# How Good is Query Optimizer in Spark?

Zujie Ren[1(✉)], Na Yun[1], Youhuizi Li[1], Jian Wan[2], Yuan Wang[3], Lihua Yu[3], and Xinxin Fan[3]

[1] School of Computer Science, Hangzhou Dianzi University, Hangzhou, China
renzju@gmail.com
[2] Department of Software Engineering, Zhejiang University of Science and Technology, Hangzhou, China
[3] Key Enterprise Research Institute of NetEase Big Data of Zhejiang Province, Netease Hangzhou, Network Co. Ltd., Hangzhou, China

**Abstract.** In the big data community, Spark plays an important role and is used to process interactive queries. Spark employs a query optimizer, called Catalyst, to interpret SQL queries to optimized query execution plans. Catalyst contains a number of optimization rules and supports cost-based optimization. Although query optimization techniques have been well studied in the field of relational database systems, the effectiveness of Catalyst in Spark is still unclear. In this paper, we investigated the effectiveness of rule-based and cost-based optimization in Catalyst, meanwhile, we obtained a set of comparative experiments by varying the data volume and the number of nodes. It is found that even when applied query optimizations, the execution time of most TPC-H queries were slightly reduced. Some interesting observations were made on Catalyst, which can enable the community to have a better understanding and improvement of the query optimizer in Spark.

**Keywords:** Spark SQL · Catalyst · Query optimization

## 1 Introduction

With the emergence of various types of big data frameworks, a group of data query processing systems have been developed, such as Apache Hadoop [1], Google Dremel [2], Cloudera Impala [3], and Apache Spark [4]. Spark supports processing structured data using either Spark SQL or DataFrame API [5–7]. Like relational database management systems, Spark implements a query optimizer, called Catalyst, which converts SQL-like queries into logical execution plans.

Query optimization techniques, including rule-based and cost-based optimization, have attracted a large number of scholars to study it [8–11]. However, few people have evaluated the effectiveness of query optimizer in Spark. Although query optimizer in relational databases can significantly accelerate the execution of SQL queries [12–15], the performance of query optimizer in Spark is still unclear. With the rapid development of Spark, Catalyst supports both rule-based and cost-based optimization since the version of Spark 2.2. A systematic evaluation of Catalyst will contribute to optimize the performance of Spark.


H. Gao et al. (Eds.): CollaborateCom 2018, LNICST 268, pp. 595–609, 2019.
https://doi.org/10.1007/978-3-030-12981-1_42

In this paper, we investigated the query execution efficiency for different optimization rules. A group of queries in TPC-H [16–18] are selected to evaluate rule-based and cost-based optimization. In the experiments, we varied both data volume and cluster scale to observe the query execution time. We found that the execution time were accelerated slightly for most query optimization rules. Optimization rules has slight effect on the optimization of SQL query executions.

## 2 Related Work

Query optimization has attracted plenty of research attention [19–22]. Many researchers focused on improving the effectiveness of optimization techniques. Lei *et al.* [23] investigated the quality of cardinality estimator in query optimizers of a group of DBMS, and found that all estimators routinely produce large errors. They found that exhaustive enumeration techniques can improve performance despite the sub-optimal cardinality estimates.

Kocsis *et al.* [24] proposed *Hylas*, a tool for automatically optimizing Spark queries in the source code by semantics-preserving transformation strategy. Liu *et al.* [25] proposed a prototype of query optimization based on cost model, and defined cost models for the common operations in relational queries. Zhang *et al.* [26] proposed an optimization scheme of partial bloom filter, it can reduce the amount of data in the shuffle stage and effectively improve the performance of equivalent connection.

Yang *et al.* [27] decided to enhance Spark SQL optimizer with detailed statistics information. This scheme is able to filter out most of the records in advance, which can reduce the amount of data in the shuffle stage and effectively improve the performance of equivalent connection.

Although a few research efforts have been put on query optimizers in Spark, the above papers are based on the improvement of optimization techniques or tools, and there is no systematic study on the optimization effect of Catalyst, it is still in infant stage. In this paper, we characterized the effectiveness of the query optimization in Spark, aiming to derive some design implications for improving the query optimizer in Spark.

## 3 Experimental Results

TPC-H benchmark are chosen to evaluate the query optimization performance of Catalyst. During the experiments, we selected a subset of TPC-H queries based on the optimization rules. Those queries include Q2, Q3, Q5, Q7, Q9, Q12, Q14, Q16, Q18, Q19 and Q22. The master and slave nodes in Spark cluster are configured with 128GB memory and 40 CPU cores.

We compared the execution time and tasks in cluster environments between optimization rules are used and not used, so as to observe the effectiveness of rule-based optimization and cost-based optimization framework in Catalyst.

### 3.1 Overview of Catalyst

Catalyst follows a typical structure of query optimizers. The main components of Catalyst and their functions are described as follows (Fig. 1).

