POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Dynamic Query Optimization in Spark

Tesi di Laurea Magistrale in
Computer Science and Engineering -
Ingegneria Informatica

Author: **Antonio Pipita**

Student ID: 928429
Advisor: Prof. Emanuele Della Valle
Co-advisors: Andrea Picasso Ratto
Academic Year: 2021-22

Abstract in english language

Considering the relevant role of data in the last decades and taking into account the increase of both its volume and velocity, query optimization on distributed massive parallel processing systems is now a hot topic. Among distributed query engines Spark is one of the most relevant ones: Spark was originally introduced as an abstraction layer over MapReduce. After the introduction of the SparkSQL module, it however became one of the most used distributed SQL query engines. In Spark SQL the task of optimizing queries is assigned to the Catalyst, which applies a two-step optimization procedure. At first, the tree representing the query is optimized in a rule-based fashion. Afterward, the optimized logical plan goes through a cost-based optimization phase, responsible mainly for selecting the join strategies. The Catalyst's optimizations, however, are purely static; as such any change to the data that might occur during the execution of a query cannot be taken into account. To bridge this gap AQE, Adaptive Query Execution, was introduced. AQE presents three main dynamic optimization procedures: dynamic join strategy selection, automatic post shuffle coalesce and skewed join handling. Despite the possibilities in the optimization field introduced by these functionalities, there is a distinct lack of information regarding AQE's inner workings. This entails a lack of insight regarding the actions that AQE undertakes when optimizing workloads. The goal of this thesis was to explore the inner workings of AQE. This was achieved via two sets of tests: the baseline set and the workload set. The baseline set saw the usage of suit-tailored data and workloads to study AQE's capabilities in an isolated fashion to fully understand how they operate. Once the first set of experiments had provided a solid understanding baseline over AQE the workload set of experiments explored how AQE interacts with production workloads. During these experiments various AQE capabilities, in different combinations and with different settings, have been studied. Both AQE's effects on Spark's efficiency and how it interacted and modified the standard query execution plan have been taken into account. The results found during the baseline tests, united with the findings in the tests over the two production workloads, have provided many insights and knowledge about AQE, thus enabling the user to better understand AQE and make better use of it.

Keywords: Spark, SparkSQL, query, optimization, distributed system, AQE, shuffle, join strategy, partitions, skewness

Abstract in lingua italiana

Considerando il ruolo sempre più rilevante che i dati hanno assunto negli ultimi decenni e considerando l'aumento in volume e velocità nella loro produzione, la questione dell'ottimizzazione delle query su sistemi distribuiti ha assunto un ruolo di grande rilevanza. Trai sistemi per query distribuiti più diffusi spicca Spark: originariamente creato come layer d'astrazione su MapReduce, con l'introduzione del modulo SparkSQL è diventato uno degli strumenti più diffusi per le query SQL distribuite. In Spark SQL l'ottimizzazione delle query è compito del Catalyst, il quale applica due layer di ottimizzazione: in primo luogo esegue un'ottimizzazione rule-based, mentre in seconda battuta esegue un'ottimizzazione cost-based atta principalmente a stablilire la strategia meno costosa per eseguire i join. Le ottimizzazioni attuate dal Catalyst sono, però, statiche e non tengono conto delle eventuali variazioni sul dato che possono avvenire durante l'esecuzione della query. Onde ovviare a questa mancanza è stato introdotto AQE, Adaptive Query Execution, il quale introduce capacità d'ottimizzazione dinamica per il Catalyst. In Spark 3.1.2 le capacità di AQE sono 3: cambiare dinamicamente la strategia scelta per i join, performare automaticamente la coalesce delle partizioni dopo una fase di shuffle e gestire join che presentino skewness. Nonostante le possibilità di ottimizzazione derivanti da queste funzionalità, le informazioni riguardo al funzionamento di AQE sono scarse, rendendo quindi difficile analizzare perchè AQE agisca su di un workload in una determinata maniera. Questa tesi si è posta lo scopo di esplorare il funzionamento interno di AQE: ciò è stato fatto tramite due gruppi di test. Il primo gruppo è andato ad isolare le capacità di AQE, testandole singolarmente su dei workload e del dato costruiti appositamente per ingaggiare la feature d'interesse. Una volta compreso il funzionamento interno di AQE il focus si è spostato sul testare l'interazione con dei workload correntemente utilizzati in produzione. Durante questi test varie capacità di AQE sono state testate, in più combinazioni e con diverse configurazioni dei parametri. I risultati ottenuti dai test sui workload, uniti ai risultati ottenuti nella prima fase di test, hanno restituito un quadro più chiaro del funzionamento di AQE, permettendo di valutarne l'impatto in maniera approfondita e, quindi, di poterlo utilizzare più efficacemente.

Parole chiave: Spark, SparkSQL, query, ottimizzazione, sistema distribuito, AQE, shuffle, strategia di join, partizioni, skewness

Contents

Abstract in english language iii

Abstract in lingua italiana v

Contents vii

Introduction 1

1 State Of The Art 3
- 1.1 Optimization Approaches . . . 4
- 1.2 Optimization in ER Databases . . . 5
- 1.3 Distributed Approaches . . . 7
- 1.4 Mapreduce . . . 11
- 1.5 Spark . . . 14

2 Problem Definition and Methodology Presentation 25

3 Baseline Tests 33
- 3.1 Dynamic join Selection . . . 33
  - 3.1.1 Without hints . . . 34
  - 3.1.2 With hints . . . 47
- 3.2 Automatic Post Shuffle Coalesce of partitions . . . 59
  - 3.2.1 Coalesce . . . 60
  - 3.2.2 Repartition . . . 67
- 3.3 Skewed Join Handling . . . 69
  - 3.3.1 Sort Merge Join . . . 70
  - 3.3.2 Broadcast Hash Join . . . 77
  - 3.3.3 Cartesian Product . . . 78
  - 3.3.4 Broadcast Nested Loop Join . . . 79
  - 3.3.5 Conclusions . . . 80

4 Workload Tests 81
- 4.1 Workload A . . . 81
  - 4.1.1 Experiment 1 . . . 81
  - 4.1.2 Experiment 2 . . . 83

4.1.3 Data From Delta Logs 84
4.2 Workload B 85
4.2.1 Experiment 1 85
4.2.2 Experiment 2 88

5 Conclusions and Future Work 91

6 Bibliography 95

7 Appendix A - Table statistics 97
7.1 Table A 97
7.2 Table B 98
7.3 Table K 98
7.4 Table Big 99
7.5 Table BigSkewed 99
7.6 Table SkewReduced 99
7.7 Table NotSkewReduced 100

8 Appendix B - Experimental results 109
8.1 Dynamic join strategy selection 109
8.1.1 Broadcast Hash Join without hints execution result 109
8.1.2 Broadcast Nested Loop Join without hints execution results 110
8.1.3 Broadcast Hash Join with hints control experiment execution results 112
8.1.4 Broadcast Hash Join, Broadcast hint with AQE off experiment execution results 113
8.1.5 Sort Merge Join, Merge hint with AQE on experiment execution results 113
8.1.6 Cartesian Product, Broadcast Hint and Broadcast Threshold without AQE 116
8.2 Automatic Coalesce of partitions Results 117
8.2.1 Broadcast Hash Join automatic post shuffle repartition test results . 117
8.2.2 Sort Merge Join automatic post shuffle repartition test results . . . 118
8.3 Skew Join Handling Results 120
8.4 Data From Delta Logs 133

Acknowledgements 135

Introduction

How to store and manage data has always been an extremely relevant topic and it has become even more so in the last couple of decades. Where before there were only centralized databases aimed at keeping track of the single transaction, nowadays for most businesses to correctly and efficiently store, manage, query and analyze data has assumed a relevant position not only at an operational level but at a strategic one too. Given the rise in the volume of data, distributed architectures have been introduced to manage it and the whole paradigm for data analysis has shifted towards new approaches. Spearheading this paradigm shift, the databases and their query engines have changed too: from centralized entities based on the entity-relationship model to new architectures with distributed query engines. Throughout all of these changes, the question of query optimization kept a central role: both cost-based and rule-based optimization have been adapted to be used in the new distributed environment, making use of statistics and indexes, both statically and dynamically. Among the most important databases and distributed data processing frameworks there are OracleDB, Presto, Dremel (and Big Query) and all the frameworks stemmed from MapReduce, Spark in particular. Other than being a distributed data processing framework, Spark has also integrated SQL functionalities via the module SparkSQL. SparkSQL also introduced the Catalyst in the Spark environment, a query optimizer capable of both rule and cost-based static optimization. Recently, however, with Spark 3.0 the Catalyst was expanded with the addition of dynamic optimization capabilities with the introduction of Adaptive Query Execution. AQE enables the Catalyst to perform optimizations based on how data behaves and changes during the execution of the queries, thus accounting for information that was not available during the static optimization of the query. In particular, according to the documentation, AQE is engaged during shuffle phases and can switch dynamically from Sort Merge joins to Broadcast Hash joins when the dimension of one of the joined tables falls below the broadcast threshold. AQE can also automatically perform coalesce over dataframe partitions after shuffle phases and can improve performances in skewed joins. While these new capabilities might introduce the possibility for relevant performance improvement, there is currently little information on AQE's inner workings, as the information provided by AQE's authors does not explore its inner workings in depth. There is also no analysis of how AQE impacts a production workload that explores why Spark's performances are impacted in that way. The experiments presented in this thesis had the objective to provide insights both on AQE's inner workings, how it affects the execution of workloads and why does it affect them as it does.

To have a strong understanding of how AQE operates the baseline experiments that have been performed had the object to provide a comprehensive description of how AQE

behaves. The experiments tested AQE's functionalities independently: the feature of interest was isolated, then a workload suit tailored for it was executed. The first group tested AQE's capabilities to dynamically change join strategies at runtime, both with and without the usage of hints. The experiments in this Section provided a great amount of insight regarding AQE's abilities, highlighting side-effects not present in the documentation and also providing greater insight on how the Catalyst selects the join strategy. The following group of experiments explored how the Automatic post-shuffle coalesce works, providing a greater understanding of the capabilities described in the documentation. The last group of baseline tests tested AQE's capabilities in handling skewed joins and it has provided a great amount of information regarding how AQE identifies skewness in joins. Once the baseline tests had established how AQE works, the workload tests aimed at exploring how AQE affects real-world ETL workloads, studying how it impacts performances, and using the insights generated in the previous tests to explain why it produces the examined effects. Two different workloads were tested, interacting with different AQE modules. The results provided insights into how to use AQE in ETL pipelines. Overall, these two groups of experiments were able to thoroughly explore both the Catalyst's and AQE's inner workings, providing useful insights applicable in Spark's usage, both with and without AQE. Among the most relevant findings, it is possible to find AQE's side effect to dynamically transform Cartesian Products into Broadcast Nested Loop joins, the effects that hints usage has over AQE, the inner workings of skewness recognition, and how AQE's usage affects the two production workloads considered.

Chapter 1 provides a historical perspective over distributed query engines, Spark's development and alternatives to Spark's usage, as well as introducing AQE and its capabilities. Chapter 2 presents the problems about AQE and introduces the rationales behind the experiments conducted to solve the problems exposed. Chapter 3 contains the experiments that tested AQE's capabilities, while chapter 4 presents the tests performed on production workloads using AQE. Chapter 5 sums up the insights and findings gained during the previous experiments and highlights possible tracks for future work on AQE. Section 7 reports the statistics of the tables used during the Baseline Tests, while Section 8 provides further experimental results obtained during the testing phases.

1 | State Of The Art

During the last decades, the relevance of data in every business and management aspect grew more and more: from being relevant only at the operational level, recording transactions and maintaining inventory information, to being used in advanced analytics applications aimed at computing key performance indexes for the upper-echelon management to use as a guide for strategic decisions. As data grew in relevance it also grew in volume, leading to new engineering problems regarding its storage: for years the most widely used databases were based on the relational model, following the ACID paradigm and being stored in a centralized fashion. The ACID acronym represents the 4 most relevant properties characteristic of the traditional approach:

1. Atomicity: an operation has either committed or has not affected the database at all, there are no states in between these two

2. Consistency: the data must not contradict itself and must be consistent across all the database

3. Isolation: the result of a transaction, may it be a commit or a rollback, must not affect the result of any other transaction

4. Durability: the effects of a commit must persist on the data up until when another commit changes them, staying constant otherwise

Relational Database Managers are based on the Entity-Relationship (ER) model: the databases are composed of tables, which are divided into various fields (columns) and the entries are represented as rows across those fields. One of the columns is described as a key, a unique identifier that must be present and different for each row. The relationships between tables are expressed as further tables that contain the keys of both the elements, thus linking the tables that share that relationship. Because of this architecture, the most common operations are joins, since searching for data across multiple tables requires navigating the tables via their keys and the external references to said keys. This is one of the greatest limitations of centralized and normalized Databases since joins can be extremely expensive operations.

Up until fairly recently, databases were deployed in a centralized fashion, meaning that both the physical and logical distribution of data was organized in a centralized way.

This was so because the characteristics of data (their structure and their volume) did not impose any form of restriction on centralizing its storage, while the technology did not make distributed solutions viable. When the requirements surpassed the resources available for a database, or any kind of computing system, the approach was to scale up in a vertical fashion, which is to substitute the machines with more powerful ones. This approach however became more and more a liability as data incremented in volume, velocity, variety and veracity. As the nature of data changed, so did the paradigm: from the ACID paradigm to the BASE one:

1. Basic Availability: data must always be available, no matter the situation

2. Soft state: the ACID properties can be disregarded

3. Eventual consistency: while data is not required to be consistent, it is required to be in a state from which it can be brought in a state of consistency in a short amount of time

As the paradigm shifted so did the architectures: from being centralized to distributed. With this shift, combined with the BASE approach, new problems regarding consistency, availability and partition tolerance:

1. Consistency: data consistent across all the nodes

2. Availability: data is always available

3. Partition tolerance: the data store bears well being divided into partitions

Among these three characteristics, it is possible to guarantee only two at a time (CAP theorem).

