SQLMR : A Scalable Database Management System for Cloud Computing

Meng-Ju Hsieh
Institute of Information Science
Academia Sinica,
Taipei, Taiwan
Email: eric7428@gmail.com

Chao-Rui Chang
Institute of Information Science
Academia Sinica,
Department of Computer Science and Information Engineering
National Taiwan University
Taipei, Taiwan
Email: crchang@iis.sinica.edu.tw

Li-Yung Ho
Institute of Information Science
Academia Sinica,
Department of Computer Science and Information Engineering
National Taiwan University
Taipei, Taiwan
Email: lyho@iis.sinica.edu.tw

Jan-Jan Wu
Institute of Information Science,
Research Center for Information Technology Innovation
Academia Sinica
Taipei, Taiwan
Email: wuj@iis.sinica.edu.tw

Pangfeng Liu
Department of Computer Science
and Information Engineering,
Graduate Institute of
Networking and Multimedia,
National Taiwan University
Taipei, Taiwan
Email: pangfeng@csie.ntu.edu.tw

***Abstract*—As the size of data set in cloud increases rapidly, how to process large amount of data efficiently has become a critical issue. MapReduce provides a framework for large data processing and is shown to be scalable and fault-tolerant on commondity machines. However, it has higher learning curve than SQL-like language and the codes are hard to maintain and reuse. On the other hand, traditional SQL-based data processing is familiar to user but is limited in scalability. In this paper, we propose a hybrid approach to fill the gap between SQL-based and MapReduce data processing.**

We develop a data management system for cloud, named SQLMR. SQLMR complies SQL-like queries to a sequence of MapReduce jobs. Existing SQL-based applications are compatible seamlessly with SQLMR and users can manage Tera to PataByte scale of data with SQL-like queries instead of writing MapReduce codes. We also devise a number of optimization techniques to improve the performance of SQLMR. The experiment results demonstrate both performance and scalability advantage of SQLMR compared to MySQL and two NoSQL data processing systems, Hive and HadoopDB.

***Index Terms*—cloud data management, NoSQL framework, SQL to NoSQL translation and optimization, MapReduce**

I. Introduction

The advent of cloud computing and hosted software as a service is creating a novel market for data management. Cloud-based DB services are starting to appear, and have the potential to attract customers from very diverse sectors of the market, from small businesses aiming at reducing the total cost of ownership, to very large enterprises seeking high-profile solutions spanning on potentially thousands of machines. At the same time, due to the ever increasing size of data sets, traditional parallel database solution can be prohibitively expensive. To be able to perform this type of analysis in a cost-effective manner, several companies have developed distributed data storage and processing systems on large clusters of shared-nothing commodity servers, including Google File System [1], BigTable [2], MapReduce [3], Hadoop [4], Amazon's Simple Storage Service (S3) [5], SimpleDB [6], Microsoft's SDS Cloud database [7]. There are also various NoSQL databases used to manage large amounts of data, including MongoDB [8], Apache CouchDB [9], Cassandra [10] and Dynamo [11].

Many of these cloud databases are designed to run on a cluster of hundreds to thousands of nodes, and are capable of serving data ranging from hundreds of terabytes to petabytes. Compared with traditional relational database servers, such cloud databases may offer less querying capability and often weaker consistency guarantees, but scale much better by providing built-in support on availability, elasticity, and load balancing.

On the other hand, data management tools are an important part of relational and analytical data management business since business analysts are often not technically advanced and do not feel comfortable interfacing with low-level database software directly. These tools typically interface with the database using ODBC or JDBC, so database software that want to work these products must accept SQL queries. Therefore, a novel technology to combine DBMS capability with Cloud-scale scalability is highly desirable.

In this paper, we propose a hybrid solution, called SQLMR,

المنارة للاستشارات
www.manaraa.com

that combines the programming advantage of SQL with the fault tolerant, heterogeneous cluster, scalable capabilities of MapReduce. Users of SQLMR can write data management programs with familiar query language or to run existing programs without modification. SQLMR provides a compiler to translate a SQL program to a MapReduce program, and execute it in a MapReduce system. To achieve high performance in data processing, we also devise a number of optimization techniques.

The major contributions of this work are summarized as follows.

- A SQL to MapReduce compiler and runtime framework, called SQLMR. Currently, SQLMR supports a subset of SQL queries that, to our knowledge, are sufficient to support various large-scale analytical data management applications, such as on-line analytical processing (OLAP), data mining, etc.
- A low-overhead data file construction technique that enables fast dynamic conversion of SQL database files to HDFS (Hadoop distributed file system) files that can be accepted as input files by the MapReduce runtime engine. This technique significantly reduces data conversion time between SQL and MapReduce.
- Effective database partitioning and indexing techniques for fast locating of queried data in HDFS and reducing disk I/O for range queries.
- A query result caching mechanism that can avoid reprocessing of redundant queries.
- Optimization techniques for Hadoop's MapReduce runtime system to further reduce query processing time.

We conduct extensive experiments to evaluate the effectiveness of SQLMR. The comparison with `Hive` and `HadoopDB`, two well-known MapReduce based NoSQL database managent systems, and MySQL and MySQL_cluster demonstrate the performance and scalability advantage of SQLMR.

The rest of the paper is organized as follows, section II describes the related data management systems, section III presents the system architecture of SQLMR and the interaction between the components of the system. Section IV presents the optimization techniques we devise for *SQLMR*. Section V reports our experiment results, and Section VI gives some concluding remarks.

II. Related Work

Current commercial Cloud DB products such as Amazon's Simple Storage Service (S3) [5], SimpleDB [6] and Relational Database Service (RDS) [12], Microsoft's SQL server [13] and SDS [7] Cloud database, all claim to support SQL. However, most of them do not satisfy many of the desirable properties of Cloud DBMS. For example, Amazon's SimpleDB only supports a small subset of SQL queries. It does not support aggregation and joins types of complex queries. Microsoft's SDS, although supports full SQL functionality, is far more inferior to traditional SQL servers in both performance and scalability.

