IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received November 28, 2019, accepted December 18, 2019, date of publication December 23, 2019,
date of current version January 2, 2020.

Digital Object Identifier 10.1109/ACCESS.2019.2961692

A Hive-Based Retrieval Optimization Scheme for Long-Term Storage of Massive Call Detail Records

XI PENG[1,2], LIANG LIU[2], AND LEI ZHANG[2]
[1]Graduate Faculty, China Academy of Telecommunication Technology, Beijing 100191, China
[2]College of Cybersecurity, Sichuan University, Chengdu 610065, China

Corresponding author: Lei Zhang (zhanglei2018@scu.edu.cn)

This work was supported by the National Key Technology Research and Development Program of China under Grant 2017YFB0802900.

ABSTRACT With the dramatic rise of mobile internet users and the administrative requirements of long-term data retention, telecom providers are facing increasingly challenging storage and retrieval issues of call detail records (CDRs). The existing storage system can only achieve the requirement of online query and offline analysis of the CDRs. However, to the best of our knowledge, few studies have focused on the topic of CDRs retrieval optimization with long-term storage. In order to improve the retrieval speed while ensuring a high compression ratio, in this paper we propose a novel hash storage scheme, termed dual-column bucketing (DCB), based on the Hive platform by making use of its Bucketing nature. Compared to the conventional scheme, the proposed DCB scheme can improve the performance both for CDRs compression and query. Second, similar storage scenarios such as storage of SMS, email and extended detail records (XDRs) are included in the optimization scope of the DCB. Experiments on real-world CDRs show that in contrast to the conventional scheme, the proposed DCB scheme can save the storage space by approximately 40%, reduces the amount of disk read to 2%, and improve the retrieval speed of known phone number queries by up to seven times.

INDEX TERMS Bucketing, call detail records, hash storage, long-term storage.

I. INTRODUCTION

Nowadays, the mobile communication network has become an indispensable part of people's daily life. Faced with billions of users, communication operators are forced to store large amounts of call detail records (CDRs) for a long period to meet business needs and national regulatory requirements. It is estimated that 150KB raw data is generated per subscriber in China per year. In November 2018, statistics on the number of subscribers of operators in China showed that China Mobile had more than 900 million subscribers, China Unicom and China Telecom had more than 300 million subscribers of each. If the billing information is required for storage for 6 months, a single operator of these three mentioned above must store 20PB data at least. Yet, according to the Law of the People's Republic of China on Network Security, half a year's data is the minimum requirement in the regulations. Internationally, the EU and Australia have already adopted similar regulations to retain CDR data several years ago [1]. This is a matter of national security.

The associate editor coordinating the review of this manuscript and approving it for publication was Xin Luo.

Although 20PB data may appear small by today's standards in memory and storage capacity, they are absolute minimum estimates of raw data. Protection of the CDRs such as encryption and random access to it should be considered because the privacy information of the service users is hidden in it. Both features can bring storage overhead and it may be tenfold from extrapolating previous experiences [2]. The above estimation may still appear small by comparison to a large business platform. However, the CDR retention system is a non-profitable investment for the operator so we cannot expect it to use significant resources in terms of purchased hardware and software. Besides, the improvement of retrieval speed is also demanded. Therefore, to increase the compression ratio and improve retrieval speed on PB-level CDRs simultaneously is a big challenge.

A. EXISTING PROBLEMS

At present, there are a variety of big data analysis techniques [3], [4]. Among them, Hadoop is an open-source software framework for distributed storage and processing for huge data sets on computer clusters built on cheap

hardware [5]. Hive [6] and HBase [7] are two major big data stores based on Hadoop and many research results have been achieved.

First, about optimizing Hadoop storage system implementation, the authors of [8] proposed a compression strategy based on HBase, using different compression algorithms for cold and hot data on TPC-H data set, which achieves a compression rate of up to 28%. The authors of [9] designed the secondary index of HBase to improve query performance on generated log data. In [10], the authors proposed PageFile, a hybrid page-based storage structure on the MapReduce framework. It has faster query processing, better disk space utility compared to Hive's RCFile [11] on the TPC-H data set. The researchers of [12]–[14] focused on content or record level of big textual data analysis that give respectively 52.4% average data size reduction compared to Huffman algorithm on real-world data sets such as Amazon movie review and food review, up to 72% on analysis performance and nearly Bzip compressor compression ratios by two-level compression on real-world data sets such as Google Server Logs and Yahoo Music Rating, and 24% improvement on analysis performance with up to 75% data size reduction by making splittable compressed content on real-world data sets such as Wikipedia Article Abstract and Genome Sequence.

Although researchers mentioned above contribute greatly from file-level optimizations including compression algorithm, the structure of file format, and record analysis to improve storage efficiency and retrieval performance on popular data sets, they do not focus on scheme optimization and target CDRs which is a special data set.

Second, about CDRs, researchers of [15] compared [16]–[22] CDR analysis solutions and [20], [22] among them with Hadoop architecture has the advantages of with good performance, high scalability, and low cost but they do not give a detail store scheme of CDRs. The authors of [23] studied evaluating the potential of call detail record data in the context of route choice behavior modeling that infers the user's chosen routes or subsets of their likely routes from partial CDR trajectories and data fusion with external sources. Reference [24] studied how to predict complex user behavior combining social, economic, and legal considerations from CDRs by developing a sophisticated model. Researchers in [25] studied degree characteristics and structural properties in large-scale social networks by analyzing tera-scale CDRs that are useful for managing and planning communication networks. Sparse mobility information in CDRs is enriched in [26] that reduces temporal sparsity in CDR by recovering 75% of daytime hours and retaining the spatial accuracy within 1km for 95% of the completed data. Moreover, authors of [27] use more than 800 million CDR to identify weekly patterns of human mobility through mobile phone data that helps local authorities for human mobility analysis and urban planning.

It can be seen that Hadoop is applicable to CDRs storage and analysis. However, for the storage aspect, low cost and scalability of storage media are considered while CDRs retrieval performance is not; for the analysis aspect, extracting user behavior information from CDRs is the hot spot and storage of CDRs is seldom involved. When combing CDRs with system storage schemes, there is little research in this field.

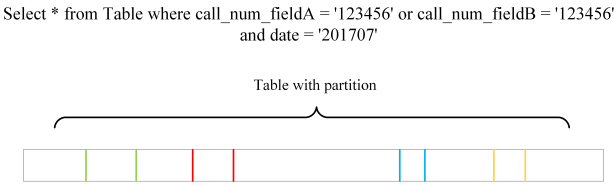

FIGURE 1. Scan range of a typical query under conventional scheme.

B. MOTIVATIONS

The CDRs are a special data set that has high volume and highly repeated information such as the call number. However, CDRs with the same call number are scattered in the storage space and cannot be gathered together then efficiently stored. Besides, the call number is the most frequent key filter in a legitimate query. A *typical query* is like:

*select * from table where call_num_fieldA = '123456' or call_num_fieldB = '123456' and date = '201707'*

This *typical query* on massive CDRs (TB or PB level) is time-consuming because this job is a ''wide-ranged'' normal linear searching process. Fig. 1 shows the *typical query*'s scan range and the data distribution under conventional storage scheme: a table with partition on date. For simplicity, we suppose the table only has one partition. In Fig. 1, the white rectangle represents the table's partition with the date on '201707' and it is filled with massive CDRs (a large city with tens of million people can produce at least hundreds of gigabytes CDRs a month) represented by many invisible lines. The colored lines represent four pairs of CDRs in the partition contains the call number of '123456'. About the ''pairs'', this is a characteristic of CDRs that they are always generated in pairs and stored. The detailed reason is described in Section III. C.

From Fig. 1, we can spot that the CDRs of a call number are scattered in the storage space which means to finish the query, the system has to scan the whole partition. Though we can slice partition to smaller pieces, it makes the query more complicated. Because we have to query more times on different partition, and the scan range of the query on the table remains unchanged indeed. Besides, performance for linear searching on tons of CDRs is poor no matter how you partition the table or adopt the file-level or record-level contributions mentioned in Section I.A. Therefore, we need to design a scheme fulfilling the following two key requirements to tackle the query problem.

- Narrow the scan-range.
- Optimize the linear searching process.

C. CONTRIBUTIONS

In this paper, we propose a dual-column bucketing (DCB) scheme that greatly improves retrieval performance on the premise of reaching a higher compression rate. The main work and contributions of this paper are summarized as follows.

