ESQP: An Efficient SQL Query Processing for Cloud Data Management

Jing Zhao, Xiangmei Hu and Xiaofeng Meng
School of Information, Renmin University of China
Beijing, China, 100872
{zhaoj, hxm2008, xfmeng}@ruc.edu.cn

ABSTRACT

Recently, the cloud computing platform is getting more and more attentions as a new trend of data management. Currently there are several cloud computing products that can provide various services. However, most cloud platforms are not designed for structured data management. So they rarely support SQL queries directly. Even though some platforms support SQL queries, their bottoms are traditional relational database, therefore, the cost for executing a subquery in RDBS may influence the overall query performance. How to improve query efficiency in cloud data management system, especially query on structured data has become a more and more important problem. To address the issue, an efficient algorithm about query processing on structured data is proposed. Our approach is inspired by the idea of MapReduce, in which a job is divided into several tasks. Based on the distributed storage of one table, this algorithm divides a user query into different subqueries, at the same time, with replicas in cloud, a subquery is mapped to k+1 subqueries. Every subquery has to wait in the queue of the slave where the query data store. To balance the load, our algorithm also takes two scheduling strategies to dispatch the subquery. Besides, in order to reduce the client's long waiting time, we adopt the pipeline strategy to process result returning. Finally, we demonstrate the efficiency and scalability of our algorithm with kinds of experiments. Our approach is quite general and independent from the underlying infrastructure and can be easily carried over for implementation on various cloud computing platforms.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query processing*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*

General Terms

Algorithms

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CloudDB'10, October 30, 2010, Toronto, Ontario, Canada.
Copyright 2010 ACM 978-1-4503-0380-4/10/10 ...$10.00.

Keywords

distributed query, query processing, query transformation

1. INTRODUCTION

With the rapid growth of the amount of data, how to manage massive information becomes a challenging problem. It changes the infrastructure of data storage and generates a new technology called cloud computing. Existing cloud computing systems include Amazon's Elastic Computing Cloud(EC2)[1], IBM's Blue Cloud[2] and Google's GFS[5]. They adopt flexible resources management mechanism and provide good scalability. There are also some open source cloud computing projects, such as Apache Hadoop project's HDFS[9] and HBase[8], which are the open source implementation of Google's GFS and BigTable[4], and Cassandra[7], which brings together Dynamo's[6] fully distributed design and Bigtable's ColumnFamily-based data model.

Although Google's BigTable[4] stores data with table structure, column based storage model and timestamps are designed to improve the flexibility of one record. In other words, column based storage model is just more applicable for unstructured and semi-structured data storage, but not for structured data.

On the other hand, Because the popularity of traditional RDBMS and data warehouse, currently the analytical data of most enterprisers used in business planning, problem solving, and decision support are structured data. Analytical data has its special characteristics: ACID guarantees are typically not needed; Particularly sensitive data can often be left out of the analysis; shared-nothing architecture is a good match for analytical data management. These characteristics of the data and workloads of typical analytical data management applications are well-suited for cloud deployment[14]. At the same time, these analysis data is close to even more than PB level[10]. How to query and analyze on these data is a challenge problem.

Cloud computing platforms contain hundreds and thousands of heterogeneous commodity hardware, and they process workloads and tasks in parallel. This is a typical characteristic of cloud computing infrastructures. When a user submits a query, master nodes in the cluster must decompose the query into subqueries, dispatch them to slave nodes for concurrently processing and merge the results returned from slave nodes. The results of the query can not returned to users until all subqueries on slave nodes execute completely. However, in cloud computing platforms, slave nodes always can not finish subqueries at the same time, some of them may execute more quickly while some may be slower. There-

fore, we have to consider load balance to help us query more efficiently rather than dispatch the same amount of subqueries to each slave node. Meanwhile, as we know, cloud computing platforms always have k replicas of data for fault tolerance, which can also be used for efficient query answering. k replicas of data means that $k+1$ equivalent subqueries on the same data partition can be generated, and we can dynamically assign one of them to the appropriate slave node according to the load of system. In other words, replicas in cloud computing platform contribute to efficient subquery scheduling, which is very essential for query response.

In Summary, this paper makes the following contributions:

- Inspired by the idea of MapReduce, we propose a new efficient SQL query processing algorithm (ESQP) using data replicas in cloud storage.
- We describe how to decompose queries into subqueries according to query operator/oprande pair, which can run in parallel.
- We propose two scheduling algorithms in query procedure to achieve load balancing, and then improve query process efficiency.
- In order to reduce response time of the query, a pipeline strategy is employed when results return.
- We perform a series of experiments on large scale of machine nodes with large volume of data. The experiment confirms that our algorithm is efficient and scalable.

The rest of this paper is organized as follows: Section 2 and Section 3 describe related works and the current query on cloud computing separately. Section 4 presents our efficient SQL query processing algorithms for cloud data management, including query transformation, subquery scheduling and execution, and result return. In Section 5, we present the experimental results to demonstrate the efficiency and scalability of our methods. Finally, we make a conclusion and discuss some possible future work in Section 6.

