Efficiency Analysis of the access method with the cascading Bloom filter to the data warehouse on the parallel computing platform

Yu A Grigoriev, V A Proletarskaya, E Yu Ermakov and O Yu Ermakov
Bauman Moscow State University

e-mail: grigoriev@bmstu.ru, JK.Ermakov@gmail.com, vilka2000@mail.ru, iHelos.Ermakov@gmail.com

Abstract. A new method was developed with a cascading Bloom filter (CBF) for executing SQL queries in the Apache Spark parallel computing environment. It includes the representation of the original query in the form of several subqueries, the development of a connection graph and the transformation of subqueries, the definition of connections where it is necessary to use Bloom filters, the representation of the graph in terms of Spark. On the example of the query Q3 of the TPC-H test, full-scale experiments were carried out, which confirmed the effectiveness of the developed method.

1. Introduction
Relational databases (RDBs) still retain their leading position in many areas of data processing. But with the advent of technologies for processing large amounts of data (Big Data), the RDB's shortcomings began to manifest itself: limited scalability, the impossibility of storing aggregates in explicit form, etc. A new class of systems, known as NoSQL [1], appeared to overcome the limitations of RDBs.

NoSQL databases have high horizontal scalability (thousands of nodes) and eliminate some limitations of the relational model (they allow storing aggregates explicitly, there is no single database schema, etc.), but they do not support the conduct of transactions, which is important for financial systems. To overcome this lack of NoSQL, the NewSQL databases [2] appeared. They support a relational model, including transaction management, and at the same time have high horizontal scalability. But NewSQL inherits the disadvantages of the relational model: you can not store aggregates explicitly, you need to develop a single database schema.

Following NoSQL, platforms that support MapReduce (MR) technology [3] appeared. They provide parallel processing of queries (in particular, SQL queries) on large-scale clusters (Hadoop – 4000 nodes). Their advantage is also the large number of replicas and, consequently, the high reliability of processing huge amounts of data.

Currently, intensive work is underway to expand the capabilities of MapReduce technology. So, along with Hadoop, the new Spark software platform became very popular [4]. Like Hadoop, Spark is not just a separate project, but an entire ecosystem of various projects based on it.

There are several methods of accessing the data warehouse within the MapReduce / Spark framework, which differ in the number of tables in the query, the number of phases, the format of storing records, the distribution of data by nodes, and the use of early or late materialization
of search results. A critical analysis of the existing methods was carried out in papers [5–9]. However, these methods have a limited scope of applicability, and also have a direct dependence on the amount of node memory.

The article proposes a method for accessing a data warehouse with an arbitrary schema. It is based on the cascading Bloom filter (CBF) and allows you to implement a virtually arbitrary SQL query in the Spark environment.

2. Existing methods for applying the Bloom filter to a data warehouse with a star schema

The Bloom filter [10] is a specially constructed array of bits based on hashing the key values of the dimension table. The corresponding foreign key of each record of the fact table is processed by the Bloom filter [11]. This filter gives a "misfire" with a probability of $1 / 2^K$, where $K$ is the Bloom filter parameter. At a sufficiently large value of $K$, it is possible to achieve that the probability of false positive filtering would be very small. In this case, the number of entries participating in the connection is reduced, and in some cases it is not necessary to connect the dimension table to the fact table at all. This allows you to significantly reduce the processing time of requests.

The Bloom filter is already used in practice. Brito and others showed the effectiveness of using Bloom filters on the example of a data warehouse of the "star" type [6]. Compared to the methods implemented in MR (Fig. 1, rectangle) this method (SBFCJ – Spark Bloom-Filtered Cascade Join) wins by elapsed time and the amount of data transferred between the nodes (shuffled bytes) and additionally saved on disk (disk spill).

