POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Executive Summary of the Thesis

Dynamic Query Optimization in Spark

Laurea Magistrale in Computer Science and Engineering - Ingegneria Informatica

Author: Antonio Pipita

Advisor: Prof. Emanuele Della Valle

Co-advisor: Andrea Picasso Ratto

Academic year: 2021-2022

1. Introduction

How to properly store, manage and query data has always been a relevant topic. The rise in both relevance and volume of data has prompted a paradigm shift in both storing and querying architectures, which moved from centralized to distributed systems. Throughout these changes, the question of how to optimize query procedures has stayed relevant. Amongst the various distributed query engines, Spark has maintained a role of relevance. Spark was born as an abstraction over MapReduce [3] based on the RDD abstraction [2] and, as time went on, distributed querying functionalities were added to it in the SparkSQL module [1]. In SparkSQL the Catalyst optimizer takes care of query optimization by combining the rule-based and cost-based optimization approaches. Traditionally, however, the Catalyst is only able to perform static optimizations, not being able thus to take into account changes of the data during the execution. This gap was bridged by the introduction of AQE, Adaptive Query Execution, which added dynamic optimization capabilities to the Catalyst [1].

AQE, in Spark 3.1.2, introduces three main capabilities: dynamically shifting from Sort Merge Join to Broadcast Hash Join when the dimensions of one of the tables fall below the broadcast threshold, automatically performing the coalesce of partitions after shuffle stages and handling skewness in joins. While these capabilities have the potential to improve Spark's performances, there is currently no in-depth analysis regarding the inner workings of AQE. As such, given this lack of information, it is not possible to make a comprehensive analysis of the impacts that AQE has on production workloads. This thesis explored the inner workings of AQE to understand its mechanisms and provide a comprehensive analysis of this technology.

[1] Spark SQL Guide. url:https://spark.apache.org/docs/3.1.2/sql-performance-tuning.html

2. Methodology

In order to better understand the inner workings of AQE two sets of experiments were used: the baseline set and the workload set. Experiments in the baseline set tested AQE's capabilities in a controlled environment, making use of suit-tailored data and workloads and being executed on a commodity laptop. In this set of experiments, AQE's capabilities were isolated and tested one at a time to better understand the underlying mechanisms. The dynamic join selection feature was tested both with and with-

out the usage of hints. AQE's ability to dynamically change the number of partitions after shuffle stages was tested both trying to increase and diminish the number of partitions. AQE's skewness handling capabilities were studied with varied settings relative to the definition of skewness. The workload set of experiments focused instead on observing how AQE's usage influences Spark's performances in real-world scenarios and interpreting its effects based on the findings in the baseline Section. Two different workloads were tested: workload A and workload B. Workload A executes delta merge operations while workload B executes a query characterized by highly filtering statements followed by joins scheduled as sort-merge joins.

3. Baseline tests

The baseline test set has returned a great deal of information both on AQE and on the catalyst. This group of tests was executed locally on a commodity laptop (16 GB RAM, i5-8350U CPU). The first component to be tested was tested with the following workload (except for the experiments on the Broadcast Hash join, which used a simplified workload):

```
val dfA =spark.read
  .parquet("spark-aqe-main/src/main/resources/bigTableA")
val dfB =spark.read
  .parquet("spark-aqe-main/src/main/resources/bigTableB")
val dfK =spark.read
  .parquet("spark-aqe-main/src/main/resources/keys")
val dfAux=dfB
  .join(dfK, dfB("LAVORO\_B")===dfK("LAVORO\_KEYS"),
        "left")
  .filter ("FILLER1==FILLER2 and FILLER2==FILLER3")
  .repartition(200)
val dfResult =dfA.join(dfAux, join_conditions, join_type)
dfResult.count()
```

While AQE was enabled, however, the automatic coalesces and the skew join handling were disabled. Broadcast threshold, join conditions and join type were set in each experiment to trigger different join strategies [2] . The first finding to be noted was obtained while testing on the sort merge join and removing the .repartition(200) command. The dynamic join selection does not behave as described in the documentation, having instead the need for a further shuffle stage between the filtering action and the subsequent sort merge join. This is corroborated both empirically by the tests' results and also in AQE's source code [3] . Among the other findings worth noting is that AQE is also able to dynamically change from a Cartesian Product to a Broadcast Nested Loop join, a possibility which is not present in the documentation footnote 1 and that may lead to otherwise avoidable out of memory errors footnote 2. Furthermore, it was found that the broadcast threshold directly influences the choice between Cartesian Product and Broadcast Nested Loop join in equi-joins, while available information states otherwise footnote 2. While testing AQE's interaction with hints it was found that broadcast

