Performance Evaluation and Optimization of Multi-dimensional Indexes in Hive

Yue Liu, Shuai Guo, Songlin Hu, Tilmann Rabl
Danyang Gu, Yue Wang, Hans-Arno Jacobsen, Jintao Li

Abstract—Apache Hive has been widely used for big data processing over large scale clusters by many companies. It provides a declarative query language called HiveQL. The efficiency of filtering out query-irrelevant data from HDFS closely affects the performance of query processing. This is especially true for multi-dimensional, high-selective, and few columns involving queries, which provides sufficient information to reduce the amount of bytes read. Indexing (Compact Index, Aggregate Index, Bitmap Index, DGFIndex, and the index in ORC file) and columnar storage (RCFile, ORC file, and Parquet) are powerful techniques to achieve this. However, it is not trivial to choosing a suitable index and columnar storage based on data and query features. In this paper, we compare the data filtering performance of the above indexes with different columnar storage formats by conducting comprehensive experiments using uniform and skew TPC-H data sets and various multi-dimensional queries, and suggest the best practices of improving multi-dimensional queries in Hive under different conditions.

Index Terms—Hadoop, Hive, multi-dimensional index, performance evaluation.

1 Introduction

For today's enterprises, fast and timely analysis on large scale data has become an essential task, since it is the cornerstone of decision-making for managers. Data models that involve multiple dimensions are fairly common, especially for traditional enterprises, for example, Smart Grid applications [29] and retail business analysis [11]. Queries often contain multi-dimensional predicates and analyze data from various perspectives. For these queries, only a subset of data is query-related and it is crucial to filter out irrelevant data, especially for I/O intensive applications.

Because of its flexible scalability, high availability, and cost efficiency, Hadoop is adopted by many enterprises as the underlying system to store various kinds of massive data. Hive [38] is a data warehouse like tool on top of Hadoop, and it provides the declarative query language HiveQL, which empowers analysts to run deep analysis on structured data stored on HDFS. After receiving a query, Hive is responsible for transforming and optimizing it into a directed acyclic graph (DAG) of MapReduce jobs. Compared with other emerging SQL on Hadoop systems, for example Presto [10], Drill [2], Pig [9], and Impala [7], Hive features the most comprehensive index techniques. Examples of supported indexes are Compact Index [5], Aggregate Index [3], Bitmap Index [4], index in the ORC file [25], and DGFIndex [29]. In addition, columnar storage is another efficient technique to filter out data, because it can skip query-irrelevant columns, examples of columnar storage in Hive are RCFile [24], ORC file [25], and Parquet [1]. Thus, index and columnar storage are effective techniques to filter query-irrelevant data horizontally and vertically.

However, it is not trivial to choose suitable kinds of indexes and columnar storage formats for specific application scenarios, the reasons are as follows: (1) the storage form of an index table is different, for example, ORC file embeds index in data file, Compact Index stores its index table as a Hive table. Different storage forms will lead to different reading efficiency of index. (2) the filtering granularity is different, for example, the minimum filtering granularity of ORC file is 10000 lines, and that of Compact Index is a split. (3) to improve the filtering performance of indexes, sorting table is a frequently used pre-processing method, currently Sort By, Order By and Cluster By are available methods to achieve this. But different sorting methods will lead to different relative order of records and different influence on index performance. (4) these indexes also have different filtering performance on different underlying storage formats. (5) query selectivity, for example, some indexes is applicable for large selectivity query and some are applicable for small selectivity query. (6) there is no comprehensive description and comparison of current indexes in Hive that can be used. Thus, it is challenging problem to choose suitable index technique in different scenarios for Hive users.

In addition, index comparison and selection has been widely studied in traditional RDBMS area [15], [32] and spatial database area [13], [26], [31], but they either focus on one-dimensional indexes, or are mainly suitable for spatial data. Moreover, because the storage and reading model of index table and data table is different between traditional database and Hive: for the former, the data is well-organized as heap file or tree-based file and processed locally in

- *Y. Liu, S. Guo, D. Gu and Y. Wang are with the Institute of Computing Technology, Chinese Academy of Sciences and University of Chinese Academy of Sciences, Beijing, China. E-mail: {liuyue01, guoshuai01, gudanyang, wangyue89}@ict.ac.cn*
- *S. Hu is with the Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China. E-mail: husonglin@iie.ac.cn*
- *T. Rabl is with the Middleware Systems Research Group, University of Toronto, Toronto, Canada. E-mail: tilmann.rabl@utoronto.ca*
- *H. Jacobsen is with the Middleware Systems Research Group, University of Toronto, Toronto, Canada. E-mail: jacobsen@eecg.toronto.edu*
- *J. Li is with the Institute of Computing Technology, Chinese Academy of Sciences. Beijing, China. E-mail: jtli@ict.ac.cn*

Manuscript received April 19, 2005; revised August 26, 2015.

one server. However, for the latter, the data is stored in distributed file system (for example HDFS), and processed by many servers parallel. Thus, the conclusions of previous work can not be directly reference by Hive user. Besides, in the SQL on Hadoop system area, the existing work [20] only did a simple one-dimensional index benchmark of ORC file, does not yet have comprehensive summary and comparison of various indexes in Hive on multi-dimensional query processing.

To provide guidelines for Hive user and find the best practice of improving the multi-dimensional query processing, we perform a detailed survey and summary on the indexes of Hive. Besides, we use uniform TPC-H [11] and skew TPC-H [37] workload to generate test data, and by doing extensive experiments, to get insights of data filtering characteristics and applicable scenarios of different indexes in Hive. Moreover, we also analyze the influence of the combination of index and columnar storage on reducing redundant data I/O. Because all SQL on Hadoop systems use HDFS as underlying storage and share the same data reading schema, thus the best practice is also applicable to other SQL on Hadoop systems.

Our contributions are three-fold:

1) We perform an extensive survey and comparison of current indexing techniques in Hive, and we also point out their strengths, weaknesses, and applicable scenarios.
2) For comparison fairness, we present two improvements to the original DGFIndex: first, we migrate it to columnar storage, RCFile, and ORC file. Second, we implement a z-order [30] based slice placement method to improve the sequential reading performance.
3) We comprehensively benchmark multiple representative indexes on columnar storage in Hive and evaluate related issues on data reading efficiency via indexes. Based on the results, we present best practices to improve multi-dimensional query performance with indexes and columnar storage. Furthermore, we summarize several important guidelines for index design in SQL on Hadoop systems.

The rest of this paper is organized as follows. In Section 2, we introduce some background knowledge about Hadoop and Hive. In Section 3, we survey current indexes in Hive. In Section 4, we summarize the different preprocessing methods to improve Compact Index and ORC file. Section 5 introduces two improvements of DGFIndex. In Section 7, we will give a detailed analysis of the experimental results. Section 6 summarizes the data filtering process of various indexes in Hive. Section 8 gives a summary of the evaluations results and provides some guidelines for best practices of multi-dimensional query processing in Hive. In Section 9, we present related work. Finally, Section 10 concludes the paper.

2 Hadoop and Hive Overview

In the following sections, we first give background information about Hadoop and Hive. Here, we mainly focus on the index-related perspective.

2.1 Hadoop

Hadoop is an open source implementation of the Google File System [23] and MapReduce [16], namely Hadoop File System (HDFS) and Hadoop MapReduce. Hadoop is a master/slave architecture. The master of HDFS is called NameNode, it stores the meta information of the file system, which is held in memory to accelerate the accessing. The meta information includes the structure of all directories, the file-block mapping, locations of blocks on the DataNodes, etc. The slaves of HDFS are called DataNodes, they store all non meta data. In HDFS, data files are divided into many equal-size chunks (default is 64MB), we refer to them as a *split* in this paper. The splits that belong to a single file are distributed over DataNodes by default with tripple replications for fault tolerance. The master of MapReduce is called JobTracker, and the slaves are called TaskTrackers. JobTracker is responsible for getting the splits' location of input data file from NameNode and assigning a mapper for each split to process it. Generally, a MapReduce job consists of three phases: Map, Shuffle, Reduce. In the Map phase, each mapper parses and reads every record from a split by iteratively calling the *next* function of a specific *RecordReader*, then applies the map function onto each record. In the Shuffle phase, a *Partitioner* will assign each output record of mappers to some reducer, the assignment method can be specified by user. In the Reduce phase, reducer applies the reduce function to the value list that has the same key.

2.2 Hive

Hive is a kind of SQL on Hadoop system, which provides HiveQL, a SQL-like declarative language. Hive can translate each HiveQL program into a DAG of MapReduce jobs. In the Map phase and Reduce phase of each MapReduce job, they consists of various Hive operators. A table in Hive is a directory in HDFS, the files in this directory store the data of this table. The file's storage formats are classified into two categories: row storage (TextFile, SequenceFile, etc.) and columnar storage (RCFile, ORC file, Parquet, etc.). The minimum reading unit of row storage is one record, and the minimum reading unit of columnar storage is a row group, whose size is much smaller than split, but much larger than one line. For example, the RCFile [24] stores the data file as a sequence of 4MB row groups, in contrast to that, the ORC file first organizes the data file into stripes and then organizes the records in each stripe into row groups, whose size is 10,000 lines.

An index stores the search key and the position information of corresponding row groups. In Hive, the data filtering process via indexes can be divided into three phases: (1) *searching the index*, after receiving query, Hive first searches the index table with the predicate to find the position information of query-relevant row groups. (2) *Filtering irrelevant splits*, second, Hive filters out the query-irrelevant splits that do not overlap with the query-relevant row groups. After the splits filtering, the candidate splits are as the input of MapReduce job, and then processed by each mapper. (3) *Filtering irrelevant row groups in each split*, in each mapper, the *next* function of RecordReader can make use of the above position information of query-relevant row groups to skip unrelated row groups in each split. Moreover, with

TABLE 1
Schema of a 3-dimensional Compact Index

Column Name	Type
index dimension 1	type in base table
index dimension 2	type in base table
index dimension 3	type in base table
_bucketname	string
_offset	array<bigint>

columnar storage, each mapper only need to read the query-related columns.

3 Indexes in Hive

In this part, we survey current indexes in Hive, including partition, Compact Index, the index in ORC file, and DGFIndex. Because Parquet does not support index in latest Hive [1], [8], we do not cover it in this section, but we still use it as a baseline in our experiments.