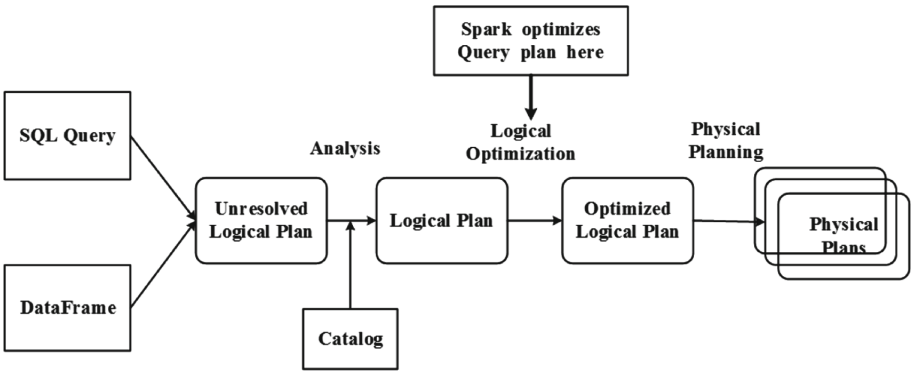



**Fig. 1.** The architecture of Catalyst.

- **SQLParser**–parses SQL statements, generates a syntax tree, and forms unresolved logical plans.
- **Analyzer**–combines the unresolved logic plan generated in the previous step with the data dictionary to bind and generate analyzed logical plans.
- **Optimizer**–applies rules to logical plans and expressions, merge and optimize tree nodes to obtain the optimized logical plans.
- **SparkPlanner**–transforms optimized logical plans into physical programs that can be recognized by processing.
- **CostModel**–selects the best physical execution plan based on some performance data.

As the kernel of Catalyst, Optimizer processes SQL queries based on the rules defined in the batches [6], including *CombineFilters*, *PushDownPredicate*, *LikeSimplication*, *CombineLimits*, *CombineUnions*, *ConstantFolding* and *NullProPagation* optimization rules.

However, the query plans automatically chosen by the Spark optimizer are not optimal, especially on the cost. In order to improve the quality, Yang *et al.* [27] decided to enhance Spark SQL optimizer with detailed statistics information. So that we can better estimate the number of output records and output size for each database operator.

### 3.2 Evaluation of Rules

**CombineFilters.** *CombineFilters* rule can recursively merge adjacent filter conditions. If this rule is not applied, the filter statements are carried out one by

one, as defined in the SQL queries. *Q2*, *Q3* and *Q18* in TPC-H are selected to drive the target system and observe the performance changes caused by *CombineFilters* rule. The results are shown in Table 1 and Fig. 2. (The prefix "U_" represents that use of the optimization rules, and the prefix of "UN_" means no use of optimization rules.) For Q2, there is only a slight differences in the number of stages and tasks, but for Q3 and Q18, they are completely identical.

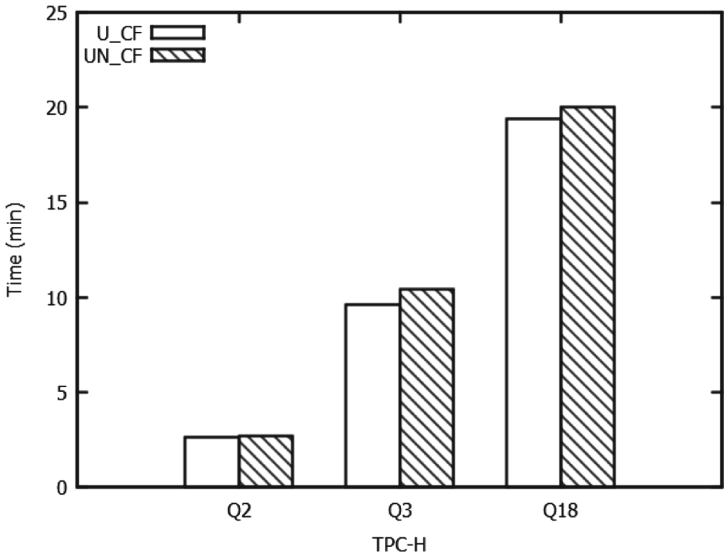
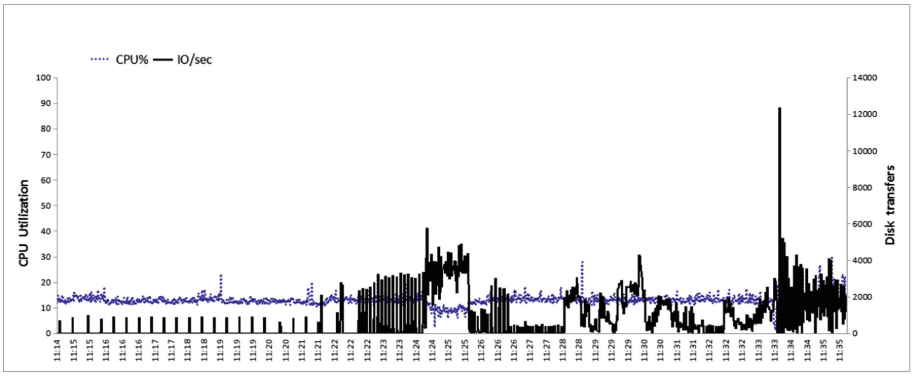



