Advanced Science and Technology Letters
Vol.43 (Multimedia 2013), pp.110-113
http://dx.doi.org/10.14257/astl.2013.43.23

Efficient Range-based Query Processing on the Hadoop Distributed File System

Jong Wook Kim*, Sae-Hong Cho**, Ilmin Kim***

*Dept. of Media Software at Sangmyung University
20-Gil, Hongji-dong, Jongno-gu, Seoul, Korea
jkim@smu.ac.kr

**Dept. of Multimedia Engineering at Hansung University
389 Samsun-Dong 3-Ga, Sungbuk-Gu, Seoul, Korea
chosh@hansung.ac.kr

***Dept. of Computer Engineering at Hansung University
389 Samsun-Dong 3-Ga, Sungbuk-Gu, Seoul, Korea
ikim@hansung.ac.kr

Abstract. There was a common belief that the performance of query in a data warehousing system built on top of Hadoop Distributed File System (HDFS) does not meet the performance in traditional data warehousing systems. Therefore, in the past five years, there has been considerable effort to improve the query performance on data warehousing systems running on the HDFS. The main goal of this work is to develop the mechanisms to support an efficient processing of range-based queries over large size of data stored in the HDFS. In particular, we present the data storage scheme which logically partitions each HDFS block based on the user-specified columns.

Keywords: HDFS, Performance, Partition, Range-based Query

1 Introduction

Recently, there has been growing interest from database research community in developing large scale data processing systems by exploiting MapReduce-based parallelization [1,2,3,4]. The main goal of these approaches is to boost the efficiency of query processing by applying indexing techniques, which are well supported by traditional parallel data warehousing systems, to the Hadoop Distributed File System (HDFS). Like the previous works in [1,2,3,4], this work is motivated to improve the performance of query processing on large Hadoop clusters. In particular, we are interested in enhancing the performance of range-based queries. For example, consider a sales analyst who wants to look for the number of automobiles which were manufactured in Alabama and sold in the United States between January 2012 and October 2012. For her analysis, she may want to have an answer to the following query;

```
SELECT   count(*)
FROM     AutoSale t1, AutoManufacture t2
WHERE    t1.auto_id = t2.auto_id
And      t1.sale_date  Between 2012-01-01 And 2012-10-31
And      t1.sale_location = "United States"
And      t2.manu_location = "Alabama"
```

Traditional data warehousing systems boost the performance of this query by exploiting partitioned tables, such as Teradata Partitioned Primary Indexes (PPIs) [5] or Oracle Range Partition [6], which help avoid the access to the irrelevant portion of data. The same, however, cannot be said for data warehousing systems built on top of the HDFS where range-based queries usually require a full table scan. Thus, there is an impending need for the mechanisms to support efficient processing of range-based queries in the data warehousing systems which are running on top of the HDFS.

In this paper, we focus on improving the performance of range-based queries over large size of data stored in the HDFS. The rest of this paper is structured as following. In the next section, we describe the partitioned table used in traditional data warehousing systems. In Section 4, we present the data storage scheme in the HDFS to support an efficient range-based query processing. In Section 5, we conclude the paper.

2 Partitioned Table in Traditional Data Warehousing Systems

In this section, we briefly describe the partitioned table used in commercial data warehousing systems. Let us consider the example query in the previous section in which a user is interested in the answer to the total number of cars sold between 2012-01-01 and 2012-10-31. If this is a very common type of query for her analysis, significant performance gain can be achieved by creating the table *AutoSale* as following:

```
Create Table AutoSale (
     auto_id        INTEGER NOT NULL,
     sale_location  CHAR(256),
     sale_date      DATE FORMAT 'yyyy-mm-dd' NOT NULL
     dealer_id      INTEGER NOT NULL)
PrimaryKey (auto_id)
Partition By (
      Range_N(sale_date  BETWEEN  '2010-01-01'  and  '2012-12-31'
            EACH INTERVAL 1 YEAR)
      Range_N(dealer_id  BETWEEN  1  and  400  EACH 200));
```

In this scenario, tuples in the table *AutoSale* are first distributed across each node using a hash algorithm to the primary key *auto_id*. Note that commercial data warehousing systems are built on shared nothing architecture where each node exclusively uses process, main memory and disk. Then, in each node, tuples are first sorted by the attribute *sale_date*, and next by the attribute *dealer_id*. Finally, these sorted tuples are first partition by year based on *sale_date* and then, tuples having the same year are further partitioned by *dealer_id* in groups of 200.

Based on the partitioning scheme above, we are able to efficiently process the example query in Section 1 because it helps avoid the overhead of fully scanning the entire table. For example, unqualified portion of data (whose *sale_date* does not lie between 2012-01-01 and 2012-10-31) can be eliminated from further consideration.

