A Middleware Supporting Query Processing on Distributed CUBRID

Hyeong-Il Kim, Min Yoon, YoungSung Shin, and Jae-Woo Chang*

Chonbuk National University, Korea
{melipion,myoon,twotoma,jwchang}@jbnu.ac.kr

Abstract. Due to the shortages of NoSQL, studies on RDBMS based bigdata processing have been actively performed. Although they can store data in the distributed servers by dividing the database, they cannot process a query when data of a user is distributed on the multiple servers. Therefore, in this paper we propose a CUBRID based middleware supporting distributed parallel query processing. Through the performance evaluations, we show that our proposed scheme outperforms the existing work in terms of query processing time.

Keywords: Middleware, distributed parallel query processing, CUBRID.

1 Introduction

Recently, studies on the bigdata processing have been actively performed [1], [2]. With the existing IT technologies, it is very hard to efficiently store, process and analyze the bigdata. The bigdata itself is hard to be used as valuable information because of the immense volume of the bigdata. Therefore, it is necessary to analyze the bigdata to extract the meaningful information. To analyze the bigdata, a large scale of computing resources and efficient bigdata management system are required. For this, studies on NoSQL have been done [3-7]. However, NoSQL cannot satisfy the ACID properties of database transactions. Therefore, bigdata processing based on RDBMS (Relational DataBase Management System) has been spotlighted.

CUBRID Shard [8] is a RDBMS that is designed to process bigdata. To support parallel query processing, CUBRID Shard stores data in the distributed CUBRID servers by dividing the database. However, if data of a user is distributed on the multiple CUBRID servers, CUBRID Shard cannot process the query. Moreover, CUBRID Shard has a low usability because a user should specify a '*shard_hint*' in the SQL when requesting the query.

To solve these problems, in this paper we propose a CUBRID based middleware which supports distributed parallel query processing. Through our proposed middleware, users who are familiar with SQL can conveniently process the bigdata by using SQL statements. In addition, the middleware can support the aggregation queries that have not been handled on the distributed parallel computing environment.

* Corresponding author.

© Springer-Verlag Berlin Heidelberg 2015

James J. (Jong Hyuk) Park et al. (eds.), *Advanced Multimedia and Ubiquitous Engineering*,
Lecture Notes in Electrical Engineering 352, DOI: 10.1007/978-3-662-47487-7_5

The rest of this paper is organized as follows. In section 2, we briefly review related work. Section 3 explains the propose middleware in detail. An empirical evaluation is presented in Section 4. Finally, we conclude this paper in section 5.

2 Related Work

NoSQL systems are increasingly used in bigdata and real-time web applications. NoSQL such as Hadoop [3], MongoDB [4], and Cassandra [5] provides a mechanism for storage and retrieval of unstructured data. The data structures used by NoSQL differ from those used in relational databases, making some operations faster in NoSQL. However, most NoSQL cannot satisfy the ACID properties of the database transactions. Especially, the major shortcoming of NoSQL is that it cannot guarantee data consistency when NoSQL supports the partition tolerance and availability.

Therefore, RDBMS have been spotlighted in the field of bigdata processing. CUBRID [9] is an object-oriented RDBMS developed by NHN (Next Human Networks). CUBRID provides predictable automatic fail-over and fail-back features based on a native CUBRID heartbeat technology. However, CUBRID cannot run on the distributed system environments because CUBRID is optimized on single machine. So, it is not efficient for dealing with bigdata. To solve the problems of CUBRID, CUBRID Shard [8] is developed. CUBRID Shard can partition the data based on the horizontal partitioning technique. CUBRID Shard allows storing a number of database shards and distributing data. With CUBRID Shard, application developers do not need to modify the application logic to divide a database into CUBRID Shards because the database system automatically handles it. CUBRID Shard provides built-in distributed load balancing, connection, and statement pooling. However, CUBRID Shard cannot process a query when data of a user is distributed on the multiple CUBRID servers. It can be a big problem when dealing with the bigdata. Moreover, CUBRID Shard has a low usability because a user should specify a '*shard_hint*' in the SQL when requesting the query.

3 Middleware Based on the Distributed CUBRID

Fig. 1 shows the overall system architecture of our proposed middleware supporting parallel query processing on the distributed CUBRID. The middleware consists of 4 components.

First, a communication component is in charge of data transmission with a user or CUBRID servers. SQL query and database connection information are transmitted through the communication component. Second, a query analysis component performs an SQL parsing to extract table names in from phrase that are used for retrieving meta tables. In addition, the component distinguishes the query types. Third, metadata retrieval component retrieves meta tables. There are 3 meta tables. i) *MinMaxTable* stores information for inserting data on the distributed CUBRID servers. The schema of the table is {*dbName*, *partition*, *tableName*, *column*, *min*, *max*}. The *column* means the name of the column that is used to partition the