3.1 Partition

Partition horizontally divides a table into several sub-tables based on the value of the partition dimension (usually it is a date or date range), each sub-table is a directory in HDFS. When Hive processes a query with partitions, it can locate relevant partitions as the input of the MapReduce job. Thus, partition can be seen as a coarse-grained index. However, for multi-dimensional queries, partition is not flexible, especially when the cardinality, which represents the number of distinct values, of each dimension is very large. Creating partitions on multiple high-cardinality dimensions will generate too many directories, which will occupy a large amount of memory of the NameNode, because all metadata of HDFS is stored in the NameNode's memory as described in Section 2.1. Besides, when processing query in the case of too many partitions, Hive needs to read lots of meta information about partitions from metastore, which will create substantial overhead. However, partition is a good complement for indexes, because indexes are always created on each partition.

3.2 Compact Index

Compact Index [5], Aggregate Index [3], and Bitmap Index [4] are the first kind of indexes that were developed for Hive. Because Compact Index is the basis of the other two indexes, we mainly focus on Compact Index. Compact Index can only filter unrelated data in split granularity. The index table is stored in a separate table of Hive, its schema is shown in Table 1. When creating a Compact Index in Hive, the HiveQL statement in Listing 1 is used to populate the index table. The *_bucketname* represents the data file name, and the *_offset* stores all the offsets in *_bucketname* of the corresponding combination of index dimensions. Thus, the Compact Index table stores all the combinations of multiple index dimensions that appear in the table and the corresponding position information in different data files.

When processing queries with Compact Index, Hive first runs a query with the same predicate of the original query on the index table to get the relevant position information,

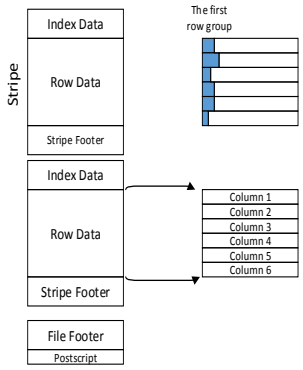


Fig. 1. ORC file structure

and then uses it to filter irrelevant splits. However, for the index dimensions with high cardinality, the Compact Index table will be very large, because the number of the combinations of multiple index dimensions is extremely large. Moreover, for queries with high-selectivity, Compact Index will easily result in a memory overflow of the JobTracker, because it needs to load the relevant position information into memory and to filter unrelated splits before starting the MapReduce job. Besides, the filtering performance of Compact Index is sensitive to the relative order of records. Thus, to improve Compact Index, users need to choose suitable sorting methods to preprocess the table before creating Compact Index.

```
INSERT OVERWRITE TABLE IndexTable
SELECT <index dimension list>,
       INPUT_FILE_NAME,
       collect_set(BLOCK_OFFSET_INSIDE_FILE)
  FROM BaseTable
 GROUP BY <index dimension list>,
          INPUT_FILE_NAME
```

Listing 1. Creation of a Compact Index

3.3 ORC File and Its Index

ORC (Optimized Row Columnar) file is a kind of columnar storage format in Hive, and it supports a lightweight index. Next, we will describe ORC file in the perspective of indexing and query processing with an index. As shown in Figure 1, an ORC file consists of a sequence of stripes, each stripe is composed of *index data*, *row data*, and the *stripe footer*. The default size of stripe is 64MB, and the default block size for ORC file is 256MB, so each block contains 4 stripes. In a stripe, the *index data* records the min and max value of each column for every 10,000 rows (a row group) as an index entry. The *row data* stores the real data in columnar style. The *stripe footer* records the location of *index data* and *row data*, and the encoding method of each column. The *file footer* records the min and max value of each column for every stripe (stripe statistics). When reading an ORC file, unrelated stripes can be filtered based on the predicate and the stripe statistics in the *file footer*. Thus, splits that contain no query-related stripes will be filtered out. Besides, unrelated row groups of query-relevant columns in each

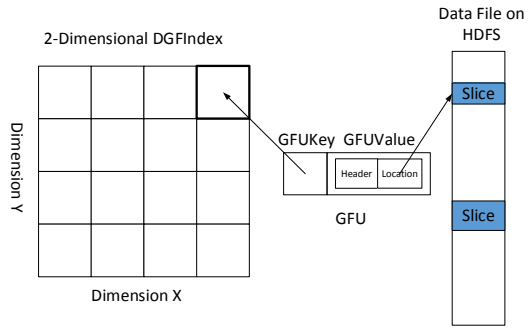


Fig. 2. DGFIndex architecture

stripe can also be filtered out based on the *index data*. That means, ORC file can filter data in row group granularity. Because the index in ORC file is based on min and max value of a column, it is sensitive to the relative order of records in the data file, and users need to choose a suitable sorting method to preprocess an ORC file to improve the filtering performance.

3.4 DGFIndex

DGFIndex [29] is a multi-dimensional index that we proposed for the case that some (or all) index dimensions have a high cardinality, which will lead to extremely large index table sizes for the Compact Index. Figure 2 shows a 2-dimensional DGFIndex, dimension X and Y can be any two index dimensions of the base table. DGFIndex splits the data space into small units with a grid file-like splitting policy. The split unit is named as grid file unit (GFU). It is stored as GFUKey/GFUValue. GFUKey is the left lower coordinate of each GFU in the data space. GFUValue consists of two parts: the header and the location of the data slice stored in HDFS. All records located in the same GFU are stored as a continuous segment of a file in HDFS, which is called a *Slice*. The header in GFUValue contains precomputed aggregations of numerical dimensions, which can be any user defined function (UDF) supported by Hive. The location in GFUValue contains the offset information of the corresponding slice. All records in a slice belong to the same GFU. In DGFIndex, the GFUKey/GFUValue pairs are stored in HBase, which accelerates the reading speed of the index with key-based and columnar reading style.

When creating a DGFIndex in Hive, the data in the base table has to be reorganized to store the records in the same GFU as a slice. Furthermore, the user has to specify the splitting policy, i.e., the minimum value and interval size of each index dimension and the aggregations that need to be pre-computed. On the other hand, when querying with DGFIndex, Hive first retrieves all related GFUKeys from the DGFIndex based on the predicates of the query, and retrieves all location information from HBase based on the GFUKeys. Then Hive filters out unrelated splits based on the location information. Third, each mapper can skip unrelated slices when processing each split based on the location information. However, for aggregation queries, Hive only needs to process the data located in the boundary GFUs (partially in the query region), then combine the sub result with the result that is computed via reading the precomputed aggregations from HBase.

TABLE 2
Comparison of Sorting Methods

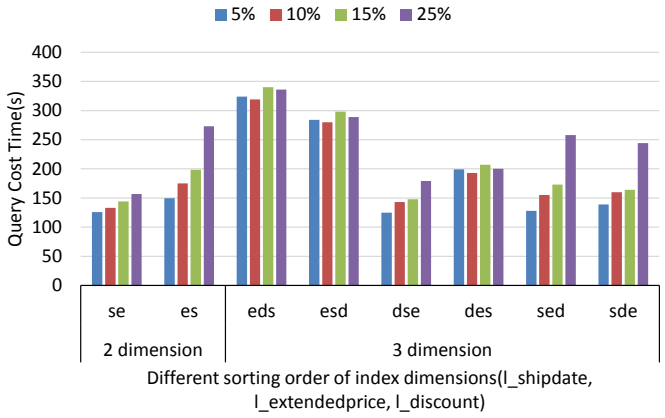
Method	STime(s)	Data actually read(MB)	QTime(s)
sort by	6890	7039	157
cluster by	7231	6878	156
order by	8366	1944	122

4 Preprocessing–Sorting

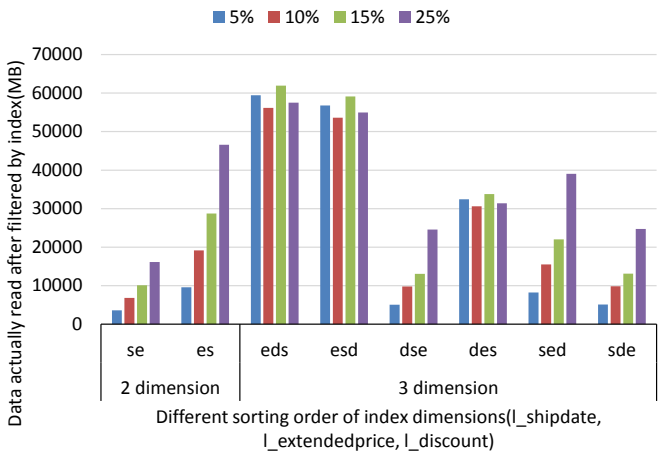
4.1 Sorting Methods

From the description in Section 3.2 and 3.3, the data filtering performance of Compact Index and the ORC file is closely related to the relative order of records in data files. In Compact Index, if query-related records are clustered in few splits, it will filter out more irrelevant splits, and because the index table does not need to record multiple locations for each combination of index dimensions, the index table will become smaller. Similarly, in ORC file, if the query-related records are clustered in few stripes and few row groups in these stripes, it will filter out more irrelevant stripes and row groups. Otherwise, if the query-related records are evenly distributed in data files, Compact Index and ORC file will not filter out much data. Thus, changing the relative order of records becomes crucial for improving the filtering performance of Compact Index and the index in ORC file.

In practice, sorting is a common and efficient method to change the relative order of records in Hive. Usually, users need to sort base tables by index dimensions beforehand. Currently, there are three kinds of sorting methods in Hive: *Sort By*, *Cluster By*, and *Order By*. In the sorting process, MapReduce first assigns each record to some reducer with specific *Partitioner*, then each reducer sorts all records that it receives based on the index dimensions: (1) *Sort By* assigns each record to reducers based on the hash value that is computed on the whole record. Records that have the same key may be distributed in multiple reducers. Therefore, Sort By can only guarantee the order of each reducer's output file, and it will not result in a global order. (2) *Cluster By* assigns each record to reducers based on the values of the index dimensions, so it can distribute the records that have the same key into the same reducer. However, it also does not result in a global order. Because it can not guarantee the order between the output file of reducers. (3) *Order By* uses the same way to assign records to reducers like *Cluster By*, and it can produce a global order. Besides, it has two implementations: the first one is serial *Order By*, it only uses one reducer, which is very inefficient on large input data and makes the time required for sorting unacceptable. The second one is parallel *Order By* [6], which first samples some data and decides the key range of each reducer to achieve a global order. From above description, if the index dimensions are highly skewed, some reducers in *Cluster By* and parallel *Order By* will receive too many records, whose performance will significantly deteriorate. However, *Sort By* still works fine for high skew data, because it just randomly assigns each record to reducer and the assignment is not dependent on the value of index dimensions.



