ELEKTROTEHNIŠKI VESTNIK 88(5): 290-298, 2021
REVIEW SCIENTIFIC PAPER

Big Data Optimization Using Hive

Vedrana Nerić[1], Nermin Sarajlić[2]

[1] *Virgin Pulse, Tuzla, Bosnia and Herzegovina*
[2] *Faculty of Electrical Engineering, University of Tuzla, Bosnia and Herzegovina*
E-mail: vedrana_neric@yahoo.com

Abstract. Traditional database management systems can not deal with big data challenges and too large and complex datasets. Big data covers a wide area of research and work with and on data – large amounts of data, new technologies for storing and processing data as well as changes in the interpretation of data. Apache Hadoop is a platform that was invented to manage big data. Apache Hive is a query and analysis engine built on top of Hadoop. Although Hadoop/Hive can process any amount of data, optimization can significantly improve the processing time and cost. The main idea of the paper is to analyze some possible big data optimization techniques for improving the query performance on Hadoop using Hive.

Keywords: big data, Hadoop, SQL-on-Hadoop, Hive, optimization

Optimizacija velikih podatkov s pomočjo Hive

Tradicionalni sistemi za upravljanje baz podatkov težje analizirajo velike količine podatkov. Velike količine podatkov pokrivajo široko področje raziskav, ki vključujejo tudi nove tehnologije za shranjevanje in obdelavo podatkov ter spremembe v njihovi interpretaciji. Apache Hadoop/Hive je platforma za upravljanje velike količine podatkov. Čeprav lahko Hadoop/Hive obdeluje poljubno količino podatkov, z optimizacijo podatkov znatno izboljšamo čas obdelave in posledično stroške. V prispevku so predstavljeni različni pristopi k optimizaciji obdelave velike količine podatkov.

1 INTRODUCTION

The big data is a field related to dealing with data sets that are too large or too complex to be handled by traditional relational database management systems (RDBMS). The big data is a high volume, high velocity, and/or high variety of information assets that require new forms of information processing to enable an enhanced insight discovery, decision making, and process optimization [1]. The big data can be described with the following characteristics [2]-[5]:

- Volume – the large amount of the generated and stored data; the size of the big data is today in the rank of even exabytes and zettabytes,
- Velocity – the high speed at which the data is generated and speed at which that data needs to be processed,
- Variety – the diversity of the available data in different sources and forms: structured, semi-structured and unstructured,
- Veracity – the accuracy of a data set that is equivalent to the data quality,
- Value – the ability to transform the data into a business value,
- Variability – the consistency of the data in terms of availability or reporting period,
- Viscosity – the speed element used to describe the delay time in the data relating to the event being described,
- Virality – the data spread rate that describes how often the data is collected and repeated by other users or events.

The paper provides an analysis of some big data optimization techniques using Hive on a platform for the distributed data storage Hadoop. The paper is organized as follows. After the introduction, Apache Hadoop is described in Chapter 2, Apache Hive in Chapter 3 and Cloudera Distribution Hadoop (CDH) in Chapter 4. Chapter 5 describes some of the Hive big data optimization techniques: cost-based optimization, statistics, predicate pushdown, parallel execution, partitioning, bucketing, join types (common join, skewed join, map join, bucket join). Chapter 6 provides concluding remarks based on the analysis and testing of the big data optimization techniques using Hive on concrete query examples.

2 APACHE HADOOP

One way to store the big data in a distributed environment and parallel processing is Hadoop. Hadoop is an Apache open-source software framework for distributed processing and querying large datasets. Hadoop is made up of the following main elements:

Received 17 July 2021
Accepted 5 November 2021

- Hadoop Distributed File System (HDFS) – file system to store the big data at multiple nodes in the cluster,
- MapReduce – programming model for parallel computing of the big data,
- Hadoop Common – configuration files and libraries required by other Hadoop modules,
- YARN – framework for job scheduling and cluster resource management [6]-[11].

Most of the tools and solutions are used to support the main Hadoop elements and provide services such as storage, analysis, and maintenance of the data. Hadoop components that together form a Hadoop ecosystem are: HDFS; YARN (Yet Another Resource Negotiator); MapReduce (data processing using programming); Spark (in-memory data processing); Pig, Hive (data processing using query – SQL-like); HBase (NoSQL database); Mahout, Spark MLlib (machine learning algorithm libraries); Solar, Lucene (searching and indexing), Zookeeper (managing cluster); Oozie (job scheduling); Apache Drill (SQL-on-Hadoop); Flume, Sqoop (data ingesting services); Ambari (provision, monitor and maintain cluster). [12]-[15]