2. RELATED WORK

2.1 Distributed Query Processing

Query processing problem is a difficult and extensive problem in distributed environments. There are many important aspects of this problem, including query decomposition, data localization, global and local optimization, etc. A detail discussion of each aspect is out of the scope of this paper, and what we want to discuss here is the cost of query processing. As we all know, *total cost* [16] is a good measure of resource consumption. And the total cost includes CPU, I/O, and communication costs in distributed database system. As network becomes faster and faster, the communication cost does not dominate local processing cost. Therefore, many researches consider a weighted combination of these three cost components rather than communication cost merely. Cloud database systems share many properties of distributed and parallel database systems, but scale well into hundreds or thousands of nodes. Although a typical cluster connect large scale of nodes via a high-bandwidth network, the communication cost is quite important due to the huge size of dataset. Data in cloud data management system is always uniformly distributed, so that the larger dataset is, the more communication cost may be produced, especially in join query. Therefore, one of our aim is to minimize communication cost at run time by exploiting the replicated data.

2.2 MapReduce

Google's MapReduce programming model mainly focuses on supporting distributed solution for web-scale data processing[3]. It decomposes data processing into two functions: *map* functions, reading an input key-value pairs and outputting intermediate key-value pairs; *reduce* functions, which merges the intermediate pairs with the same key into the final output. All *map* and *reduce* operations can be performed in parallel by partitioning the input dataset and handling different partitions concurrently by cluster.

This model provides good load balancing, fault tolerance and low communication cost. In order to achieve dynamic load balancing, TaskTrackers are assigned tasks as soon as they finish them. As communication cost component is probably the most important factor considered in distributed query, so that Master schedules *map* tasks on the machine that contains a replica of the corresponding input data. Furthermore, MapReduce programming model spawn backup tasks for the tasks run on slow workers to shorten job completion time and reexecute completed or in-progress *map* tasks and in-progress *reduce* tasks to ensure fault tolerance of workers.

However, this model has its own limitations. Users have to translate their applications into *map* and *reduce* tasks to achieve parallelism. Due to the commonality of this model, it takes sorting as the necessary step before *reduce* function. But this translation and sorting is really unnecessary for some simple SQL operations such as selection and projection. Furthermore, as indicated in [17], complex applications such as join, which requires extra stages of *map* and *reduce*, does not quite fit into this model. The implementation of *map* and *reduce* functions, especially the strategies of functions optimization would get users into trouble.

So we try to employ the basic idea of MapReduce programming model, including partition, single task re-execution, scalability and fault tolerance. We adopts the strategy of this model by decomposing a SQL query into multiple subqueries according to the corresponding data replicas. Meanwhile, we take advantages of techniques of traditional DBMS and parallel database system.

2.3 MapReduce and SQL

There are some work on combining ideas of MapReduce with database system. Typical examples include Apache's Hive[12], Yale's HadoopDB[11], Microsoft's SCOPE[13], etc. However, some of these work focus on system hybrid, while others focus on the SQL-like interface. HadoopDB[11] provides a hybrid solution at system level, using MapReduce framework for query distribution, inheriting the scheduling from Hadoop for fault tolerance and coordination ability, and take PostgreSQL servers as database engine for query processing. SCOPE and Hive separately provides a kind of SQL-like language. They integrate SQL-like language into MapReduce-like software to increase user productivity and system efficiency.

We do not use any database system for query processing but we employ some key techniques, including index and

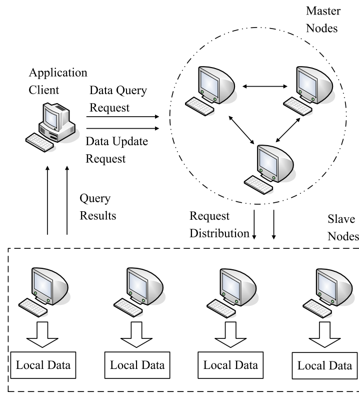


Figure 1: Framework of Query Processing in Cloud

pipeline, to improve the efficiency of subquery processing. Moreover, although we employ the basic idea of MapReduce, we design a structure for query distribution and processing, which does not base on or combine with Hadoop, so that we can take control of the whole progress of query processing and ESQP can be easily carried over for implementations on various of cloud computing platforms

3. QUERY IN THE CLOUD

As we know, a cloud computing platform(a cluster) consisting of hundreds or thousands of PC is responsible for data computing and storage. As Figure 1 shows, there are two types of nodes in the cluster: master nodes and slave nodes. Master nodes store some meta data about the whole cluster while slave nodes store the regular data. In other words, slave nodes store data records and their replicas for security. So the query on the cloud platform is different from central or parallel database. In the cloud platform, client query is often presented against the master nodes. After that the mater nodes decide which slave nodes are relevant to the query and then the query is passed to the slave nodes to do the query processing directly. The general query processing in cloud computing platform is in Figure 1. So a typical query in the cloud computing platform can be divided into two phases: locate the slave nodes which stores the relevant data and process query on the slave nodes directly. The procedure cloud be expressed as algorithm 1.

