ACORE: A Query Optimization Approach for Spark SQL Based on Cost Model and Markov Prediction Model

Lanxin Su* and Huiyong Liu

State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing, Beijing, 100000, China

*Corresponding author’s e-mail: sulx511@bupt.edu.cn

Abstract. Spark SQL is Spark’s module for working with structured data. However, there exist two problems that may impact its query execution efficiency: 1) Users may attempt to cache datasets used multiple times in a query to speed up the execution, but due to the predicate pushdown optimization and the overhead of writing cached data, the cached plan may turn out to have worse performance. 2) When using Spark dynamic resource allocation strategy, the release of idle executors would lead to the loss of cached data, introducing recalculation cost. To address these, we present an approach, named ACORE, which adaptively cache datasets and optionally release executors to optimize query execution of Spark SQL. The approach includes two models: a cost model and a Markov chain prediction model. The cost model is applied to estimate the execution cost for cached and un-cached plans, and decide which one to be the actual execution plan. The Markov prediction model is used to predict the changing trend of intervals between queries, guiding to optionally release executors to save cached data. We test the efficiency of our approach with data generated by TPC-H tool and the experimental results show that the execution performance can be improved by up to 51%.

1. INTRODUCTION

Spark is an open-source, distributed processing system for big data workloads, with built-in modules for streaming, SQL, machine learning and graph processing [1]. Spark SQL is Spark’s module for working with structured data, which integrates relational processing with Spark’s functional programming API. It conveniently blurs the lines between RDDs (Spark’s distributed datasets) and relational tables, allowing users to intermix SQL commands querying external data with complex analytics within one application [2]. Catalyst, the optimizer of Spark SQL, performs algebraic optimization for SQL queries submitted by users and generates Spark workflow for execution [3].

However, there exist two problems that may impact the efficiency of Spark SQL:

1) Cache policy. Users may attempt to cache datasets that are used multiple times in a query to speed up the execution. However, in some cases this may not work as expected. When a dataset is cached, the predicate pushdown operation may accordingly change, making execution time even longer than the un-cached plan. Besides, if the cached table is large, the cost of writing cached data can’t be ignored. This may also impair the execution performance.

2) The release of executor with cache data. Dynamic resource allocation is a Spark feature that allows for adding and removing executors dynamically to match the workload. This mechanism is suitable for Spark long-lived applications, such as Spark SQL. However, releasing the idle executors would also lose cached data. If the next query uses the missing cached data, there generates extra cost for recalculation.
To optimize the execution of Spark SQL, [4] proposed a cost model that estimated whether the cached plan has a lower cost. The cost was estimated by summing up cost of all stages. They assumed that all the stages are executed serially, and the cost of writing cache is not taken into account. [5] aimed at optimizing the implementation of join operation. They presented a method which estimated the cost of physical plans by summing up cost of all the operators. [6] designed and implemented an intermediate data cache layer between the underlying file system and the upper Spark core to reduce the cost of random disk I/O. [7] constructed a non-active duration prediction model based on Markov theory. They assumed that all tables have been cached before users submitting queries, which was not that practical. This paper aims to optimize the cache and releasing executor policy to improve the performance of Spark SQL. Our main contributions are as follows:

- Propose a cost model for Spark physical plan. When a SQL statement is submitted, datasets with cache potential will be detected and then there generate two execution plans (cached VS uncached). The cost model is applied to estimate which plan has a lower cost and the lower one would be the actual execution plan. With this model, we can always make the right decision on whether to cache to speed up the query execution.
- Optimize Spark’s dynamic resource allocation strategy with Markov prediction model. When a SQL query completes, we use the Markov model to predict the interval between the next query’s arrival time and the current query’s finish time, optionally release executors according to the prediction results.
- Present an approach, named ACORE, which applies the above two models to adaptively cache datasets and optionally release executors to improve Spark SQL query execution performance.