- We propose a novel DCB scheme by extending the existing hash storage scheme. In DCB, CDRs with the same call number gathered in a bucket by hash then are highly compressed. By utilizing the Hive's bucket sampling statement (Tablesample), newly propose DCB retrieval algorithm significantly improves the retrieval speed on *Typical queries*.
- We propose multiple key columns sorting under the DCB scheme. The multiple key columns sorting in each bucket file improves the *typical query*'s running speed further with key columns as filter conditions.
- Implement the system prototype and evaluate the scheme's effectiveness with real-world data and the result shows that our DCB saves storage space by about 40% and improves typical query speed up to 7x.

The remainder of this paper is organized as follows. Section II describes the preliminary techniques. Section III illustrates the DCB scheme proposed in this paper. In Section IV, an experimental evaluation with real CDRs is given. Section V describes work most related to our contributions. Section VI concludes this paper.

II. PRELIMINARIES

The relevant Hive techniques involved in this paper provide essential support for the implementation of our DCB scheme.

A. HIVE BUCKET

Table in Hive supports partitioning and Bucketing. When partitioning, Hive creates subdirectories under table file directory, named after partition fields, and partition data is stored in corresponding partition folders. The essence of partitioning is to slide data using folders. When querying with partition, Hive will only read the corresponding directory files that save query time. For Bucketing, Hive first hashes the data by bucket column (or columns) designated by the user, and then use the hash results to mod the number of buckets to get the remainder. Finally, Hive distributes the records according to the remainder. Assuming that F is the bucketed field, N is the number of buckets, then the formula for calculating bucket number B for each record is as follows:

$$B = Hash(F) \ mod \ N \tag{1}$$

After Bucketing, the data is stored evenly with smaller granularity, and the data in the bucket is sorted by the chosen bucketed field (or fields). In this way, querying does not necessary to read the whole partitioned file, which further improves the retrieval performance. Because of the sorting of the bucketed column in bucket files, the query speed and compression efficiency of the bucketed column is improved.

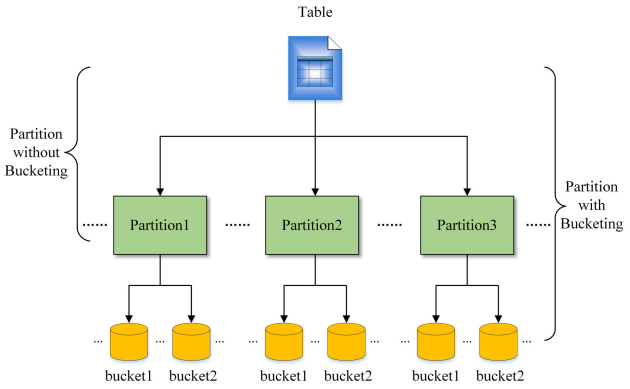

FIGURE 2. Table structure.

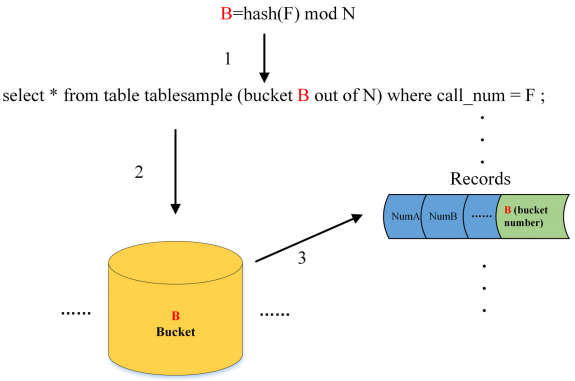

FIGURE 3. Hive Tablesample principle.

The comparison of the table structure with and without Bucketing is shown in Fig. 2.

B. HIVE TABLESAMPLE

Hive has special support for bucketed table: Tablesample, which makes it is possible to specify a bucket to read data through the Tablesample statement when querying, thus avoiding scanning the whole table data (or the whole partition). For example, if the call number F and the number of buckets N is known, according to "(1)," bucket number B can be calculated. And the HQL statements like the following one can be used to query:

*select * from table tablesample (bucket B out of N) where call_num = F;*

Fig. 3 shows the principle of the Tablesample query. If the corresponding bucket number of a call number in CDRs can be calculated beforehand, the Tablesample statement can greatly improve the query performance.

C. HIVE FILE FORMAT

The establishment of Hive data tables requires specifying the file format. Hive supports row-based and column-based storage formats. In version 1.2.1, row-based storage includes TextFile, SequenceFile, and Avro; row-column storage includes Parquet, RCfile, and ORC. Different data

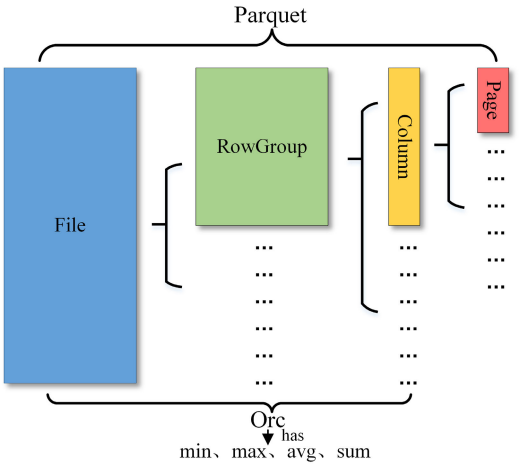

FIGURE 4. File structure of Parquet and ORC.

formats differ in their storage structure and compression algorithm. According to the file format characteristics supported by Hive and the business habits of querying CDRs, the appropriate storage format can be selected to improve query and storage efficiency. Since row-based files are not efficient for storage and query on big data, this paper focuses on row-column files.

Parquet is a row-column file format. The idea of nested data structure design originates from the algorithm in Dremel's paper. It reduces storage costs and improves computing performance, which is also one of its greatest advantages. In point of its structure, the file is divided into row group, column chunk, and page, totally three levels. The file has page-level min and Max statistics (The version in this paper does not have.) and the data is compressed in pages that provide a variety of encoding methods.

RCfile and ORC [28] are row-column file formats, similar in structure to Parquet, but without page-level partitioning. ORC improves the file compression method based on RCfile. It has column block-level compression encoding and improves the efficiency of file compression. Compared with Parquet, ORC also has column-level "avg" and "count" statistics, which Hive can use for block-level filtering when querying. Besides, Bloom filter has been added to ORC for filtering that is equivalent to a further step in the statistical index, which improves the query speed. Fig. 4 is a schematic diagram of the current version in this paper of Parquet and ORC.

Considering that the Hive has original support for ORC file and ORC file format has good encodings, compression algorithms, and multi-dimensional statistical information [29], the data in this paper is stored in ORC file format.

III. THE PROPOSED SCHEME

A. OVERVIEW

Fig. 5 illustrates the CDR Retention System structure in this paper. The system is based on a typical Hadoop distribution

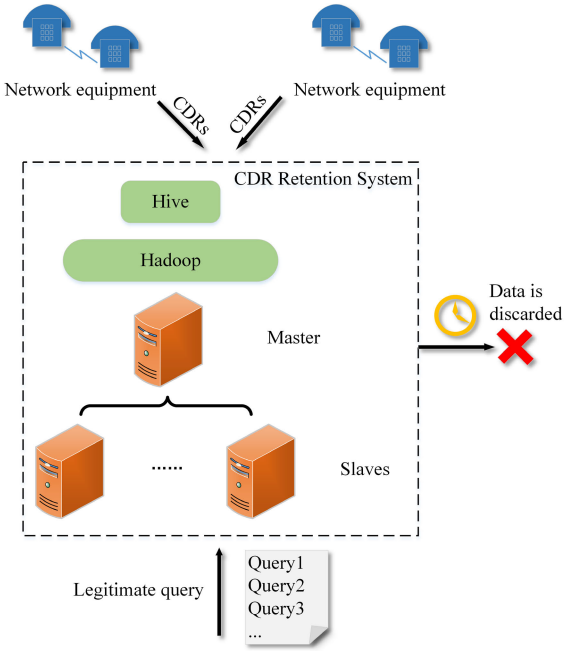

FIGURE 5. DCB system architecture.

system architecture: A master node with Hadoop Namenode and Hive, several slave nodes with Hadoop Datanode. Besides, Fig. 5 performs the life cycle of CDRs: CDRs are produced by the operator's network equipment first and then are stored in the CDR Retention System. At last, after a certain period, the "old CDRs" are discarded.