3 APACHE HIVE

Apache Hive is a data warehouse software that enables writing SQL-like queries to efficiently extract the data from Apache Hadoop. Traditional SQL queries must be implemented in the MapReduce Java API to execute over the distributed data. To bypass writing Java and simply access data using SQL-like queries, Facebook developed Apache Hive data warehouse. Apache Hive is initially developed by Facebook, but it is also used and developed by other companies (Netflix, Financial Industry Regulatory Authority). For querying the data stored in a Hadoop cluster, Hive Query Language (HiveQL) can be used. Apache Hive translates the input program written in the HiveQL language to one or more Java MapReduce, Tez, or Spark jobs that can run in Hadoop YARN. Apache Hive organizes the data into tables for HDFS and runs the jobs on a cluster to produce an answer. [16]-[19]

Some of the Hive features [17]:

- Hive provides a simpler query model with less coding than MapReduce,
- HiveQL and SQL have a similar syntax,
- Hive provides lots of functions that lead to an easier analytics usage,
- The response time is typically much faster than other types of queries on the same type of huge datasets,
- Hive supports running on different computing frameworks,
- Hive supports the ad hoc querying data on HDFS,
- Hive supports the user-defined functions, scripts, and a customized I/O format to extend its functionality,
- Hive is scalable and extensible to various types of the data and bigger datasets,
- Matured JDBC and ODBC drivers allow many applications to pull the Hive data for seamless reporting,
- Hive allows users to read the data in arbitrary formats, using SerDes and I/O formats,
- Hive has a well-defined architecture for the metadata management, authentication, and query optimizations,
- There is a big community of practitioners and developers working on and using Hive.

The major Hive components are [20]:

- Hive Client – allows writing applications using different types of clients such as thrift server, JDBC driver for Java, and applications that use the ODBC protocol.
- Hive Services
  - User Interface – enables external users to interact with Hive by submitting queries, instructions and monitoring the process status. Hive Web UI, command line interface (CLI), and Hive HD Insight (in windows server) are supported by the user interface.
  - Hive Driver – a component which receives queries from different sources and clients like the thrift server, JDBC, and ODBC using the Hive Server and directly from Hive CLI and Web UI, and after receiving the queries, it transfers them to the compiler.
  - Compiler – a component that parses the query, does semantic analysis on the query blocks and query expressions, and eventually generates an execution plan with the help of the table and partition the metadata looked up from the metastore. The compiler converts the query to an abstract syntax tree (AST). After checking for the compatibility and compiled time errors, it converts AST to a directed acyclic graph (DAG). DAG divides operators into MapReduce stages and tasks based on the input query and data.
  - Optimizer – performs transformations on the execution plan to get an optimized DAG.
  - Execution Engine – a component which executes the execution plan created by the compiler. The plan is a DAG of stages. The execution engine manages the dependencies between different stages of the plan and executes these stages on appropriate system components.
- Hive Storage (Metastore) – stores the metadata about the database like a scheme of the table, location in the HDFS, data types of the columns, etc.

4 CLOUDERA DISTRIBUTION HADOOP

Cloudera Inc. is an American software company that provides a software platform for data engineering, data warehousing, machine learning, and analytics. The most popular distribution of Hadoop is the Cloudera open-source platform CDH (Cloudera Distribution Hadoop). It is a solution for batch processing, interactive SQL, interactive search, and continuous availability. CDH delivers the core elements of Hadoop scalable storage and distributed computing with additional components such as the user interface, plus necessary enterprise capabilities such as security, and integration with a broad range of hardware and software solutions. The Cloudera Manager is an end-to-end application for managing the CDH clusters. It enables fast, easy, and secure deployment, monitoring, alerting, and management of the Cloudera platform. The Cloudera Manager provides a granular visibility into and control over every part of the CDH cluster. [21]

Apache Scoop is a part of CDH and a tool that easily transfers the structured data from RDBMS into HDFS, while preserving the structure. The Scoop job can be launched in terminal:

```
[cloudera@quickstart ~]$ sqoop import-
all-tables \
    -m 1 \
    --connect
jdbc:mysql://quickstart:3306/retail_db \
    --username=retail_dba \
    --password=cloudera \
    --compression-codec=snappy \
    --as-parquetfile \
    --warehouse-dir=/user/hive/warehouse
\
    --hive-import
```

The command is launching the MapReduce jobs to pull the data from the MySQL database and write the data to HDFS, distributed across the cluster in the Apache Parquet format, and also it creates tables to represent the HDFS files in Impala/Apache Hive with a matching scheme. Parquet is a format designed for analytical applications on Hadoop to optimize the data storage and retrieval. Instead of grouping the data into rows, it groups the data into columns. [22]

End users can interact with the data warehouse using Hue. Hue is a query editor web application. Hue runs in a browser and provides an easy-to-use interface to several applications to support an interaction with Hadoop. Using Hue, any of the following tasks can be performed [23]:

- query Hive data stores,
- create, load, and delete the Hive tables,
- work with the HDFS files and directories,
- create, submit, and monitor the MapReduce jobs,
- manage users and groups.

