Hindawi
Scientific Programming
Volume 2020, Article ID 6364752, 12 pages
https://doi.org/10.1155/2020/6364752

Research Article

Query Execution Optimization in Spark SQL

Xuechun Ji,[1,2] **Maoxian Zhao**,[3] **Mingyu Zhai**,[2] **and Qingxi Wu**[2]

[1]*School of Computer Science and Engineering, Southeast University, Nanjing, China*
[2]*NARI Research Institute NARI Technology, Nanjing, China*
[3]*College of Mathematics and Systems Science, Shandong University of Science and Technology, Qingdao, China*

Correspondence should be addressed to Maoxian Zhao; sdzmx66@163.com

Received 23 October 2019; Accepted 11 January 2020; Published 7 February 2020

Guest Editor: Jianming Bian

Spark SQL is a big data processing tool for structured data query and analysis. However, due to the execution of Spark SQL, there are multiple times to write intermediate data to the disk, which reduces the execution efficiency of Spark SQL. Targeting on the existing issues, we design and implement an intermediate data cache layer between the underlying file system and the upper Spark core to reduce the cost of random disk I/O. By using the query pre-analysis module, we can dynamically adjust the capacity of cache layer for different queries. And the allocation module can allocate proper memory for each node in cluster. According to the sharing of the intermediate data in the Spark SQL workflow, this paper proposes a cost-based correlation merging algorithm, which can effectively reduce the cost of reading and writing redundant data. This paper develops the SSO (Spark SQL Optimizer) module and integrates it into the original Spark system to achieve the above functions. This paper compares the query performance with the existing Spark SQL by experiment data generated by TPC-H tool. The experimental results show that the SSO module can effectively improve the query efficiency, reduce the disk I/O cost and make full use of the cluster memory resources.

1. Introduction

With the increasing popularity of e-commerce, social network, artificial intelligence and other new Internet applications, the amount of data being stored and processed by governments, enterprises and research institutions has increased dramatically. A substation in the power system generates 100000 alarm data per minute, and the Facebook generates more than 400 TB of logs every day. It is an urgent problem for enterprises and research institutions to store such large-scale data on hard disk persistently and retrieve the information required by users in a short time. Data storage and processing system in big data environment has received more and more attention in recent years.

The early big data processing systems mainly revolved around Hadoop platform. In order to solve the problem that Hadoop platform frequently read and wrote intermediate data in HDFS, a method to cache Hadoop's Shuffle data in memory was proposed by Shi et al. [1] Although this method could effectively reduce the large amount of random disk I/O cost caused by reading and writing intermediate data, it was inflexible as the cache size created by this method was fixed for different applications. A certain amount of memory would be wasted for some applications with small amount of Shuffle data. To solve the problems of Hadoop platform, the memory-based distributed computing framework Apache Spark emerged.

Spark is a high-speed and versatile big data processing engine. It is based on the implementation of RDD [2] (Resilient Distributed Datasets) and implements data distribution and fault tolerance. As shown in Figure 1, the current Spark ecosystem consists of three layers: the bottom layer, the middle layer, and the top layer. The bottom layer can read input data from HDFS [3], Amazon S3, HyperTable and HBase [4]. The middle layer uses resource scheduling management platforms such as Standalone, Yarn [5] and Mesos [6] to complete the analysis and processing of applications. The top layer consists of a series of advanced tools, including Spark Streaming [7], GraphX [8], BlinkDB [9], MLlib [10] and Spark SQL [11]. Spark SQL is developed from Shark [12], it provides functions like Hive [13] which allows users to process structured data directly by entering

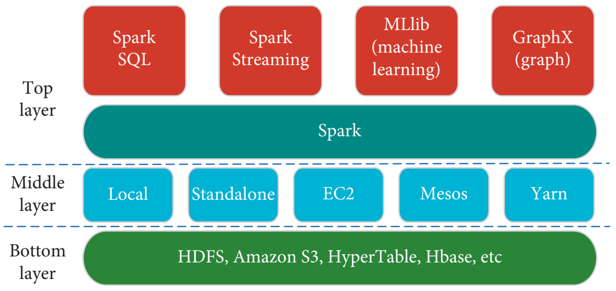

Figure 1: Spark ecosphere.

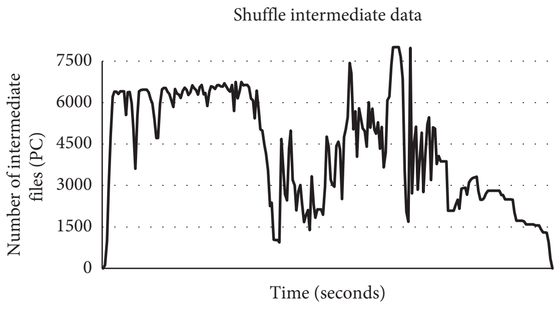

Figure 2: Wave pattern of spark shuffle intermediate data.

SQL statements. The Catalyst which generates and optimizes execution plan of Spark SQL will perform algebraic optimization for SQL query statements submitted by users and generate Spark workflow and submit them for execution.

However, the Spark SQL system currently faces two problems. One problem is that the frequent reading and writing of intermediate data in the data interaction process between Spark tasks lead to serious random disk I/O cost. The number of intermediate files produced by a simple program on Spark is shown in Figure 2.

The other problem is that there are no suitable optimization rules for Spark workflow. Spark jobs with intermediate data correlation need to read the same input data from disk repeatedly, resulting in redundant disk I/O cost.

Under the above background, this paper aims to improve the execution efficiency of Spark SQL. The main contributions are as follows.

(1) Reduce the random disk I/O cost in the Shuffle phase by adding an intermediate data cache layer between the Spark core layer and the underlying distributed file system.

(2) Reduce the read/write cost of the same intermediate data between Spark jobs by using a cost-based correlation merging algorithm, then further improving the performance of the data analysis system.

2. System Architecture

The SSO (Spark SQL Optimizer) prototype system is designed and developed. Its system architecture is shown in Figure 3.

The existing Spark system runs in a master-slave mode. The Driver process on the main node receives query requests submitted from Client. The Worker process on the slave node executes specific query tasks. SSO system optimizes and improves the original Spark, adding a dynamic Shuffle optimizer module named Dynamic Shuffle Optimizer on the main node. Spark's original Catalyst framework is also modified to include a cost model module renamed SQL workflow optimizer. Reading and writing interfaces for distributed memory file system are added to the slave node for the Shuffle Map Task and Shuffle Reduce Task. The process of optimizing the users' queries by SSO system is described below.