(a) Query cost time



(b) Amount of bytes actually read after index

Fig. 3. Query cost time, and amount of bytes actually read after being filtered by index for different sorting order of index dimensions

Table 2 shows the cost time of different sorting methods to preprocess base table and the influence on the query performance, *STime* represents the cost time of sorting, *data actually read* represents the amount of data that actually read after being filtered by index, *QTime* represents the query processing time, we use parallel *Order By*. For fairness, we set the same number of reducers for all three methods. In this experiment, we use the cluster setup in Section 7.1 and Uniform500 data set in Section 7.2, which is stored as ORC file and Q6 of TPC-H. From the result, we can see that *Order By* has the best query performance, it is 22% faster than *Sort By* and *Cluster By*. Furthermore, it can filter much more data than other two methods. However, because *Order By* requires an extra sampling procedure before sorting the base table, it costs more time than other two methods. Because parallel *Order By* has the best query performance, we will use it to preprocess the RCFile (Compact Index) and the ORC file in our benchmarks of Section 7.

4.2 Sorting Order

In addition, the different order of index dimensions when sorting will lead to different relative order of records, which will also influence the filtering performance of indexes. Because the cardinality of index dimensions may vary, different sorting order will place records into different row groups, which causes different content of row groups. Figure 3 shows the influence of different sorting orders on the query execution time and filtering performance. We use the acronyms to represent the combination of *l_discount*(d, cardinality is 11), *l_shipdate*(s, cardinality is 2526), *l_extendedprice*(e, cardinality is 3791046). In this experiment, we use the cluster setup described in Section 7.1 and Uniform500 data set stored as ORC file and the queries with different number of index dimension and different selectivity in Section 7.2. From the results, we can see that *se* has the best query performance in 2-dimensional scenario, and *dse* has the best query performance in 3-dimensional scenario. That is, sorting the base table in the order of ascending cardinality will lead to better query efficiency. The reason is that putting the low-cardinality dimension to the front will lead to more clustered data file, so the index can filter more query-irrelevant data based on the min and max value. In our benchmarks of Section 7, we will use this guideline to preprocess base table for ORC file and RCFile (for Compact Index).

5 Improved DGFIndex

In this part, we describe two limitations of original DGFIndex based on the description in Section 3.4, and propose two corresponding improvements.

5.1 Storage Format

Original DGFIndex only supports TextFile format, which is a raw row storage format that is inefficient on encoding and decoding records compared to binary storage formats [17]. Moreover, it cannot decrease data I/O for queries that do not read all columns of a table. Columnar storage formats are efficient for OLAP query processing [36]. Currently, there are several kinds of columnar storage formats in Hive, for example, RCFile, ORC file, and Parquet. As an improvement to the original DFGIndex, we have migrated DGFIndex to RCFile and ORC file.

RCFile: RCFile is a widely used binary columnar storage format. To enhance the limitation of row storage formats, we have migrated DGFIndex to RCFile. In the original RCFile, the table is horizontally split into many row groups, the default size of a row group is 4MB. Each row group is stored as a *key/value* pair, the *value* part stores the real data in column-wise fashion, and the *key* part records the metadata of each column, for example, the number of rows, the length of each column and the length of each value in each column. The row group is the minimum reading unit in RCFile. We migrate DGFIndex to RCFile as follows: if the slice is smaller than row group, then we store it as a row group (may be smaller than 4MB), and DGFIndex records the start offset of this row group. On the other hand, if the slice is larger than row group, we store it as a sequence of row groups, and DGFIndex records the offset of the first row group and the last one.

ORC file: We use the reorganization method of DGFIndex to create a row group in each stripe of a ORC file, not like the original method (create a row group every 10,000 rows), which means all records in the same GFU are stored

as a row group. In this case, DGFIndex becomes very similar with the index of the ORC file, so we do not store the index in HBase any more and directly use the index of the ORC file to filter slices. Besides, the original index in ORC file on other dimensions are not influenced. The reorganization method of DGFIndex is a kind of multi-dimensional hashing method, which is different with the sorting preprocessing method of ORC file. This is because multi-dimensional hashing methods treat each index dimension equally, while sorting method needs to decide the importance of multiple index dimensions.

For DGFIndex users, it is hard to choose an optimal splitting policy based on the file storage format and the features of an application. According to our experiments, 32MB and 16MB are empirical better slice sizes for DGFIndex on RCFile and DGFIndex on ORC file. Therefore, a user only needs to specify the min value and interval size for each index dimension, and makes Equation 1 equals this slice size. We will use these two slice size in our experiments.

$$\prod_{i=1}^{\#dimension} \frac{max(d_i) - min(d_i)}{interval(d_i)} \quad (1)$$

5.2 Slice Placement Method

In the construction of the original DGFIndex, each mapper calculates the GFUKey for every record and the partitioner randomly assign each GFUKey to a reducer based on the default hash function of the MapReduce framework. In each reducer, the slices are sorted by the GFUKey. This can be seen as a random slice placement method. The random slice placement method may occupy too many mapper slots when processing a query and each mapper reads only few slices or frequently skips unrelated slices, which causes inefficient batch data reading. For example, as shown in Figure 4, the query-relevant slice set consists of three slices, two of them are placed in the first split, and the another one is placed in the second split. When processing the query, Hadoop will use two mappers to read the data, the first mapper reads *Slice 1*, then skips unrelated data and then read *Slice 2*, which does not achieve good sequential reading performance. To overcome this, we implement a z-order-based slice placement method. Z-order [30] is a kind of space filling curve, whose purpose is mapping multi-dimensional data to one-dimensional data while maintaining the data locality in the data space. In the partitioner, it divides the whole z-order value range into several ranges, where the number of ranges is same as the number of reducers. In each reducer, the slices are sorted by the z-order value of each slice. By doing this, we can significantly decrease the number of occupied mapper slots and increase the efficiency of batch data reading.

Table 3 shows the influence of different DGFIndex improvements on the number of mappers, the amount of data actually read after being filtered by index and query cost time. This experiment uses the cluster setup in Section 7.1 and Uniform500 data set in Section 7.2, and Q6 in TPC-H. From the results, We can see that when using the z-order based slice placement method, the number of mappers and the query processing time both decrease. The reason is that it puts slices in less splits, which are processed by less mappers and also improve the sequential reading efficiency of each mapper. When migrating DGFIndex to the RCFile and ORC file, which are both columnar storage, the number of bytes that need to read considerably decrease in comparison to the TextFile format because both can filter out query-irrelevant columns when reading each row group. Moreover, ORC file has an efficient data type-based encoding method, so it can further reduce the amount of data read compared to RCFile. For ORC file, discontinuous stripes in the same split will use individual mappers for processing, so the number of mappers is bigger than for TextFile and RCFile, even when using the z-order-based slice placement method.

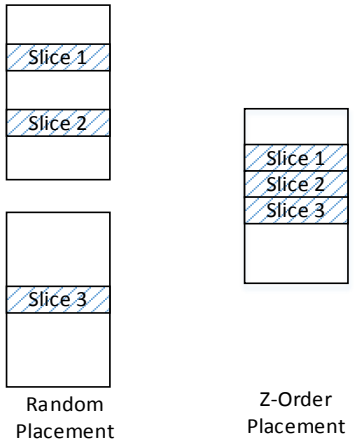


Fig. 4. Random placement and Z-order placement methods

TABLE 3
Performance of DGFIndex Optimizations

Item	#Mapper	#GB to read	#QTime(s)
TextFile	382	13.1	123.6
TextFile_zorder	95	13.1	107.3
RCFile	369	4.5	99.1
RCFile_zorder	91	4.5	79.7
ORC	1159	2.6	109.1
ORC_zorder	1163	2.6	93.6

6 Data Filtering Process Summary

In this part, we summarize the index and columnar storage based data filtering process in Hive, it is divided into 4 steps: (1) *searching the index*, (2) *filtering irrelevant splits*, (3) *filtering irrelevant row groups in each split*, (4) *filtering irrelevant columns*

TABLE 4
Summary of Index Size and Reading Method

Name	Size	Read
Compact	$\sum^{idx\ combinations} num(row\ groups) * c$	scan
ORC file	$(\frac{\lvert table\rvert}{\lvert stripe\rvert} * o_1 + \frac{\lvert table\rvert}{\lvert row\ group\rvert} * o_2) * colNum$	scan
DGFIndex	$\prod_{i=1}^{\#dimension} \frac{max(d_i) - min(d_i)}{interval(d_i)} * d$	key-based

in each row group. Besides, we discuss the potential factors in each step, which may influence the data filtering efficiency. At last, we explain reasons why the filtering process is the computation framework independent.

6.1 Searching the Index

Searching the index is formalized as Equation 2, that is, getting the offsets of query-related data(row groups or stripes). The performance of reading index is related with two factors: *the index size* and *reading method*, that is summarized in Table 4.

$$offsets_{query} = serach(index, predicate) \quad (2)$$

For Compact Index, the index size is proportional to the product of the number of index dimensions combinations and corresponding offsets of row groups. c is a constant representing the size of each index entry. For some table, the number of index dimensions combinations is fixed, and it is proportional to the cardinality of index dimensions. The number of row groups that some combination locate in depends on the layout of records in data files. Sorting can cluster these records with same combination in less row groups. Thus, sorting can decrease greatly the index size of Compact Index. The reading method of Compact Index is MapReduce-based full table scan.

For the index in ORC file, its index size is proportional to the table size and column number. Compared with Compact Index, its index size is not related with records order in data file, but the filtering performance is. Actually, the index in ORC file is 2-level index, first it records the min and max value for each column of stripe, which is coarse-grained. Second, in each stripe, it records the min and max value for each row group, which is fine-grained. In this step, ORC file only needs to read the first level index. o_1 and o_2 represent the size of index entry in different levels. The first level index reading is scan-based, and the second level index reading is also scan-based, but only read the index entry chosen by first level.

For DGFIndex, its index size is proportional to the number of GFU, which is decided by the splitting policy specified by user. Similar with the index in ORC file, its size is not related with the records orders in row group and the row groups order in data files. The reading method is key-based, only reading the query related GFUKeys.

6.2 Filtering Irrelevant Splits

After index reading, the data filtering process in Hive is showed in Figure 5. This step happens in the InputFormat.getSplit(). Filtering irrelevant splits is formalized as Equation 3. The number of chosen splits represents the parallelism of query processing and the computing resource(for example, the mapper slot) occupation.

$$Splits_{chosen} = getSplits(offsets_{query}, Splits_{all}) \quad (3)$$

For Compact Index, the number of chosen splits is related with records order in data file. We can find that sorting not only decreases the index size, but also reduces the number of chosen splits, thus sorting is crucial to improve the performance of Compact Index.

