Leveraging MapReduce with Column-Oriented Stores: Study of Solutions and Benefits

Narinder K. Seera and S. Taruna

Abstract The MapReduce framework is a powerful tool to process large volume of data. It is becoming ubiquitous and is generally used with column-oriented stores. It offers high scalability and fault tolerance in large-scale data processing, but still there are certain issues when it comes to access data from columnar stores. In this paper, first, we compare the features of column stores with row stores in terms of storing and accessing the data. The paper is focused on studying the main challenges that arise when column stores are used with MapReduce, such as data co-location, distribution, serialization, and data compression. Effective solutions to overcome these challenges are also discussed.

Keywords MapReduce · Data Co-location · Column stores · Data distribution · Serialization

1 Introduction

In the digital era, there is an emerging discrepancy between the volume of data being generated by a variety of applications and the ability to analyze this huge data. As the size of data is growing day by day, it is getting challenging both to store and process this large-scale data so as to analyze and derive meaning out of it. All recent database systems use B-tree indexes or hashing to speed up the process of data access. These data structures keep data sorted and allow very fast and efficient searching, sequential accessing of data and even insertion and deletion of data from the underlying storage. Modern DBMS also incorporate a query optimizer that optimizes query before execution and may use either an index file or may execute a sequential search for accessing data.

N.K. Seera (✉) · S. Taruna
Banasthali Vidyapeeth, Jaipur, India
e-mail: sk.narinder@gmail.com

S. Taruna
e-mail: staruna71@yahoo.com

V.B. Aggarwal et al. (eds.), *Big Data Analytics*, Advances in Intelligent Systems and Computing 654, https://doi.org/10.1007/978-981-10-6620-7_5

The problem of processing and analyzing huge data sets has been answered by MapReduce, a programming model where users write their programs while concentrating only on program details ignoring the internal architecture of MR. MapReduce has no indexing feature and hence it always performs brute force sequential search. However, most of the column-oriented data storage systems that use MapReduce use index mechanisms in their underlying data storage units. Apart from this, sometimes MapReduce also suffer from a serious performance drawback due to large number of disk seeks. This can be illustrated with the following example.

In MapReduce model, M output files are produced by *N* map instances. Each of the M output files is received by a different reduce instance. These files are stored on the local machine running the map instance. If *N* is 500 and *M* is 100, the map phase will generate 50,000 local files. When the reduce phase begins, each of the 100 reduce instances reads its 500 input files by using FTP protocol to "pull" each of its input files from map nodes. With 100 of reducers operating concurrently, it is expected that multiple reducers will try to access their input files from the same mapper (map nodes) in parallel—producing huge number of disk seeks and bringing down the effective data transfer rate of disk by a factor of 20 or more. Due to this reason, parallel database systems do not materialize their split files and use "push" rather than "pull".

Below, we discuss the motivation behind using column-oriented stores and employing MapReduce techniques with such systems. The rest of the paper is organized as follows: Sect. 3 introduces how the data is stored and read from column-oriented stores. In Sect. 4, we explore the main issues related to the use of column-oriented stores with MapReduce. Finally, in Sect. 5, we conclude the paper (Fig. 1).

2 Motivation

The main features provided by row oriented stores are:

1. The speed at which data is loaded in HDFS blocks is very fast and no additional processing overhead is incurred.

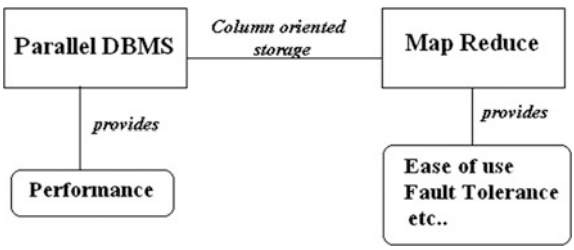

Fig. 1 Parallel DBMS versus MapReduce

2. All columns of the same tuple or row can be accessed from the same HDFS block.

Besides these features, row stores suffer from few serious drawbacks, which are listed below:

1. All columns of the same row are rarely accessed at the same time.
2. Additional processing overhead is added due to compression of different types of columnar data (as data types of different columns are generally different).

Figure 2 depicts how read operations are performed in row stores. The read operation is a two-step process. First, the rows from data nodes are read locally at the same time. Second, the undesired columns are discarded.

To overcome the limitations of row stores, column stores are generally gaining popularity and are believed to be best compatible with MapReduce. In the next section, we discuss how the data is managed in column stores and is used by MapReduce. We also discuss the challenges and solutions of adhering column stores with MapReduce.