5 HIVE BIG DATA OPTIMIZATION

Although Hive is built to deal with the big data, the query performance is still very important. Most of the time, Hive can rely on the smart query optimizer to find the best execution strategy as well as the default settings of configuration parameters. However, for an efficient data processing and query execution, optimization needs to be done. In this chapter, some of the Hive query optimization techniques will be discussed. [24]

5.1 Cost-based optimization and statistics

Hive optimizes each query's physical and logical execution plan before submitting it for the final execution, but optimization techniques are not based on the cost of the query. The cost-based optimization (CBO) is a new feature and a core component in the Hive query processing engine. CBO offers a better Hive query performance regarding the cost, resulting in different decisions: which types of joins to perform, how to order joins, degree of parallelism, etc. To use CBO, the following properties (Table 1) should be set at the beginning of the query. [25], [26]

Table 1. CBO configuration parameters [27]

Configuration Parameter	Setting	Description
`hive.cbo.enable`	`true`	Enables cost-based query optimization.
`hive.stats.autogather`	`true`	Enables automated gathering of table level statistics for newly created tables and table partitions, such as tables created with the `INSERT OVERWRITE` statement. The parameter does not produce column level statistics, such as those generated by CBO. If disabled, administrators must manually generate the table level statistics for newly generated tables and table partitions with the `ANALYZE TABLE` statement.
`hive.stats.fetch.column.stats`	`true`	Instructs Hive to collect column level statistics.
`hive.stats.fetch.partition.stats`	`true`	Instructs Hive to collect partition level statistics.
`hive.compute.query.using.stats`	`true`	Instructs Hive to use statistics when generating query plans.

Data preparation for CBO is done by running `ANALYZE` command to collect various statistics on the tables for which CBO will be used. The Hive statistics are a data collection, such as the number of rows, number of files, number of partitions if the table is partitioned, and row data size in bytes, that describe more details of the objects in the Hive database. Statistics is a metadata of the Hive data and an input to

the cost-based optimizer that will pick the query plan with the lowest cost in terms of the system resources required for the query completion. [17], [28]

The syntax for ANALYZE command is [29]:

```
ANALYZE TABLE [db_name.]tablename
[PARTITION (partcol1[=val1],
partcol2[=val2],...)]
COMPUTE STATISTICS
  [FOR COLUMNS]
  [CACHE METADATA]
  [NOSCAN];
```

Q1 – query example for the top ten most popular product categories:

```
SELECT c.category_name,
COUNT(order_item_quantity) AS count
FROM order_items oi
INNER JOIN products p ON
oi.order_item_product_id = p.product_id
INNER JOIN categories c ON
c.category_id = p.product_category_id
GROUP BY c.category_name
ORDER BY count DESC
LIMIT 10;
```

Table 2. CBO execution results

CBO	Setting	Q1 Execution time (s)
Off	`SET hive.cbo.enable = false; SET hive.compute.query.using .stats = false;`	49.7
On	`SET hive.cbo.enable = true; SET hive.compute.query.using .stats = true; SET hive.stats.fetch.column. stats = true; ANALYZE TABLE order_items COMPUTE STATISTICS; ANALYZE TABLE products COMPUTE STATISTICS; ANALYZE TABLE categories COMPUTE STATISTICS;`	41.1

5.2 Predicate pushdown

The predicate pushdown is a traditional RDBMS term, whereas, in Hive, it works as a predicate pushup. For the query performance optimization it is important to execute expressions like filters as early as possible. The predicate pushdown is enabled by setting the following property: `SET hive.optimize.ppd = true;`. When executing a query in a basic manner, filtering happens very late in the process. A significant performance improvement can be provided by moving filtering to an early phase of the query execution and in that way, non-matches can be eliminated earlier, and the cost of processing can be saved at a later stage. The predicate pushdown is important for minimizing the amount of the data scanned and processed by an access method, as well as reducing the amount of the data passed into Hive for a further query evaluation. [30], [31]

Q2 – query example for calculating the total revenue per product and showing the top ten revenue generating products:

```
SELECT p.product_id, p.product_name,
r.revenue FROM products p INNER JOIN
(SELECT oi.order_item_product_id,
 SUM(cast(oi.order_item_subtotal AS
 float)) AS revenue
 FROM order_items oi
 INNER JOIN orders o ON
 oi.order_item_order_id = o.order_id
 WHERE o.order_status <> 'CANCELED'
 AND o.order_status <> 'SUSPECTED_FRAUD'
 GROUP BY order_item_product_id) r
ON p.product_id = r.order_item_product_id
ORDER BY r.revenue DESC
LIMIT 10;
```