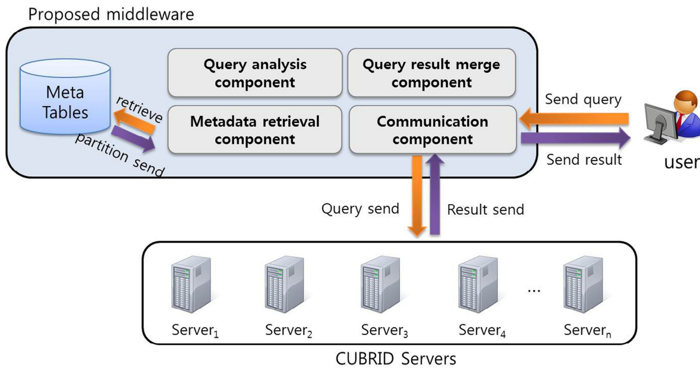

Fig. 1. The overall system architecture

tableName table. The *partition* means a CUBRID server which stores records whose values of the *column* are between *min* and *max*. ii) *SearchTable* stores information required for retrieving data that are stored on the distributed CUBRID servers. The schema of the table is {*userID*, *dbName*, *tableName*, *partition*}. By using the table, we can determine the partitions storing the *tableName* table that are necessary to process the query of the *userID*. iii) *IpPortTable* stores connection information of each CURED server. The schema of the table is {*partition*, *ip*, *port*}. Finally, a query result merge component merges results sent from CUBRID servers. The middleware prepares a buffer for each CBURID server to receive each query result in parallel without any collisions. In addition, the query result merge component eliminates duplicated results and aggregates query results if needed. Finally, the query result merge component sends the final query result to the query issuer.

The overall query processing procedure with the proposed middleware is as follows. i) A user sends an SQL query to the middleware. ii) By using the query analysis component, the middleware distinguishes a type of the query. iii) The middleware reconstructs the SQL query to be processed on the distributed CUBRID servers. iv) By using the query analysis component, the middleware extracts table names in from phrase. v) By using metadata retrieval component, the middleware finds a list of CUBRID servers holding the required data to process the query. vi) The middleware generates a packet for each CUBRID server. vii) By using the communication component, the middleware sends packets to the CUBRID servers. In addition, the middleware prepares a buffer for each CBURID server to receive query results in parallel. viii) The middleware receives a query result from each CUBRID server that processes the query. ix) By using the query result merge component, the middleware draws the final query result. x) The middleware finishes the query request by sending the final query result to the client.

Meanwhile, the middleware plays a different role according to the query type. Following describes how our proposed middleware processes each query type. First, in case of Insert phrase, the middleware stores data into the distributed CUBRID servers. To handle data insertion, data partitioning strategy of the designated table should be stored in *MinMaxTable*. By referring the table, the system can

automatically store the data into the appropriate partition. For example, for a given SQL query "Insert into *Student*(*ID*, *name*) values(*20*, '*KIM'*)", the middleware can notice that the data should be inserted into the *Student* table. By referencing the *MinMaxTable*, the middleware confirms that the *Student* table is partitioned based on the *ID* column and the record with the *ID* value of 20 is related to the *partition* 1. Then, the middleware retrieves the *IpPortTable* to find the connection information of the *partition* 1. Table 2 shows an example of the *IpPortTable*. By retrieving the *IpPortTable,* the middleware finds that the *ip* and *port* of the *partition* 1 are "123.456.789.001" and "9001" respectively. So, the middleware performs the data insertion by sending the SQL query to the CUBRID server (*partition* 1). Through the mechanism, the middleware achieves the distributed data insertion.

Table 1. MinMaxTable

dbName	partition	TableName	column	min	Max
db01	1	Student	ID	0	50
db01	2	Student	ID	50	100
db01	1	Graduate	ID	0	50
db01	2	Graduate	ID	50	100

Table 2. IpPortTable

partition	ip	port
1	123.456.789.001	9001
2	123.456.789.002	9002
10	123.456.789.010	9010

Table 3. *SearchTable*

id	dbName	TableName	Partition
user01	db01	Student	1, 2
user02	db09	Professor	1

Second, in case of Select phrase, the middleware retrieves databases in distributed manner. For this, the middleware determines which tables should be retrieved by analyzing the SQL query and retrieves *SearchTable* to find partitioning information of the tables. For example, assume that *user01* sends a query like "Select * from *Student* where *age*=21". By analyzing the query, the middleware can notice that the *Student* table is required to process the query. When we consider the *SearchTable* shown in Table 3, the middleware can find that *Student* table of the *user01* is distributed in *partition* 1 and *partition* 2. Then, the middleware accesses the *IpPortTable* to retrieve the connection information of the CUBRID servers. By sending the query to these CUBRID servers, data retrieval can be performed in parallel. Meanwhile, when processing the select query type, the middleware should consider following. The query result of each CUBRID server is sorted based on the order by conditions. If there is no order by phrase in the query, the query result of each CUBRID server is sorted based on the key value by default. So, the middleware should re-sort the query result sent from each CUBRID server based on the order by conditions to make the final query result. In addition, the middleware eliminates a duplicated record during re-sorting the query results. If there is a limit phrase in the query, the middleware terminates the query processing when the middleware writes the required number of records to the final result. Finally, the middleware completes the select query processing by sending the final query result to the client.