3 MapReduce with Column Stores

Compared with row stores, I/O cost in column stores can be reduced to a great extent. The reason for this is that only desired columns are loaded and these columns can be easily compressed individually. Figure 3 illustrates the read operations in column stores. As an example, to access columns A and C, which are available at data node 1 and 3, respectively, first the columns from both the data

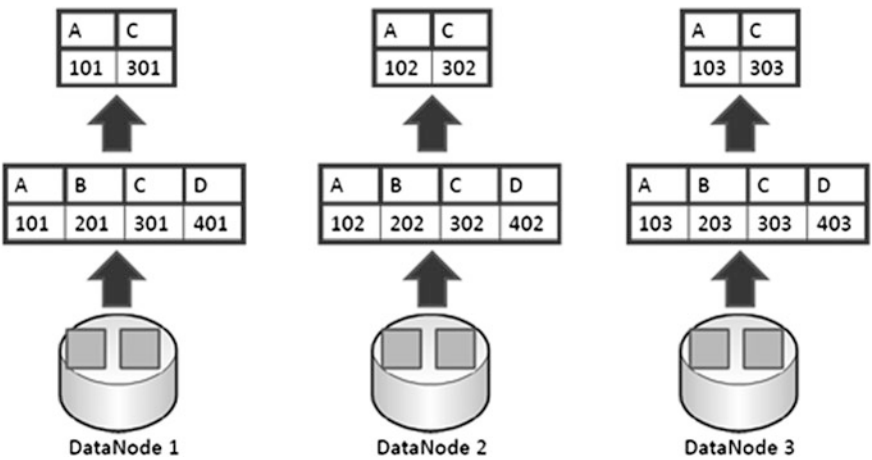

Fig. 2 Read operation in row stores

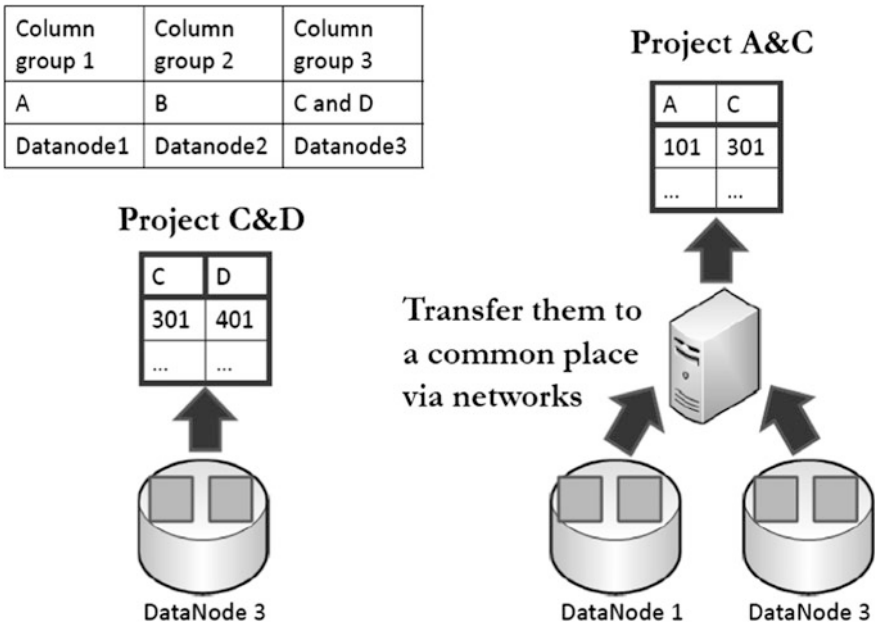

Fig. 3 Read operation in column stores

nodes are fetched at one common place, and then projection is performed over attributes A and C.

The only drawback of column stores is that—accessing columns from different data nodes entail additional data transfers in network.

The biggest motivation behind using column stores is to increase the performance of I/O in two ways [1]:

1. Minimize the data transfer in network by eliminating the need to access unwanted columns
2. Reducing the processing overhead to compress all the columns individually.

Distributed systems and programming model such as MapReduce also prefer column data stores due to the features they offer. HadoopDB [2] also adheres to columnar data store—Cstore [3] as its underlying data storage unit. Dremel [4]—an interactive ad hoc query system also use nested columnar data storage, for processing large data sets of data. It employs column-striped data storage for reading data from large storage space and reducing I/O costs due to inexpensive compression. Bigtable offered column family store for grouping multiple columns as a single basic unit of data access. Hadoop—an open source implementation of Java, also gained popularity because of its underlying column-wise storage, called HFile.

