Research on Runtime Query Optimization Technology of Spark SQL

Yong Zhao and Rong Chen

***Abstract*—Spark SQL uses SQL to describe the task of data analysis and optimizes it according to the theory of query optimization, which effectively improves the efficiency of execution. However, the query optimization of Spark SQL still has the following shortcomings at present. It requires the operator to collect statistics information explicitly through the collection commands of statistics information. In addition, because the collected statistics information is not accurate enough, the optimization effect will be poor. To solve the above problems, this paper proposes an algorithm that collects statistics at runtime and optimizes the query adaptively. The algorithm uses Bloom Filter Pruning to prune data that does not meet the join conditions before a join operation is executed. In order to estimate the cardinality of the intermediate relationship of the join more accurately, the algorithm uses AMS Sketch and Bloom Filter to estimate the cardinality. Finally, the algorithm generates an optimization algorithm of the join based on the connection of graph. Experiments have proven that the BFP algorithm can prune the input of join by up to 12% without considering the join order. The algorithm for join plan generation can produce the optimal plans in 14 out of 18 queries without pre-collecting statistics data and save execution time by up to 31%, and the time spent on the collection of statistics information is no more than 5% of the total execution time.**

***Index Terms*—Query optimization, spark SQL, bloom filter, sketch.**

I. Introduction of Spark SQL and Query Optimization Theory

Spark SQL uses SQL-like syntax as a high-level data manipulation API, which greatly narrows the difficulty of data analysis. Trummer *et al*. proposed a method for credible estimation of the cardinality of the relationship within a certain cost range, and proposed a query optimization algorithm based on accurate estimation of cardinality [1]. Literature [2] proposed an algorithm named APPROXJOIN based on Spark, which uses Bloom Filter and stratified sampling algorithm to reduce data transmission and collects statistics information of output data. Avnur *et al*. proposed an adaptive runtime query optimization algorithm named eddies [3]. Agarwal *et al*. proposed an adaptive optimizer named RoPE for parallel data computing systems. RoPE is the first optimizer proposed collecting statistics information such as the number of unique values in a column and hotspot data during the execution of parallel computing tasks [4]. Literature [5] proposed a query optimizer named Optimus. Optimus runs on the parallel computing framework Dryad [6], and realizes adaptive query optimization by dynamically modifying the execution plan based on statistics information. Literature [7] proposed a method called Universe Sampling to solve the problem of a decrease in the estimation of ratio while using a general sampling method to estimate the predicate selection rate during join. Karanasos *et al*. proposed an optimization algorithm called pilot run. Experiments show that this idea has at least twice the performance improvement compared to the optimal left deep tree query written manually. But the algorithm cannot improve the efficiency of the query when the columns used for the join are not primary keys or foreign keys, when the query contains complex predicates or custom functions, and when there is a correlation between the joined columns [8]. To sum up, since Spark SQL currently requires users to explicitly execute statistics information collection commands according to their needs [9], there is still a large space for exploration in optimization of join algorithm, estimation of intermediate cardinality of join, generation of join plan and other aspects.

The query execution process in a database system based on the relational model consists of three steps: syntax analysis, logical plan generation and physical plan generation. Query optimization mainly occurs in two steps: logical plan and physical plan. As to physical query optimization, Selinger *et al*. proposed a cost-based model, estimating the CPU and IO costs of several physical execution plans according to the collected statistics, and selecting the physical execution plan with the lowest cost as the final execution plan [10].

II. Runtime Query Optimization Algorithm Design

This section introduces the runtime query optimization algorithm proposed in detail. The algorithm includes three aspects. 1) Using Bloom Filter to prune the input of the join before the join is executed. 2) Using statistics collected by AMS Sketch and Bloom Filter to replace existing statistics information about Spark SQL, and this goes for a more accurate estimation of the cardinality of the intermediate relations of the join. 3) Using a graph-based runtime algorithm for join plan generation.

A. Prune the Input of Join Based on Bloom Filter

In this section, a join algorithm named BFP Join is proposed. The Bloom Filter is used to prune the input data during the execution of the join, so as to decrease the amount of data transferred between nodes during the shuffle process and improve the efficiency of the execution. Due to its distributed characteristics, Spark SQL needs to redistribute the data on both sides of the join to several nodes through shuffle operation before performing the join operation, so that the data with the same key value falls into the same node,

Manuscript received May 20, 2021; revised August 14, 2021.

The authors are with the University of Electronic Science and Technology of China, Chengdu, 611731, China (e-mail: yongzhao1@qq.com).

