Towards Query Performance Improvement in Big Data Environment

Jianbo Li[1], Jianghua Luo[2(✉)], Zhiwen Song[3], and Zhao Luo[3]

[1] Information Section of Chongqing Municipal Educational Examinations Authority, Department of Psychology of Southwest University, Chongqing, China
lijb@cqksy.cn

[2] Information Section of Chongqing Municipal Educational Examinations Authority, Chongqing, China
luojh@cqksy.cn

[3] College of Computer Science of Chongqing University of Posts and Telecommunications, Chongqing, China
{zwsong,zhaoluo}@cqupt.edu.cn

Abstract. In recent years, big data has exploded in the communications industry, especially in the 4G industry card data. However, the large amount of data and processing dimensions of 4G industry card data incur a large overhead of query performance. In order to improve its query efficiency, this paper studies the impact of data organization on query performance, and proposes a data organization framework suitable for 4G industry card data. The proposed framework integrates data organization patterns of Spark SQL and HBase + Phoenix, and can automatically select the appropriate data organization for different business types. The experimental results show that the proposed data organization framework improves the query efficiency by 35.35% on average.

Keywords: Query efficiency · Data organization · Spark SQL · HBase + Phoenix

1 Introduction

The big data in communication areas represented by 4G industry card data is exploding, which drives its processing technology transferring from traditional single- machine processing to distributed cluster processing. Although distributed frameworks such as Hadoop [1] and Spark [2] can significantly improve computing efficiency, there are also some limitations. On one hand, 4G industry card data is standard structured data, hence its business requirements are flexible and often require complex operations such as full table scans, row-by-row calculations, and a large number of multi-dimensional statistics and multi-table associations. On the other hand, its solutions such as Hadoop-based solutions typically use data warehousing [3] or NoSQL [4] to store data. Due to their distributed implementation nature, the support for OLAP and OLTP [5] is not as good as that of relational databases [6],i.e.,they are not fully compatible with CURD operations

Z. Zhan et al. (Eds.): ICBDIE 2022, AHCS 5, pp. 772–788, 2023.
https://doi.org/10.2991/978-94-6463-034-3_80

(Create, Update, Retrieve, Delete), and complex association operations (Union, Join, Group by, etc.). For instance, the Partition file based calculation process of Spark SQL [7] makes it difficult to modify or delete a single record; while HBase [8], the typical representative of NoSQL database, can quickly retrieve a single record, but is not good as Spark SQL in supporting complex association operations [9].

In order to improve the processing efficiency of 4G industry card data, this paper studies the impact of data organization structure on query performance, and proposes an integrated data organization framework, which improves the processing efficiency of 4G industry card data significantly, i.e., reducing the average processing time by 35.35%. The proposed framework is of great importance for improving processing performance of Spark SQL structured data, and provides reference for the processing of IoT data and 5G industry card data. The main contributions of this work are as follows.

- The Spark API for reading and writing Parquet files has been improved significantly based on inheritance, overwriting and reflection.
- By establishing the secondary index of Spark SQL file through Hbase + Phoenix, and combining the data of Spark and Hbase, the query speed of 4G industry card data is accelerated greatly.

The rest of the paper is organized as follows: Sect. 2 describes related work. Section 3 presents the data organization structure of 4G industry card, and shows the differences between Spark and Hbase when processing 4G industry card data both theoretically and experimentally. Section 4 analyzes the data organization of Spark SQL, compares the impact of different formats, and improves the Parquet read-write API. Section 5 analyzes the data organization of HBase and compares the effects of different formats. Section 6 integrates Spark SQL with Hbase to form an improved data organization framework. In Sect. 7, we conclude the paper.

2 Related Work

The Hadoop + Spark based solution usually builds a data warehouse on HDFS and queries it with Hive/Spark SQL.

However, due to the Partition file-based operation granularity of Spark SQL, the query speed is slow without index. Besides, a certain record can not be modified and deleted flexibly, and it is easy to have memory exception. Therefore, many researcher s are working on the improvement of storage methods, for a better query performance. E.g., columnar compression storage formats such as Parquet [10], ORC [11], and RCFile [12] have been confirmed to be effective in filtering unnecessary fields, compressing files and improving query efficiency.

However, when the amount of data to be processed is large for columnar compression storage format, the query is still slow due to the lack of primary keys and indexes in tables of Spark SQL. Spark SQL has provided file partitioning and bucketing strategy. Based on this, [13] proposed an adaptive data partitioning scheme to improve query efficiency. In addition, a set of OLAP schemes for big data in the power industry is proposed in [14], in which a fine-grained index structure (TrieIndex) based on prefix tree is established.

The above studies are based on the built-in data warehouse of Spark SQL, but the data warehouse does not support the update and delete operations of data as the database, since the data warehouse is only logical, not a physical database. As column-based physical database [15], HBase supports CURD operations better. But Hbase only has Rowkey-based indexes. When it comes to multi-dimensional complex associations, the efficiency is very low. To address this issue, a Solr-based Hbase massive data secondary index scheme is proposed in [16]. Besides, a secondary index is established for Hbase [8], such that the query efficiency can be improved.

Although Hbase has many APIs, it is necessary to use Phoenix [17] to perform simple queries using SQL statements. Complex queries are still not implemented. Therefore, the SQL compiler was rebuilt in [9], but its compatibility is not as good as the Catalyst parser built in Spark SQL [7].

Based on the above research, we consider the compressed storage format, index and NoSQL features, then combine Spark SQL and Hbase + Phoenix to store 4G industry card data. The proposed framework first obtains Spark SQL file information (e.g., block size, file location, offset, etc.) through Hbase, and then accurately retrieves specific records through the improved Parquet API.