Bigtable [2] is a share-nothing archtecture and it is column-oriented, which is optimized for read. It is designed for high availability and high performance for massive read and write operations on a hundred-to-thousand machines cluster. It provides low level API for data manipulation. It also provides a self-managing policy to achieve load balancing and dynamic addition/removal of servers. However, Bigtable does not support SQL query. It only supports query by row key [14]. This makes it harder for users to deploy existing applications which work on transitional databases. Moreover, since general-purpose applications may concurrently access a row but different columns, supporting only single-row transaction may degrade the performance. It may need a more fine-grain lock rather than a single row lock to allow concurrent access on a row but different columns. Since Bigtable is built on Google File System, it can provide good scalability and fault tolerance. However, it lacks support to interface with existing data management tools.

S3 [5] provides a simple web service interface to store and retrieve data. It gives developer the highly scalable, reliable and fast data accessing. Unlike S3, SimpleDB does not store raw data, it takes the data as input and expands it to create indices across multiple dimensions, which enables fast query of data. In addition, S3 uses dense storage device to store large object and SimpleDB uses less dense device to store small bits of data.

The structured data of SampleDB is organized in domains, like a spreadsheet, in which user can insert data, retrieve data and run queries. Each domain consist of items which are described by attribute name-value pairs. An attribute can have multiple values. SimpleDB keeps multiple copies of each domain. When the data is updated, all copies of the data are updated. However, it takes time for the update to propagate to all storage locations. The data will eventually be consistent, but an immediate read might not show the change. SimpleDB may be suitable for application like shopping cart. For applications which need strict consistency, SimpleDB is not a feasible solution. Moreover, SimpleDB does not support aggregate operations like join, group and sorting. The developers have to implement these operations by themselves.

Amazon RDS [12] is a web service to set up, operate and scale a relational database in the cloud. It provides full capabilities of MySQL database (5.1). This means the applications which work with existing MySQL databases can work seamlessly with Amazon RDS.

Traditional DBMS like MySQL [15] has been widely used in many domains. However, to statisfy ACID properties, it uses locking mechanism to guarantee concurrent data processing, which limits the scalability of such system. Cloud DBMS has more relaxed ACID constraints, for example, many NoSQL databases only guarantee eventually consistency and result in greater scalability.

MySQL_cluster [16] is a share-nothing, distributed real-time database. It uses synchronous replication through a two-phase commit mechanism and it guarantees data availability. It claims the fast data accessing by storing the indexed columns

المنارة للاستشارات
www.manaraa.com

in main memory and the ability to service tens of thousands of transactions per second. MySQL_cluster provides support to interface with current data management tools. However, its scalability in both data size and machine size is not acceptable for cloud computing.

Declarative language like SQL are often unnatural and restrictive to programmers, in an other way, the code written by procedural language like MapReduce is hard to maintain and reuse. Several efforts have been devoted to combining the DBMS capability and the scalability of MapReduce, including Pig, Hive and HadoopDB, as described in the following.

Yahoo! develops a new language, Pig Latin [17] to fit a sweet spot between these two sides. They also develop Pig compiler to compile Pig Latin language to a plan of MapReduce jobs. However, legacy SQL codes are not compliant with Pig Latin, which limits the portability of existing applications to Pig.

Similar to Pig, Hive [18] also provides a SQL-like query language named HiveQL, which makes it easy to be compatible with existing applications using SQL queries. Hive also compiles HiveQL to map-reduce code. Compared with Pig, Hive produces more efficient map-reduce code.

HadoopDB system [19] is a data management system that combines DBMS capability and MapReduce techniques. It targets analytical workloads on structured data and is designed to run on commodity machines. HadoopDB inherits the scalability of Hadoop and the authors claim it achieves superior perofomance compared with current parallel DBMS.

The major differences between these hybrid data management systems (SQLMR, Pig, Hive and HadoopDB) can be summarized as follows. We devise a number of novel optimization techniques to improve performance of query processing in SQLMR: (1) a low-overhead data file construction technique that enables fast dynamic conversion of SQL database files to HDFS (Hadoop distributed file system) files that can be accepted as input files by the MapReduce runtime engine. This technique significantly reduces data conversion time between SQL and MapReduce, (2) a set of effective database partitioning and indexing techniques for fast locating of queried data in HDFS and reducing disk I/O for range queries, (3) a query result caching mechanism that can avoid re-processing of redundant queries, and (4) optimization techniques for Hadoop's MapReduce runtime system to further reduce query processing time. Our experiment results in later section demonstrate the performance and scalability advantage of SQLMR against these three systems.

III. System Architecture

In this section, we describe the SQLMR system architecture. Since most users are familiar with SQL-like languages, the goal of the SQLMR system is to design a framework that combines the programming advantage of SQL and the scalability and fault tolerance of MapReduce. Figure 1 illustrates the concept of SQLMR. The system accepts SQL queries as input and translates them to a sequence of MapReduce jobs. When the MapReduce jobs are completed, the system returns the query results to the user in SQL form.

Figure 1 depicts the system architecture of SQLMR. There are four main components in SQLMR: *SQL-to-MapReduce Compiler*, *Query Result Manager*, *Database Partitioning and Indexing Manager*, and the *Optimized Hadoop* system. The database partitioning/indexing manager maintains the information of table scheme, indexed files and metadata. The other three components interact with database partitioning/indexing manager for acquiring necessary information when processing a query. For simplicity, we did not draw arrows between database partitioning/indexing manager and the other three components in Figure 1. We describe each component in the following paragraphs.