DOI: 10.7763/IJCTE.2022.V14.1305

and the join of two large tables is decomposed into the join of several small tables.

BFP Join can be divided into unilateral pruning and bilateral pruning according to different pruning methods.

1) Unilateral pruning. In unilateral pruning, the smaller table in the join is used to construct a Bloom Filter. The filter is used to prune the larger table. The specific execution process can be divided into the following steps. a) Comparing the sizes of the two tables participating in the join, and setting the smaller table as R and the larger table as S. Use the value of the join attribute column in R to construct a Bloom Filter. b) Using the join attributes of each partition of R to construct Bloom Filter partitions. c) Converging the Bloom Filter partitions to the master node and merge them into a single Bloom Filter. d) Sending the merged Bloom Filter to all partitions where S is located. e) Using the Bloom Filter to prune the data of each partition of S. After completing the construction of the Bloom Filter, when the Bloom Filter is used to prune the large table, the merged Bloom Filter needs to be distributed to each partition where the large table is located.

2) Bilateral pruning. When the two sides participating in the join are large, we can consider establishing a Bloom Filter on both sides separately, and pruning both sides of the join at the same time. The execution steps of bilateral pruning are similar to that of unilateral pruning. Let relations participating in the join to be R and S respectively. The specific steps are as following. a) Constructing Bloom Filter partition in each partition based on join properties for R and S in parallel. b) Combining the Bloom Filter partitions in each partition of R and S into the final Bloom Filter and converging it to the Driver node, which is called BF_R and BF_S. c) Broadcasting BF_R and BF_S to all partitions where S and R are located respectively. d) Using BF_R and BF_S to prune the input of S and R respectively.

3) Join algorithm selection. Spark SQL uses the table size estimated based on statistics to identify which join method to use. When the size of the party participating in the join is less than the size of broadcast Join limit, Broadcast Join can be used to avoid shuffling of larger data sets. When the conditions of Broadcast Join are not met, Shuffle Hash Join will be tried. The Shuffle Sort Merge Join algorithm is used as the ultimate default join strategy. After pruning the input of join using Bloom Filter, the size of input data of the join will change. At this time, the size of the table participating in the join can be re-estimated, and a more efficient join method can be selected.

B. Method for Estimating Intermediate Result Cardinality of Join

1) Estimation of intermediate result cardinality based on AMS sketch. AMS Sketch is mainly used to estimate the F_2 value of the data stream frequency vector [11]. There is a zero matrix C in the figure. After inputting a data 1 into this AMS Sketch, the update operation of AMS Sketch is triggered, so that the elements in the red box are updated. Then calculating the inner product of the row vector and itself for each row of the matrix, and taking the median of all inner products as an estimation of the frequency vector F_2.

2) Estimation of intermediate result cardinality based on bloom filter. Bloom Filter can not only prune the joined input but also estimate the size of the intermediate result. Assume that each discrete value of the attribute is uniformly distributed, that is, the number of records for each discrete value is the same. Therefore, it is only necessary to obtain the number of discrete values in the result and the average number of records corresponding to all discrete values. Eq. (1) expresses the estimated value of the number of discrete elements in the Bloom Filter.

$$uniq\_count = -n\frac{\ln(1-\frac{n}{m})}{k} \tag{1}$$

Suppose that when the Bloom Filter is used to perform bilateral pruning on $R \infty_k S$, the total data of R and S are $size(R)$ and $size(S)$, respectively, and after pruning, they are $size(R\_pruned)$ and $size(S\_pruned)$. $uniq(R)$ is the number of unique values in R, $uniq(R\_prune)$ is the number of unique values in *R pruned*, which satisfies Eq. (2) and Eq. (3).

$$\frac{uniq(R)}{uniq(R\_pruned)} = \frac{size(R)}{size(R\_pruned)} \tag{2}$$

$$\begin{aligned} uniq(R\_pruned) &= uniq(S\_pruned) \\ &= uniq(Join\_result) \end{aligned} \tag{3}$$