2. PROBLEM STATEMENT

2.1. Cache Problem

Consider a SQL query as follow:

```
select l_shipmode from orders, lineitem
where o_orderkey = l_orderkey and o_orderpriority = '2-HIGH'
union select l_shipmode from orders, lineitem
where o_orderkey = l_orderkey and l_shipmode in ('MAIL', 'SHIP')
```

It can be seen that the two join operations are identical, naturally we may intend to cache the join result set to accelerate the execution. Sometimes caching does bring performance improvement, however, in this case it may not work as expected. Figure 1 shows the optimized logical plan for the initial statement and for where the join result set is cached. We can see that when it is cached, the filter operation is not pushed down to before the join operation, this results in larger datasets involved in join operation, therefore may lead to longer execution time, as join is a costly operation.

Figure 1. (a) is the optimized logical plan for the initial statement, (b) is the optimized logical plan where join result set is cached.
What’s more, the cost of writing cached data may be greater than the cost saved by using the cache, especially when the cached data is too large. So we need a method which can estimate the cost of cached and un-cached plans and do the right decision on whether to cache.

2.2. Spark Dynamic Resource Allocation Strategy
Spark Dynamic Resource Allocation allows for adding and removing executors dynamically to increase resource utilization. In Spark SQL, it can acquire executors from the cluster manager whenever users submit SQL queries and remove idle executors if no queries are executed. However, cached data is saved in executor, as the release of the executor, cached data would also be lost. If the next query uses the missing cached data, there generates extra cost for recalculation which could have been avoided.

To solve this we use the Markov chain model to predict the changing trend of query interval. With the prediction results, the ACORE approach optionally releases idle executors to save cached data.

3. COST MODEL
Our goal is to estimate the execution cost for different query plans and then cache datasets wisely to maximize execution efficiency. We choose the physical plan to apply the cost model.

The cost of a physical plan can be represented as the sum of reading table, writing cache and execution cost of all the non-leaf operators in the plan tree:

\[ Cost_{total} = Cost_{read\ table} + Cost_{write\ cache} + \sum_{i=1}^{k} Cost_{opi} \]  

For the un-cached plan, \( Cost_{write\ cache} = 0 \).

In the following part we first give the result set size estimation of the operator, then based on it we present the execution cost estimation of all kinds of operators. This part is based on [5][6][7], and we have made some modifications to make it more suitable for our research.

3.1. Result Set Estimation of Operator
In a physical plan tree, leaf nodes represent scanning table, obviously their result sets size is the size of input table. For un-leaf nodes, their input set is result set carried out by the operations in sub-tree, and their result set size depends on the input set size and operator itself.

(a) Selection
\[ |D_{out}(filter)| = \beta_{filter} \times |D_{in}| \]  

\( \beta_{filter} \) represents the proportion of the number of tuples that meet filter condition to the total number of tuples in original relation. We can estimate it by sampling the table.

(b) Join
\[ |D_{out}(L_{join=\ R_{join}})| = \beta_{join} \times |D_{L}| \times |D_{R}| \]  

\[ \beta_{join} = \frac{1}{\max(\text{diff}_{L_{join}} \times \text{diff}_{R_{join}})} \]  

\( |D_{L}| \) and \( |D_{R}| \) represent the number of tuples of L, R. We assume that in the join operation, all values of the attribute with fewer different values appear in the range of another attribute. Based on this we can easily estimate \( \beta_{join} \). through (4).