4 Challenges and Solutions

In the above section, we see how the data is stored and accessed in column-oriented stores. In this section, we throw a light on the main issues in concern with the problems of using MapReduce with column-oriented stores and few possible solutions.

a. Generating equal size splits—The problem is—in order to parallelize the job effectively over the nodes of a cluster, the data set must be partitioned into almost equal size splits. This can be done by partitioning the dataset horizontally and placing all partitions in separate sub-directories in Hadoop; where each sub-directory will serve as a separate split.
b. Data Co-location [5]—The default data placement policy of Hadoop randomly allocates the data among nodes for load balancing and simplicity. This data placement policy is fine for those applications which need to access data from a single node. But if any application wants to access data from different nodes concurrently, then this policy shows performance degradation, as:

   - It raises the cost of data shuffling
   - It increases network overhead due to data transfer
   - It decreases the efficiency of data partitioning.

   The problem here is that this data placement policy does not give any data co-location guarantee. So how can we improve the data co-location on the nodes so that the related values among different columns in the dataset are available on the same node executing the map task (or mapper).
   Babu [6] proposed an algorithm to resolve this issue, named dynamic co-location algorithm. This algorithm decreases the average number of nodes which are engaged in processing a query by placing the frequently accessed data sets on the same node, thereby reducing the data transfer cost. This algorithm dynamically verifies the relation between data sets and reshuffles the data sets accordingly. This algorithm has shown significant improvements in the execution time of MapReduce jobs.
   Figure 3 illustrates how two files A and B can be co-located using a locator table. File A has two blocks and file B has three blocks. All the blocks of both the files A and B are replicated on the same data nodes. A locator table is used to keep track of all the co-located files. It stores locator's information along with a list of files on the locator (Fig. 4).
c. Data Distribution [7]—In Hadoop, all the nodes store input and output files related to job currently executing on them. These nodes manage the distribution of file blocks over other nodes of the cluster, as required. When any node needs a file, available on any other node, only the desired file block is copied on it to avoid unnecessary traffic. The method of dividing the data for the map tasks can be defined by the user.

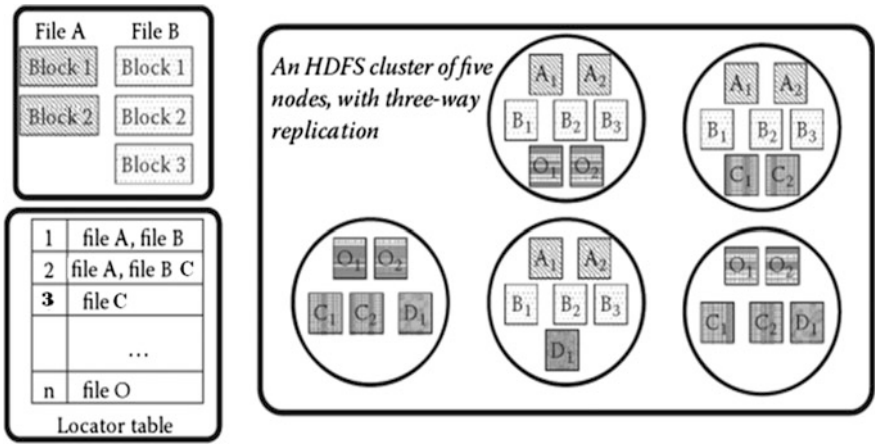

Fig. 4 Data co-location using a common locator table

Hadoop always try to schedule the job execution on the map instance that requires minimal amount of data transfers. In other words, a map instance is provided with a task which can be performed on the files already available on it. In case where a node has all of the required data blocks but is busy in running another task, Hadoop will allot the task to some other node. This may increase file transfer, but it is still more feasible than waiting for the previous job to finish.

d. Serialization and Lazy record Construction [8]—Serialization is the technique of converting structured objects into a byte stream. There are two main objectives of serialization:

- To transmit an object from one node to another over a network for the purpose of inter-process communication.
- To write an object to a persistent storage.

In Hadoop, the inter-process communication among multiple nodes is achieved by means of RPC (Remote Procedure Calls). RPC also uses serialization to convert the original message (to be sent over the network) into a byte stream. The receiver node receives the bye stream and again converts it into the original message. This reverse process is called deserialization.

In Hadoop, the main advantage of this technique is that only those columns are deserialized which are actually retrieved by the map nodes. Hence, it reduces the deserialization overhead as well as unnecessary disk I/O.

e. Columnar Compression [9]—Generally, columnar formats are likely to show fine compression ratios due to the fact that data within a column is expected similar than data across different columns. There are various compression methods which are adopted by column-oriented stores such as ZLIB, LZO, etc. All these methods have some advantages and certain limitations. For example,

ZLIB provides superior compression ratios but puts extra CPU overhead while decompression. LZO is generally employed in Hadoop to give better compression rates with lesser decompression overhead. It is usually adopted in cases where low decompression overhead is more required rather than the compression ratio. These compression methods use a special compression approach called block compression algorithm.