1.1. Optimization Approaches

Throughout the changes in paradigm, the problem of optimizing the workloads remained relevant. Queries over data are translated into execution plans, a set of sequential or parallel steps to be done to perform the query: for a given query there may be multiple execution plans, that in turn entail different resources usage and response times. To have an efficient database, independently from whether centralized or distributed, or whether based on the ACID or the BASE paradigm, the queries have to be executed in the most efficient ways possible. While the database user can improve the database's performances by formulating the queries optimally, there are also other methodologies to minimize the queries' costs and improve efficiency over query execution. In the next paragraph the most widely used optimization techniques will be introduced: indexing, statistics, rule and cost-based optimization and dynamic optimization. Afterward, the focus will instead move in their implementation in both centralized and distributed databases and data processing frameworks. One of the most used optimization methods is indexing: indexes

are one of the most widespread query optimization techniques. By index we refer to any kind of data structure that can speed up the execution of a query. Indexes can be defined as either primary or secondary. A primary, or clustered, index will refer to the primary key of the indexed table, as such it does not have duplicates and requires the rows in the data blocks to be ordered by the primary key. Secondary indexes, instead, do not refer to the primary key of the indexed table. It can thus have duplicates and there may be multiple secondary indexes for one table. Both primary and secondary indexes can be built via various data structures; the B+ tree is the most common to be found [15] [24]. Another optimization strategy is the usage of statistics. Both the number of statistics and their usage are strictly tied to the database optimizer. The query execution plan can be optimized mainly in two ways: rule-based and cost-based optimization. The optimizers that adopt a cost-based strategy for optimization simulate various access and query execution strategy and, based on the collected statistics and available indexes, produce an estimation of the cost associated with a particular strategy, thus choosing the strategy that has the lowest estimated cost. Rule-based optimization is instead based on optimizing query execution by manipulating the queries via a set of rules, with a cost associated with every transformation applied via those rules. While both rule-based and cost-based optimization strategies have proven to be effective in optimizing queries [15] [8] [1] [18], being able to dynamically adapt execution strategies to the new contexts and situations that can arise during query execution, thus being able to use improvements over the query plan unavailable in the initial conditions.

1.2. Optimization in ER Databases

Regarding the implementation of the aforementioned optimization techniques, an interesting example is the DB2 advisor: while usually B+ trees (balanced and ordered trees where the internal nodes have two or more sons and leaf nodes are connected between them, making them ideal for bulk information retrieval) are extremely useful in optimizing data access, there are instances in which other indexes can outperform them [24]. To make the process of identifying such situations easier DB2 can automatically detect the best index to use [24]. To do so the DB2 advisor employs the "Smart Column Enumeration For Index Scans", which considers 5 sets of columns and analyzes the query to determine the candidates for the best index. These candidates are then compared to the ones obtained via a brute force algorithm, used as a baseline. Once the best candidates are found the optimizer will select the best ones via cost-based optimization. The costs are computed on the statistics on tables and b+ tree indexes. If statistics are not possible to obtain easily they are then estimated with a pessimistic approach. The indexes obtained for query optimization are used as a starting point to optimize the whole workload. Benchmarks show that this approach has a good impact on performances [24].

Another relevant example in the query optimization field for relational databases is OracleDB and its query optimizer. OracleDB is a relational database based on vertical scalability and designed for OLTP purposes. Its optimizer tries to generate the optimal execution plan for a SQL statement by choosing the plan with the lowest projected cost among all of the candidate plans. This is done by taking into account statistics, I/O,

CPU and communication factors. Given the number of statistics collected, the optimizer is usually capable of optimizing and optimally executing queries. Given the nature of SQL, particularly not being a procedural language, the optimizer can reorganize the steps of the queries as it sees fit. The optimal access strategy is determined via cost-based optimization: the costs of the various queries are determined by simulating the costs of the various access strategies and index usages and assigning them to the various query steps, to then choose the least costly one. It is worth noting that cost-based optimization is a fairly recent addition to the OracleDB optimizer; the legacy optimizer is instead rule-based [15]. The recommended method is represented in the form of an execution plan, which shows the recommended actions to undertake for each execution step to execute the plan optimally. Each step can consist of row retrieval or row preparation for the next statement. The execution plan also contains the estimated costs, both for the single steps and the overall execution. Each SQL "SELECT" statement is represented by a query block, query blocks can be a top-level statement, a subquery or an unmerged view; for each query block, a query subplan is generated, which are then optimized in a bottom-up fashion.

The Optimizer itself is composed of three parts: the query transformer, the estimator and the plan optimizer. The query transformer determines whether it would be convenient to rewrite the original query into a semantically equivalent new one which entails lower costs. The estimator instead determines the costs of a given execution plan based on three parameters: selectivity (fraction of rows that a predicate, or a group of predicates, refers to), cardinality (number of rows returned by the operations in an execution plan) and cost. Last but not least, the plan generator explores the possible plans for a query block assigning a cost to each one and afterward picking the one with the lower cost. A further development in OracleDB optimization are Adaptive Query Plans (AQP): an AQP enables the optimizer to make decisions regarding query execution at runtime. Adaptive plans are composed of multiple predetermined subplans and a statistics collector: based on the statistics collected at runtime the dynamic plan coordinator chooses amongst the subplans the most convenient one. Dynamic plans are composed of a set of subplan groups. Each subplan group is composed, in turn, by a set of subplans, which represent the alternatives from which the optimizer can choose at runtime. As an example, the optimizer could switch from a subplan containing a Sort Merge join strategy to a Hash Broadcast one at runtime. The optimizer statistics collector is instead a row source: it is inserted at key points of the execution plan to collect the updated statistics related to cardinality and histograms to be able to make decisions based on updated information. Given how cardinality misestimation can lead to suboptimal decisions, the ability to update plans at runtime can lead to great performance improvements. On an operational note, a statement execution begins with the execution of the initial plan. During execution, the statistics collector gathers information about the execution (if the process is distributed, each slave process collects the statistics, then the aggregator collects them and sends them to the clients). Based on the collected statistics the dynamic plan coordinator decides which plan to execute. It has to be noted that adaptive query plans are not feasible for all kinds of plan changes. For optimizations such as changing the join order, the optimizer uses the information gathered during the execution to decide how to improve the next execution.

1.3. Distributed Approaches

As data storage shifted from centralized to distributed other approaches started to emerge. One of such approaches can be found in Dremel, a scalable ad-hoc distributed query system developed by Google. Instead of being based on a MapReduce-like logic, Dremel was built on the ideas of web search and parallel DBMS, while having its architecture based on the serving trees used in distributed search engines [13]. Like in a web search, the query is pushed down in the tree and gets rewritten at each step, then the results are built aggregating the results of each step. Dremel also makes use of a proprietary SQL-like query language and, in contrast with other similar solutions, executes the query natively, without transforming them into MapReduce Jobs. Another relevant characteristic of Dremel is that it makes use of column-striped storage representation, enabling it to read fewer data from secondary storage and reduce CPU costs because of the reduced compression costs.

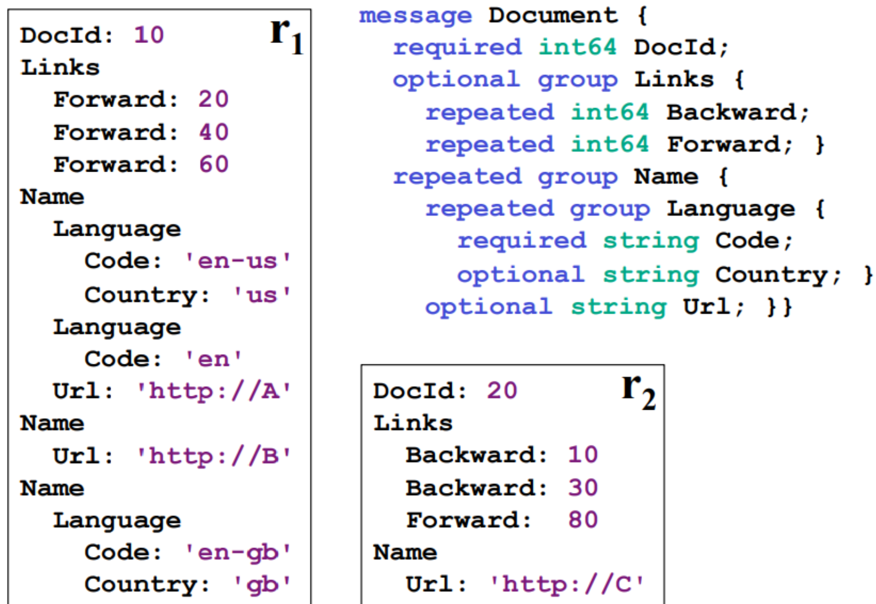

Figure 1.1: Dremel's data model [13]

Dremel's data model is based on strongly typed nested records following the syntax $\tau = dom|\langle A1 : \tau[|?], ..., An : \tau[|?]\rangle$, an example of which is reported in Figure 1.1. To improve retrieval efficiency, Dremel aims to store all values of a given field consecutively. To do so, data is stored in a columnar format. In order to adopt a columnar format without loss of data repetition and definition levels are introduced. Repetition levels represent at what repeated field in the field's path the value has repeated, while definition levels represent, for each value of a field with a certain path, how many fields along the path that could be undefined are undefined. Each column is stored as a set of blocks, each block containing the repetition and definition levels and the compressed values of the fields. The records are split into columns following the column-striping algorithm, which recurses into the record's structure and computes the repetition and definition levels. The opposite process of assembling the records is done via a finite state machine that reads

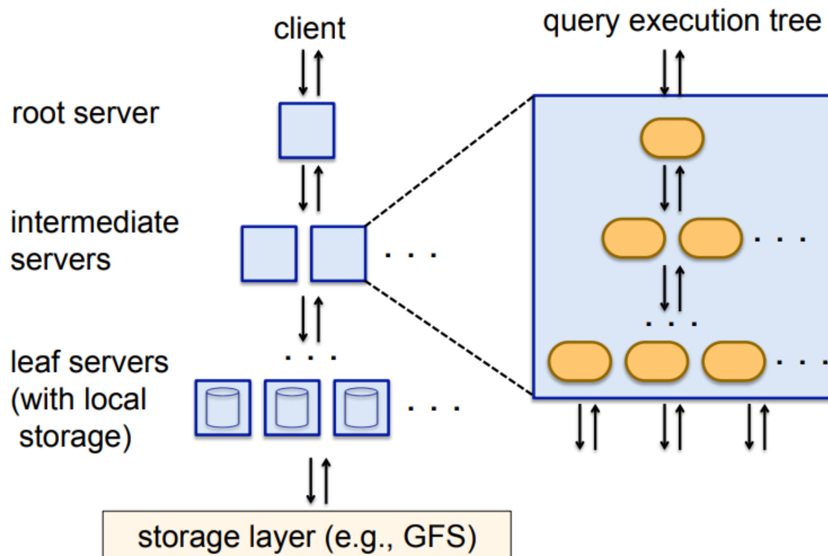

Figure 1.2: Dremel's architecture [13]

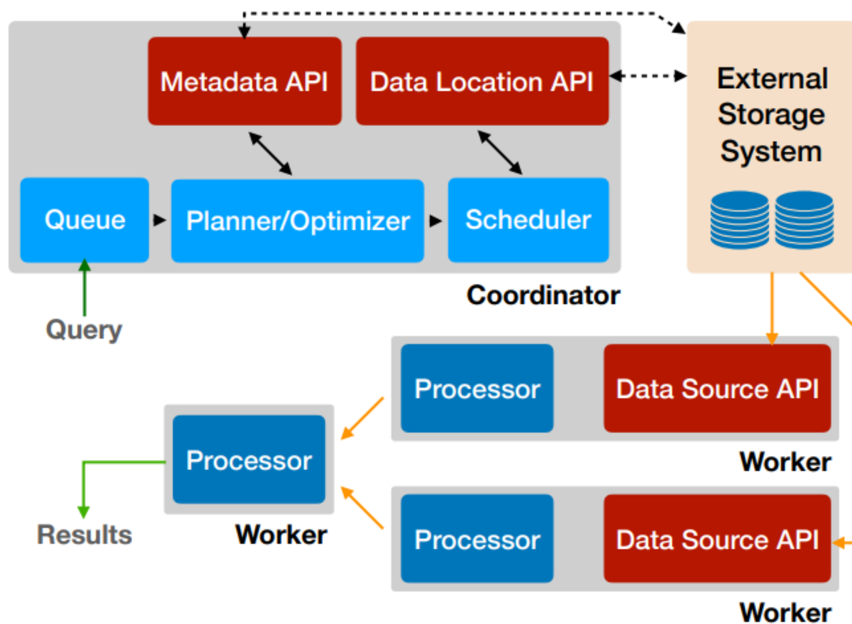

Figure 1.3: Presto's architecture [18]

repetition and definition levels for each selected field and appends the values to the output records. The queries are expressed in a SQL-like language and are served following the paradigm of a multi-level serving tree, as shown in Figure 1.2: the root server receives the queries and, based on the tables' metadata, routes the queries to the next level in the tree, while the leaves servers communicate with the storage layer. At each serving layer,

the source table is divided into a group of disjointed subsets until the query reaches the leaf nodes, then results are aggregated as they travel up the tree towards the root. Given that Dremel is a multi-user system, multiple users are expected to dispatch their queries simultaneously. The query dispatcher schedules query execution based on the queries' priority and acts as a load balancer.

Based on experimental results, it is possible to conclude that not only does Dremel achieve good performances, but also that MapReduce jobs can benefit from columnar storage and that MapReduce and query processing can be used in a complementary fashion: while MapReduce was designed for batch processes, Dremel was designed as an interactive data analysis tool for large datasets [2]. Dremel has been used outside of Google Company in the form of BigQuery [2] and its record shredding and assembling algorithms were used to develop Apache Parquet, a columnar file format [22].

Among the alternatives to Spark SQL, Presto assumes a role of relevance: Presto is an open-source distributed query engine used at Facebook, Netflix and Bloomberg among others [18]. Its use cases comprehend interactive analytics, Batch ETL processes, A/B testing and Developer/Advertiser analytics. Architecturally a Presto cluster consists of one coordinator node and one or more worker nodes, as shown in Figure 1.3: the coordinator admits, parses, plans and optimizes query execution, while worker nodes are responsible for query processing. The coordinator distributes the execution plan to the workers, starts the execution of the tasks and enumerates the splits, which are the handles to an addressable chunk of data, assigning them to worker nodes. The worker nodes running the tasks process the splits. Execution is pipelined and data flow between tasks as soon as it is ready. In some cases, Presto can return partial results while the remainder of the workload is processed. Presto is easily extensible, thus allowing the presence of connectors, which enable Presto to communicate with external data sources. The queries are expressed in ANSI SQL and submitted to the coordinator via a RESTful HTTP interface (or a JDBC client) and are then parsed and converted into a syntax tree. The logical planner then uses the syntax tree to generate an intermediate representation of the query plan as a tree of plan nodes, where each node represents a physical or logical operation. The logical plan is then transformed into a physical execution strategy by the optimizer by a greedy evaluation of a set of rules and is further optimized with a cost-based strategy involving join strategy selection and join reordering.

