Hindawi
Scientific Programming
Volume 2018, Article ID 2682085, 9 pages
https://doi.org/10.1155/2018/2682085

Research Article

Improving I/O Efficiency in Hadoop-Based Massive Data Analysis Programs

Kyong-Ha Lee,[1] **Woo Lam Kang**,[2] **and Young-Kyoon Suh**[3]

[1]*Research Data Hub Center, Korea Institute of Science and Technology Information, Daejeon, Republic of Korea*
[2]*School of Computing, KAIST, Daejeon, Republic of Korea*
[3]*School of Computer Science and Engineering, Kyungpook National University, Daegu, Republic of Korea*

Correspondence should be addressed to Young-Kyoon Suh; yksuh@knu.ac.kr

Received 30 April 2018; Revised 24 October 2018; Accepted 6 November 2018; Published 2 December 2018

Academic Editor: Basilio B. Fraguela

Apache Hadoop has been a popular parallel processing tool in the era of big data. While practitioners have rewritten many conventional analysis algorithms to make them customized to Hadoop, the issue of inefficient I/O in Hadoop-based programs has been repeatedly reported in the literature. In this article, we address the problem of the I/O inefficiency in Hadoop-based massive data analysis by introducing our efficient modification of Hadoop. We first incorporate a columnar data layout into the conventional Hadoop framework, without any modification of the Hadoop internals. We also provide Hadoop with indexing capability to save a huge amount of I/O while processing not only selection predicates but also star-join queries that are often used in many analysis tasks.

1. Introduction

Data volumes in scientific areas are unprecedently skyrocketing, and new sources and types of information are proliferating. We witness that the rate at which the data is being generated is faster, and the amount of the generated data is enormously larger than ever before. Apache Hadoop [1], an open-sourced implementation of Google's MapReduce [2], is a prominent data processing tool that processes a massive volume of data with a shared-nothing architecture in a parallel and efficient manner. Therefore, it has been widely recognized as an efficient tool for large-scale data analysis in the era of big data. Many algorithms for data analysis and mining have been rewritten for being run on Hadoop. That said, MapReduce has raised a nontrivial concern of exhibiting a clear tradeoff between I/O efficiency and fault tolerance [3]. Pavlo addressed that Hadoop MapReduce was 2 to 50 times slower than conventional parallel database systems except in the case of data loading [4]. Anderson and Tucek also noted that Hadoop was remarkably scalable but achieved very low efficiency per node, less than 5 MB per second processing rate [5]. The community thus has exerted to obtain efficient I/O, especially by building new frameworks over Hadoop [6–10].

In the same line, this manuscript addresses the issue of I/O inefficiency in MapReduce-based programs. Specifically, we focus on improving the I/O efficiency in massive data analysis when using Apache Hadoop. It is of critical importance to eliminate I/O bottleneck in Hadoop-based programs, considering a wide use of Hadoop in many scientific areas.

In this regard, we propose *ColBit*, a combination of bitmap indexes and an efficient columnar data layout for data blocks stored on the Hadoop distributed file system, *aka*, HDFS. *ColBit* dramatically improves the performance of Hadoop-based programs by reducing a huge amount of I/O while processing data analysis tasks. It is achieved by both (i) skipping unnecessary data reads during analysis and (ii) reducing the size of overall intermediate data size through the substitution of most of intermediate results with compressed bitvectors. At loading time, *ColBit* automatically transforms data block replicas into their corresponding columnar layouts. Therefore, users do not need to know details about internal data layouts. Both the layout transformation and the index-building tasks are performed at the loading time so that no overhead is imposed while processing data analysis tasks. In addition, a modification of the Hadoop internals is not necessary in our approach. Moreover, bitmap indexes facilitate the processing of not only selection predicates but also star-join queries, which are widely used in data warehouses, on Hadoop. Also, the bitmap indexes help save a large amount of I/O with compressed bitvectors. Finally, our join technique equipped with the bitmap indexes allows us to use only bitvectors for joining relations. The rest of this article is organized as follows. The following section discusses a rich body of existing literature related to our work. In turn, we propose a novel idea of columnar layout exploiting bitmap indexes. Next, we elaborate on query processing using the proposed bitmap indexes (ColBit), which are then evaluated by a microbenchmark test. Finally, Section 6 concludes our discussion by summarizing our contributions.

2. Related Work

Hadoop MapReduce is an open-sourced framework that supports MapReduce programming model introduced by Google [2]. The main idea of this model is to hide details of parallel execution so as to allow users to focus only on their data processing tasks. The MapReduce model consists of two primitive functions: *Map()* and *Reduce()*. The input for a single MapReduce job is a list of (*key*1, *value*1) pairs, *Map()* function is applied to each pair, and, then intermediate key-value pairs are computed as results. The intermediate key-value pairs are then grouped together on the key-equality basis, i.e., (*key*2, *list_of_value*2). For each *key*2, *Reduce()* function works on the list of all values, then produces zero or more aggregated results. Users can define *Map()* and *Reduce()* functions. Each processing job in Hadoop is broken down to as many Map tasks as input data blocks and one or more Reduce tasks. Hadoop MapReduce also utilizes HDFS as an underlying storage layer [11]. HDFS is a block-structured file system that supports fault tolerance by data partitioning and block replication, managed by a single or two master nodes.