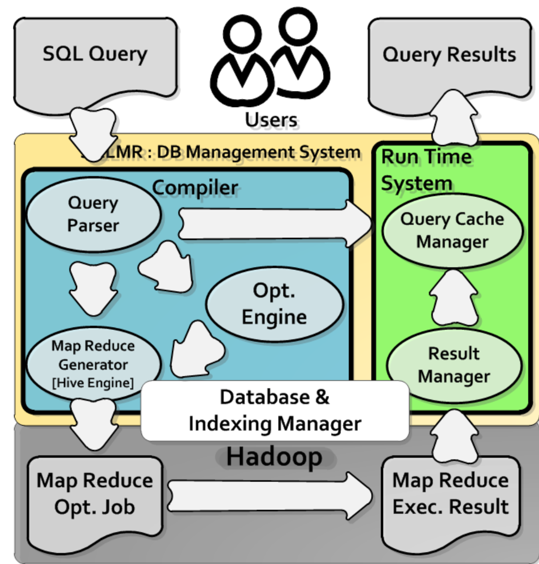

Fig. 1. System architecture of SQLMR.

a) SQL-to-MapReduce Compiler: . The compiler takes SQL queries as input and translates them to a sequence of map-reduce jobs. In the following example, let student_hw be a database table containing 3 columns (id, hw, score). The following SQL query counts the number of students who's hw 1 score is higher than 80:

```
SELECT COUNT(s.id)
FROM student_hw as s
WHERE s.score > 80 AND s.hw=1
```

The SQL query is translated to a pair of map-reduce job. First, the map phase reads records from the student_hw table and produces an output record with two parts. The first part is called the "key", which is populated with student id(s.id) who's hw1 score is higher than 80. The second is the "value", which simply contains "1". The reduce phase then reads the key-value pairs and adds all the "1"s to obtain the total count. The query result returned by SQLMR is a value that represents the total count. Note that the number of mappers and reducers to execute a query are decided by the underlying Hadoop

المنارة للاستشارات
www.manaraa.com

runtime system. In the future, we may provide the functionality for users to decide the number of mappers and reducers for processing a query.

Table I outlines the set of query operations currently supported in SQLMR. Note that these query operations are read-only. This is because, unlike transactional data processing, analytical data processing (such as OLAP) usually reads and processes large input data sets without modifying them. Currently, SQLMR connects with multiple SQL servers to form a hybrid data management system, in which smaller-size queries and write operations are processed by the SQL servers, while large-size read operations are processed by SQLMR. The management of transactional data processing on the multiple SQL servers and the interaction between SQLMR and SQL servers in the hybrid system is out of the scope of this paper due to page limit.

TABLE I
THE QUERY OPERATIONS CURRENTLY SUPPORTED IN SQLMR

Type	Supported Functions
Basic Operations	SELECT WHERE ATTRIBUTES (Single, Multiple, *)
Computing Operations	SUM DISTINCT JOIN COUNT JOIN MULTI-TABLE SUB-QUERY
Condition Operations	GROUP BY BETWEEN-AND MULTI-CONDITION ORDER BY (DESC, ASC) DATA OPERATION

b) Query Result Manager: . The query result runtime system caches the result for each query. When a new query enters SQLMR, the compiler first passes the query to Query Result Manager to compare the query with previous ones in the log. If there are valid cached result for that query, the result is returned to user without re-processing the query. Otherwise, the compiler will parse the query and generates optimized MapReduce code. The cached results will be invalid when a user updates or deletes data from the database.

c) Database Partitioning and Indexing Manager (DPIM).: This system component manages data files and indexing. When new data is added into the system, DPIM partitions the new data and creates index for the new data. With smart partitioning and indexing, SQLMR can do fast locating of queried data blocks as well as identifying exact data blocks that need be accessed in range query in order to reduce disk I/O. The partitioning and indexing techniques are what distinguish our work from other related efforts, which typically export the entire data file to the MapReduce runtime system.

d) Optimized Hadoop.: Hadoop system is a software framework for distributed processing of large data sets on compute clusters. Our Compiler generates optimized map-reduce jobs and execute the jobs on the Hadoop system. We devise a set of optimizations, such as cross-rack communication optimization, to improve the performance of the Hadoop system.

IV. Performance Optimization

A. Data Partitioning and Pre-processing

In this section, we describe SQLMR's approch to transferring data from traditional RDBMS to the Hadoop MapReduce system. We also give an overview of HadoopDB's approach, which will be used as a basis for comparison in our experiments. Figure 2 is the flowchart of HadoopDB. At the beginning, the user needs to export all data from PostgreSQL to a comma-separated value (CSV) text file and put the file in HDFS for subsequent hash-partitioning pre-processing. The purpose of hash-partitioning is to push more query logic into databases (e.g. joins). This can be done in two phases. First, the data file needs to be loaded into HDFS. Then, a HadoopDB custom-made Hadoop job, named GlobalHasher, re-partitions data into a specified number of partitions (e.g. number of nodes in a cluster). The next step is to download all partitioned data from HDFS to the local disk and import the split data to local PostgreSQL server on each node. In contrast, in our SQLMR framework, all database files are stored in HDFS directly without having to pre-process them. This design can reduce the pre-processing time significantly.

Although the hybrid design of HadoopDB allows users to access PostgreSQL database directly, there are two problems that need be overcome. First, in HadoopDB each PostgreSQL server only stores a partition of the database. Therefore, the user has to merge all partial query results returned from each database partition to get the final result. Second, users need to recovery the system manually should a PostgreSQL Server crashes. In comparison with HadoopDB, our SQLMR framework stores all data in HDFS, which saves the user the burden of gathering the partial query results. The process of recovery can also be done by Hadoop MapReduce System automatically. Furthermore, storing data in HDFS inherits the fault-tolerant capability provided by HDFS. HDFS replicates the data remotely to ensure data availability if machines crash. SQLMR relies on HDFS for data replication, and the replication level of data is also decided by HDFS.

In order to further reduce the time for data loading and speed up data processing, we develop a number of optimizations in the SQLMR framework, as shows in Figure 3. At the beginning, SQLMR analyzes the table schema to get the data size of one record. Next, SQLMR reads all data from database server and partitions the data according to the analyzed schema and block size of HDFS. Finally, the hashed and partitioned table data is stored in HDFS. The details of partition will be described in next subsection.

HadoopDB implements a hybrid database system by developing a Database Connector to retrieve data from traditional PostgreSQL database. The Database Connector in HadoopDB is similar to a normal database client implemented in JDBC (Java Database Connectivity). During the HadoopDB experiment, we found that HadoopDB takes a lot of time to retrieve

المنارة للاستشارات
www.manaraa.com

data from PostgreSQL and causes very heavy I/O loading. In order to overcome the issue, we also develop a low-overhead data file construction technique that enables fast dynamic conversion of SQL database files to the format that can be accepted as input files by the MapReduce runtime engine. The file construction technique implements the InputFormat interface of Hadoop MapReduce system. The interface is called by Mapper function to read the needed data from HDFS and can be designed to read any data in any format. In SQLMR, we implements a custom InputFormat Java class, labeled as Data Connector in Figure 4, which can read MySQL database files directly without having to export database as text files. This technique significantly reduces data conversion time between SQL and MapReduce.