Queries submitted by users are first parsed by the parser in the SQL workflow optimizer module to form a logic execution plan. Then, the optimized execution plan is submitted to Dynamic Shuffle Optimizer and DAG scheduler. Dynamic Shuffle Optimizer calculates the size of intermediate data generated by the optimized SQL queries using the query pre-analysis module. Then the allocation module at the cache layer performs buffer allocation on the distributed memory file system. Finally, the workflow generated by the execution plan will be submitted to the DAG scheduler for task assignment at the slave node.

3. Spark Shuffle Intermediate Data Cache Layer

In the existing Spark system, data interaction between Stages will frequently generate disk I/O overhead. Therefore, a strategy for caching immediate data is proposed. The dynamic Shuffle optimizer is used to create buffers with different sizes on the distributed memory file system dynamically for different Spark SQL workflow. The random disk I/O cost of the Shuffle phase will be reduced by caching Shuffle intermediate data in memory.

3.1. Query Pre-Analysis Module. The following query execution process is analyzed to show when Spark SQL writes intermediate data to disk. The query statement is:

select

l_partkey, l_quantity, l_extendedprice

from

lineitem, part

where

p_partkey = l_partkey
and l_quantity <40

Analysis shows that the query includes only one join operation. The execution process of the query under Spark SQL is shown in Figure 4.

The above execution process of Spark is divided into three Stage. Stage 0 reads the part table and runs projection operations on the attribute p_partkey. Stage 1 reads the lineitem table and performs projection operations on the three attributes of l_partkey, l_quantity and l_extendedprice as well as selection operations of l_quantity <40. Stage 2 reads the intermediate results of Stage 0 and Stage 1,

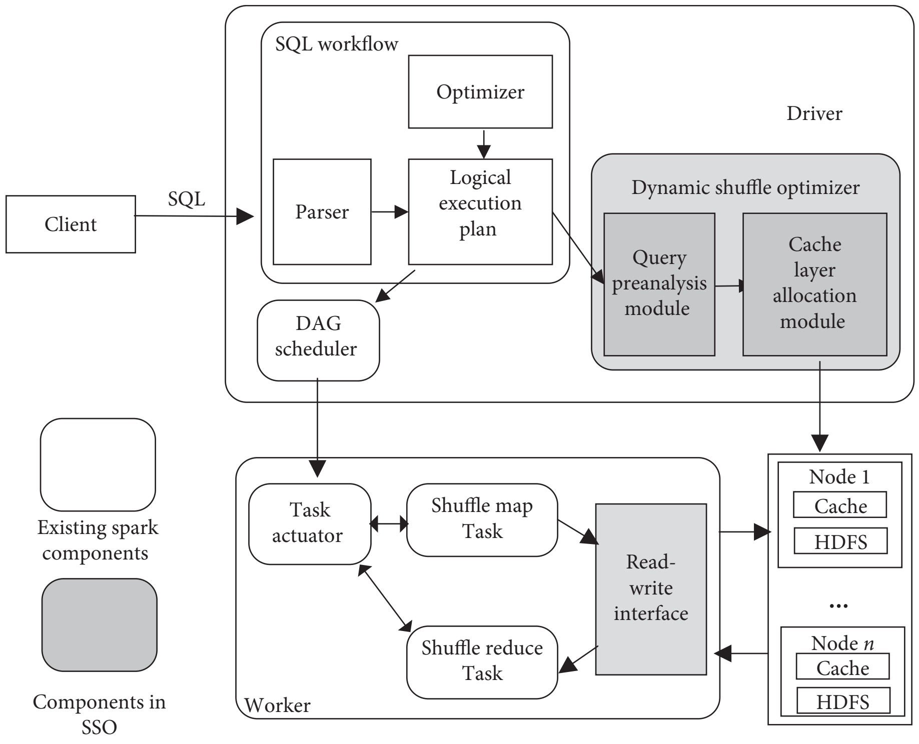

Figure 3: System architecture of SSO.

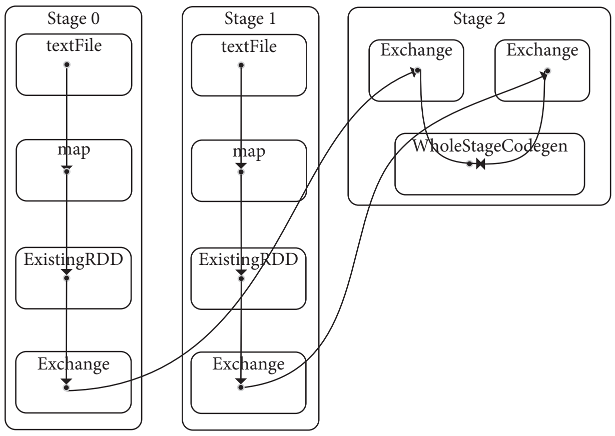

Figure 4: The execution process of spark.

performs join operations of p_partkey = l_partkey and finally writes the query results to disk.

Analysis of this query show that the query execution workflow in Spark can be divided into two stages. One is S-Stage for projection, selection and aggregation operations, and the other is J-Stage for join operations.

For S-Stage, the following rules can be used to calculate the size of output data at this stage.

(1) Projection

$$\left|D_{\text{out}}(\text{pro})\right| = \beta_{\text{pro}}\left|D_{\text{in}}\right|,$$
$$\beta_{\text{pro}} = \frac{L_{\text{pro}}}{L_D}. \tag{1}$$

L_{pro} represents the length of projection attribute, and L_D represents the total length of all attributes.

(2) Selection

$$\left|D_{\text{out}}(\text{fil})\right| = \beta_{\text{fil}}\left|D_{\text{in}}\right|. \tag{2}$$

The value of β_{fil} is related to specific selection conditions and data distribution of the original table.

(3) Aggregation

$$\left|D_{\text{out}}(\text{agg})\right| \approx 0. \tag{3}$$

Aggregation operation returns the sum or average value of an attribute, so the output size is negligible.

For example, Figure 5 is the running state diagram of this SQL query on Spark. The input data size read by Stage 0 is 233.2 MB, known Lpartkey = 10, $L_D = 194$. Through the above analysis, the output of this Stage can be calculated $|D_{\text{out}}| = \beta_{\text{pro}}|D_{\text{in}}| = 10/194 * 233 \approx 13$ MB. It approximates the actual output of Stage 0 of 10 MB. Similarly, the input data size read by Stage 1 is 7.3 GB, known Lpartkey = 10, Lquantity = Lextendedprice = 15, $L_D = 231$. Sampling shows that the tuples with l_quantity >40 account for about 60% of the total tuples. Therefore the output of this Stage is $|D_{\text{out}}| = \beta_{\text{pro}} * \beta_{\text{fil}}|D_{\text{in}}| = (10 + 15 + 15)/231 * 0.6 * 7.3\ \text{GB} \approx 776$ MB, which approximates the actual output of 824 MB generated by Stage 1.