3 4G industry Card Data Organization Analysis

The 4G industry card data covered in this paper is extracted from the mobile operator's Long Term Evolution (LTE) network architecture. The LTE network architecture is shown in Fig. 1, where UE, MME and HSS represent user equipment, mobility management entity, and user home subscription server, respectively. This paper focuses on the modules of UE, HSS, MME and their signaling data s1-mme, S6a, S11, etc.

The 4G industry card data is closely related to the Internet of Things, and has certain reference value for the processing of 5G industry card data. Based on 4G industry card

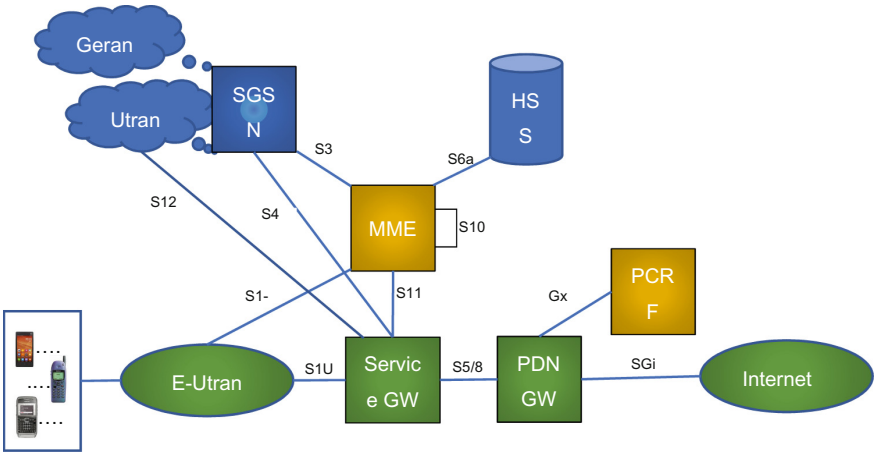


Fig. 1. LTE Network architecture.

data, we theoretically analyze the difference between Spark SQL and HBase in the following paragraphs.

The operational granularity of Spark SQL is Partition file. Suppose a table has a total of n records, m Partition files, and each Partition file stores k rows of data, then we can get the following formula.

$$m = \left\lceil \frac{n}{k} \right\rceil \tag{1}$$

where n, m, k are variables. When n is constant, m is inversely proportional to k. We first analyze the case where n is constant.

When the size of a file reaches 128 MB (the default maximum of HDFS), the file can store up to k' rows of data (k' varies with different compression algorithms). i.e., when $k' = k_{\max}$, $m = m_{\min}$.If we want to find a specific record from n records, assume the lookup parallelism to m', then the query will take m/m' rounds, that is, searching from m' files at the same time in each round until this particular record is found in a file. As a result, the worst case query time complexity satisfies:

$$O(n) = O(\left\lceil \frac{m}{m'} \right\rceil * k) \tag{2}$$

1. When m' < m, m/m' rounds off queries are required at worst, and the time complexity is as shown in formula (2);
2. When $m' >= m$, only need to query m Partition files in one round, and the time complexity is $O(k)$;

It is not difficult to see that when $m' = m_{\max}$ is used for query, resources has the least impact on query performance. If $m' = m$ is always guaranteed, k becomes a factor that affects query performance. Therefore, this paper focuses on reducing $O(k)$.

When k = 1, then m = $m_{\max}$, and the time complexity is O (1). But when the amount of data is very large, such as when $n \to \infty$, $m \to \infty$, we cannot guarantee that the limited cluster resources m = m', then the query time t is proportional to m/m'.

When $k = k'$, then $m = m_{\min}$ and the cluster resources (m') can satisfy m files in parallel, with its time complexity being $O(k)$. Thus when k is large, the query time t is proportional to k. We reduce the $O(k)$ by accurately retrieving data through the improved Parquet API.

The above analysis is in the case that n is fixed, but the number of data rows of the table will increase continuously in practical applications. Even if k is maximized, m will continue to grow and be much larger than the parallelism m', resulting that the query bottleneck lies in O(m/m'). Hence we need to reduce m/m' by merging small files. Affected by this property, the query time of Spark SQL lies in seconds or even minutes, which is hard to achieve real-time statistical analysis.

Unlike Spark SQL based on file queries, HBase is based on a single record. Suppose a table has n records and n rowkeys that are arranged in order. If the data is divided into m regions and each region stores k records. The query request goes through zookeeper, the ROOT table and the META table in order, and then locate the data in one of regions after three requests with the time complexity of each request being $O(1)$. In the region, some of data is cached in memory, and the extra is written to disk.

Assuming that each region divides the k rows data into y blocks equally. Each block stores x records. Then we have x = k/y or y = k/x. There is one data block in the memory, and y-1 data blocks in the disk. The data of each block is organized according to the tree structure, and all the data blocks together form a large tree structure.

If the query hits the cache, then the time complexity is O(log x). Otherwise, Hbase needs to continue to query other data blocks. The time complexity of finding the data block is O(log (y -1)). Thus the total time complexity is:

$$O(n) = O(log\ (k/x - 1)) + O(log\ x) \tag{3}$$

According to the above analysis, the query time overhead of Hbase is less than Spark SQL. Since Hbase is suitable for simple random queries and Spark SQL is suitable for complex offline analysis, we combine their advantages, that is, we use Hbase to build an index for Spark SQL.