The data connector assigns one mapper per SQL DB, because assigning too many mappers to one host will result in too heavy disk I/O access and decrease the throughput. In addition, the data connector balances the workload on the mappers using the following strategy. First, each mapper connects to every SQL DB and issues ”SELECT COUNT” to get the total number of needed data stored in each server. Next, each mapper randomly selects the SQL DB it will retrieve data from. Finally, each mapper retrieves approximately the same amount of partial data from its selected SQL DB using the ”SELECT LIMIT” command.

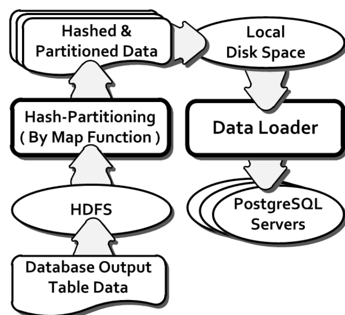

Fig. 2. The SQL to HDFS data loading process in HadoopDB.

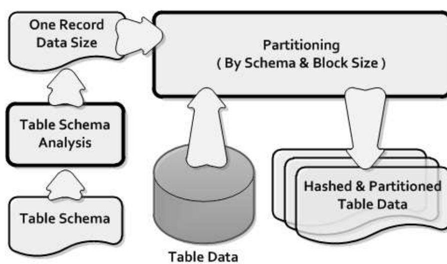

Fig. 3. The SQL to HDFS data loading process in SQLMR.

B. Data Indexing

Index is a data structure to facilitate and improve the performance of data retrieving and searching in traditional DBMS. For cloud DBMS, it becomes even critical since the stored data is numerous and we need to identify the data we are interested in very soon. In SQLMR, we employ two indexing techniques to accelerate data searching. SQLMR chooses a suitable index technique depending on the characteristics of the database. We next introduce the two index techniques.

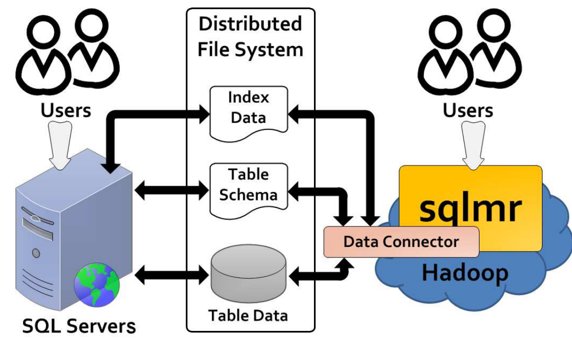

Fig. 4. The Data Connector that enables reading data from database files.

1) Partition Index: In this approach, the files storing database tables are split into fixed length files. The size can be determained by the size of block in HDFS, such that a file would be completely contained in one block. Each file contains the records with a range of series keys. The range of keys is decided by the schema of table and the size of a file. For example, in Figure 5, there are 4 columns in a table. After table schema analysis, a record is 2KB and a file is 64MB. Then, SQLMR will partition one table data file into multiple partitioned table data files. A file will contain a series of data rows in which the keys range from 1 to 32,768. The next file will contain data rows with keys ranging from 32,769 to 65,537, and so on. The insertion, deletion and search operation for a key are all constant time $O(1)$, because the file contains the corresponding record and the offset in a file can be calculated by the key of the target record. This approach is suitable for data with dense key space, since we pre-allocate the space for a record in a file. For general cases, we use B+ tree index.

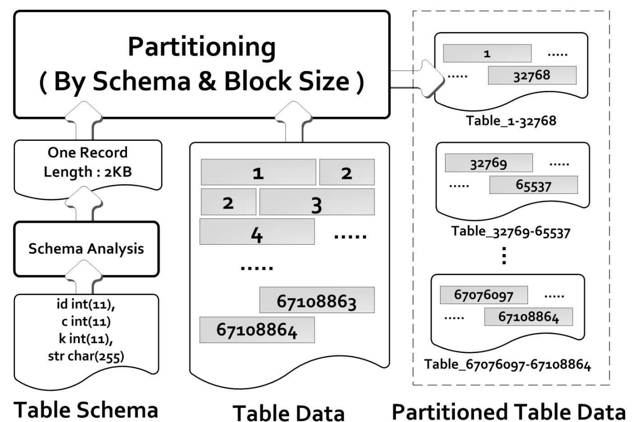

Fig. 5. Illustration of data partitioning in SQLMR.

المنارة للاستشارات
www.manaraa.com

2) B+ tree Index: B+ tree index are extensively and commonly used in database indexing. Many open-source and commercial database product, like Oracle [20] and MySQL, apply B+ tree to index the data. It is a general approach and applicable to various applications. In SQLMR, the B+ tree structure is maintained by the HDFS master, and we modify the DFSClient module in Hadoop such that we can query the block loaction by a key through the tree. The search, deletion, and insertion are all logarithmic amortized time $O(logN)$ where N is the number of nodes in the tree. In order to build the index tree, we need to query the master node for the block information. The master node returns all the locations of a block, including the master copy and its replicas. Our B+ tree node stores all the replicated blocks information to maintain high availability of data blocks information.

The query process is as follows. SQLMR reveives a query of a search key and finds the corresponding block through the tree. The internal node of the tree only stores the information of key for seaching, and the leaf node of the tree stores the data of block information, including the block ID and location. Figure 6 illustrates the structure of the B+ tree. To facilitate block mergence and split, the keys in a block have to be sorted. We sort the key in a block when the blocks is merged or split to reduce the extra overhead of sorting and since the number of keys in a block is bound by a constant C, where

$$C = \frac{\text{the size of block}}{\text{the size of a record}}$$

Therefore, the extra time complexity of sorting is $O(1)$.