Finally, the number of records $Avg(R)$ and $Avg(S)$ corresponding to each unique value in R and S is used as the number of discrete values, and the final estimation for the intermediate result is Eq. (4):

$$size(result) = \frac{size(R\_pruned) \times size(S)}{uniq(S)} \tag{4}$$

C. Runtime Join Plan Generation Algorithm

Unlike the join plan based on dynamic programming [10], the following requirements need to be met to generate the join plan at runtime. 1) Collecting statistics information according to requirements during the execution process without relying on statistics information obtained in advance. 2) Traversing all its possible joins, when there are multiple joins in a relationship, and estimating the cost of each join method. 3) Minimizing the time cost of collecting statistics information as much as possible, so as not to cause a longer time than the wrong join plan. Based on the above requirements, this paper proposes a graph-based runtime join plan generation algorithm based on greedy algorithm. The algorithm uses a graph to represent the join plan selection problem that needs to be solved, uses the end of the graph to represent the relationship to be joined, and uses the edge of the graph to represent a join that needs to be executed. Each join is about to merge the two endpoints. When the algorithm is executed, the graph will shrink to an isolated point, which is the result of the join. The execution of the algorithm is divided into the following two steps.

1) Initialization process. Initializing all relations as nodes of the graph and all join operations as edges between nodes.

*2) Plan generation process. T*his is an iterative process, which can be divided into the following three steps. a) When estimating the size of the result of the join between the two

endpoints of each edge in the graph, if the two tables corresponding to this edge have not changed during the last join and the cost has been calculated, there is no need to recalculate. b) Selecting the edge with the smallest estimated join result to perform the join operation, and joining the edges associated with the two nodes before the join to the node after the merge. c) Updating the joined nodes and edges. First, the node list needs to be updated to remove the joined nodes, and then the join cost needs to be recalculated for the corresponding edges before the two nodes participating in the join.

III. Experiment Design and Result Analysis

The experiment uses TPC-DS data set [12].

A. Experiment Design and Analysis of BFP Join

The experiment based on BFP Join consists of two experiment groups and a control group. Experiment group 1 uses the smaller relation to the input of the join to construct the Bloom Filter, and uses the constructed Bloom Filter to filter the larger relations in the join. In experiment group 2, we generate Bloom Filter for both datasets participating in the join, and the Bloom Filter is used to filter both sides of the join simultaneously. For control group, we use unmodified Spark SQL to execute the query. Sizes of 1GB, 5GB and 20GB data are generated built on the TPC-DS dataset for the query. This article designs three custom query statements to analyze the cost of the join process separately, as showed in Table I. The three custom queries are executed 10 times, the one that consumes the most time and the one that consumes the least time are removed, and the average of the remaining 8 times is taken as the final result.

TABLE I: Three Custom Query Statements

Cutom Query	Describtion
Custom Query 1	The total amount of returned goods sold in the store
Custom Query 2	The 10 highest-selling products sold in the store
Custom Query 3	The number of items that are returned through the store after purchasing items from catalog shopping channels

From Fig. 1, it can be seen that experiment group 1 has a certain performance improvement when running Custom Query 1 on data sets 1GB, 5GB, and 20GB.

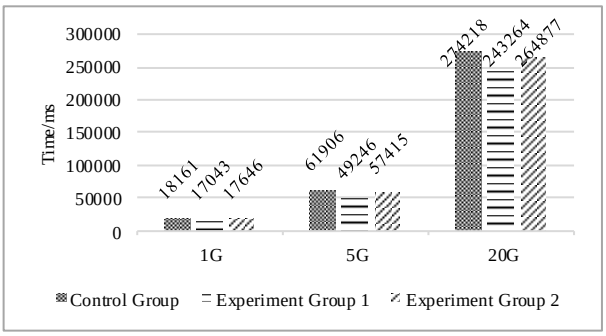

Fig. 1. Comparison of execution results of Custom Query 1.

The increases are 6%, 20%, and 12% respectively. Compared with experiment group 1, the execution time of experiment group 2 is slightly long, but there is still a certain improvement compared to the control groups. From Fig. 2, in Custom Query 2, the control groups perform better than the experiment groups in all cases. Since Custom Query 2 uses table *store sales* and table *item* to join, the intersection of the two tables is large, so most of the data cannot be pruned in the pruning stage. As a result, the experiment groups are slower than the control groups because of the cost of collecting statistics. Among them, the performance loss of experiment group 2 is the largest when using 20GB data, reaching 7%.

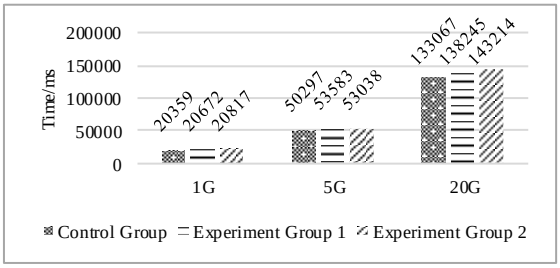

Fig. 2. Comparison of execution results of Custom Query 2.