Algorithm 1 Process query on cloud

```
1: procedure Set processQuery(Query q)
2:     Set nodes = empty;
3:     nodes.add(getRelativeNodes(q));
4:     Set results = empty;
5:     for (each node n in the nodes) do
6:         results.add(n.retrieveRecords(q));
7:     end for
8:     returnresults;
9: end procedure
```

From the above discussion,we can see that:

- The query processing problem is much more difficult in cloud computing environments than in centralized ones, because the query processing is not complete by one machine.
- The huge scale of cluster leads query processing in cloud environment problem be different from in parallel ones.
- Most of cloud computing systems decide which replica of data to be used for query before query processing. These predefined replica may result in more cost in some cases.

In order to query efficiently, we have to improve query processing by some means. In the following part of the paper, we will discuss how to query more efficiently on the cloud platform. The details will be listed below.

4. QUERY PROCESSING

As we mentioned earlier, the key problems of structured query processing in cloud database system lie in structured query translation, load balance of the whole system and data transfer among nodes. In order to query efficiently, our approach employs four key ideas:

- We exploit replicas in cloud database system for query translation in order to provide better alternatives for scheduler.
- A scheduler and scheduling metric are developed to ensure load balancing and reduce the total runtime of each query.
- We adopt DigestJoin [15] to reduce the size of data that has to be transferred in join operation.
- In order to avoid client's longtime waiting, pipeline and ASAP are employed in subquery processing model.

In the remainder of this section, we discuss the details of our design. First, we describe the data and query model the algorithm is optimized for, and then present the execution of query, including query transformation, subquery scheduling and execution, and result returning. Finally, we focus on transformation of query for a number of relational operators.

4.1 Data and query model

Due to our method in general supporting common SQL query in cloud computing system, the data need to be distributively stored in cloud system. Generally, a table is a collection of records, each of which is identified by a unique key, and each table is divided into n parts, each part replicated k times and are stored in different nodes in cluster. k is usually much smaller than the number of nodes in cluster while $k = 2$ for most cloud computing system. The meta information, such as the storage information about each replica of each partition. These information is reported to master nodes in cluster, which are in charge of subqueries scheduling. Typical cloud computing systems usually provide good support for key/value based queries, therefore, we assume that the data in cloud computing system has an index in key field, based on which we provide an efficient join processing method.

We focus on providing low latency for read-only SQL query, including generalized selection, projection, aggregation and join. Generalized selection means retrieving not only a single record by primary key but also a number of records satisfied any condition in any fields of a table. All these operations need to scan all data without index. Therefore, low latency means that the first record of query results should return as soon as possible to avoid client's longtime waiting.

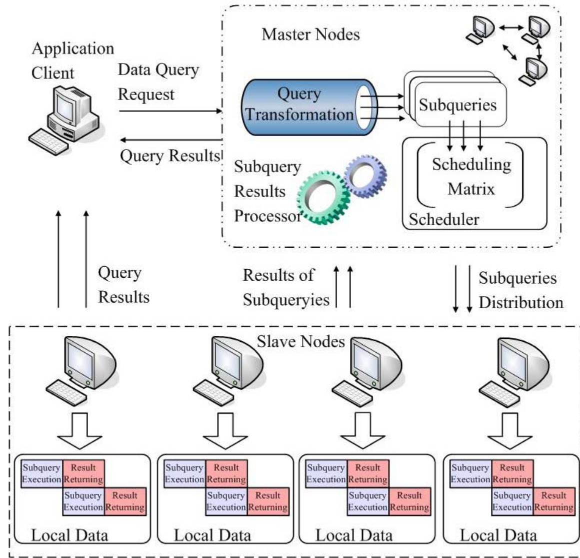


Figure 2: Execution Overview

4.2 Query execution

A key goal of our processing algorithm is to minimize the response time of a query in cloud database system. As figure 1 shows, in the cloud platform, master nodes store some meta data about the whole system and are in charge of distributing query to coordinate slave nodes. When a slave node receives the request from master nodes, it retrieves data locally or communicate with other slave nodes for relevant data according to the operation type and stores results locally. Results are returned to client directly after result generating.

From this progress we can see that the key components of query processing which influence latency are:

- Query transformation: A user query should be transformed into a set of independent subqueries that can be execute parallelly on nodes of cluster. Parallelism can significantly reduce query latency in all. Local execution is another important aspect for low latency, so that we try to make sure that the percentage of subqueries that could be executed locally as large as possible.

- Query dispatch: Assignment of subqueries plays an important role in query processing. System can achieve load banlancing via good and reasonable scheduling of subqueries, and then minimize the total runtime of the query.