Combining the storage and retrieval requirement described in Section I., two aspects should be considered.

- About compression: CDRs have high volume and lots of repeated information such as the call number. However, the same call number is scattered in the storage space and cannot be gathered together then efficiently stored.
- About query speed: Randomly distributed call numbers bring low efficiency for retrieval especially querying in a massive CDRs storage database.

Therefore, clustering similar CDRs is the key to solve the problem, and hash storage comes to mind. Based on our knowledge of Hive, we try to introduce a hash storage to reach our goal. The existing technique in Hive does have a hash option named Bucketing, but it only brings limited improvement: Bucketed tables are fantastic in that they allow much efficient sampling than do non-bucketed tables, and they may later allow for time-saving operations such as map-side joins [30]. The more detailed reason and solution are given in section III.B. To tackle this problem, we propose a well-fitting DCB scheme for long-term CDRs storage that successfully takes the best of the advantage of hash storage. As a result, we greatly improved retrieval performance on the premise of improvement on the compression rate.

The DCB scheme optimization can be divided into the following two parts:

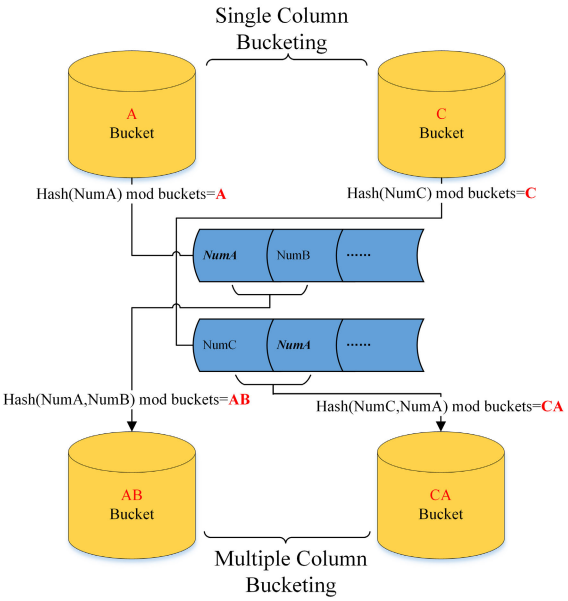

FIGURE 6. Problem of Bucketing.

1) Bucketing Optimization: Implement our hash strategy based on Hive's Bucketing characteristic.

2) Multiple Key Columns Sorting and De-Duplication: Sort the bucket files based on multiple key columns and eliminate redundancy in the files.

B. DCB: BUCKETING OPTIMIZATION

The basic principle of Bucketing and Tablesample has been described in II.B. Directly apply Bucketing mechanism to call number column can do improve storage and query performance that means normal hash strategy is useful.

However, improvement is poor. Fig. 6 shows an example of two different conventional Hive original Bucketing schemes that bring two different problems. And the two schemes are SCB (Single Column Bucketing) on the top and MCB (Multiple Column Bucketing) on the below. *A Bucket* and *C Bucket* are the buckets of SCB; *AB Bucket* and *CA Bucket* are the buckets of MCB. Two horizontal bars in blue represent two different records of CDR with two call number fields given. *NumA*, *NumB*, and *NumC* represents three different call numbers. If we focus on *NumA*, the SCB cannot allocate two records both containing *NumA* to a single bucket because *NumA* and *NumC* have different hash values. Turn to MCB, we also focus on *NumA* and it is found that since *(NumA, NumB)* and *(NumA,NumC)* has different hash values, we cannot allocate two records both containing *NumA* to a single bucket either.

This result means that we cannot do a *typical query* of NumA on a single bucket and have to query on every bucket that contains NumA which has no difference with the query on the scheme without Bucketing. These two schemes of hash do not consider a *typical query* performance. In other words, the traditional hash storage scheme only takes query on one column primary key into consideration, when there has a dual-column primary key, the conventional hash storage scheme needs to be changed.

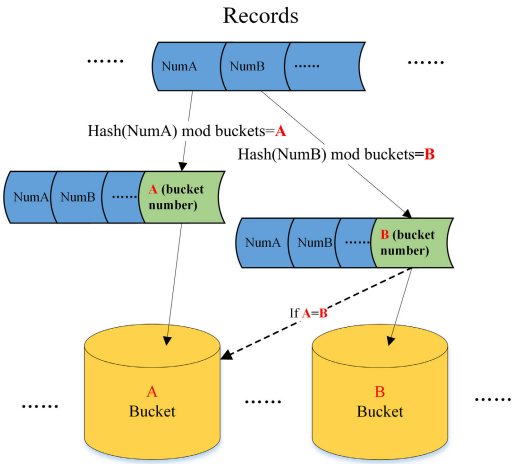

FIGURE 7. Solution scheme.

Algorithm 1 Hash Code for String

Input: char *value*[]
Output: int *h*

```
int h = hash;
if h = 0 && value.length > 0 then
    char val = value;
    for int i = 0; i < value.length; i ++ do
        h = 31 * h + val[i];
    hash = h;
return h;
```

To solve this problem, Fig. 7 illustrates the scheme of our hash storage design and the detail is as follows:

1) In data pretreatment, manual hashing operation is performed on two columns of call number fields separately.

2) A new column field is added to the table for playing the role of a bucket column.

Considering the first design, since two call number fields are allocated to buckets respectively, the same call number in two columns of number fields will fall into the same bucket, which can ensure that only one bucket is needed to retrieve all records of a call number. About the hash function, the Hive's hash algorithm of String type is as follows:

Algorithm 1 uses the remainder of an int's overflow as a hash code of a String. The hash table size is the length of an int that has 8 bytes which means that the hash table size is over 4 billion. In our research field, a country normally has fewer subscribers than the hash table size and thus this hash function is qualified to satisfy our requirement for allocating call numbers evenly to buckets. Besides, the multiplier 31 is a large enough odd prime that effectively limits the conflicts and the hashing process is fast due to the multiplication that can be replaced by a shift and a subtraction. Therefore, we use this hash function to hash the call number.

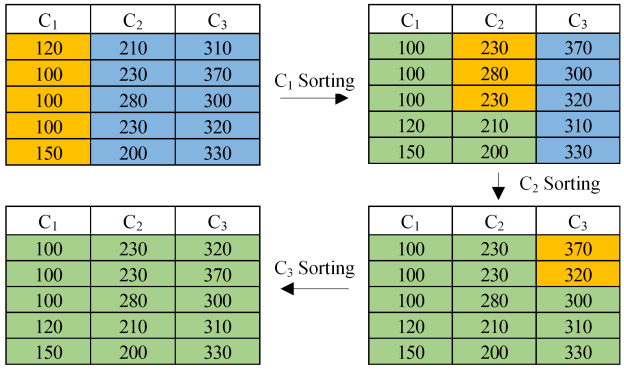

FIGURE 8. Multiple Columns Sorting.

For the second design, after adding a column of fields for indicating the bucket, we can utilize the field to do the Tablesample query of a call number now. Besides, due to the field added in each bucket file has the same value, this bucket column occupies almost no storage space.

In general, the purpose of the Tablesample query on a designated bucket is achieved by manual hashing of two columns and adding a bucket column. The scheme clusters all of the similar CDRs to a predictable small storage space called bucket that changes the data structure of traditional hash storage scheme (from each record has a unique hash value to similar records share a hash value.) The scheme can also applied to storage of SMS, email, and extended detail records (XDRs) which has similar two-party communication features to CDRs. In addition, this is a scheme of trading space for time. Because during the hashing phase, a call detail record has been copied to another bucket. However, CDRs are a special data set that always contain duplication. The next subsection describes and tackles this problem.

C. DCB: MULTIPLE KEY COLUMNS SORTING AND DE-DUPLICATION

With Bucketing operation, files have been sliced to sufficiently small on size. Thus sorting within bucket files becomes easy and feasible. When reading an ORC file in Hive, the predicate will be pushed down to the row filter, thus sorting the frequently used conditional fields will improve query performance. Therefore, this paper sorts those two call number fields in each bucket.

Fig. 8 shows the diagram of multiple key columns sorting. In the diagram, the data are sorted in ascending order according to the weighted priority: C1>C2>C3. Yellow represents cells to be sorted, blue represents cells unsorted, green represents cells have sorted. In addition, because many identical numbers are aggregated in the same bucket after Bucketing, the sorted data could occupy less space in storage.