From Fig. 3, the experiment groups in Custom Query 3 has a better optimization effect than the control groups. When 1GB of data is used, the performance is improved by 22%. After analysis, it is found that the original Shuffle Hash Join is small enough after pruning to call the Broadcast Join to join, which directly avoids shuffle for larger data sets and brings a greater performance improvement.

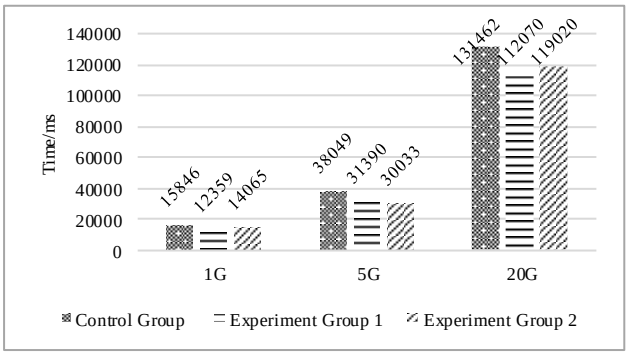

Fig. 3. Comparison of execution results of Custom Query 3.

When using 5GB of data to execute Custom Query 2, the optimization effect of experiment group 2 slightly exceeds that of experiment group 1. This is because as the data set becomes larger, the bilateral pruning algorithm eliminates most of the data, which significantly reduces the time cost of the join process. The above three sets of experiments respectively perform the BFP Join algorithm test for the cases where there is a small intersection between the two joined parties and a large intersection between the two joined parties, and the two joined parties have a small intersection and both joined parties can be effectively filtered by the Bloom Filter. Experiment results show that when the intersection of the two sides of the join is small, using the BFP Join algorithm can smoothly reduce the execution time of the join. When there is a large intersection between the two joined parties and the input data cannot be effectively cut, the time cost of BFP Join can also be tolerated. In the last case, the use of bilateral pruning can more effectively prune the input data of the join, and the optimization effect is better than that of unilateral filtering.

B. Experiment Design and Analysis of Runtime Query Optimization Algorithm

The experiment of runtime query optimization algorithm is intended to verify that a better execution plan can be generated without pre-collection of statistics information, and the time cost of collecting statistics information and making plan selection is controllable. Based on the above purpose, four groups of control experiments are set up. Control group 1 uses the order in the *From* clause of the query to execute joins. This is the default behavior of Spark SQL when the query optimizer is not enabled. Control group 2 uses the optimal join order based on the dynamic programming algorithm, control group 2 will be used as a benchmark to compare whether the experiment groups can obtain the optimal execution order. Experiment group 1 uses AMS Sketch to estimate the cardinality of the intermediate relationship of the join, and uses the graphbased join plan generation algorithm to select the order of join. Experiment group 2 uses Bloom Filter to estimate the cardinality of the intermediate relations of the join, and also uses the graph-based join plan generation algorithm to select the order of join. The running time comparison of each group in Query 7 is shown in Fig. 4.

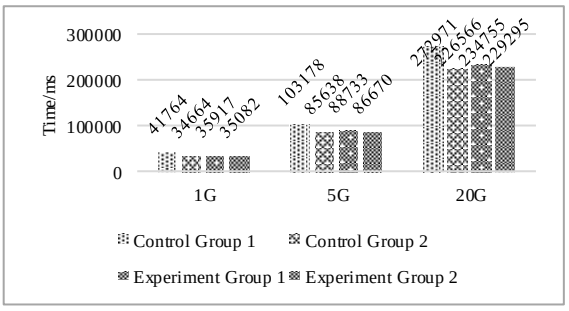

Fig. 4. The execution time comparison for Query 7.

In Query 7, the execution time of control group 1 is the longest, and control group 2 with the adjusted join order achieves the highest execution efficiency, which is 17% faster than control group 1. Compared with control group 1, the performance of experiment group 1 and experiment group 2 increase by 14% and 16% respectively.

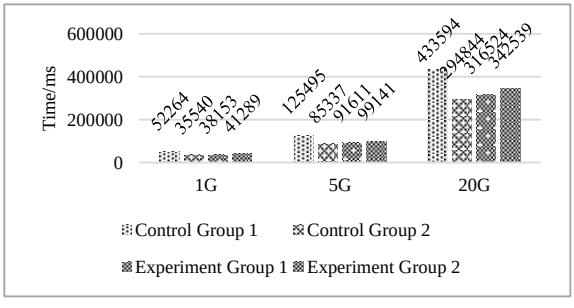

Fig. 5. The execution time comparison for Query 17.

