J Inf Process Syst, Vol.13, No.3, pp.518~532, June 2017
https://doi.org/10.3745/JIPS.01.0016

ISSN 1976-913X (Print)
ISSN 2092-805X (Electronic)

JOURNAL OF INFORMATION PROCESSING SYSTEMS JIPS

Development of a CUBRID-Based Distributed Parallel Query Processing System

Hyeong-Il Kim*, HyeonSik Yang**, Min Yoon***, and Jae-Woo Chang**

Abstract
Due to the rapid growth of the amount of data, research on bigdata processing has been highlighted. For bigdata processing, CUBRID Shard is able to support query processing in parallel way by dividing the database into a number of CUBRID servers. However, CUBRID Shard can answer a user's query only when the query is required to gain accesses to a single CUBRID server, instead of multiple ones. To solve the problem, in this paper we propose a CUBRID based distributed parallel query processing system that can answer a user's query in parallel and distributed manner. Finally, through the performance evaluation, we show that our proposed system provides 2–3 times better performance on query processing time than the existing CUBRID Shard.

Keywords
CUBRID, Distributed Parallel Environment, Query Processing

1. Introduction

Due to the rapid growth of the amount of data, research on bigdata processing has been highlighted [1-6]. To extract valuable information from the bigdata, a huge amount of computing resources and efficient bigdata management system are essential. As a result, research on analytical bigdata processing has been done to effectively analyze the bigdata. However, the researches have some problems that they support only limited types of data formats for their applications and require expensive equipment to establish their computing environment. Meanwhile, NoSQL-based researches, such as Hadoop [7], MongoDB [8] and Cassandra [9], have been performed. However, NoSQL has a problem that it cannot guarantee data consistency while supporting partition tolerance and availability. As a result, much attention has been paid to RDBMSs for bigdata processing.

CUBRID Shard [10] is a RDBM (relational database management system) that was designed to deal with bigdata. CUBRID Shard divides data into multiple CUBRID servers by applying horizontal partitioning technique. By distributing the database, CUBRID Shard can process queries of a large number of users in parallel. However, CUBRID Shard can answer a user's query only when the query is required to gain accesses to a single CUBRID server, instead of multiple ones. In other words, it is

Manuscript received January 29, 2015; first revision November 6, 2015; second revision November 23, 2016; accepted November 30, 2016.
Corresponding Author: Jae-Woo Chang (jwchang@jbnu.ac.kr)
* The 1st Missile Systems PMO, Agency for Defense Development, Daejeon, Korea (hikim@add.re.kr)
** Dept. of Information and Technology, Chonbuk National University, Jeonju, Korea (gustlr1222@gmail.com, jwchang@jbnu.ac.kr)
*** The 1st R&D Institute - 4th Directorate, Agency for Defense Development, Daejeon, Korea (myoon@jadd.re.kr)

possible for CUBRID Shard to answer multiple queries in a parallel way by using the database distributed into several CUBRID servers, but it is impossible to answer a single query in a parallel way by using the distributed database. Moreover, CUBRID Shard has inconvenience because a user should state a '*shard_hint*' in the SQL query whenever the user wants to issue a query.

To tackle the problem, in this paper we propose a CUBRID based distributed parallel query processing system that can answer a user's query in parallel and distributed manner. Because the proposed system runs on RDBMSs, users who are experienced with SQL statements can easily deal with the bigdata through SQL queries. Besides general SQL statements, the proposed system can process the aggregation queries (e.g., *min*, *max*, *count*, *sum* and *average*) that have not been dealt with for the distributed parallel data processing.

The rest of this paper is organized as follows. Section 2 introduces related work. In Section 3, we present the overall query processing procedure of the proposed system. In Section 4, we compare our proposed system with the existing CUBRID Shard. Finally, we conclude this paper with future work in Section 5.

2. Related Work

NoSQL systems are widely used for bigdata and real-time web applications. Hadoop [7], MongoDB [8], and Cassandra [9] provides us a way for storing and managing unstructured data. NoSQL approaches provide simple design, horizontal scaling capability, and high availability. The data structures considered by NoSQL differ from those used in relational database systems, thus making some operations faster in NoSQL [11-13]. However, NoSQL has a problem that it cannot guarantee data consistency while supporting partition tolerance and availability. However, the use of low-level query languages, the shortage of standardized interfaces, and high maturity of the existing RDBMS have become the barriers to the wider adoption of NoSQL.

As a result, much attention has been paid to RDBMSs for bigdata processing. CUBRID [14] and CUBRID Shard [10] are typical RDBMSs. First, CUBRID is a relational database that provides high accuracy, predictable automatic fail-over and fail-back properties. In addition, there is no service down time even during system maintenance (e.g., OS/software upgrade and device replacement/expansion). However, CUBRID is inefficient when analyzing bigdata because it is optimized on a single machine.