In reality, CDRs are always generated in pairs. When a call occurs, both the calling and the called billing system (or base station) will record an identical call record of the call, like ''twins''. When an operator collects CDRs from different billing systems in different areas together and stores them in the storage system, without a high-cost full scan, the same records cannot be de-duplicated. However, combined with the scheme in Fig. 7, two identical records will be allocated to the same bucket. To save storage space and decrease the data inflation effect brought by the scheme, the same records in a bucket are de-duplicated. The cost of de-duplication is low because the records are sorted and the time complexity is about O(n). After all these pretreatments, the number of records should be bigger than that of original records because there are still some calls that do not have a ''twin'' but they are duplicated. The size of the processed CDRs also should be bigger than that of the original one because one more field for bucket number is stored and some ''twin'' is produced. The compression ratio of the scheme depends on how many ''twins'' there are among the original data.

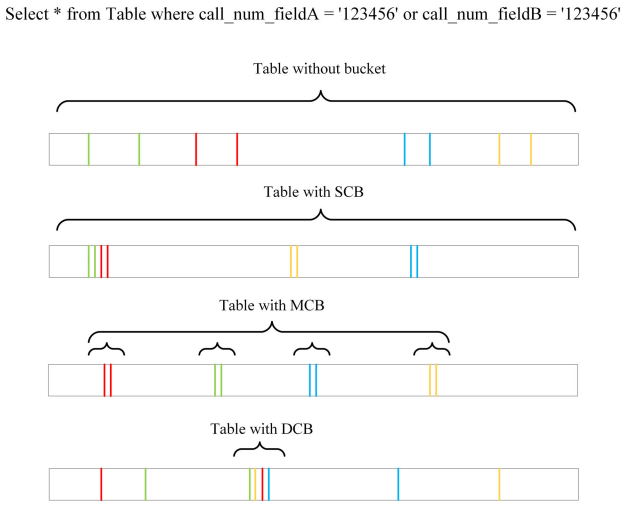

FIGURE 9. Data distribution and scan range of a typical query on four schemes.

D. DCB: RETRIEVAL ALGORITHM

To illustrate the advantage and principle of the DCB retrieval algorithm, Fig. 9 shows a typical query and four storage schemes (for simplicity, we omit the filtering field of time in the query.) In the figure, the white rectangle represents a CDRs storage space that stored massive CDRs which is represented by many invisible lines. The highlighted 8 lines of four colors represent 4 pair of CDRs that contains call number '123456' and especially, green and red lines indicates call number field A's call number is '123456'; blue and yellow lines represent call number field B's call number is '123456'. Each pair contains the same CDR. The brace on top of each rectangle means the scan range of the query on the scheme. First, we focus on the 4 pairs of CDRs' distribution under these different schemes.

- Considering the table scheme without Bucketing, because CDRs are collected by chronological order, every CDR of each pair gathered by pairs and scattered in the storage space.

Algorithm 2 Conventional Scheme Retrieval of TQ

Input: $L_T[r_1, r_2, r_3, \ldots, r_n]$, *call_num*
Output: $L_R[r_1, r_2, r_3, \ldots, r_m]$

```
//a full table scan
for i = 1; i ≤ n; i++ do
    if match(L_T[i], call_num) then
        add(L_R, L_T[i])
return L_R
```

Algorithm 3 DCB Scheme Retrieval of TQ

Input: $L_T[B_1[r_1^1, \ldots, r_{n_1}^1], \ldots, B_n[r_1^n, \ldots, r_{n_n}^n]]$, *call_num*
Output: $L_R[r_1, r_2, r_3, \ldots, r_m]$

```
//calculate the target bucket
b = Hash(call_num) mod n
//a scan of the target bucket
for i = 1; i ≤ n_b; i++ do
    if match(B_b[i], call_num) then
        add(L_R, B_b[i])
return L_R
```

- When adopting SCB scheme, assume we bucket the call number field A, since green and red lines represent call number field A's call number is '123456' that means they are the same, these two pair of CDRs are allocated to the same bucket. In Fig. 9, we put them in a very close position to express. Because we don't know what yellow and blue pairs of CDRs' call number field A are, the records are allocated to different buckets respectively and they are randomly distributed in the storage space.
- About the MCB scheme, since two fields of call number make up the bucket field, each pair has different hash value and they will be allocated to different buckets. Also, they are scattered in the storage space.
- Concerning the DCB scheme, Fig. 9 shows that one record in each pair of CDRs will be allocated to the same bucket because they all contain the call number of '123456' wherever the call number is in call number field A or call number field B. The other record in each pair will appear in other bucket.

Turn to the query, Algorithm 2,3,4,5 shows the retrieval algorithms of *typical query* (TQ) on each scheme. We define $L_T[r_1, r_2, r_3, \ldots, r_n]$ as a list of CDRs without bucketing, $L_T[B_0[r_1^0, \ldots, r_{n_1}^0], \ldots, B_{n-1}[r_1^{n-1}, \ldots, r_{n_n}^{n-1}]]$ as a list of CDRs with bucketing, $L_H[h_1, h_2, \ldots, h_p]$ as a list contains MCB hash value with call numbers, *call_num* as the target number, and $L_R[r_1, r_2, r_3, \ldots, r_m]$ as a list of target CDRs.

Combine Fig. 9 with these four retrieval algorithms, it can be seen that the conventional scheme and SCB scheme scans the whole storage space to obtain target CDRs respectively, especially, SCB scheme needs two scans to finish the query; MCB scheme need access an auxiliary table L_H to know what buckets contain the *call_num* first, then search each bucket to obtain target CDRs. The scan times equals to the length of L_B. Generally, considering the system startup cost brought by multiple scans and scan range in SCB and MCB, their query speed is slower than that of the conventional scheme.

Compared to these algorithms, our proposed DCB retrieval algorithm has the simplest process and the fastest speed that can complete the query through a single scan of a bucket.

Algorithm 4 SCB Scheme Retrieval of TQ

Input: $L_T[B_0[r_1^0, \ldots, r_{n_1}^0], \ldots, B_{n-1}[r_1^{n-1}, \ldots, r_{n_n}^{n-1}]]$, *call_num*
Output: $L_R[r_1, r_2, r_3, \ldots, r_m]$

```
//calculate the target bucket
b = Hash(call_num) mod n
//a scan of the target bucket
for i = 1; i ≤ n_b; i++ do
    if match(B_b[i], call_num) then
        add(L_R, B_b[i])
//a scan of rest buckets
if b > 0 then
    for i = 0; i < b; i++ do
        for j = 1; j ≤ n_i; j++ do
            if match(B_b[j], call_num) then
                add(L_R, B_b[j])
if b < n − 1 then
    for i = b + 1; i < n; i++ do
        for j = 1; j ≤ n_i; j++ do
            if match(B_b[j], call_num) then
                add(L_R, B_b[j])
return L_R
```

Algorithm 5 MCB Scheme Retrieval of TQ

Input: $L_T[B_1[r_1^1, \ldots, r_{n_1}^1], \ldots, B_n[r_1^n, \ldots, r_{n_n}^n]]$, $L_H[h_1, h_2, \ldots, h_p]$, *call_num*
Output: $L_R[r_1, r_2, r_3, \ldots, r_m]$

```
L_B = []
//a scan of auxiliary table to obtain target buckets
for i = 1; i ≤ p; i++ do
    if match(L_H[i], call_num) then
        add(L_B, h_i mod n)
//a scan of target buckets
for i = 0; i < L_B.length(); i++ do
    for j = 1; j ≤ n_{L_B[i]}; j++ do
        if match(B_{L_B[j]}, call_num) then
            add(L_R, B_{L_B[j]})
return L_R
```

TABLE 1. Configuration of a node.

OS	CPU	Memory	Hard disk	Software
Red Hat 7.2	2CPU (E5-2690 v4) 6 cores 2.60GHz	128GB	1TB	Hadoop 2.7.3 Hive 1.2.1

TABLE 2. Statistics of table fields.

Field	Size(B)
calldirection	2.0
callera	12.0
callerb	12.2
imsia	15.99
imsib	15.99
calldate	7.0
starttime	7.0
endtime	7.0
time	3.3
Total	82.48

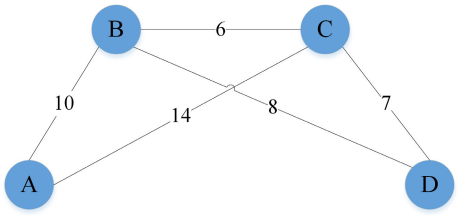

FIGURE 10. Call Degree on undirected Graph.