**Fig. 2.** Execution time changes when applying *CombineFilters*

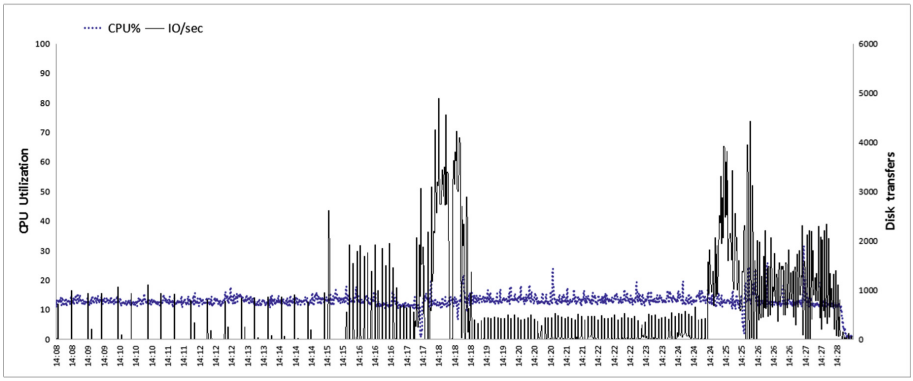
**Table 1.** The results when applying *CombineFilters* and not.

| Experiment cases | Division of stages | Number of tasks |
|---|---|---|
| *U-Q2* | 0∼16 | 2618 |
| *UN-Q2* | 0∼17 | 2620 |
| *U-Q3* | 0∼4 | 3377 |
| *UN-Q3* | 0∼4 | 3377 |
| *U-Q18* | 0∼7 | 6149 |
| *UN-Q18* | 0∼7 | 6149 |

For *CombineFilters* optimization rule, there are slight differences on the processing time of SQL statements (Fig. 2). However, I/O fluctuates and disk transfers are much frequent in the condition without *CombineFilters* rule. *CombineFilters* rule can reduce disk interaction in the optimization of *Q18* (Fig. 3).




(a) Q18 with *CombineFilters* rule




(b) Q18 without *CombineFilters* rule

**Fig. 3.** The resource utilization with and without *CombineFilters*.

**PushDownPredicate.** *PushDownPredicate* optimization rule can push the predicate in SQL statements into the subqueries, thereby reduce the number of subsequent data processing. We selected *Q5*, *Q7*, *Q16* of TPC-H to carry on experiments. For the same SQL statements, the results are shown in Table 2. For Q5, Q7, Q16, the number of stages and tasks is exactly the same when applying *PushDownPredicate* and not.

As shown in Fig. 4, the time consumed when not using *PushDownPredicate* rule is more than that of using the optimization rule in the optimization process for *Q5*. However, the processing time of SQL statements are almost same for *Q7* and *Q16*.

**LikeSimplification.** *LikeSimplification* optimization rule can simplify "*LIKE*" expression to avoid the full scan of tables with extra calculation burden. For example, it can optimize the sentence "*%N*"(*%N* represents the demo beginning with *N*) to "*StartsWith*" for operations. *Q2*, *Q9* and *Q14* of TPC-H are selected to drive experiments.

**Table 2.** The results when applying *PushDownPredicate* and not.

| Experiment cases | Division of stages | Number of tasks |
|---|---|---|
| *U-Q5* | 0∼11 | 4186 |
| *UN-Q5* | 0∼11 | 4186 |
| *U-Q7* | 0∼10 | 4184 |
| *UN-Q7* | 0∼10 | 4184 |
| *U-Q16* | 0∼0(Job0) 1∼5(Job 1) | 5(Job0) 1038(Job1) |
| *UN-Q16* | 0∼0(Job0) 1∼5(Job 1) | 5(Job0) 1038(Job1) |