Query 17 is more complicated than Query 7, from Fig.5, the average performance improvement of control group 2 has reached 32% compared to control group 1. Experiment group 1 and control group 2 produce the same join plans, and compared with control group 1, the average optimization effect has reached 27%. Compared with control group 1, experiment group 2 has an average optimization effect of only 21%. Because the collected statistics information is not accurate enough, Broadcast Join cannot be used to perform the join between table *date_dim* and *store_returns*. In query 25, the join plan for control group 2 is the same as query 17. From Fig. 6, in query 25, the performance improvement of control group 2 reaches 34% compared to control group 1, and the performance improvements of experiment group 1 and experiment group 2 are 29% and 20% compared to control group 1, respectively. Experiment group 1 accurately generates the best join plan again, while experiment group 2 generates the correct join sequence when the data is 1GB and 5GB, and the optimization effect reaches 31%.

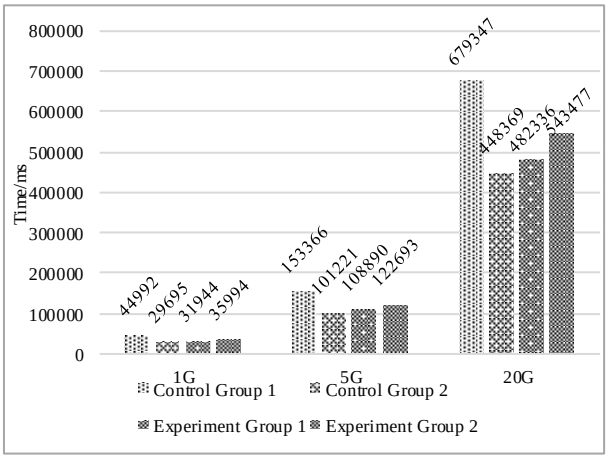

Fig. 6. The execution time comparison for Query 25.

The above experiments show that the query optimization algorithm based on runtime statistics information collection proposed in this article has a certain optimization effect. On the one hand, it can spend less time collecting statistics information to generate the optimal join plan, on the other hand, as the amount of data increases, the proportion of time spent on statistics information collection becomes smaller and smaller, and the optimization effect becomes more and more obvious. It should be pointed out that in this experiment, control group 2 directly uses the best join order to compare with the runtime query optimization algorithm proposed in this article. This can only be used as the upper limit of the optimization effect of all optimizers, and does not represent the effect of the query optimizer integrated by the current Spark SQL.

C. Analysis of Comparative Experiment Results of Related Work

This section compares the query optimization algorithm proposed in this article with the optimization method in literature [8], and uses it as control group 3.

As can be seen from Fig. 7, in all three queries, compared with control group 1 without any optimization, the two experiment groups and control group 3 have a significant performance improvement. At the same time, there is a certain performance loss compared with control group 2 that directly executes the best join plan without any additional cost. In query 7, experiment group 1 and experiment group 2 have a very small performance improvements compared to control group 3. Through analysis of the execution plan, it is found that the three experiment groups all produce the optimal execution plans. The execution time of the three

groups is also slightly different because of the different costs of the optimization processes. In query 17, the optimization effect of experiment group 2 has obvious advantages over experiment group 1 and control group 3. Compared with control group 1, the performance of experiment group 2 has increased by 25% and the performance improvements of experiment group 1 and control group 3 are only 17% and 15%, respectively. In query 25, the optimization effects of experiment group 1, experiment group 2, and control group 3 are 29%, 26%, and 20%, respectively. Among them, experiment group 1 and control group 3 have generated correct join plans.

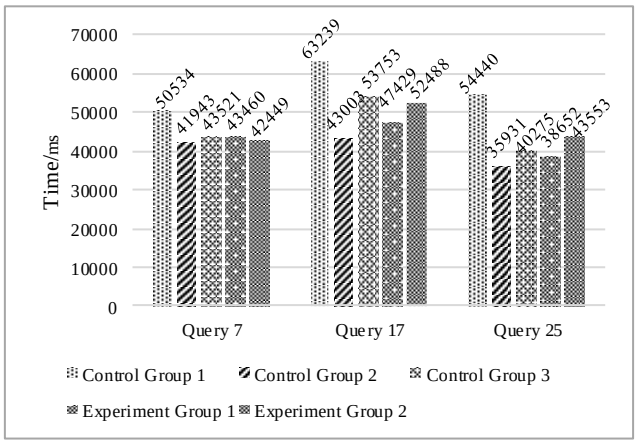

Fig. 7. Comparison of related work results.