- Subquery execution: Slave nodes employ the idea of pipeline to accelerate a number of subqueries processing rather than repeat the following three steps, receive subquery, process subquery and return result serially, parallel execution of previous result returning and current query processing can save much time in that we don't need to wait for results transfer.

We present the details of execution in the following. Figure 2 shows this in diagram form.

4.2.1 Query transformation

Each user query is transformed into a set of subqueries according to the partition of involved tables, each of which can be executed independently. There are kinds of SQL operators, which lead to various transformations. But all kinds of transformations are based on the partitions of tables and their replicas. We consider a user query as an *operator/operand* pair. *Operator* includes generalized selection, projection, aggregation and join, and *operand* here means the data blocks of table where operators retrieve from. Because the operator is constant to the specific SQL operation, we generate subqueries by modifying the operands of the original query. The operand can be classified into two categories: single table for the first three operators and multiple tables for join operator. We maintain a list of subqueries, each of which has two kinds of transformed operands set for multiple tables, while one set for single table. The procedure could be expressed as algorithm 2, and now we present the transformation of operands in detail:

Single Table: Operands of the first three kinds of operators are single table. We simply replace the original operand, a single table in FROM clause, with a number of location sets of replicas to create the subqueries. Each set contains all copies of a partition in one table. For instance, table R is divided into m parts R_1, R_2, ..., and R_m with a backup factor $k = 2$, hence we create m sets, each of which is composed of R_i $(i = 1, 2, ..., m)$, R_{i_1}, R_{i_1}, ..., R_{i_k}, where R_{i_j} is the copy of R_i. These subqueries can be run in parallel, locally and independently.

Multiple Tables: We consider two tables here because operation on multiple tables can be split into a set of operations on two tables. The transformed operands are classified into two kinds: one is partitions of tables without intersection and the other is partitions of tables with intersection, which implies that there is a slave machine that store two replicas of blocks belonging to different tables. Therefore, we create three sets for a subquery to store the location information of two blocks, one of which shows the location of replicas from two blocks that stored in the same slave node, called $intersectionset$, and the other two separately express the location information of replicas from different tables, called $replicaset$. It is necessary to state that, employing the basic idea of DigestJoin[15], the operator on $intersectionset$ operand is original joining, while the operator of subquery without intersectant replicas includes not only JOIN but also EXTRACT and RELOAD, where EXTRACT means extracting digest data from nearest node before JOIN operation and RELOAD refers to reloading relevant data to compose query result after JOIN operation.

For a cluster of n slaves, take table R joins table S for example. As our data model stated above, supposing R_i is a part of R and S_j is a part of S, then we have replicas R_{i_1}, R_{i_2},..., R_{i_k} for R_i and S_{j_1}, S_{j_2}, ..., S_{j_k} for S_j. We suppose that R_i, R_{i_1}, R_{i_2},..., R_{i_k} are stored in $slave_i$, $slave_{i_1}$, $slave_{i_2}$,..., $slave_{i_k}$ and S_i, S_{i_1}, S_{i_2},..., S_{i_k} are stored in $slave_j$, $slave_{j_1}$, $slave_{j_2}$,..., $slave_{j_k}$. We declare that R_i intersect S_j, if and only if there is any $i_s == j_t$, where $s, t = 1, 2, ..., k$. We store i_s in intersection set when intersection occurred and set the other sets into empty, while assigning i, i_1, i_2,..., i_k to one replica set, j, j_1, j_2,..., j_k to anther and intersection set is set to empty when there is no intersection. Figure 3 shows how to maintain these informa-

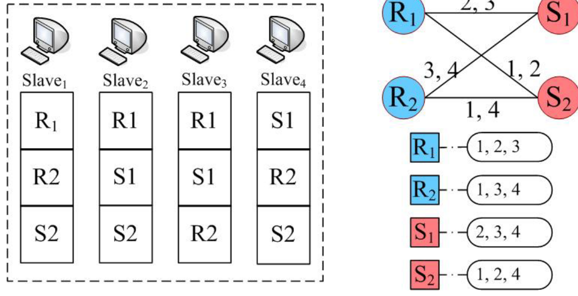


Figure 3: Totally 3 copies of partitions R_1, R_2, S_1 and S_2 are separately stored in four slave nodes. The edges of bipartite graph shows the intersection information of R_i and S_j, and lists in the the corner express location of replicas of each partition, each of which is assigned to coordinate subquery when intersection set is empty.

tion for partitions R_1, R_2, S_1 and S_2 with replica factor of $k = 2$.