Table 3. Predicate pushdown execution results

PPD	Setting	Q2 Execution time (s)
Off	`SET hive.optimize.ppd = false;`	79
On	`SET hive.optimize.ppd = true;`	68

5.3 Parallel execution

The Hive queries are commonly translated into several stages (MapReduce stage, sampling stage, merge stage, limit stage, etc.) that are executed by the default sequence, one after the other. These stages are not always dependent on each other and can run in parallel to save the overall job running time. A parallel execution can be enabled with the following setting `hive.exec.parallel` to `true` (default `false`) and the expected number of the jobs running in parallel `hive.exec.parallel.thread.number` can be set (default 8). [32], [33]

Q3:

```
SELECT p.product_id, p.product_name,
SUM(cast(oi.order_item_subtotal AS
float)) AS revenue FROM
(SELECT order_item_product_id,
 order_item_order_id, order_item_subtotal
 FROM order_items) oi INNER JOIN
(SELECT order_id FROM orders WHERE
 order_status <> 'CANCELED' AND
 order_status <> 'SUSPECTED_FRAUD') o
ON oi.order_item_order_id = o.order_id
INNER JOIN
(SELECT product_id, product_name FROM
 products) p ON p.product_id =
oi.order_item_product_id
GROUP BY p.product_id, p.product_name;
```

Table 4. Parallel execution results

Parallel Execution	Setting	Q3 Execution time (s)
Off	`SET hive.exec.parallel = false;`	58.94
On	`SET hive.exec.parallel = true;` `SET hive.exec.parallel.thread.number = 16;`	47.97

5.4 Partitioning and bucketing

Partitioning in Hive is a very effective method for improving the query performance on large tables. It is a way of dividing a table into related parts based on the values of a particular column. Using partitions, the data is stored in subdirectories on HDFS, and it is easy to do queries on slices of the data. In this way, the query execution time is reduced because of looking at the required partition only instead of querying the entire dataset. Some commonly used dimensions as partitions keys are partitions by the date and time, locations, and business logic. [34], [35]

Here is the syntax for creating partitions in a Hive table:

```
CREATE TABLE table_name (column1
datatype, column2 datatype,…)
PARTITIONED BY(partition1 datatype,
partition2 datatype,…);.
```

There are two ways of creating partitions in a table [18]:

- Static Partitioning (default) – the data must be inserted in different partitions of a table manually. While creating static partitions, it should be specified for which value a partition will be created.
  - For inserting the data from a file to a Hive table in specified partitions, the `LOAD` command can be used. If there are more than one partition columns in a table, the values for all partitioning columns should be specified:

    ```
    LOAD DATA [LOCAL] INPATH 'filepath'
    [OVERWRITE] INTO TABLE tablename
    [PARTITION(partcolumn1=value1,
    partcolumn2=value2 ...)]
    ```

  - For inserting the data from a query result of another Hive table, the `INSERT` command can be used. The `INSERT OVERWRITE` statement will insert the data into a partition and it will overwrite the existing data of that partition:

    ```
    INSERT OVERWRITE TABLE tablename1
    [PARTITION (partcolumn1=value1,
    partcolumn2=value2 ...)]
    select_statement1 FROM
    from_statement;
    ```

  - The `INSERT INTO` statement will insert the data into a partition and it will not delete any existing data of that partition and will append the new data to that partition:

    ```
    INSERT INTO TABLE tablename1
    [PARTITION (partcolumn1=value1,
    partcolumn2=value2 ...)]
    select_statement1 FROM
    from_statement;
    ```

- Dynamic Partitioning – while inserting the data, the values for partition columns do not have to be specified in the `PARTITION` clause. Only the name of the partition columns should be specified, and the partitions will be created based on the unique values of that partition column. The dynamic partition columns must be specified in the last among the columns in the `SELECT` statement and in the same order in which they appear in the `PARTITION` clause. Dynamic Partitioning can be enabled by setting the following properties:

  ```
  SET hive.exec.dynamic.partition = true;
  SET hive.exec.dynamic.partition.mode =
  nonstrict;
  ```

Partitioning is efficient for increasing the query performance only if there is a limited number of partitions. Partitioning will not perform well on a column with a large number of unique values where there will be a large number of partitions. To overcome the partitioning problem, Hive provides bucketing. Similar to partitioning, bucketing organizes the data into separate files in HDFS. The bucketing concept is based on the hashing principle, where the same type of the keys is always sent to the same bucket. Bucketing can be enabled by setting the following property [18]:

```
SET hive.enforce.bucketing = true;
```

Here is the syntax for dividing the Hive table into buckets:

```
CREATE TABLE table_name(column1 datatype,
column2 datatype,…)
PARTITIONED BY (partition1 datatype,
partition2 datatype,…)
CLUSTERED BY (column1, column2,…) INTO
num BUCKETS;.
```