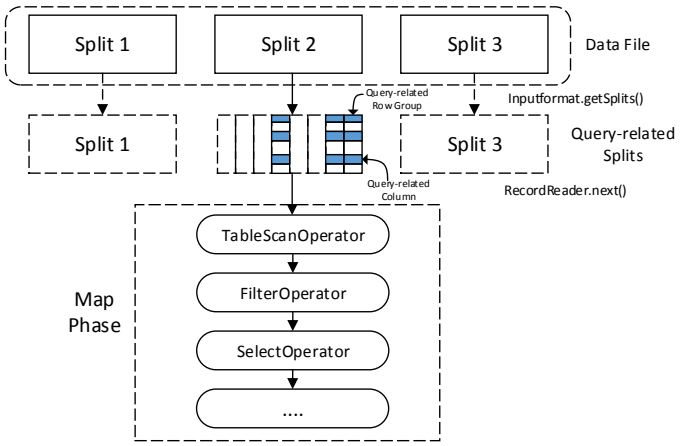


Fig. 5. index-based data flow

For ORC file, the number of chosen splits is also related with records order in data file. Besides, the adjacent query-related stripes in each HDFS block are seen as one split, the nonadjacent query related stripes in each HDFS block are seen as different split.

For DGFIndex, the number of chosen splits is not related with the records order in row group, but related with the row groups order in data file. That is, the query-related row groups are fixed, different row group placement methods will influence the number of chosen splits and the query-related data amount in each split.

6.3 Filtering Irrelevant Row Groups in Each Split

This step happens in the RecordReader.next() of Figure 5, it is formalized as Equation 4. This step helps Hive skip query-irrelevant row groups. The data filtering process of Compact Index does not include this step, because it only can filter data in split granularity.

$$row\ groups_{chosen} = next(offsets_{query}, row\ groups_{all}) \quad (4)$$

For the index in ORC file, it will read the second level index to get the offsets of query-related row group in each stripe. The number of chosen row row groups is related with records order in each stripe. We can find that sorting not only decrease the number of chosen splits, but also the number of chosen row groups. Thus, sorting is crucial for ORC file to improve the index performance.

For DGFIndex, it still uses the offsets in step 1 to get the chosen row groups. The number of chosen row groups is related with row groups order in data files. We can find that z-order based slice placement method can decrease the number of chosen splits and increase the number of chosen row groups in each split, but the total number of chosen row groups does not change.

6.4 Filtering Irrelevant Columns in Each Row Group

Index only affect the first three steps. In this step, columnar storage helps Hive only read the query-related columns in

each row group. Which is showed in Figure 5 and formalized as Equation 5.

$$columns_{query} = read(columns_{all}) \quad (5)$$

The cost of reading each column is related with the encoding method of different columnar storage, ORC file and Parquet use more encoding methods than RCFile, we expect that they will have better reading efficiency than RCFile.

6.5 Computation Framework Independence

The indexes in Hive currently only influence the single table reading process, the data reading-related operators in the Map phase are showed in Figure 5. The *TableScanOperator* is responsible for gathering statistics when needed, it has not effect in index-based data reading. The *FilterOperator* is responsible for filtering records with query predicate. The *SelectOperator* is responsible for selecting query-related columns. From the process, we can find that no matter which kind of index we use, the number of records after *FilterOperator* is the same. Moreover, for current SQL on Hadoop systems and SQL on Spark systems, they all use HDFS as the main underlying storage system, and they also use the same columnar storage(RCFile, ORC file and Parquet) and corresponding InputFormats with Hive. Thus, the conclusions and best practice are also applicable for these systems, they are computation framework independent.

7 Experimental Results

7.1 Cluster Setup

We conduct the experiments on a cluster of 32 virtual nodes, each one has 12 cores CPU, 26GB memory and 600GB disk. These virtual nodes are hosted on 8 physical servers. One node is used for the JobTracker, one for the Namenode and Hive, one for the HMaster, all other nodes are worker nodes of Hadoop and HBase. All nodes run CentOS 7.0, Java 1.7.0_65 64bit, Hadoop-1.2.1, and HBase-0.94.23 (stores the DGFIndex table). DGFIndex is implemented in Hive-0.14.0, we also use other indexes and columnar file formats in Hive-0.14.0. Every worker node in Hadoop is configured with 6 mappers and 2 reducer. The block size of HDFS is set to 256MB, which is consistent with the default block size of the ORC file. The *io.sort.mb* is set to 400MB, and *mapred.child.java.opts* is set to *-Dchild -Xmx1000m -Xmx 3000m*. All other configurations of Hadoop, HBase, and Hive are default values. Before each experiment, we free the OS cache to make each query read from disk, not from memory. Each experiment runs three times and the average result is reported.

7.2 Workloads

Datasets: In real applications, the distribution of index dimension may be uniform, with low skew, or with high skew. Thus, in our experiments, we use uniform [11] and skewed [37] TPC-H data generators to generate uniform data and skewed data. Skewed TPC-H can generate skewed data with *Zipf* distribution. We use the *lineitem* table which is the largest table in TPC-H. Table 5 shows the data sets

TABLE 5
Data Sets

Name	Size(GB)				SkewFactor
	TextFile	RCFile	ORC	Parquet	
Uniform250	263	244	149	135	0
Uniform500	529	488	299	270	0
LowSkew250	263	243	139	94	1
LowSkew500	529	483	276	188	1
HighSkew250	264	243	136	76	2

TABLE 6
Cardinality of Index Dimensions

Dimension(abbreviation)	Cardinality	Data Set
l_dicount(d)	11	all
l_quantity(q)	50	all
l_shipdate(s)	2,526	all
l_extendedprice(e)	82,561	HighSkew250
l_extendedprice(e)	3,790,000	others
l_orderkey(o)	525,000,000	250GB dataset
l_orderkey(o)	1,050,000,000	500GB dataset

that are used in the following experiments. The LowSkew data sets are generated by setting skew factor z=1 and the HighSkew data sets are generated by setting z=2. For example, the percentage of records with *l_discount* value 0.04 is 9% for Uniform data set, 13% for LowSkew data set, and 62% for HighSkew data set. RCFile and Parquet do not use any compression by default, while ORC file uses compression by default. For fairness, we do not enable the default compression of the ORC file. In addition, because the performance of Compact Index is related to the cardinality of index dimension, we summarize the cardinalities in Table 6, which also shows the abbreviation of each dimension name.

```
SELECT SUM(l_extendedprice*l_discount)
       AS revenue
FROM lineitem WHERE <predicate>
```

Listing 2. SQL template

Queries: The queries are shown in Table 7. The abbreviation in parentheses represent the query dimensions or index dimensions. For example, *2-d index(q,s)* means we create index on dimension *l_quantity* and *l_shipdate*, and *2-d query(q,s)* means the query predicate contains *l_quantity* and *l_shipdate*. The query identifier pattern is *Q-query dimensions-index dimensions*. These queries are designed on the consideration of two common query parttern: (1) the query dimensions are fixed, we can create index on these dimensions to improve query performance. That is, the query dimensions are same with index dimensions. The corresponding queries are the first four ones in Table 7, they are 1-dimensional, 2-dimensional and 3-dimensional query respectively. For 3-dimensional query, we add a high-cardinality dimensions query (Q-dse-dse) to show the its influence on index performance. (2) the query dimensions are always changing. That is, the query dimensions may only part of index dimensions. The corresponding queries are the rest queries in Table 7. As described in Section **??**, query different index dimension may have different index performance, so we query any one or two dimensions of the three index dimensions.

TABLE 7
Workloads

Identifier	Query Type	Index Type
Q-o-o	1-d query(o)	1-d index(o)
Q-qs-qs	2-d query(qs)	2-d index(qs)
Q-qds-qds	3-d query(qds)	3-d index-1(qds)
Q-dse-dse	3-d query(dse)	3-d index-2(dse)
Q-q-qds	1-d partial query(q)	3-d index-1(qds)
Q-d-qds	1-d partial query(d)	3-d index-1(qds)
Q-s-qds	1-d partial query(s)	3-d index-1(qds)
Q-e-dse	1-d partial query(e)	3-d index-2(dse)
Q-qd-qds	2-d partial query(qd)	3-d index-1(qds)
Q-qs-qds	2-d partial query(qs)	3-d index-1(qds)
Q-ds-qds	2-d partial query(ds)	3-d index-1(qds)

We design the queries with the SQL template(Q6 in TPC-H) in Listing 2 by changing the dimensions and the number of dimensions in *predicate*. Besides, we select point, 5%, 10%, 15% and 25% (the selectivity is an approximate value) query selectivity for each query type by altering the query dimension range in *predicate*. Because of the limitation of space, all detailed query forms are listed at the website given in Section 10.

The reasons of using Q6 and the *lineitem* table of uniform and skew TPC-H are as follows: First, the benchmark workload TPC-H is widely used. Second, after searching the index, Q6 can be processed by Hive in one MapReduce job, which is easy to analyze and compare the influence of indexes on query cost time, query-irrelevant data filtering performance. Besides, since the indexes in Hive only affect the data reading action of single table, even for complex queries that contain multiple tables joining, the indexes only work for each individual table, so it is sufficient to choose Q6, an aggregation query on a single table to benchmark the filtering performance of indexes in Hive. Third, Q6 is a typical representative of multi-dimensional queries in TPC-H.

In our experiments, the index benchmark candidates are: Compact Index on sorted RCFile, sorted ORC file, DGFIndex on RCFile, and DGFIndex on ORC file. The sorted RCFile and sorted ORC file mean the base table is preprocessed with the guidelines driven in Section 4. Besides, we use the z-order-based slice placement method for DGFIndex on RCFile and DGFIndex on ORC file. For fairness, we do not use the pre-computation technique of DGFIndex. In addition, because partition has similar effect on the different indexing techniques, we do not partition the base table.