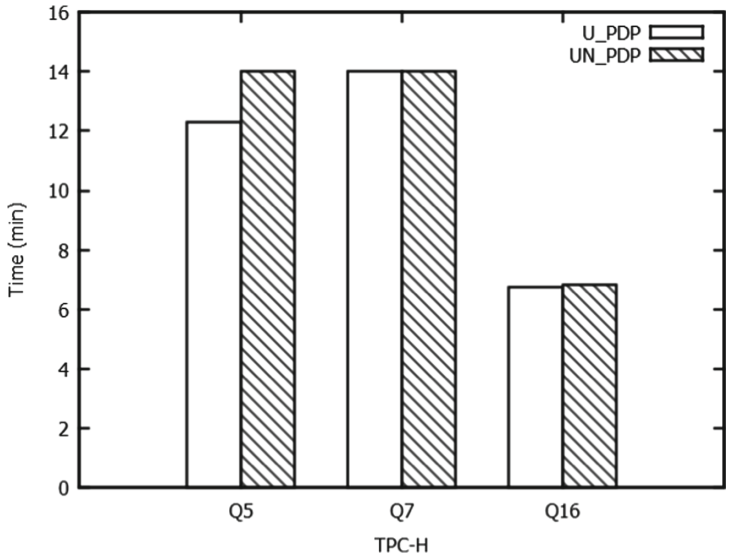



**Fig. 4.** Execution time changes when using *PushDownPredicate*.

**Table 3.** The results when applying *LikeSimplification* and not.

| Experiment cases | Division of stages | Number of tasks |
|---|---|---|
| *U-Q2* | 0∼16 | 2618 |
| *UN-Q2* | 0∼16 | 2618 |
| *U-Q9* | 0∼11 | 4548 |
| *UN-Q9* | 0∼11 | 4548 |
| *U-Q14* | 0∼3 | 2647 |
| *UN-Q14* | 0∼3 | 2647 |

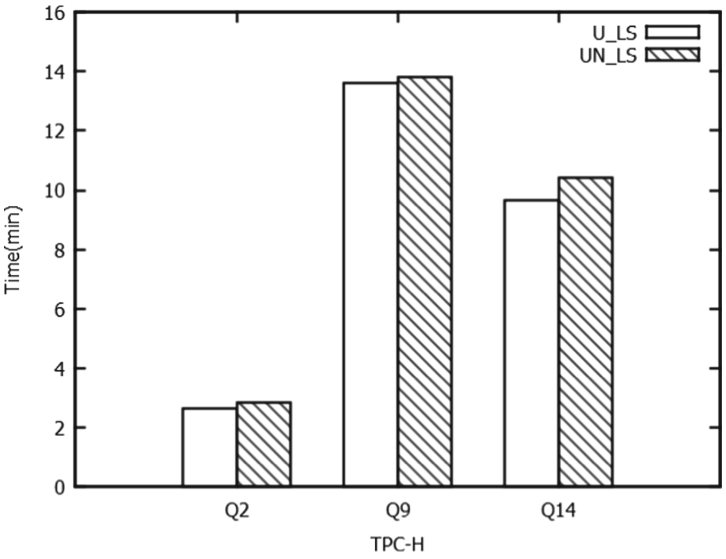



**Fig. 5.** Execution time changes when using *LikeSimplification* ("LS" refers to *LikeSimplification* rule).

Stages and tasks remain unchanged during the processing of performing *Q2*, *Q9*, and *Q14* (Table 3), Fig. 5 depicts the results for *Q2*, *Q9*, and *Q14*. The suffix of "LS" refers to *LikeSimplification* rule. The "*%N*" involved in SQL statements are optimized to "*StartsWith*" for operations when using *LikeSimplification* optimization rule. Figure 5 shows that the execution efficiency is slightly improved when using LikeSimplication.

### 3.3 Evaluation with Special Queries

In this section, we focused on the optimization strategies of other rules. TPC-H benchmark do not contain these rules in SQL statement. Same principles as those mentioned above, we selected representative SQL statements to do experiments, those queries include *CombineLimits*, *CombineUnions*, *ConstantFolding* and *NullPropagation*. The query are executed in cluster environments that use the corresponding optimization rules and do not use.

*CombineLimits* rule compares adjacent "*Limit*" statements in SQL, the small one retains and returns as a result, it can avoid counting "*Limit*" statements many times during the process of calculation. *CombineUnions* rule recursively merges adjacent "*Union*" statements. *ConstantFolding* rule can calculate expressions that are calculated directly in advance, there is no need to put expressions into the physical execution to generate objects to operate. *NullPropagation* rule replaces "*Null*" value, expressions that determine the value of "*Null*" are calculated at the logical stage, can avoid propagation of "*Null*" values on syntax trees.

**Table 4.** The results when applying optimization rules.

| Experiment cases | Division of stages | Number of task |
|---|---|---|
| U-*CombineLimits* | 0∼3(Job0) 4∼6(Job1) | 639(Job0)<br>2447(Job1) |
| UN-*CombineLimits* | 0∼3(Job0) 4∼7(Job1) | 639(Job0)<br>2647(Job1) |
| U-*CombineUnions* | 0∼2(Job0) | 2904(Job0) |
| UN-*CombineUnions* | 0∼1(Job0) 2∼2(Job1) 3∼3(Job2) | 2904(Job0) 4(Job1)<br>17(Job2) |
| U-*ConstantFolding* | 0(Job0) | 1(Job0) |
| UN-*ConstantFolding* | 0(Job0) | 1(Job0) |
| U-*NullPropagation* | 0∼1(Job0) | 2373(Job0) |
| UN-*NullPropagation* | 0∼1(Job0) | 2373(Job0) |

We executed the same SQL statements in cluster environments when the optimization rules are using and not. The results are shown in Table 4, more tasks are needed to perform under the condition that *CombineLimits* or *Combine Unions* rule is not used, but stages and tasks remain unchanged when applying *CombineUnions* and *NullPropagation* rule (Each job gets divided into smaller sets of tasks called stages that depend on each other, similar to the map and reduce stages in MapReduce).