Example:

```
SET hive.exec.dynamic.partition = true;
SET hive.exec.dynamic.partition.mode =
nonstrict;

CREATE TABLE customers_part_state(
customer_id int,
customer_fname string,
customer_lname string,
customer_email string,
customer_password string,
customer_street string,
customer_city string,
customer_zipcode string)
PARTITIONED BY (customer_state string)
ROW FORMAT DELIMITED FIELDS
TERMINATED BY ',';
```

```
INSERT INTO customers_part_state
partition(customer_state)
SELECT customer_id, customer_fname,
customer_lname, customer_email,
customer_password, customer_street,
customer_city, customer_zipcode,
customer_state
FROM customers;
```

Q4:

```
SELECT * FROM customers
WHERE customer_state = 'PR';
```

Q5:

```
SELECT * FROM customers_part_state
WHERE customer_state = 'PR';
```

Q6:

```
SELECT SUM(cast(oi.order_item_subtotal AS
float)) AS total FROM order_items oi
INNER JOIN orders o ON o.order_id =
oi.order_item_order_id
INNER JOIN customers c ON c.customer_id =
o.order_customer_id
WHERE c.customer_state = 'PR'
AND o.order_status <> 'CANCELED'
AND o.order_status <> 'SUSPECTED_FRAUD';
```

Q7:

```
SELECT SUM(cast(oi.order_item_subtotal AS
float)) AS total FROM order_items oi
INNER JOIN orders o ON o.order_id =
oi.order_item_order_id
INNER JOIN customers_part_state c ON
c.customer_id = o.order_customer_id
WHERE c.customer_state = 'PR'
AND o.order_status <> 'CANCELED'
AND o.order_status <> 'SUSPECTED_FRAUD';
```

Table 5. Partitioning execution results

Partitioning	Query	Execution time (s)
Off	Q4, `customers` table without partition	17.56
On	Q5, `customers_part_state` with partition `customer_state`	0
Off	Q6, `customers` table without partition	70
On	Q7, `customers_part_state` with partition `customer_state`	58.5

5.5 Joins

A join in Hive is used for the same purpose as in the traditional database systems. It is used to combine and fetch the data from multiple tables based on a common value or field. `JOIN` is performed whenever multiple tables are specified inside the `FROM` clause of the statement. [18]

5.5.1 Common join

The default join type in Hive is a Common Join, which is also called Distributed Join, or Shuffle Join, or Reduce Side Join, or Sort Merged Join. This join has a complete cycle of MapReduce. With a Common Join, all rows from the joined tables are distributed to all nodes based on the join keys and values from the same join keys end up on the same node.

When performing a normal join, the job is sent to a MapReduce task which splits the main task into two stages: map stage and reduce stage. The map stage interprets the input data and returns the output to the reduce stage in a form of the key-value pairs. The next goes through a shuffle stage where they are sorted and merged. The reducer gets the sorted data and completes the join job.

A Common Join works with tables of any size but performs poorly when the data is skewed. If the join keys have a large proportion of the data, the corresponding reducers will be overloaded. When the majority of the reducers have completed the join operation while a few reducers are still running there will be a typical skewed data issue. A Common Join can be identified when using the `EXPLAIN` command. A Join Operator can be seen just below Reduce Operator Tree. [36], [37]

5.5.2 Skewed join

The Skewed Join is helpful when a table is skewed (a table that is having values that are present in large numbers in the table compared to the other data). The skew data is stored in a file while the rest of the data is stored in a separate file. The Skewed Join targets the skewed data issue when the query waits for the longest running reducers on the skewed keys while the majority of the reducers complete the join operation. At the runtime, it scans the data and detects the keys with a large skew, which is controlled by the `hive.skewjoin.key` parameter (100000 by default), and stores those keys in the HDFS directory temporarily instead of processing. Then these skewed keys are processed in a MapReduce job and that would be much faster since it would be a Map Join. A Skewed Join can be enabled with the following parameter: `SET hive.optimize.skewjoin = true;` and can be identified when using the `EXPLAIN` command, `handleSkewJoin:true` can be seen below the Join Operator and Reduce Operator Tree. [38], [39]

5.5.3 Map join

A Map Join, also called an Auto Map Join, or Map Side Join, or Broadcast Join, is efficient when one of the join tables is small enough so that it can be loaded into the memory and a join is performed in the map phase of the MapReduce job. A Map Join is much faster than a regular join because there is no involved reducer. Hive

can convert a Common Join into a Map Join based on the input file size with the following setting: `SET hive.auto.convert.join = true;`. During the join, the determination of the small table is controlled by the `hive.mapjoin.smalltable.filesize` parameter, that is by default 25MB.