In order to observe the difference between Spark SQL and Hbase more intuitively, we further compares their query performance through an experiment. The experiment estab lishes a Partition (coarse-grained index) for the Spark SQL table to improve the retrieval speed, and also establishes a secondary index table for Hbase by means of Phoenix. The experiment uses the S6a interface signaling table(int s6a) in the 4G industry card data. The table has 34 fields and about 143 million rows of data, which is partitioned according to their time in minute and stored in Parquet format. At the same time, the table is imported into Hbase and stored in LZ4 format. The experimental results are shown in Fig. 2 (a).It is observed in Fig. 2 (a) that Spark SQL partition queries are faster than no partitions, and Hbase queries are faster than Spark SQL partition queries. The time for "get" operations in Hbase are in milliseconds. But in HBase, "count" operation is the slowest, and more complex "max" and "sum" operations are not supported. At the same time, the time for Phoenix to query primary key and index fields is also in milliseconds, which is much faster than querying non-primary and non-indexed fields.

This shows that Hbase + Phoenix is significantly better than Spark SQL for simple queries. But for complex aggregation operations such as count(), sum(), max(), and "group by", etc., it is not as good as Spark SQL. So neither of them are compatible with CURD operations and complex association operations independently. We will next analyze Spark SQL and Hbase respectively in this paper.

4 Spark SQL Data Organization

Spark SQL is a module that deal with structured data in Spark. It inherits from the Hive data warehouse and is basically compatible with Hive syntax. A Spark SQL statement will eventually generate an RDD to execute. The RDD loads the corresponding Partition file to complete the calculation. In essence, the Spark SQL operate the Partition file.

From the perspective of HDFS, the data warehouse is a directory (e.g., /user/spark/warehouse), where the database is its first level subdirectory (e.g., /user/spark/warehouse/lte.db), and the table is the first level subdirectory of the database directory (e.g., /user/spark/warehouse/lte.db/tb s6a). The real data of the table, the Partition files, however are located in the table directory. If the amount of data in the table is large, the data will be split into multiple Partition files to store. But when the number of

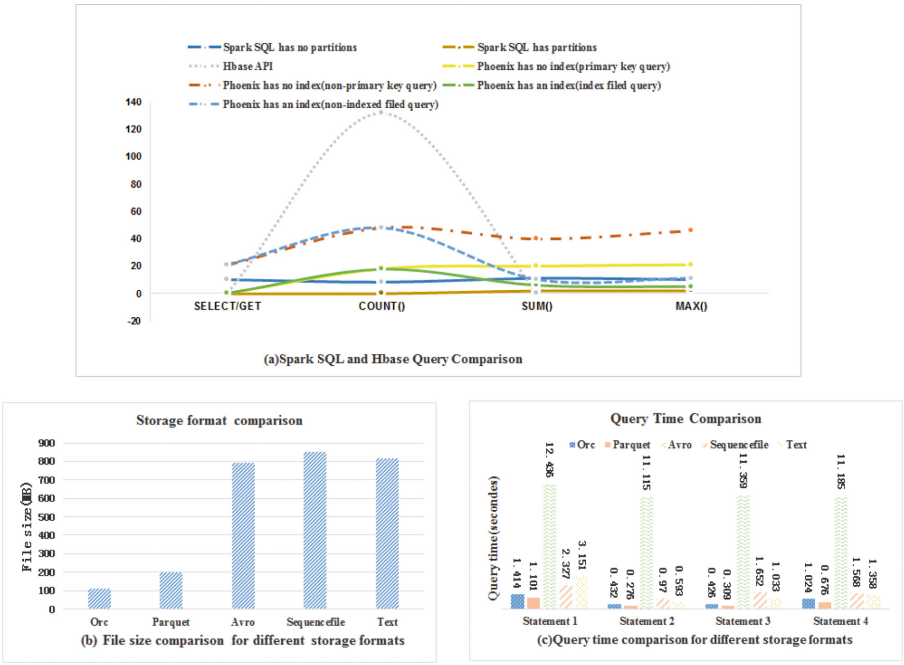


Fig. 2. The experiments of Spark SQL.

files reaches a certain level, the query speed will be seriously affected, as it is possible to trigger a full table scan. In this case, a subdirectory under the table directory is needed for the storage of multiple partition files, which is created as follows:

$$/user/spark/warehouse/lte.db/tb\ s6a/time = 202101$$
$$/user/spark/warehouse/lte.db/tb\ s6a/time = 202102$$
$$/user/spark/warehouse/lte.db/tb\ s6a/time = 202103$$

This layered directory structure plus metadata forms a logical data warehouse. The Spark SQL statement will eventually manipulate the files in the above directory so that Spark SQL can essentially be seen as building a layer of SQL mapping for the files. The data size of a table is defined as:

$$F = \sum_{i=0}^{m} f_i \tag{4}$$

In formula (4), m represents the number of Partition files, f_i represents the size of the i^{th} file. It can be seen that the key to the data organization of the data warehouse is the Partition file itself. In order to compress the file size and improve the query speed, we can change the format of the Partition file, such as Text, SequenceFile [5], Avro [7], Parquet, ORC, etc., and can also cooperate with compression algorithms, such as GZIP, LZO, SNAPPY, LZIP, etc. In addition, there are two main ways to organize files: row-oriented

storage and column-oriented storage. Text, SequenceFile, Avro, are all stored in a row format, while Parquet, ORC, are all stored in a column format. Row-oriented storage incurs larger read-write overhead, compared to column-oriented storage.

