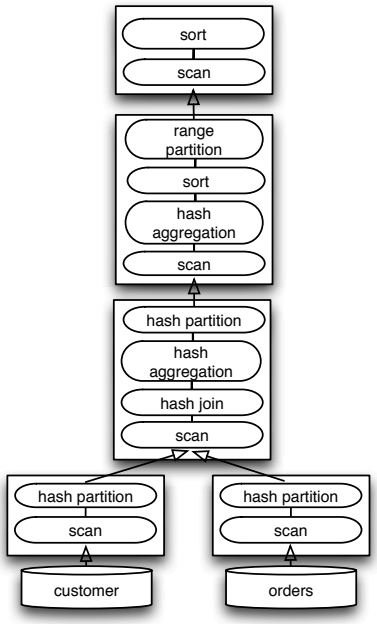

Fig. 3. The global plan for TPC-H Q12

workers. For example, the hash aggregation can be used to do an aggregation in some workers that have enough memory, whereas the sort aggregation can be used in others. This feature maximizes the utilization of resources and enables Tajo to work well in heterogeneous environments. Also, the global query engine receives the statistics of the finished query units from workers. Based on the statistics, it can add some optimization hints to unassigned query units of the same subquery.

B. Query Execution

For input and output of tabular data, Tajo has scanners and appenders. A scanner reads input data from HDFS or local file system, whereas an appender writes output data to either of them. In the current implementation, the scanners/appenders of Tajo support CSV and row-based binary file formats. We have designed the scanner/appender interface to enable users to develop their own scanners/appenders for custom file formats. Also, the scanner interface has two forms: basic scanner and seekable scanner. The basic scanner has the *next()* function to sequentially get one tuple for each call, whereas the seekable scanner additionally provides *seek(offset)* function to can move the reading position to the offset. By using seekable scanner, we have developed the native index scanner. This accelerates the query processing when the query only needs a small portion of a relation.

As we mentioned above, a worker executes a DAG of physical operators and Tajo can use various repartition methods. By combining these characteristics, Tajo can do more optimized and efficient query processing. For example, consider an SQL query with the same group-by keys and order-by keys. In

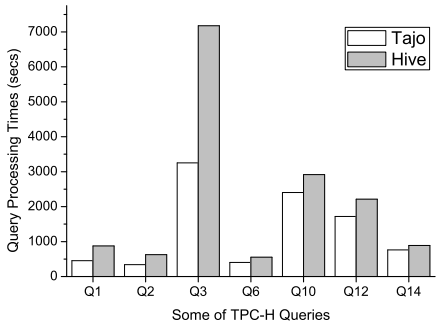

Fig. 4. Performance Evaluation using TPC-H

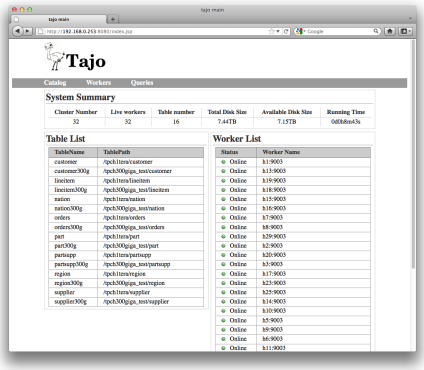

Fig. 5. UI: Summary of System Status

general, the hash-based approach is efficient. However, if the aggregated result is not small, the distributed sort is very costly. In this case, Tajo can use the range repartition and then sort-aggregation approach. As a result, Tajo can eliminate the sort phase. Also, the join query can be executed by the hash repartition and hash-join algorithm if a Tajo cluster consists of workers which are equipped with enough memory. This leads to much performance boost as shown in Q3 of Fig. 4. In addition, if two big relations are sorted and indexed on a join key, Tajo only needs to repartition one relation to workers in which the corresponding part of another relation resides. Then, both relations can be joined through the merge join.

C. Fault Tolerance

Task and node failures in large clusters are no longer exceptions. Tajo also needs to be fault tolerant. Since HDFS is fault tolerant, we only consider the fault tolerance with reference to the query execution strategy.

Tajo aims at DW/OLAP queries which take a long time (about minutes or hours). Tajo adopts the fault tolerance strategy of MapReduce that reassigns failed tasks to other workers. That is, when the master detects the query unit failures, it reassigns the failed query units to other workers.

IV. Experiments

In order to illustrate the performance of Tajo, we evaluated and compared the performance of Tajo and Hive by using 1TB TPC-H benchmark set. For this experiment, we used an in-house cluster of 32 nodes, each of which is equipped with 16GB RAM, 4TB HDD, and an Intel i5 quad core CPU. Fig. 4 shows the experimental results. In the figure, the x-axis means the TPC-H queries, the y-axis indicates the processing times. The results show that our architecture outperforms Apache Hive on the top of MapReduce. In the current implementation, Tajo does not support some predicates (e.g., *in* and *exists*) and correlated nested queries yet, so some queries are missing in this figure.

V. Demonstration Detail

In order to demonstrate how Tajo works, we will prepare a 32 node Tajo/Hive cluster as mentioned in Section IV for side-by-side comparison. The demonstration system allows users to submit some of TPC-H queries and to observe the overall system status and the plan, schedule information of the running queries on Tajo cluster. For this demonstration, we will prepare 100G and 1TB TPC-H benchmark data sets on the cluster.

Basically, Tajo provides the command line interface (CLI) to allow users to submit TQL queries. In addition, as shown in Fig. 5, Tajo system provides a web-based user interface. The UI has four main features: an overall summary viewer, a catalog viewer, a system status viewer, and the query execution interface. The catalog viewer shows all tables and their related information. The system status viewer shows an overall system status and provides users with the detail information of a specific node. The query execution interface enables users to submit a TQL query to the Tajo system. This interface will show the progress of running queries. In addition, users can observe the logical plan of a running query with annotations and the scheduled information.

Acknowledgment

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (No. 2010-0025218).

References

[1] "Apache hadoop," http://hadoop.apache.org.
[2] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Usenix OSDI*, vol. 51, no. 1. ACM, 2004.
[3] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin, "Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads," *Proc. VLDB Endow.*, vol. 2, no. 1, pp. 922–933, aug 2009.
[4] A. Thusoo, J. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy, "Hive - a petabyte scale data warehouse using hadoop," in *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, march 2010, pp. 996 –1005.
[5] G. Graefe, "Query evaluation techniques for large databases," *ACM Comput. Surv.*, vol. 25, no. 2, pp. 73–169, 1993.
[6] D. Kossmann, "The state of the art in distributed query processing," *ACM Comput. Surv.*, vol. 32, no. 4, pp. 422–469, dec 2000.