With a Map Join, before the original MapReduce task, a local MapReduce task is created. It reads the data of the small table from HDFS and saves it into an in-memory hash table and then into a hash table file. When the original join MapReduce task starts, it moves the hash table file to the Hadoop Distributed Cache, which will populate the file to each mapper local disk. All the mappers can load this hash table file into the memory and then do the join in map stage. For example, for a join with big table A and small table B, for every mapper for table A, table B is read completely. As the smaller table is loaded into the memory and then a join is performed in the map phase of the MapReduce job, no reducer is needed, and the reduce phase is skipped. The Map Join is faster than the regular default join and can be identified when using the `EXPLAIN` command, a Map Join Operator can be seen just below the Map Operator Tree.

The query using a Map Join can be specified with a hint. The general syntax for a Map Join is as follows:

```
SELECT /*+ MAPJOIN(table2) */ column1,
column2, column3
FROM table1 [alias_name1]
JOIN table2 [alias_name2] ON
table1 [alias_name1].key =
table2 [alias_name2].key
```

where: `table1`: is the bigger or larger table, `table2`: is the smaller table, `[alias_name1]`: is the alias name for `table1`, `[alias_name2]`: is the alias name for `table2`. [40]-[43]

5.5.4 Bucket join

The Bucket Join is used when all join tables are large and the table data has been distributed by the join key. It is also called the Collocated Join. The Bucket Join is a special type of the Map Join applied on the bucket tables. The join tables must be the bucket tables, join on the buckets columns, and the bucket number in bigger tables must be a multiple of the bucket number in the small tables. If one table has 2 buckets, then the other table must have either 2 buckets or a multiple of 2 buckets (2, 4, 6, etc.). In this case, the efficiency of the query is improved because the join can be done at the only mapper side, only the required buckets are fetched, not the complete table, and only the matching buckets of all small tables are replicated onto each mapper. Otherwise, a normal inner join is performed. The following properties need to be set to true for the query to work as a Bucket Join: `SET hive.optimize.bucketmapjoin = true; SET hive.optimize.bucketmapjoin.sortedmerge = true;`. The Bucket Map Join can be identified when using the `EXPLAIN` command, Sorted Merge Bucket Map Join Operator can be seen below the Map Operator Tree. [44]

5.5.5 Join order

The query performance is affected by the order of the join tables, because of the generated intermediate data sets. The number of the possible join orders increases exponentially with the number of the involved tables. It is not possible to evaluate the execution cost of each join order, but the aim is to find the join order with a maximum reduction of the intermediate rows generated. A query execution can be accelerated if the least amount of the data, that are to be worked on, is identified early enough. [45]

5.5.6 Join examples

Table 6. Join execution results

Join	**Setting**	**Description using** `EXPLAIN`	**Q2 Execution time (s)**
Common	`SET hive.auto.convert.join = false;`	`… Reduce Operator Tree: Join Operator condition map: Inner Join 0 to 1 …`	78
Map	`SET hive.auto.convert.join = true;`	`… Map Operator Tree: Map Join Operator condition map: Inner Join 0 to 1 …`	67

Q8:

```
SELECT p.product_id, p.product_name,
SUM(cast(oi.order_item_subtotal AS
float)) AS total FROM products p
INNER JOIN order_items oi ON
oi.order_item_product_id = p.product_id
INNER JOIN orders o ON o.order_id =
oi.order_item_order_id
WHERE o.order_status <> 'CANCELED'
AND o.order_status <> 'SUSPECTED_FRAUD'
GROUP BY p.product_id, p.product_name
ORDER BY total DESC;
```

Q9:

```
SELECT p.product_id, p.product_name,
SUM(cast(oi.order_item_subtotal AS
float)) AS total FROM order_items oi
INNER JOIN products p ON p.product_id =
oi.order_item_product_id
INNER JOIN orders o ON o.order_id =
oi.order_item_order_id
WHERE o.order_status <> 'CANCELED'
AND o.order_status <> 'SUSPECTED_FRAUD'
```

```
GROUP BY p.product_id, p.product_name
ORDER BY total DESC;
```

Q10:

```
SELECT p.product_id, p.product_name,
SUM(cast(oi.order_item_subtotal AS
float)) AS total FROM orders o
INNER JOIN order_items oi ON
oi.order_item_order_id = o.order_id
INNER JOIN products p ON p.product_id =
oi.order_item_product_id
WHERE o.order_status <> 'CANCELED'
AND o.order_status <> 'SUSPECTED_FRAUD'
GROUP BY p.product_id, p.product_name
ORDER BY total DESC;
```

Q11:

```
SELECT p.product_id, p.product_name,
SUM(cast(oi.order_item_subtotal AS
float)) AS total FROM order_items oi
INNER JOIN orders o ON o.order_id =
oi.order_item_order_id
INNER JOIN products p ON p.product_id =
oi.order_item_product_id
WHERE o.order_status <> 'CANCELED'
AND o.order_status <> 'SUSPECTED_FRAUD'
GROUP BY p.product_id, p.product_name
ORDER BY total DESC;
```