In order to present the impact on the performance brought by different storage formats, we select 4 different file formats for comparison. Two of them are SequenceFile and Avro in row-oriented storage and the other two are Parquet and ORC in column-oriented storage. We compare them to the default Text format. The compressed data size is shown in Fig. 2 (b).

As can be seen from Fig. 2(b), compared to the default text format (Text), the ORC file size is the smallest and the Parquet file size is the second. In order to further test the query performance of the above five storage formats, we select four SQL statements as shown in Table 1 to conduct experiments, and the experimental results are shown in Fig. 2(c).

It is observed in Fig. 2(c) that the Parquet format gains the fastest query speed for the four different SQL statements, followed by the ORC format. The query time of Avro, SequenceFile and Text stored in rows is much slower than Parquet and ORC that stored in columns. In addition, as can be seen from Fig. 2(b), although ORC saves more space than Parquet, the query time of Parquet is faster than ORC. This is because Parquet files are generated by different tools with different performance. The performance of Parquet files generated by Spark SQL is optimal [10], which means that ORC format performs better on Hive, and Parquet format performs better on Spark SQL. Therefore, we will focus on the structure of Parquet and its impact on query performance. Although Parquet's comprehensive performance is optimal, its read and write speed is seriously limited to Spark's DataFrameReader and DataFrameWriter APIs. We analyze the source code of the Parquet file, and then override the Parquet Read-API and Parquet Write-API by inheritance.

The Parquet format is a nestable structure. We take the characteristics of the 4G industry card table (tb_imsi_tel) as an example, as shown in Table 2.

The "info" of Table 2 represents a composite field, which contains two sub-contents. The schema information of a nested structure is as follows:

```
message tb_imsi_tel
{required int64 IMSI;
repeated int64 TelNum;
repeated group info
{optional binary CreateTime (UTF8);
optional binary APN (UTF8);}}
```

The binary and UTF8 in the Parquet format represent the string type, and the int64 represents the long type. If the schema is converted into a tree, the table name *tb_imsi_tel* is the root node, *IMSI*, *TelNum*, *CreateTime*, *APN* are its leaf nodes, and *info* is the branch node. That is, the basic types (e.g., int, long, string, etc.) are used as leaf nodes, which store values, and the composite type (group) is used as the branch node, which does not contain values.

Table 1. Test SQL statement

Sql1	with tab a as (select from unixtime(cast(startdate/1000000 as int),"yyyy-MM-dd") as etl_date,city as city_id,sum(1) as http_req_times,sum(case when code > 0 and code < 400 then 1 else 0 end) as http_rsp_succ_times,sum(case when code <= 0 or code >= 400 then 1 else 0 end) as http_fail_times,sum(case when code > 0 and code < 400 then responsetime else 0 end) as http_rsp_succdelay,sum(responsetime) as first_rsp_delay from tb_sque where p_app = 5 and apptype = 5 group by from_unixtime(cast(startdate / 1000000 as int), "yyyy-MM-dd"), city) select a.etl_date,a.city_id,case when http_req_times > 0 then round(http_rsp_succ_times/http_req_times*100, 2) else 0 end as http_rsp_succ_rate,case when http_rsp_succ_times > 0 then round(first_rsp_succdelay / http_rsp_succ_times / 1000, 2) else 0 end as http_rsp_succdelay,http_req_times as total http_req_times,http_rsp_succ_times as total http_rsp_succ_times,first_rsp_succdelay as total_first_rsp_succdelay,first_rsp_delay as total_first_rsp_delay from tab a a;
Sql2	select count(1) from tb squre;
Sql3	select responsetime,windowsize,version from tb_squre where startdate >= 1515628800000000 and enddate <= 1515718800000000;
Sql4	select apn,count(1) from tb squre group by apn

Table 2. Tb imsi tel table information

Column name	Datatype	Description
IMSI	int64	International Mobile Subscriber Identification Number
TelNum	int64	User phone number
info.CreateTime	string	Creation time
info.APN	string	Bind APN name

"required" indicates that the field appears once, "repeated" indicates 0 or more times, and "optional" indicates that the field appears 0 or 1 times. The basic type of data, the leaf node, also contains three attributes: (Repetition level, Definition level, value). When traversing the tree, we need to rely on Repetition level and Definition level to get the value. As the field is defined in the leaf node, when traversing the path of a field from the

root node, the depth when a node is empty (undefined) is usually taken as the Definition level of the field. The repetition level indicates at which depth the field is repeated.

Spark SQL will read these meta information to organize the data. However, DataFrameReader and DataFrameWriter have very slow read and write speed due to their own defects. Therefore, we first uses the reflection technique to construct the schema information of the table dynamically. Then we rewrite ParquetReader and ParquetReader API. Finally, the improved API is called by multithreading to read and write Parquet files in parallel.

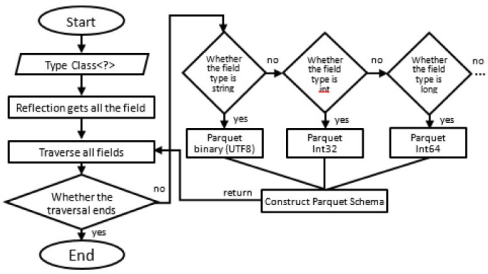
Regarding the unity and scalability of API, the structural definitions of Read API and Write API are consistent in this paper. They share reflection modules *bean2Schema()* and *reflectBean()*. In order to make it easier to distinguish, we stipulate that custom classes inherit from the original API class with the prefix "My". For example, MyParquetReader inherits from ParquetReader and follows the original inheritance relationship. The main steps to improve Read API are as follows:

1. Customize a class *MyParquetReader.class*, which inherits from *ParquetReader.class*. We construct objects by using builder mode(*Builder*) to create *ReadSupport*;
2. Customize a inner class *MyBuilder* of *MyParquetReader.class*, which inherits from *ParquetReader. Builder*;
3. Customize a class *MyParquetReadSupport.class*, which inherits from *ParquetReadSupport.class*. We use this class to receive and parse schema;
4. Create the schema by reflection. *MessageType schema = bean2Schema(T.class)*;
5. Create the handle of *ParquetReader*. *ParquetReader < String[] > pqReader = new MyBuilder(new Path(inPath),schema).build()*;
6. Traverse the source file and turn each of the original records into an array of strings. *String[] array = reflectBean(stu)*;
7. Use the handle of *ParquetReader* to read every line: *pqReader.read(array)*.