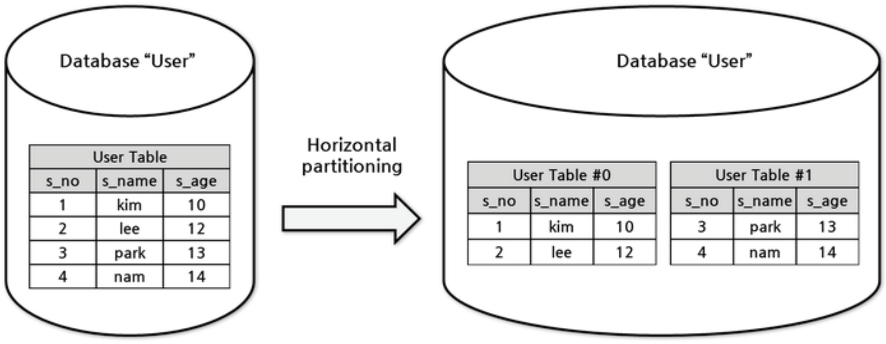


Fig. 1. Horizontal partitioning of the CUBRID Shard.

Secondly, to tackle the shortcomings of CUBRID, CUBRID Shard divides database based on a horizontal partitioning method. For example, as shown in Fig. 1, 'User' table in 'User' database can be partitioned into 2 'User' tables (e.g., 'User' table #0 and #1). CUBRID Shard allows distributing data into unlimited number of database shards (or servers). Developers can put their own library into the system to compute the *SHARD_ID* using a complicated algorithm. A third-party management tool is not required. With CUBRID Shard, application developers are not asked to modify their application to partition their databases into CUBRID Shards because it is automatically supported by the system. CUBRID Shard provides efficient query processing, distributed load balancing and statement pooling. In addition, it requires reasonable costs for the configuration of multiple master and slave database nodes. However, CUBRID Shard can answer a user's query only when the query is required to gain accesses to a single CUBRID server, instead of multiple ones. Therefore, CUBRID Shard cannot support a Join operation that is an essential one to deal with the bigdata. Moreover, CUBRID Shard has inconvenience because a user should state a '*shard_hint*' in the SQL whenever the user wants to issue a query.

Meanwhile, recent works on analytical processing are as follows. First, Saravanan et al. [15] proposed an efficient bigdata processing technique by designing a pipelining scheme on the multi-core environment. They designed left-right (LR) algorithm to reduce stalls in pipelined processors. The main advantage of the LR algorithm is that it can support the quick data processing for a large amounts of data. Second, Li et al. [16] proposed a data mining system using an index on the spatial bigdata. They not only use an R-tree-based global tree to organize real-time location data, but also utilize a B-tree-based local tree to manage historical data. Both index methods can efficiently handle location-based queries for monitoring by using JSON query. Finally, Lee et al. [17] proposed a model to extract medical information from big data using continuity of care document. The proposed model can support effective management and provision of medical data because it utilizes a convergence data model based on characteristics and semantic relations of medical data. However, recent works consider only limited types of data formats for their special applications.

3. CUBRID-Based Distributed Parallel Query Processing System

This section describes our proposed system that supports parallel query processing on the distributed CUBRID. The proposed system can aid users who are experienced with SQL to easily deal with the bigdata through SQL queries. In addition, the system can process the aggregation queries that have not been dealt with for the distributed parallel data processing.

3.1 System Architecture

Fig. 2 shows the overall architecture of our CUBRID based distributed parallel query processing system. We adopted the system architecture proposed in our previous work [18]. The system consists of four components; *communication component*, *query analysis component*, *metadata retrieval component* with meta tables, and *query result merge component.*

First, a *communication component* is responsible for data transmission with a user or CUBRID servers. The transmitted data are SQL query and database connection information such as *dbName*, *ip*, *port*, *userID* and *password*. Second, a *query analysis component* is responsible for the parsing of SQL statements from a user. From a SQL statement, the component obtains table names that are necessary when retrieving meta tables. The component determines a type of the query, like *insert*, *select*, and *aggregation*. If the query type is determined as an *aggregation* query, especially *average*, the component rewrites a query statement to process it on multiple database servers. This is because it is impossible to directly calculate the final result from the *average* results received from CUBRID servers. Therefore, the component rewrites the query statement with *sum* and *count* operations instead of *average* operation. By utilizing these *sum* and *count* results, the component can calculate the actual *average* result. Third, the *metadata retrieval component* is responsible for handling three meta tables, like *MinMaxTable*, *SearchTable*, and *IpPortTable*. Each table is defined below.

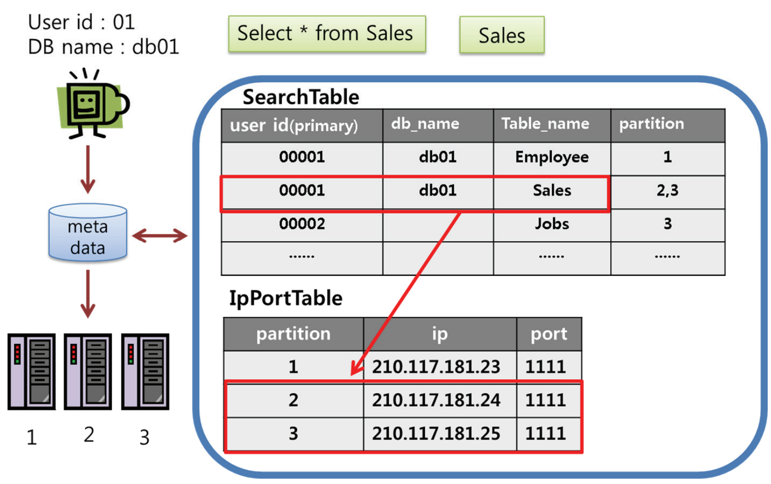
Definition 1. *MinMaxTable*. A meta table which stores information required for data insertion. The table consists of {*dbName*, *partition*, *tableName*, *column*, *min*, *max*}.