Algorithm 2 Query Transformation

```
1: procedure Set transformQuery(Query, q)
2:     Set subqueries = empty;
3:     if (q.type is JOIN) then
4:         Set partsA = getPartitionsOfTable(q.tableA);
5:         Set partsB = getPartitionsOfTable(q.tableB);
6:         for (each part p in partsA) do
7:             Set replicasA = getReplicasOfPart(p);
8:             for (each part p in partsB) do
9:                 Set intersection = empty;
10:                Set locationsA = empty;
11:                Set locationsB = empty;
12:                Set replicasB = getReplicasOfPart(p);
13:                if (replicasA and replicasB intersect) then
14:                    intersection.add(location of intersected replicas);
15:                else
16:                    locationsA = replicasA;
17:                    locationsB = replicasB;
18:                end if
19:                subqueries.add(intersection,locationsA, locationsB);
20:            end for
21:        end for
22:    else
23:        Set parts = getPartitionsOfTable(q.table);
24:        for (each part p in parts) do
25:            Set blocks = getReplicasOfPart(p);
26:            subqueries.add(blocks);
27:        end for
28:    end if
29:    return subqueries;
30: end procedure
```

4.2.2 Subquery Scheduling

While the subqueries can be executed in parallel, according to the expression above, the number of subqueries is equivalent to the number of table's partitions or the product of numbers of two tables' parts, which far exceeds the number of nodes in cloud platform, and different performances of machines in the cluster lead to heterogeneous load, so we develop a scheduler and scheduling matrix to coordinate the

	a	b	c	d	L_i		$(L_i-a)(L_i-c)$	
$slave_1$	1	0	1	0	2		1	1
$slave_2$	0	1	0	0	1	$\Rightarrow$	1	1
$slave_3$	1	1	0	1	3		2	3
$slave_4$	0	0	1	1	2		2	1

Figure 4: A constructed matrix with 4 slave nodes and 4 subqueries to be scheduled. Parameter L_i of each slave node represents the number of subqueries waiting for slave node $slave_i$.

execution of subqueries, which dynamically changes loads on slave nodes to minimize the response time of the query.

In general, each slave only executee subqueries on replicas of parts that it is stored locally, particularly for JOIN operator, a subquery is dispatched to the slave store one of its operands. And every subquery is composed of a operator and a operand set which contains location information of subquery, in other words, we regard a subquery as a set of at least $k + 1$ equivalent subqueries. Therefore, we exploit a scheduling matrix to decide which subqueries are given to a slave node. We take subqueries as horizontal axis and slave nodes as vertical axis. The element of this matrix are numbers in a union $\{0, 1, 2\}$, where $M_{ij} = 0$ means that the partition where subquery $subQ_j$ retrieve does not have any copies stored in slave node $slave_i$, $M_{ij} = 1$ expresses that one of the copies of subquery $subQ_j$'s partition is stored in $slave_i$ or both retrieved tables' partition have copies in $slave_i$, and $M_{ij} = 2$ implies subquery's original operator is JOIN and only one table involved has copies stored in $slave_i$. In the other word, we consider a row of the matrix as an unordered of subqueries which are waiting for dispatching. Figure 4 shows the scheduling matrix for a cluster consisted of 4 slave nodes, with a backup factor $k = 1$. In this figure, each slave node has a parameter representing the number of subqueries that are waiting for execution on a certain node.

The greedy scheduler grants a subquery as soon as possible to a slave node when it becomes free. The system achieves load balancing effectively through such an approach because fast slave nodes can take on more workload to lighten slower nodes. Moreover, scheduler creates a subquery list which consists of the operator, operand and the status of this query, including *waiting*, *processing*, *processed*, *gettingResult*, and *finished*, and a status list of slave nodes. It communicates with slave nodes according to these two lists and scheduling matrix through two message types. The scheduler sends a slave node a *dispatch* message, which notifies it to start processing subquery. As a subquery is assigned to slave node, the scheduler changes the status of it from *waiting* to *processing* and removes all equivalent subqueries from scheduling matrix. And when slave node has finished execution of current subquery, it returns a $free$ message to the scheduler, which will change the status of the subquery to *processed* and reset the slave node's state to *free*. The procedures are described as algorithm 3, 4 and 5.

There are kinds of scheduling algorithms. We have implemented two: **Random Scheduling.** Whenever a slave node becomes free, our scheduler randomly chooses a subquery from its waiting queue. **Global Scheduling.** We adopt the idea of global optimization. A subquery which balances all waiting queues is chosen. Before choosing a subquery, we compute the length of waiting queue for each slave

Algorithm 3 Initialize Scheduling Matrix

```
1: procedure MATRIX INITIALIZEMATRIX(ListsubQueries)
2:     Matrix M = empty;
3:     for (each subquery subq_i in subqueries list) do
4:         if (subq.operator is a unary operator) then
5:             for (each location loc in subq.operand.locs) do
6:                 M_loci = 1;
7:             end for
8:         else
9:             if (subq.operand.intersectionset is empty) then
10:                for (each loc in subq.operand.firstSet) do
11:                    M_loci = 2;
12:                end for
13:                for (each loc in subq.operand.secondSet) do
14:                    M_loci = 2;
15:                end for
16:            else
17:                for (each loc in subq.operand.intersection) do
18:                    M_loci = 1;
19:                end for
20:            end if
21:        end if
22:        subq.status = waiting;
23:    end for
24:    return M;
25: end procedure
```