![Figure 1](image)

**Figure 1.** System response time, depending on the amount of data transferred across the network (a), and the amount of data stored on the disk due to lack of RAM (b) [6]

It is competing with the SBJ (Spark Broadcast Join) method, in which all the measurement tables are cached in RAM on each node (Fig. 1, circle). However, when using the SBJ method, strong fluctuations in the amount of used RAM are observed at sufficiently large storage sizes and small RAM volumes [6]. Figure 2 shows the dependencies of the query execution time Q4.1 from the Star Schema Benchmark (SSB) test [12] on the amount of RAM provided to one executor (Java machine).
Figure 2. Comparison the characteristics of SBJ and SBFCJ with 20 executors depending on the size of the RAM for one executor [6]: the query execution time and the average time for the task.

The experiment was performed with 20 executors. The SSB test database has a star schema (4 dimensions, 1 fact table); Query Q4.1 – a connection of 4 measurements and fact tables, grouping, sorting. The request was made to fill SF = 200: 1.2 billion records in the fact table (tf), the volume of tf – 130 GB. The cluster had the following characteristics: the number of nodes – 21 (1 master and 20 slaves), Hadoop 2.6.0 and Apache Spark 1.4.1, OS GNU / Linux installation (CentOS 5.7), each node – 2GHz AMD CPUs and 8GB RAM.

When the volume of RAM for one executor is less than 400 MB, there is a sharp increase in the execution time of the query and a separate task when using the SBJ method. Thus, the SBFCJ method is more stable when the volume of the RAM changes. Therefore, the Bloom filter was selected by the authors to develop a method for parallel processing a query to a data store with an arbitrary structure.

3. Implementation of the CBF method in SPARK

The developed method includes the following steps. 1) Representation of the original query in the form of several subqueries. Each subquery performs a search in one table. 2) Development of connection graph and subquery transformation. Identify the connections where you need to use Bloom filters. 3) Representation of a graph in the language of Spark. Let us illustrate this method with concrete examples.

Example 1. Take a fairly complex Q17 query from the TCP-H test, which includes a correlated select subquery (table 1, column 1). This query determines how much the annual revenue will be lost if the orders are not sold for small quantities of certain lots.

Table 1. Query Q17

| Subqueries | Query Q17 |
|------------|-----------|
| select sum(l_extendedprice)/7.0 as avg_yearly from lineitem, part where p_partkey = l_partkey and p_brand = '[BRAND]' and p_container = '[CONTAINER]' and l_quantity < ( select 0.2 * avg(l_quantity) from lineitem where l_partkey = p_partkey ); | Subquery P1: select p_partkey as pr1 from part where p_brand = '[BRAND]' and p_container = '[CONTAINER]'; |
Based on the description of the query Q17, we form the subqueries P1, L1, L2, which are presented in Table 1 (column 2). Figure 3 shows the connection graph and the subquery transformation. Here the following designations are accepted: rectangles designate the original tables; Rectangles with rounded corners – tables (RDD), which are formed by Spark after subqueries P1, L1, L2; Rectangles with dashed border – the results of joining tables of subqueries; Ovals – additional operations (filtering, aggregation). Arcs with arrows denote the initial data for the formation of tables or operations, arcs without arrows – the conditions of the connections of the corresponding tables. The circles indicate the Bloom filters (above them are the attributes for which these filters are built).

Executing the subquery P1 allows you to reduce the power of the pr1 attribute. Therefore, the use of the Bloom filters BF1 and BF2 here is appropriate.

From the description of the sequence of execution of the original query Q17, it becomes clear that Bloom compact filters are sent over the network, not the measurement tables. The join with the fact tables is performed only after Bloom filters are applied to them, i.e. with tables that are smaller in volume.