3.2. Cost Analysis of Join Operation. For the J-Stage, the following rules are used to calculate the output data size of this Stage.

Stage ID	Duration	Tasks: succeeded/total	Input	Output	Shuffle read	Shuffle write
2	8 s	200/200			833.9 MB	
1	19 s	232/232	7.3 GB		833.9 MB	824.0 MB
0	4 s	8/8	233.2 MB		833.9 MB	9.9 MB

Figure 5: The result of query execution.

$$\left|D_{\text{out}}(\text{join})\right| = \gamma\left(\left|D_1^{\text{in}}\right| \times_C \left|D_2^{\text{in}}\right|\right). \qquad (4)$$

If there is no join condition C, it becomes Cartesian product and $\gamma = 1$. If no tuple satisfies join condition C, γ equals 0. Normally $0 \le \gamma \le 1$. If the join condition is $D_1 \cdot A = D_2 \cdot B$, there are three special cases.

(1) If A is the primary key of D_1, each tuple in D_2 matches at most one tuple in D_1, that is $|D_{\text{out}}(\text{join})| \le |D_{\text{in}}\ 2|$. So, $\gamma \le 1/|D_{\text{in}}\ 1|$. If B is the primary key of D_2, $\gamma \le 1/|D_{\text{in}}\ 2|$.

(2) If A is not the primary key of D_1 and B is not the primary key of D_2, and the attributes A and B obey uniform distribution on the same domain M, then $\gamma = 1/|M|$.

(3) If A is not the primary key of D_1 and B is not the primary key of D_2, and the attributes A and B do not obey uniform distribution, γ needs to be sampled according to the specific data set, table structure and data size. As for the cost analysis method of join operation under non-uniform distribution, the idea of histogram method mentioned in [14] is referred to develop a method suitable for Spark SQL. Assuming that R is a relationship, field C is an attribute of R, and the value range of C is [min, . . ., max], where min and max are the minimum and maximum values of field C in relation R respectively. The [min, . . ., max] is divided into several intervals, called straight or bucket. Then the number of tuples whose C attribute values are in these intervals is counted. In order to analyze the cost of join operation by histogram method, the first step is to construct the variable-width distribution histogram. The flow of the algorithm is shown in Algorithm 1. The input of the algorithm is frequency distribution histogram which records the value of each attribute and its occurrence times. The attribute values are sorted from small to large. The output is variable-width distribution histogram which consists of histogram buckets. Each histogram bucket records the starting and ending values of the attribute, and the frequency sum of attribute within the range. There is no intersection between buckets. In the algorithm, a variable max is declared to record the maximum frequency in the current histogram bucket. For each of the following attributes and frequency pairs, if the difference between the frequency value and max only accounts for 5% or less of the frequency value, it can be considered that this attribute obeys the same distribution as each attribute in the current histogram bucket. Therefore, this attribute is also included in the current histogram bucket. If the above conditions are not met, a new histogram bucket is created whose starting value is the current attribute value and max is the frequency value of the corresponding attribute value. Continue the above steps until all attributes and frequency pairs have been traversed.

After constructing the variable-width distribution histogram, the algorithm to calculate the total tuple number after joining is shown in Algorithm 2.

The input of the algorithm is the variable-width distribution histogram of relation R and relation S. The output is the total number of tuples after the join operation of the two relations. The algorithm seeks overlaps between two histogram buckets from the first histogram bucket of relation R and relation S histogram. As the histogram bucket is uniformly distributed, templeft and tempright can be obtained by dividing the size of the overlaps by the width of the histogram bucket, and then multiplying the result by the total frequency of the histogram bucket. Templeft and tempright represent the frequency of the equivalent attribute in the histogram bucket of relation R and relation S respectively. Multiply templeft by tempright and divide by the size of the overlaps, then the total tuple number generated by the join operation in the first histogram bucket with overlaps is obtained. If there is no overlap between two histogram buckets and the range of bucket $h_i r$ is smaller than the range of bucket $h_j s$, the range of bucket $h_{i+1} r$ and bucket $h_j s$ need to be compared. Continue the above steps until all the histogram buckets in relation R or relation S are traversed.

3.3. Cache Layer Allocation Module

After calculating the total size of the cache layer, the next question is how much memory is allocated for each node in the cluster. Due to the principle of Spark Data Locality, when reading HDFS files, Spark will assign the nearest Executor to the data storage. For nodes that store more input data, more memory should be allocated. The input data in HDFS is organized as blocks, and the default size of a block is 128 MB. So, after parsing users' query statements to obtain the tables to be required, it is necessary to analyze the distribution of blocks in the cluster for each table. Since the size of each block in HDFS is the same, the memory size M_i allocated to each node can be obtained by calculating the ratio of the block number of each node B_i to the total blocks B_{total}.

$$M_i = \frac{B_i}{B_{\text{total}}} M_{\text{total}}. \qquad (5)$$

For the Q1 query mentioned before, the size of table Lineitem is 7.2 GB including 60 blocks. The size of table Part is 233 MB including 2 blocks. The distribution of these blocks in the cluster is shown in Figure 6.

```
Construct variable-width distribution histogram by frequency distribution histogram
input: $H_{\mathrm{fre}}$ = {<attr1, freq1>, <attr2, freq2>, . . ., <attrn, freqn>}
output: $H_{\mathrm{width}}$ = {<start$_1$, end$_1$, times$_1$>,<start$_2$, end$_2$, times$_2$>, . . ., <start$_m$, end$_m$, times$_m$>}
procedure
  $i \longleftarrow 1$; $H_{\mathrm{width}} \longleftarrow \{\}$
  start ⟵ attr$_1$; end ⟵ attr$_1$;
  max ⟵ freq$_1$; $T \longleftarrow$ freq$_1$
  while $i \le n$ do
    $i \longleftarrow i + 1$
    if |max-freq$_i$|/freq$_i$ < 0.05 then
      end ⟵ attr$_i$
      $T \longleftarrow T$ + freq$_i$
      if freq$_i$ > max then
        max ⟵ freq$_i$
      end if
    else
      $H_{\mathrm{width}} \longleftarrow H_{\mathrm{width}}$ + <start, end, $T$>
      start ⟵ attr$_i$; end ⟵ attr$_i$
      max ⟵ freq$_i$; $T \longleftarrow$ freq$_i$
    end if
  end while
end procedure
```