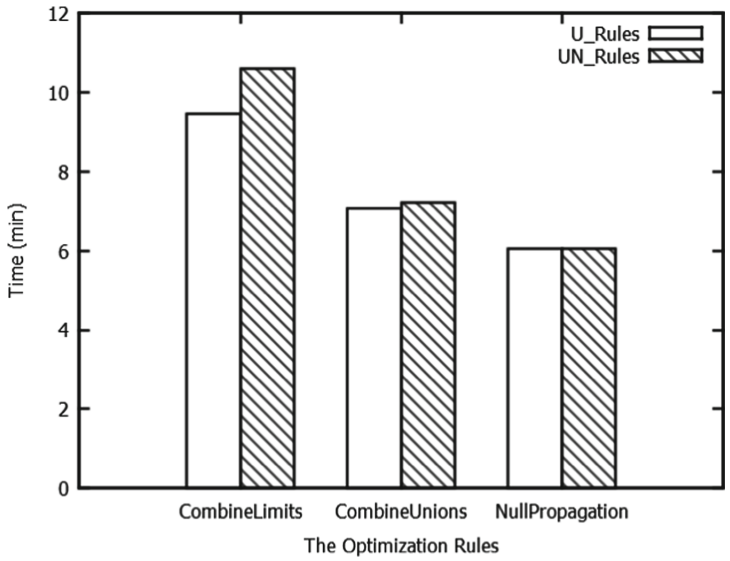
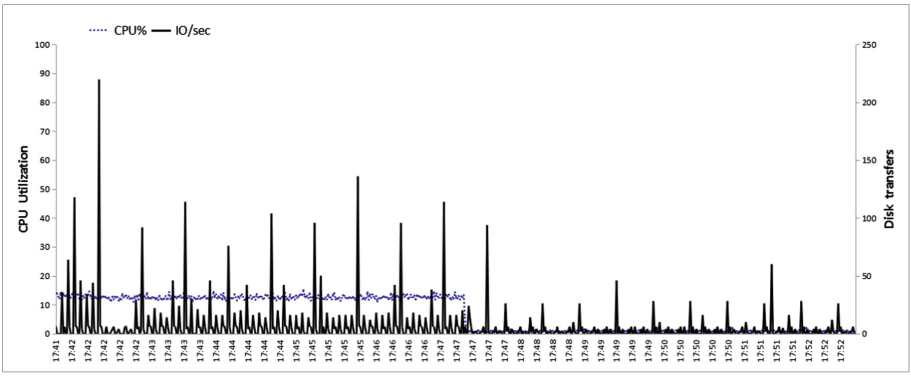
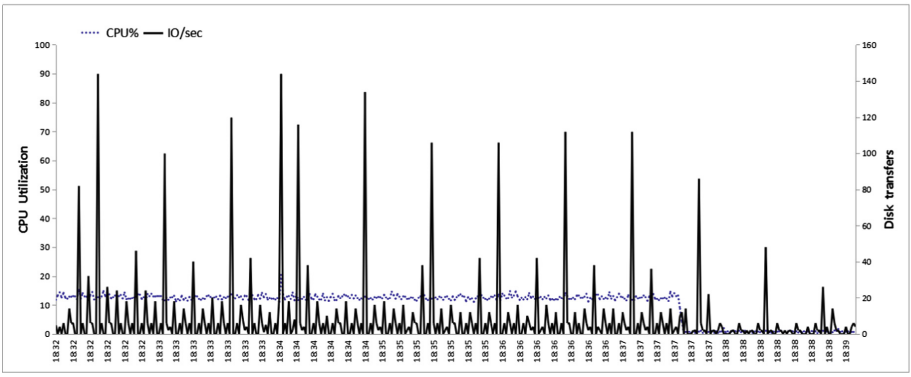



**Fig. 6.** Execution time changes when using optimization rules.




(a) With *NullPropagation* rule




(b) Without *NullPropagation* rule

**Fig. 7.** The resource utilization with and without *NullPropagation*.

More time are needed to perform under the condition that *CombineLimits* rule is not used, and there are slight differences for *CombineUnions* and *NullPropagation* (Fig. 6). But as far as resource consumption is concerned, more CPU and I/O resources are needed to process the same SQL statements without using the corresponding optimization rules (Fig. 7).

### 3.4 Varying Data Sizes

Spark implements cost-based optimization framework to improve the quality of query execution plan. In this section, we analyzed the optimization effects of CBO and RBO under different sizes of data.

The scala factor *(SF)* was set as 10 and 100, respectively. Evaluation queries include Q2, Q3, Q5, Q7, Q9, Q12, Q14, Q16, Q18, Q19 and Q22. The results are shown in Fig. 8. Meanwhile, we set SF = 10 and 100 when RBO is applied. Experiments are carried out on *CombineFilters* (Fig. 9a), *PushDownPredicate* (Fig. 9b) and *LikeSimplication* (Fig. 9c) rules.