[2] Developers' comments on how Spark chooses the join strategy. URL: https://github.com/apache/spark/blob/branch-3.1/sql/core/src/main/scala/org/apache/spark/sql/execution/SparkStrategies.scala#L109

[3] How AQE optimizes future stages. URL: https://github.com/apache/spark/blob/branch-3.1/sql/core/src/main/scala/org/apache/spark/sql/execution/adaptive/AdaptiveSparkPlanExec.scala#L454

Table 1: Effects of broadcast hints on a Sort Merge Join

Hints	No Broadcast	Broadcast threshold greater than 0
No hints	Sort Merge Join	Depends on the threshold value: if at least one of the tables is below the broadcast threshold value then a Broadcast Hash Join is performed, otherwise a Sort Merge Join is performed
Outer tables	Sort Merge Join	Depends on the threshold value: if the outer table is below the broadcast threshold value then a Broadcast Hash Join is performed, otherwise a Sort Merge Join is performed
Inner tables	Broadcast Hash Join	Broadcast Hash Join
Both tables (left outer, right outer and inner joins)	Broadcast Hash Join	Broadcast Hash Join

hints usage does not follow the available information about it footnote 2, following instead the behavior outlined in table 1. It was also found that, while the usage of a merge hint can hinder AQE's optimization abilities, forcing the catalyst to choose a Sort Merge join in place of the Broadcast Hash join that would instead be chosen because of AQE's intervention.
AQE's ability to perform coalesces and repartitions was tested in an isolated fashion too: skew handling was disabled, while dynamic changes to the join strategy were avoided by proper tuning of the broadcast threshold. The following workload was used for tests on the coalesce:

```
val dfA=sparkNoAQE.read
  .parquet("spark-aqe-main/src/main/resources/bigTableA")
  .sample(false, 0.0001)
  .repartition(200)
val dfB =sparkNoAQE.read
  .parquet("spark-aqe-main/src/main/resources/bigTableB")
  .sample(false, 0.0001)
  .repartition(200)
val dfResult_ =dfA
  .join(dfB,dfA("LAVORO_A")===dfB("LAVORO_B"), "left")
print(dfResult_.explain())
dfResult_.count()
```

The following workload was, instead, used for the tests on the repartition:

```
val dfA=spark.read
    .parquet("spark-aqe-main/src/main/resources/bigTableA")
    .coalesce(1)
val dfB =spark.read
  .parquet("spark-aqe-main/src/main/resources/bigTableB")
  .coalesce(1)
val dfResult_ =dfA
  .join(dfB,dfA("LAVORO_A")===dfB("LAVORO_B"), "left")
print(dfResult_.explain())
dfResult_.count()
```

AQE's automatic post shuffle coalesce proved to behave as described, being able to coalesce partitions but not to repartition them into a higher number. The only caveat is an apparent dismissal of the minimum number of partitions set.
AQE's ability to handle skewed joins was tested on the following workload:

```
val dfA =spark.read
  .parquet("spark-aqe-main/src/main/resources
  /veryBigTableSkew")
val dfB =spark.read
  .parquet("spark-aqe-main/src/main/resources/veryBigTable")
val dfResult_ =dfA
  .join(dfB, join_expression, join_type)
```

The tables used to create `dfA` were heavily skewed in the column used as the join key. Join expression, join type and the broadcast threshold were tuned for each experiment to trigger the desired join strategy. AQE and its skew handling module were enabled while the automatic post shuffle coalesce was disabled via the configurations and the dynamic join strategy selection was avoided via proper tuning of the broadcast threshold. The main caveat regarding skewness handling lies in the definition of skewness. AQE recognizes skewness according to two settings: `skewedPartitionThresholdInBytes`, acting as an absolute threshold, and `skewedPartitionFactor`, acting as a threshold relative to the median of the sizes of the partition. If a partition is greater than the `skewedPartionFactor` multiplied by the median of the partitions' dimension and the `skewedPartitionThresholdInBytes` then it is considered skewed and it will be divided into partitions of the size set in `advisoryPartitionSizeInBytes`. Tuning these configurations to correctly define when a partition is to be considered skewed has proven to be quite complex, yet vital for the correct usage of this feature of AQE's. Another relevant finding is that the join type will determine the side that will be checked for skewness. As an example, if the side is set as `left`, only the left table of the join will be checked for skewness. Another caveat is that skewness in the join column directly translates into skewness in the join tasks only when a sort merge join is used.
Overall, these experiments have shown how AQE strays from the behavior described in the documentation while also highlighting details over how to use AQE to the full extent of its capabilities.