7.3 Metrics

In practice, the data that needs to be analyzed is usually first stored as TextFile, for example, collected log files. Then, for efficient storage and query performance, the data is converted into other storage formats, and then preprocessed to create an index on it. In our experiments, we model similar data workflow: first we suppose that there is a *lineitem* table stored as TextFile, then we convert it into RCFile or ORC file with the SQL in Listing 3, which can sort the data in the meantime. Third, we create Compact Index or DGFIndex on it. At last, we run the various kinds of query on the base table. In the experiments, we mainly focus on the below metrics:

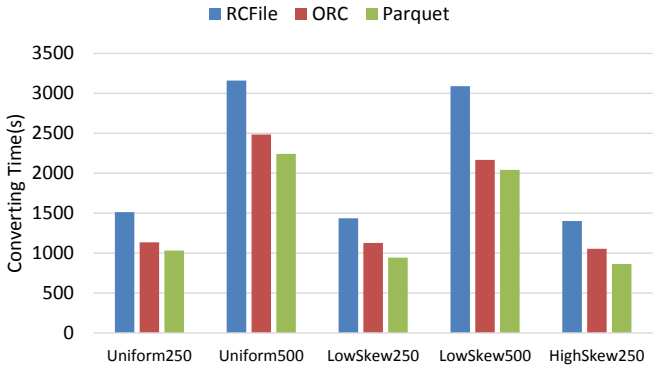


Fig. 6. Converting time from TextFile to other format

```
INSERT OVERWRITE TABLE lineitem_rc/orc
SELECT *
FROM lineitem_textfile
[ORDER BY <index dimensions>]
```

Listing 3. Storage format converting SQL

(1) **Index creation time** means the cost time to make base table ready to efficient query, including the time on storage format converting and creating index. For instance, for Compact Index on RCFile, the time includes the time of converting (sorting is also done in this step) *lineitem* table from TextFile into RCFile and the time of creating Compact Index. For sorted ORC file, the time only includes the time of converting *lineitem* table from TextFile into ORC file, because the sorting is done during the conversion.

(2) **Storage efficiency** includes base table storage efficiency and index table storage efficiency. Because different file storage formats have different data encoding methods, the table size will be different. Besides, Compact Index and DGFIndex store the index table as a separate table, the size of index table will affect the reading efficiency of index, so we also treat it as a metric.

(3) **Query efficiency** includes four metrics: (1) the time to process a query, which represents the elapsed time from emitting the query to getting the result. (2) The number of mappers used, which represents the computing resources usage, it may affect the concurrent throughput of the SQL on Hadoop systems. (3) The number of records actually read after being filtered by index, which represents the efficiency and accuracy of index. (4) The amount of bytes actually read after being filtered by index, which represents the encoding efficiency of corresponding storage file format and accuracy of index. We use the Counter *Map input records* and *HDFS_BYTES_READ* to get the value of (3) and (4).

7.4 Baseline

We use various columnar storage formats (RCFile, ORC file, Parquet) without index as the baseline to highlight the efficiency of index in the following parts.

Figure 6 shows the converting time from TextFile to RCFile, ORC file, and Parquet. From the results, we can see that Parquet has smallest file size as shown in Table 7 and it takes minimum converting time for different data sets. ORC file is 6%-22% slower than Parquet, RCFile is 27%-42%

slower than ORC file, and 40%-60% slower than Parquet. The query efficiency results of RCFile, ORC file, and Parquet will be shown and analyzed with formal index benchmark results in Section 7.7.

7.5 Index Creation Time

In this part, we analyze the index creation time of various indexes in Hive. The cost time of converting file format and creating index of Uniform250, Uniform500, LowSkew250 and LowSkew500 is shown in Figure 7. Because the result (index creation time and query efficiency) of HighSkew250 is very different from the other results, we will discuss it separately in Section 7.8. For 3-dimensional index *3-d index-2(dse)*, to avoid the extreme big index table of Compact Index, we only create 2-dimensional index on low-cardinality dimensions (*ds*). For example, the index size reaches to 150GB for Uniform250.

From the results, we can see that (1) for Compact Index, the index creation time increases with the increasing of cardinality, for example, for each data set, the cost time of 1-d index cost more time than others. Besides, the converting and sorting take the majority of time, because we limit the cardinality of the index dimensions, thus the index creation does not cost too much time. (2) For sorted ORC file, compared with original ORC, the sorting takes 1.6-3.4 times more time than converting, but it takes 16%-120% less time than DGFIndex. (3) For DGFIndex on RCFile and DGFInex on ORC file, they cost the maximum time among all indexes, and the index creating cost the majority time. Besides, compared with the results of Uniform250 and Uniform500, we can find that the index creation time of LowSkew250 and LowSkew500 cost more time for all indexes. For Compact Index on RCFile and sorted ORC file, the reason is that the parallel *Order By* relies on sample data to split the value range for each reducer. Since the data is skewed, some reducers may receive much more data than other reducers. As a result, these reducers will take more time to sort the received records, which will slow down the sorting process. DGFIndex assigns a value range for each reducer by evenly splitting the whole z-order value range, therefore, some reducer may receive much more slices than other reducers. In summary, sorted ORC file cost the least time for index creation, DGFIndex cost the most. The index creation time of Compact Index is acceptable for low-cardinality index dimensions. Moreover, the skew of index dimension will increase the index creation time.

7.6 Storage Efficiency

Table size. From Table 5, we can see that compared with RCFile and ORC file, Parquet has the best storage efficiency and it has the minimum file size for the same data set. The more skewed the data set is, the smaller the data size is. The reason is that when the data is more skewed, there are more repeated records and the encoding efficiency turns better, for example, the run length encoding technique.

Index size. Because the results of Uniform data and LowSkew data are similar, we only show the results of Uniform data sets, as shown in Figure 8. Since the index in ORC file is embedded in file, it is difficult to get its index size, we skip it here. As described in Section 6.1, the size of

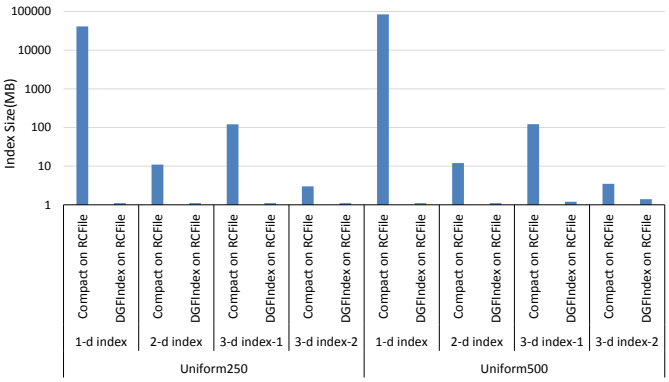


Fig. 8. Index size of Compact Index and DGFIndex

the Compact Index is proportional to the cardinality of index dimension and not related with data size. When creating 1-dimensional Compact Index on *l_orderkey*, the size of index table is 41GB (for Uniform250) and 82GB (for Uniform500). The reason is that the cardinality of *l_orderkey* of Uniform500 is two times of that of Uniform250. Besides, since the cardinality of other index dimensions does not change for different data sets, the index size of Compact Index does not change. In addition, we also find that after sorting the RCFile, the size of the Compact Index is significantly reduced. The reason is that before sorting, the records with same value are scattered in many row groups, therefore, the Compact Index needs to store all combinations of index dimensions and corresponding offset of row groups. After sorting, the records with the same value are clustered in less row groups, therefore, only few row group offsets need to be stored for each combination of index dimensions. For DGFIndex, it divides the high-cardinality dimension into many intervals, and its size is proportional to the number of intervals of index dimensions and, therefore, is much smaller than the Compact Index.

7.7 Query Efficiency

Because the results of Uniform250 and Uniform500, LowSkew250 and LowSkew500 have similar tendency, here we only analyze the results of Uniform500 and LowSkew500, the results of Uniform250 and LowSkew250 are shown in the web site given at the end of this paper. Figure 9 and Figure 10 show the results of Uniform500 and LowSkew500, including query cost time, mapper number that is used to process the query, number of records, and amount of data actually read after being filtered by index. The baseline results are represented with a dotted line, the results of various indexes are represented with solid lines. From the results, we can make the following observations:

For the baseline: RCFile, ORC file, and Parquet, on Uniform500, the query performance of ORC file is 1.5-2.6 times faster than RCFile, and Parquet is 30%-80% faster than ORC file. The reason is that Parquet has smaller data size than RCFile and ORC file for the same data set. The fluctuation of RCFile, ORC file and Parquet for different queries in Figure 9(d) shows the data size of different query-related column set. As shown in Figure 9(c)(d), for each query except *Q-o-o*, RCFile, ORC file and Parquet read all the