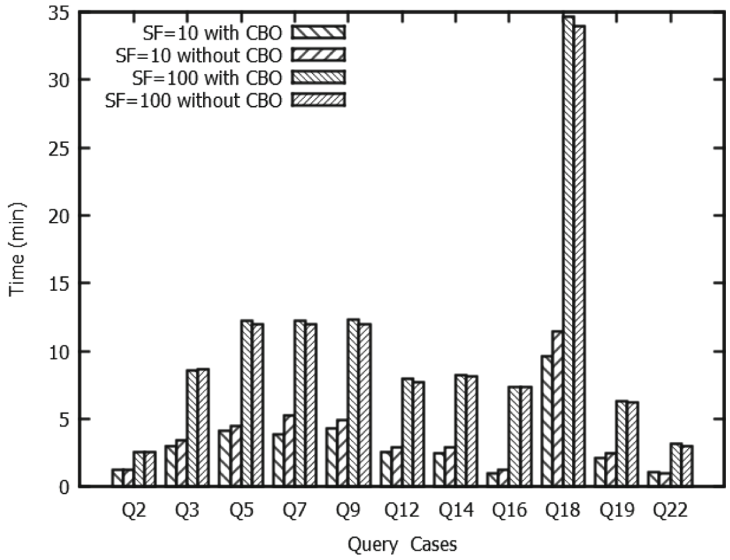



**Fig. 8.** Execution time changes with and without CBO.

The results are shown in Fig. 9. With the increase of the data volume, the processing time for the same SQL statements is increased correspondingly. For the same data scale, the execution time reductions are still slight.

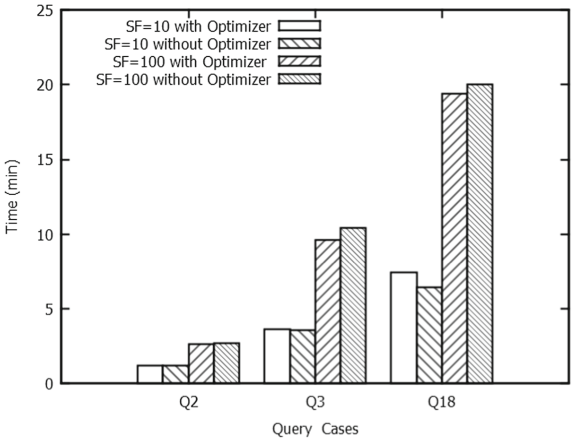
### 3.5 Varying Cluster Scale

In this section, we compared the optimization effects of CBO and RBO under different cluster scales. At the same time, we guaranteed that the amount of data processed on each slave node is up to 10G.

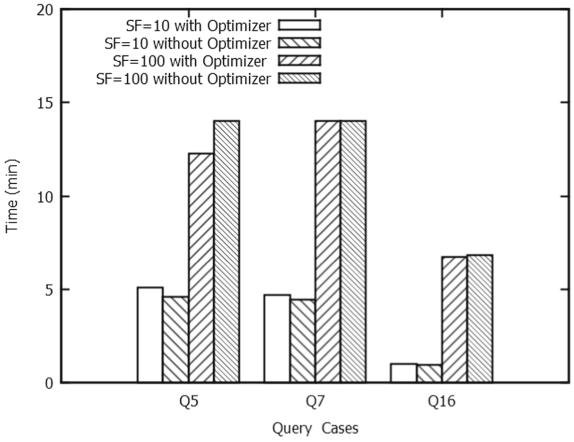
The number of slave nodes are ranged from 1 to 15. The rules of *CombineFilters*, *PushDownPredicate* and *LikeSimplication* are applied. The processing time results are shown in Fig. 10, which shows that the improvement achieved by *CombineFilters* rule for Q3 is slight, and there is a downward trend for Q9 with the increase of cluster scales. For Q7, the execution time is reduced if not applying optimization rule. Less time is spent without using the optimization rule.

Similarly, SQL queries with and without CBO framework are executed. The results of experiments are shown in Fig. 11. For Q12, the expansion of cluster scales has limited effect. For Q5, the SQL processing time has a downward trend without the usage of CBO framework. However, with the increase of cluster scales, the time needed to use CBO optimization rule is small for Q9.

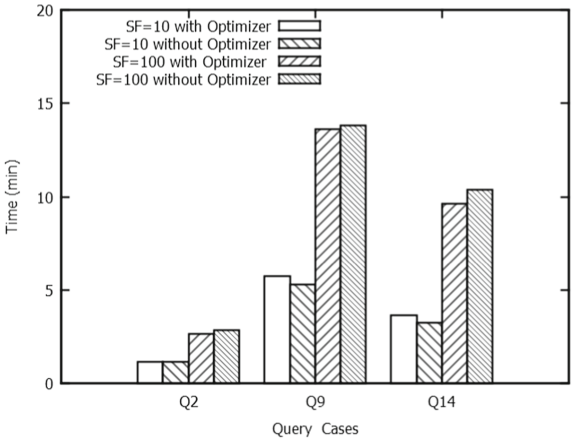
When the number of slave nodes varies from 1, 5, 10 to 15, neither rule-based optimization nor CBO framework have much effect. Rule-based optimization and CBO framework have different optimization effects for different SQL statements. However, the differences are not obvious.