4. Workload tests

In this Section experiments focused on analyzing the impact that using AQE can have over production workload by analyzing how Spark's performances changed when executing workloads A and B with AQE enabled.

Workload A is responsible for updating delta tables. The experiments on this workload were divided in two phases, each of the duration of one week. Workload A is executed over a cluster with 4 workers, each having 4 CPU cores and 20GiB of RAM. During the first week the following configurations were used [4]:

- `adaptive.enabled`: True , in order to enable AQE;

[4] in order to obtain the full name of the configuration prepend `spark.sql.`

	AQE enabled	AQE enabled, week 1	AQE enabled, week 2	AQE disabled, week 1	AQE disabled, week 2	AQE disabled
count	825.000000	356.000000	469.000000	444.000000	252.000000	696.000000
mean	45.665838	60.621676	34.313433	39.813363	98.361905	61.011973
std	122.448013	175.515619	52.267998	68.476828	302.731585	192.041586
min	0.916667	0.916667	3.000000	0.733333	2.900000	0.733333
25%	11.000000	12.000000	9.000000	14.000000	9.900000	13.000000
50%	18.000000	15.000000	18.000000	20.000000	18.000000	20.000000
75%	30.000000	29.000000	30.000000	30.750000	28.000000	29.000000
max	2238.000000	2238.000000	330.000000	540.000000	2304.000000	2304.000000

Figure 1: Table describing the data regarding both the two experiments and older executions of Workload A (execution times in minutes)

Metric	Percentage variation at week one	Percentage variation at week two
Files added	-79.28%	-78.57%
Rewrite time	-6.08%	-2.06%
Scan time	7.75%	4.12%

Table 2: Percentage variations from delta logs between the two weeks without AQE and the two experiments performed

- `adaptive.coalescePartitions.enabled`: True in order to enable the feature studied in this phase;
- `adaptive.skewJoin.enabled`: False;
- `autoBroadcastJoinThreshold`: -1, as such both for reasons related to the workload and to disable the dynamic join selection;
- `adaptive.coalescePartitions.min PartitionNum`: used to set the minimum number of partitions after the automatic coalesce has been performed. The value is set to default;
- `adaptive.advisoryPartitionSize InBytes`: given the dimensions of both the tables and the updates and taking into account the trade-off explained before, this value is set to 256 MiB; and
- `adaptive.coalescePartitions.initial PartitionNum`: left to the default value of `none`, as it was considered to be irrelevant to this experiment.

As the configurations show the only AQE element active was the automatic post shuffle coalesce. During the first week, it was possible to observe that Spark's execution times have not been affected by the introduction of AQE, as shown in Image 1. At the same time, the number of partitions involved was reduced drastically, as shown in Table 2. This, however, did not affect delta update times. The second week of experimentation saw the tuning of the minimum number of partitions to 16 to always fully make use of the available degree of parallelism. The second week of experimentation also saw the introduction of a broadcast threshold of 20 MiB, thus enabling both statically and dynamically scheduled Broadcast Hash joins. As shown in figure 1 the average execution time was drastically reduced. This is due to the statical scheduling of broadcast hash joins in place of sort merge joins. Dynamic join strategy selection was not triggered in these experiments: the workload did not involve filtering statements followed by sort merge joins. While Spark's performances improved, delta merge execution times stayed stationary as shown in Table 2. Experiments over workload B tested, at first, dynamic join selection, then automatic post shuffle coalesce. The first sperimental phase lasted eight days and saw the usage of the following configurations 4:

- `adaptive.enabled`: True , in order to enable AQE;
- `adaptive.coalescePartitions .enabled`: False;
- `adaptive.skewJoin.enabled`: False; and
- `autoBroadcastJoinThreshold`: 20 MiB,

	AQE enabled, week 2	AQE enabled, week 1	AQE disabled, week 1	AQE disabled, week 2	AQE enabled	AQE disabled
count	18.000000	15.000000	17.000000	12.000000	27.00000	29.000000
mean	50.427778	56.160000	63.165020	78.050000	51.97037	69.324322
std	53.387560	57.147138	58.433245	73.643768	53.54341	64.322606
min	7.900000	8.800000	0.005333	9.800000	7.90000	0.005333
25%	17.000000	13.500000	16.000000	14.750000	15.00000	16.000000
50%	21.000000	24.000000	23.000000	33.500000	21.00000	31.000000
75%	79.250000	126.000000	120.000000	157.500000	105.00000	144.000000
max	150.000000	144.000000	156.000000	174.000000	150.00000	174.000000

Figure 2: Table describing the data regarding both the two experiments and older executions of Workload B (execution times in minutes)

the value is set at this value as, after analyzing multiple executions of this workload, it enables the Catalyst to schedule Broadcast Hash joins without incurring in the risk of crashing the driver.