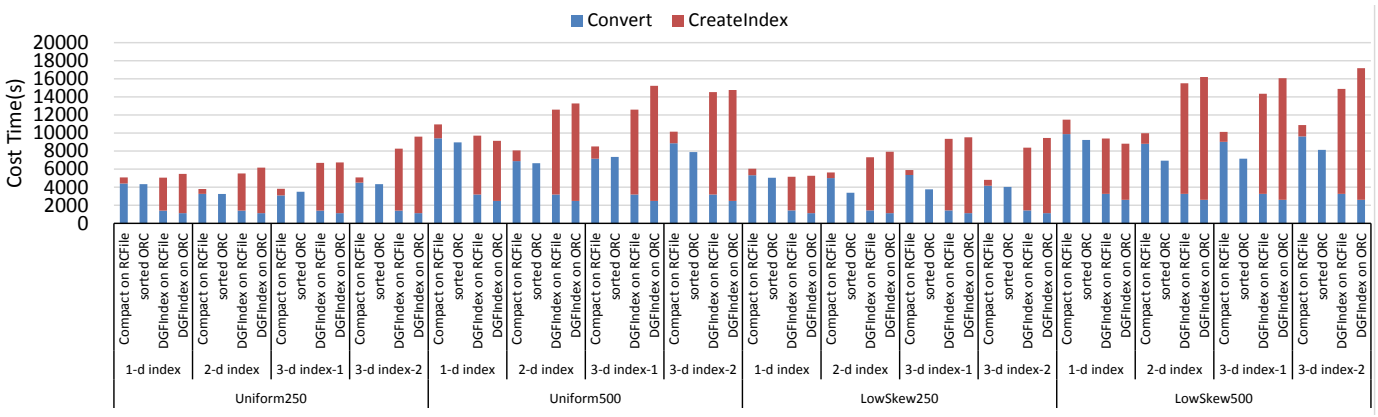


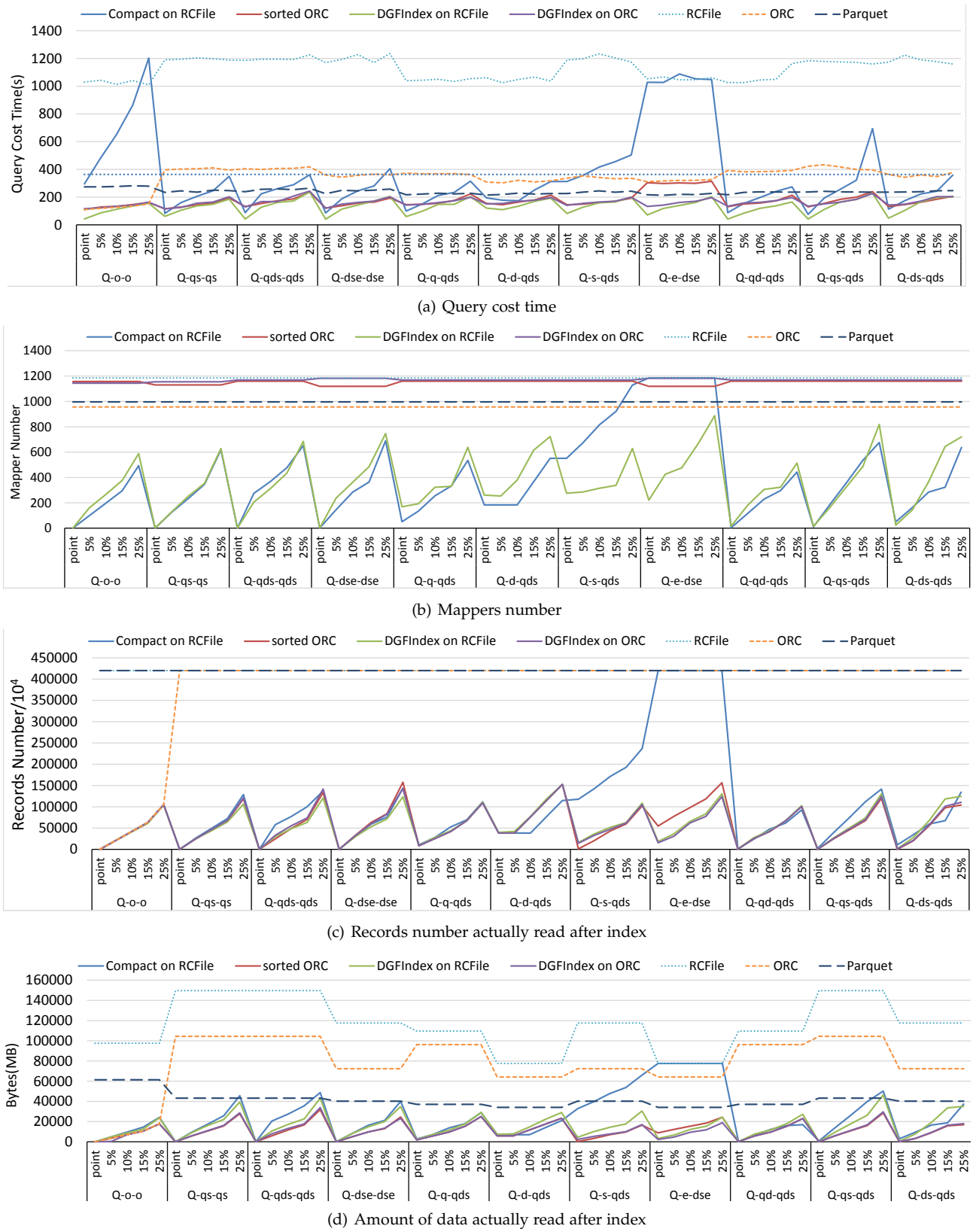
Fig. 7. Cost time of converting and creating index

records from base table, but they read different amount of data, which shows the encoding efficiency of corresponding columnar storage. Besides, as shown in Figure 9(c), the index in ORC file does not filter any records, since the query-related records evenly distributed in every row groups, in this situation, even only one record in a row group is query-related, the whole row group needs to be read. On LowSkew500, the query performance of ORC file is 2.3-3.8 times faster than RCFile, and Parquet is 37%-88% faster than ORC file. We can see that when the data becomes skewed, the query performance of ORC file and Parquet gets better. The reason for ORC file is that, for skew data, the records are more clustered than uniform data for some values, for example *l_discount*(0.04), thus ORC file can filter out more query-irrelevant row groups than Uniform500, which leads to the reduction of reading records and data size as shown in Figure 10(c)(d). For Parquet, the reason is that more repetitive records improve the efficiency of encoding method, for example, run-length encoding, which causes the data size of LowSkew500 is much smaller than Uniform500. Thus, compared with 9(d), Parquet read much less data than Uniform500, so the query performance is better. However, *Q-o-o* is an exception, in this case, ORC file is 0.8-1.5 times faster than Parquet on Uniform500, and 0.4-1 times on LowSkew500. Because *l_orderkey* is originally sorted in TPC-H, the index in ORC file can help it filter lots of query-irrelevant row groups. In summary, unless the data is original sorted by query dimensions, Parquet has the best query performance among all widely used columnar storage formats in Hive. Surprisingly, for LowSkew500 data set, Parquet's query performance is almost same with all indexes when the query selectivity is over 15%.

For Compact Index on RCFile, as shown in Figure 9(a) and 10(a), sorting the base table by index dimensions and creating proper Compact Index can improve query performance 2-13 times over the original RCFile. However, the limitation is that we only can create Compact Index on low-cardinality dimensions, otherwise, the large index table will reduce the query performance, like *Q-o-o* with 25% selectivity. For *Q-e-dse*, because we only create index on *l_discount* and *l_shipdate*, the query will not use the index. Among four kinds of indexes, Compact Index on RCFile has the worst query performance, the reason is that (1) Compact Index needs to start another MapReduce job to scan the index table, which costs more time than other indexes, especially when the index table is big. (2) Compact Index only can filter query-irrelevant data in split granularity, which will cause redundant data reading as shown in 9(d). However, Compact Index on RCFile uses much less mappers to process the same query than sorted ORC file and DGFIndex on ORC file as shown in 9(b) and 10(b). In addition, because the RCFile is sorted by the index dimensions, when querying the latter dimensions in the sorting order, data filtering performance of Compact Index becomes worse. For example, *Q-s-qds*, we sort RCFile in the order of increasing cardinality: *dqs*, which causes that the value of *l_shipdate* is more scattered than *l_discount* and *l_quantity*, causing more redundant data reading. In summary, if the data is stored as RCFile, and the index dimensions do not have large-cardinality, then Compact Index is suitable candidate index to improve multi-dimensional query performance.

For sorted ORC file, sorting can improve query performance 1-2.4 times over the original ORC file. In addition, because the ORC file can filter out data in row group granularity, it has better performance than Compact Index, especially for high-selectivity queries. The sorted ORC file has similar query performance as DGFIndex on ORC file, but *Q-e-dse* is an exception, for this query, the sorted ORC file's query performance is about 2 times worse than DGFIndex on RCFile. The reason is same with *Q-s-qds* of Compact Index on RCFile. But for DGFIndex on ORCFile, its data reorganization method is based on multi-dimensional hash, all index dimensions are treated equally, thus it will not have the problem like sorted ORC file. Furthermore, when the query selectivity is over 15%, sorted ORC file has similar query performance as DGFIndex on RCFile and DGFIndex on ORC file. However, ORC based indexes, for example ORC file, sorted ORC file, and DGFIndex on ORC file, use much more mappers than DGFIndex on RCFile and Compact Index on RCFile, which may influence the concurrent throughput of Hive. In summary, if the user does not care about the concurrent throughput, and the multi-dimensional query always has high selectivity, for example over 15%, then sorted ORC file is a suitable candidate index.

DGFIndex on RCFile has similar query performance with sorted ORC file, and better query performance when



(a) Query cost time

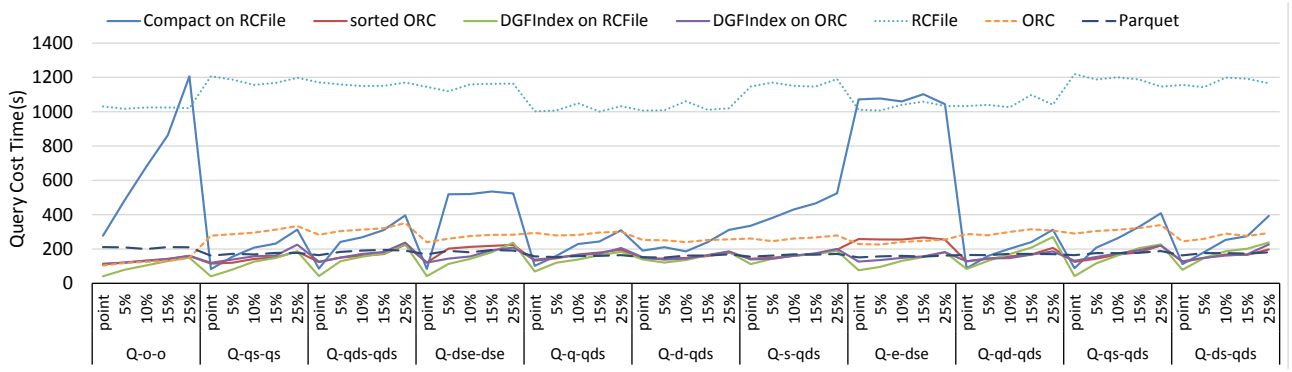
(b) Mappers number

(c) Records number actually read after index

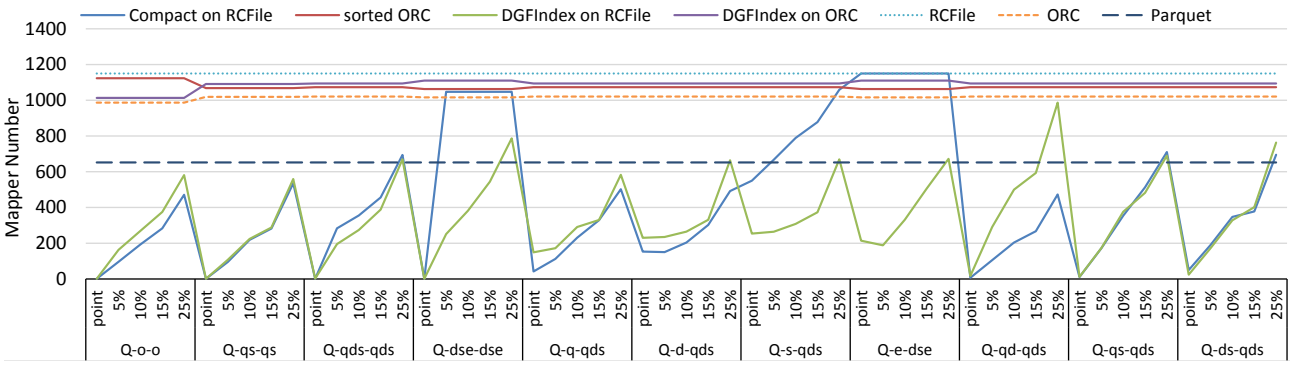
(d) Amount of data actually read after index