(c) Group By
\[ |D_{out}(\text{group by } a, b, \ldots n)| = \beta_{groupby} \times |D_{in}| \]  

\[ \beta_{groupby} = \frac{\min\{|D_{in}|, \text{distinct}(a) \times \text{distinct}(b) \times \ldots \times \text{distinct}(n)|}{|D_{in}|} \]  

\( \text{distinct}(i) \) represents the number of different values of attribute i. If there is a histogram of attribute i, we can estimate \( \text{distinct}(i) \) by summing up the number of different values in each interval. Otherwise we set \( \beta_{groupby} \) with default value 0.05.
3.2. Cost Estimation of Operator

For those operators which don’t involve shuffle (filter, selection, etc.), we only consider CPU calculation cost; for those with shuffle, such as join, there would be disk and network cost brought by shuffle. Cost mode parameters and functions are shown in Table 1.

| Parameter          | Description                                      |
|--------------------|--------------------------------------------------|
| disk               | disk I/O cost                                    |
| $HDFS_{read}$      | the cost of reading from HDFS                    |
| $cache_{read}$     | the cost of reading cached data                  |
| $cache_{write}$    | the cost of writing cached data                  |
| network            | network transmission cost                        |
| $CPU_{op}$         | CPU cost of operation op                         |
| nodes              | the number of nodes in the cluster               |
| $|T|$               | the number of tuples of relation T               |
| $createHash(T)$    | the cost of creating hash table for relation T   |

(a) Cost of Shuffle
Firstly, the map node sorts the RDD according to the key in parallel. This process will generate multiple disk files and bring disk I/O overhead. Then, these files are merged and sorted to produce the final shuffle file. This process would bring CPU sort cost. Finally the reduce nodes would fetch these files, bringing network overhead. Suppose there are $n$ map nodes, then the cost of shuffle can be represented as:

$$\text{Cost}_{\text{shuffle}}(T, n) = \text{Cost}_{\text{i/o}} + \text{Cost}_{\text{cpu}} + \text{Cost}_{\text{network}}$$

$$= 4 \times \text{disk} \times \frac{|T|}{n} + \text{CPU}_{\text{sort}} \times \frac{|T|}{n} + \text{network} \times |T|$$

(b) Cost of Scanning Relation
When relation $T$ has been cached, the cost of scanning $T$ is:

$$\text{Cost}_{\text{read}}(T) = cache_{\text{read}} \times |T|$$

When relation $T$ hasn’t been cached:

$$\text{Cost}_{\text{read}}(T) = HDFS_{\text{read}} \times |T|$$

(c) Cost of Selection
The selection operation (i.e. where) is mapped to the filter operator in Spark, which is a narrow dependency operator, only involving CPU calculation cost.

$$\text{Cost}_{\text{filter}}(T) = CPU_{\text{boolean}} \times |T|$$

(d) Cost of Projection

$$\text{Cost}_{\text{project}}(T) = CPU_{\text{project}} \times |T|$$

(e) Cost of Broadcast Hash Join
This applies to the join operation that one of the relation is particularly small. The small relation is distributed with broadcast, and then perform hash join on each node. We assume that $L$ is the small relation, and $R$ is evenly distributed in the cluster.

$$\text{Cost}_{\text{broadcast join}}(L, R) = \text{nodes} \times \text{network} \times |L| + CPU_{\text{hashjoin}} \times \frac{|R|}{n} + createHash(L)$$

As the calculation in each node are parallel, so we only consider one node’s CPU cost when performing hash join.

(f) Cost of Shuffle Hash Join
Suppose there are $N_L$, $N_R$ map nodes for relation $L$, $R$, and $M$ reduce nodes.
\[\text{Cost}_{\text{shuffle join}}(L, R) = \text{Cost}_{\text{shuffle}}(L, N_L) + \text{Cost}_{\text{shuffle}}(L, N_R) + \frac{\text{CPU}_{\text{hash join}} \times |D_{\text{out}}(L_{Ljoin=Rjoin} R)|}{M}\]  

(g) Cost of Sort Merge Join  
This applies to the join with two large relations. Firstly repartition L, R with shuffle, meanwhile sort them by key. Then perform merge join on each node.