Some approaches for improving the I/O performance in MapReduce-based programs have been proposed. Readers are referred to a recent survey for MapReduce and its improvements [3]. *Llama* has a columnar layout called *CFile* to help the join processing [12]. The idea is that input data are partitioned and sorted based on the selected column and stored column-wise on HDFS. However, since HDFS randomly determines the block placement at runtime, associated column values in a single row may not be located together in a node. Record Columnar File (*RCFile*) [13] developed by Facebook and used in Apache Hive and Pig projects rather chooses another approach similar to the PAX layout [4]. A single *RCFile* consists of a set of row groups, acquired by horizontally partitioning a relation, and then, values are enumerated and stored column-wise in each row group. A weak point of *RCFile* is that data placement in HDFS is simply determined by the master node at runtime. Therefore, all related fields in the same record cannot guarantee to be saved in the same node if each column is saved in a separate file in HDFS. *CoHadoop* [14] was also devised to locate associated files together in a node. To achieve this, *CoHadoop* extends HDFS with a file-level property, and files marked with the same locator are placed on the same set of slave nodes. Floratou et al. also propose a binary column-oriented storage format that stores each column in a spate file [10]. Both Floratou's work and RCFile exploit a column-wise data compression in a row group. Hadoop itself also provides data compression for mapped outputs to raise I/O efficiency while checkpointing intermediate results [1].

Hive [6] is an open-source project, which aims at providing a data warehouse solution on the Hadoop framework. It supports ad hoc queries with an SQL-like query language. Hive evaluates its SQL-like query by compiling the query into a directed acyclic graph that is composed of multiple MapReduce jobs. Hive also maintains a system catalog that provides schema information and data statistics, similar to other relational database systems. Hive currently adapts RCFile [13] and Apache ORC format, which is an improved version of RCFile that features block groups, as its mandatory storage types. HBase [7] is an open-source Java implementation of Google's Bigtable [8]. HBase is a wide-column store, which maps two arbitrary string values (row key and column key) and timestamp into an associated arbitrary byte array, working on HDFS. It features data compression, the use of bloom filter for checking the existence of data, and a log-structured storage. HBase is not a relational database, rather known to be a sparse, distributed multisorted map which works better for treating sparse data such as web addresses.

3. Columnar Layout Equipped with Bitmap Indexes

3.1. Columnar Storage Layout for HDFS. In the original HDFS, a logical file is physically partitioned into equal-sized blocks, and then, values in the logical file are enumerated row-wise in each physical block on local file systems. While this row-wise data layout provides fast data loading, it involves two major problems in the task of data analysis [10, 15]. First, a row-wise data layout requires unnecessary columns to be read even when only a few columns in a relation are accessed during query processing. Second, MapReduce itself leads to many I/Os as it simply delivers unnecessary columns to the next stages, i.e., reduce tasks and the next MapReduce job, checkpointing every tuple with unnecessary column values into disks or HDFS at each stage.

Inspired by the columnar storage model in read-optimized database systems [16] and bitmap index techniques [17], we devise our data layout equipped with bitmap indexes for HDFS. When relations are loaded, our system first partitions each relation into multiple groups such that the size of the base column values in each group is the same as the HDFS block size. In other words, the size of each data block in each group does not exceed the physical block size, i.e., basically 64 MB. This makes other columns have roughly the same block size. We then partition each group column-

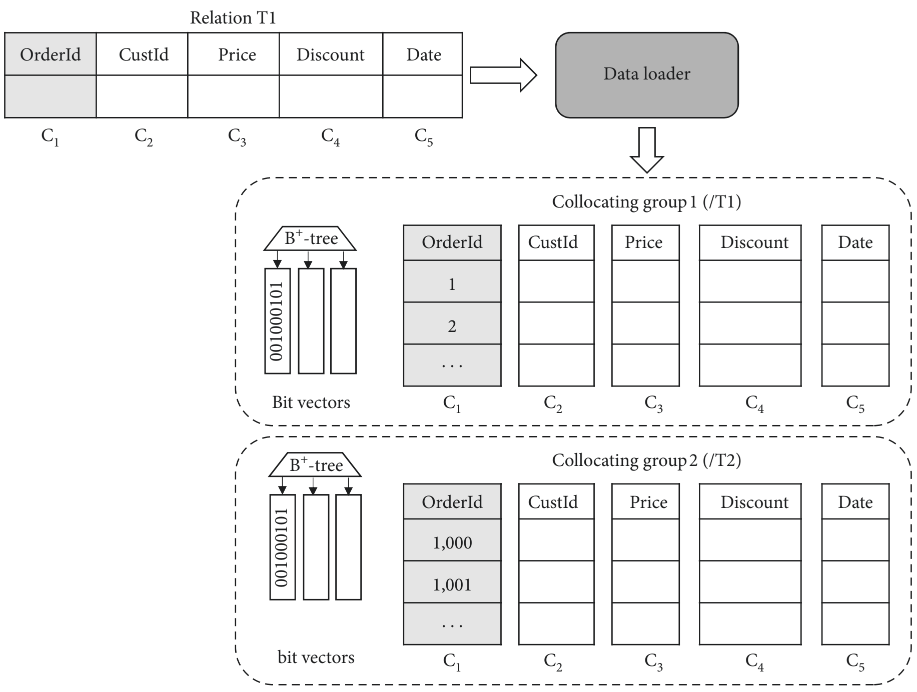

Figure 1: Data layout transformation.

wise and store each column in a separate binary file. All the columns in each group are stored as binary files, which can be optionally compressed in ZLIB [18], in a subdirectory associated with the group. Figure 1 illustrates how a single relation is partitioned and stored column-wise on HDFS in our approach.

In the figure, Relation T1 is loaded on our system and then each column, i.e., $C_1 \sim C_5$ in T1, and is basically stored as a single data block on HDFS. As such, to process an analytic query that treats only a few columns, *ColBit* does not need to read unnecessary columns.

Moreover, we allow users to group multiple columns into a single block to gain more I/O efficiency for query processing.