Fig. 9. Query cost time, Mapper numbers used to process query, records number and amount of data actually read after being filtered by index on Uniform500

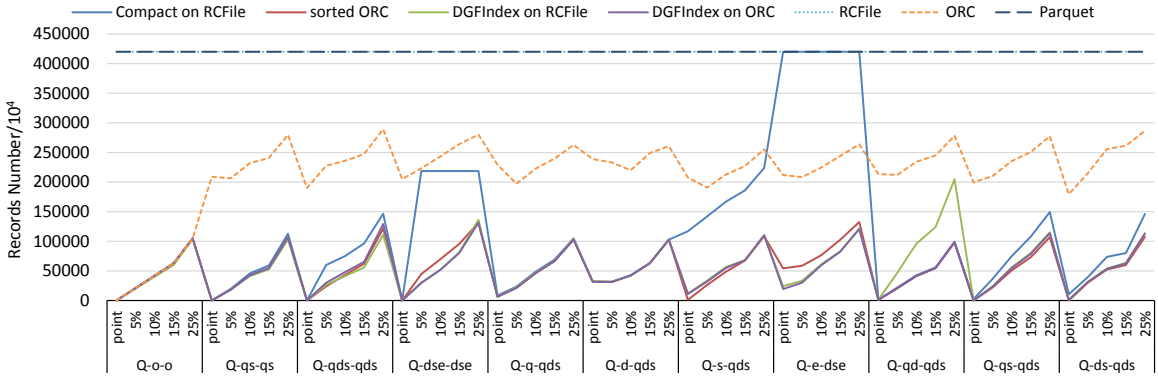
the query selectivity is equal or less than 15%. Besides, DGFIndex on RCFile occupies much less mapper slots than sorted ORC file and DGFIndex on ORC file, which can improve the concurrent throughput and computation resource utilization of clusters. We also find that although DGFIndex on RCFile reads more amount of data than sorted ORC file



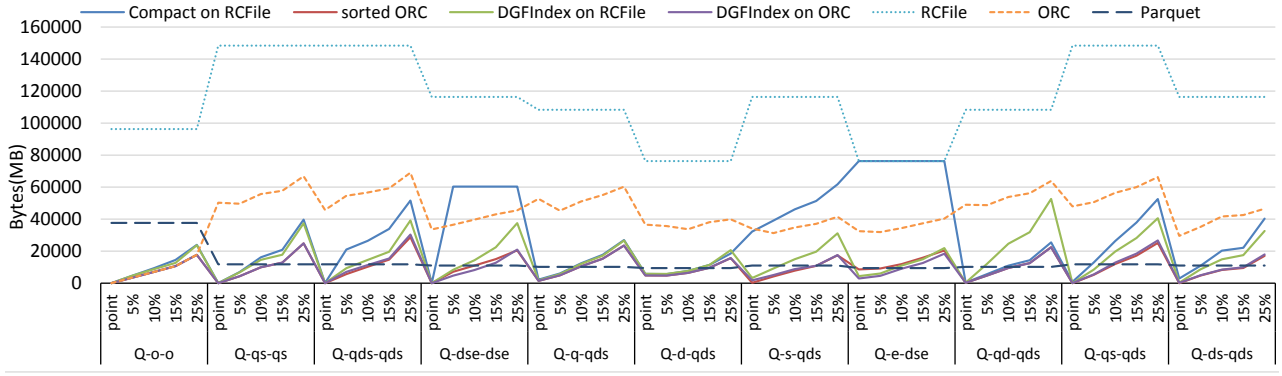
(a) Query cost time



(b) Mappers number



(c) Records number actually read after index



(d) Amount of data actually read after index

Fig. 10. Query cost time, Mapper numbers used to process query, records number and amount of data actually read after being filtered by index on LowSkew500

as shown in Figure 9(d) and Figure 10(d), it has better query performance. The reason is that processing sorted ORC file uses many mappers, which needs to run in multiple waves. In other hand, for DGFIndex on ORC file, it has almost the

same query performance with sorted ORC file, but it does not have the drawback of sorted ORC file, for example *Q-e-dse*. In summary, if the user cares about the concurrent throughput and computation resource utilization and the query selectivity is not very large, then DGFIndex on RCFile is a suitable candidate index. DGFIndex on ORC file is a better fit if the query selectivity is large and user is not sensitive to the concurrent throughput.

7.8 HighSkew250

As described in Section 4.1 and 5.2, *Cluster By*, parallel *Order By*, and the z-order-based slice placement method of DGFIndex all rely on the value of the index dimension to assign records to reducers. Thus, for the HighSkew250 data set, when sorting or creating DGFIndex on base table, few reducers will receive the majority of the data and the index creation time becomes unacceptably long. Therefore, DGFIndex on RCFile and DGFIndex on ORC file are not suitable for high skew data. For sorted ORC file and Compact Index on RCFile, we use *Sort By* to sort the base table. The query cost time is shown in Figure 11. Because Sort By can not make the base table as clustered as parallel Order By, the query performance of sorted ORC file is not better than original ORC file, but even worse. Although Compact Index still improves the query performance on RCFile 1-7 times, it is much worse than sorted ORC file, ORC file, and Parquet. Since Parquet has the minimum file size, its query performance is faster 10%-150% than ORC file and sorted ORC file. In Summary, for high skew data, original ORC file or Parquet is more suitable than other file formats and indexes, sorting does not improve the query performance.

8 Best Practice and Discussion

With our extensive summary and comprehensive experiments of current indexes and columnar storage formats in Hive, we suggest the best practice to improve the multi-dimensional indexes: (1) Choosing low-cardinality partition dimensions. Usually, date or enumeration dimension are good candidates, under the assumption that they do not generate too many partitions. (2) Choosing query dimensions always used by user as index dimensions, then follow the steps in Figure 12 to choose a suitable indexing technique: (a) whether users require good concurrent throughput or computation resource occupation ratio, if yes, the user needs to use DGFIndex on RCFile or Compact Index on RCFile. (b) Using *group by* to analyze data distribution, if it is high skew, Parquet is the best indexing technique. (c) Analyzing the user's queries, if the query selectivity is small, for example less than 15%, DGFIndex on RCFile is suitable. (d) At last, whether there is an index dimension with extremely large cardinality and corresponding partial query, if yes, DGFIndex on ORC file should be chosen, if no, sorted ORC file should be chosen. After choosing a suitable index based on the features of the application, the index performance can be optimized as follows: (1) for sorted ORC file and Compact Index on RCFile, the data needs to be sorted in the base table by the order of increasing cardinality of the index dimensions with *parallel Order By*. (2) for DGFIndex on RCFile and DGFIndex on ORC file, the z-order-based slice placement method should be used, and choose empirical 32MB and 16MB slice size respectively.

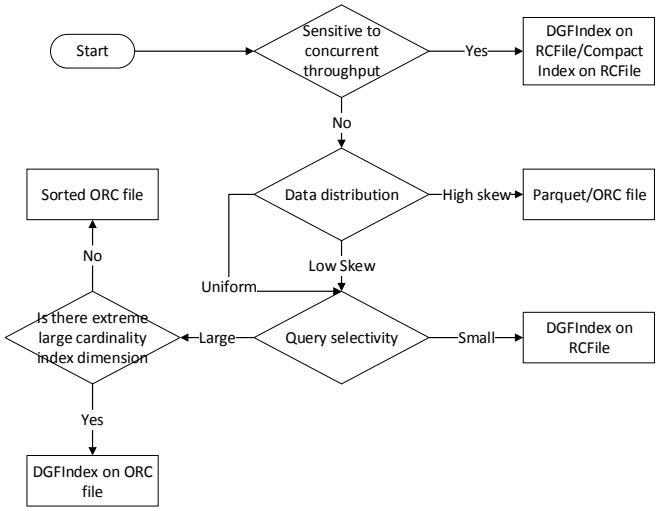


Fig. 12. Best practice of improving multi-dimensional query in Hive

From the experimental results, we find that each index or columnar storage format still has some limitations: Sorted ORC file occupies too many mapper slots, even if the query-relevant amount of data is small. The creation of DGFIndex costs too much time. Parquet still has no index support (the index is under development). A Compact Index' index table becomes too large for high cardinality dimensions. However, each index has its sweet spot and no one fits all use cases. Furthermore, based on our experiments, we summarize some important index design guidelines: (1) **Two-level indexes are more efficient than one-level indexes**. Two-level indexes mean first filtering data in coarse-grained level (split or stripe), then filtering data in fine-grained level (slice or row group), like DGFIndex and sorted ORC file. This way, the indexes can filter large amount of irrelevant data. (2) **Clustering frequently used records always is crucial to improve index performance**. Like sorting in sorted ORC file and reorganization in DGFIndex, storing the records that frequently are read together can greatly improve the sequential reading efficiency and reduce the amount of unnecessary data reads (3) **Considering data skew**. Less or high data skew may reduce the index performance or even fail the index design.

9 Related Work

In this paper, we analyzed various indexes on columnar storage, which has the advantage of eliminating query-irrelevant I/O. In recent years, many excellent columnar storage file formats have been proposed for Hadoop and Hive, such as CFile [28], Column Format [21], Record Columnar File [24], Optimized Record Columnar File (ORC file) [25] and Parquet [1]. They all provide a way to filter data vertically and can be improved with efficient index techniques.

In the context of index in Hadoop and Hive, HadoopDB [12] uses the index data engine of relational databases to accelerate the query processing on Hadoop,

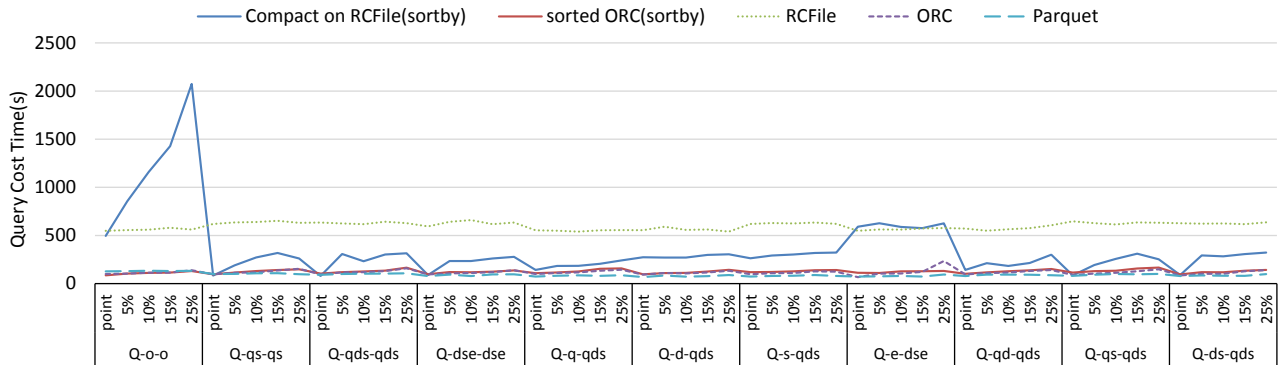


Fig. 11. Query Cost Time of HighSkew250

we have compare it with DGFIndex in [29]. [27] proposes a 1-dimensional index on sorted files of HDFS, each index entry records the location information of the corresponding data slice. [18] proposes the Trojan index and the Trojan join index, the Trojan index records the first key, end key, and the number of records in each split, and the Trojan join index is used to co-partition two tables that are joined and improves the join performance. [19] proposes a range index, that creates an inverted index for string type dimensions. [34] proposes a zero-overhead static and adaptive index for Hadoop. The above works either focus on one-dimensional indexing, or only can filter unrelated data in a coarse-grained manner. Also, they can not be used in Hive directly.