Presto's optimizer takes into account multiple factors while optimizing queries:

1. Data layouts: the connector Data Layout API, when enabled, reports the location and other data properties to the optimizer, which can then use them to optimize query execution.

2. Predicate pushdown: the optimizer can work with the connectors to decide to push-down range and equality predicates

3. Inter-node parallelism: the optimizer identifies the stages to be executed in parallel and connects them via shuffle phases

4. Intra-node parallelism: the optimizer identifies which sections of the stages can be parallelized inside one node.

Once the plan has been optimized, the coordinator distributes the plan stages to the workers as tasks and links the tasks together from one stage to the next one. Tasks can contain multiple pipelines between them, where pipelines are chains of operators. Scheduling strategies are divided into two groups, stage scheduling and task scheduling. During stage scheduling the coordinator chooses between the all-at-once and the phased stages scheduling policies. Once the scheduler has chosen one of the two policies it assigns tasks of that stage to the worker nodes. During task scheduling the task scheduler classifies stages as either leaf stages, responsible for reading data from the connectors or intermediate stages, responsible for processing intermediate results. When a task for a leaf stage begins execution the worker node upon which it is running signals itself as available for receiving a split. The coordinator is responsible for assigning splits to the workers: the connectors are asked to enumerate small batches of splits which are then assigned lazily to the workers. Once a split has been assigned to a thread on a worker node it is then executed by the driver loop. The driver loop operates on pages, which are columnar encodings of sequences of rows, and continuously moves them between operators until either the scheduling quanta is complete or the operators can no longer make progress. Presto uses in-memory buffered shuffles to exchange intermediate results: workers store intermediate results on these buffers for the other workers to consume. The engine also adapts the degree of concurrency according to the traffic on the buffers, while also tuning the concurrency degree for the writes.

A relevant characteristic of Presto is its resource management system, fully integrated and fine-grained, which permits Presto to maximize CPU, IO and memory resources. Any given split is allowed to run on a thread for a maximum quanta of one second, after which it relinquishes the thread and returns to the queue. After a split has relinquished a thread Presto chooses the next split based on how much CPU time the splits in the queue have accumulated, which classifies the split in one of five levels. Each one of the levels is allocated a fraction of the available CPU time. Memory allocations are instead divided into system and user memory. User memory is memory usage that the user can easily reason about, while system memory is memory usage that is a byproduct of Presto's implementation. The engine imposes limitations on user and total memory: queries that exceed the global or local limits are killed, while if a node runs out of memory query memory reservations are blocked by halting the processing of tasks. Presto also offers two mechanisms to deal with lack of memory: spilling and reserved pools. With spilling if a node runs out of memory the memory revocation procedure is invoked on tasks in ascending order of execution time and revocation is processed by spilling on disk. With reserved pools, instead, when a node runs out of available memory the memory pool is divided between a general and a reserved pool: once the general memory is completely deleted the most memory demanding query on a node is promoted to the reserved pool on all the worker nodes. Only one query at a time can enter the reserved pool and if the general pool on a node is exhausted while a query is already occupying the reserved pool then all further memory requests are stalled until the reserved pool becomes vacant again, at which point memory requests are unstalled. When it comes to fault-tolerance,

Presto is able to use low-level retries, while it is unable to recover from coordinator failure. Benchmark results demonstrate that Presto's characteristics guarantee performances competitive in the distributed SQL engine landscape.

1.4. Mapreduce

Among the distributed data processing frameworks, MapReduce is one of the most relevant. MapReduce is a programming model, and implementation of the above-mentioned model, aimed at processing large sets of data on a cluster of commodity machines in a distributed fashion [5]. The programming model is based on taking key/value pairs as input and return key/value pairs as output. The computation is divided into two steps: map and reduce, both defined by the user. The map function generates a set of intermediate key/value pairs, in which all values relative to a certain intermediate key are grouped before being passed to the reduce function. To manage large sets of values, the values are passed to the reduce function as iterators. The reduce function instead accepts as input the intermediate keys and the values associated with them and proceeds to merge the aforestated values, when possible, in order to obtain a smaller set of values associated with each key. Conceptually, map accepts as input a pair (k1, v1) and returns a list(k2, v2), while reduce accepts as input a pair (k2, list(v2)) and returns list(v2). MapReduce

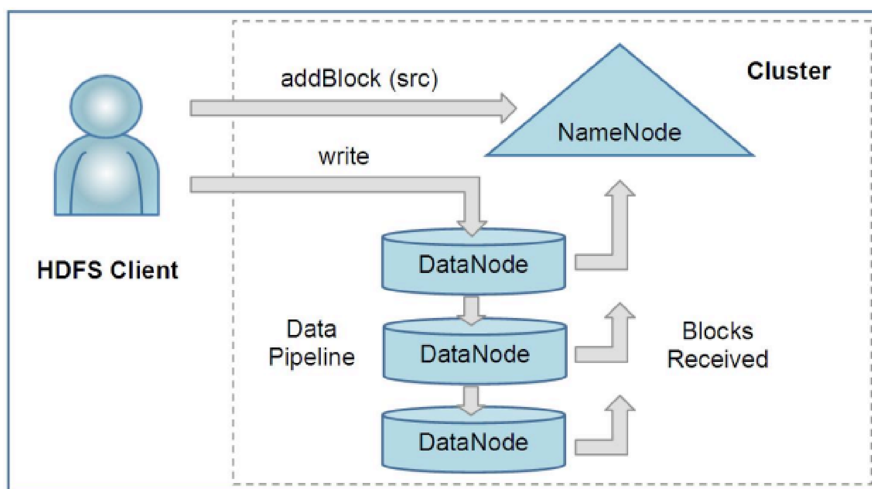

Figure 1.4: HDFS's architecture [19]

was introduced as part of the Hadoop project [19], together with other components. Of particular interest are HDFS and Hive. Apache HDFS is a distributed file system, its architecture is reported in Figure 1.4 [19]. Given how MapReduce has proven itself to be able to efficiently cover a lot of programming problems, it became the framework of choice for distributed data processing jobs [3]. However, the need for the user to be able to understand C++ (or Java) and distributed data processing made it inaccessible to casual users. As a result, multiple abstraction layers over MapReduce were implemented, such as Flume-Java, PIG, HIVE [23] or HadoopDB [3], while, at the same time, distributed DBMS vendors started to introduce MapReduce into their engines to provide an

alternative to SQL for analytical purposes.

Hive was built on top of the infrastructure provided by HDFS as a data warehouse with SQL query capabilities; acting, as previously stated, as an abstraction layer between the user and the underlying MapReduce logic [23]. Queries are expressed in HiveQL, a subset of SQL language, and the statements are submitted via CLI, webUI or one of the external clients available. The driver component then passes the statement to the compiler which parses, type checks and performs semantic analysis over the statement using the data provided by the metastore, a component acting as a catalog for hive. It contains information about tables, schemas, partitions and so on. The compiler generates the execution plan using the data from the metastore first by parsing the query, obtaining an abstract syntax tree. The compiler then performs type checking and semantic analysis by fetching information from the metastore transforming the abstract syntax tree in a query block and then the query block into a DAG. The DAG is then optimized in a rule-based fashion via a series of transformations. Optimization opportunities are found via a walk on the DAG: for each visited node the optimization rules are checked and, if a rule is satisfied, the corresponding optimization is applied. The possible optimization that Hive is capable of comprehend column pruning, predicate pushdown, partition pruning, map side joins, join reordering, repartition of data to handle skewness in GROUP BY statements and hash-based partial aggregation in the mappers. The logical plan is then traduced in multiple MapReduce and HDFS tasks. While the previously introduced solutions suffered from high latency, low efficiency or low SQL compatibility [3], Tenzing managed to avoid these problems. Tenzing is a distributed query engine built on top of MapReduce providing functionalities for both the operational and deep analytics needs.

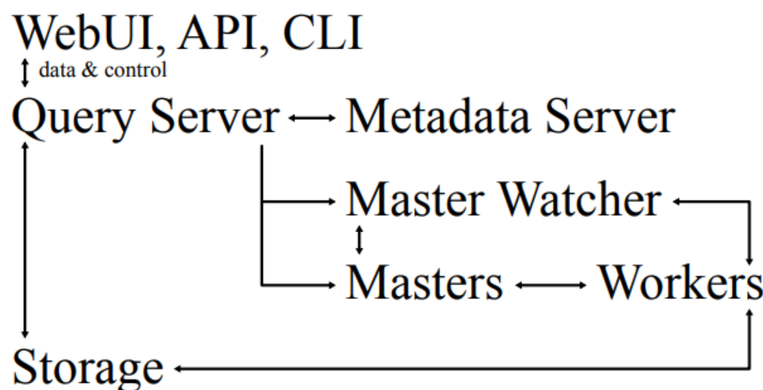

Figure 1.5: Tenzing Architecture [3]

Tenzing's architecture is shown in 1.5. When a query is submitted to Tenzing via the WebUI, the API or the CLI it is parsed by the query server into an intermediate parse tree. The query server, to enrich and complete the intermediate format, then fetches the required metadata from the metadata server and the optimizer optimizes the obtained intermediate format. The optimized plan is composed of various MapReduce operations: for each one of them, the query server selects an available master via the master watcher

and submits the query to it. At this point, the data has been divided into different shards. Idle workers poll the master for work and, once the operation is concluded, reduce workers write the intermediate results on intermediate storage. The query server monitors the intermediate area and, once the results are ready, streams them back to the client. While providing support for projection, filtering, aggregation, various join strategies (broadcast joins, distributed hash joins and remote lookup joins), standard SQL set operations and nested queries, Tenzing also provides support for analytic functions and OLAP extensions (such as ROLLUP() and CUBE(), as examples). Tenzing, thanks to the improvements introduced over MapReduce(such as the workerpool, streaming and in-memory caching as examples), and its scalability capabilities manages to obtain competitive results when it comes to performances.

Another attempt at bridging the gap between MapReduce and SQL was made by DryadLINQ: DryadLINQ acts as an abstraction layer between SQL, MapReduce and Dryad and the user [8]. DryadLINQ's architecture involves a job manager, which is responsible for instantiating a job's dataflow graph, scheduling the process on the cluster, providing fault tolerance (executing again slow or failed processes), monitoring the jobs' statistics and transforming the job dynamically according to the collected statistics. The cluster itself is usually controlled by a task scheduler as a separate entity. The instructions are passed to DryadLINQ in the form of LINQ statements. DryadLINQ then transforms the LINQ expressions into a DAG referred to as execution plan graph (EPG). The EPG is a blueprint of the Dryad dataflow and the computations that will be executed: each node will be used at runtime to generate an execution stage. Both the nodes and the edges are annotated by the optimizer with metadata properties. The optimizer performs both static and at runtime optimizations: static optimizations comprehend pipelining, redundancy removal, eager aggregation and I/O reduction. Dynamic optimizations are done according to the statistics collected at runtime, changing the topology of the EPG accordingly. The supported dynamic optimizations are dynamic hash and range partitioning. Optimization for OrderBy statements and MapReduce plans are done in a hybrid manner, part statically, part dynamically.

DryadLINQ was also used as a starting point for Optimus, a dynamic rewriting framework for data-parallel execution plans integrated with DryadLINQ [12]. Optimus was engineered to bridge the semantic gap between Dryad's rewriting logic and a data parallel program: while a data-parallel query is being executed Optimus collects statistics (both default and user-defined) at the various vertices and creates a graph rewriting message, to then send it to a graph rewriter which modifies the EPG according to the collected statistics. The graph rewriter can dynamically partition the data both on a range and a hash base, while also being able to optimize MapReduce workloads. Another optimization that Optimus is capable of dynamically optimizing join strategies. Last, but not least, Optimus can optimize iterative workload and matrix multiplication operations.