Suppose that a given analytic query Q1 requires projection on T1 so that a few columns, e.g., C1, C2, and C3, and a join key column is C5. Then, our system allows users to build only two blocks for the four column values: the three column values, i.e., C1, C2, and C3, are stored in a single block row-wise and C5 column values are stored in another block. Consequently, a MapReduce job reads a single block to get those column values for selection at once. This enables us to skip late materialization for showing row-wise results from column-wise storages [10, 15].

A problem in our approach is that the column values that constitute a single row may not be physically located together in a single node. This causes many I/Os as it is necessary to access them through a network during query processing. The reason is that Hadoop's original block placement policy does not guarantee the colocation of related blocks since it determines the block placement at runtime with no semantics about related blocks. *RCFile* [11] follows the PAX layout [14] to avoid this problem. In RCFile, a relation is first partitioned into a set of row groups, and each HDFS block is filled with these row groups, where the values are stored column-wise in each. Since all the related columns reside in a row group, it avoids the colocation problem while providing a columnar layout. However, it still suffers from unnecessary data reads since all columns are located in a single block. We instead solve this problem by providing a new block placement policy, which is allowed in HDFS since Hadoop release 0.21.0 [2]. With the new block placement policy, blocks are located together with other blocks in the same group in a physical node.

3.2. Bitmap Index for HDFS. While transforming a data layout, users are also allowed to build indexes on selected columns for each group. *ColBit* provides a bitmap index, which is considered to be better than traditional B+-tree indexes for analytical workloads [17]. A bitmap index is a collection of bitvectors that consists of only 0's and 1's. The bitvectors in a bitmap index correspond to distinct values in a certain value domain. The 1's in a single bitvector identify tuples in a relation that is represented by the bitvector. Several encoding schemes for bitmap indexes can be used to determine how to map 1's positions into distinct values in a value domain. For example, equality encoding scheme maps each 1's position into the position of a tuple that contains a certain column value which is represented by a bitvector. Shortly, the number of bitvectors in a bitmap index is the same as the cardinality of a column the index is built on. On the other hand, in range and interval encoding scheme, a single bitvector represents not only a single distinct values but also multiple column values. Readers are referred to Chan's work [19, 20] for understanding various encoding schemes used for building bitmap indexes. In our system, users are allowed to choose their encoding schemes for better support of various query types. For example, users are allowed to choose range encoding scheme rather than equality encoding scheme for efficiently processing range selection queries. Currently, *ColBit* provides three major encoding schemes including equality, range, and interval encoding.

To further improve the I/O efficiency, *ColBit* compresses all the bitvectors by using *run-length encoding* scheme, where a "run" represents a consecutive sequence of the same bits, i.e., either 0's or 1's. We also exploit WAH (word-aligned hybrid) compression scheme [17] that groups bits into compressed words and uncompressed words. The major profit is that bitwise operations are executed very efficiently benefiting from that bitwise operations are actually performed on word units in a computer system. Furthermore, *ColBit* is also devised to facilitate query processing by performing bitwise operations on compressed bitvectors without decompression.

Figure 2 presents how a bitvector of 8,000 bits is compressed into four 32-bit words in a system. The first bit of each word indicates whether the word is compressed. If the first bit is 1, the word is compressed with a sequence of bit values that the second bit called fill bit represents. For example, in the figure, the second word is compressed to represent $256 * 31$ 0's. The fourth word keeps the rest bits, which stores the last few bits that cannot be a single word by themselves.

To fetch the position of tuples fast from a bitvector that represents a certain distinct value, we build a virtual cursor that can run on the compressed bitvectors to compute the next 1's positions with no decompression. This virtual cursor consists of three values: (1) the position of the word W that a virtual cursor C is currently located at, (2) the position of the bit within W that C is located at, and (3) the position of bit within the rest word if C is located at the word.

Example 1. Suppose that a virtual cursor C currently indicates the last bit in the first word of a bit vector at the bottom of Figure 2, then the current cursor indicates the 31st tuple in a certain relation. If we find the position of the next 1 to fetch a tuple that contains the value that the bitvector represents, it begins from the first bit in the second word compressed. The word is filled with 0's since fill bit in the word indicates 0. We thus simply skip counting the position of the next 1 bit by bit simply moving cursor C by *run_length* of 0's without decompression. Thus, the next tuple that we must fetch will be 7,998($31 + 256 * 31 + 31$)th tuple.

We also exploit B+-tree index [21] to efficiently get relevant bitvectors for given certain values for selection queries. Note that compressed bitvectors are stored as a single file for each block group and also located together with their corresponding data blocks in the block group for I/O efficiency.

4. Query Processing with *ColBit*

In our system, MapReduce jobs are developed to widely utilize *ColBit* for processing analytical queries over a massive volume of data for better improving I/O efficiency. It is noteworthy that our approach is not restricted to processing only relational data but also processing other data models including graphs such as RDF dataset which also requires selection and join queries. Since MapReduce programming model does not have any dependency on data model and schema, it is widely accepted in the literature that the MapReduce programming model can deal with irregular or unstructured data more easily than they do with DBMS [3].

Figure 3 illustrates the execution plans for two popular queries in relational data analysis: *selection* and *star-join queries*.

In the MapReduce model, selection predicates are usually performed by mappers, and reducers group mapped outputs using a table name and column name pair. *ColBit* facilitates selection query processing using both columnar data layout and bitmap indexes, as shown in Figure 3(a). To achieve this, we extend the original input format class of the Hadoop framework. Our input format class first selects bitmap indexes built on selected columns and then, reads only the columnar files associated with the selected columns. While reading the values from the column files, our input format class outputs only values in the rows indicated by bit positions whose values are set to 1. As unnecessary column values are not read, we save many I/Os during query processing. Note that we follow the late materialization policy [15] so that tuple reconstruction is delayed to the reduce stage.