![Figure 3. Connection graph and subquery conversion for Q17](image_url)

Example 2. Table 2 contains a query Q3 (without a correlated subquery) from the TCP-H test and a description of its subqueries.

| Query Q3 | SubQuery |
|----------|----------|
| select l_orderkey, sum(l_extendedprice*(1-l_discount)) as revenue, o_orderdate, o_shippriority from customer C_, orders O_, lineitem L_ where c_mktsegment = 'SEGMENT' and c_custkey = o_custkey and l_orderkey = o_orderkey and o_orderdate < date 'DATE' and l_shipdate > date 'DATE' group by l_orderkey, o_orderdate, o_shippriority order by revenue desc, o_orderdate; | SubQuery C1: select c_custkey as kc1 from C_ where c_mktsegment = 'SEGMENT'; SubQuery O1: select o_custkey as kc1, o_orderkey as ko1, o_orderdate as dt1, o_shippriority as sh1 from O_ where o_orderdate < date 'DATE' and o_custkey=BF1; SubQuery L1: select l_orderkey as ko1, l_discount as d1, l_extendedprice as e1 from L_ where l_shipdate > date 'DATE' and l_orderkey=BF2; |

Figure 4 shows the join graph and the subquery transformation for Q3, constructed according to the rules given above.
In general, in each "star" can be present several tables. In this case, the original tables can be repeated in one or different snowflake nodes. For example, the table Lineitem in query Q17 is repeated in two nodes (see Table 1, column 1).

The sequence of transformations and actions in Spark forms a DAG (Directed Acyclic Graph) for the execution of the query. In the general case, the plan can be complex and have the form of an acyclic graph. It works as a pipeline, in parallel "pushing" the partitions (partitions) of RDD (fragments of tables that are stored in the nodes of the cluster) between the nodes of the graph. Figure 5 shows the plan for the "star" scheme, which can act as a measurement in another "star". The track for the promotion of the sections has the following form (in figure 5, the numbers of stages are indicated in figures).

Step 1. Reading the change tables Di, filtering the records, obtaining the projection (k, s).

Step 2. Keeping keys k on the local disk (before shuffle), collecting measurement keys on one Executor (for each measurement), building Bloom filters (for each measurement), broadcasting Bloom filters for all Executors where the fact table filtering (FT ). This is the synchronization point for all previous processes.

Step 3. Reading the fact table, filtering records with condition F and using Bloom filters, obtaining the projection ({fk}, k, s).

Step 4. Serial connection of the filtered fact table with the filtered tables of measurements (Join i). Grouping and sorting, if it is provided for this "star". Go to the next "star".

Figure 4. Connection graph and subquery conversion for Q3

Figure 5. DAG
4. Experiment description and the result analysis

Virtual cluster was deployed to carry out an experimental comparison of the developed CBF method and the method without using the Bloom filter (NBF). The cluster consisted of 8 nodes, one of which was installed HDFS management services, Hive, Spark 2, Yarn. The main characteristics of each virtual node were as follows: one dual-core processor, 8 GB of RAM, 200 GB SSD disk, Ubuntu OS 14.16.

| NBF                  | Time (min.) execution, SF=500, N=40 ml. |
|----------------------|----------------------------------------|
| 0                    | Reading C_ and Bloom filter bloom1     |
|                      | (Stage0) creating                      |
| 1                    | Reading O_ and bloom1 (Stage1)         |
|                      | using                                  |
| 2                    | map C_ \Join O_ (Stage2)               |
|                      | creating                               |
| 3                    | map C_ \Join O_ and bloom2 (Stage3)    |
|                      | creating                               |
| 6                    | Reading L_ and bloom2 (Stage7)         |
|                      | using                                  |
| 8                    | L_ \Join (C_ \Join O_), group by, order|
|                      | by (Stage8)                            |
| SUM: 0,4+1,5+0,6+7,3+1,2=11 min |                                        |

Figure 6. NBF stages

First consider the results of Q3 query execution using the NBF method (see Table 2, column 1). Figure 6 shows the stages of the NBF and the time of their implementation in Spark for SF = 500 (this corresponds to approximately 400 GB of processed data). The \Join mark indicates the table join operation.

You can see that reading the source tables L_, C_, O_ is done in parallel. The query execution time was 25 minutes.

Figure 7 shows the characteristics of the NBF task execution stages (SF = 500). The executor is the Java virtual machine (JVM) on which the stage tasks are performed. By default, the number of CPU cores per one instance of the executor is 1 (parameter spark.executor.cores).