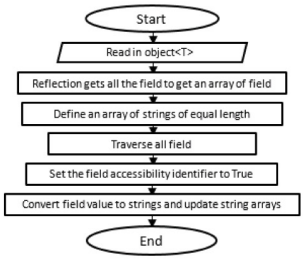
The main steps of the improved Write API in this paper are as follows:

1. Customize a class *MyParquetWriter.class*, which inherits from *ParquetWriter.class*.We construct objects by using builder mode(*Builder*) to create *ReadSupport*;
2. Customize a inner class *MyBuilder* of *MyParquetWriter*, which inherits from *ParquetWriter. Builder*;
3. Customize a class *MyParquetWriteSupport.class*, which inherits from *ParquetWriteSupport.class*. We use this class to receive and parse schema;
4. Create the schema by reflection. *MessageType schema = bean2Schema(T.class)*;
5. Create the handle of *ParquetWriter*. *ParquetWriter < String[] > pqReader = new MyBuilder(new Path(inPath),schema).build()*;
6. Traverse the source file and turn each of the original records into an array of strings. *String[] array = reflectBean(stu)*;
7. Use the handle of *ParquetWriter* to write every line into disk: *pqWriter.write(array)*.

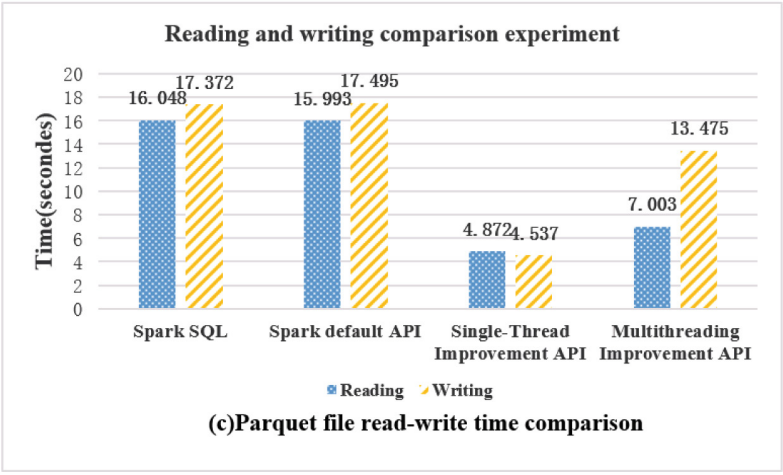
Among them, the key code that creates *MessageType* automatically by using reflection technology of JavaBean is encapsulated in method *bean2Schema()*. Its flow chart is shown in Fig. 3(a).



(a) Reflection creates a Message Type schema



(b) Reflection get object value



(c)Parquet file read-write time comparison

Fig. 3. The experiments of improved API.

In addition, the key code to convert the object into a string is encapsulated in the method *reflectBean()*, and its flowchart is shown in Fig. 3(b).

In order to verify the performance of the improved API in this paper, we compare it with the default method of Spark. As is shown in Fig. 3(c), the results show that no matter which method is used, writing data is slower than reading data. Besides, the improved API in this paper is faster than Spark's default API in both writing and reading data.

The improved read-write API will be used in the data organization framework such that it can ensure the fast read and write speed of large-scale 4G industry card data storage.

5 HBASE Data Organization

As a physical database, the table of Hbase has both logical model and physical model, which is different from the logical Spark SQL. Hbase's data organization is based on the LSM tree structure, where data is located by key-value pairs, that is, by using the combination key $< rowkey, columnfamily, column\text{-}name, timestamp >$, the unique value Cell can be located. A logical view of its data is shown in Table 3.

Table 3. Logical view of the Hbase table

RowKey	Timestamp	cf1		cf2	
		name	age	class	addr
rowkey1	t3	Bob	23		
	t2	Bob		English	
	t1	Bob			Beijing
rowkey1	t5	Nick		Math	
	t4	Nick			

The data shown in Table 3 looks like a nested Json string from Json's perspective:

{ rowkey1: – –

[cf1:name:Bob,age:23,

cf2:class: English,addr: Beijing],

rowkey2:[cf1:name: Nick, cf2:class:Math]}

In order to improve the query performance of Hbase, we first synchronously associate Phoenix with Hbase such that both Hbase API and Phoenix API point to the same table. On the basis of synchronous correlation, then we use Phoenix to index the non-Rowkey fields of Hbase table in order to greatly improve the speed of random query. Take the table *tb imsi tel* in Table 2 as an example, the relationship between the two structures is illustrated as follows:

1. We first create the table *tb imsi tels* in Phoenix, which is displayed as table *TB IMSI TEL* (capitalized by default). Thereafter, Hbase automatically associates the table. The structure of the table is as follows:

```
create table tb imsi tel (
IMSI varchar not null primary key,
TelNum varchar,
CreateTime varchar, APN varchar );
```