As MapReduce is initially designed to process a single large input, join processing on Hadoop has been challenging. Blanas et al. compared various join techniques devised for running on Hadoop MapReduce [22]. *Repartition join* is the most general join technique for Hadoop MapReduce [4, 22]. In a repartition join, each mapper appends a tag as a key value to each row so as to identify which relation the rows come from. Rows with the same key value are then transferred to a reducer during a shuffling phase. Finally, each reducer joins the rows on a key-equality basis. However, a repartition join does not support star-join queries well. The rationale behind this assertion is that a star-join query usually is executed by multiple binary joins on a fact table and multiple dimension tables. Therefore, multiple MapReduce jobs are usually required to perform multiple binary joins on Hadoop [4, 22]. Moreover, many I/Os are consumed just for data transmission to the next stages, i.e., reduce tasks or the next MapReduce jobs, while process join operations since the philosophy of MapReduce is to sacrifice I/O efficiency for guaranteeing fault tolerance by utilizing frequent checkpointing and block-level data replication.

With *ColBit*, our initial idea is to save many I/Os by delivering compressed data structures, rather than data themselves, to the next stage. We further developed the idea to holistically improve star-join query processing on Hadoop. Star-join queries usually restrict a set of tuples from the fact table using selection predicates on one or more dimension tables and then perform some aggregations on the restricted fact table, often grouping tuples by the attributes of other dimension tables. As a result, join operations should be performed between the fact table and dimension tables for each selection predicate and for each

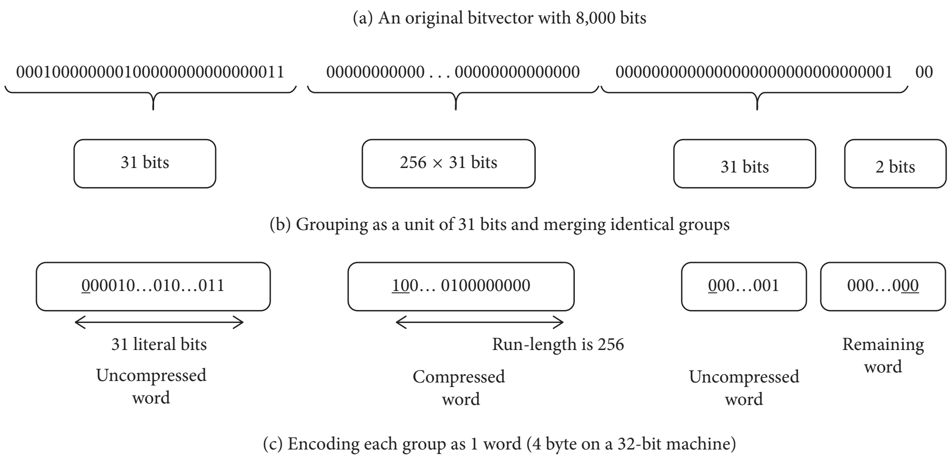

FIGURE 2: Example of bitvector compression.

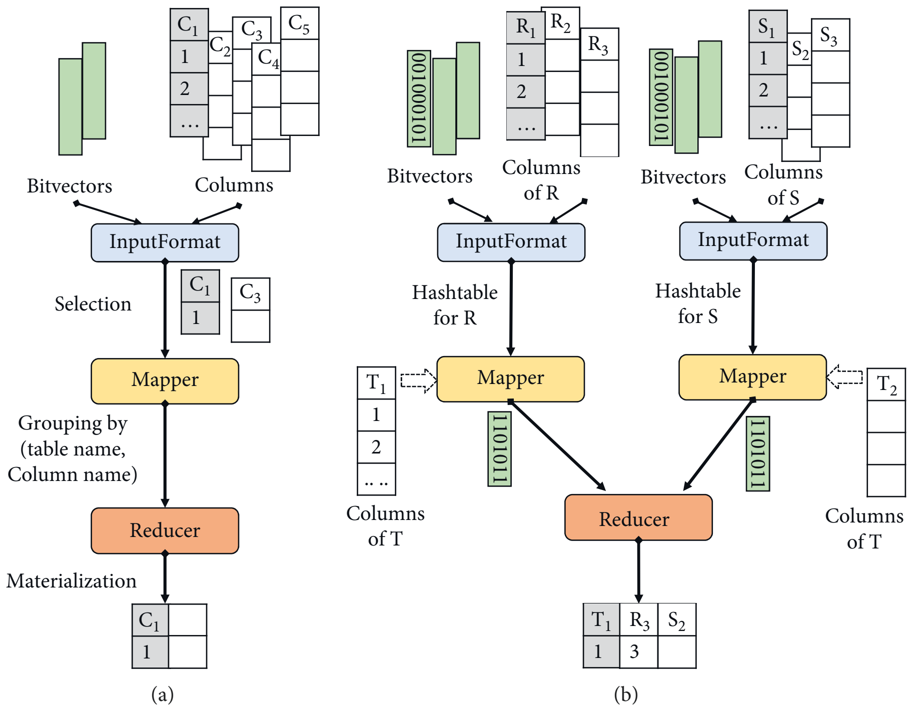

FIGURE 3: MapReduce plans for relational data processing. (a) Selection query. (b) Star-join query.

group-by-aggregation. A sample query in Example 2 is to find the total revenue of customers in Asia who purchased a product supplied by Asian suppliers between the years 1992 and 1998, and the total revenue must be grouped by each nation and transaction year.