Algorithm 4 Subquery scheduling

```
1: procedure BOOLEAN SCHEDULE(List subQueries)
2:     InitializeMatrix(subQueries);
3:     while (scheduling matrix != 0) do
4:         for (each free slave slave_i) do
5:             subq = chooseSubquery(slave_i);
6:             dispatch(subq,slave_i);
7:             for (each element in column set of subq) do
8:                 element=0;
9:             end for
10:            subq.status = processing;
11:            slave_i.status = busy;
12:        end for
13:    end while
14:    return true;
15: end procedure
```

Algorithm 5 Select a subquery

```
1: procedure SUBQUERY SELECT(int slaveLoc)
2:     SubQuery subq = empty;
3:     double variance = POSITIVE INFINITY;
4:     List length = QueueLength(Matrix M);
5:     for (each subquery q in waiting list) do
6:         generate length list lengths;
7:         if (variance > varianceOf(lengths)) then
8:             variance = varianceOf(lengths);
9:             subq = q;
10:        end if
11:    end for
12:    return subq;
13: end procedure
```

nodes by removing every possible subquery, which comes from the waiting queue of free slave node, thus we have l length lists, where l is the number of possible subqueries. The variance of each list is calculated and the subquery corresponding to the smallest variance is assigned to the slave node.

As is shown by Figure 4, L_i represents the number of subqueries waiting for distribution of each slave node. Supposing $slave_1$ is free, a and c are two probable subqueries in its waiting queue. Thus we separately pre-compute the length of each slave node's waiting queue removing a and c. **Random** would randomly assign a or c to $slave_1$, while **Global** would chose a for load balance of the system due to the variance of $L_i - a$ is 0.33, which is smaller than $L_i - c$'s variance 1.

4.2.3 Subquery Execution and results returning

When a slave node receives *dispatch* message sent by scheduler, it starts the execution thread, storing results locally. Instead of returning results as quickly as it generates, the slave node sends the $free$ message back to scheduler to report that the subquery is completed, and the master node creates a result handling thread to get back the results asynchronously. The slave node processes subqueries in full sail rather than being distracted by transportation of results, thus the subquery execution on slave nodes seems a pipeline, returning of the results of previous subquery and current subquery's processing go simultaneously, which reduces the overall runtime of a query in a sense. The subquery execution and result returning procedures are as algorithm 6 and 7, and the implementation of algorithm 6 is invoked by slave nodes repeatedly while the algorithm 7's implementation is called by scheduler.

Algorithm 6 Subquery Processing

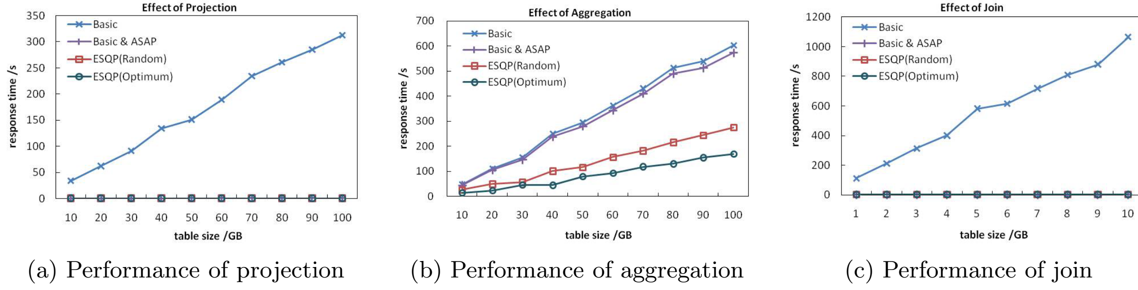
```
1: procedure VOID PROCESSSUBQUERY(SubQuery subq)
2:     Result results = getResult(subq);
3:     String fileName = storeResults(results);
4:     send free message back to scheduler;
5: end procedure
```

Algorithm 7 Result Returning

```
1: procedure RESULTS GETRESULTS(SubQuery subq, String loc)
2:     subq.status = gettingResult;
3:     Results middleR = fetchResult(subq,loc);
4:     return middleR;
5: end procedure
```