Algorithm 1: Algorithm to construct variable-width distribution histogram.

```
Estimate the size of join operation by histogram method
input: $H_R = \{h_1r, h_2r, \ldots, h_nr\}$, HS = $\{h_1s, h_2s, \ldots, h_ms\}$
Output: Total tuples Sum after join;
procedure
  $i \longleftarrow 1$; $j \longleftarrow 1$; Sum ⟵ 0;
  while $i \le n$ and $j \le m$ do;
    if $h_i$ and $h_j$ have overlap then;
      Overlap ⟵ Overlap of two histogram buckets;
      templeft ⟵ $h_i$.times ∗ Overlap/($h_i$.end-$h_i$.start)
      tempright ⟵ $h_j$.times ∗ Overlap/($h_j$.end-$h_j$.start)
      Sum ⟵ Sum + templeft ∗ tempright/Overlap
      if $h_i$.end < $h_j$.end then
        $i \longleftarrow i + 1$
      else
        $j \longleftarrow j + 1$
      end if
    else
      if $h_i$.end < $h_j$.start then
        $i \longleftarrow i + 1$
      else
        $j \longleftarrow j + 1$
      end if
    end if
  end while
end procedure
```

Algorithm 2: Calculation of the tuple number of join operation by histogram method.

After obtaining the distribution of the input data in the cluster, according to the total size of the cache layer calculated by query pre-analysis module and the formulas in the cache layer allocation module, the memory size required for each node is $790 * 2/62 \approx 25.48$ MB, $790 * 21/62 \approx 267.58$ MB, $790 * 3/62 \approx 38.22$ MB, $790 * 20/62 \approx 254.83$ MB, $790 * 4/62 \approx 50.96$ MB, $790 * 5/62 \approx 63.71$ MB. The results approximate to the actual Shuffle data in Table 1.

Since Spark is a framework based on memory computing, the operations on Resilient Distributed Datasets are

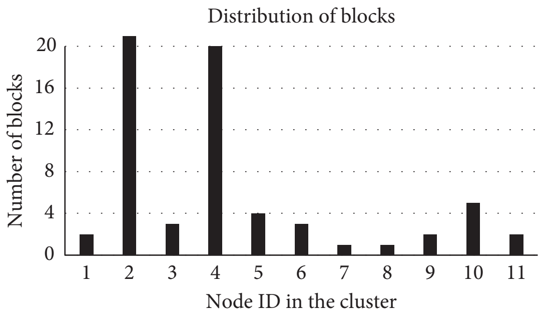

Figure 6: Distribution of blocks in the cluster.

Table 1: Comparison between calculated and actual shuffle sizes.

ID	1	2	3	4	5	6	7	8	9	10	11
General task	9	114	14	119	7	14	9	22	14	7	7
Input (MB)	256	2688	384	2560	512	384	128	128	256	640	256
Calculated shuffle size	27	282.4	40.3	269	53.81	40.35	13.45	15.05	28.03	67.25	29.06
Actual shuffle size	25	267.58	38.2	254.8	50.96	38.22	12.76	12.76	25.48	63.71	25.48

all carried out in memory before or after Shuffle operations. If the cluster has enough memory, even if a certain size of memory space has been allocated for intermediate data cache layer, the remaining memory is enough for Spark task to perform calculation. If the memory resources of the cluster are limited, the query pre-analysis module will calculate the cache size of Shuffle before the query runs, and allocate the memory through the cache layer allocation module. This will lead to occupying the memory before the Shuffle operation is carried out. This is not a reasonable approach obviously. The cache layer allocation module adopts a delay allocation scheme to solve this problem. The Spark program allocates as much memory as possible to Spark works in the non-Shuffle phase. In the Shuffle phase, Spark works do not need memory resources for calculation. Then part of the memory resources in Spark works and the remaining memory resources in the cluster are called for the intermediate data cache layer.

4. Cost-Based Correlation Merging Algorithm

The optimization process of SQL query by the Catalyst framework in the traditional Spark system is shown in Figure 7. First, the query statements entered by users are parsed by SQL parser to form a logical execution plan tree. The plan tree is then algebraically optimized to some extent by the SQL optimizer. In the current Spark SQL workflow, there is a case of repeatedly reading and writing the same intermediate data. There are multiple tasks at the nodes of the logical execution plan tree output the same intermediate data, resulting in additional disk I/O cost. Therefore, an optimization rule can be added to the original SQL optimizer to merge the same intermediate data. Although merging will reduce the cost of writing output data on disk, subsequent tasks will read the intermediate data they do not need which also brings extra disk reading cost. So, it needs cost calculation whether to merge tasks with intermediate data correlation. A cost model module is introduced based on the original Catalyst framework. When merging tasks with intermediate data correlation, SQL optimizer will determine whether to execute the optimization rule by cost calculation. The optimized Catalyst framework is named as SQL workflow optimizer and its framework is shown in Figure 7.

4.1. Cost Model.

4.1. Cost Model. The cost model for Spark tasks execution should be established in order to merge Spark tasks with intermediate data correlation. Then the benefits and extra costs of merging should be calculated based on model to decide whether to merge by comparison.

For establishing the task execution cost model in Spark, we improve the method proposed by Singhal and Singh [15] and add the cost generated by sorting operation. When calculating the Stage cost, reading input data, merging and sorting intermediate data, and writing output data are considered, that is

$$C(\text{Stage}) = C_{\text{read}}(\text{Stage}) + C_{\text{sort}}(\text{Stage}) + C_{\text{write}}(\text{Stage}). \tag{6}$$

Since both $C_{\text{read}}(\text{Stage})$ and $C_{\text{write}}(\text{Stage})$ are I/O cost, the cost calculation formula is $C_{I/O} = C_0\text{T} + C_1 x$. The definition of each parameter is shown in Table 2.