Example 2. A sample star-join query from Star Schema Benchmark [23].

SELECT c. nation, s. nation, d. year, sum (lo.revenue) as revenue

FROM customer AS c, lineorder As lo, supplier AS s, dwdate AS d

WHERE lo.custkey = c.custkey

AND lo.suppkey = s.suppkey

AND lo.lo.orderdate = d. datekey

AND c.region = "ASIA"

AND s.region = "ASIA"

AND d. year ≥1992 and d. year ≤1998

GROUP BY c. nation, s. nation, d. year

ORDER BY d. year asc, revenue desc;

We adapted *invisible join* technique, which was initially devised for column-oriented databases, to reduce the volume of data accessed out of order from dimension tables [15]. Unlike traditional join techniques, invisible join is a variant of late materialized join that focuses on minimizing the values required to be extracted out of order. Therefore, it rewrites join operations on fact table columns so that joins can be performed at the time when other selection predicates are evaluated to the fact table. The selection predicates are evaluated by using other data structures such as hash table lookups. We tailored the invisible join technique to fit the MapReduce model with *ColBit*'s bitmap index. Figure 3(b) illustrates our join technique implemented with two MapReduce jobs on Hadoop.

In the first MapReduce job, we extend the InputFormat class to apply each selection predicate to dimension tables for selecting key values that satisfy the predicate from the dimension tables. The InputFormat class then returns a Java HashMap object that is used for checking which key values satisfy the predicate. At the map stage, each mapper uses the hash maps for finding tuples of the fact table that satisfy the selection predicate. This is performed by looking up the hash table with values of the foreign key column in the fact table. It then creates a bitvector in which the positions of 1's represent tuples in the foreign key column that satisfy the selection predicate. Bitvectors from all mappers are then transferred and grouped by row groups and finally merged into a single bitvector by a bitwise-AND operator at the reduce stage. A merged bitvector represents the positions of all tuples that satisfy all predicates in the fact table. Note that when reading fact table columns at the map stage, mappers directly read only relevant columns from columnar files stored on HDFS. In the second MapReduce job, we use the merged bitvector for the actual joining process. Each mapper reads fact table columns and extracts foreign key values in the rows, where the corresponding bits are set to 1 in the bitvector. The foreign key values are then used to extract column values from the dimension tables. In the second MapReduce join, grouping and aggregations are applied to the selected and joined results that came from the first MapReduce job. Based on the observation that a MapReduce job simply works like a group-by-aggregation query in database systems, it is reasonable that grouping and aggregating are assigned to a separated MapReduce job. Note that the second MapReduce job is omitted in Figure 3 owing to a space limitation.

The major contributions of our approach are summarized in the following: first, as intermediate results are delivered as compressed bitvectors, we can save a huge amount of I/O. Second, by applying bitwise operations on compressed bitvectors, we can easily compute multiple selection predicates. Finally, when it comes to handling a star-join query, no more than two MapReduce jobs are sufficient in our approach while much more MapReduce jobs are needed in other join techniques. Note (or Recall) that a single MapReduce job implements each join operation.

5. Experimental Study

We performed our microbenchmark test on a 9-node cluster, where each node is equipped with an Intel i7-6700 3.4 GHz processor, 16 GB of memory, and a 7200 RPM SATA3 HDD, running on CentOS 7.5. All nodes are connected through a gigabit switching hub. We used a subset of the TPC-H benchmark dataset [24] with three queries (Q1 and Q6 for selection queries, and Q3 for a star-join query), as shown in Table 1. We compared our approach with the original HDFS data layout, RCFile [13], and ORC [25]. Moreover, we compared our approach with an improvement for Hadoop, i.e., Apache Hive version 2.3.3., which currently accepts RCFile and ORC as its storage types [25].

Table 1: Statistics of the selected dataset from TPC-H [24].

Table name	Data size	The # of rows in a table
Customer	S ∗ 24 MB	S ∗ 150,000
Orders	S ∗ 171 MB	S ∗ 1,500,000
Lineitem	S ∗ 759 MB	S ∗ 6,001,215

S: scale factor.

All programs were implemented using JDK 8 and Hadoop release 2.7. Figure 4 shows the elapsed time for loading TPC-H datasets on. In all cases, all of the data layout schemes scaled linearly with an increase in the volume of input data. Among them, Hadoop's row-wise data layout (sequential file) showed the best loading time. This was because it does not require any layout transformation. All the column-wise data layouts exhibited relatively long loading time. ORC shows the worst loading time followed by RCFile. Among the column-wise data layouts, *ColBit* exhibited the best performance regardless of bitvector encoding schemes applied to bitmap indexes. Specifically, *ColBit* with bitmap indexes paid slightly more time for an additional bitmap index buildup. However, fast query processing time could compensate for the marginal loss of loading time.

Figure 5 shows the sizes of data reads at the map stage for TPC-H Q1 and Q6. *ColBit* substantially reduced the size of data reads using mappers. Specifically, *ColBit* with bitmap indexes showed the smallest data reads as it further skipped many unnecessary values during the query processing. RCFile and ORC recorded the second and third best performance at the size data read. The row-wise layout labeled "Original" recorded the worst performance since all of the data should be understandably read for batch processing with Hadoop.

To see how much I/O affected the overall performance, we measured elapsed time for TPC-H queries. These queries were selected with care for fair comparison. Note that Apache Hive can exploit two columnar layouts for query processing: RCFile and ORC. Accordingly, we configured Apache Hive with the two columnar layouts and measured the performance of each of the layouts. Also, note that MapReduce jobs could use the columnar layout without any help of Apache Hive. So, we examined which columnar layout could influence the performance of MapReduce jobs as well.