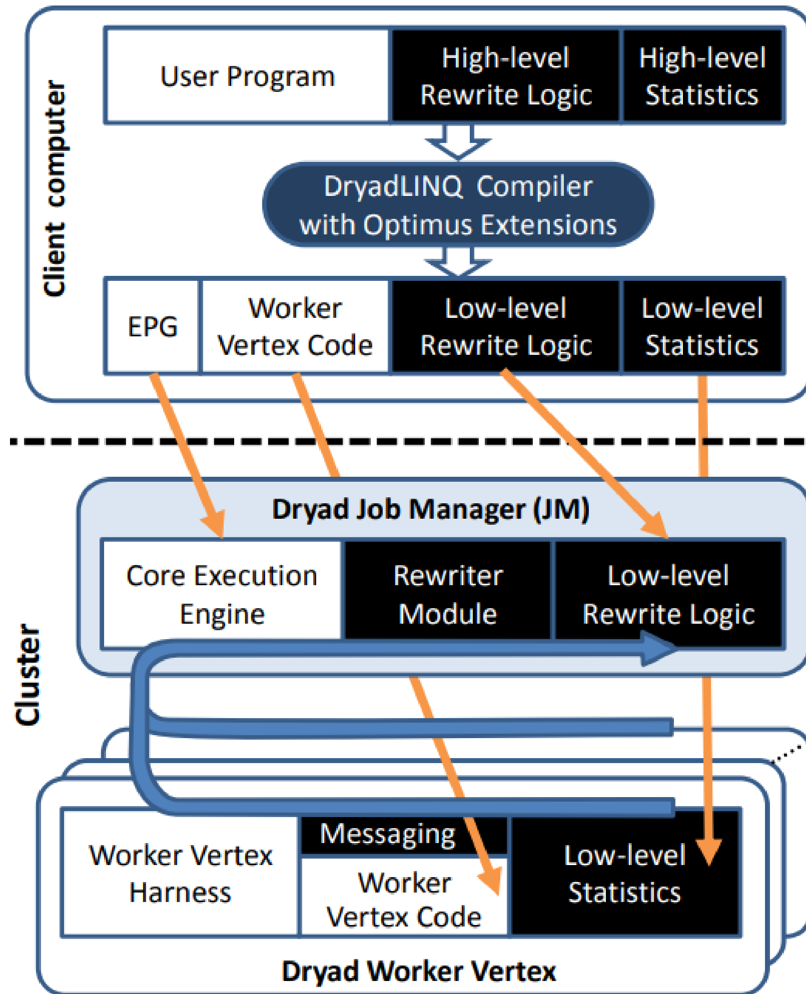

Figure 1.6: Optimus integration in DryadLINQ [12]

1.5. Spark

Among the frameworks built on top of MapReduce Spark is one of great relevance: it aimed to maintain the best characteristics of MapReduce, such as scalability and fault tolerance, while also being built for reusing distributed datasets in a parallel fashion [27]. In Spark, the game-changer feature is the caching which is executed in memory. To do so, Spark introduces the concept of Resilient Distributed Datasets (RDDs). By definition, an RDD is an in memory read-only partitioned collection of records created through deterministic operations either on data in storage or on other RDDs. While the MapReduce model is well suited for a large set of problems, it has been reported as insufficient in two particular use cases: iterative jobs (used commonly by machine learning applications in algorithms such as gradient descent) and interactive analytics. The main abstraction behind Spark are RDDs, which represent a collection of read-only objects partitioned across the memory of multiple machines, that can be rebuilt if a partition is lost because of a fault in one of the machines. Fault tolerance is achieved by an RDD's lineage: to

recover from the loss of data it is sufficient for Spark to know the set of operations from which the RDD was obtained. RDDs can be obtained via applying deterministic operations (transformations) to either data in stable storage or other RDDs. Other relevant aspects of RDDs are their persistence and partitioning properties: users can indicate which RDDs they will reuse and choose a storage strategy for them, while also asking RDDs to be partitioned across different machines based on a partitioning key. This can lead to significant join optimization possibilities. Spark exposes RDDs via APIs, in a fashion similar to DryadLinq (and FlumeJava): each dataset is represented as an object and transformations are invoked via methods on said objects. Like DryadLINQ Spark evaluates transformations lazily and materializes the final RDD after an action invocation, where an action is a method that either returns a value to the application or export data to a storage system. In the table 1.1 RDDs are compared to the Distributed Shared Memory (DSM), a different and more traditional approach in which applications read and write in arbitrary locations in the global address space, providing fine-grained writes.

Aspect	RDDs	DSM
Reads	Coarse- or fine-grained	Fine-grained
Writes	Coarse-grained	Fine-grained
Cosistency	Trivial(Immutable)	Up to app/runtime
Fault Recovery	Fine-grained and low-overhead using lineage	Requires checkpoints and program rollback
Straggler Mitigation	Possible using backup tasks	Difficult
Work placement	Automatic based on data locality	Up to app (runtimes aim for transparency)
Behavior if not enough RAM	Similar to existing data flow systems	Poor performances (swapping?)

Table 1.1: Comparison between RDDs and DSM architectures [26]

The main difference between RDDs and DSM is that RDDs can only be created through coarse-grained transformations. This entails a more optimal fault tolerance: RDDs do not need the overhead of checkpointing, as they can be recovered through lineage, and in case of an error only the lost partitions need to be recomputed. It is important to note that RDDs do not need to be materialized at all times: an RDD has enough data about the transformations used to derive it that it is always able to compute its partitions from data in stable storage. The main challenges in implementing RDDs are linked to providing efficient fault tolerance: in other cluster in-memory storage solutions, such as DSM, key-value stores, databases and Piccolo (a data-centric programming model for in-memory applications [17]), the interface offered is fine-grained. This way the only ways to provide fault tolerance are to either replicate data across the different machines or to log updates across them: both of these approaches are expensive when dealing with data-intensive workloads. RDDs provide instead an interface based on coarse-grained transformations (such as map, filter and join) that apply the same operation to a large number of data items. This way it is possible to log the transformations enacted to build

the lineage of the RDDs. Spark was implemented in Scala and was originally deployed over a Mesos cluster. In order to use Spark the developer has to write a Driver program

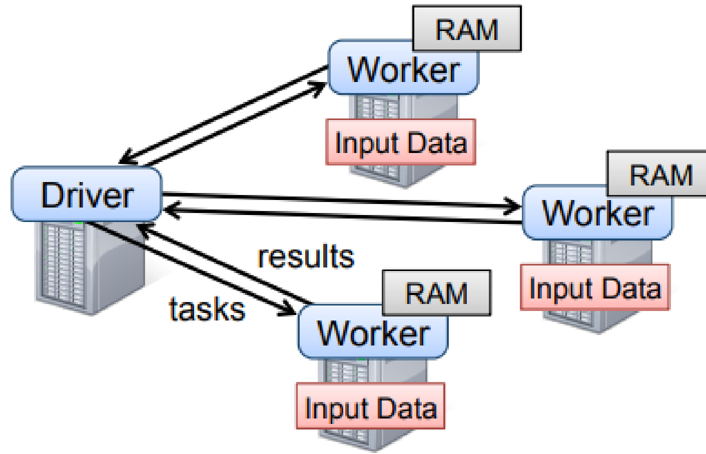

Figure 1.7: Driver and workers architecture [26]

that connects to a cluster of workers, as shown in Figure 1.7. The driver defines the RDDs, keeps track of the lineage and invokes the actions on them, while the workers are long-lived processes that can store RDD partitions in RAM. Spark's scheduler represents RDDs as a set of partitions, atomic elements of the dataset, a set of dependencies on parent RDDs, a function to compute the dataset based on its parents, metadata about its partitioning scheme and data placement. Dependencies are classified as either wide or narrow, according to whether each partition of the parent RDD is used at most by one partition of the child RDD (narrow) or if partitions of the parent are used by more than one partition of the son (wide). This distinction enables pipelining of narrow dependencies on each cluster node and facilitates fault recovery on narrow dependencies. Examples of

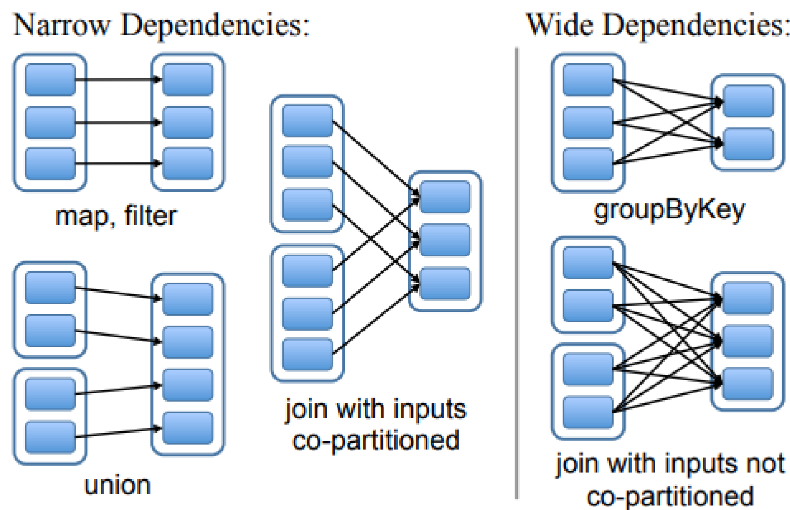

Figure 1.8: Comparison between narrow and wide dependencies [26]

wide and narrow dependencies are reported in Figure 1.8. When the user runs an action

on an RDD, the scheduler examines the RDD's lineage to build a DAG of stages to be executed. The stages contain as many narrow transformations pipelined as possible, with their boundaries defined either by shuffle operations needed for the wide dependencies or by an already computed partition that can shortcircuit the computation. The scheduler then assigns tasks to the various executors based on data locality. If a task fails is run again on another node, as long as the parent RDD is still available; while if some stages become unavailable the tasks needed are re-submitted in parallel. As a further measure towards fault tolerance, Spark supports checkpointing. Further development upon the RDD paradigm was provided by the creation of Shark, a framework that aimed to use the RDD abstraction in order to provide fine-grain fault tolerance while performing most operations in memory, to be able to run SQL statements and to do so while retaining low latency [25].

In order to execute SQL queries efficiently the RDD paradigm needed to be expanded upon. One of the most relevant of these extensions was the introduction of the Partial Dag Execution (PDE). The standard DAG execution does not allow for alterations to the DAG at runtime: PDE modifies this mechanism in two ways. In the first place, it gathers customizable statistics at runtime; second, it allows to modify the execution at runtime. The evaluation on whether to modify the execution phase happens at the shuffle phases (wide transformations). The changes that PDE can make on the execution plan are two: changing the join strategy at runtime (according to the collected statistics) and handling skewness in data and degree of parallelism. After Shark further refinement over Spark was provided by SparkSQL. When compared to previous attempts to incorporate the SQL language in the Spark framework Spark SQL provides two main additions: tighter integration between relational and procedural processing and the addition of the Catalyst optimizer [1]. When comparing Shark to Spark SQL, shark shows three main shortcomings: it was not useful for relational queries on data inside Spark, it could be engaged from Spark programs only from a SQL string, and it was difficult to extend. Spark SQL was therefore created with the goals to support relational processing on both RDDs and external data sources via an accessible API, provide high performances on DBMS techniques, easily integrate new data sources and enable extensions with advanced analytics algorithms. The main abstraction behind Spark SQL's API is a DataFrame, defined as a distributed collection of rows with the same schema. While being equivalent to a table in a relational database it can also be manipulated in ways similar to the ones used on RDDs; thus being possible to consider a DataFrame as an RDD of Row objects. The interfaces to Spark SQL and its interaction with Spark are shown in Figure 1.9. The user can manipulate DataFrames either via a domain-specific language similar to the one used in R for data frames and in Python for Pandas, or via SQL statements. As Shark did, Spark SQL also supports in-memory caching, while also adding support for UDFs. The other relevant addition provided by Spark SQL, as already mentioned, is the Catalyst optimizer

The Catalyst's design had two purposes: make it possible to easily add new optimization techniques and features and to make it possible for external developers to extend it. The Catalyst supports both rule-based and cost-based optimization and, at its core, it contains a general library for representing trees and rules on how to manipulate these

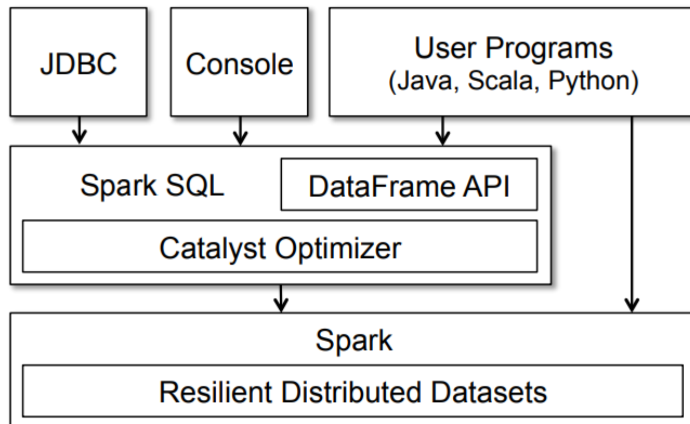

Figure 1.9: Spark SQL interfaces [1]

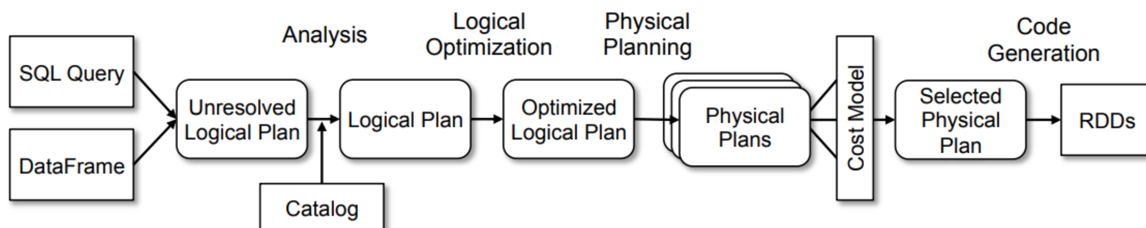

Figure 1.10: Catalyst optimization phases [1]

trees [1]. Plans are represented as tree objects, composed of node objects having a node

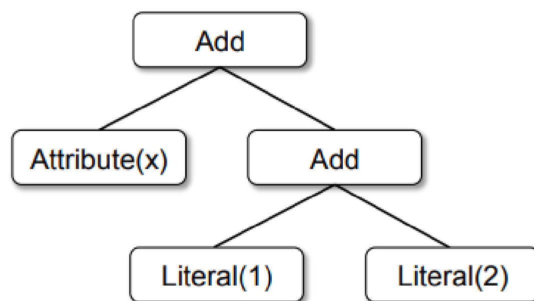

Figure 1.11: Catalyst tree for the expression x + (1+2) [1]

type and, optionally, node children. An example of this representation for the expression "x + (1+2)" is reported in Figure 1.11. The Catalyst's optimization is based on the optimization used in the Exodus [10] and Cascades [9] optimization frameworks, with the significant difference that it does not require domain-specific language usage to interact with it. The optimization process is divided into multiple steps: analysis, logical optimization, physical planning and code generation. In the analysis phase, the relation

that has to be computed by Spark SQL may contain unsolved attribute references: Spark SQL starts by building up a logical plan, then uses Catalyst rules and a catalog object, tracking the tables in the data sources, to solve the aforementioned attributes. The logical optimization step applies rule-based optimization to the logical plan. Given that the rules are expressed via Scala syntax, adding new ones proves to be reasonably easy. In the Physical optimization step, Spark SQL generates one or more physical plans for the logical plan obtained in the previous step and then selects one of them based on the cost model. If there are joins in the workload, the join strategy is selected in this phase. In order to pick the join strategy, the Catalyst follows the decision-making process reported in Figure 1.12. Regarding Figure 1.12, the conditions of applicability for the shuffle hash