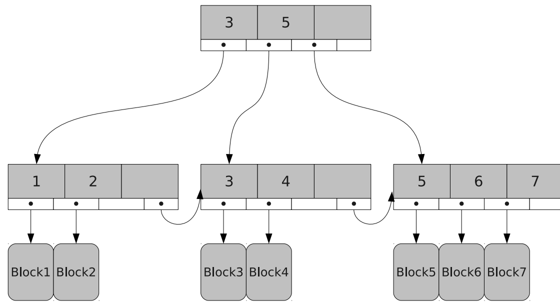

Fig. 6. The structure of B+ tree.

C. Hadoop Optimization

User query is compiled as MapReduce jobs and run on Hadoop, therefore, the performance of Hadoop framework is critical to SQLMR perofrmance. We employ ours previous work on cross-rack optimization in Hadoop framework to improve the performance of Hadoop. We give a brief introduction to this optimization technique in the following.

MapReduce employs all-to-all communication model between mappers and reducers. This results in saturation of network bandwidth of top-of-rack switch in shuffle phase and straggles some reducers and increases job execution time. In current Hadoop implementation, the placement of reducers is random which may result in network load unbalancing and make the reducers on a busy rack become stragglers. In our previous work, we model the traffics in shuffle phase and give two optimal algorithms to balance the network load among racks by placing the reducers to racks properly. The experiment shows the improvement achieves 32% in PageRank application.

In [21], we propose a *Reducer Placement Problem (RPP)* which is defined as follows. Give the number of racks, the number of mappers on each rack and the number of reducers to schedule, how do we determine the number of reducers run on each rack? We give a simplified traffics model and derive an objective funtion to represent the amount of traffics of a rack. In this model, the traffics of a rack is a function of number of reducers (r_i) run on it and we formulate *RPP* as a minmax optimization problem.

Formally, suppose we have N racks, M mappers and R reducers, the number of mapppers on each rack $\{m_1, m_2, \ldots, m_N\}$ are known. The traffics of rack i is $f_i(r_i)$, we want to find the number of reducers on each rack $\{r_1, r_2, \ldots, r_N\}$ such that

$$min\{ \underset{r_i, i \in \{1,\cdots,N\}, \sum_i r_i = R}{\text{argmax}} \{f_1(r_1), \cdots, f_N(r_N)\} \}$$

One of our optimal algorithm is in greedy manner. It places one reducer to a rack at a time. The main idea is that, always place the reducer on the rack with minimum traffics currently. Algorithm 1 demonstrates the pseudo code of our greedy algorithm. Note that we use an array $state_tuple$ to store the number of reducers on each rack. For example, if we have four racks and ten reducers to schedule, our greedy algorithm reutrns $state_tuple = [1, 2, 3, 4]$, whcih means placing one reducer in rack 1, two in rack 2, three in rack 3 and four in rack 4.

We choose five popular and representative applications as our benchmark suite and conduct the experiment in a four-rack cluster. The evaluation metric is the speedup rate compared with un-optimized Hadoop. Table II shows the speedup of each application. The best improvement is PageRank which achieves 32% and FPM does not have significant improvement. This is because FPM has a very skewed distribution in intermediate data size and our traffic model assumes the size of intermediate data is a constant. In constrast, other applications have approximately the same size of intermediate data.

Benchmark	Speedup (%)
Grep	9.35
WordCount	12.37
PageRank	32.84
K-mean	14.7
FPM	1.76

TABLE II
SPEEDUP OF BENCHMARKS

المنارة للاستشارات
www.manaraa.com

Algorithm 1 Greedy Algorithm for RPP

Require: The number of mappers on each rack : $\{m_1, m_2, \ldots, m_N\}$
Ensure: A reducer state tuple : $\{r_1, r_2, \ldots, r_N\}$
  $N \leftarrow$ number of racks
  $M \leftarrow$ number of total mappers
  $R \leftarrow$ number of total reducers
  $state_tuple[N] \leftarrow \{0, 0, \ldots, 0\}$
  for $i = 1$ to R **do**
    $minimal \leftarrow \infty$
    for $j = 1$ to N **do**
      $traffic = (M - 2m_j) \cdot (state_tuple[j] + 1) + m_j R$
      if $traffic < minimal$ **then**
        $candidate = j$
      end if
    end for
    $state_tuple[candidate] + +$
  end for
  return $state_tuple$

V. Experiment

A. Experiment Setting

In the experiment, we use SysBench as database benchmark and compare SQLMR with other database systems, including standalone MySQL_on_Ceph, in which data files are stored on the Ceph distributed file system, MySQL_cluster, and two MapReduce-based systems: Hive and HadoopDB. SysBench is a modular, cross-platform and multi-threaded benchmark tool for evaluating OS parameters that are important for a system running a database application under intensive load. We use the OLTP module of SysBench to benchmark a real database performance. OLTP can generate a lot number of sequential data indexed by column id. It can also generate transactional queries.

Since Hive and HadoopDB only handle read operations with the MapReduce framework, in our experiments, we only compare performance of read operations, including range sum and join queries. We use Ceph DFS as the underlying distributed file system for MySQL in order to allow MySQL to accommodate large dataset. MySQL cluster loads all data into memory to achieve fast response time. However, the total size of memory limits its scalability. Hive is a data warehouse infrastructure built on top of Hadoop. Hive defines a simple SQL-like query language that enables users familiar with SQL to query the data without writing MapReduce codes. HadoopDB is an architectural hybrid of MapReduce and DBMS technologies for analytical workloads.

The experiment contains two parts: data scalability and system scalability. The former is to show the scalability w.r.t. increase in data size while the number of nodes is fixed at 10 and the latter to show the scalability w.r.t. increase in system size with fixed 10GB data size per node and totally 64 nodes. Each node contains 2 CPU cores rated at 2.27GHz, 4GB memory, 200GB disk space and all of them are connected by one Gigabit Ethernet switch. The result time of each experiment is measured by the 'time' command and each data point is the average of 10 runs.

B. Experiment results

1) Effect of Data Size: This set of experiments compare the scalability w.r.t. increase in data size. The number of nodes is fixed at 10, and the data size varies from 512MB to 1TB.