In the context of benchmark in Hadoop and Hive. [33] and [35] compare the performance of Hadoop and parallel relational databases. In recent years, the industry hopes to analyze big data using SQL in high speed, which has resulted in the development of many SQL on Hadoop systems, examples are Hive, Impala, Shark, Presto, and Tajo. [20] benchmarks the performance of Hive and Impala, and tests the efficiency of the range index in the ORC file on processing one-dimensional queries. The authors do not benchmark multi-dimensional query performance. [14] compares the query performance of several SQl on Hadoop systems. [22] compares the performance of MongoDB, SQL Server, Hive and parallel databases with TPC-H. All of the above work either does not test the efficiency of indexes on improving query processing performance, or only test the simple one-dimensional case.

In the context of index benchmark in spatial database and traditional RDBMS, [26] characterizes the spatial-temporal indexing problem and proposes a benchmark for the performance comparison of spatial indexes. [13] mainly focus on moving-object indexes, and propose a benchmark for the comparison of existing and future indexing techniques. [31] propose a performance benchmark for Location-Based Services(LBS) related dynamic spatial indexing. The three work mainly focus on spatial data processing, not traditional data processing like ours. [32] proposes a star schema benchmark for the index comparison of RDBMS, but no detailed comparison. [15] proposes several kinds of B-Tree index, but no experimental evaluations on them. Moreover, because the cost model and computation model of these work are different with Hive, thus user can not directly use these conclusions. As far as we know, our work is the first work on comparison of multi-dimensional indexes in Hadoop and Hive area.

10 Conclusion

In this paper, we benchmark the efficiency of various indexes and columnar storage formats in Hive on processing multi-dimensional queries. Based on the experimental results, we provide the best practices of improving multi-dimensional query processing based on the requirements of applications. Furthermore, we also summarize the important design guidelines of future index design in SQL on Hadoop systems.

The source code of DGFIndex, detailed query forms and the results of Uniform250 and LowSkew250 are available online[1].

References

[1] Apache parquet. http://parquet.apache.org/.
[2] Drill. https://drill.apache.org/.
[3] Hive aggregate index. https://issues.apache.org/jira/browse/HIVE-1694.
[4] Hive bitmap index. https://issues.apache.org/jira/browse/HIVE-1803.
[5] Hive compact index. https://issues.apache.org/jira/browse/HIVE-417.
[6] Hive parallel order by. https://issues.apache.org/jira/browse/HIVE-1402.
[7] Impala. http://www.cloudera.com/content/www/en-us/products/apache-hadoop/impala.html.
[8] Parquet pushdown. https://issues.apache.org/jira/browse/HIVE-10666.
[9] Pig. http://pig.apache.org/.
[10] Presto. https://prestodb.io/.
[11] Tpc-h. http://www.tpc.org/tpch/.
[12] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads. *Proceedings of the VLDB Endowment*, 2(1):922–933, 2009.
[13] S. Chen, C. S. Jensen, and D. Lin. A benchmark for evaluating moving object indexes. *Proceedings of the VLDB Endowment*, 1(2):1574–1585, 2008.
[14] Y. Chen, X. Qin, H. Bian, J. Chen, Z. Dong, X. Du, Y. Gao, D. Liu, J. Lu, and H. Zhang. A study of sql-on-hadoop systems. In *Big Data Benchmarks, Performance Optimization, and Emerging Hardware*, pages 154–166. Springer, 2014.
[15] D. Comer. Ubiquitous b-tree. *ACM Computing Surveys (CSUR)*, 11(2):121–137, 1979.

1. https://github.com/aimago

[16] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
[17] J. Dean and S. Ghemawat. Mapreduce: a flexible data processing tool. *Communications of the ACM*, 53(1):72–77, 2010.
[18] 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*, 3(1-2):515–529, 2010.
[19] M. Y. Eltabakh, F. Özcan, Y. Sismanis, P. J. Haas, H. Pirahesh, and J. Vondrak. Eagle-eyed elephant: split-oriented indexing in hadoop. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 89–100. ACM, 2013.
[20] A. Floratou, U. F. Minhas, and F. Ozcan. Sql-on-hadoop: Full circle back to shared-nothing database architectures. *Proceedings of the VLDB Endowment*, 7(12), 2014.
[21] A. Floratou, J. M. Patel, E. J. Shekita, and S. Tata. Column-oriented storage techniques for mapreduce. *Proceedings of the VLDB Endowment*, 4(7):419–429, 2011.
[22] A. Floratou, N. Teletia, D. J. DeWitt, J. M. Patel, and D. Zhang. Can the elephants handle the nosql onslaught? *Proceedings of the VLDB Endowment*, 5(12):1712–1723, 2012.
[23] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *ACM SIGOPS operating systems review*, volume 37, pages 29–43. ACM, 2003.
[24] Y. He, R. Lee, Y. Huai, Z. Shao, N. Jain, X. Zhang, and Z. Xu. Rcfile: A fast and space-efficient data placement structure in mapreduce-based warehouse systems. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 1199–1208. IEEE, 2011.
[25] Y. Huai, S. Ma, R. Lee, O. O'Malley, and X. Zhang. Understanding insights into the basic structure and essential issues of table placement methods in clusters. *Proceedings of the VLDB Endowment*, 6(14):1750–1761, 2013.
[26] C. S. Jensen, D. Tiešytė, and N. Tradišauskas. The cost benchmarkcomparison and evaluation of spatio-temporal indexes. In *Database Systems for Advanced Applications*, pages 125–140. Springer, 2006.
[27] D. Jiang, B. C. Ooi, L. Shi, and S. Wu. The performance of mapreduce: An in-depth study. *Proceedings of the VLDB Endowment*, 3(1-2):472–483, 2010.
[28] Y. Lin, D. Agrawal, C. Chen, B. C. Ooi, and S. Wu. Llama: leveraging columnar storage for scalable join processing in the mapreduce framework. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 961–972. ACM, 2011.
[29] Y. Liu, S. Hu, T. Rabl, W. Liu, H.-A. Jacobsen, K. Wu, J. Chen, and J. Li. DGFIndex for Smart Grid: Enhancing Hive with a Cost-Effective Multidimensional Range Index. *Proceedings of the VLDB Endowment*, 13(7):1496–1507, 2014.
[30] G. M. Morton. *A computer oriented geodetic data base and a new technique in file sequencing*. International Business Machines Company, 1966.
[31] J. Myllymaki and J. Kaufman. Dynamark: A benchmark for dynamic spatial indexing. In *Mobile data management*, pages 92–105. Springer, 2003.
[32] P. E. ONeil, E. J. ONeil, and X. Chen. The star schema benchmark (ssb). *Pat*, 2007.
[33] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 165–178. ACM, 2009.
[34] S. Richter, J.-A. Quiané-Ruiz, S. Schuh, and J. Dittrich. Towards zero-overhead static and adaptive indexing in hadoop. *The VLDB Journal*, 23(3):469–494, 2014.
[35] M. Stonebraker, D. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. Mapreduce and parallel dbmss: friends or foes? *Communications of the ACM*, 53(1):64–71, 2010.
[36] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, et al. C-store: a column-oriented dbms. In *Proceedings of the 31st international conference on Very large data bases*, pages 553–564. VLDB Endowment, 2005.
[37] V. N. Subhasis Chaudhuri. Program for tpc-d data generation with skew. ftp://ftp.research.microsoft.com/users/viveknar/TPCDSkew/.
[38] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. Hive-a petabyte scale data warehouse using hadoop. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 996–1005. IEEE, 2010.

Yue Liu received the BSc degree from College of Computer Science and Technology, Jilin University, Changchun, Jilin, China, in 2011. He is currently working toward the Phd degree with the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China. His research interests include distributed systems, database system, big data processing.

Shuai Guo received the BEng degree from the School of Computer Science and Engineering, Hebei University of Technology, Tianjin, China. He is currently working toward the Master degree with the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China. Hi research interests include distributed system and big data processing.

Songli Hu received his PhD degree from Beihang University, Beijing, China, in 2001. He was promoted as an associate professor in 2002 at the Institute of Computing Technology, and now work as a professor at the Institute of Information Engineering, both in Chinese Academy of Sciences, Beijing, China. His research interests include enterprise level big data processing, distributed system, service computing, etc.

Tilmann Rabl received Phd degree from the University of Passau, Germany, in 2011. He is Post-Doctoral Fellow of the Middleware Systems Research Group of University of Toronto, Canada. His research interests include benchmarking, big data processing and analytics.

Danyang Gu received the BEng degree from School of Software Technology, Dalian University of Technology, Dalian, Liaoning, China, in 2013. She is currently working toward the Master degree with Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China. Her research interests include distributed system, big data processing.

Yue Wang received the BEng degree from School of Software, Wuhan University, Wuhan, Hubei, China. He is currently working toward the Phd degree with the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China. His research interests include distributed systems, query optimization.

Hans-Arno Jacobsen received his M.A.Sc. degree from the University of Karlsruhe, Germany in 1994. He received his Ph.D. degree from Humboldt University, Berlin in 1999. He is a professor of University of Toronto, Canada. He has served as program committee member of various international conferences, including ICDCS, ICDE, Middleware, SIGMOD, OOPSLA and VLDB. His research interest include event processing, publish/subscribe, service-orientation, aspect-orientation, and green middleware.

Jintao Li received his PhD degree from the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China. He is a professor of the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China. His research interests include multimedia technology, virtual reality and ubiquitous computing etc.