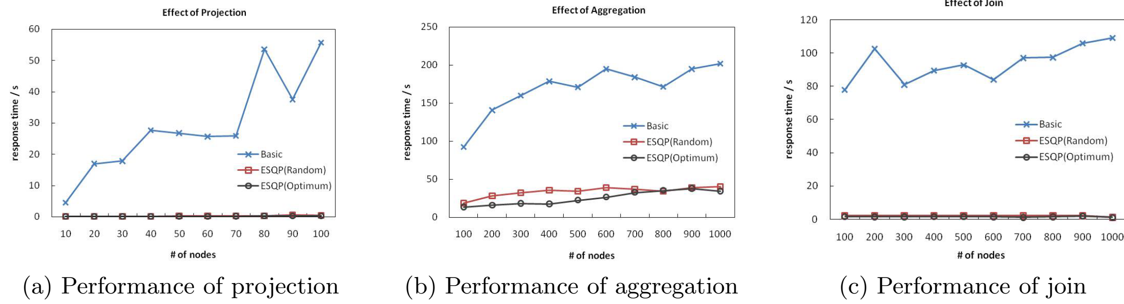
4.2.4 Result Representation

In order to minimize the *response time* of the query, the main idea of our approach is that returning the results to clients as soon as possible, even if only one record of all results is ready. Some of results could be returned to users once the result processor get them, for example, the results of generalized selection, projection and joins, but in some cases, the results of subqueries are not the just result of original query, so that some retreatments are required, as aggregation. The combination involves *order* and *aggregate*:
Order: Although the results of the subqueries are well sorted locally on slave nodes, a global sorting must be carried out to form an ordered result of original query.
Aggregate: J. Gray et al.[18] classified aggregate function F() into three categories: *Distributive*,*Algebraic* and *Holistic*. We only consider first two types in SQL aggregate operators. These aggregate functions are equivalent to aggregation of original functions, such as *COUNT()*, *MIN()*, *MAX()*, *SUM()*, or combination of additional functions, for instance *AVERAGE()*. The result processor compute the aggregate result of query according to different aggregate functions. Take *MIN()* and *AVERAGE()* for example, result processor take the minimum of values fetched from slave



(a) Performance of projection (b) Performance of aggregation (c) Performance of join

Figure 5: Query response time for different queries by scaling up the size of table



(a) Performance of projection (b) Performance of aggregation (c) Performance of join

Figure 6: Query response time for different queries by scaling up the number of slave nodes

nodes to find the final minimum value, while slave nodes report *SUM()* and *COUNT()* of subset for *AVERAGE()* function and result processor adds these two components and then divides to produce the global average.

4.3 Fault Tolerance

Inspired by the approach taken by MapReduce[3], the fault tolerance strategy is to restart all subqueries which are marked as *processing*, *processed*, or *gettingResult* when the corresponding slave node is out of contact. Although our algorithm can deal with the failure of slave nodes, we will take this as a future work for lack of space.

5. PERFORMANCE EVALUATIONS

We now evaluate the performance and scalability of our query processing in cloud databases. Testing for query processing breaks down into two suites: efficiency and scalability tests, to demonstrate the effect and scalability of of our query processing technique, and load balancing tests, to test our subquery execution scheduling.

5.1 Experiment Setup

Our testing infrastructure includes 11 machines which are connected together to simulate cloud computing platforms - 1 master and 10 slaves. Each contains a Inter Core 2 2.33GHz CPU, 8GB of main memory and 2TB hard disk. Machines ran Ubuntu 9.10 Server OS. Communication bandwidth was 1Gbps.

We use this infrastructure to simulate different size of cloud computing systems. We conducted 10 simulation experiments, ranging from 100 nodes to 1000 nodes. Each time 100 more nodes are considered to be added into the cloud computing system. Our algorithm is implemented in Java. We use a small telecom CDR data sample to generate 100 GB of data with about 200 bytes per tuple. These data are used as 10 different sets, ranging from 10 GB to 100 GB with 10 GB increment. Three typical queries are hired to testify our algorithm's efficiency and scalability, including projection, aggregation and join requirements. And for join query the dataset is a little different from above dataset. We use data from 1 GB to 10 GB as a table and join two tables with the same size.

5.2 Performance of typical queries

We design two sets of experiments to evaluate the performance of the query processing of three typical queries. For each query, we separately scale up the size of data, which indicates the total number of subqueries of one original query due to the fixed size of one data block in the system, and the number of slave nodes in the cluster. First of all, for a fixed 50 nodes cluster, we increase the size of data, and then for a fixed 10 GB data, we scale up the number of slave nodes. Response time, which is the interval between the query started and the first result of the query returned to the user, is used as the metric in the experiments. Respectively, we use four methods to execute each typical query, including basic method, ASAP based basic method, random optimum scheduling method and global optimum scheduling method. Basic method decomposes query into multiple certain subqueries rather than $k + 1$ equivalent subqueries for each table partition, and it is marked *busy* until all results are returned to master nodes. And finally the master nodes return the result of original query after all subqueries execution finished, while ASAP based basic method return results as soon sa possible. Optimal methods not only take the advantage of table's multiple backups, but also employ ASAP approach, pipeline and scheduler to reduce the response time as best as we can. All results are obtained based on 5 runs.

Figure 5 and 6 show our results straightforwardly. Results show very good performance. Random and Global ESQP have similar performance in our dataset in that our data is distributed uniformly. As can be seen in figure 5, the cost of optimal method answering the projection and join query in 50 nodes and 100 GB only is less than 1 second, and aggregation query is only 100-300 seconds, which shows that our method is very efficient. ASAP based basic method also has good performance in figure 5(a) and 5(c) because the response time is determined by query decomposition and the fastest subquery execution, where our method has no obvious superiority. But according to our experimental records, the total cost of a query execution in this method is much more than ESQP method.