Figure 7. characteristics of the NBF task execution stages (SF = 500)

On fig. 7 arrows shows that the reading of the original tables (stages 0 ÷ 2) is performed in parallel on 13 executors (13 CPU cores on 7 nodes). The average number of tasks performed by one core, in this case is (455 + 24 + 100) / 13 = 44.5. The analysis of the task execution time quantiles shows that the stages 0 ÷ 2 have a long right tail of the distribution function, and the tasks of stages 4, 9, 10 have a long left tail. You can also notice that "garbage collection" and writing to disk before "Shuffle Write" does not have a significant impact on the execution time of the request.
Figure 8. Data volumes and the number of reads from the HDFS

Figure 8 shows the data volumes and the number of reads from the HDFS (Input) file system read from the local disks and transmitted over the virtual network during the "shuffle" stage when the Reduce (Shuffle Read) tasks written to local disks before the "Shuffle" by all the tasks of the corresponding stages. The arrows indicate the components of the corresponding volumes of Shuffle Read. It should be noted the high compression ratio of the original tables. On the average, it amounted to (see Tables 1 and 3) \( 500 \times (26 + 149 + 641) / (32.1 \times 1024) = 12.4 \).

Now consider the results of the same query Q3, but using CBF developed method (see Table 2, column 2). Fig. 9 shows the step (stage) CBF time and their implementation in Spark for SF = 500, the number of elements in the filter Bloom N = 40 million or probability of a false positive response filter equal to P = 0.01 [11]. Stage2 run on the task map, which scan set RDD "customer" (in MemoryTableScan RDD), has created Stage0 stage. Thus some data read from HDFS (0.079Gb / 1501 see Fig. 8 – … Is not C_ table). These tasks are placed recording RDD "customer" on the local drive (Shuffle Write) for further "shuffling" and compounds with a table at O_ Stage3 step. Tasks Stage1 and Stage2 are performed in parallel. Similar actions Spark performs on stage with Stage6 C1O1 table. So tasks Stage6 and Stage7 are performed in parallel too. Query execution time was 11 min, which is a \( 25/11 = 2.3 \) times smaller than the same query without using the Bloom filter (see Fig. 6).

Figure 9. CBF Stages

Figure 10 shows the characteristics of the CBF tasks by performing steps (SF = 500, N = 40 million, P = 0.01). The arrow indicates that the stages 1 and 2 are performed in parallel.
on 13 executors. The average number of tasks performed by a single core, in this case is \((100 + 24) / 13 = 9.5\). Stages 6 and 7 are also performed in parallel.

![Figure 10. Characteristics of the CBF tasks](image)

Figure 10. Characteristics of the CBF tasks

Figure 11 shows the amount of data and number of entries Input, Shuffle Read and Shuffle Write. It should be noted that the volumes Shuffle Read and Shuffle Write using the method CBF smaller compared with NBF respectively 10.5 and 5.7 times (see Table 4).

![Figure 11. Characteristics of the CBF tasks](image)

Figure 11. Characteristics of the CBF tasks

Fig. 12 shows the influence of the parameter \(N\) (mill) Bloom filter (the number of elements in the bitmap) to the performance indicators Q3 CBF query.

![Figure 12. Shuffle write amount (a) and query execution time (b) on number of Bloom filter elements](image)

Figure 12. Shuffle write amount (a) and query execution time (b) on number of Bloom filter elements
Volume Shuffle Read (Shuffle Write) decreases substantially at increase N up to 30 million for SF=250 and up to 40 million for SF=500. Further growth N does not lead to a significant reduction of volume. This is due to the reduction in the number of false positives Bloom filter. Time shown on the fig. 8 b, is random (impact that cluster deployed in a virtual environment). But when SF = 500 is observed a tendency to reduce the query time until N = 40 Mill. In this experiment, the influence of N on the execution time of the request is lower than the amount of data on the "shuffling": for SF=500 at increase N from 15 million to 40 million volume decreased in $21.4/5.11=4.2$ times, and time – only in $16.6/11=1.5$ times. This is due to the fact that the network between the nodes is a virtual, i.e. "Fast" (some components reside on the same physical server).