The *dbName* means a name of a database. The *column* stands for the name of the column that is used when horizontally partitioning the *tableName* table. The *partition* represents a CUBRID server maintaining records where the values corresponding to the *column* are between *min* and *max*.

Definition 2. *SearchTable*. A meta table which stores information required when retrieving data stored in the distributed CUBRID servers. The table consists of {*userID*, *dbName*, *tableName*, *partition*}.

By utilizing this table, we can confirm partitions (e.g., CUBRID servers) storing the *tableName* table required for processing the query of the *userID*. The *IpPortTable* is referenced by both *MinMaxTable* and *SearchTable* because the table contains connection information of each *partition*.

Definition 3. *IpPortTable*. A meta table which stores the connection information of CUBRID servers. The table consists of {*partition*, *ip*, *port*}.



user id(primary)	db_name	Table_name	partition
00001	db01	Employee	1
00001	db01	Sales	2,3
00002		Jobs	3
......		......	......

partition	ip	port
1	210.117.181.23	1111
2	210.117.181.24	1111
3	210.117.181.25	1111

Fig. 2. A usage example of the meta tables.

The *ip* and *port* are network information to connect a *partition* (e.g., CUBRID server). Fig. 2 shows a usage example of the meta tables. Assume that the system receives a query like “Select * from Sales”. Based on the *userID* (*user01*), *dbName* (*db01*), and *tableName* (*Sales*), the metadata retrieval component retrieves *SearchTable* to find CUBRID servers required to process the query. From the *SearchTable*, we can find that *Sales* table is distributed on the *partition* 2 and *partition* 3. So, the middleware retrieves the *ip* and *port* of these partitions in the *IpPortTable* to send them the query.

Finally, a *query result merge component* merges results sent from each CUBRID server. In this case, a mechanism for receiving each query result without any collisions is required. For this, the system prepares a buffer for each CBURID server. By doing so, the system can receive the query results without collisions in parallel way. As a result, the efficiency of the data transmission is greatly improved. Once the component obtains an actual query result by performing an aggregation, it transmits the actual query result to the query issuer.

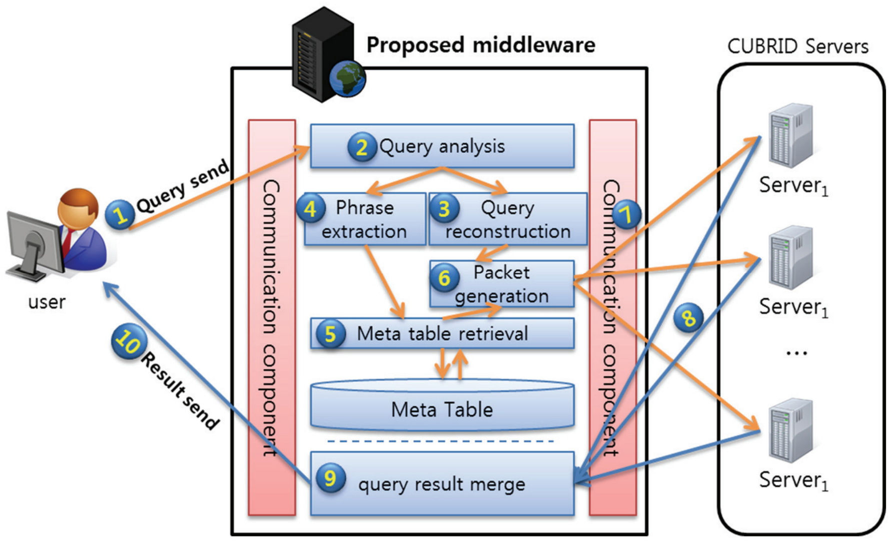


Fig. 3. The query processing procedure.

3.2 Overall Query Processing Procedure

Fig. 3 shows the overall query processing procedure of our CUBRID based distributed parallel query processing system. First, a user sends an SQL query to our system. Second, our system determines the type of a user’s query through the *query analysis component*. The types of a query as follows.

1) *Insert* phrase: our system distributes data into multiple servers.

2) *Select* phrase: our system retrieves data being distributed into severs in parallel manner.

3) *From* phrase: our system finds the required tables to process the user’s query.

4) *Join* phrase: our system performs an Equi-Join on the distributed CUBRID servers.

5) *Where* phrase: our system extracts the records that satisfy the conditions being described by the user.

6) *Order by* phrase: our system sorts query results sent from each CUBRID based on the sorting conditions.