Figure 6 presents the execution time for two selection queries. Again, *ColBit* outperformed other approaches by up to two orders of magnitude as it significantly reduced I/O from the data reading. It is noteworthy that neither I/O efficiency nor query processing time was improved by RCFile. The reason for this was that with PAX-similar layout, unnecessary columns in a row group were still read in RCFile

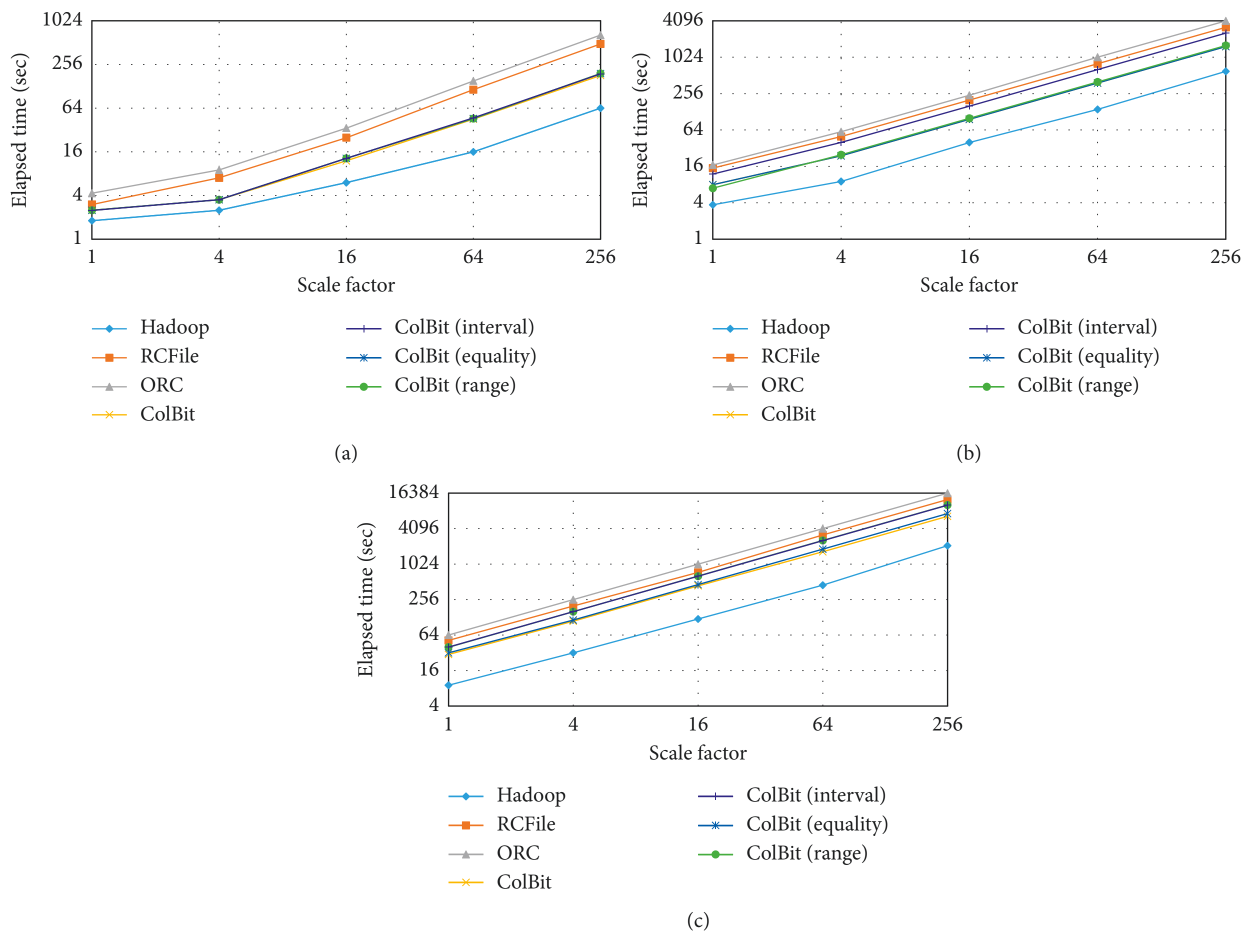

Figure 4: Data loading time. (a) Customer table. (b) Orders table. (c) Lineitem table.

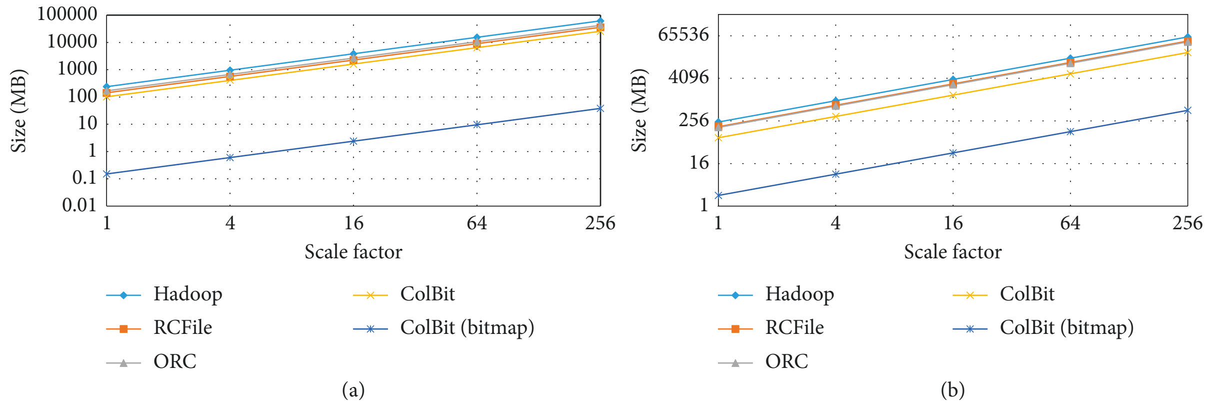

Figure 5: Sizes of input data read at the map stage. (a) TPC-H Q1 query. (b) TPC-H Q6 query.

during query processing as noted earlier in Section 2. Apache Hive that utilized RCFile and ORC as a storage format also exhibited better performance than that of original Hadoop MapReduce jobs whose inputs were either RCFile or ORC. This was because an up-to-date version of Apache Hive bettered the performance of selection query processing by (i) removing unnecessary Map phases and (ii) adopting vectorized query execution model [25].