The cost of Stage reading phase $C_{\text{read}}(\text{Stage})$ can be calculated by

$$C_1 x = |D_{\text{in}}| t_r + \alpha |D_{\text{in}}| t_b = (t_r + \alpha t_b)|D_{\text{in}}|. \tag{7}$$

$|D_{\text{in}}|$ is determined by the size of the source input data or the output data of other Stage. The value of α is 0.3. It is determined by the default three-copy storage strategy of HDFS, one locally, one on the same rack and one on the remote rack. The number of I/O occurrence depends on the specific Stage. For S-Stage, if the source input data is read, T equals 1 as the source data is stored continuously. If the

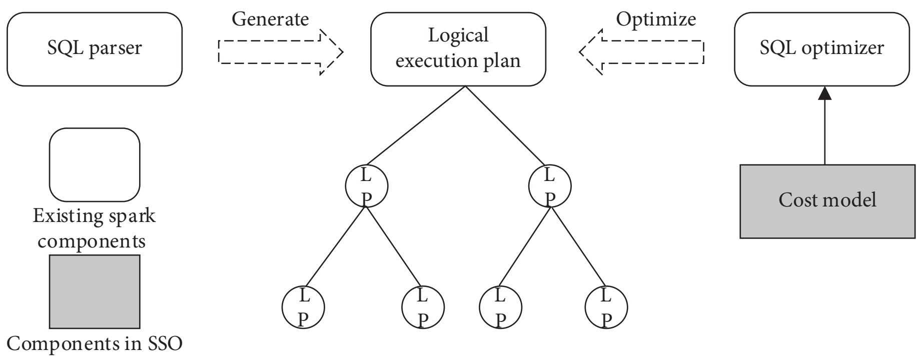

Figure 7: The framework of SQL workflow optimizer.

Table 2: Parameters in the cost model.

Parameter	Meaning
X	Size of read/written data
C_0	Seeking time and rotation delay time
C_1	Time required to transmit 1 MB data
A	Proportion of non-local data to total data
T	Number of I/O occurrences
$\vert D_{\text{in}}\vert$	Size of stage input data
$\vert D_{\text{out}}\vert$	Size of stage output data t_r time to read 1 MB data locally
t_w	Time to write 1 MB data locally
t_b	Time to transfer 1 MB data over network
B	Buffer size of spark task m task number in stage

output of other Stages is read, the value of T is determined by the file number of intermediate data generated at the previous Stage. Spark tasks write the intermediate data to the buffer first, and then overwrite the disk to form a file when the buffer is full. Suppose that the previous Stage generates data of size $|D_{\text{out}}|$, then $|D_{\text{out}}|/B$ intermediate files will be generated, where B is the size of the Spark task buffer. So, the number of I/O occurrence is $T = |D_{\text{out}}|/B$. For J-Stage, the input of this Stage must be the output of the other two stages due to the join operation of two tables. Assuming that the previous two stages produce a total of $|D_{\text{out}}|$ size output data, if the two-way merge sort join algorithm is used, $\lceil \log 2(|D_{\text{out}}|/B)\rceil$ times of scanning are needed, and the number of I/O occurrence is $|D_{\text{out}}|/B\lceil \log 2(|D_{\text{out}}|/B)\rceil$. In summary, the calculation formula for the cost in the reading phase of Stage is as follows.

$$C_{\text{read}}(\text{Stage}) = C_0 T + (t_r + \alpha t_b)|D_{\text{in}}|. \tag{8}$$

For the cost of the writing phase of Stage C_{write} (Stage), since the intermediate data is overwritten to the local disk, this process does not involve the network transmission cost. The calculation formula is

$$C_1 x = |D_{\text{out}}| t_w, \tag{9}$$

where $|D_{\text{out}}|$ can be calculated by the method described in Chapter 2. The number of I/O occurrence T is determined by the number of intermediate files. The number of intermediate files is calculated by $|D_{\text{out}}|/B$. Then the calculation formula of the writing cost in Stage is given by

$$C_{\text{write}}(\text{Stage}) = \frac{C_0|D_{\text{out}}|}{B} + t_w|D_{\text{out}}|. \tag{10}$$

Each task in Stage needs to sort and merge all the intermediate data files generated by itself. If a total of $|D_{\text{out}}|/B$ intermediate files are generated, it can be considered that each task generates $|D_{\text{out}}|/B_m$ intermediate files, where m is the number of tasks in each Stage. So, the cost of the sorting phase C_{sort}(Stage) is calculated by

$$C_{\text{sort}}(\text{Stage}) = \frac{C_0|D_{\text{out}}|}{B_m}\left\lceil \log 2\left(\frac{|D_{\text{out}}|}{B_m}\right)\right\rceil + t_r|D_{\text{out}}|. \tag{11}$$

Assuming that the number of sorting P equals $|D_{\text{out}}|/B_m\lceil \log 2(|D_{\text{out}}|/B_m)\rceil$, the execution cost of a Stage in Spark is given by

$$\begin{aligned} C(\text{Stage}) &= (t_r + \alpha t_b)|D_{\text{in}}| + (t_r + t_w)|D_{\text{out}}| \\ &\quad + C_0\left(T + P + \left(\frac{|D_{\text{out}}|}{B}\right)\right). \end{aligned} \tag{12}$$

4.2. Format of Shuffle Intermediate Data. In order to merge the intermediate data, the format of the Shuffle intermediate data is first introduced and then the shared field merging algorithm is proposed.

The MapTask in the Spark Shuffle phase extracts the required data by scaning each line of the source data file. Then it stores the data in the form of key and value to the <key, value> pairs and outputs to the local disk. Take the work Stage 1 in Figure 8 as an example, the corresponding query is *select l_partkey, l_quantity, l_extendedprice from lineitem, part where p_partkey = l_partkey.*

The parsed workflows are all joined at the Reduce Task in Spark SQL. So, in the MapTask phase, the data is read from the lineitem table by row, then l_partkey is saved as key, and l_quantity and l_extendedprice are saved as values. To facilitate the join operation at Reduce, the actual <key, value> pairs are of the form < l_partkey, l_quantity|l_extendprice|>.

Similarly, For Stage 3 in Figure 8, the query is select l_partkey, l_quantity from lineitem, supplier where s_suppkey = l_suppkey and s_nationkey = 2. The <key, value> pairs generated in the Shuffle phase is <l_suppkey, l_partkey|l_quantity>.