7) *Limit* phrase: our system extracts result records as much as the user specifies. Third, our system

rewrites an SQL query so that the query can be processed on the multiple CUBRID servers. Fourth, our system extracts table names from a SQL statement by using the *query analysis component*. Fifth, by searching the *metadata retrieval component*, our system confirms a list of CUBRID servers that contain the required parts of the data for the given query. In addition, it determines the connection information (e.g., *ip* and *port*) of each CUBRID. Sixth, our system constructs packets to be transmitted to each CUBRID server based on the connection information and the reconstructed query. Seventh, our system transmits the constructed packets to the selected CUBRID servers by using the *communication component*. In addition, our system prepares a buffer for each CBURID server to receive a query result in parallel way. Eighth, each CUBRID server that receives the query from our system performs query processing on the data it has. After query processing, each CUBRID server returns a query result to the *communication component* of our system. Ninth, through the *query result merge component*, our system obtains the final query result by performing an aggregation of the results received from multiple servers. Finally, our system sends the final query result to the querying user through the *communication component*.

3.3 Query Processing Procedure according to the Query Type

In this section, we describe how our system processes each query type in detail. First, we explain the role of our system for *insert* phrase that is related to the distributed data insertion. Next, we describe the *select* and *join* phrase that are associated with data retrieval. Then, we deal with mechanisms for various aggregation functions. Finally, we show how our system processes the *order by* and *limit* phrases.

3.3.1 Insert

When our system analyzes that a query includes an *insert* phrase, the system stores data into the distributed CUBRID servers. To perform data insertion, the information of the relevant tables should be maintained in *MinMaxTable*. If the information of the relevant table does not exist in the *MiMaxTable*, our system determines a data partitioning strategy of the table. The data partitioning strategy is manually determined by an administrator by considering types of columns in the considered database and the number of CUBRID servers. Then, the administrator inserts {*dbName*, *partition*, *tableName*, *column*, *min*, *max*} record into the *MinMaxTable*. By referring the table, our system automatically stores the data into the appropriate partitions. For example, assuming that an appropriate partitioning column of the *Student* table in *set1* database is the *unique_number* column, our system can construct *MinMaxTable* as shown in Table 1. The *MinMaxTable* indicates that the records whose *unique_number* values are ranged between 0 and 50 are stored in *partition* 1. The records whose *unique_number* values are between 50 and 100 will be stored in the *partition* 2.

If the information of the relevant table is maintained in the *MiMaxTable*, our system executes data insertion on the appropriate CUBRID servers. For example, assuming that an SQL query is given as "Insert into *employee*(*unique_number*, *name*) values(*10*, '*LEE*')", our system can know that the data should be stored in the *employee* table by analyzing the given SQL query. By checking the *MinMaxTable*, our system knows that the *employee* table is partitioned based on the *unique_number* column and records with the *unique_number* value of 10 corresponds to the *partition* 1. Then, our system searches the *IpPortTable* to confirm the connection information of the *partition* 1. An example of the *IpPortTable* is shown in Table 2. By searching the *IpPortTable,* our system can extracts the connection information (i.e., *ip* = "111.112.113.111" and *port* = "8880") of the *partition* 1. Therefore, our system executes the data insertion by transmitting the SQL query to *partition* 1. As a result, our system can perform the distributed data insertion.

Table 1. An example of *MinMaxTable*

dbName	partition	TableName	column	min	max
set1	1	employee	unique_number	0	50
set1	2	employee	unique_number	50	100
⋮	⋮	⋮	⋮	⋮	⋮
set1	1	employee	unique_number	0	50
set1	2	employee	unique_number	50	100

Table 2. An example of *IpPortTable*

partition	IP	port
1	111.112.113.111	8880
2	111.112.113.112	8881
⋮	⋮	⋮
10	111.112.113.120	8890

3.3.2 Select

If our system analyzes that a query includes a *select* phrase, our system retrieves relevant CUBRID servers in a distributed way. First, our system determines a list of tables that should be retrieved for the given query by analyzing the SQL statement. Then, the system retrieves *SearchTable* to confirm the information of the relevant tables that are required to process the query. For example, assuming that *user01* transmits a query like "Select * from *employee* where *age*=32", our system can decide that the *Student* table should be retrieved for the given query. Assuming that the constructed *SearchTable* is given as Table 3, our system can know that the *Student* table of the *user01* is horizontally distributed into both *partition* 1 and *partition* 2. Then, our system searches the *IpPortTable* to obtain the connection information of the relevant CUBRID servers. By transmitting the query of *user01* to *partition* 1 and *partition* 2 servers, our system can perform data retrieval in a parallel way.

Table 3. An example of *SearchTable*

id	dbName	TableName	partition
user01	set1	employee	1, 2
⋮	⋮	⋮	⋮
user09	set1	employee	1

To process the *select* operation, our system should consider the following property. A query result generated from each CUBRID server is basically sorted based on a key value even though an order by phrase does not exist in the query. Therefore, our system should re-sort each query result transmitted from each CUBRID server based on the key value to generate the actual query result. To do this, our system performs the following steps. First, our system confirms the key column of the table that the query is related to. Second, our system checks a type of the key column. These information can be defined by using CUBRID API. Third, our system obtains one result record from each buffer that temporarily stores a query result transmitted from each CUBRID server. Fourth, our system compares the extracted records based on the type of the key column and sorts them in ascending order of the key. Then, our system appends the first record of the sorted result to the final result. Fifth, our system extracts another record from the buffer where the record written to the final result is extracted. At this time, our system ignores a duplicated record. These steps are repeated until all query results stored in the buffers are processed. Finally, our system terminates the *select* query processing by transmitting the final result to the querying user. Note that this procedure provides good performance when databases are partitioned based on the key column, which is a general approach in DBMSs.