Figure 7 shows the execution time of MySQL, MySQL_cluster, Hive, HadoopDB and SQLMR on `SELECT` operation with different data sizes. Figure 8 and Figure 9 give graphical illustrations of the performance comparison. The SQL query is as follows.

```
SELECT sum(id) FROM table
WHERE id >= max(id)/2 and id <= max(id)
```

For small data size (Figure 8), we found that MySQL and MySQL_cluster outperform MapReduce-based systems when the data size is smaller than 4GB. When the data size increases beyond 8GB, both of them are outperformed by MapRedue-based systems. The reason is that MySQL does not parallelize processing of single query. MySQL_cluster employs in-memory database technique, which writes all data to memory before starting database operation and thus is limited by the size of the physical memory. In our experiment environment, MySQL_cluster would crash due to out of memory when the data size reaches 64GB.

In Figure 9, MySQL crashes when the data size reaches 772GB. The reason is that a MySQL data table can only accommodate 2^{32} data records. 772GB is the maximum data size that can be generated by the `Sysbench` benchmark because of such constraint. For MapReduce-based systems, we found that the execution time of HadoopDB increases dramatically with the increase in data size. The reason is that HadoopDB incurs higher I/O workload caused by the multi-phase data pre-processing described in Section IV-A. SQLMR consistently outperforms HadoopDB because of the performance improvement by various optimizations described in Section IV. SQLMR is 2.82 times faster than HadoopDB with 32GB data size and 13.35 times faster than HadoopDB with 1TB data size. Furthermore, SQLMR is 1.41 times faster than Hive on average.

Figure 10 shows the execution time of Hive, HadoopDB and SQLMR on `JOIN` operation with different data sizes. Figure 11 and Figure 12 give graphical illustrations of the performance comparison. The SQL query is as follows.

```
SELECT sum(tabl1.id) FROM table1 JOIN table2
ON (table1.id = table2.id)
WHERE table2.id >= max(table2.id)/2 and
             table1.id <= max(table1.id)
```

For small data size (Figure 11), we found that the execution time of HadoopDB increases dramatically with the increase in data size. The reason is that HadoopDB incurs higher I/O workload caused by the multi-phase data pre-processing described in Section IV-A. SQLMR is 1.47 times faster than HadoopDB with 1GB data size and 3.55 times faster than

المنارة للاستشارات
www.manaraa.com

	MySQL	MySQL(C)	Hive	HadoopDB	SQLMR
512MB	7.56	3.32	34.34	38.38	32.14
1GB	11.69	6.02	33.27	41.35	31.64
2GB	20.32	12.11	34.24	43.36	33.37
4GB	42.58	23.98	34.27	38.28	33.41
8GB	84.66	47.15	40.08	40.48	37.29
16GB	165.25	94.72	49.53	70.17	41.89
32GB	334.14	188.98	58.11	136.03	48.23
64GB	659.07	378.50	91.50	281.37	70.39
128GB	1,305.01		157.43	706.50	122.19
256GB	2,578.99		295.79	1,955.53	209.22
512GB	5,180.53		586.70	4,070.14	387.56
772GB	7,058.54		866.25	6,820.45	552.35
1TB			1,145.80	9,570.77	717.15

Fig. 7. Comparison of execution time between different database systems on SELECT query with different query sizes.

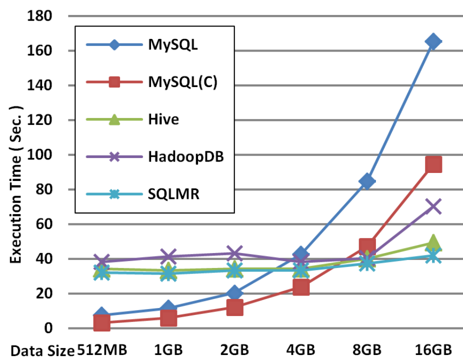

Fig. 8. Comparison of execution time of SELECT query between different database systems with small data sizes.

HadoopDB with 16GB data size. Furthermore, SQLMR is 1.18 times faster than Hive on average.

For large data size (Figure 12), the speed-up factors of SQLMR against HadoopDB and Hive are even more significant. SQLMR is 8.23 times faster than HadoopDB with 32GB data size and 28.11 times faster with 1TB data size. SQLMR is 1.29 times faster than Hive on average.

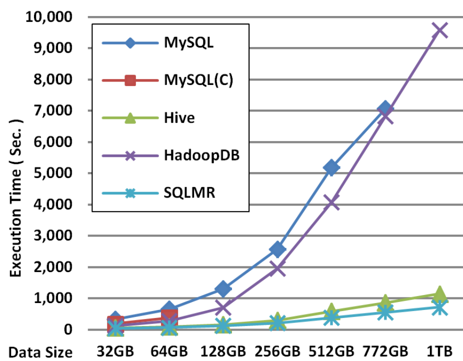

Fig. 9. Comparison of execution time of SELECT query between different database systems with large query sizes.

	Hive	HadoopDB	SQLMR
1GB	62.07	81.39	55.30
2GB	67.66	97.58	58.86
4GB	87.44	119.87	66.01
8GB	90.67	189.43	84.35
16GB	107.76	308.99	86.93
32GB	117.67	790.35	96.01
64GB	145.74	1,524.20	128.54
128GB	182.68	2,698.23	175.86
256GB	310.41	4,475.29	231.81
512GB	601.65	9,381.90	397.94
1TB	1,130.28	21,292.22	757.55

Fig. 10. Comparison of execution time between different database systems on JOIN query with different query sizes.

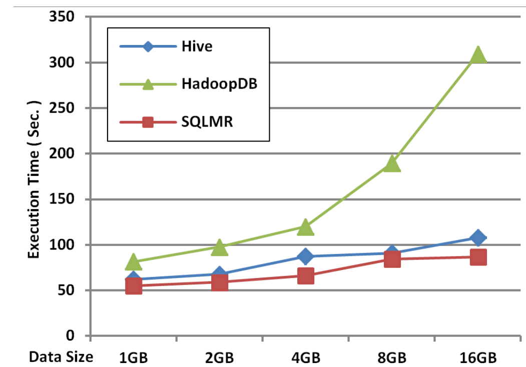

Fig. 11. Comparison of execution time between different database systems on JOIN query with small data sizes

2) Effect of System Size : This set of experiments compare the scalability of different database systems w.r.t. increase in system size. The number of physcial nodes varies from 1 to 16. Each physical node runs four virtual machines (i.e., virtual nodes). The data size per virtual node is fixed at 10GB. Figure 13 compares the result of SELECT (range sum) query, and Figure 14 shows the result of JOIN query.