Through these experiments, it can be seen that the runtime query optimization algorithm proposed in this paper overall has lower optimization cost while it can generate the optimal join plan compared with the algorithm in literature [8]. In addition, while the method in [8] can produce wrong execution plans, the proposed Bloom Filter-based method can still generate the correct plans.

IV. Conclusion

Combining runtime adaptive query optimization theory and Spark SQL implementation principles, this paper proposes an optimized prototype that collects statistics information at runtime and adaptively generates a query plan iteratively, removing the dependence on the user's pre-executed statistics information. 1) Using Bloom Filter to prune the input of join. Before the join is executed, the disjoint part of the data is pruned by Bloom Filter, thus reducing the network cost of data transmission in the shuffle process and the cost of subsequent sorting and hash table construction. 2) Using AMS Sketch and Bloom Filter to estimate the intermediate cardinality of join, and the algorithm process and execution cost are set out in detail. 3) Proposing a graph-based join optimization algorithm based on the above two strategies, collecting statistics information and adaptively adjusting the execution plan during the execution of a query, and converting the execution process of the join into a process of graph shrinkage. This paper uses TPC-DS dataset to test the algorithms above. The experiments prove the effectiveness of the statistics information collection algorithm proposed in this article. For future work, we look at ways to develop global optimal solutions.

Conflict of Interest

The authors declare no conflict of interest.

Author Contributions

Assoc. Prof. Yong Zhao proposed the idea, conducted the research, and wrote the paper. Rong Chen reviewed the article of whole research.

Acknowledgment

Thanks to Chenfei Liu for his support and contribution to the work of this paper.

References

[1] I. Trummer, "Exact cardinality query optimization with bounded execution cost," in *Proc. the 2019 International Conference on Management of Data*, 2019, pp. 2–17.
[2] D. L. Quoc, I. E. Akkus, P. Bhatotia *et al*., "Approxjoin approximate distributed joins," in *Proc. the SoCC '18 ACM Symposium on Cloud Computing*, October 2018, pp. 426–438.
[3] R. Avnur and J. M. Hellerstein, "Eddies continuously adaptive query processing," in *Proc. SIGMOD*, 2000, pp. 261–272.
[4] S. Agarwal, S. Kandula, N. Bruno, *et al*., "Re-optimizing data-parallel computing," in *Proc. NSDI*, 2012.
[5] Q. Ke, M. Isard, and Y. Yu, "Optimus a dynamic rewriting framework for dataparallel execution plans," in *Proc. EuroSys*, 2013, pp. 15–28.
[6] M. Isard, M. Budiu, Y. Yu *et al*., "Dryad distributed dataparallel programs from sequential building blocks," *SIGOPS Oper. Syst. Rev*., vol. 41, no. 3, pp. 59–72, 2007.
[7] S. Kandula, A. Shanbhag, A. Vitorovic *et al*., "Quickr lazily approximating complex adhoc queries in bigdata clusters," in *Proc. SIGMOD*, 2016, pp. 631–646.
[8] K. Karanasos, A. Balmin, M. Kutsch, F. Ozcan *et al*., "Dynamically optimizing queries over large scale data platforms," in *Proc. SIGMOD*, 2014 pp. 943–954.
[9] Databricks Inc., *Analyze Table Command*, 2019.
[10] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin *et al*., "Access path selection in a relational database management system," in *Proc. SIGMOD*, 1979, vol. 79, pp. 23-34.
[11] N. Alon, Y. Matias, and M. Szegedy, "The space complexity of approximating the frequency moments," *Data Basis; Randomized Algorithms; Space Complexity*, pp. 20-29, 1996.
[12] R. O. Nambiar and M. Poess, "The making of tpc-ds," in *Proc. the 32nd International Conference on Very Large Data Bases (VLDB) Endowment*, 2006, pp. 1049-1058.

Yong Zhao is an associate professor at the University of Electronic Science and Technology of China. He received his Ph. D. from the University of Chicago under the supervision of Prof. Ian Foster. His research interests include blockchain, big data, data intensive science and workflow.

Rong Chen received the B.E. degree from Chongqing University of Posts and Telecommunications. She is currently pursuing the M.E. degree in University of Electronic Science and Technology of China. Her research interests include cloud computing and big data.