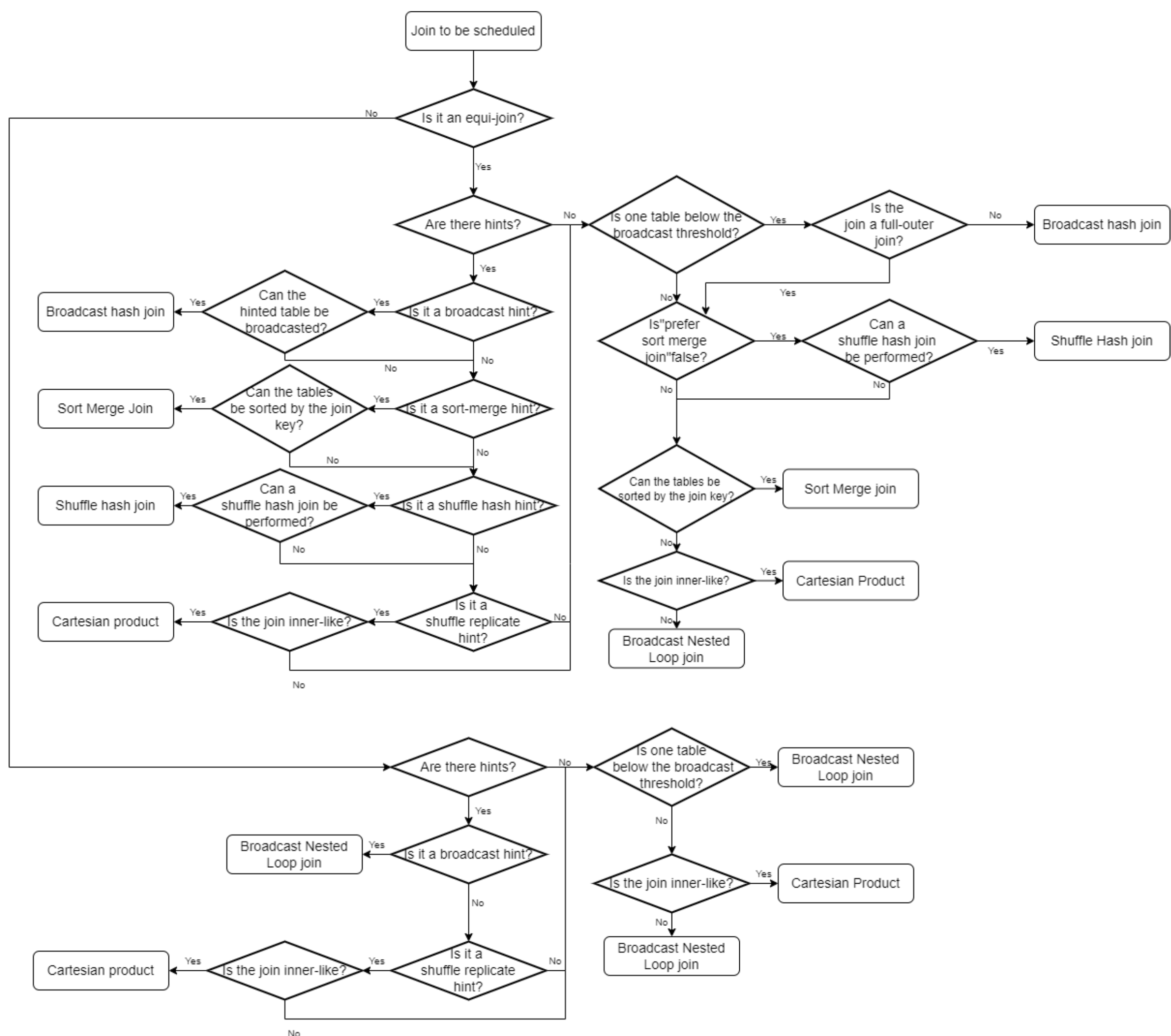

Figure 1.12: Spark join strategy selection flowchart [7]

join are reported here for readability reasons: "if one side is below the local hash map threshold and is at least three times smaller than the other side the shuffle hash join strategy is applicable". It has to be noted that this join type is disabled by default [14]. In the code generation phase, the Catalysts uses Scala's quasiquotes to transform the tree

representing the SQL expression into an abstract syntax tree (AST), a tree in which the nodes represent the operations described either via Dataframe API or SQL. The AST is then evaluated by Scala code, then compiled and run. One of the main drawbacks in this approach is that, once the execution of the query is called by calling an action the plan is evaluated, optimized and executed statically, thus without the possibility of taking into account variations in table dimensions, and thus join strategies. It also does not take into account the impact that partitioning can have on the performances, both in partition number and in partition skewness. SparkSQL also supports extensions for data sources, user-defined types and deep analytics, thus supporting semistructured data and integration with Spark's machine learning library [1]. User feedback and benchmarks have shown Spark SQL to have simplified significantly the development and organization pipelines while offering noticeably better performances than other Spark-based SQL engines.

A further optimization over Spark workloads was introduced in the form of hyperspace, an indexing system for datalakes. [16]. With Azure Synapse Microsoft introduced a tool for data ingestion, storage and querying in both Apache Spark and T-SQL over data both in the datalake and in warehouse tables. Hyperspace was created in order to introduce an indexing subsystem aimed at datalakes, operated via a serverless strategy. Hyperspace is agnostic to data formats and supports indexes for the most common of formats. The indexes are also themselves considered to be data and stored in the datalake in parquet format. Developers are also provided with a framework that allows them to define their own structures, in order to fit a greater variety of use cases. Last but not least, hyperspace has been integrated with Spark's query optimizer in a way transparent to the user. The Hyperspace characteristics described this far allow Hyperspace to have several useful features: in the first place Hyperspace is able to incrementally refresh its indexes. It also supports history tracking for the indexes, bin-packing the indexes and hybrid scan mechanism which allows the user to exploit stale indexes without loss of correctness.

The indexes are non-clustered and have the following three characteristics: they are columnar in format (usually stored in parquet format), they are hash partitioned and can be sorted, allowing to reduce the amount of data to be read for filters and joins with equality predicates and support lineage tracking. This allows table scans to be reduced, or even substituted, with index scans. Figures 1.13 and 1.14 provide an overview of the indexes architecture and how they are stored in the datalake. The most interesting features of Hyperspace indexes, other than their integration with Spark's optimizer, is that they are linked to their management, their multi-user concurrency control mechanisms and their maintenance. The indexes metadata contain information regarding their contents, their lineage and their state. Regarding the state, an index can be in 6 different trainsient stages: creating, when the index is being created, active, when the index has been created successfully and is available for use, refreshing, if a user is refreshing the index, restoring, if it is being restored after deletion, and deleting, if it is being deleted. Indexes also have 3 stable stages: active, deleted or empty/DNE. Figure 1.15 exemplifies how indexes change state. Having a stateful index management mechanism allows for serverless index management. The rules presented in Figure 1.15 are also used as a base for optimistic concurrency control while guaranteeing support for multiple readers and multiple writers.

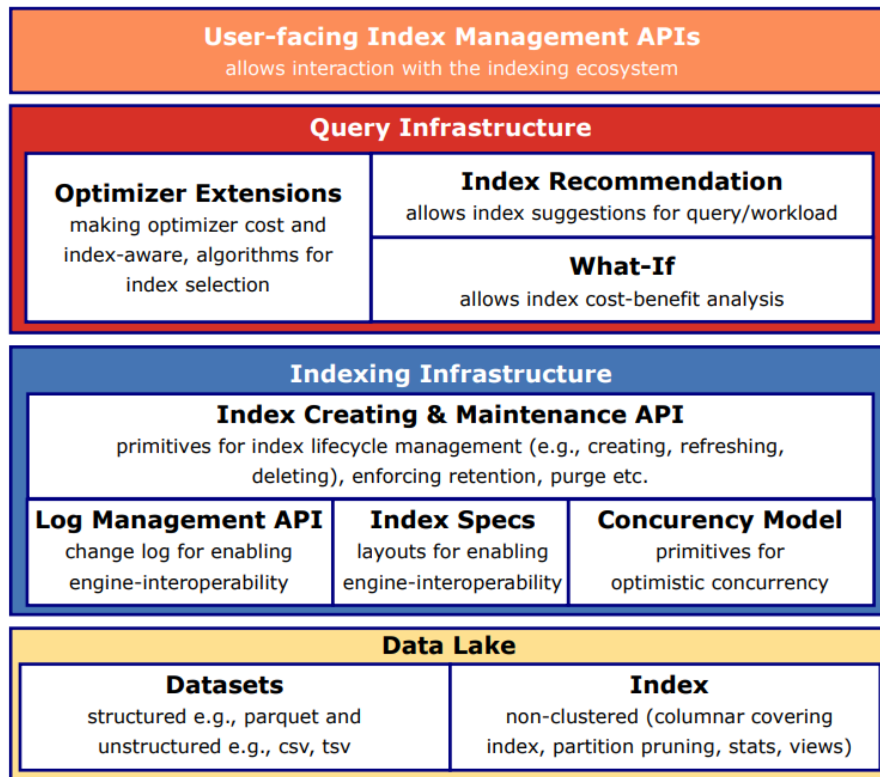

Figure 1.13: Hyperspace indexes architecture [16]

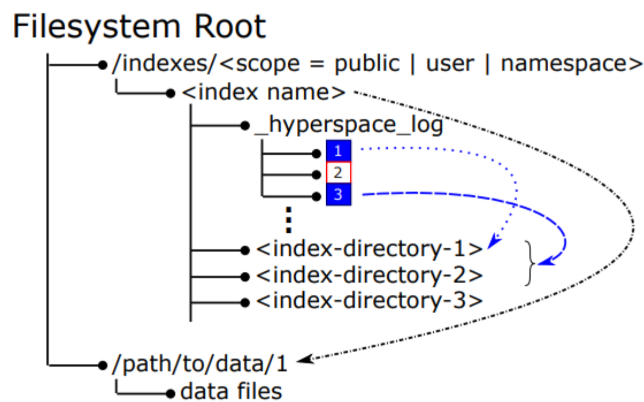

Figure 1.14: How hyperspace indexes are stored in the datalake [16]

It cannot, however, guarantee external consistency. Another relevant characteristic of Hyperspace's indexes is incremental index maintenance: in order to accommodate for the different enterprise workloads the following index update strategies are supported:

1. Full refresh: full rebuild of the index

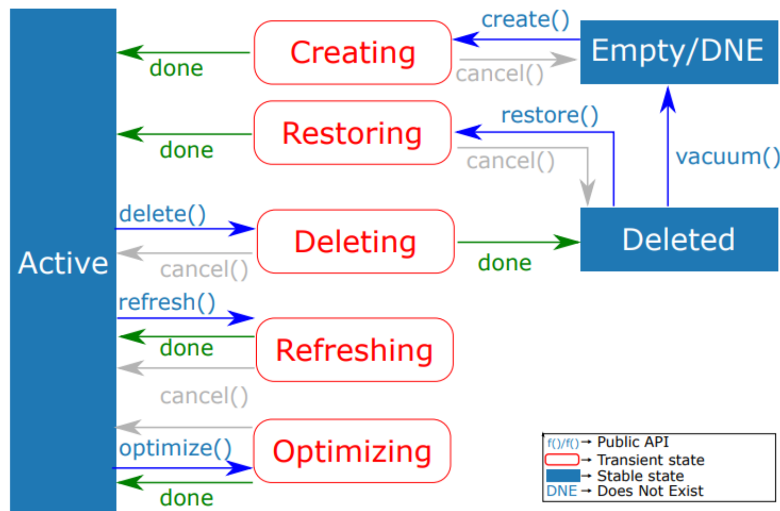

Figure 1.15: Index state machine [16]

2. Incremental refresh: hyperspace determines the updates and variations on the data, then indexes the newly appended data and uses lineage to determine what data was removed and modifies the indexes accordingly. The new index blocks are committed as a new incremental version of the index and metadata is updated accordingly

3. Quick refresh: Hyperspace scans the list of variations in the data and updates the metadata accordingly. At query time Hyperspace will resort to a hybrid scan. Hybrid scan consists in using the existing indexes, even if stale, and updating them with the changes observed on the data sources.

4. Optimize: Hyperspace performs compaction of the smaller indexes, following a user-provided criterion

If Hyperspace is interacting with ACID data or ACID layers over data (e.g. DeltaLake) it will use their transaction logs to update the indexes. As already mentioned, Hyperspace extends the Spark Optimizer in order to integrate indexes usage in the optimization procedures. It does so by adding two new rules to the logical optimization phase: the FilterIndexRule, which substitutes a filtering scan over a table with a filtering scan over an index, where possible, and the joinIndexRule, which enables the usage of indexes in join queries. Experimental results [16] show that the usage of Hyperspace's indexes can provide substantial improvements on the performances of complex workloads. Throughout the years Spark SQL managed to remain relevant in the distributed computation engine landscape, with updates refining its components and capabilities while also adding new ones. In particular Spark 3.0 added, amongst its updates, the adaptive query execution (AQE) capabilities, which enable Spark SQL to enact optimizations at runtime, like other distributed frameworks and query engines did before [15] [8] [12] [18]: AQE collects statistics at runtime and can change the execution plan during shuffle phases [6]. In particular, according to Spark SQL 3.1.2 documentation, AQE boasts three capabilities [20]:

1. Dynamic join selection: if a Sort Merge join is scheduled and one of the tables' dimension fall below the adaptive broadcast threshold the join is then executed as a Broadcast Hash join

2. Automatic post shuffle coalesce: if, after a shuffle, the Catalyst notices a number of partitions too high for a certain dataframe then the dataframe is coalesced in a smaller number of partitions

3. Skew join handling: if skewness in partitions is found the join's execution is altered to reduce the impact of skewness on the performances with the skewed partitions being split

In the next Sections, the efficiency and efficacy of AQE will be extensively tested: in Section 3 AQE capabilities will be considered one at a time and the experiments will be aimed to check whether AQE follows the behavior described in the documentation. In Section 4, instead, the experiments will study the impact of AQE on various workloads.