Finally, Figure 7 presents the performance of our star-join query analysis based on MapReduce. As *ColBit* reduces many data reads with bitvectors and a columnar data layout, we witnessed a significant performance gain in the query execution time. It is also noteworthy that for the star-join query, Apache Hive did not show better performance even when compared to original Hadoop MapReduce jobs that utilizes invisible joining whose inputs are either RCFile or

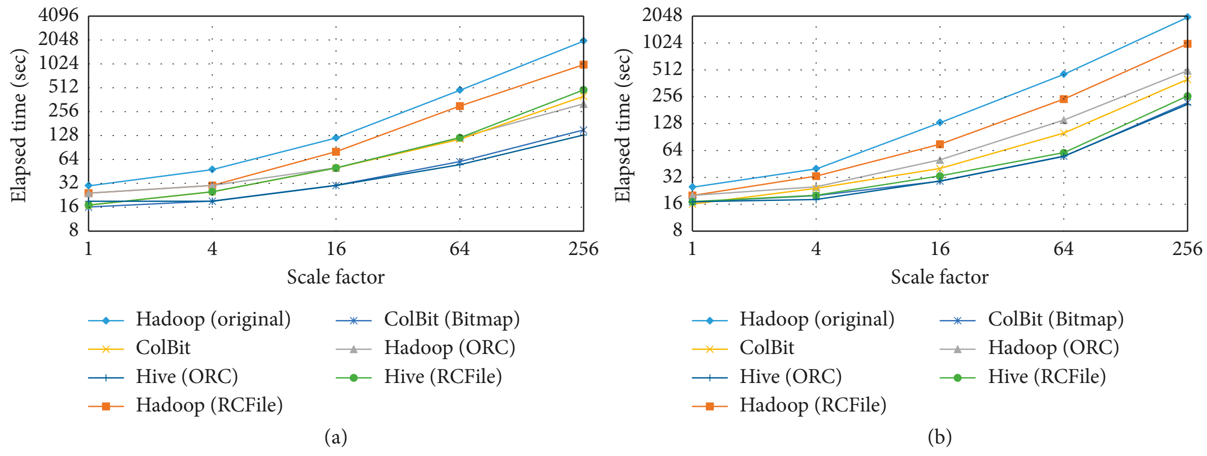

Figure 6: Execution time of the selection queries. (a) TPC-H Q1 query. (b) TPC-H Q6 query.

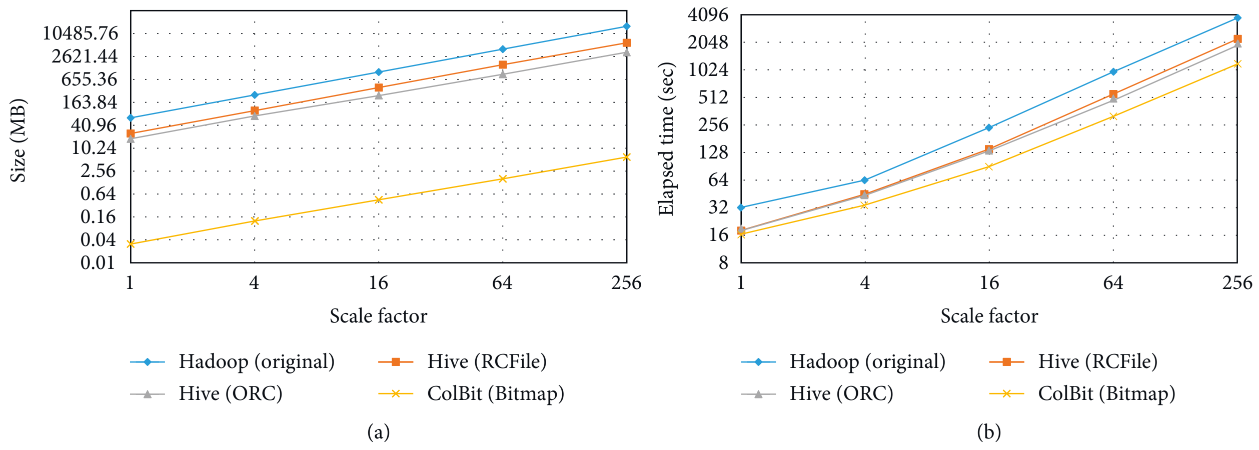

Figure 7: Correlation between query execution time and the size of the intermediate results (TPC-H Q3). (a) Intermediate result size. (b) Execution time.

ORC. The reason was that Apache Hive's query planner builds a multiple MapReduce for a given user-written HiveQL statement, as Hive's query planner builds a single MapReduce job for each binary join operator. But our solution could finish the star-join query only with two MapReduce jobs involving this performance gap.

6. Conclusion

We address the problem of I/O inefficiency in Hadoop-based data analysis with a columnar data layout and compressed bitmap indexes in this article. Experimental results exhibit that our approach outperforms both the Hadoop-based programs and Apache Hive that utilize RCFile and ORC, recent columnar data layouts for Hadoop. Furthermore, our techniques do not require any modification of Hadoop internals.