![Figure 13. Shuffle Read Amount on SF](image)

**Figure 13.** Shuffle Read Amount on SF

Figure 13 shows that the amount of intermediate data stored on the local disk and transmitted over the network in the process of “dicing” (Shuffle Read), is significantly higher for the method NBF than CBF.

Method CBF loses to method NBF on the time performances of the enquiry (fig. 10). Up to SF=250 NBF falls behind slightly CBF. But at SF=500 the advantage of the CBF is evident.

![Figure 14. Query Execution time on SF](image)

**Figure 14.** Query Execution time on SF

5. Conclusion

Spark drivers for the Q3 query of TPC-H test have been development both with and without Cascading Bloom Filter.
A series of full-scale experiments was carried out on a cluster of 7 work units implemented in the cloud architecture, in order to compare the methods of the CBF and NBF access to the data store by the example of Q3.

Using the developed CBF method, the amount of intermediate data stored on the disk and transmitted over the network when the query is executed is much smaller. So for $SF = 500$ these volumes differ by an order of magnitude (see Fig. 13). When executing a query, the load on the disk and the network for the CBF method is less. CBF wins by the time the query is executed. For $SF = 500$, the CBF method is 2.3 times better than the CBF over time (see Fig. 14). It is expected that the effect will be even higher if the cluster would not be implemented in a virtual environment.

It is shown that the $N$ parameter of the Bloom filter significantly affects the performance of the query using the CIBF method (see Fig. 12).

It is planned to develop a common cost model for the CBF access method and use the measurement results presented in this article to adapt the model and assess its adequacy.

References

[1] Venkat Goodwada, Dana Rao, Vijay Raghavan 2016 Renaissance DBMS: the problem of choosing Open Systems. DBMS 3 p 12–17
[2] Neljubin D and Kosjanenko G 2014 What's new in NEWSQL? Immersion in the latest databases Hacker 8 (187) p 127–131
[3] Dean J and Ghemawat S 2004 MapReduce: Simplified data processing on large clusters The Sixth Conference on Operating System Design and Implementation, Berkeley, CA p 10
[4] Riza S, Leserson Wu, Owen Sh and Wills D 2016 Spark for Professionals: Modern Patterns for Large Data Processing (St. Petersburg: Peter)
[5] Ernakov E Yu and Proletarskaya V A 2016 Method of access to the data store using MapReduce / Spark technology without caching of measurement tables in RAM Information and measuring systems 12 p 90–97
[6] Jaqueline Joice Brito, Thiago Mosquero, Ricardo Rodrigues Ciferri and Cristina Dutra de Aguiar Ciferri 2016 Faster cloud Star Joins with reduced disk spill and network communication ICCS 2016. The International Conference on Computational Science 80 p 74–85
[7] Grigoriev Yu A and Proletarskaya V A 2015 Method of early materialization of access to the repository using MapReduce technology Informatics and control systems 3 p 3–16
[8] Grigoriev Yu A and Proletarskaya V A 2016 Comparison of methods for processing requests to the repository using MapReduce technology Informatics and control systems 1 p 3–13
[9] Zhou G, Zhu Y and Wang G 2013 Cache Conscious Star-Join in MapReduce Environments. Cloud-I '13 Proceedings of the 2nd International Workshop on Cloud Intelligence p 1
[10] Bloom B H 1970 Space/time trade-offs in hash coding with allowable errors Communications of the ACM 13 p 422–426.
[11] Grigoriev Yu A, Proletarskaya V A and Ermakov E Yu 2017 Method of access to the data warehouse using SPARK technology with cascading Bloom filtering Informatics and control systems 1 p 3–14
[12] O’Neil P et al 2009 The star schema benchmark and augmented fact table indexing Technology Conference on Performance Evaluation and Benchmarking p 237–252