Table 7. The query execution time using a different join order

Setting	Query	Execution time (s)
`SET hive.auto.convert.join = true;` `SET hive.optimize.ppd = true;`	Q8, Q9	99
	Q10, Q11	79

Table `products` has 1345 rows, table `orders` has 500.000 rows, and table `order_items` has 1.000.000 rows. For the above queries, Q8 and Q9 first join the `products` table and the `order_items` table, and then the `orders` table, while Q10 and Q11 join the `orders` table and the `order_items` table first, and then join the result and the `products` table. Q10 and Q11 queries are more efficient because of joining the filtered table orders and largest table `order_items` first. The enabled predicate pushdown will cause filtering the data before a join. In this way, by adjusting the join order in a combination with the predicate pushdown, the size of the intermediate result is reduced and the query performance is improved.

6 CONCLUSION

The big data brings new opportunities as well as challenges. One of these challenges is optimization. The application of optimization techniques enables raising the quality of the process in terms of improving the management and processing of large amounts of the data when achieving desired results. The paper describes and analyzes different techniques for improving the query performance. Various queries are executed and reviewed by using different Hive optimization techniques, like cost-based optimization, statistics, predicate pushdown, parallel execution, partitioning, bucketing, different join types and join orders. The results of the analysis show that big data optimization techniques using Hive on Hadoop can significantly speed up queries. The improvement can more or less depend on the amount of the data, query operations and complexity, combination of the Hive features, and configuration parameters.

REFERENCES

[1] http://www.gartner.com/it-glossary/big-data, accessed: April 2021.

[2] A. K. Bhadani, D. Jothimani, "Big Data: Challenges, Opportunities, and Realities", chapter in an edited volume Effective Big Data Management and Opportunities for Implementation, 2016.

[3] V. Ganjir, Dr. B. K. Sarkar, R. R. Kumar, "Big Data Analytics for Healthcare", International Journal of Research in Engineering, Technology and Science, vol. VI, special issue, pp. 2-5, 2016.

[4] N. Khan, I. Yaqoob, I. A. T. Hashem, Z. Inayat, W. K. M. Ali, M. Alam, "Big Data: Survey, Technologies, Opportunities, and Challenges", The Scientific World Journal, Hindawi Publishing Corporation, 2014.

[5] V. Nerić, T. Konjić, N. Sarajlić, N. Hodžić, "A Survey on Big Data in Medical and Healthcare with a Review of the State in Bosnia and Herzegovina", The International Symposium on Computer Science – ISCS, 10th Days of BHAAAS in Bosnia and Herzegovina, Jahorina, B&H, 2018. (Advanced Technologies, Systems, and Applications III, Proceedings of the International Symposium on Innovative and Interdisciplinary Applications of Advanced Technologies (IAT), vol. 2, Springer, pp. 494-508, 2019.)

[6] A. Pothuganti, "Big Data Analytics: Hadoop-Map Reduce & NoSQL Databases", International Journal of Computer Science and Information Technologies (IJCSIT), vol. 6 (1), pp. 522-527, 2015.

[7] A. Y. Zomaya, S. Sakr, "Handbook of Big Data Technologies", Springer, 2017.

[8] T. White, "Hadoop: The Definitive Guide", O'Reilly, 2015.

[9] R. Jhajj, "Apache Hadoop Cookbook", Exelixis Media P. C., 2016.

[10] C. Lam, "Hadoop in Action", Manning Publications Co., 2010.

[11] A. Holmes, "Hadoop in Practice", Manning Publications Co., 2012.

[12] Y. Chen, X. Qin, H. Bian, J. Chen, Z. Dong, X. Du, Y. Gao, D. Liu, J. Lu, H. Zhang, "A Study of SQL-on-Hadoop Systems", Springer International Publishing Switzerland, 2014.

[13] X. Qin, Y. Chen, J. Chen, S. Li, J. Liu, H. Zhang, "The Performance of SQL-on-Hadoop Systems: An Experimental Study", IEEE 6th International Congress on Big Data, 2017.

[14] D. Abadi, S. Babu, F. Ozcan, I. Pandis, "Tutorial: SQL-on-Hadoop Systems", VLDB Endowment, vol. 8, no. 12, 2015.

[15] A. Tapdiya, D. Fabbri, "A Comparative Analysis of state-of-the-art SQL-on-Hadoop Systems for Interactive Analytics", IEEE Big Data Conference, 2017.

[16] M. A. Kukreja, "Apache Hive: Enterprise SQL on Big Data Frameworks", Technical Report, 2016.

[17] D. Du, "Apache Hive Essentials", Packt Publishing, 2015.