2 | Problem Definition and Methodology Presentation

The introduction of dynamic optimization amongst Spark's capabilities has the potential to improve the Catalyst's ability to optimize execution plans. Traditionally, once a query execution plan has been statically optimized, the execution begins and the plan can not be optimized further. The ability to dynamically adapt the query execution to how data evolves during the execution of the workload grants the possibility to perform further optimizations that take into account the changes that happen during execution. While the concept of dynamic query optimization is not a new one, AQE is still to be extensively tested. The documentation provides insight over the tuneable configurations involved in AQE's usage, however the only real insight on its inner working can be currently found in the source code. This, while incredibly helpful in understanding how AQE engages in its activities, does not provide a good amount of insight on how AQE behaves when interacting with the other Spark's components. There is currently no detailed definition of its behavior, its strengths and its shortcomings, as well as its consistency in the operations it performs. To better understand AQE's impact on performances the next experiment Sections will test extensively its components. The baseline group of tests will be used to isolate and test each component individually in a controlled and local environment. This will be done via suit-tailored workloads and data to understand how they work from a purely technical standpoint. The workload test group will instead make use of workloads currently used in production. The workload tests will aim to provide insight on how AQE's usage affects performances in real-world scenarios. The baseline group of experiments is entirely dedicated to testing the AQE functionalities offered by Spark 3.1.2 in an isolated environment. In this Section, dynamic join selection, which allows Spark to change the physical execution plans during execution, automatic coalesce of partitions, which allows Spark to automatically recognize situations in which coalescing partitions would be beneficial to performances, and handling skewed joins by recognizing skewed partitions and repartitioning them [20] will be tested. These experiments will share light on how they work, how to tune Spark to make them effective and how they interact with Spark's static optimization procedures. AQE is engaged between stages [20]: once a stage is done the new statistics regarding tables and partitions are collected and the execution plan is altered accordingly. As such, to divide the execution plan into stages and trigger AQE's optimization procedure a shuffle phase is needed [6]. The following experiments target the different AQE capabilities once at a time: this is done to better analyze the singular components.

All the baseline experiments were executed on a laptop (16 GB RAM, i5-8350U CPU) with Spark in local mode. To enable AQE it is necessary to set the `spark.sql.adaptive.enabled` configuration as true. This however enables all AQE features [20], therefore further configuration was required to isolate them. Section 3.1 will explore how AQE can dynamically change the join strategy according to the original join choice. According to the documentation, in Spark 3.1.2 AQE can switch from a Sort Merge join to a Broadcast Hash join dynamically when one of the tables involved in the join falls below the Broadcast Hash join threshold during execution. The experiments aim to check whether Spark behaves as documented, checking its interaction with the most common join strategies. To isolate the feature and to test the interaction with the various join types, following the specification from documentation and source code [20] [14] [7], the following configurations were modified:

- `spark.sql.adaptive.enabled`: True , in order to enable AQE
- `spark.sql.adaptive.coalescePartitions.enabled`: False
- `spark.sql.adaptive.skewjoin.enabled`: False
- `spark.sql.autoBroadcastjoinThreshold`: threshold for the Broadcast Hash join strategy, both statically and dynamically scheduled. The value at which it is set varies according to the experiment's needs

The Dynamic join Selection Section 3.1 is divided into two Subsections, with and without hints. This is so since the usage of hints entails a different decision process from the Catalyst from the one used without them. Furthermore, the usage of hints implies pushing the Catalyst towards picking join strategies it otherwise wouldn't have picked. Given these considerations, the usage of hints has been considered separately. In Subsection 3.1.1 the experiments study how AQE interacts with the various join strategies without the usage of hints. The join strategies that are included in these tests are Broadcast Hash join, Sort Merge join, Cartesian Product and Broadcast Nested Loop join. The shuffle hash join has not been considered, given that in Spark 3.1.2 it is disabled by default via the `spark.sql.preferSortMergejoin` configuration set as true [14]. To trigger each of the considered join strategies various workloads have been defined and the Spark configurations have been tuned to satisfy the conditions according to which the Catalyst, without AQE's intervention, chooses the join strategy [7]. For each of the join strategies, the workload will be executed twice, with AQE enabled and disabled. The obtained results will then be compared to provide an accurate picture of how AQE affects the Catalyst's decisions in front of the most common join types. Once the interaction with the join strategies will have been defined the experiments will explore the interaction between the Dynamic join Selection and hints usage. As already mentioned, the usage of hints alters the Catalyst's decision process, therefore interfering with its regular join strategy evaluation. These experiments will test if and how hints may affect AQE's Dynamic join Selection. They will also test whether it is possible to obtain the same results that AQE would yield by using hints, thus eliminating AQE's overhead costs. To perform these controls, the workloads used in the previous Section will be executed again with

hints and different Spark configurations. The join strategies investigated in this Section are Broadcast Hash join, Sort Merge join and Cartesian Product. Shuffle Hash join has been discarded, again because of being disabled by default [14], as was discarded also the Broadcast Nested Loop join: the Broadcast Nested Loop is used as a last resort when all of the other strategies cannot be enacted [7]. As such, it is reasonable to expect that the other strategies would not be applicable, even with hints involved. As per the kinds of hints used, with Broadcast Hash joins only broadcast hints will be used, as not to hint towards worse-performing join strategies, to provide a null case. Concerning Sort Merge join, Broadcast and Sort Merge hints will be used, respectively to check whether the same results obtained via AQE can be obtained with hints and to observe whether the usage of a hint could override AQE's actions. Last, but not least, concerning Cartesian Product only broadcast and cartesian hints will be used, similarly as was done with Sort Merge join. The Sort Merge hint has not been considered since, according to the documentation [7], a Cartesian Product will only be scheduled when a Sort Merge join is not possible (or in the presence of a cartesian hint), therefore it would have no effect. To perform these experiments, the same workloads used in the last Section will be used again, with the addition of hints and different Spark configurations as needed. Once the tests on this component are done, the next one to be tested is the Automatic post shuffle coalesce. According to documentation, AQE should be able to automatically coalesce partitions after a shuffle phase when certain conditions, defined by some Spark configurations, are met. This opens to two main improvements: join execution time and data read/write performances. To better understand the capabilities of this component of AQE's, its capability to coalesce was tested both on joins containing shuffle phases and joins that don't. The test were thus conducted on a Sort Merge join, which contains a shuffle phase during the sort, and a Broadcast Hash join, which does not involve shuffle phases. A test checking if the setting of the minimum number of partitions works properly was also included. Other than checking whether AQE could coalesce partitions, another group of experiments was introduced to check whether AQE is also able to increase the number of partitions if needed via an automatic repartition. In this group the Broadcast Hash join and Sort Merge join have been selected as representative of joins that are not preceded by a shuffle phase and that are, respectively. Both when testing the coalescing and repartitioning capabilities, on both classes of joins, the workloads have been executed with and without the AQE module of interest enabled, to provide a term to compare AQE's effect against a base case, thus having the necessary data to evaluate how AQE performs. As already done in the previous Section, the AQE module of interest was tested in an isolated fashion. This was achieved by both Spark configuration, tuned according to documentation [14] [20], and by design of the workload:

- `spark.sql.adaptive.enabled`: True , in order to enable AQE
- `spark.sql.adaptive.coalescePartitions.enabled`: True in order to enable the feature studied in this Section
- `spark.sql.adaptive.skewjoin.enabled`: False
- `spark.sql.autoBroadcastjoinThreshold`: threshold for the Broadcast Hash join

strategy, the value at which it is set varies according to the experiment's needs

- `spark.sql.adaptive.coalescePartitions.minPartitionNum`: used to set the minimum number of partitions after the automatic coalesce has been performed. The value varies according to the needs of the experiment

- `spark.sql.adaptive.advisoryPartitionSizeInBytes`: left at the default value of 64MB

- `spark.sql.adaptive.coalescePartitions.initialPartitionNum`: default value of `none`

It is worth noting that, while the skew join component can be disabled via Spark's configurations, the dynamic join selection can't. Both the experiments' workloads and the broadcast settings have been tuned to avoid the dynamic selection of join strategies. The third AQE component to be tested is its capability of handling skew joins. When this component is enabled AQE can identify skewness in joins, which means it can identify unbalanced tasks used in the aforementioned joins, and rebalance them. This is done by repartitioning the partitions related to the skewed tasks and duplicating the corresponding partitions on the other side of the join [20]. To test this functionality, various join strategies will be applied on data presenting skewness in the join column, with AQE enabled and disabled. The obtained results will then be compared to give a comprehensive view of how AQE acts upon sewed joins. The join strategies considered will be Broadcast Hash join, Sort Merge join, Cartesian Product and Broadcast Nested Loop join. Once again, Shuffle Hash join has not been considered, since it is disabled by default on Spark. Particular attention will be given to the Sort Merge join strategy: handling skewed partitions relies on the CustomShuffleReader [4] to identify skewness in joins. The Sort Merge join strategy is the only join strategy considered that involves a shuffle phase: it can thus prove to be a good testing ground to shed light on the inner workings of skewness recognition and correction, particularly concerning the configurations involved in the definition of skewness. Once again, the queries have been engineered to trigger a particular kind of join based on the comments in Spark's source code [7], while the triggering and isolation of the feature has been achieved via Spark configuration [20] [14]:

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

- `spark.sql.adaptive.coalescePartitions.enabled`: False in order to disable the feature

- `spark.sql.adaptive.skewjoin.enabled`: False

- `spark.sql.autoBroadcastjoinThreshold`: both static and dynamic threshold for the Broadcast Hash join strategy, the value at which it is set varies according to the experiment's needs

- `spark.sql.adaptive.advisoryPartitionSizeInBytes`: sets the advised size for

the partitions obtained by repartitioning the skewed ones, tuned according to the needs of the single experiment

- `spark.sql.adaptive.skewjoin.skewedPartitionThresholdInBytes`: sets the absolute threshold for skewed partitions, tuned according to the needs of the single experiment

- `spark.sql.adaptive.skewjoin.skewedPartitionFactor`: sets the relative skewness threshold as the median partition dimension multiplied by this factor. For a partition to be considered skewed its dimension must be over both thresholds. Tuned according to the needs of the single experiment

- `spark.sql.shuffle.partitions`: 50, sets the automatic shuffle partitions number. Here it is set to the same number as the distinct values that the skewed join key can take to ensure skewness in the partitions.

As previously stated it is important to note that, while the automatic coalesce of partitions can be disabled via Spark's configurations, the dynamic join selection can't. To avoid dynamic join strategy changes the workloads and the broadcast threshold have been tuned here too.

The results of this Section will explain how AQE features work and will provide a baseline over their usage. In the following Section the testing will move from the curated and isolated environment used this far to AQE's usage on workloads currently in use in production ETL pipelines. The testing will involve 2 workloads, here called workload A and B. Workload A is responsible for the updates over delta tables, while workload B executes a query whose DAG involves highly filtering statements followed by joins usually scheduled as sort-merge joins.

Workload A is executed over a cluster presenting 4 executors, each with 4 cores and 27GiB of memory, overhead included. Testing over workload A will involve two experiments: in the first one, which will last one week, the automatic coalesce of post-shuffle partitions will be enabled. The experiment will aim at testing how changing the number of partitions will impact the joins performances and how the different number of partitions influences the read and write operations: the number of partitions translates into the number of files that have to be modified on disk. Smaller and more numerous partitions will require more operations and more files modified, however leaving more rows untouched. On the opposite side, larger partitions will require rewriting more data, however making use of fewer tasks. Keeping this trade-off in mind, the first experiment will see the usage of the automatic coalesce with the following AQE-related settings:

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

- `spark.sql.adaptive.coalescePartitions.enabled`: True in order to enable the feature studied in this phase

- `spark.sql.adaptive.skewjoin.enabled`: False
- `spark.sql.autoBroadcastjoinThreshold`: -1, as such both for reasons related to the workload and to disable the dynamic join selection
- `spark.sql.adaptive.coalescePartitions.minPartitionNum`: used to set the minimum number of partitions after the automatic coalesce has been performed. The value is set to 16, equal to the number of CPU cores available to the executors, in order to fully make use of the available parallelism
- `spark.sql.adaptive.advisoryPartitionSizeInBytes`: 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
- `spark.sql.adaptive.coalescePartitions.initialPartitionNum`: left to the default value of `none`, as it was considered to be irrelevant to this experiment.

After this experiment, the possibility of dynamic join selection will be added in a second one by setting the broadcast threshold to 20 MiB and the minimum number of partitions will be set to 16, which is the number of cores among the executors in the cluster. While workload A is not expected to have any instance in which a Sort Merge join could be transformed into a Broadcast Hash join. These experiments will test how checking for such transformations when these transformations cannot be applied acts on performances, while also providing further data on the impact of the automatic post-shuffle coalesce when the number of partitions is set to take full advantage of the available parallelism. Once again AQE will be evaluated by the average workload execution time and its impact on delta merge operations.

Workload B is executed over a cluster, presenting 2 executors, each having 4 cpu cores and 43 GiB of RAM, overhead included. The experiments on workload B will instead involve testing, at first, AQE's ability to dynamically switch join strategy outside of an environment created to enable it to do so. The Spark configurations related to AQE are as follows:

- `spark.sql.adaptive.enabled`: True , in order to enable AQE
- `spark.sql.adaptive.coalescePartitions.enabled`: False
- `spark.sql.adaptive.skewjoin.enabled`: False
- `spark.sql.autoBroadcastjoinThreshold`: 20 MiB, the value is set at this value as, after analyzing multiple executions of this workload, 20 MiB is high enough to allow broadcasting some of the joined tables, while being low enough to avoid crashing the driver or timing out on the broadcast.

AQE will be evaluated based on the impact it has on the average execution time of

Workload B over the period of eight days. This experiment will be followed by another one which will instead be focused on testing the automatic post shuffle coalesce, once again because of the benefits it could bring in join execution and in read/write operation, thus using the following configurations:

- `spark.sql.adaptive.enabled`: True , in order to enable AQE
- `spark.sql.adaptive.coalescePartitions.enabled`: True in order to enable the feature studied in this phase
- `spark.sql.adaptive.skewjoin.enabled`: False
- `spark.sql.autoBroadcastjoinThreshold`: -1, in order to disable the dynamic join selection
- `spark.sql.adaptive.coalescePartitions.minPartitionNum`: 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
- `spark.sql.adaptive.advisoryPartitionSizeInBytes`: 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
- `spark.sql.adaptive.coalescePartitions.initialPartitionNum`: left to the default value of 'none', as it was considered to be irrelevant to this experiment.

This experiment too will last 8 days.

Given that the experiments in Chapter 3 will be executed locally on a commodity laptop and the experiments in Chapter 4 will be executed on a computation cluster, AQE's behavior will be compared amongst the two platforms. This is so in order to identify any discrepancy in AQE's behavior depending on how Spark is deployed.

In Chapter 5, the results of both the experimental phases will be brought together and analyzed, to give a comprehensive view of AQE's functionalities and their impact on the sample workloads, the reasons behind their effects, and insights regarding the overall usage of AQE.