As shown in Figure 13, HadoopDB exhibits unstable system scalability while SQLMR and Hive behave similarly and both

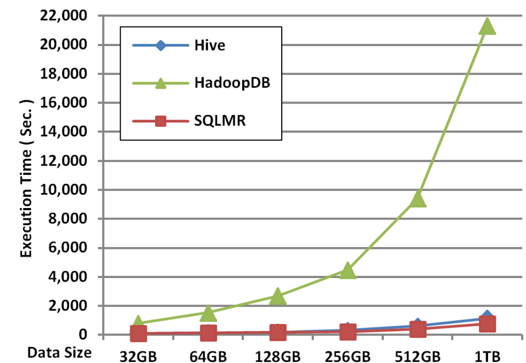

Fig. 12. Comparison of execution time between different database systems on JOIN query with large data sizes.

المنارة للاستشارات
www.manaraa.com

exhibit more stable scalability than HadoopDB. All of the systems perform worst when the number of virtual nodes (i.e. virtual machines) is four. This is because the four virtual machines residing on the same physical node saturate source utilization on that node. When the number of virtual machines increases from 4 to 16, the workload is shared between multiple physical nodes, which increases parallelism and results in decrease of execution time. When the number of virtual machines increases to 32, the network becomes the bottleneck and causes increase in the execution time.

From Figure 13, we can see that the performance improvement ratio of SQLMR against HadoopDB ranges from 4.16 (1 node) to 4.95 (64 nodes). The performance improvement ratio of SQLMR against Hive ranges from 1.67 to 2.05.

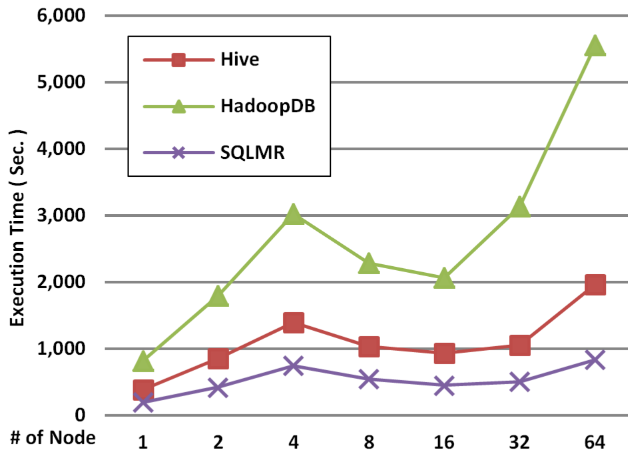

Fig. 13. Comparison of execution time between different database systems on SELECT query, with fixed data size per node and varied number of nodes.

Figure 14 also shows that HadoopDB exhibits unstable system scalability while SQLMR and Hive behave similarly and both exhibit more stable scalability than HadoopDB. The main reason for HadoopDB's poor system scalability is that, HadoopDB uses PostgreSQL server on each local node as the database storage without the support of distributed storage system. HadoopDB uses the *Map* function to collect all the queried data and send the whole set of data to the reducer for computation of the final result. From Figure 13, we can see that the performance improvement ratio of SQLMR against HadoopDB ranges from 6.03 (2 nodes) to 10.57 (64 nodes). The performance improvement ratio of SQLMR against Hive ranges from 1.65 to 2.24.

3) Comparison of Data Preprocessing: Figure 15 compares the breakdown of data pre-processing overhead between SQLMR and HadoopDB. Recall that HadoopDB requires four phases of data pre-processing (Section IV-A), while SQLMR only requires two phases. For the partition phase, SQLMR uses the *split* function in the Linux OS to partition database table data into small files.

As shown in Figure 16, HadoopDB requires 140.44 times more partitioning time than SQLMR with 512MB data size. This is because HadoopDB relies on the *Map* function in

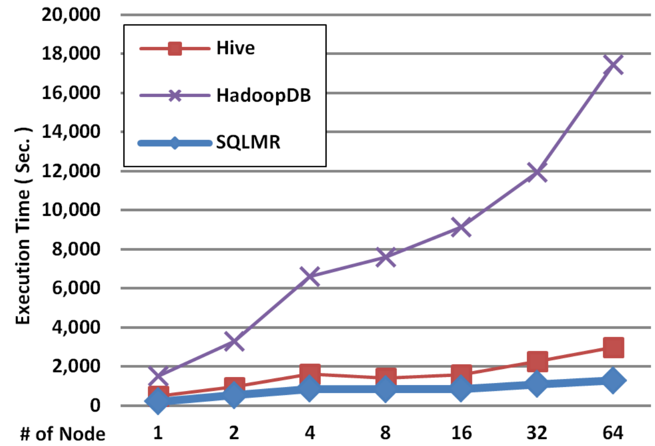

Fig. 14. Comparison of system scalability between different database systems on JOIN query, with fixed data size per node and varied number of nodes.

the MapReduce system for the data partitioning phase. This approach benefits from large parallelism provided by the MapReduce system. However, it suffers large overhead when partitioning small data file. On the other hand, the performance advantage of SQLMR decreases when the data size increases. At 1TB data size, HadoopDB requires 70% partitioning time of SQLMR (Figure 17). The reason is that the *split* function is not parallelized, hence the partitioning time of SQLMR increases with the increase in data size. We expect the partitioning phase of SQLMR to be improved when a parallel partitioning function is available.

In terms of total pre-processing time, SQLMR is 11.48 times faster than HadoopDB with 512MB data size and 2.68 times faster than HadoopDB with 1TB data size. The average speed up factor of SQLMR against HadoopDB over various data sizes is 3.94.