\[\text{Cost}_{\text{sort merge join}}(L, R) = \text{Cost}_{\text{shuffle}}(L, N_L) + \text{Cost}_{\text{shuffle}}(L, N_R) + \frac{\text{CPU}_{\text{merge join}} \times |D_{\text{out}}(L_{Ljoin=Rjoin} R)|}{M}\]  

(h) Cost of Group and Aggregation  
Firstly perform shuffle on relation T, then reduce nodes pull the data, and performs the aggregation operation. We assume that data is evenly distributed on M reduce nodes after shuffle.

\[\text{Cost}_{\text{groupby}}(T) = \text{Cost}_{\text{shuffle}}(T, N) + \frac{\text{CPU}_{\text{aggregation}} \times |T|}{M}\]  

4. MARKOV CHAIN PREDICTION MODEL  
A Markov chain model is a stochastic model used to model pseudo-randomly changing systems. It is assumed that future states depend only on the current state, not on the events that occurred before it [8]. We believe that in Spark SQL interactive query, the changing trend of new interval (time between the finish time of current query and submission time of next query) is irregular, it is only related to current interval, as user is likely to query the same tables in adjacent SQL statements.

We consider the changing trend of query interval as the state in the Markov process. The definition of state space is:

\[X = \{x_1, x_2, x_3\}\]  

(16)  
\(x_1\) represents the query interval becomes longer, \(x_2\) represents the interval remains unchanged, \(x_3\) represents shorter.

Suppose current state is \(x_i\), and after the transition, the possible state is \(x_j\) (\(x_i, x_j \in X\)). Then the possibility of transition from \(x_i\) to \(x_j\) can be calculated as:

\[p_{ij} = \frac{N_{x_i x_j}}{\sum N_{x_i}}\]  

(17)  
\(N_{x_i x_j}\) represents the times of transition from \(x_i\) to \(x_j\) in historical data. \(\sum N_{x_i}\) represents all possible times that transfer from \(x_i\) in historical data. Once we get \(p_{ij}\), then we can define one-step state transition matrix \(P\):

\[P = \begin{bmatrix} p_{11} & p_{12} & p_{13} \\ p_{21} & p_{22} & p_{23} \\ p_{31} & p_{32} & p_{33} \end{bmatrix}\]  

(18)  

Once current query is finished, the ACORE approach constructs \(P\), to predict the next most probable state to predict how the coming query interval changes.

5. ACORE APPROACH  
Based on the cost model and Markov model proposed above, the process of our ACORE approach is as follows:

When a new query is submitted:

Step (1). Generate query plan tree and detect dataset that is used more than once, which is considered to have caching potential. As in [5] we only consider the one closest to the root node to simplify calculation.

Step (2). Generate two execution plans, one is cached plan, where the dataset found in step (1) is cached, and the other is un-cached plan, which does not make any changes to the original query statement.
Step (3). Apply the cost model to estimate execution cost of their physical plans. The one with lower cost will be the actual execution plan.

Step (4). After execution, if the cached dataset is an intermediate result (such as join result set), then invoke Spark API to remove the cached data. If the cached data is a table, go to Step (5).

Step (5). Construct one-step state transition matrix $P$. Predict the change of the query interval. If the query interval becomes shorter or remains according to the matrix, do not release the executor when it expires.

In Step (1)-(3), the approach applies the cost model to assure always doing the right decision on whether to cache to accelerate the execution of one query. In Step (4), if the cached data is a table, we consider it might be reused in next query, so it is necessary to predict the change of coming interval. If the interval becomes shorter or remains, the approach would keep the executors to save the cached data, which might improve the performance of next query.

6. EXPERIMENTS

Our experimental environment consists of three virtual machines, the configuration is shown in Table 2. The experiment data is generated from TPC-H tool.