2. Then we look up the table *TB IMSI TEL* from Hbase shell. The structure of the table is shown in Figure4(a).

From Fig. 4(a), we can see that the primary key *IMSI* of table *tb imsi tel* in Phoenix corresponds to the default name *ROW* of *RowKey* in HBase. The default column cluster

```
ROW                          COLUMN+CELL
 460004512345678             column=0:\x00\x00\x00\x00, timestamp=1551966721787, value=x
 460004512345678             column=0:\x80\x0B, timestamp=1551966721787, value=13512345678
 460004512345678             column=0:\x80\x0C, timestamp=1551966721787, value=2017/5/1 10:05:34
 460004512345678             column=0:\x80\x0D, timestamp=1551966721787, value=testApn.gz
1 row(s) in 0.0100 seconds
```

(a) TB_IMSI_TEL structure in Hbase

```
ROW                          COLUMN+CELL
 460004512345678             column=cf:APN, timestamp=1551967905183, value=testApn.gz
 460004512345678             column=cf:CreateTime, timestamp=1551967904095, value=2017/5/1 10:05:34
 460004512345678             column=cf:TelNum, timestamp=1551967904078, value=13512345678
 460004512345678             column=cf:_0, timestamp=1551967905183, value=
1 row(s) in 0.0480 seconds
```

(b) Reasonable structure of tb_imsi_tel

Fig. 4. The structure of tb imsi tel.

is 0, the colon is followed by the column name in hexadecimal, and the *value* indicates the value of the column. That is to say, Hbase's RowKey corresponds to the primary key of Phoenix, and *column cluster: column name* corresponds to the column of Phoenix.

However, this association is not intuitive, so we should first create the table *tb imsi tel* in Hbase and then create the same named table *tb imsi tel* in Phoenix (table names and fields should be double quoted). The advantage is that the columns in the Hbase table are created manually and correspond to that of Phoenix table one by one. The data structure of the table *tb imsi tel* in Hbase is shown in Fig. 4(b).

Then we create the associated table *tb imsi tel* in Phoenix with the following structure:

```
create table "tb_imsi_tel" (
  IMSI varchar not null primary key,
  "cf"."TelNum" varchar,
  "cf"."CreateTime" varchar,
  "cf"."APN" varchar );
```

By this way, the corresponding column in Hbase is no longer a transcoded hexadecimal character, but a manually specified column name. Therefore, when we write the program, we can get the specified column name directly from the Hbase native API.

Then we test how the table files in Hbase are organized. Hbase does not compress data blocks by default (that is, the default storage format is set to NONE). In order to compress the storage space and improve the query speed, alternative compressed formats such as GZ, LZ4, LZO, SNAPPY can be used. For simplicity, we choose easy-to-use GZ, LZ4 in our experiment, and compare them with the default NONE format in terms of file size and query speed. Two different scale tests are extracted from the same table, which are 1 million rows of data and 10 million rows of data, respectively. The experimental results are shown in Fig. 5(a) and Fig. 5(b).

As shown in Fig. 5(a), GZ compression saves the most space compared to NONE, and LZ4 compression is in the middle.

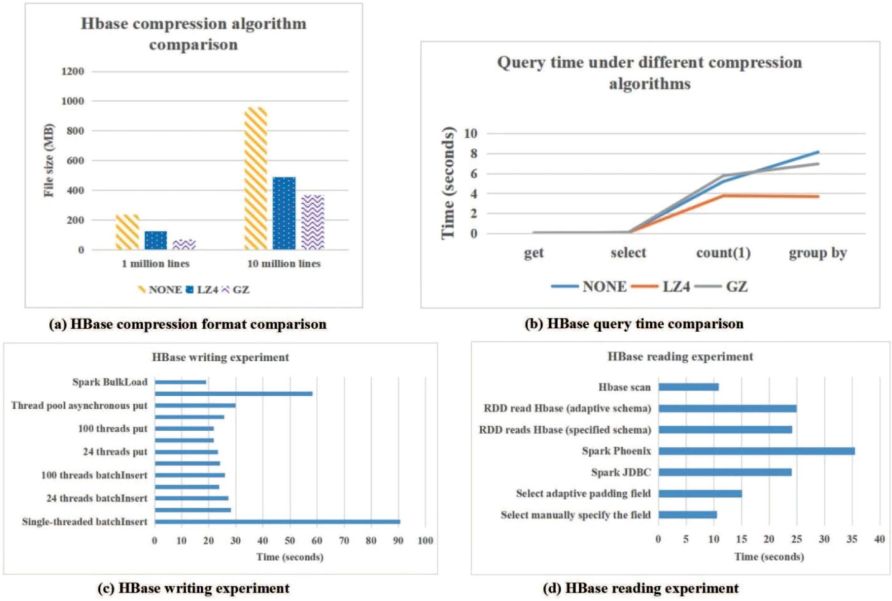


Fig. 5. The experiments of HBase.

In Fig. 5(b), the query time for LZ4 is the fastest, while the query time for GZ with high compression ratio is relatively slow. As a result, in order to balance storage space and query time, we choose the LZ4 format to store the tables in Hbase.

To further verify the performance of LZ4 format, the reading and writing experiment of LZ4 file is carried out. For the Hbase writing test, we test different import schemes for 1 million pieces of data, such as single-thread batch insertion, multi-thread Put, thread pool Put, MapReduce import and Spark import. The experimental results are shown in Fig. 5(c).