3.3.3 Join

Our system can process a *join* query with the following conditions. First, *MinMaxTable* should contain the data partitioning strategies of the tables that are described in the query. Second, the tables should be partitioned based on the identical partitioning column and should follow the same data partitioning strategy. For example, assuming that our system gets an SQL query like "Select * from *employee, superior* where *age*=32", our system can execute *join* operation on *employee* and *superior* tables by using the *MinMaxTable* shown in Table 1. By searching the Table 1, we can confirm that both tables use the *ID* column for data partitioning and the *partition* 1 is responsible for maintaining records whose values of the *unique_number* column are between 0 and 50 for both *employee* and *superior* tables. Because the criteria for *join* operation are satisfied, our system can execute the *join* operation on the tables.

Meanwhile, a procedure to process the query which includes *join* phrase is as follows. First, our system analyzes the query to find a list of CUBRID servers which store the designated tables in the given query. Second, our system sends the query to the selected CUBRID servers and receives a query result from each CUBRID server. Third, to generate the actual query result, our system merges the query results sent from CUBRID servers that are participated in the query processing. Finally, our system terminates the query processing by transmitting the final query result to the querying user.

3.3.4 Aggregation

Our system supports *aggregation* queries (e.g., *min*, *max*, *count*, *sum*, *average*). First, our system confirms a type of *aggregation* operations that are requested to process a given query by using the *query analysis component*. Based on the type of the *aggregation* operation, our system computes final result as follows. (1) If the type of the *aggregation* operation is *min*, our system sends the query and receives a query result (i.e., minimum value) from each CUBRID server. Among them, our system obtains the smallest value as the final result. (2) If the type of the *aggregation* operation is *max*, our system sends the query and receives a query result (i.e., maximum value) from each CUBRID server. Among them, our system obtains the largest value as the final result. (3) If the type of the *aggregation* operation is *count*, our system transmits the query and receives a query result (i.e., the number of records) from each CUBRID server. Our system calculates the sum of these values and sets the summed value as the final result. (4) If the type of the *aggregation* operation is *sum*, our system sends the query and receives a query result (i.e., *sum*) from each CUBRID server. Our system calculates the sum of these values and sets the calculated result as the final result. (5) If the type of the *aggregation* operation is *average*, it is impossible to obtain the actual average result by using average results transmitted form CUBRID servers. Thus, our system should reconstruct the given query by using *sum* and *count* operations, instead of directly using the *average* operation. After our system transmits the query and receives query results (i.e., *count* and *sum*) from each CUBRID server, it can calculate the actual *average* value (*total sum / total count*).

3.3.5 Order by

To process the *order by* phrase, by analyzing the query, our system checks the number of order by conditions and designated columns with their data types (e.g., numeric data, character strings) in the given query. For example, assume that an SQL query is given as "Select * from *employee* where *age*=21 Order by *unique_number acs*, *name desc*". By analyzing the SQL query, our system finds that the number of *order by* conditions is 2 (i.e.., *unique_number acs* and *name desc*). For this query, records should be sorted by *unique_number* in ascending order first. If there are records with same *unique_number* values, they are sorted by *name* in descending order. In addition, the middleware confirms the type of both *unique_number* and *age* columns by using CUBRID API. After CUBRID servers process the query, they send the query result that is sorted based on the *order by* conditions to the buffers of the middleware. To make the final query result, our system should re-sort the query results transmitted from CUBRID servers that are participated in the query processing. The mechanism for processing the *order by* phrase is very similar with that of the *select* phrase. The difference is that the system sorts the query results based on the *order by* conditions that are extracted from the query.

3.3.6 Limit

To process the *limit* phrase, by analyzing the query, our system determines how many records should be transmitted to the querying user. For example, assume that our system receives a query like "Select * from *employee* where *age*=21 Limit 10". By analyzing the query, the middleware finds that the number of result to be sent to the client is 10. The mechanism for processing the *limit* phrase is very similar with that of the select phrase. The difference is that the middleware does not read all the query results transmitted from the CUBRID servers that are participated in the query processing. Our system finishes processing the query when the system writes the designated number of records to the final result.

4. Performance Evaluation

In this section, we show the extensive experimental results of our CUBRID based distributed parallel query processing system. Table 4 summarizes the comparison of our system with the existing systems, with respect to the essential requirement for bigdata processing, i.e., ACID property, SQL query support, and distributed and parallel processing.

Table 4. Comparison of the existing schemes with our system

Scheme	ACID	SQL query support	Distributed and parallel processing
Hadoop [7]	X	O (with Hive)	O
MongoDB [8]	X	X	O
Cassandra [9]	X	X	O
CUBRID Shard [10]	O	O	△
Saravanan et al. [15]	X	X	O
Li et al. [16]	X	X	O
Lee et al. [17]	X	X	X
Our middleware	O	O	O

Most of the existing works do not satisfy the ACID feature, except CUBRID-Shard and our system. In addition, the existing works fail to support a SQL-like query, except CUBRID-Shard and our system. Although Hadoop can support a SQL-like query, it requires an additional tool, such as Hive [19]. On the other hand, most of the existing works can support distributed and parallel processing, except Lee et al.'s work. Because only the CUBRID-Shard can satisfy the three requirements of bigdata processing, we compared our system with the existing CUBRID Shard, in terms of the query processing time for SQL operations, such as *select*, *order by*, *limit*, *projection*, *join*, and *average* [20].