Table 2. Hardware and software environment

|                  | Intel(R) Core(TM) 2.7GHz |
|------------------|--------------------------|
| CPU              | 6G                       |
| Memory           | 6G                       |
| Operating System | CentOS 7                 |
| JDK Version      | 1.8.0_121                |
| Hadoop Version   | 2.7.7                    |
| Spark Version    | 2.4.0                    |

6.1. Experimental results of cost model

We first verify the effectiveness of our cost model. We construct 10 SQL queries based on the data generated by TPC-H. Each query has intermediate results or tables with caching potential (That is, they are used more than once in a query). When a SQL query is submitted, we detect this kind of datasets, generate two execution plans (cached VS un-cached), and apply the cost model to estimate which plan has lower cost. Experimental results are shown in Table 3 and Figure 2.

Table 3. Experimental results for cost model accuracy verification

| Queries | Cache Decision Made by Cost Model | Execution Time of un-cached plan | Execution Time of cached plan |
|---------|-----------------------------------|----------------------------------|-------------------------------|
| q1      | not to cache                      | 115s                             | 183s                          |
| q2      | not to cache                      | 84s                              | 120s                          |
| q3      | not to cache                      | 38s                              | 64s                           |
| q4      | cache                             | 70s                              | 102s                          |
| q5      | not to cache                      | 96s                              | 45s                           |
| q6      | cache                             | 108s                             | 43s                           |
| q7      | cache                             | 90s                              | 47s                           |
| q8      | not to cache                      | 56s                              | 102s                          |
| q9      | cache                             | 37s                              | 26s                           |
| q10     | not to cache                      | 66s                              | 78s                           |
Figure 2. Comparison of execution time between cached and un-cached plan

It can be seen that, most of the time the cost model could guide to make the right decision on whether to cache. For q1~q4, in their un-cached plans, with predicate pushdown, the filter operation greatly reduced the input data size of join operation, but in cached plan, the filter wasn’t pushed down to before the join operation. The result shows that the cost saved by caching join result set didn’t exceed the cost saved by pushing down filter operation, so the join result wasn’t cached, the execution time was much shorter. q5~q10 are cases where the same table appears multiple times in a query. It is shown that sometimes caching this kind of table does bring performance improvement. For q6, q7, one of the join tables was cached, and the join operation was realized in the way of broadcast join rather than sort merge join. It is known that broadcast join has lower execution cost, as shuffle is avoided, which is extremely costly. That’s why their cached plans were executed much faster than un-cached plans.

q4 is an example of wrong decision, this may due to the inaccuracy of the estimation of the join result sets size, as in order to simplify calculation we have made some assumptions of data distribution in join operation. And the cost of writing cache is set to a fixed value, we only consider the cached data to be saved in memory. This may not be suitable when the cached datasets is too large, part of which is written to disk.

This experiment validates the efficiency of our cost model, also it shows that indeed not in all cases caching will bring performance improvement, this relies on the SQL statement itself, the size of cached datasets, and hardware of cluster, etc. Therefore it is really necessary to apply such a cost model to estimate whether to cache to maximize execution efficiency.

6.2. Experimental Results of ACORE Approach
We test the efficiency of our ACORE approach. We choose 10 SQL statements, submit them at randomly generated intervals. If two statements are related, such as querying the same table, their submission interval will be relatively short, as user is likely to submit a series of related queries in a short period of time. Spark dynamic resource allocation mechanism is used in this experiment. The expiration time of the executor is set to 30s. Figure 3 shows the execution time of each statement with ACORE approach VS ordinary Spark dynamic resource allocation strategy.

It can be seen that the execution efficiency of q3, q4, q5, q8, q9, q10 was improved by about 30%, 26%, 48%, 33.3%, 14%, 51%, and 28% when using ACORE approach. In q3, a table is automatically cached to accelerate the execution, and the prediction result shows that the next interval would become longer, so the executors are released when expired to improve resource utilization. Also when executing q4 the approach has automatically cached table, making q4 execution time shorter. And the Markov model predicted that next interval would become shorter, so executors are not released when expired, therefore the performance of q5 was improved with cached data in executors. For q8, a table was used multiple times, ACORE approach cached it to accelerate execution; and the prediction results indicated
that next interval would become shorter, so the executors were preserved when expired. When executing q9 and q10, for ordinary Spark dynamic resource allocation strategy, the executors were new allocated so there’s no cached data to use, leading to longer execution time compared with ACORE.