It is observed in Fig. 5(c) that there is little difference in write time between using Phoenix SQL insertion and using Hbase API for Put in the case of multithreading and single- threading. 50 to 100 threads have the fastest write speed and single thread has the slowest write speed. In addition, the experimental results show that the import speed of MapReduce is poor while Spark BulkLoad is the fastest. This is because Spark skips the Hbase API and directly generates the required data file (HFile).

For the Hbase reading experiment, we test different reading methods under 1 million pieces of data based on LZ4 format, such as Phoenix SQL read, Spark JDBC read, Spark phoenix read, Hbase to RDD mode, Hbase scan mode. The experimental results are shown in Fig. 5(d).

As shown in Fig. 5(d), using Phoenix SQL to read the specified field is the fastest, as well as Hbase scan. There is little difference between Spark JDBC and RDD, and the slowest is Spark Phoenix.

As Hbase has the advantage of data retrieval (in milliseconds), we combine Hbase with Phoenix to create an index for Spark SQL to improve the query speed of Spark SQL, i.e., for each record of Spark SQL, store the metadata information such as file path,

block location, size, etc., in Hbase, and set up the associated table and secondary index through Phoenix. This will be shown in the next section.

6 Improved Data Organization Framework

We integrate Spark SQL and HBase to form an overall data organization framework. As shown in Fig. 6(a), the framework combines Hbase with Phoenix to build an index for Spark SQL data, and then retrieve the data accurately through the improved Parquet API, which greatly improves the processing speed of the data.

As shown in Fig. 6(a), the data organization framework is as follows. First of all, we use Spark to collect the original signaling data from the server. After ETL and compression, we store it in the data warehouse of HDFS. When writing to the data warehouse, we synchronize the data with Hbase, including the file path, block location, offset, size and other necessary information of each data. Then we set up the associated table and secondary index through Phoenix.

When the foreground command retrieves the data, we first locate the data in the data warehouse through Phoenix, and then load the corresponding data block file to obtain the needed data. The main steps are as follows:

We associate the table structure of Spark SQL and H- base + Phoenix to realize the mapping relationship;

When traversing the data, we extract the parquet file information of the Spark SQL block file and return the file name, size, file location and offset corresponding to the data;

Get the query field to which the data will be written and combine it with the parquet file information;

Use improved Parquet API to write data and file information into Spark SQL and Hbase respectively;

Use Phoenix to establish a secondary index for multiple query fields, so as to speed up the retrieval speed.

The improved data organization scheme in this paper mainly integrates the random query advantage of Hbase with the statistical analysis ability of Spark SQL. We store the file information of Spark SQL in Hbase, and use Phoenix to speed up retrieval. In order to verify the effectiveness of the proposed scheme, we select 500 thousand rows of data from the S6a table to carry out the experiment. The experimental results are shown in Fig. 6(b).

We can see that the proposed scheme greatly improves the speed of data retrieval. We compare precise search with fuzzy query. The results show that there is little difference in query speed before the improvement, but with the proposed scheme, the query speed is obviously improved, especially for the precise search.

We use Hbase and Phoenix to schedule Parquet le information. Since the random query in Hbase is in milliseconds, the time cost of locating a Partition file is also in milliseconds. Then we use the improved Parquet interface to load the data. With the improved Spark interface, the worst-case time complexity of reading a Partition file on a specific node and looking up qualified data in that file is just O(n), where n represents the number of records in a Partition file. Assuming that a table has k Partition files, the

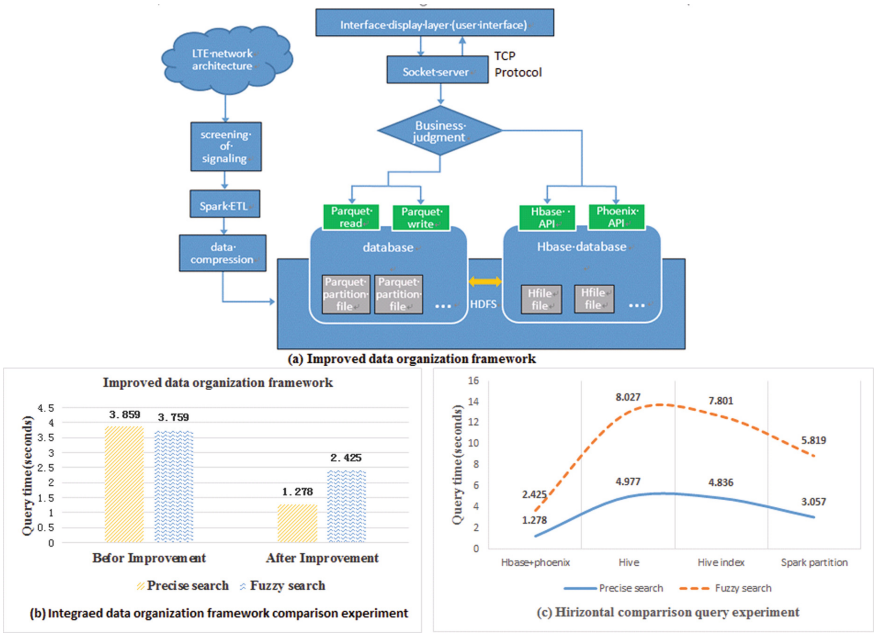


Fig. 6. The experiments of improved data organization framework.

maximum time overhead under the Spark default query mechanism is O(n*k). That is, the integrated data organization framework reduces the time cost to $\frac{1}{k}$ of the original.