However, the CUBRID Shard does not fully support parallel query processing in distributed environments. If a horizontally divided database of a user is distributed on a number of CUBRID servers, the CUBRID Shard should send a query of the user to each CUBRID server sequentially. However, because the CUBRID Shard cannot merge the query results processed by multiple servers, we implemented a simple merge component for the CUBRID Shard that aggregates the query results sent by CUBRID servers. But we do not implement the *join* and *aggregation* operations because they cannot be originally supported by the CUBRID Shard. Table 5 show experimental environments for our performance analysis.

Table 5. Experimental environments

CPU	Intel Core i5 Quad-Core 2.90 GHz
Memory	4 GB
O/S	Ubuntu 12.4
Compiler	g++ 4.6.3
CUBRID version	2.2.0

Fig. 4 shows the query processing time for a *select* operation by varying the number of data. The query processing time increases as the number of data becomes larger. When the proportion of result data is 40% of all data, which means that a user receives 40% of the all data as a result, the query processing time of our system requires 3.87 seconds. On the other hand, the existing CUBRID Shard requires 10.49 seconds. On average, our system shows 2.9 times better performance than the CUBRID Shard. Meanwhile, Fig. 5 depicts the query processing time for an *order by* operation by varying the number of data. We set the *order by* for all attributes. The query processing time increases as the number of data becomes larger. When the proportion of result data is 40% of all data, our system requires 5.71 seconds for the query processing while the CUBRID Shard requires 12.69 seconds. On average, our system shows 2.8 times better performance than the CUBRID Shard. The more attributes the *order by* operation should consider, the more time is required than the select operation.

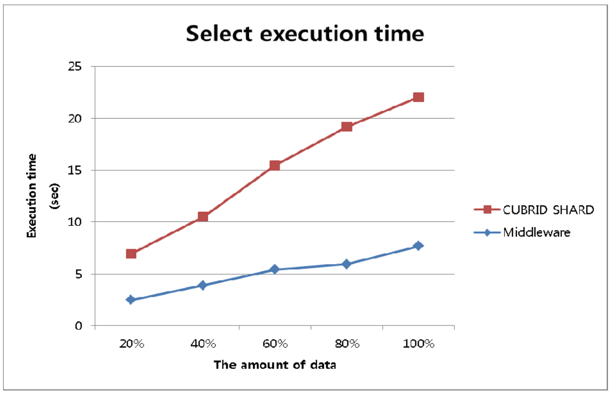


Fig. 4. The query processing time for select operation.

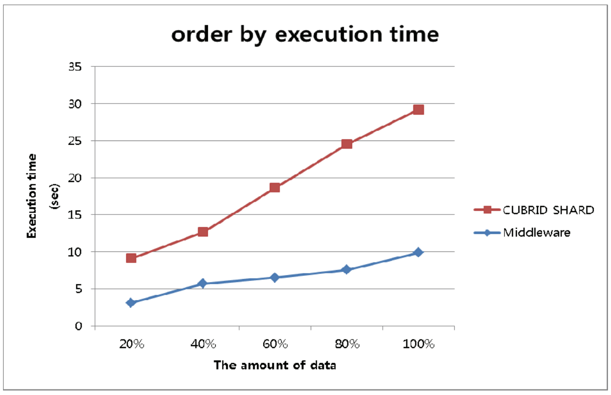


Fig. 5. The query processing time for order by operation.

For both operations, our system outperforms the CUBRID Shard. The reason is that our system processes a query in parallel manner on a distributed environment while the CUBRID Shard cannot support parallel query processing. In addition, the result of each CUBRID server is sent to the buffer assigned by the *query result merge component*. So, our system aggregates results in memory as soon as a set of results are transmitted from each CUBRID server. On the contrary, the CUBRID Shard can generate a final result after all the results of CUBRID servers are completely written in the file.

Fig. 6 shows the query processing time for a *limit* operation by varying the number of data. The query processing time increases as the number of data becomes larger. When the proportion of result data is 20% of all data, our system requires 4.68 seconds for the query processing while the CUBRID Shard requires 9.13 seconds. On average, our system shows 2.2 times better performance than the CUBRID Shard. Fig. 7 depicts query processing time for a *projection* operation by varying the number of data. For the *projection* query, we extract one attribute to compare with the performance of the *select* operation. When the proportion of data is 40% of all data, our system requires 1.36 seconds for the query processing. On the other hand, the existing CUBRID Shard requires 3.77 seconds. On average, our system shows 2.7 times better performance than the CUBRID Shard. Because the *projection* query requires the less number of data to be transmitted, the less time is needed than the *select* operation. Our system outperforms the CUBRID Shard because our system supports parallel query processing for a distributed environment.