3 | Baseline Tests

The tests in this Chapter will provide insights into how AQE works, how to trigger its features and how they alter the execution plan that would be otherwise executed. The experiments in this Chapter have been executed on a commodity laptop (16GB RAM, i5-8350U CPU) with Spark in local mode, while making use of suit-tailored data and workloads.

3.1. Dynamic join Selection

In this part, the experiments aim to explore the interactions between AQE's ability to dynamically switch join strategy and the various join strategy that can be enacted by Spark. According to documentation, AQE should be able to transform Sort Merge joins in Broadcast Hash joins when the size of one of the joined tables falls below the broadcast threshold [20] [14]. Without AQE Spark is only able to select the type of join statically: once the join types have been selected and the DAG has been defined it will not be altered during the execution of the transformations and actions part of the workflow. With AQE, instead, it is possible to re-engage the Catalyst during shuffle phases [6] to reconsider the join strategy and choose the one that best fits the current situation. Given that the user can suggest the type of join to the Catalyst via the usage of hints, this part will be divided in 2 Sections: in Section 3.1.1 the behavior of AQE will be analyzed without the use of hints; while in Section 3.1.2 the behavior of AQE will be analyzed in the presence of hints.

In the experiments in this Section the following Spark configurations were used:

- With AQE:

```
val spark= SparkSession
.builder()
.appName("SortMergejoinAQEOn")
.master("local[*]")
.config("spark.sql.adaptive.enabled","true")
.config("spark.sql.adaptive.coalescePartitions.enabled","false")
.config( "spark.sql.adaptive.skewjoin.enabled","false")
.config("spark.sql.autoBroadcastjoinThreshold", threshold)
.config("Spark.eventLog.enabled", "true")
.config("Spark.eventLog.dir","Spark-aqe-main/src/main/resources
```

```
    /historyServer/logs")
.getOrCreate()
```

- Without AQE:

```
val spark= SparkSession
.builder()
.appName("SortMergejoinAQEOff")
.master("local[*]")
.config("spark.sql.adaptive.enabled","false")
.config("Spark.eventLog.enabled", "true")
.config("Spark.eventLog.dir","Spark-aqe-main/src/main/resources
    /historyServer/logs")
.config("spark.sql.autoBroadcastjoinThreshold", threshold)
.getOrCreate()
```

The experiments in this Section make use of the tables A, B and Keys as data sources for, respectively, dfA, dfB and dfK

3.1.1. Without hints

In this Section, the experiments explore how AQE dynamically optimizes the join strategy in the absence of hints. According to documentation, AQE should be able to recognize when a Sort Merge join could be replaced with a Broadcast Hash join [20]. While verifying whether this is true, these experiments explore also the interaction between AQE and the various join strategies that can be enacted by Spark.

In order to trigger a specific join type the necessary variations of the join conditions [7] and Spark configurations [14] were applied ; otherwise the experiments in this Section share the same 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 two joins are present, the one of interest is the second one, the one that as result yields dfResult. The first one is used to mask the statistics from the Catalyst. Given that the filters used are pushed down to the read stage during logical optimization the obtained dataframe is small enough to be broadcasted. By adding another join the Catalyst can no longer be sure about the dimension of the dataframe and thus the Broadcast Hash join is

no longer chosen as the join strategy statically. It has to be noted that joining dfA with dfK returns a dataframe with the same number of rows as dfA. The repartition is used to trigger a shuffle and engage AQE [6]. The only experiments using a different workload in this Section are the ones in the Broadcast Hash join section.

Broadcast Hash Join

The experiment in this Section checks whether the usage of AQE's dynamic join strategy could have any effect in a situation in which the strategy chosen by the Catalyst would be a Broadcast Hash join.

The settings of this experiment are as follows:

```
val threshold= 3 * 1024 * 1024
```

As already mentioned, the experiments in this Section use a different workload:

```
val dfA=spark.read.parquet("Spark-aqe-main/src/main/resources/bigTableA")
val dfK=spark.read.parquet("Spark-aqe-main/src/main/resources/keys")
                .repartition(200)
val dfResult_ =dfA.join(dfK, dfA("LAVORO_A")===dfK("LAVORO_KEYS"), "left")

print(dfResult_.explain())
dfResult_.count()
```

This is so because adding an intermediate join phase would obscure the Catalyst's estimates for the second join and prompt it to choose a Sort Merge join strategy. Executing the workload with both AQE enabled and disabled shows that there are no differences between the two(Appendix B, Section 8.1.1). This is not surprising: as stated in the documentation AQE can only change join strategy when the one selected statically is a Sort Merge join.

Sort Merge Join

According to documentation, AQE should be able to dynamically transform a Sort Merge join in a Broadcast Hash join, when one of the two tables falls under the broadcast threshold during execution [20].

The settings for this particular experiment were as follows:

```
val threshold= 1 * 1024 * 1024

join_conditions= dfA("LAVORO_A")===dfAux("LAVORO_B")

join_type=left
```

The same workload was also executed removing the repartition on dfAux, to test whether AQE could detect the possibility to dynamically switch from a Sort Merge join to a Broadcast Hash join in the absence of an intermediate shuffle action.

Executing the workload with AQE disabled it is possible to see that the Catalyst statically chooses Sort Merge join as join strategy:

- Physical plan with AQE and no intermediate repartition:
  - Initial plan:

```
+- == Initial Plan ==
   HashAggregate (unknown)
   +- Exchange (unknown)
      +- HashAggregate (unknown)
         +- Project (unknown)
            +- SortMergejoin LeftOuter (unknown)
               :- Sort (unknown)
               :  +- Exchange (unknown)
               :     +- Scan parquet (1)
               +- Sort (unknown)
                  +- Exchange (unknown)
                     +- Project (unknown)
                        +- BroadcastHashjoin LeftOuter BuildRight
                            (unknown)
                           :- Project (unknown)
                           :  +- Filter (unknown)
                           :     +- Scan parquet (6)
                           +- BroadcastExchange (unknown)
                              +- Filter (unknown)
                                 +- Scan parquet (10)
```

  - Final plan:

```
== Physical Plan ==
AdaptiveSparkPlan (26)
+- == Final Plan ==
   * HashAggregate (25)
   +- ShuffleQueryStage (24)
      +- Exchange (23)
         +- * HashAggregate (22)
            +- * Project (21)
               +- SortMergejoin LeftOuter (20)
                  :- * Sort (5)
                  :  +- ShuffleQueryStage (4)
                  :     +- Exchange (3)
                  :        +- * ColumnarToRow (2)
                  :           +- Scan parquet (1)
```

```
                  +- * Sort (19)
                     +- ShuffleQueryStage (18)
                        +- Exchange (17)
                           +- * Project (16)
                              +- * BroadcastHashjoin LeftOuter
                                  BuildRight (15)
                                 :- * Project (9)
                                 :  +- * Filter (8)
                                 :     +- * ColumnarToRow (7)
                                 :        +- Scan parquet (6)
                                 +- BroadcastQueryStage (14)
                                    +- BroadcastExchange (13)
                                       +- * Filter (12)
                                          +- * ColumnarToRow (11)
                                             +- Scan parquet (10)
```

- Physical plan with AQE:
  - Initial plan:

```
+- == Initial Plan ==
   HashAggregate (unknown)
   +- Exchange (unknown)
      +- HashAggregate (unknown)
         +- Project (unknown)
            +- SortMergejoin LeftOuter (unknown)
               :- Sort (unknown)
               :  +- Exchange (unknown)
               :     +- Scan parquet (1)
               +- Sort (unknown)
                  +- Exchange (unknown)
                     +- Exchange (unknown)
                        +- Project (unknown)
                           +- BroadcastHashjoin LeftOuter BuildRight
                               (unknown)
                              :- Project (unknown)
                              :  +- Filter (unknown)
                              :     +- Scan parquet (6)
                              +- BroadcastExchange (unknown)
                                 +- Filter (unknown)
                                    +- Scan parquet (10)
```

  - Final plan:

```
== Physical Plan ==
AdaptiveSparkPlan (27)
+- == Final Plan ==
```

```
* HashAggregate (26)
+- ShuffleQueryStage (25)
   +- Exchange (24)
      +- * HashAggregate (23)
         +- * Project (22)
            +- * BroadcastHashjoin LeftOuter BuildRight (21)
               :- CustomShuffleReader (5)
               :  +- ShuffleQueryStage (4)
               :     +- Exchange (3)
               :        +- * ColumnarToRow (2)
               :           +- Scan parquet (1)
               +- BroadcastQueryStage (20)
                  +- BroadcastExchange (19)
                     +- ShuffleQueryStage (18)
                        +- Exchange (17)
                           +- * Project (16)
                              +- * BroadcastHashjoin LeftOuter
                                 BuildRight (15)
                                 :- * Project (9)
                                 :  +- * Filter (8)
                                 :     +- * ColumnarToRow (7)
                                 :        +- Scan parquet (6)
                                 +- BroadcastQueryStage (14)
                                    +- BroadcastExchange (13)
                                       +- * Filter (12)
                                          +- * ColumnarToRow (11)
                                             +- Scan parquet (10)
```

- Physical plan without AQE:

```
== Physical Plan ==
* HashAggregate (22)
+- Exchange (21)
   +- * HashAggregate (20)
      +- * Project (19)
         +- SortMergejoin LeftOuter (18)
            :- * Sort (4)
            :  +- Exchange (3)
            :     +- * ColumnarToRow (2)
            :        +- Scan parquet (1)
            +- * Sort (17)
               +- Exchange (16)
                  +- Exchange (15)
                     +- * Project (14)
                        +- * BroadcastHashjoin LeftOuter BuildRight (13)
                           :- * Project (8)
                           :  +- * Filter (7)
                           :     +- * ColumnarToRow (6)
```

```
:       +- Scan parquet (5)
+- BroadcastExchange (12)
   +- * Filter (11)
      +- * ColumnarToRow (10)
         +- Scan parquet (9)
```

As shown by the physical execution plans obtained with AQE enabled, the join strategy was dynamically switched from Sort Merge join to Broadcast Hash join in the presence of an intermediate shuffle stage (step 18 without AQE, 21 with AQE, 20 with AQE and no repartition).

It is possible to conclude that if, there is an intermediate shuffle stage between the filtering of a table, in which the table becomes small enough to be broadcasted, and a Sort Merge join in which the table is involved, AQE dynamically switches from Sort Merge join to Broadcast Hash join. This is, however, different from how the documentation [20] and the developers' comments [6] define AQE's action. According to the aforementioned sources, AQE should be able to identify the changes in the tables' dimensions and act accordingly. The execution of the workload without the repartition, however, shows the opposite, as the join type is not changed dynamically. The reason behind this behavior can be found in how AQE optimizes the further stages [11]: once an exchange phase starts AQE will optimize the subsequent stages according to the collected statistics. It is not, however, able to optimize the current stage. As such, AQE cannot change the join strategy before the exchange phase needed for the Sort Merge join and it thus requires a further intermediate shuffle stage.

Cartesian Product

According to documentation, the ability of AQE to change the join type dynamically is limited to the Sort Merge join strategy. These experiments are aimed to check whether AQE could have any side effect when the Cartesian Product is statically chosen as join strategy.

The join conditions in the following experiments are as follows:

```
val threshold= 1 * 1024 * 1024

join_conditions= dfA("LAVORO_A")===dfAux("LAVORO_B") or
                 dfA("AGE")===dfAux("FILLER1")

join_type=inner
```

A first test, executed with the broadcast threshold set as 1 * 1024 * 1024 bytes highlights the following discrepancy between the result returned by the explain() function and the physical plan in the SparkUI:

```
explain() function:
```

```
== Physical Plan ==
CartesianProduct ((LAVORO_A#2 = LAVORO_B#10) OR (AGE#4 = FILLER1#11))
:- *(1) ColumnarToRow
:  +- FileScan parquet [NOME#0,COGNOME#1,LAVORO_A#2,BIRTHPLACE#3,AGE#4]
    Batched: true, DataFilters: [], Format: Parquet, Location:
    InMemoryFileIndex[file:/C:/Users/pipitaan/IdeaProjects/Sparkaqe/Spark-aqe-main
    /src/main/resources..., PartitionFilters: [], PushedFilters: [],
        ReadSchema:
        struct<NOME:string,COGNOME:string,LAVORO_A:string,BIRTHPLACE:string,AGE:double>
+- Exchange RoundRobinPartitioning(200), REPARTITION_WITH_NUM, [id=#61]
   +- *(3) BroadcastHashjoin [LAVORO_B#10], [LAVORO_KEYS#18], LeftOuter,
       BuildRight, false
      :- *(3) Filter ((((((FILLER1#11 = FILLER3#13) AND isnotnull(FILLER1#11))
          AND isnotnull(FILLER2#12)) AND isnotnull(FILLER3#13)) AND (FILLER1#11
          = FILLER2#12)) AND (FILLER2#12 = FILLER3#13))
      :  +- *(3) ColumnarToRow
      :     +- FileScan parquet [LAVORO_B#10,FILLER1#11,FILLER2#12,FILLER3#13]
          Batched: true, DataFilters: [(FILLER1#11 = FILLER3#13),
          isnotnull(FILLER1#11), isnotnull(FILLER2#12), isnotnull(FILLER3#13),
          ..., Format: Parquet, Location:
          InMemoryFileIndex[file:/C:/Users/pipitaan
      /IdeaProjects/Sparkaqe/Spark-aqe-main/
        src/main/resources..., PartitionFilters: [], PushedFilters:
            [IsNotNull(FILLER1), IsNotNull(FILLER2), IsNotNull(FILLER3)],
            ReadSchema:
            struct<LAVORO_B:string,FILLER1:double,FILLER2:double,FILLER3:double>
      +- BroadcastExchange HashedRelationBroadcastMode(List(input[0, string,
          false]),false), [id=#57]
         +- *(2) Filter isnotnull(LAVORO_KEYS#18)
            +- *(2) ColumnarToRow
               +- FileScan parquet [LAVORO_KEYS#18] Batched: true, DataFilters:
                   [isnotnull(LAVORO_KEYS#18)], Format: Parquet, Location:
                   InMemoryFileIndex[file:/C:/Users/pipitaan/IdeaProjects/Sparkaqe
                /Spark-aqe-main/src/main/resources..., PartitionFilters: [],
                    PushedFilters: [IsNotNull(LAVORO_KEYS)], ReadSchema:
                    struct<LAVORO_KEYS:string>
```