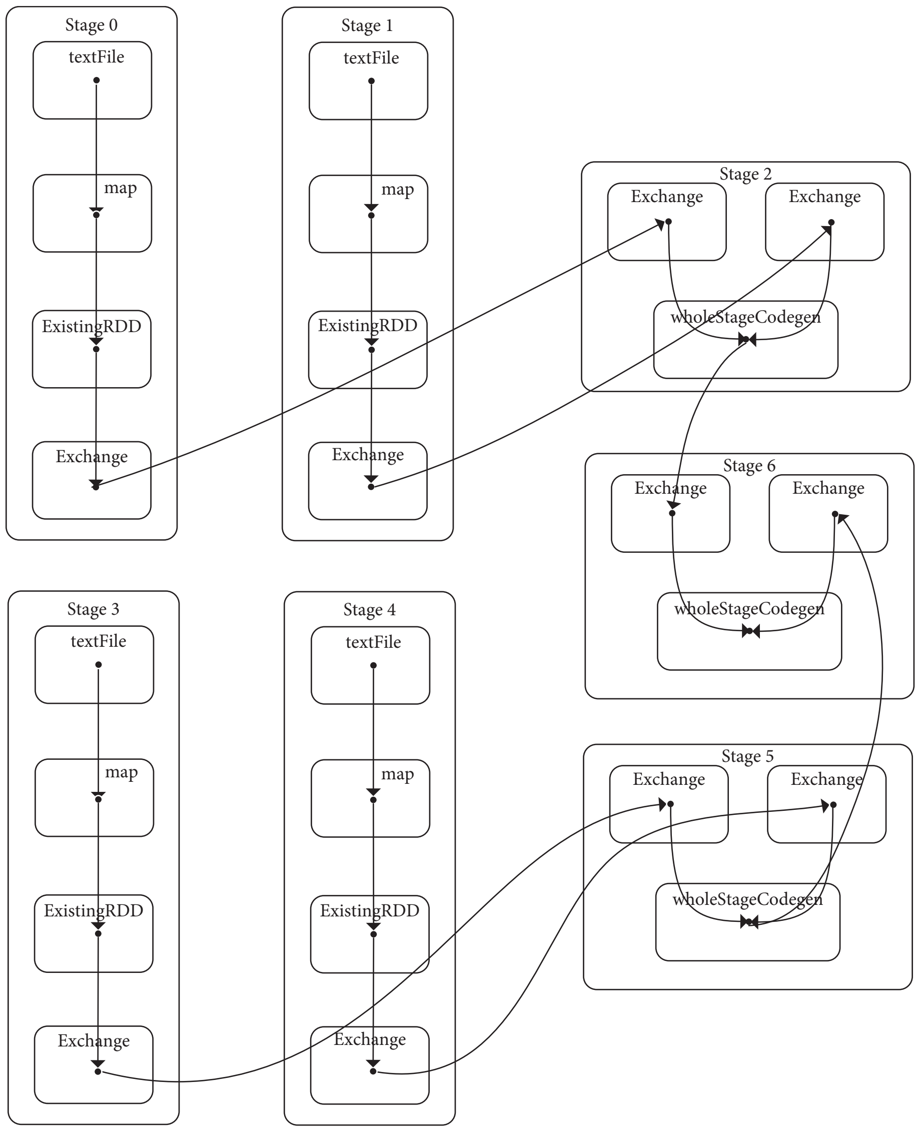

FIGURE 8: The query execution flow of TPC-H Q17.

The field l_partkey appears in the intermediate data output of both Stage 1 and Stage 3, one as the Key and the other as the Value. Meanwhile, l_quantity also appears in the Value of both stages. Since both sub-queries involve only join and projection operations and no selection operations, the output data from MapTask of Stage 1 and Stage 3 contains all the rows in the lineitem table. It can be considered that the two types of pairs contain all the l_partkey and l_quantity in the lineitem table, so they can be merged. Although merging reduces the writing cost of the current Stage, it also increases the reading cost of the subsequent Stages. For example, Stage 2 joins the part table and lineitem table, but reads the unwanted l_suppkey field after merging. Similarly, Stage 5 joins the supplier table and lineitem table, but reads the unwanted l_extendedprice field after merging. Therefore, the writing cost of saved field and the reading cost of increased field should be weighed to determine whether to merge the Shuffle intermediate data or not. The problem can be solved by the shared field merging algorithm below.

4.3. Shared Field Merging Algorithm. Assuming that the SQL statement of Stage$_i$ is *select* $i_1, i_2, \ldots, i_n$ *from table*$_i$, the format of <key, value> pairs generated is $<i_1,\ i_2|i_3|\ldots|i_n>$. The format of <key, value> pairs generated by Stage$_j$ is $<j_1,\ j_2|j_3\ldots|j_m>$. Let $S_i = \{i_2, i_3, \ldots, i_n\}$ and $S_j = \{j_2, j_3, \ldots, j_m\}$, then the format of <key, value> pairs after merging is $<i_1 \cup j_1,\ S_i \cup S_{j\text{-}i1} \cup j_1>$. The keys of two pairs are first compared. Unify them into one field if the field names are the same, otherwise, join the two fields together separated by "|." Next, merge the value sets of the two pairs and remove the fields that have already appeared in the Key. In summary, the data format after merging the intermediate data of Stage 1 and Stage 3 is <l_partkey |l_suppkey, l_quantity |l_extendedprice>.

To measure the writing benefit of saved fields and the reading cost of increased fields, the definition of cost model is

$$\text{Savings} = \frac{C_0\left|D_{\text{write}}\right|}{B} + t_w\left|D_{\text{write}}\right|,$$

$$\text{Costs} = C_0\frac{\left|D_{\text{read}}\right|}{B}\left\lceil \log_2\left(\frac{\left|D_{\text{read}}\right|}{B}\right)\right\rceil + \left(t_r + \alpha t_b\right)\left|D_{\text{read}}\right|, \tag{13}$$

where $|D_{\text{write}}| = |D_{\text{out}}| \quad L_{\text{save\_cols}}/L_{\text{total}}$, $|D_{\text{read}}| = |D_{\text{out}}| \quad L_{\text{add\_cols}}/L_{\text{total}}$. $L_{\text{save\_cols}}$ is the length of saved fields of writing. $L_{\text{add\_cols}}$ is the length of increased fields of reading. L_{total} is the length of the total fields. $|D_{\text{out}}|$ can be calculated by the method described in Chapter 2. For example, after merging into the above pairs, the repeated writing of l_parkey and l_quantity can be reduced, and $L_{\text{save\_cols}} = L_{\text{l\_partkey}} + L_{\text{l\_quantity}}$. Similarly, the redundant reading of l_extendedprice and l_suppkey is increased, and $L_{\text{add\_cols}} = L_{\text{l\_extendedprice}} + L_{\text{l\_suppkey}}$. Let $E = \text{Savings}–\text{Costs}$. If $E > 0$, it means that the reduced cost of sharing redundant fields is greater than the cost of reading additional fields. The public fields in the intermediate data can be merged. Otherwise, if Earn <0, these fields will not be merged. Whether to merge the intermediate data needs comparison between the value of Savings and the value of Costs.

5. Experiments

We verified the performance of SSO system in this section. The test data was generated by the TPC-H benchmarking tool. Part of the query statements provided by TPC-H was selected to test the performance of this system in two parts: intermediate data caching test and intermediate data correlation merging algorithm test.