In addition, we compare the proposed data organization framework with Hive, Hive index and Spark partition, where Hive is the mainstream offline processing engine. Because Spark SQL and Hive are mostly compatible, and the schema information of the two is exactly the same, it is easy to migrate 4G industry card data to the Hive warehouse. The experimental results are shown in Fig. 6(c).

It is observed that the proposed data organization framework has the shortest query time. Hive's query time is relatively long due to its MapReduce engine – when the hive database uses an index query, it first looks for the offset of the field in the index table, and then retrieves the data from the original table, among which process, the size of HDFS Read is about 6.874 GB and the size of HDFS Write is about 12.0313 GB. But the original data after compression is only 98.6 MB, which shows the great advantages of our single threaded and multithreaded API.

In addition, since data distribution in Spark may span multiple partitions, the partition retrieval process may trigger a full table scan during a fuzzy query. If the retrieval data is concentrated in a partition, the retrieval speed will be further accelerated. To sum up, the improved data organization framework in this paper has certain advantages in reading, writing and querying data.

7 Conclusion

The different data organization methods have a large impact on query performance in big data environment. We analyze and test the data organization of Spark SQL and Hbase respectively. In view of the various needs of 4G industry card data business, such as full table scanning, line-by-line calculation, a large number of multi-dimensional statistics, multi-table association, etc., we propose an integrated data organization scheme. The scheme fully considers the data storage format and improves the read, write and index methods, which greatly accelerate the processing speed of 4G industry card data.

In order to further improve the speed and compress the storage space, we will optimize the performance of proposed data organization framework in future.

Acknowledgment. This paper is supported by the following foundations or programs, including National Social Science Foundation of China (No. 17XFX013), Demonstration Projects of Technology Innovation and Application of Chongqing Science and Technology Committee (No. Cstc2018jscx-msybX0332), the Science and Technology Research Program of Chongqing Municipal Education Commission (Grant No. KJQN201900641), the State Key Laboratory of Computer Architecture Research Fund (CARCH201902).

References

1. M. R. Ghazi and D. Gangodkar, “Hadoop, mapreduce and hdfs: a developers perspective,” Procedia Computer Science, vol. 48, pp. 45–50, 2015.
2. S. Salloum, R. Dautov, X. Chen, P. X. Peng, and J. Z. Huang, “Big data analytics on apache spark,” International Journal of Data Science and Analytics, vol. 1, no. 3-4, pp. 145–164, 2016.
3. V. Garg, “Optimization of multiple queries for big data with apache hadoop/hive,” in 2015 International Conference on Computational Intelligence and Communication Networks (CICN). IEEE, 2015, pp. 938–941.
4. Z. Yu-Jie and Y. U. Shuang-Yuan, “Overview on big data query,”
5. Computer and Modernization, pp. 82–88, 2017.
6. H. Lang, T. Mu¨hlbauer, F. Funke, P. A. Boncz, T. Neumann, and
7. Kemper, “Data blocks: Hybrid oltp and olap on compressed storage using both vectorization and compilation,” in Proceedings of the 2016 International Conference on Management of Data. ACM, 2016, pp. 311–326.
8. H. Q. Zhang, “Relational database and nosql database,” Computer Knowledge & Technology, pp. 4802–4804, 2011.
9. M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley,
10. X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi et al., “Spark sql: Relational data processing in spark,” in Proceedings of the 2015 ACM SIGMOD international conference on management of data. ACM, 2015, pp. 1383–1394.
11. M. Zhu and Z. Wang, “Query optimization of large data based on hbase,”
12. Intelligent Computer & Applications, pp. 59–61, 2017.
13. X. Chen, R. Zoun, E. Schallehn, S. Mantha, K. Rapuru, and G. Saake,
14. “Exploring spark-sql-based entity resolution using the persistence capability,” in International Conference: Beyond Databases, Architectures and Structures. Springer, 2018, pp. 3–17.

15. X. Li and W. Zhou, "Performance comparison of hive, impala and spark sql," in 2015 7th International Conference on Intelligent Human Machine Systems and Cybernetics, vol. 1. IEEE, 2015, pp. 418–423.
16. K. Rattanaopas, S. Kaewkeerat, and Y. Chuchuen, "A comparison of orc-compress performance with big data workload on virtualization," in Applied Mechanics and Materials, vol. 855. Trans Tech Publ, 2017, pp. 153–158.
17. J. Wang, H. Duan, G. Min, G. Ying, and S. Zheng, "Goldfish: In-memory massive parallel processing sql engine based on columnar store," in 2016 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData). IEEE, 2016, pp. 142–149.
18. C. Guo, Z. Wu, Z. He, and X. S. Wang, "An adaptive data partitioning scheme for accelerating exploratory spark sql queries," in International Conference on Database Systems for Advanced Applications. Springer, 2017, pp. 114–128.
19. Y. Wang, Y. Liu, J. Hong, W. Cui, L. I. Yanhu, S. U. Yipeng, G. Huang,
20. M. Zhang, and W. Liu, "Spark/shark-based olap system for smart grid applications," Journal of University of Science & Technology of China, pp. 66–75, 2016.
21. Siddiqa, A. Karim, and A. Gani, "Big data storage technologies: a survey," Frontiers of Information Technology & Electronic Engineering, vol. 18, no. 8, pp. 1040–1070, 2017.
22. W. Wang, X. Chen, H. Wang, W. U. Xiaosong, and S. University, "A secondary index scheme of big data in hbase based on solr," Netinfo Security, pp. 39–44, 2017.
23. S. Akhtar and R. Magham, "Using phoenix," in Pro Apache Phoenix. Springer, 2017, pp. 15–35.