	HaddopDB				sqlmr	
Data Size	Upload to HDFS	Partition	Download from HDFS	Upload to PostgreSQL	Upload to HDFS	Partition
512MB	4.42	41.34	3.17	5.16	4.42	0.29
1GB	11.05	78.67	9.12	9.08	11.05	1.13
2GB	13.46	84.89	10.24	15.96	13.46	12.24
4GB	23.61	98.71	21.30	30.19	23.61	28.03
8GB	47.20	153.25	42.82	58.87	47.20	58.29
16GB	99.85	221.59	95.46	141.58	99.85	126.09
32GB	210.84	356.71	204.22	312.21	210.84	246.09
64GB	444.87	558.89	436.46	656.03	444.87	518.50
128GB	929.78	729.02	911.44	1,400.93	929.78	917.87
256GB	1,980.44	1,328.76	1,950.02	3,084.37	1,980.44	1,376.80
512GB	3,962.86	2,152.91	3,884.28	6,771.58	3,962.86	2,065.20
772GB	5,980.25	2,510.42	5,910.74	9,885.89	5,980.25	3,097.80
1TB	8,399.89	3,237.01	8,266.74	15,055.64	8,399.89	4,646.70

Fig. 15. Comparison of Data Preprocessing time between HadoopDB and SQLMR

VI. Conclusion

In this paper, we have proposed a hybrid solution, called SQLMR, that combines the programming advantage of SQL

المنارة للاستشارات
www.manaraa.com

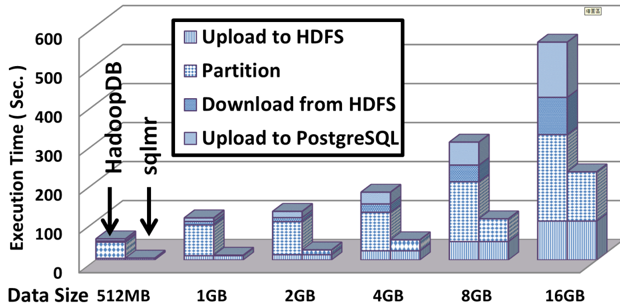

Fig. 16. Comparison of Data Preprocessing time between HadoopDB and SQLMR with small data sizes

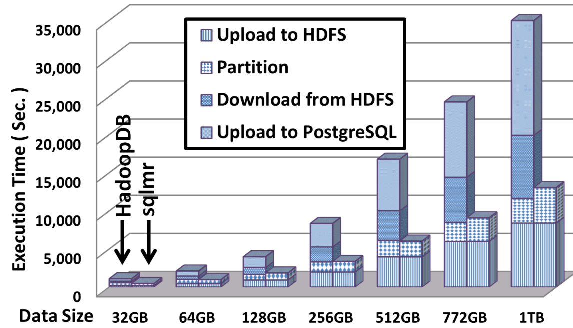

Fig. 17. Comparison of Data Preprocessing time between HadoopDB and SQLMR with large data sizes

with the fault tolerant, heterogeneous cluster, scalable capabilities of MapReduce. Users of SQLMR can write data management programs with familiar query language or to run existing programs without modification. SQLMR provides a compiler to translate a SQL program to a MapReduce program, and execute it in a MapReduce system. To achieve high performance in data processing, we also devise a number of optimization techniques, including efficient data pre-processing, data partitioning, data indexing, query result caching, and optimization of the Hadoop runtime system.

We conducted experiments using the widely used Sysbench benchmark to evaluate both data scalability and system scalability of SQLMR. We compare SQLMR with MySQL, MySQL_cluster and two MapReduce-based database system: Hive and HadoopDB. Our experiment results demonstrate that SQLMR achieves significant improvement in query processing time, with improvement ratio of 13.35 against HadoopDB and 1.41 against Hive for range SELECT queries, and 28.11 against HadoopDB and 1.29 against Hive for JOIN queries. Our experiments also show that SQLMR has good scalability w.r.t. increase in system size.

VII. Acknowledgements

This work is supported in part by National Science Council of Taiwan under grant number NSC-99-2218-E-001-009. We would also like to thank the Academia Sinica Computing Center for providing computing and storage facilities.

References

[1] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, ser. SOSP '03. New York, NY, USA: ACM, 2003, pp. 29–43. [Online]. Available: http://doi.acm.org/10.1145/945445.945450

[2] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, pp. 4:1–4:26, June 2008. [Online]. Available: http://doi.acm.org/10.1145/1365815.1365816

[3] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," in *Proceedings of the 6th conference on Symposium on Opearting Systems Design and Implementation*, ser. OSDI 04, vol. 6. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10.

[4] "Hadoop," http://hadoop.apache.org/.

[5] "Amazon simple storage service," http://aws.amazon.com/s3/.

[6] "Amazon simpledb," http://aws.amazon.com/simpledb/.

[7] "Microsoft sql azure," http://msdn.microsoft.com/en-us/windowsazure/sqlazure/default.aspx.

[8] "Mongodb," http://www.mongodb.org/.

[9] "Couchdb," http://couchdb.apache.org/.

[10] "Cassandra," http://cassandra.apache.org/.

[11] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, ser. SOSP '07. New York, NY, USA: ACM, 2007, pp. 205–220. [Online]. Available: http://doi.acm.org/10.1145/1294261.1294281

[12] "Amazon relational database service," http://aws.amazon.com/rds/.

[13] "Microsoft sql server," http://www.microsoft.com/sqlserver/en/us/default.aspx.

[14] "Nchc bigtable," http://trac.nchc.org.tw/cloud/wiki/BigTable.

[15] "Mysql database," http://www.mysql.com/.

[16] "Mysql cluster," http://www.mysql.com/products/cluster/.

[17] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: a not-so-foreign language for data processing," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '08. New York, NY, USA: ACM, 2008, pp. 1099–1110. [Online]. Available: http://doi.acm.org/10.1145/1376616.1376726

[18] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: a warehousing solution over a map-reduce framework," *Proc. VLDB Endow.*, vol. 2, pp. 1626–1629, August 2009. [Online]. Available: http://portal.acm.org/citation.cfm?id=1687553.1687609

[19] 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, pp. 922–933, August 2009. [Online]. Available: http://portal.acm.org/citation.cfm?id=1687627.1687731

[20] "Oracle database," http://www.oracle.com/us/products/database/index.html.

[21] L.-Y. Ho, J.-J. Wu, and P. Liu, "Optimal algorithms for cross-rack communication optimization in mapreduce framework," In press.

المنارة للاستشارات
www.manaraa.com