These experiments confirmed what was already seen while testing AQE's dynamic join strategy selection capabilities. To trigger the dynamic join selection a shuffle phase between the filtering action and the subsequent sort merge join is needed. Since the DAGs involved in the execution of Workload B do not present the aforementioned shuffle phase between the filter and the join, AQE did not trigger. As such, the improvement that can be seen in execution times in figure 2 can be considered a consequence of the static scheduling of broadcast hash joins in place of sort merge joins. It has to be noted however that, due to the reduced sample size and the high variance in samples, the comparison between the averages of execution times is of little statistical significance.

The second experiments group also lasted eight days and verted on testing the automatic post-shuffle coalesce functionalities 4:

- `adaptive.enabled`: True , in order to enable AQE;
- `adaptive.coalescePartitions .enabled`: True in order to enable the feature studied in this phase;
- `adaptive.skewJoin.enabled`: False;
- `autoBroadcastJoinThreshold`: -1, as such both for reasons related to the workload and to disable the dynamic join selection;
- `adaptive.coalescePartitions.min PartitionNum`: used to set the minimum number of partitions after the automatic coalesce has been performed. The value is set to 8, equal to the number of CPU cores available to the executors, in order to fully make use of the available parallelism;
- `adaptive.advisoryPartitionSize InBytes`: given the dimensions of both the tables and the updates and taking into account the trade-off explained before, this value is set to 256 MiB; and
- `adaptive.coalescePartitions.initial PartitionNum`: left to the default value of `none`, as it was considered to be irrelevant to this experiment.

The results are in contrast with what was observed during Workload A tests: while during the baseline tests and tests over workload A the minimum number of partitions was ignored by Spark, here it was taken into account. The minimum number of partitions was above eight even in situations in which 8 partitions of 256 MiB would have been sufficient.

As Image 2 shows, execution times were drastically reduced on average. However, as already stated for the previous experiment, the comparison between the averages is of little statistical meaning, given the low number of samples and the high variance among the samples.

5. Conclusions and future work

The experiments performed returned a great amount of insight regarding AQE's inner workings. It was found that AQE does not perform dynamic join selection as described in the docu-

mentation. While the documentation states that AQE will switch join strategy at shuffle phases when one of the joined tables falls below the broadcast threshold, AQE does not behave as described. As both baseline experiments and experiments over Workload B have shown, a shuffle phase is needed between the filtering action and the following sort-merge join to trigger the dynamic join strategy selection. It was also found that AQE can transform cartesian products into Broadcast nested loop joins. Furthermore, baseline experiments have shown that the usage of broadcast hints over sort merge joins can have effects not described in the documentation, as shown in Table 1.

While the baseline experiments and experiments over workload A have highlighted that the Automatic post shuffle coalesce does not seem to respect the parameter that sets the minimum number of partitions, experiments over workload B have proven otherwise. This discrepancy will require further study, as the reasons behind it are not clear.

Experiments over the skew join handling have shown that it behaves as described in the documentation. These experiments have also shown that properly tuning AQE to recognize skewness is as complex as relevant. it was also found that the join type influences what tables are considered when searching for skewness.

Experiments over workloads A and B have provided further proof that the dynamic join selection needs a shuffle phase between the filtering action and the subsequent join. They have also shown the previously mentioned discrepancy between local and cluster execution regarding the automatic post shuffle coalesce. Both workload A and Workload B have seen better performances when using AQE (Images 1 and 2), while workload A has shown that a reduction of the partitions involved in delta merge operations does not translate into better performances. This points to the data transmission over the internet as the bottleneck for these operations.

While these studies have provided a clear picture of some of AQE's capabilities, there are still more to do. As already mentioned, the behavior of the automatic post shuffle coalesce is still in need of further clarifications. Another thing to note is that AQE's skewness handling capabilities have not yet been tested on production workloads. Furthermore, given how parameter tuning affects AQE's performances, more tests with finer tuning on configurations are needed to fully utilize AQE. Another field that needs research is AQE's interaction with Hyperspace's indexing system. Hyperspace modifies the Catalyst to make it operate on an index-aware basis. Given that the usage of indexes can lead to skipping shuffle stages, the possible conflicts and trade-offs between Hyperspace's indexes and AQE need to be studied.

6. References

[1] Michael Armbrust et al. "Spark sql: Relational data processing in spark". In: *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 2015, pp. 1383–1394.

[2] Matei Zaharia et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing". In: *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. 2012, pp. 15–28.

[3] Matei Zaharia et al. "Spark: Cluster computing with working sets." In: *HotCloud* 10.10-10 (2010), p. 95.