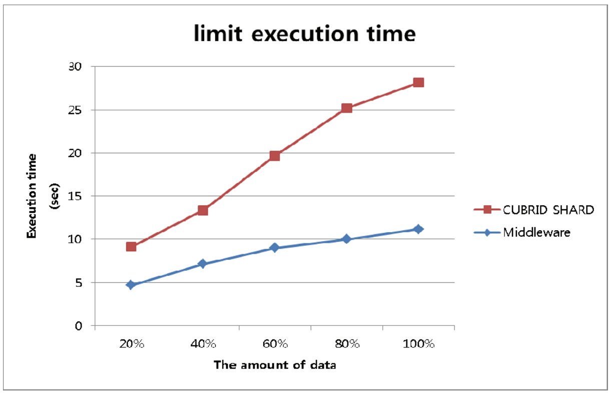


Fig. 6. The query processing time for limit operation.

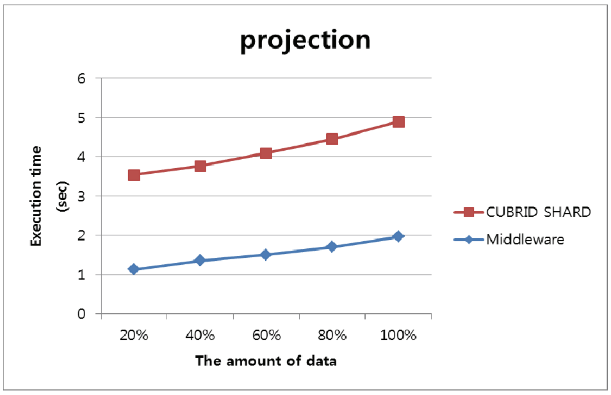


Fig. 7. The query processing time for projection operation.

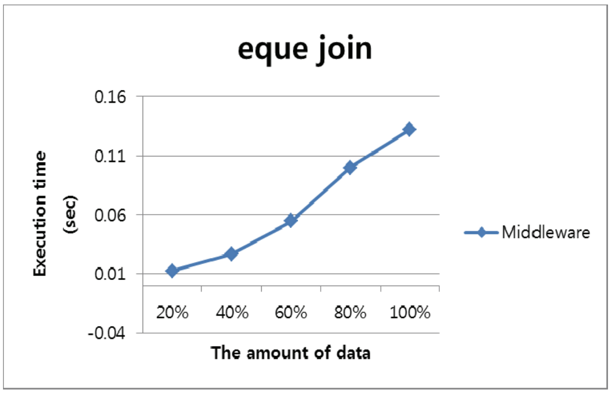


Fig. 8. The query processing time for join operation.

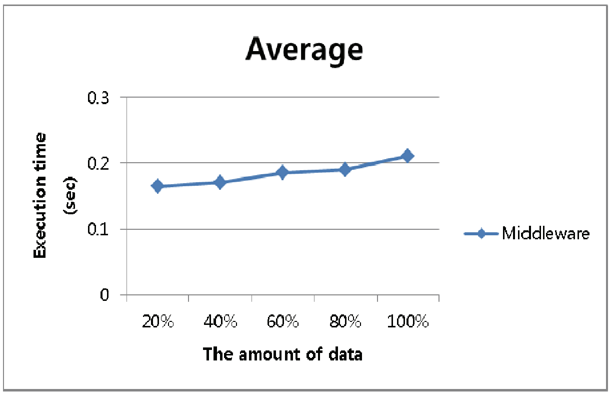


Fig. 9. The query processing time for average operation.

The CUBRID Shard cannot support both a *join* query and an *aggregation* one. Therefore, we only provide the performance of our system for the *equi-join* query and the *average* query. In case of the *join* operation, we use 10,000 data. Fig. 8 depicts the query processing time for the *join* operation by varying the number of data. When the proportion of data is 20% of all data, our system requires 0.0125 seconds for the query processing. On the other hand, Fig. 9 shows the query processing time for the *average* operation by varying the number of data. Our system can support the *aggregation* query, especially the *average* query, with the help of the *query analysis component*. The query processing time increases as the number of data becomes larger. When the proportion of data is 40% of all data, our system requires 0.17 seconds for the query processing. The query processing time is much less than other operations because most of the computation is performed on each CUBRID server in a parallel way and the only aggregated result needs to be transmitted to the client.

5. Conclusions

Due to the rapid growth of the amount of data, research on bigdata processing has been highlighted. However, the existing works have some problems that they cannot guarantee the ACID properties of database transactions and fail to support a sql-like query. Therefore, much attention has been paid to RDBMSs for bigdata processing. For bigdata processing, CUBRID Shard can support parallel query processing by dividing the database into multiple CUBRID servers. However, CUBRID Shard can answer a user's query only when the query is required to gain accesses to a single CUBRID server, instead of multiple ones.

Therefore, in this paper we proposed a CUBRID based distributed parallel query processing system that can answer a user's query in parallel and distributed manner. Our system can allow users to easily deal with the bigdata through SQL queries. Finally, we showed from our performance evaluation that our proposed system provides 2–3 times better performance on query processing time than the existing CUBRID Shard.

As a future work, we have a plan to apply our system to real database applications to show the efficiency of our system. In addition, we plan to support holistic aggregation operators with reasonable efficiency by expanding our proposed system.