5.1. Experimental Environment and Data Set. The hardware environment was as follows. The experimental cluster included a main node and 10 sub-nodes. The nodes were connected through gigabit ethernet. Each node was configured with 2.7 GHz CPU(8-core 16 threads, 20M Cache, Turbo frequency), 64 GB DDR3 1600 and 500 GB SAS mechanical hard disk.

As for the software environment, the operating system used in the header node and the slave node was Centos6.7. Each node was installed with JDK 1.8.0, Hadoop 2.6.0, Spark 2.0.1 and Scala 2.1.0. The programming language used in the experimental development were Java and Scala. IDEA IDE was used in the development environment.

The experimental data set used here was generated by TPC-H benchmarking tool. TPC-H was a standard for DBMS performance testing developed by TPC (American Transaction Processing Performance Council), a non-profit organization. It could be used to simulate the business application environment in real life and was widely used to evaluate the comprehensive performance of decision support system. The test data was generated using the command *dbgen -s x* provided by TPC-H where *x* represented the data size in GB to be generated. In addition to the test data set, TPC-H also provided 22 query statements Q1-Q22. Users utilized command *qgen* provided by TPC-H to generate these queries for performance testing.

5.2. Spark Shuffle Intermediate Data Caching Test. After completing the memory allocation of Spark Shuffle cache layer in each node, several representative queries in TPC-H were chosen which were Q1, Q5, Q9, Q18 and Q19. Q1 and Q19 had larger input data and fewer intermediate data. Q5, Q9 and Q18 had fewer input data and larger intermediate data. The queries were programmed to be submitted to SSO system and the original Spark system, and the query execution time was recorded. The experimental results are shown in Figure 9.

Results shows that for Q5, Q9 and Q18 whose intermediate data was much larger than the input data, the intermediate data cache layer could solve the problem of high random disk *I/O* cost. The optimization effect was obvious. But for Q1 and Q19 whose input data was larger than the intermediate data, the reading of the input data accounted for the cast majority of disk *I/O*. The optimization effect of the intermediate data cache layer for those queries was relatively limited.

The change of disk *I/O* rate over the query process was shown in Figure 10. As can be seen from the figure, since the source data was stored sequentially, the process of reading input data produced centralized disk *I/O*. In both systems, the disk read rate was around 80 MB/S which is the peak performance of stable mechanical hard disk. During the running process, the program entered the Shuffle phase of reading and writing intermediate data. At this time, the original Spark system generated more random disk *I/O* and the disk read rate had a great fluctuation. It took a long time to complete the Shuffle process in the original Spark system. The SSO system used the intermediate data cache layer, and the intermediate data were read and written in memory. So, the Shuffle phase of SSO system did not generate disk I/O cost. The SSO system completed the query task faster than the original Spark system. Figure 11 showed that the memory usage of the two systems was similar in the first half of query process and both were steadily increasing. When entering the Shuffle phase, SSO system calculated the required memory size by query pre-analysis module and allocated the memory at each node by cache layer allocation module. Therefore, the memory usage of SSO system would increase instantaneously at this time. This part of memory would be released after the end of query. In the whole process of query, the main node had enough memory (64 GB), while the size of intermediate data generated by query Q9 was only 14.1 GB. It was possible to create a piece of memory space as intermediate data buffer. SSO system had a higher memory usage rate, which was one of the main reasons for the higher execution efficiency of SSO system.

5.3. Intermediate Data Correlation Merging Algorithm Test. By merging the intermediate data with the same fields, the size of intermediate data generated could be further reduced,

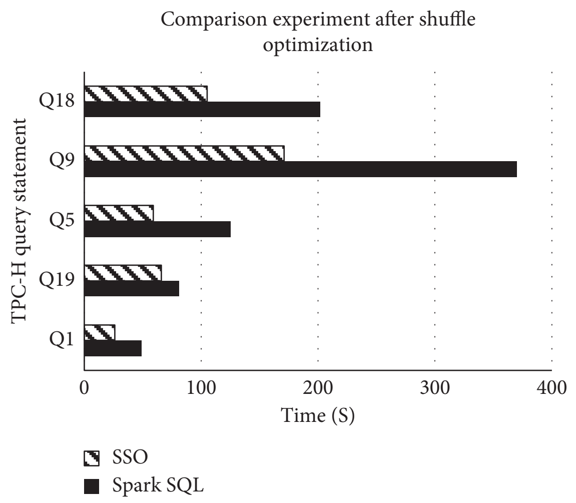

Figure 9: Comparison experiment before and after intermediate data caching.

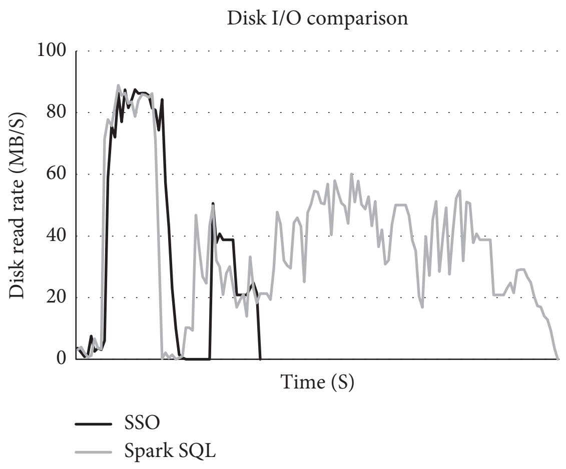

Figure 10: Disk *I*/*O* comparison.

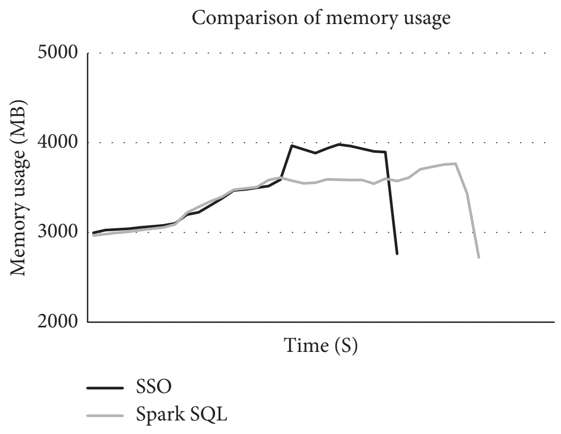

Figure 11: Comparison of memory usage.

thus reducing the amount of memory used by the intermediate data cache layer. To verify the effectiveness of the algorithm, in this section, query Q17 of TPC-H was selected to be submitted to SSO system and the original Spark SQL system respectively. Meanwhile, without using the intermediate data correlation merging algorithm, the original Spark SQL system was tested in two cases: using the intermediate data cache layer or not. The query execution time was recorded and shown in Figure 12.