IV. EXPERIMENT

A. EXPERIMENT ENVIRONMENT

This experiment uses the development environment of a communication company. There are five nodes in the experimental cluster, including two management nodes and three data nodes. A single node environment is shown in Table 1.

B. DATA SET DESCRIPTION

The test data uses a month's real CDRs of a certain place in China of 2017. The original size of the data was 13.4 GB, totaling about 170 million rows. The data fields and field statistics of the data are shown in Table 2.

To accurately evaluate the characteristic of the data set, we define a new index: Average Call Degree (ACD). The data structure of CDRs can be considered as an undirected graph like Fig. 10.

In Fig. 10, the A, B, C, and D represents four different call numbers; the number on each edge represents the number of calls made between these two call number. Then, the ACD is calculated by "(2)."

$$ACD = \frac{\sum_{i}^{n} C_i}{N} \tag{2}$$

TABLE 3. Statistics of calls.

max calls	min calls	ACD
2943	1	4.5

**max calls* is a number of maximum calls made by a call number; *min calls* is a number of minimum calls made by a call number.

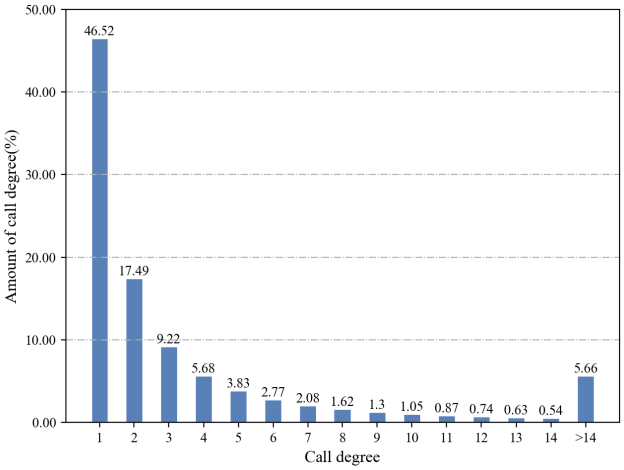

FIGURE 11. Distribution of call degrees on call degree.

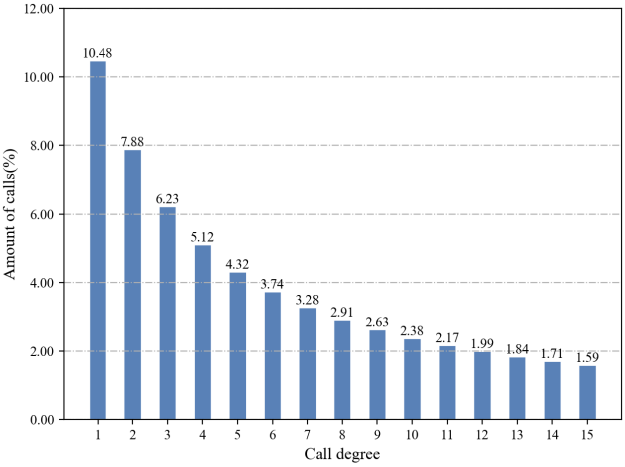

FIGURE 12. Distribution of amount of calls on call degree.

C_i is calls made between these two call numbers, and N is the number of pairs of calls. The ACD of Fig. 10 is not (10+6+7+14+8)/4=11.25 but (10+6+7+14+8)/5=9. ACD reflects how repetitive of each edge is, in other words, it reflects each edge can be compressed to what extent. Higher the ACD is, which means more calls were made between a pair of call numbers, the better compression will be. It is a tool to predicate the DCB compression ratio.

Table 3 shows the basic statistics of the test data.

Two groups of distribution information of the data are shown in Fig. 11 and Fig. 12. Call Degree (CD) is a concept like ACD but without average operation.

Generally, Fig. 11 and Fig. 12 illustrates that the data distribution is in a reasonable range which is compliance with

TABLE 4. Contrast of Experimental Conditions.

Condition	normal	opt
Partitioning	✓	✓
DCB Bucketing	×	✓
MKCS and Dd	×	✓

**MKCS*=Multiple Key Columns Sorting, *Dd*=De-duplication.

Pareto's principle which means almost 80% of calls are made by 20% of callers. Although Fig. 11 shows that call degree 1 occupies almost half of all degree in data which brings bad compression, Fig. 12 shows that it has about 10 percent of calls in total which means the compression will only be affected limitedly.

C. EXPERIMENT DESIGN

The experiment was divided into control group (*normal*) and optimization group (*opt*). The control group adopt the conventional storage scheme in Hive. The optimization group adopt the DCB scheme. Table 4 shows the relevant condition setting information:

Considering that the size of compressed data is slightly smaller than that of HDFS file blocks, the table is divided into 24 buckets to ensure that ORC file blocks do not cross file boundaries. Besides, to eliminate the influence brought by file format, the ORC and Parquet file formats were both used in the control group and the optimization group. The ORC file was compressed by ZLIB, and the Parquet file was compressed by GZIP.

The experiment will compare:

1. The data loading time of and space occupied by the two file formats in the two groups.

2. Query speed of two file formats in two groups. All queries are just like the *typical query* that are most frequently used in the legitimate query. The performance test is divided into two sets:

- The first set is predictable bucket number query, including seven queries. The query of NO.1-5 are aggregate queries returns a call number's aggregation information in a period and the queries include keywords for min, max, count, sum, avg. Query NO.6 is a small-ranged query that returns all the records of a call number in a period (returns 73 records). Query NO.7 is a point query that returns a single record of a call number that happened at a moment.
- The second set is unpredictable bucket number query, including two query statements NO.8 and NO.9: query NO.8 is cut-end query and query NO.9 is cut-start query and they both limit the call number.

3. The amount of disk read by two file formats for different queries in two groups.

4. The number of records read and CPU time in two sets of two groups.

In this experiment, the control group uses the "*normal*" tag, the optimization group uses the "*opt*" tag, the text file uses the "*text*" tag, the ORC file uses the "*orc*" tag, the Parquet file uses the "*parq*" tag, and the number of rows of files uses the "row" tag.

TABLE 5. Data size and number of rows before and after pretreatment.

Type	opt	normal
text	13.8GB	13.4GB
row	177,482,982	173,778,775

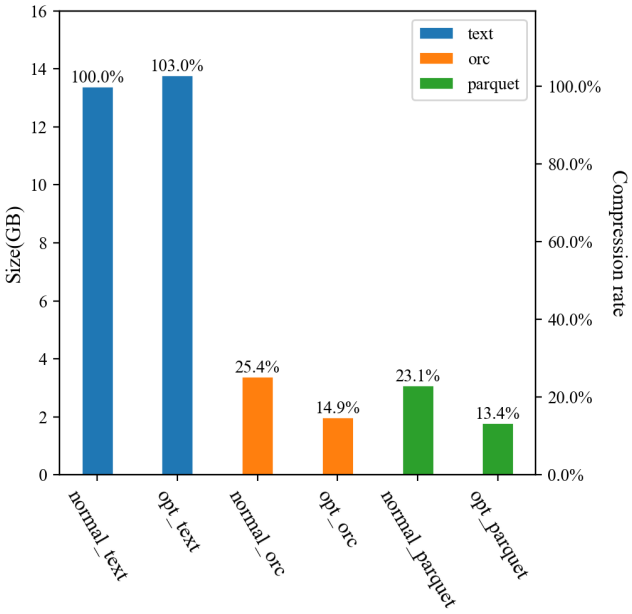

FIGURE 13. Compression comparison.

D. EXPERIMENT RESULT AND ANALYSIS

Table 5 shows that the number of records in the optimization group is slightly larger than that of the original one which means that there are a majority of "twin" records in the given data. Correspondingly, the opt data size increases a few by about 3%. This result is consistent with the prediction in Section III. C.

Fig. 13 shows the size and compression rate of different file formats with different schemes of CDRs. The data storage space of the optimization group is significantly reduced. ORC and Parquet format files in the optimization group save more than 40% of the storage space. The overall compression rate is about 14%.

To simulate the worst condition, which means there are no "twin" in data, we use "brute force" to sort the whole data set and de-duplicate all redundancies and then compare the data size and rows. The result is shown in Table 6. We use the *De-* tag to denote the no "twin" data. It can be seen that the optimized group data size is still slightly smaller than that of the other group, which proves that although our scheme is a trading space for time strategy, it still avoids data inflation and even improved the compression rate.