Third, in case of Join phrase, the middleware can process the query when the following criteria are satisfied. i) *MinMaxTable* should store the partitioning strategies of the designated tables. ii) The tables should be partitioned based on the same column and should follow the same partitioning strategy. For example, assume that the middleware receives a query like “Select * from *Student, Graduate* where *age*=21” and the *MinMaxTable* is given as like Table 1. *Student* and *Graduate* tables use *ID* column for partitioning and their partitioning strategy is identical (e.g., *partition* 1 is in charge of storing records whose *ID* values are between 0 and 50 for both tables). In this case, the middleware can perform the join operation on the tables.

Finally, in case of Aggregation phrase, the middleware finds what kinds of aggregation operations are included in the query. According to the type, the middleware operates in different way. i) When the type is *min* or *max*, the middleware receives the minimum or maximum value from each CUBRID server and sets the smallest or largest value as the final result. ii) When the type is *count* or *sum*, the middleware receives the number of records of sum from each CUBRID server and calculates the final result by adding result values. iii) When the type is *average*, it is impossible to draw the final result by using average results sent form CUBRID servers. Therefore, the middleware reconstructs the query by using *sum* and *count* instead of *average*. Then, the middleware receives the query result (*count* and *sum*) from each CUBRID server. The middleware calculates the sum of these values respectively and calculates the average value (*total sum* / *total count*).

4 Performance Evaluation

We compare our middleware with the existing CUBRID in terms of query processing time varying the number of data. Because CUBRID does not support parallel query processing in distributed environments, we perform query processing of CUBRID in a sequential way. We use one master node and 3 slave nodes for the performance evaluation. We use CUBRID version 2.2.0 and compile the middleware using g++ 4.6.3 running on the Linux 3.5.0-23 with Intel® Core™ i3-3240 3.40Ghz CPU and 8 GB memory. According to Wisconsin Benchmark [10], we generate a million data for select and average operations, and 10,000 data for join operation.

Fig. 2 describes query processing time for select operation. The query processing time is increased as the number of data increases. When the number of data is 60% of all data, the query processing time of our scheme and CUBRID are 7.76 and 14.49 seconds, respectively. The middleware shows about 47% better performance than CUBRID. Fig. 3 shows the query processing time for join operation. When the number of data is 60% of all data, the query processing time of our scheme and CUBRID are 0.14 and 0.35 seconds, respectively. Our proposed middleware shows about 60% better performance than CUBRID. Fig. 4 shows query processing time for average operation. When the number of data is 60% of all data, the query processing time of our scheme and CUBRID are 0.12 and 0.31 seconds. Overall, our proposed middleware outperforms the existing CUBRID because our middleware supports parallel query processing in a distributed environment. Especially, in case of join operation, the middleware shows much performance improvement because join operation requires more computations than the select operation.

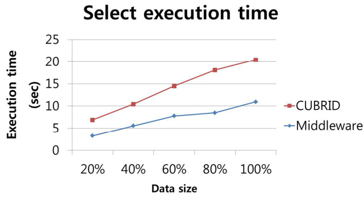

Fig. 2. Select operation performance

Fig. 3. Join operation performance

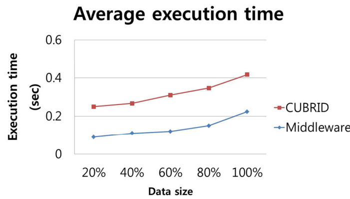

Fig. 4. Average operation performance

5 Conclusion

Existing distributed systems have problems when processing bigdata. Therefore, in this paper we propose a CUBRID based middleware which supports distributed parallel query processing. The middleware can support users who are familiar with SQL to conveniently process the bigdata by using SQL statements. In addition, the middleware can support various aggregation operators. Through the performance evaluations, we show that our proposed scheme outperforms the existing work in terms of query processing time. As a future work, we plan to expand our middleware to support various types of join with reasonable efficiency.

Acknowledgements. This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (2014065816).

References

1. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Communications of the ACM 51(1), 107–113 (2008)
2. Rabl, T., Sadoghi, M., Jacobsen, H.: Solving Big Data Challenges for Enterprise Application Performance Management. VLDB Endowment 5(12), 1724–1735 (2012)
3. Apache Software Foundation, Apache Hadoop, http://hadoop.apache.org
4. Chodorow, K.: MongoDB: the definitive guide. O'Reilly Media Inc. (2013)

5. Dietrich, A., Mohammad, S., Zug, S., Kaiser, J.: ROS meets Cassandra: Data Management in Smart Environments with NoSQL. In: 11th International Baltic Conference (2014)
6. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The Hadoop Distributed File System. In: 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST), NV (2010)
7. Han, J., Haihong, E., Guan, L.: Survey on NoSQL Database. In: 6th IEEE International Conference on Pervasive Computing and Applications, Port Elizabeth, pp. 363–366 (2011)
8. CUBRID Shard, http://www.cubrid.org/manual/91/en/shard.html
9. CUBRID, http://www.cubrid.com
10. DeWitt, D.J.: The Wisconsin Benchmark: Past, Present, and Future. In: Database and Transaction Processing System Performance Handbook (1993)