Results showed that the execution time of SSO system and Spark SQL system with intermediate data cache layer was both shorter than that of Spark SQL system without any optimization, while the execution time of the former two systems tended to be the same. In order to compare the two systems, the monitoring results of the cache size used by each node were shown in Table 3.

The cache usage of SSO system with intermediate data correlation merging algorithm was less than that of Spark SQL system without optimization algorithm in each node. This was because by sharing the same fields l_partkey and l_quantity, intermediate data of 1.5 GB could be reduced and writing the data caused disk cost of 19.2 seconds. Redundant reads of field l_extendedprice and l_suppkey were also generated. Reading the 1.26 GB data caused disk *I*/*O* of 16.4 seconds. Comparison showed that the saved cost was greater

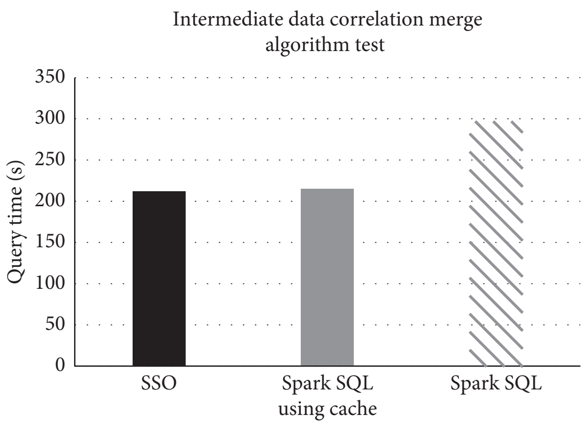

Figure 12: Intermediate data correlation merging algorithm test.

Table 3: Comparison of cache size with and without intermediate data correlation merging algorithm.

Id	1	2	3	4	5	6	7	8	9	10	11
Input data (MB)	256	896	384	1024	1792	1664	640	384	256	640	256
Spark buffer (MB)	227	792.2	340	908.8	1590	1476	568	340.8	227.2	568	227.2
SSO buffer (MB)	178	625.8	268	715.2	1251	1162	447	268.2	178.8	447	178.8

than the generated cost. So, in this case, the algorithm proposed would merge the same intermediate data. However, the intermediate data were read and written in memory after adopting the cache layer and the memory read/write rate was more than dozens of GB per second. On the premise that the cache layer cached all the intermediate data, the amount of cache usage had little effect on task execution efficiency. That was the main reason why the query time of the first two experiments tended to be consistent. But in the case of tight cluster memory resources, the intermediate data correlation merging algorithm could reduce the cache usage, thus saving memory resources effectively.

6. Conclusion

With the maturity of memory computing framework, Spark, as the representative of memory computing framework, has attracted more and more attention from enterprises and research groups. As a bridge between data analysts and Spark systems, Spark SQL plays an important role. To optimize the performance of Spark SQL query, the existing Spark SQL was improved and the SSO prototype system was developed. By adding the Spark Shuffle intermediate data cache layer, the high disk *I*/*O* cost caused by random reading and writing of intermediate data in Shuffle phase was reduced. In order to solve the problem of redundant read-write for intermediate data of Spark SQL, a cost-based correlation merging algorithm was proposed. It was used to determine whether to merge tasks with correlation by weighing the benefits and the extra costs of merging, thus improving the query execution efficiency. The experimental platform was built and the SSO system was developed. The benchmarking tool TPC-H was used to generate test data. The performance comparison with the existing Spark SQL system was carried out to verify the effectiveness of the work in this paper.

Data Availability

All data used to support the findings of this study are included within the article.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work is supported by the SGCC Science and Technology Program called "Research and Application of Key Technologies of Common Components of New Generation Power Grid Dispatch Control System Platform," and partially supported by Collaborative Innovation Center of Novel Software Technology and Industrialization and Collaborative Innovation Center of Wireless Communications Technology.

References

[1] X. Shi, M. Chen, L. He et al., "Mammoth: gearing hadoop towards memory-intensive mapreduce applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 8, pp. 2300–2315, 2015.

[2] M. Zaharia, M. Chowdhury, T. Das et al., "Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, USENIX Association, San Jose, CA, USA, April 2012.

[3] K. Shvachko, H. Kuang, S. Radia et al., "The hadoop distributed file system," in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–10, IEEE, Incline Village, NV, USA, May 2010.

[4] Apache HBase, http://hbase.apache.org/.

[5] V. K. Vavilapalli, A. C. Murthy, C. Douglas et al., "Apache hadoop yarn: yet another resource negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ACM, Santa Clara, CA, USA, October 2013.

[6] B. Hindman, A. Konwinski, M. Zaharia et al., "Mesos: a platform for fine-grained resource sharing in the data center," *NSDI*, vol. 11, p. 22, 2011.

[7] M. Zaharia, T. Das, H. Li et al., "Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters," *HotCloud*, vol. 12, p. 10, 2012.

[8] R. S. Xin, J. E. Gonzalez, M. J. Franklin et al., "Graphx: a resilient distributed graph system on spark," in *Proceedings of the 1st International Workshop on Graph Data Management Experiences and Systems*, ACM, New York, NY, USA, 2013.

[9] S. Agarwal, B. Mozafari, A. Panda et al., "BlinkDB: queries with bounded errors and bounded response times on very large data," in *Proceedings of the 8th ACM European Conference on Computer Systems*, pp. 29–42, ACM, Prague, Czech Republic, April 2013.

[10] X. Meng, J. Bradley, B. Yavuz et al., "Mllib: machine learning in apache spark," 2015, https://arxiv.org/abs/1505.06807.

[11] M. Armbrust, R. S. Xin, C. Lian et al., "Spark sql: relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pp. 1383–1394, ACM, Melbourne, Australia, June 2015.

[12] R. S. Xin, J. Rosen, M. Zaharia et al., "Shark: SQL and rich analytics at scale," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pp. 13–24, ACM, 2013.

[13] A. Thusoo, J. S. Sarma, N. Jain et al., "Hive: a warehousing solution over a map-reduce framework," *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1626–1629, 2009.

[14] D. Vengerov, A. C. Menck, M. Zait, and S. P. Chakkappen, "Join size estimation subject to filter conditions," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1530–1541, 2015.

[15] R. Singhal and P. Singh, "Performance assurance model for applications on SPARK platform," in *Proceedings of the Technology Conference on Performance Evaluation and Benchmarking*, pp. 131–214, Springer, Munich, Germany, September 2017.