3 Supporting an Efficient Range-based Query on the HDFS

The HDFS partitions the input data into fixed-size HDFS block (the default size is 64MB), physically makes copies of each HDFS block which are called as replica, and then stores each replica on different datanodes. In generally, given a query, the full scanning of data block in datanode is required, due to the lack of an effective indexing scheme in the HDFS. In order to overcome this shortcoming, the recent works, such as [1,2,3], have proposed the approaches that build indexes on each data block. However, none of these approaches is not suitable for supporting range-based queries, and thus, the full scanning of the HDFS is still required.

If partition conditions are specified by users when creating a table, then, each replica of an HDFS block is first sorted based on the user specified partition columns. For example, in the example in Section 2, we first sort each data block by the column *sale_date*, and then by the column *dealer_id*. The sorting of data block is performed by using main memory, since the block size of each data block is from 64MB to 1GB that is well fit into main memory.

Then, as can be seen in Figure 1, the proposed approach builds and appends the index file to each original data block. The index files consist of a list of triple, v, p_d, and p_i where

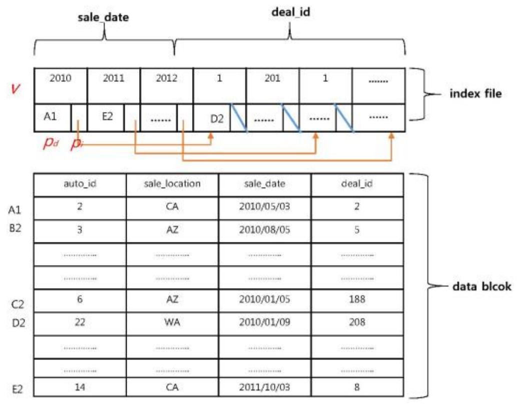
- v is a value which is determined by the user-specified interval,
- p_d is a pointer to the offset from which data being greater than or equal to the value v starts in original data block, and
- p_i is a pointer to the next level partition column. If the current one corresponds to the last level partition column, then p_i is set to 0.

Note that our partitioning scheme does not physically partition the original data block.

The proposed approach in this paper does not affect the behavior of non-range based query. However, if a user query is a range-based query in which any column in range conditions is part of columns used in creating the index file, then, we would benefit from using the logically partitioned data block. In this case, we, first, load the index file into main memory. Then, by using the index file loaded in main memory, we directly access to the data block which satisfies the range conditions in the query. This query processing scheme enables to avoid to read un-relevant data, and thus, results in improved query performance.

4 Conclusion

In this paper, we present the storage scheme to support the high performance of range-based query on data warehousing systems built on top of the HDFS. The proposed scheme in this paper builds the multi-level data partition index for each HDFS block. Then, by using the proposed partitioning scheme, we can achieve significant performance gain at query processing time by eliminating irrelevant data from further consideration.



	auto_id	sale_location	sale_date	deal_id
A1	2	CA	2010/05/03	2
B2	3	AZ	2010/08/05	5
	....	....	....	....
	....	....	....	....
C2	6	AZ	2010/01/05	188
D2	22	WA	2010/01/09	208
	....	....	....	....
	....	....	....	....
E2	14	CA	2011/10/03	8

Fig. 1. Data block is attached with index file which provide the random access to data file.

References

1. A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. Saltz: Hadoop GIS: a high performance spatial data warehousing system over mapreduce, Proceedings of the VLDB Endowment, Vol. 6, No 11, August (2013)
2. J. Dittrich, J.A. Quiané-Ruiz, S. Richter, S. Schuh, A. Jindal, and J. Schad: Only Aggressive Elephants are Fast Elephants, Proceedings of the VLDB Endowment, Vol. 5, No 11, August (2012)
3. J. Dittrich, J.A. Quiané-Ruiz, A.Jindal, Y. Kargin, V. Setty, and J. Schad: Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing), Proceedings of the VLDB Endowment, Vol. 3, No 1, August (2010)
4. M.Y. Eltabakh, F. Ozcan, Y. Sismanis, P.J. Haas, H. Pirahesh, and J. Vondrak: Eagle-eyed elephant: split-oriented indexing in Hadoop, Proceedings of the 16th International Conference on Extending Database Technology, pp 89-100 (2013)
5. P. Sinclair: Using PPIs to improve performance, Teradata Magazine, www.teradata.com/tdmo/v08n03/pdf/AR5731.pdf, September (2008).
6. Partitioning in Oracle Database 11g, http://www.oracle.com/technetwork/database/enterprise-edition/partitioning-11g-whitepaper-159443.pdf, June (2007)