Figure 5 and 6 also illustrate the scalability of our methods. These graphs show that our distributed efficient SQL query processing method scales almost linearly with the table size or the number of nodes. Benefiting from pipeline strategy, when the queries don't have aggregation, we return the result of query as soon as we get it from subquery execution node, and we stop the mission timer at the point that first result is received by client. Therefore, the response time of this kind query is only influenced by query decomposition, which is always dominated by subqueries dispatching time. On the other hand, the response time of aggregation query is consist of query decomposition time, query dispatch time and time of the lowest subquery's execution and result return. Although the scale up of the cluster makes nonunhiform distribution of subqueries in basic method that leads to the load unbalance problem, which causes the relevant curves in figure 6 are not smooth or monotonic, the figure shows our method is high available and scale to hundreds of nodes, and figure 5 shows the performance of our method is very good when data size is scaling up.

6. CONCLUSION AND FUTURE WORK

In this paper, we presented a newly more efficient query algorithm to deal with SQL query. According to different kinds of queries, we adopted different subqueries dispatch. Besides, the algorithm took advantage of the idea of divide and conquer. In order to get higher efficiency, we not only used scheduling algorithms to get load balance, but also we utilized pipeline technique to process result return. Finally, we proved the efficiency and scalability of our approach with vast experiments.

For future work, as the number of slave nodes increases, although our query processing algorithm has very good scalability, the query cost does not reduce lineally because of the computation of the large matrix. Therefore, we will study more advanced query schedulers to make our algorithm more scalable, and do more research on large-scale concurrent queries answering which brings further scalability problems. Besides, currently our scheduling algorithms choose subquery according to a heuristic method-variance to achieve maximum load balancing. So we also plan to research a more accurate measure of load balance.

7. ACKNOWLEDGMENTS

The authors thank anonymous reviewers for their constructive comments. This research was partially supported by the grants from the Natural Science Foundation of China (No.60833005); the National High-Tech Research and Development Plan of China (No.2007AA01Z155čň2009AA011904); and the Doctoral Fund of Ministry of Education of China (No. 200800020002).

8. REFERENCES

[1] M. Lynch. Amazon elastic compute cloud (amazon ec2). [Online]. Available: http://aws.amazon.com/ec2/
[2] IBM. Ibm introduces ready-to-use cloud computing. [Online]. Available: http://www-03.ibm.com/press/us/en/pressrelease/22613.wss
[3] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, pp. 107–113, January 2008.
[4] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber, "Bigtable: A distributed storage system for structured data," in *Proceedings of the 7th Conference on USENIX Symposium on Operating Systems Design and Implementation*, Seattle, Washington, November 2006, pp. 205–218.
[5] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *Proceedings of SOSP'03*, New York, USA, December 2003, pp. 29–43.
[6] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazons highly available key-value store," in *Proceedings of the 21st ACM Symposium on Operating Systems Principles(SOSP '07)*, Washington, USA, October 2007, pp. 205–220.
[7] Cassandra. [Online]. Available: http://incubator.apache.org/cassandra/
[8] HBase. [Online]. Available: http://hadoop.apache.org/hbase
[9] Hadoop. [Online]. Available: http://hadoop.apache.org
[10] C. Monash. The 1-petabyte barrier is crumbling. [Online]. Available: http://www.networkworld.com/community/node/31439
[11] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz and A. Rasin, "HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads," in *Proceedings of VLDB'09*, Lyon, France, August 2009, pp. 922–933.
[12] A. Thusoo, J. Sen 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," in *Proceedings of VLDB'09*, Lyon, France, August 2009, pp. 922–933.
[13] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver and J. Zhou, "SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets," in *Proceedings of PVLDB'08*, Auckland, New Zealand, August 2008.
[14] D. J. Abadi, "Data Management in the Cloud: Limitations and Opportunities," in *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 2009, pp. 3–12.
[15] Y. Li, S. T. On, J. Xu, B. Choi, and H. Hu, "DigestJoin: Exploiting Fast Random Reads for Flash-based Joins," in *Proceedings of the 10th International Conference on Mobile Data Management (MDM '09)*, Taipei, Taiwan, May 2009, pp. 152–161.
[16] M. S. Sacco and S. B. Yao, "Query Optimization in Distributed Data Base Systems," in *Adcances in Computers*, New York, United States, 1982, pp. 225–273.
[17] R. Pike, S. Dorward, R. Griesemer and S. Quinlan, "Interpreting the Data: Parallel Analysis with Sawzall," in *Scientific Programming Journal*, 2005, pp. 227–298.
[18] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow and H. pirahesh, "Data cube: A relational aggregation operator generalizing group-by, cross-tabl, and sub-totals," in *J. Data Mining and Knowledge Discovery*, 1997, vol. 1, pp. 29–53.