Block Compression: This approach compresses a block of columnar data at once. After compressing multiple blocks of the same column, they are loaded into a single HDFS block. The rate of compression and the overhead of decompression both are affected by using this strategy. Further, the size of compressed blocks, which can be defined at load time, also influences these factors. Each compressed block contains a header that gives information about the number of values in the block and the size of the block. By looking at the header, the system comes to know whether any value has been accessed in it. If there is no value in the block, then it can be skipped easily. And if, the header shows the presence of some values in it, the whole block is accessed and then decompressed.

f. Joins [5]—To easily and efficiently implement the join strategy, the schema and expected workload must be known in advance, as it helps in co-partitioning the data at the loading phase. The fundamental idea is—for given two input data sets, better performance can be achieved by:

- applying the similar partitioning function on join compatible attributes of both the data sets at loading phase and
- storing the co-group pairs with same join key on the same node would result in better performance.

As a result, join operations can be performed locally within each node at query time. Executing joins with this idea do not need any modifications to be made in the current implementation of Hadoop framework. The modifications need to be made only at the internal process of the data splitting.

5 Conclusion

MapReduce programming model was devised by Google to process large data sets. In this paper, we introduced column stores with row stores in terms of reading and accessing data. The paper also explored the features of parallel database systems in contrast with MapReduce systems. The main challenges of using column stores with MapReduce such as data co-location, data distribution, serialization, compression, joins, etc., have been discussed along with some feasible solutions.

References

1. Lin, Y., Agrawal, D., Chen, C., Ooi, B.C., Wu, S.: Llama: leveraging columnar storage for scalable join processing in the MapReduce framework. In: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD), pp. 961–972 (2011)
2. Abouzeid, A., Bajda-Pawlikowski, K., Abadi, D.J., Rasin, A., Silberschatz, A.: Hadoopdb: an architectural hybrid of mapreduce and Dbms technologies for analytical workloads. PVLDB **2**, 922–933 (2009)
3. Stonebraker, M., Abadi, D.J.. Batkin, D.J., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E., O'Neil, P., Rasin, A., Tran, N., Zdonik, S.: C-store: a column-oriented dbms. In: VLDB (2005)
4. Melnik, S., Gubarev, A., Long, J.J., Romer, G., Shivakumar, S., Tolton, M., Vassilakis, T.: Dremel: interactive analysis of web-scale datasets. PVLDB **3**(1), 330–339 (2010)
5. Kaldewey, T., Shekita, E., Tata, S.: Clydesdale: structured data processing on map-reduce. ACM (2012) (978-1-4503-0790-1/12/03)
6. Babu, S.: Dynamic colocation algorithm for Hadoop. In: Advances in Computing Commmunication and Informatics ICACCI (2014)
7. Peitsa: Map-reduce with columnar storage. Seminar on columnar databases
8. Floratou, A., et al.: Column oriented storage techniques for Map-Reduce. Proc. VLDB Endowment **4**(7) (2011)
9. Chen, S.: Cheetah: a high performance, custom data warehouse on top of MapReduce. Proc. Endowment (PVLDB) **3**(2), 1459–1468 (2010)
10. He, Y., Lee, R., Huai, Y., Shao, Z., Jain, N., Zhang, X., Xu, Z.: RCFile: a fast and space-e_cient data placement structure in MapReduce-based warehouse systems. In: Proceedings of International Conference on Data Engineering (ICDE), pp. 1199–1208 (2011)
11. Dittrich, J., Quian_e-Ruiz, J.-A., Jindal, A., Kargin, Y., Setty, V., Schad, J.: Hadoop++: making a yellow elephant run like a cheetah (without it even noticing). Proc. VLDB Endowment (PVLDB) **3**(1), 518–529 (2010)
12. Dittrich, J., Quian_e-Ruiz, J.-A., Richter, S., Schuh, S., Jindal, A., Schad, J.: Only aggressive elephants are fast elephants. Proc. VLDB Endowment **5**(11), 1591–1602 (2012)
13. Eltabakh, M.Y., Tian, Y., Ozcan, F., Gemulla, R., Krettek, A., McPherson, J.: CoHadoop: exible data placement and its exploitation in Hadoop. Proc. VLDB Endowment (PVLDB) **4** (9), 575–585 (2011)
14. Pavlo, A., et al.: A comparison of approaches to large scale data analysis. In: SIGMOD 2009, June 29–July 2, 2009, USA
15. Dean, J., Ghemawat, S.: Map-reduce: simplified data processing on large clusters. In: OSDI 2004, p. 10 (2004)
16. Dittrich, J., Quian_e-Ruiz, J.-A.: Efficient big data processing in Hadoop MapReduce. Proc. VLDB Endowment (PVLDB), **5**(12), 2014–2015 (2012)
17. Jindal, A., et al.: Trojan data layouts: right shoes for a running elephant. In: SOCC 2011, Portugal (2011)
18. Apache Software Foundation: Hadoop Distributed File System: Architecture and Design (2007)