Acknowledgement

This work was partly supported by the Human Resource Training Program for Regional Innovation and Creativity through the Ministry of Education and National Research Foundation of Korea (No. NRF-2016H1C1A1065816). This work was also supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No. R0113-16-0005, Development of an Unified Data Engineering Technology for Large-scale Transaction Processing and Real-time Complex Analytics).

References

[1] D. H. Lee, "Personalizing information using users' online social networks: a case study of CiteULike," *Journal of Information Processing Systems*, vol. 11, no. 1, pp. 1-21, 2015.

[2] J. Lv, J. Guo, and H. Ren, "Efficient greedy algorithms for influence maximization in social networks," *Journal of Information Processing Systems*, vol. 10, no. 3, pp. 471-482, 2014.

[3] D. Jiang, G. Chen, B. C. Ooi, K. L. Tan, and S. Wu, "epiC: an extensible and scalable system for processing big data," *Proceedings of the VLDB Endowment*, vol. 7, no. 7, pp. 541-552, 2014.

[4] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107-113, 2008.

[5] H. C. Yang, A. Dasdan, R. L. Hsiao, and D. S. Parker, "Map-reduce-merge: simplified relational data processing on large clusters," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Beijing, China, 2007, pp. 1029-1040.

[6] T. Rabl, S. Gomez-Villamor, M. Sadoghi, V. Muntés-Mulero, H. A. Jacobsen, and S. Mankovskii, "Solving big data challenges for enterprise application performance management," *Proceedings of the VLDB Endowment*, vol. 5, no. 12, pp. 1724-1735, 2012.

[7] Apache Software Foundation, "Apache Hadoop," 2014 [Online]. Available: http://hadoop.apache.org/.

[8] K. Chodorow, *MongoDB: The Definitive Guide*, 2nd ed. Sebastopol, CA: O'Reilly Media Inc., 2013.

[9] A. Dietrich, S. Mohammad, S. Zug, and J. Kaiser, "ROS meets Cassandra: data management in smart environments with NoSQL," in *Proceedings of the 11th International Baltic Conference on DB and IS*, Tallinn, Estonia, 2014.

[10] CUBRID Shard [Online]. Available: http://www.cubrid.com/manual/91/shard.html.

[11] M. Stonebraker, "SQL databases v. NoSQL databases," *Communications of the ACM*, vol. 53, no. 4, pp. 10-11, 2010.

[12] R. Cattell, "Scalable SQL and NoSQL data stores," *ACM SIGMOD*, vol. 39, no. 4, pp. 12-27, 2011.

[13] J. Han, E. Haihong, and G. Le, "Survey on NoSQL database," in *Proceedings of 2011 6th international conference on Pervasive computing and applications (ICPCA)*, Port Elizabeth, South Africa, 2011, pp. 363-366.

[14] CUBRID [Online]. Available: http://www.cubrid.com/.

[15] V. Saravanan, K. D. Pralhaddas, D. P. Kothari, and I. Woungang, "An optimizing pipeline stall reduction algorithm for power and performance on multi-core CPUs," *Human-centric Computing and Information Sciences*, vol. 5, no. 1, article no. 2, 2015.

[16] Y. Li, D. Kim, and B. S. Shin, "Geohashed spatial index method for a location-aware WBAN data monitoring system based on NoSQL," *Journal of Information Processing Systems*, vol. 12, no. 2, pp. 263-274, 2016.

[17] M. Lee, Y. S. Park, M. H. Kim, and J. W. Lee, "A convergence data model for medical information related to acute myocardial infarction," *Human-centric Computing and Information Sciences*, vol. 6, no. 1, article no. 15, 2016.

[18] H. I. Kim, M. Yoon, M. Choi, and J. W. Chang, "A new middleware for distributed data processing in CUBRID DBMS," *Procedia Computer Science*, vol. 52, pp.654-658, 2015.

[19] 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," *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1626-1629, 2009.

[20] D. J. DeWitt, "The Wisconsin benchmark: past, present, and future," University of Wisconsin, 1993.

Hyeong-Il Kim

He received the B.S., M.S., and Ph.D. degrees in computer engineering from Chonbuk National University, Korea, in 2009, 2011, and 2016, respectively. He is currently a senior researcher in Agency for Defense Development. His research interests include database encryption, privacy-preserving query processing, and cloud computing.

HyeonSik Yang

He is a B.S. in the Chonbuk National University. His research interests include hardware transaction memory (HTM) and database in parallel environment.

Min Yoon

He received the B.S., M.S., and Ph.D. degrees in computer engineering from Chonbuk National University, Korea, in 2008, 2010, and 2017, respectively. He is currently a senior researcher in Agency for Defense Development. His research interests include privacy preservation in sensor network and database in parallel environment.

Jae-Woo Chang

He is a professor in the Department of Information and Technology, Chonbuk National University, Korea from 1991. He received the B.S. degrees in Computer Engineering from Seoul National University in 1984. He received the M.S. and Ph.D. degrees in Computer Engineering from Korea Advanced Institute of Science and Technology (KAIST) in 1986 and 1991, respectively. His research interests include spatial database, privacy-preserving query processing, and context awareness and storage system.