(a) *CombineFilters* optimization rule




(b) *PushDownPredicate* optimization rule




(c) *LikeSimplication* optimization rule

**Fig. 9.** Execution time with different data volumes.

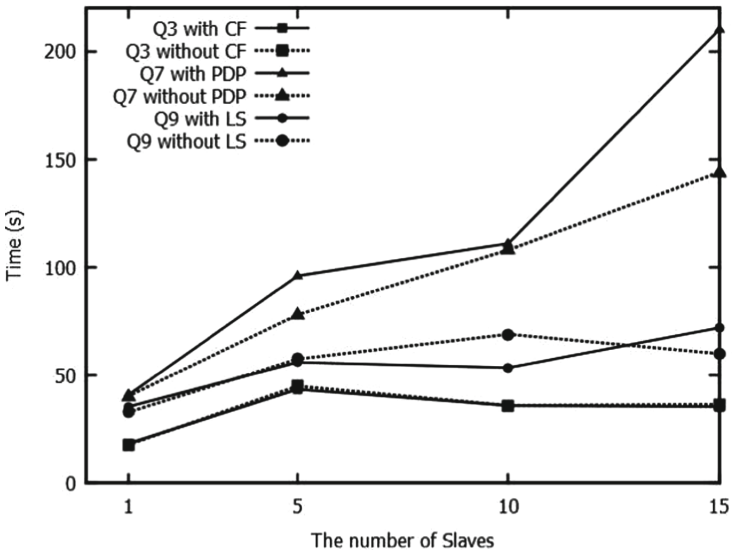



**Fig. 10.** Execution time changes when using rule-based optimization.

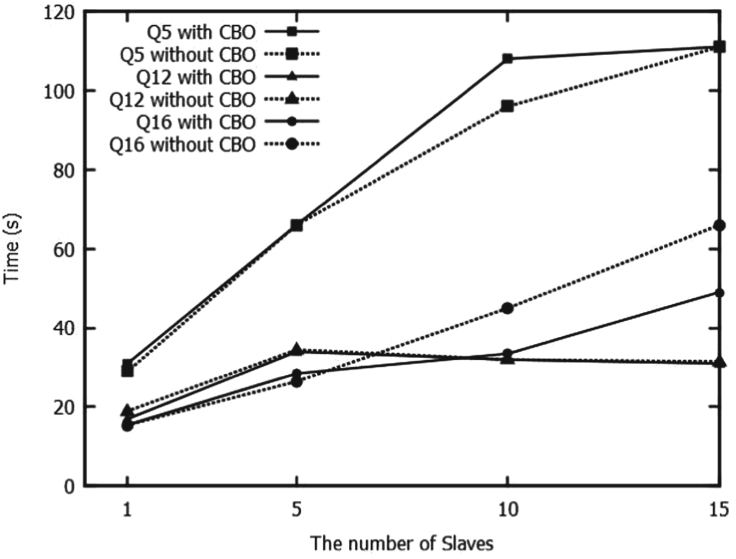



**Fig. 11.** Execution time changes when using cost-based optimization.

## 4 Discussion

Based on the experimental results, the resource consumption by Spark SQL in runtime can be realized and choose the optimization strategy better, so that we can further decrease the system overhead and query time. To achieve that, we must understand the optimization strategy of optimization rules and its behaviors. The written SQL statements should be standardized, and conform to the syntax requirements of the optimization method. Thus, faster and more accurate query optimization of SQL statements can be achieved.

As the kernel of Catalyst, optimizer is responsible for optimizing the syntax tree, it contains many rules defined in the batches, including *CombineFilters*, *PushDownPredicate*, *LikeSimplication*, *CombineLimits*, *CombineUnions*, *ConstantFolding* and *NullProPagation* optimization rules. The corresponding optimization rules are summarized in Table 5.

**Table 5.** The list of optimization rules.

| The optimization rules | Introduction of corresponding optimization strategies |
|---|---|
| CombineFilters | Recursively merge adjacent filter conditions |
| PushDownPredicate | Push the predicate in SQL statements into the subquery, reduce the number of subsequent data processing |
| LikeSimplication | Simplify "LIKE" expression to avoid the full scan of tables |
| CombineLimits | Compare adjacent "Limit" statements, and return the small one. |
| CombineUnions | Recursively merge adjacent "Union" statements |
| ConstantFolding | Calculate expressions in advance that are calculated directly |
| NullPropagation | Replace "Null" value |

After evaluating the Catalyst optimizer, we investigated the effectiveness of the optimization rules and cost-based optimization in Catalyst. We derived the following implications:

- The query optimizer has little effect on execution time reductions. Different SQL statements correspond to different optimization rules. However, optimization strategies are not always the optimal choice in optimizer.
- For different SQL statements, rule-based optimization and CBO framework have little effect under different cluster scales.
- For the same SQL statements, the processing time grows with the increase of the workload data volume. However, even if the amount of data grows, the reduction of execution time will not become obvious.