TABLE 6. Size and number of rows under the worst condition.

Type	opt	De-normal
text	13.8GB	6.8GB
orc	2.0GB	2.1GB
parq	1.8GB	1.83GB
row	177,482,982	88,741,491

De- tag is used to denote the processed CDRs without "twin" data.

TABLE 7. Average size of fields.

Field	opt_orc(B)	normal_orc(B)	De-normal_orc(B)
callera	0.59	2.49	3.11
callerb	1.66	3.95	4.95
imsia	0.72	2.79	4.15
imsib	1.72	4.4	5.85
other	7.67	7.20	7.66
Total	12.36	20.83	25.72

TABLE 8. Size of fields(simulation).

Field	ACD5(B)	ACD10(B)
callera	0.60	0.26
callerb	1.77	0.77
imsia	0.78	0.4
imsib	1.81	0.87
other	7.67	7.67
total	12.63	9.97

By analyzing the metadata files of ORC, Table 7 shows the average size of each field with the file format.

It is found that the average space for two call number fields of each record in the control group accounted for 6.44 bytes, while these fields in the optimization group occupied only 2.25 bytes. And because the IMSI field corresponds to the call number field one by one, the storage space of the two IMSI fields is also significantly reduced, from 7.19 bytes to 2.44 bytes. In fact, the ACD in Table 3 can explain the effect very well. To verify the effectiveness of ACD, we set two groups of randomly generated data to simulate different data with different ACD (5 and 10). Because callera, callerb, imsia, and imsib are four main fields to compress, we keep the other fields unchanged. Besides, we also basically simulate the distribution mentioned in Fig. 9 and Fig. 10 to approach a more accurate simulation. The result is shown in Table 8.

The goal of setting the ACD5 column in Table 8 is to simulate the experiment data adopted in this paper (ACD4.5) and the result shows that the average error of these four fields is about 5.4% which proves the effectiveness of the simulation. The second column is a reasonable inference that provides a bigger ACD because our experiment only includes a single month's CDRs that is one-sixth of the retention requirement. The result shows that we can save at least more than 20% storage space if we can experiment with bigger data. The compression ratios of ACD5 and ACD10 are 15% and 12% respectively. ACD provides a new index for other researchers to predicate the compression rate of their data under the DCB scheme.

TABLE 9. Data loading time.

Type	opt(s)	normal(s)
orc	311	410
parq	238	307

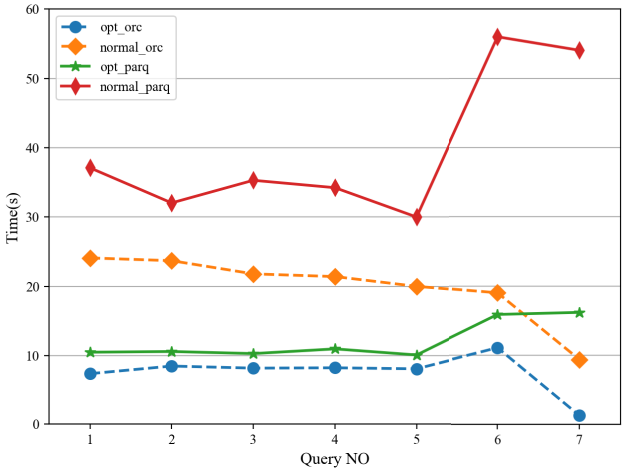

FIGURE 14. Query performance of the first set.

Table 9 shows the loading time of the control group is longer than that of the optimized group when specifying file formats. In fact, for these four modes, the unit loading time is 155.5s/GB (opt with orc), 120.5s/GB (normal with orc), 132s/GB (opt with parq) and 99s/GB (normal with parq), respectively. It can be seen that in the case of the Bucketing group, the actual unit loading time increases due to an additional step of data processing (Bucketing), but the overall loading time decreases due to the reduction of the total size of the data.

The result in Fig. 14 of the first set shows that the query performance of the optimization group is better than that of the control group. For ORC files, the average speed of aggregate query increased to about 2.7 times, that of small data sets is about 1.7 times and that of point query is about 7.2 times. For Parquet files, the average speed of aggregate query is 3.1 times faster than the control group, that of small data sets is 3.5 times and that of point query is 3.3 times. Generally speaking, the improvement of query performance is obvious.

As can be seen from Fig. 15, the amount of query disk read by the optimization group is significantly lower than that of the control group. Query 7 reduces the most disk reads, about 99% in ORC file format and 97% in Parquet format. Overall, for aggregated queries numbered 1-5, ORC files and Parquet files are reduced by about 98% on average; for small data set queries, ORC files and Parquet files are reduced by about 97%. This shows that the optimization model is consistent with the expectation, that is, by utilizing

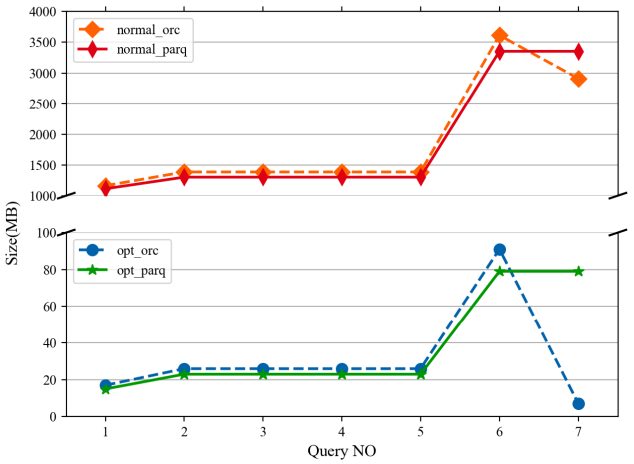

FIGURE 15. Disk read amount of the first set.

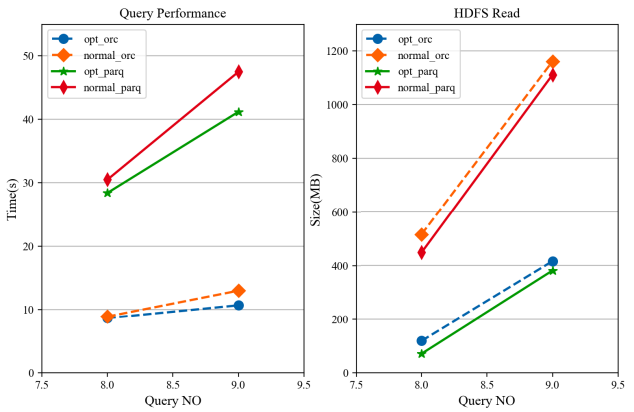

FIGURE 16. Query performance and disk read amount of the second set.

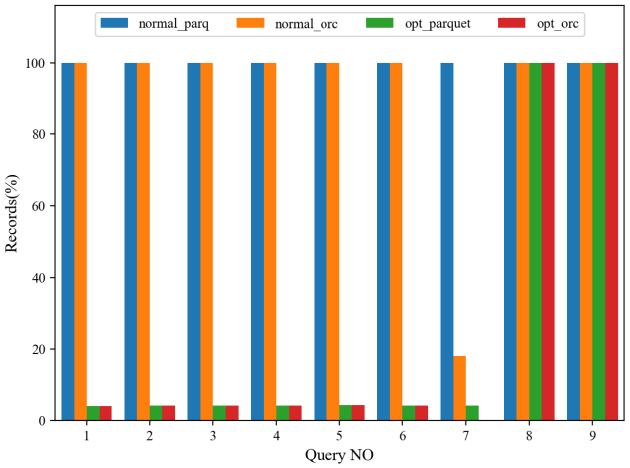

FIGURE 17. Records read amount.

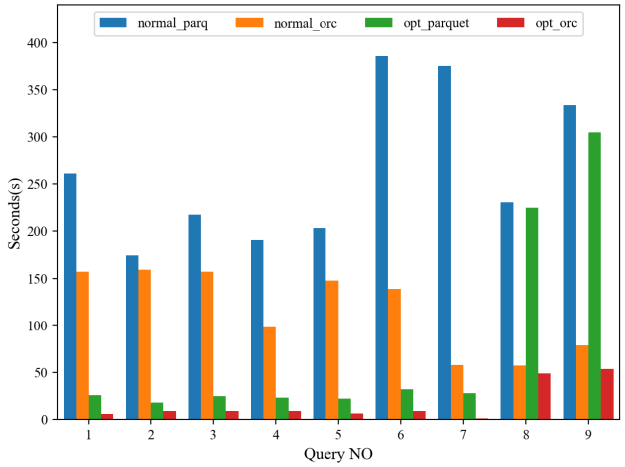

FIGURE 18. Accumulated CPU time consumed.