[18] H. Bansal, S. Chauhan, S. Mehrotra, "Apache Hive Cookbook", Packt Publishing, 2016.

[19] J. Rutherglen, D. Wampler, E. Capriolo, "Programming Hive: Data Warehouse and Query Language for Hadoop", O'Reilly, 2012.

[20] https://cwiki.apache.org/confluence/display/Hive/Design, accessed: April 2021.
[21] Cloudera Inc., "Apache Hive Guide", 2021.
[22] Cloudera Deployment Guide, "Getting Started with Hadoop Tutorial", 2021.
[23] Cloudera Inc., "Hue Guide", 2021.
[24] V. Nerić, N. Sarajlić, "A Review on Big Data Optimization Techniques", B&H Electrical Engineering, vol. 14, pp. 13-18, 2020.
[25] S. Bagui, K. Devulapalli, "Comparison of Hive's Query Optimisation Techniques", Int. J. Big Data Intelligence, 2018.
[26] https://cwiki.apache.org/confluence/display/Hive/Cost-based+optimization+in+Hive, accessed: April 2021.
[27] https://docs.cloudera.com/HDPDocuments/HDP2/HDP-2.6.4/bk_hive-performance-tuning/content/ch_cost-based-optimizer.html, accessed: April 2021.
[28] A. Gruenheid, E. Omiecinski, L. Mark, "Query Optimization Using Column Statistics in Hive", IDEAS11, 2011.
[29] https://cwiki.apache.org/confluence/display/Hive/StatsDev, accessed: April 2021.
[30] Q. Liu, H. Hong, H. Zhu, H. Fan, "Research and Comparison of SQL Optimization Techniques Based on MapReduce", International Conference on Computer Science and Application Engineering (CSAE), 2017.
[31] https://cwiki.apache.org/confluence/display/Hive/FilterPushdown Dev, accessed: April 2021.
[32] A. Barman, D. Paranjpe, "Improving the Performance of Hive by Parallel Processing of Massive Data", International Journal of Recent Development in Engineering and Technology (IJRDT), vol. 7, issue 4, 2018.
[33] S. Wu, F. Li, S. Mehrotra, B. C. Ooi, "Query Optimization for Massively Parallel Data Processing", SOCC, 2011.
[34] E. Costa, C. Costa, M. Y. Santos, "Partitioning and Bucketing in Hive-Based Big Data Warehouses", Trends, and Advances in Information Systems and Technologies, pp. 764-774, 2018.
[35] E. Costa, C. Costa, M. Y. Santos, "Evaluating Partitioning and Bucketing Strategies for Hive-based Big Data Warehousing Systems", Journal of Big Data, no. 34, 2019.
[36] https://weidongzhou.wordpress.com/2017/06/06/join-type-in-hive-common-join/, accessed: April 2021.
[37] https://cwiki.apache.org/confluence/display/Hive/LanguageManual+JoinOptimization, accessed: April 2021.
[38] https://weidongzhou.wordpress.com/2017/06/08/join-type-in-hive-skewed-join/, accessed: April 2021.
[39] https://cwiki.apache.org/confluence/display/Hive/Skewed+Join+Optimization, accessed: April 2021.
[40] https://weidongzhou.wordpress.com/2017/06/07/join-type-in-hive-map-join/, accessed: April 2021.
[41] https://cwiki.apache.org/confluence/display/Hive/MapJoinOptimization, accessed: April 2021.
[42] K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon, "Parallel Data Processing with MapReduce: A Survey", SIGMOD Record, vol. 40, no. 4, pp. 11–20, 2011.
[43] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian, "A Comparison of Join Algorithms for Log Processing in MapReduce", SIGMOD International Conference on Management of data, New York, NY, USA, pp. 975–986, 2010.
[44] https://weidongzhou.wordpress.com/2017/06/09/join-type-bucket-join/, accessed: April 2021.
[45] S. Pal, "SQL on Big Data – Technology, Architecture, and Innovation", Apress, 2016.

Vedrana Nerić graduated in 2006 and received her M.Sc. degree in 2013 from the Faculty of Electrical Engineering of the University of Tuzla, Bosnia and Herzegovina. During her studies, she was presented three Silver and a Gold Medal by the same university. Currently, she is a Ph.D. student. She is employed with Virgin Pulse, Tuzla, as a senior data engineer. In 2014, she was nominated to a teaching assistant in the field of Computer and Information Science at the same faculty.

Nermin Sarajlić graduated in 1987 and received his M.Sc. degree in 1997 from the Faculty of Electrical Engineering and Faculty of Electrical Engineering and Mechanical Engineering, respectively, and his Ph.D. degree in 2002 from the Faculty of Electrical Engineering of the University of Tuzla, Bosnia and Herzegovina. His field of interest is the calculation of the coupled electromagnetic-temperature fields, cryptography, crypto analysis.