## 5 Conclusion

In this paper, the optimization effects of rule-based and cost-based optimization framework in Catalyst optimizer in Spark were studied. We evaluated their optimization performance under various queries. At the same time, some comprehensive validation experiments was carried out by varying the data volume and cluster scale. The results show that even if query optimization rules are applied, the execution time of most benchmark queries were slightly reduced, and optimization rules have slight effect on the executing of SQL statements.

**Acknowledgement.** This work is supported by Key Research and Development Program of Zhejiang Province (No. 2018C01098), and the Natural Science Foundation of Zhejiang Province (NO. LY18F020014).

## References

1. Taylor, R.C.: An overview of the Hadoop/MapReduce/HBase framework and its current applications in bioinformatics. BMC Bioinform. **11**, S1 (2010)
2. Melnik, S., et al.: Dremel: interactive analysis of web-scale datasets. Proc. VLDB Endow. **3**(1–2), 330–339 (2010)
3. Ducarme, P., Rahman, M., Brasseur, R.: IMPALA: a simple restraint field to simulate the biological membrane in molecular structure studies. Proteins Struct. Funct. Bioinform. **30**(4), 357–371 (1998)
4. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: cluster computing with working sets. In: USENIX Conference on Hot Topics in Cloud Computing, p. 10 (2010)
5. Salloum, S., Dautov, R., Chen, X., Peng, P.X., Huang, J.Z.: Big data analytics on apache spark. Int. J. Data Sci. Anal. **1**(3–4), 145–164 (2016)
6. Armbrust, M., et al.: Spark SQL: relational data processing in spark. In: SIGMOD 2015, pp. 1383–1394. ACM (2015)
7. Zaharia, M., et al.: Apache spark: a unified engine for big data processing. Commun. ACM **59**(11), 56–65 (2016)
8. Ma, J., et al.: Logical query optimization for cloudera impala system. J. Syst. Softw. **125**, 35–46 (2017)
9. Naacke, H., Curé, O., Amann, B.: SPARQL query processing with apache spark. arXiv preprint arXiv:1604.08903 (2016)
10. Graefe, G.: The cascades framework for query optimization. IEEE Data Eng. Bull. **18**(3), 19–29 (1995)
11. Esawi, A.M.K., Ashby, M.F.: Cost-based ranking for manufacturing process selection. In: Batoz, J.L., Chedmail, P., Cognet, G., Fortin, C. (eds.) Integrated Design and Manufacturing in Mechanical Engineering, pp. 603–610. Springer, Dordrecht (1999). https://doi.org/10.1007/978-94-015-9198-0_74
12. Wu, J.-M., Zhou, J.: Research of optimization rule of SQL based on oracle database. J. Shaanxi Univ. Technol. (2013)
13. Antoshenkov, G., Ziauddin, M.: Query processing and optimization in oracle RDB. VLDB J. Int. J. Very Large Data Bases **5**(4), 229–237 (1996)
14. Chaudhuri, S.: An overview of query optimization in relational systems. In: Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, pp. 34–43. ACM (1998)

15. Herodotou, H., Babu, S.: Profiling, what-if analysis, and cost-based optimization of mapreduce programs. Proc. VLDB Endow. **4**(11), 1111–1122 (2011)
16. Chiba, T., Onodera, T.: Workload characterization and optimization of TPC-H queries on apache spark. In: IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 112–121. IEEE (2016)
17. Liang, W., Zheng, Y.: TPC-H analysis and test tool design. Comput. Eng. Appl. (2007)
18. Transaction processing performance council. http://www.tpc.org
19. Ioannidis, Y.E.: Query optimization. ACM Comput. Surv. (CSUR) **28**(1), 121–123 (1996)
20. Roy, P., Seshadri, S., Sudarshan, S., Bhobe, S.: Efficient and extensible algorithms for multi query optimization. ACM SIGMOD Rec. **29**, 249–260 (2000)
21. Graefe, G., DeWitt, D.J.: The EXODUS Optimizer Generator, vol. 16. ACM (1987)
22. Barbas, P.M.: Database query optimization, 21 January 2014. US Patent 8,635,206
23. Leis, V., Gubichev, A., Mirchev, A., Boncz, P., Kemper, A., Neumann, T.: How good are query optimizers, really? Proc. VLDB Endow. **9**(3), 204–215 (2015)
24. Kocsis, Z.A., Drake, J.H., Carson, D., Swan, J.: Automatic improvement of apache spark queries using semantics-preserving program reduction. In: Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion, pp. 1141–1146. ACM (2016)
25. Liu, C.: Research on SparkSQL query optimization based on cost model (2016)
26. Zhang, L.: Research on query analysis and optimization based on spark system (2016)
27. Wang, Z.: Spark issue. https://issues.apache.org/jira/browse/SPARK-16026