Bucketing and Tablesample to reduce the amount of reading to improve the query performance. For the comparison of file formats, observing the log records of Query 7 shows that ORC reads fewer records than parquet from the same bucket file. It proves that ORC's column-level statistical index plays its role, which makes irrelevant column blocks skipped in query and improves query performance.

However, although query performance has been improved to some extent, the dramatic reduction of query reading is not proportional to the improvement of query speed. Theoretically, according to the amount of disk read, the query speed should be increased by up to 50 times. Considering that the TEZ engine this experiment adopts still uses the Map-Reduce framework, the overall start-up and intermediate running costs of the framework are expensive (including network IO time, read-write intermediate file time). Adding the condition that queries of long-term CDRs are simple, they explained the reason for relatively poor retrieval performance. If the experiment adopts a more efficient querying engine with low costs that are compatible with Hive, the improvement should be significant.

The result of the second set is shown in Fig. 16. The purpose of the second set test is to compare the query performance of the optimization group and the control group when the bucket number is cannot be predicted. As can be noticed, the overall query performance has been improved slightly, by about 10%. Since the bucket number cannot be predicted, the system reads more data, but the size of the bucketed group still reduces by about 70% on average. To find out which factor (bucketing or sorting) plays a more important role in the reduction of reading bytes, we create a Bucketing table without multiple key columns sorting and test it with query 8 and 9. The result shows that the size difference of reading between this only bucketed table and the optimized group is reduced by about 37%. Therefore, it can be inferred that sorting plays a major role in reducing the amount of reading. About the poor performance improvement, after observing other statistic information, it is spotted that the difference of query time of the query 8 and 9 between the two groups is small while the reading amount differs greatly which proved that the reading time is not the bottle affecting the performance and other system cost should be the main reason for performance degradation in this case.

Fig. 17 and Fig. 18 are supplementary experiments that support retrieval performance improvement by reducing the

reading amount and saving CPU running time. Fig. 16 shows an obvious reduction of the number of the reading records of set 1 queries which is decreased to about 4.2% on average except for query 7 and the same number of reading records of set 2 queries. Query 7 only read 0.005% data and the bar in the figure can be hardly seen. Fig. 17 shows that Bucketing saves an average of more than 90% of CPU time of set 1 queries and the time does not differ much between the different groups of set 2 queries that saving about 14% CPU running time on average. This fact also supports that in set 2 queries, except for CPU running time and data reading time mentioned above, other system costs should be the main factor for poor performance improvement.

E. OTHER DISCUSSION

The experiment result proves that the DCB scheme can bring remarkable improvement both on compression and retrieval. To maximize the performance of the DCB scheme, the other two factors should be considered.

First, the number of buckets determines how much data can be read during Tablesample that further determines the retrieval performance. The more buckets there are, the fewer records will be allocated in each bucket, so that the fewer records will be processed in query and finally achieving a higher querying speed. However, when there are too many buckets, a large number of small files will be generated. Under this situation, firstly, the compression ratio of a bucket file will be reduced for there is less redundancy. Secondly, Hadoop is not suitable for storing a large number of small files, which will cause a large number of disk fragments, affect the efficiency of system IO, and impose a huge burden on Namenode. Therefore, setting the number of buckets reasonably can find the proper balance between query performance and storage efficiency, and optimize the overall performance of the system. We suggest that the bucket file size is just smaller than the HDFS block size to minimize the IO cost.

Second, the overall data size has a great impact on performance. In our scheme, bucket size is relatively fixed (approximately equal to HDFS block size) which means when querying on larger data, since we only need to access one bucket in a query, we still just need to read similar size of data to complete the query. Thus, we can save more time than the conventional scheme when doing retrieval on larger data. Moreover, CDRs with longer periods have higher ACD which means better the compression rate will be. Therefore, our scheme is especially useful for massive data storage. In our experiment, the data presents a medium city with about a million people's CDRs in a month. In China, there are over 100 cities have more than a million population and there 6 month's CDRs need to be stored which means there exists much larger data. We estimate CDRs with 6 months is easy to reach ACD30 or further and the compression rate will reach 10% or better. We do believe that our contribution will help the operators save more costs and help the government increase the efficiency of enforcement.

V. RELATED WORK

About CDRs storage schemes, there is little research. Researchers of [2] studied the security requirements of CDR retention ten years ago. In their paper, they discuss the importance of the retention task regarding politics, privacy, non-repudiation, integrity, performance and cost requirements. Finally, several protection schemes for long-term data storage are proposed from the security level. However, the scheme mainly focuses on how to prevent CDRs from row changing, unauthorized row creating and row deleting attacks, etc. while it does not consider the performance of retrieval and storage. Similarly, the authors of [31] proposed a management system that suggests moving CDRs to a trusted third party to store while do not give an efficient storage scheme either. Compared to our contributions, we implemented a prototype of a high-performance CDRs storage system.

Besides, the followings are the most related work focusing on improving storage efficiency and retrieval performance that enlightened our work.

About local sorting, to improve the storage performance of structured data, the authors in [32] proposed a solution based on *multiple key columns sorting*, KCGS-Store. Test on TPC-H and HttpLog datasets shows that the file format has been improved in query speed, compression ratio and loading time compared with the ordinary Hive file format. In [33], the authors studied the compression strategy of HBase and proposed a compression storage strategy based on *sorting in the region*, which can sort data before compression and then select the appropriate compression strategy after sorting. The strategy has been tested on the TPC-DS dataset and achieves a compression rate of 34%.

About Hive bucketing, the authors of [34] studied the performance of Hive for different file formats and different compression algorithms supported by Hive. By adjusting the configuration of the Hive, especially setting the number of Maps and Reduces and adopting *the Bucketing method*, the efficiency of the Hive can be increased by up to 30%.

VI. CONCLUSION AND FUTURE WORK

This paper proposed a novel DCB scheme, in which a Hive-based extended hash storage, other than the traditional sequential storage, is employed. The experiments demonstrated that the proposed DCB scheme can significantly improve the retrieval speed of CDRs while ensuring a high compression ratio. In addition, this scheme is extensible. For example, it can be applied to the email, SMS, and other XDRs storage scene because they have similar characteristics of two-party communication. Also, we put forward an evaluation index ACD to predicate the compression effect on CDRs. Other researchers can calculate the ACD in their data to predict DCB performance on the data. Furthermore, our scheme is more effective in larger CDRs storage scenes that means the retrieval speed will be improved further.

In the future, we will make a study on the query of unpredictable bucket number and try to improve its query efficiency. In addition, the system startup cost of the MR

framework in Hive is expensive. Different computing engines (such as Spark) will be investigated for different data volume queries in an attempt to achieve better query performance.

REFERENCES

[1] *Data Retention*. Accessed: 2019. [Online]. Available: https://en.m.wikipedia.org/wiki/Data_retention

[2] F. Vancea, C. Vancea, D. E. Popescu, D. Zmaranda, and G. Gabor, "Secure data retention of call detail records," *Int. J. Comput. Commun.*, vol. 5, no. 5, pp. 961–967, Dec. 2010, doi: 10.15837/ijccc.2010.5.2260.

[3] S. P. Menon and N. P. Hegde, "A survey of tools and applications in big data," in *Proc. IEEE 9th Int. Conf. Intell. Syst. Control (ISCO)*, Coimbatore, India, Jan. 2015, pp. 1–7.

[4] P. Jayawardhana, D. Perera, A. Kumara, and A. Paranawithana, "Kanthaka: Big data caller detail record (CDR) analyzer for near real time telecom promotions," in *Proc. 4th Int. Conf. Intell. Syst., Modelling Simul.*, Bangkok, Thailand, Jan. 2013, pp. 534–538, doi: 10.1109/ISMS.2013.40.

[5] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *Proc. IEEE 26th Symp. Mass Storage Syst. Technol. (MSST)*, May 2010, pp. 1–10.

[6] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: A warehousing solution over a map-reduce framework," *Proc. VLDB Endowment*, vol. 2, no. 2, pp. 1626–1629, Aug. 2009, doi: 10.14778/1687553.1687609.

[7] M. N. Vora, "Hadoop-HBase for large-scale data," in *Proc. Int. Conf. Comput. Sci. Netw. Technol.*, Harbin, China, Dec. 2011, pp. 601–605, doi: 10.1109/ICCSNT.2011.6182030.