Therefore, any Hadoop version can be incorporated with our techniques with no effort. As future work, given a set of queries, we intend to find an efficient method to choose index types and to group columns into a few column groups so as to automatically maximize the I/O efficiency and minimize the chance of tuple reconstructions.

Data Availability

The TPC-H decision support benchmark dataset which has been used to perform our experiments is provided by TPC (Transaction Processing Performance Council) at http://tpc.org/tpch. The source code used for our experiments is also deliverable upon request.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was funded by the Korea Institute of Science and Technology Information, Korea (grant number K-18-L11-C03 and K-18-L15-C02-S18). This work was also supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2018R1A6A1A03025109).

References

[1] Apache Hadoop, "Apache Hadoop project," December 2010, http://hadoop.apache.org.

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

[3] K. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon, "Parallel data processing with MapReduce: a survey," *ACM SIGMOD Record*, vol. 40, no. 4, pp. 11–20, 2012.

[4] A. Pavlo, "A comparison of approaches to large-scale data analysis," in *Proceedings of SIGMOD Conference*, pp. 165–178, Providence, RI, USA, June 2009.

[5] E. Anderson and J. Tucek, "Efficiency matters!," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 1, pp. 40–45, 2010.

[6] A. Thusoo, J. S. Sarma, N. Jain et al., "Hive: a warehousing solution over a map-reduce framework," *Proceedings of VLDB Endowment*, vol. 2, no. 2, pp. 1626–1629, 2009.

[7] L. George, *HBase: the Definitive Guide: Random Access to your Planet-Size Data*, O'Reilly Media Inc., Sebastopol, CA, USA, 2011.

[8] F. Chang, J. Dean, S. Ghemawat et al., "Bigtable: a distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, pp. 1–26, 2008.

[9] M. Zaharia, M. J. Franklin, A. Ghodsi et al., "Apache Spark: a unified engine for big data processing," *Communications of ACM (CACM)*, vol. 59, no. 11, pp. 56–65, 2016.

[10] A. Floratou, U. F. Minhas, and F. Özcan, "SQL-on-Hadoop: full circle back to shared-nothing database architectures," *Proceedings of VLDB Endowment*, vol. 7, no. 12, pp. 1295–1306, 2014.

[11] K. Shvachko, "The Hadoop distributed file system," in *Proceedings of IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–10, Lake Tahoe, NV, USA, May 2010.

[12] Y. Lin, D. Agrawal, C. Chen, B. C. Ooi, and S. Wu, "Llama: leveraging columnar storage for scalable join processing," in *Proceedings of ACM SIGMOD Conference*, pp. 961–972, Athens, Greece, June 2011.

[13] Y. He, R. Lee, Y. Huai et al., "RCFile: a fast and space efficient data placement structure in mapreduce-based warehouse systems," in *Proceedings of 27th IEEE ICDE Conference*, pp. 1199–1208, Hannover, Germany, April 2011.

[14] M. Y. Eltabakh, Y. Tian, F. Özcan, R. Gemulla, A. Krettek, and J. McPherson, "CoHadoop: flexible data placement and its exploitation in Hadoop," *Proceedings of the VLDB Endowment*, vol. 4, no. 9, pp. 575–585, 2011.

[15] D. J. Abadi, S. R. Madden, and N. Hachem, "Column-stores vs. row-stores: how different are they really?," in *Proceedings of ACM SIGMOD International Conference on Management of Data-SIGMOD'08*, pp. 967–980, Vancouver, BC, Canada, May 2008.

[16] S. Harizopoulos, V. Liang, D. J. Abadi, and S. Madden, "Performance tradeoffs in read- optimized databases," in *Proceedings of VLDB Conference*, pp. 487–498, Seoul, Korea, September 2006.

[17] K. Wu, E. J. Otoo, and A. Shoshani, "Optimizing bitmap indices with efficient compression," *ACM Transactions on Database Systems*, vol. 31, no. 1, pp. 1–39, 2006.

[18] P. Deutsch and J.-L. Gailly, "ZLIB compressed data format specification version 3.3," No. RFC 1950, Internet Engineering Task Force, Fremont, CA, USA, 1996.

[19] C. Y. Chan and Y. E. Ioannidis, "An efficient bitmap encoding scheme for selection queries," in *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data-SIGMOD'99*, pp. 215–226, Philadelphia, PA, USA, June 1999.

[20] C. Y. Chan and Y. E. Ioannidis, "Bitmap index design and evaluation," *ACM SIGMOD Record*, vol. 27, no. 2, pp. 355–366, 1998.

[21] D. Comer, "Ubiquitous B-tree," *ACM Computing Surveys*, vol. 11, no. 2, pp. 121–137, 1979.

[22] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian, "A comparison of join algorithms for log processing in MapReduce," in *Proceedings of ACM SIGMOD Conference*, pp. 975–986, Indianapolis, IN, USA, June 2010.

[23] P. O'Neil, E. O'Neil, X. Chen, and S. Revilak, "The star schema benchmark and augmented fact table indexing," in *Technology Conference on Performance Evaluation and Benchmarking*, Springer, Berlin, Germany, 2009.

[24] Transaction Processing Performance Council, "TPC-H benchmark specification," April 2008, http://www.tpc.org/hspec.html.

[25] Y. Huai, X. Zhang, A. Chauhan et al., "Major technical advancements in Apache Hive," in *Proceedings of ACM SIGMOD Conference*, pp. 1235–1246, Snowbird, UT, USA, June 2014.