Figure 3. Comparison of execution time between Spark dynamic resource allocation strategy and ACORE approach.

7. CONCLUSION
To optimize the query execution of Spark SQL query, we present an approach, named ACORE, which adaptively cache datasets and optionally release executors. This approach includes two models: a cost model, which is applied to decide which plan (cached VS un-cached) to be the actual execution plan, and a Markov prediction model, which is used to predict the changing trend of intervals between queries, guiding to optionally release executors to save cached data. Experimental results show that our approach has good performance, increasing the execution efficiency by up to 51%.

References
[1] Rohan Joseph. (n.d.) What is Spark? https://chartio.com/learn/data-analytics/what-is-spark/.
[2] Apache Spark (n.d.) Spark SQL, DataFrames and Datasets Guide. https://spark.apache.org/.
[3] Zhai M, Song A, Qiu J, et al. Query optimization Approach with Shuffle Intermediate Cache Layer for Spark SQL[C]//2019 IEEE 38th International Performance Computing and Communications Conference (IPCCC). IEEE, 2019: 1-6.
[4] Haim Cohen (2018) Smart Resource Utilization with Spark Dynamic Allocation. https://dzone.com/articles/spark-dynamic-allocation.
[5] Jun Chen. (2018). Research and implementation of Spark performance optimization for police data processing (Master's thesis, University of Electronic Science and Technology of China). https://kns.cnki.net/KCMS/detail/detail.aspx?dbname=CMFD201802&filename=1018991146.nh.
[6] Chunlei Liu. (2016). Research on Spark SQL query optimization based on cost model (Master's thesis, University of Electronic Science and Technology of China).
[7] M. Zhai, A. Song, J. Qiu, X. Ji and Q. Wu, (2019) Query optimization Approach with Shuffle Intermediate Cache Layer for Spark SQL, 2019 IEEE 38th International Performance Computing and Communications Conference (IPCCC). London. pp. 1-6.
[8] Yi Liang, Shifan Cheng, Shilu Chang, Fei Liu. (2018) Spark dynamic resource allocation strategy based on prediction[J]. Software Guide, 17(12):43-47.
[9] John Pulkka. (2013) Introducing Cost Based Optimizer to Apache Hive. https://issues.apache.org/jira/browse/HIVE-5775.
[10] L. Baldacci and M. Golfarelli. (2019) A Cost Model for SPARK SQL. IEEE Transactions on Knowledge and Data Engineering, vol. 31, no. 5, pp. 819-832.
[11] Kaizhen Cai. (2016) Research on SQL to SPARK query optimization mechanism (Master's thesis, Southeast University). https://kns.cnki.net/KCMS/detail/detail.aspx?dbname=CMFD201701&filename=1016325348.nh.

[12] Yutong Wan. (2018) Spark SQL query execution optimization technology research (Master's thesis, Southeast University). https://kns.cnki.net/KCMS/detail/detail.aspx?dbname=CMFD201901&filename=1019821810.nh.

[13] Xin Lian. (2018) Cost-based optimization of Spark SQL (Master's thesis, Chongqing University of Posts and Telecommunications). https://kns.cnki.net/KCMS/detail/detail.aspx?dbname=CMFD202001&filename=1019631475.nh.

[14] Chris I. (2019) Predicting the Weather with Markov Chains. https://towardsdatascience.com/predicting-the-weather-with-markov-chains-a34735f0e4df.

[15] Ajmal Sirajudeen. (2017) How to Predict Sales Using Markov Chain. https://blog.arkieva.com/markov-chain-sales-prediction/.