[8] H. Y. Wang and C. W. Fu, "Compression strategies selection method based on classification of HBase data," *J. Commun.*, vol. 37, no. 4, pp. 12–22, 2016.

[9] H. Xue, "Research on storage and retrieval optimization of bigdata," M.S. thesis, Dept. Info. Sw. Eng., Univ. Electron. Sci. Technol. China., Chengdu, China, 2018.

[10] Y. F. Li, J. J. Le, D. H. Chen, M. Wang, and B. Zhang, "PageFile: The return of classical page storage structure on MapReduce framework," *J. Internet Technol*, vol. 18, no. 1, pp. 65–75, 2017.

[11] 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 *Proc. IEEE 27th Int. Conf. Data Eng.*, Hannover, Germany, Apr. 2011, pp. 1199–1208, doi: 10.1109/ICDE.2011.5767933.

[12] F. Gao, A. Dutta, and J. J. Liu, "Content-based textual big data analysis and compression," in *Proc. Int. Conf. Comput. Big Data (ICCBD)*, Charleston, Sc, USA, 2018, pp. 1–7, doi: 10.1145/3277104.3277107.

[13] D. Dong and J. Herbert, "Record-aware two-level compression for big textual data analysis acceleration," in *Proc. IEEE 7th Int. Conf. Cloud Comput. Technol. Sci. (CLOUDCOM)*, Vancouver, BC, Canada, Nov./2015, pp. 9–16, doi: 10.1109/CloudCom.2015.32.

[14] D. Dong and J. Herbert, "Record-aware compression for big textual data analysis acceleration," in *Proc. IEEE Int. Conf. Big Data*, Santa Clara, CA, USA, Oct. 2015, pp. 1183–1190, doi: 10.1109/BigData.2015.7363872.

[15] S. B. Elagib, A.-H. A. Hashim, and R. F. Olanrewaju, "CDR analysis using big data technology," in *Proc. Int. Conf. Comput., Control, Netw., Electron. Embedded Sys. Eng. (ICCNEEE)*, Khartoum, Sudan, Sep. 2015, pp. 467–471, doi: 10.1109/ICCNEEE.2015.7381414.

[16] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear, "The vertica analytic database: C-store 7 years later," *Proc. VLDB Endowment*, vol. 5, no. 12, pp. 1790–1801, Aug. 2012, doi: 10.14778/2367502.2367518.

[17] (2019). *Optiva*. [Online]. Available: https://en.wikipedia.org/wiki/Optiva

[18] S. Yang, B. Wang, H. Zhao, Y. Gao, and B. Wu, "DisTec: Towards a distributed system for telecom computing," in *Cloud Computing* (Lecture Notes in Computer Science), vol. 5931. Berlin, Germany: Springer, 2009, ch. 19, pp. 212–223, doi: 10.1007/978-3-642-10665-1_19.

[19] E. Bouillet, R. Bouillet, V. Kumar, L. Mignet, S. Nathan, A. Ranganathan, D. S. Turaga, O. Udrea, and O. Verscheure, "Processing 6 billion CDRs/day: From research to production (experience report)," in *Proc. 6th ACM Int. Conf. Distrib. Event-Based Syst. (DEBS)*. New York, NY, USA: ACM, 2012, pp. 264–267, doi: 10.1145/2335484.2335513.

[20] J.-C. Tseng, H.-C. Tseng, C.-W. Liu, C.-C. Shih, K.-Y. Tseng, C.-Y. Chou, C.-H. Yu, and F.-S. Lu, "A successful application of big data storage techniques implemented to criminal investigation for telecom," in *Proc. 15th Asia–Pacific Netw. Oper. Manage. Symp. (APNOMS)*, Hiroshima, Japan, Sep. 2013, pp. 1–3.

[21] P. Valduriez, "Parallel database systems: Open problems and new issues," *Distrib. Parallel Databases*, vol. 1, no. 2, pp. 137–165, Apr. 1993, doi: 10.1007/BF01264049.

[22] C. Şenbalcı, S. Altuntaş, Z. Bozkus, and T. Arsan, "Big data platform development with a domain specific language for telecom industries," in *Proc. High Capacity Opt. Netw. Emerg./Enabling Technol.*, Magosa, Cyprus, Dec. 2013, pp. 116–120, doi: 10.1109/HONET.2013.6729768.

[23] A. Bwambale, C. Choudhury, and S. Hess, "Modelling long-distance route choice using mobile phone call detail record data: A case study of Senegal," *Transportmetrica A, Transp. Sci.*, vol. 15, no. 2, pp. 1543–1568, Nov. 2019, doi: 10.1080/23249935.2019.1611970.

[24] C. Doyle, Z. Herga, S. Dipple, B. K. Szymanski, G. Korniss, and D. Mladenic, "Predicting complex user behavior from CDR based social networks," *Inf. Sci.*, vol. 500, pp. 217–228, Oct. 2019, doi: 10.1016/j.ins.2019.05.082.

[25] H. Aksu, L. Korpeoglu, and O. Ulusoy, "An analysis of social networks based on tera-scale telecommunication datasets," *IEEE Trans. Emerg. Topics Comput.*, vol. 7, no. 2, pp. 349–360, Apr./Jun. 2019, doi: 10.1109/TETC.2016.2627034.

[26] G. Chen, S. Hoteit, A. C. Viana, M. Fiore, and C. Sarraute, "Enriching sparse mobility information in call detail records," *Comput. Commun.*, vol. 122, pp. 44–58, Jun. 2018, doi: 10.1016/j.comcom.2018.03.012.

[27] E. Thuillier, L. Moalic, S. Lamrous, and A. Caminada, "Clustering weekly patterns of human mobility through mobile phone data," *IEEE Trans. Mobile Comput.*, vol. 17, no. 4, pp. 817–830, Apr. 2018, doi: 10.1109/TMC.2017.2742953.

[28] Y. Huai, A. Chauhan, A. Gates, G. Hagleitner, E. N. Hanson, O. O'Malley, J. Pandey, Y. Yuan, R. Lee, and X. Zhang, "Major technical advancements in apache hive," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2014, pp. 1235–1246.

[29] M. Rodrigues, M. Y. Santos, and J. Bernardino, "Big data processing tools: An experimental performance evaluation," *WIREs Data Mining Knowl. Discov*, vol. 9, no. 2, Mar. 2019, Art. no. e1297, doi: 10.1002/widm.1297.

[30] S. Ladymon. *LanguageManual DDL BucketetTables*. Accessed: 2016. [Online]. Available: https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL+BucketedTables

[31] A. Kushwaha, R. Das, and C. Sharma, "Trusted third party-based CDR management system," *J. Telecom., Electron. Comput. Eng.*, vol. 6, no. 1, pp. 43–46, 2014.

[32] T. Xu, "Storage and querying optimization for large scale structured data," Ph.D. dissertation, Dept. Comput. Sci. Technol., Tsinghua Univ., Beijing, China, 2016.

[33] J. Sun and T. Lu, "Optimization of column-oriented storage compression strategy based on HBase," in *Proc. Int. Conf. Big Data Artif. Intell. (BDAI)*, , Beijing, China, Jun. 2018, pp. 24–28, doi: 10.1109/BDAI.2018.8546673.

[34] M. Zhang, F. Liu, Y. Lu, and Z. Chen, "Workload driven comparison and optimization of hive and spark SQL," in *Proc. 4th Int. Conf. Inf. Sci. Control Eng. (ICISCE)*, Changsha, China, Jul. 2017, pp. 777–782, doi: 10.1109/ICISCE.2017.166.

XI PENG received the B.Sc. degree from the College of Mathematics and Informatics, South China Agricultural University, Guangzhou, China, in 2017. He is currently pursuing the master's degree with the Graduate Faculty of the China Academy of Telecommunication Technology, Beijing, China. His current research interests include big data analysis in communication systems and deep learning.

LIANG LIU received the M.S. degree from Sichuan University, Chengdu, China, in 2010. He is currently an Assistant Professor with the College of Cybersecurity, Sichuan University. His current research interests include big data analysis, malicious detection, network security, and artificial intelligence.

LEI ZHANG received the M.S. and Ph.D. degrees in computer science and technology from Sichuan University, Chengdu, Sichuan, China, in 2010 and 2015, respectively. He is currently an Assistant Researcher with the College of Cybersecurity, Sichuan University. His research interests include big data analysis, machine learning, and mobile security.