SparkUI:

```
== Physical Plan ==
* HashAggregate (19)
+- Exchange (18)
   +- * HashAggregate (17)
      +- * Project (16)
         +- BroadcastNestedLoopjoin Inner BuildLeft (15)
            :- BroadcastExchange (3)
            :  +- * ColumnarToRow (2)
            :     +- Scan parquet (1)
```

```
         +- Exchange (14)
            +- * Project (13)
               +- * BroadcastHashjoin LeftOuter BuildRight (12)
                  :- * Project (7)
                  :  +- * Filter (6)
                  :     +- * ColumnarToRow (5)
                  :        +- Scan parquet (4)
                  +- BroadcastExchange (11)
                     +- * Filter (10)
                        +- * ColumnarToRow (9)
                           +- Scan parquet (8)
```

The previous results show a change in the join strategy without AQE: from the Cartesian Product join seen in the first line of the .explain() result to the Broadcast Nested Loop join seen in stage 15. To further explore the matter another test was executed, however changing the broadcast threshold, setting it at 3 * 1024 * 1024 bytes, with the following results:

```
      explain() function:
== Physical Plan ==
BroadcastNestedLoopjoin BuildLeft, Inner, ((LAVORO_A#2 = LAVORO_B#10) OR
   (AGE#4 = FILLER1#11))
:- BroadcastExchange IdentityBroadcastMode, [id=#53]
:  +- *(1) ColumnarToRow
:     +- FileScan parquet [NOME#0,COGNOME#1,LAVORO_A#2,BIRTHPLACE#3,AGE#4]
   Batched: true, DataFilters: [], Format: Parquet, Location:
   InMemoryFileIndex[file:/C:/Users/pipitaan/IdeaProjects/Sparkaqe
/Spark-aqe-main/src/main/resources..., PartitionFilters: [], PushedFilters:
   [], ReadSchema:
   struct<NOME:string,COGNOME:string,LAVORO_A:string,BIRTHPLACE:string,
   AGE:double>
+- Exchange RoundRobinPartitioning(200), REPARTITION_WITH_NUM, [id=#65]
   +- *(3) BroadcastHashjoin [LAVORO_B#10], [LAVORO_KEYS#18], LeftOuter,
      BuildRight, false
      :- *(3) Filter ((((((FILLER1#11 = FILLER3#13) AND isnotnull(FILLER1#11))
         AND isnotnull(FILLER2#12)) AND isnotnull(FILLER3#13)) AND (FILLER1#11
         = FILLER2#12)) AND (FILLER2#12 = FILLER3#13))
      :  +- *(3) ColumnarToRow
      :     +- FileScan parquet [LAVORO_B#10,FILLER1#11,FILLER2#12,FILLER3#13]
         Batched: true, DataFilters: [(FILLER1#11 = FILLER3#13),
         isnotnull(FILLER1#11), isnotnull(FILLER2#12), isnotnull(FILLER3#13),
         ..., Format: Parquet, Location:
         InMemoryFileIndex[file:/C:/Users/pipitaan/IdeaProjects/Sparkaqe
      /Spark-aqe-main/src/main/resources..., PartitionFilters: [],
         PushedFilters: [IsNotNull(FILLER1), IsNotNull(FILLER2),
         IsNotNull(FILLER3)], ReadSchema:
         struct<LAVORO_B:string,FILLER1:double,FILLER2:double,FILLER3:double>
      +- BroadcastExchange HashedRelationBroadcastMode(List(input[0, string,
```

```
        false]),false), [id=#61]
        +- *(2) Filter isnotnull(LAVORO_KEYS#18)
           +- *(2) ColumnarToRow
              +- FileScan parquet [LAVORO_KEYS#18] Batched: true, DataFilters:
                  [isnotnull(LAVORO_KEYS#18)], Format: Parquet, Location:
                  InMemoryFileIndex[file:/C:/Users/pipitaan/IdeaProjects/Sparkaqe
               /Spark-aqe-main/src/main/resources..., PartitionFilters: [],
                  PushedFilters: [IsNotNull(LAVORO_KEYS)], ReadSchema:
                  struct<LAVORO_KEYS:string>


       SparkUI:
== Physical Plan ==
* HashAggregate (19)
+- Exchange (18)
   +- * HashAggregate (17)
      +- * Project (16)
         +- BroadcastNestedLoopjoin Inner BuildLeft (15)
            :- BroadcastExchange (3)
            :  +- * ColumnarToRow (2)
            :     +- Scan parquet (1)
            +- Exchange (14)
               +- * Project (13)
                  +- * BroadcastHashjoin LeftOuter BuildRight (12)
                     :- * Project (7)
                     :  +- * Filter (6)
                     :     +- * ColumnarToRow (5)
                     :        +- Scan parquet (4)
                     +- BroadcastExchange (11)
                        +- * Filter (10)
                           +- * ColumnarToRow (9)
                              +- Scan parquet (8)
```

In this situation both of the physical plans show the same result (first line of .explain() function result, step 15 of the physical plan), being that the chosen strategy is a Broadcast Nested Loop join. Thus, it is logical to conclude that the broadcast threshold has a direct effect on the choice between Broadcast Nested Loop and Cartesian Product join strategies. This, however, is not quoted in the source code comments [7]. As a further stage in the testing, the broadcast threshold was lowered to 256 * 1024 bytes, obtaining the following results:

```
       explain() function:
== Physical Plan ==
CartesianProduct (((LAVORO_A#2 = LAVORO_B#10) OR (AGE#4 = FILLER1#11)) OR
   (AGE#4 = FILLER2#12))
:- *(1) ColumnarToRow
:  +- FileScan parquet [NOME#0,COGNOME#1,LAVORO_A#2,BIRTHPLACE#3,AGE#4]
```

```
    Batched: true, DataFilters: [], Format: Parquet, Location:
    InMemoryFileIndex[file:/C:/Users/pipitaan/IdeaProjects/Sparkaqe/
    Spark-aqe-main/src
    /main/resources..., PartitionFilters: [], PushedFilters: [], ReadSchema:
       struct<NOME:string,COGNOME:string,LAVORO_A:string,BIRTHPLACE:string,
       AGE:double>
+- Exchange RoundRobinPartitioning(200), REPARTITION_WITH_NUM, [id=#61]
   +- *(3) BroadcastHashjoin [LAVORO_B#10], [LAVORO_KEYS#18], LeftOuter,
      BuildRight, false
      :- *(3) Filter ((((((FILLER1#11 = FILLER3#13) AND isnotnull(FILLER1#11))
         AND isnotnull(FILLER2#12)) AND isnotnull(FILLER3#13)) AND (FILLER1#11
         = FILLER2#12)) AND (FILLER2#12 = FILLER3#13))
      :  +- *(3) ColumnarToRow
      :     +- FileScan parquet [LAVORO_B#10,FILLER1#11,FILLER2#12,FILLER3#13]
         Batched: true, DataFilters: [(FILLER1#11 = FILLER3#13),
         isnotnull(FILLER1#11), isnotnull(FILLER2#12), isnotnull(FILLER3#13),
         ..., Format: Parquet, Location:
         InMemoryFileIndex[file:/C:/Users/pipitaan/IdeaProjects/Sparkaqe
         /Spark-aqe-main
       /src/main/resources..., PartitionFilters: [], PushedFilters:
          [IsNotNull(FILLER1), IsNotNull(FILLER2), IsNotNull(FILLER3)],
          ReadSchema:
          struct<LAVORO_B:string,FILLER1:double,FILLER2:double,FILLER3:double>
      +- BroadcastExchange HashedRelationBroadcastMode(List(input[0, string,
         false]),false), [id=#57]
         +- *(2) Filter isnotnull(LAVORO_KEYS#18)
            +- *(2) ColumnarToRow
               +- FileScan parquet [LAVORO_KEYS#18] Batched: true, DataFilters:
                   [isnotnull(LAVORO_KEYS#18)], Format: Parquet, Location:
                   InMemoryFileIndex[file:/C:/Users/pipitaan/IdeaProjects/Sparkaqe
                 /Spark-aqe-main/src/main/resources..., PartitionFilters: [],
                    PushedFilters: [IsNotNull(LAVORO_KEYS)], ReadSchema:
                    struct<LAVORO_KEYS:string>

       SparkUI:
== Physical Plan ==
* HashAggregate (18)
+- Exchange (17)
   +- * HashAggregate (16)
      +- * Project (15)
         +- CartesianProduct Inner (14)
            :- * ColumnarToRow (2)
            :  +- Scan parquet (1)
            +- Exchange (13)
               +- * Project (12)
                  +- * BroadcastHashjoin LeftOuter BuildRight (11)
                     :- * Project (6)
                     :  +- * Filter (5)
                     :     +- * ColumnarToRow (4)
```

```
:       +- Scan parquet (3)
+- BroadcastExchange (10)
   +- * Filter (9)
      +- * ColumnarToRow (8)
         +- Scan parquet (7)
```

As shown by the previous results, lowering the broadcast threshold influences the join strategy chosen (first line of the result returned by .explain(), step 14 in the physical plan). This proves that the broadcast threshold has a direct effect on the choice between Broadcast Nested Loop and Cartesian Product join strategies. With a high enough broadcast threshold the Broadcast Nested Loop will be chosen, with a low broadcast threshold the Cartesian Product will be preferred, while with the broadcast threshold within a certain range the join strategy will be switched from Cartesian Product to Broadcast Nested Loop. To check whether AQE could have any effects on the join strategy, the last experiment was repeated with AQE enabled, yielding the following results

```
       explain() function:
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- CartesianProduct ((LAVORO_A#2 = LAVORO_B#10) OR (AGE#4 = FILLER1#11))
   :- FileScan parquet [NOME#0,COGNOME#1,LAVORO_A#2,BIRTHPLACE#3,AGE#4]
      Batched: true, DataFilters: [], Format: Parquet, Location:
      InMemoryFileIndex[file:/C:/Users/pipitaan/IdeaProjects/Sparkaqe
      /Spark-aqe-main
    /src/main/resources..., PartitionFilters: [], PushedFilters: [],
       ReadSchema:
       struct<NOME:string,COGNOME:string,LAVORO_A:string,BIRTHPLACE:string,
       AGE:double>
   +- Exchange RoundRobinPartitioning(200), REPARTITION_WITH_NUM, [id=#37]
      +- BroadcastHashjoin [LAVORO_B#10], [LAVORO_KEYS#18], LeftOuter,
         BuildRight, false
         :- Filter ((((((FILLER1#11 = FILLER3#13) AND isnotnull(FILLER1#11))
            AND isnotnull(FILLER2#12)) AND isnotnull(FILLER3#13)) AND
            (FILLER1#11 = FILLER2#12)) AND (FILLER2#12 = FILLER3#13))
         :  +- FileScan parquet [LAVORO_B#10,FILLER1#11,FILLER2#12,FILLER3#13]
            Batched: true, DataFilters: [(FILLER1#11 = FILLER3#13),
            isnotnull(FILLER1#11), isnotnull(FILLER2#12),
            isnotnull(FILLER3#13), ..., Format: Parquet, Location:
            InMemoryFileIndex[file:/C:/Users/pipitaan/IdeaProjects/Sparkaqe
           /Spark-aqe-main/src/main/resources..., PartitionFilters: [],
               PushedFilters: [IsNotNull(FILLER1), IsNotNull(FILLER2),
               IsNotNull(FILLER3)], ReadSchema:
               struct<LAVORO_B:string,FILLER1:double,FILLER2:double,
               FILLER3:double>
         +- BroadcastExchange HashedRelationBroadcastMode(List(input[0, string,
            false]),false), [id=#35]
            +- Filter isnotnull(LAVORO_KEYS#18)
```

```
          +- FileScan parquet [LAVORO_KEYS#18] Batched: true, DataFilters:
             [isnotnull(LAVORO_KEYS#18)], Format: Parquet, Location:
             InMemoryFileIndex[file:/C:/Users/pipitaan/IdeaProjects/Sparkaqe
          /Spark-aqe-main/src/main/resources..., PartitionFilters: [],
             PushedFilters: [IsNotNull(LAVORO_KEYS)], ReadSchema:
             struct<LAVORO_KEYS:string>

     Initial Plan:
+- == Initial Plan ==
  HashAggregate (unknown)
  +- Exchange (unknown)
    +- HashAggregate (unknown)
      +- Project (unknown)
        +- CartesianProduct Inner (unknown)
          :- Scan parquet (unknown)
          +- Exchange (unknown)
            +- Project (unknown)
              +- BroadcastHashjoin LeftOuter BuildRight (unknown)
                :- Project (unknown)
                :  +- Filter (unknown)
                :    +- Scan parquet (3)
                +- BroadcastExchange (unknown)
                  +- Filter (unknown)
                    +- Scan parquet (7)

     Final Plan:
== Physical Plan ==
AdaptiveSparkPlan (24)
+- == Final Plan ==
  * HashAggregate (23)
  +- ShuffleQueryStage (22)
    +- Exchange (21)
      +- * HashAggregate (20)
        +- * Project (19)
          +- BroadcastNestedLoopjoin Inner BuildRight (18)
            :- * ColumnarToRow (2)
            :  +- Scan parquet (1)
            +- BroadcastQueryStage (17)
              +- BroadcastExchange (16)
                +- ShuffleQueryStage (15)
                  +- Exchange (14)
                    +- * Project (13)
                      +- * BroadcastHashjoin LeftOuter BuildRight (12)
                        :- * Project (6)
                        :  +- * Filter (5)
                        :    +- * ColumnarToRow (4)
                        :      +- Scan parquet (3)
                        +- BroadcastQueryStage (11)
                          +- BroadcastExchange (10)
```

```
+- * Filter (9)
   +- * ColumnarToRow (8)
      +- Scan parquet (7)
```

As shown by the results of the last two experiments, it is possible to highlight that the usage of AQE can prompt a switch of join strategy at runtime from Cartesian Product to Broadcast Nested Loop join (first line of the plan returned by .explain(), stage 18 in the final physical plan). This is possible if the dimensions of the tables change during execution: AQE re-engages the Catalyst for the join selection during shuffle phases [6] and that the broadcast threshold influences the choice between Broadcast Nested Loop and Cartesian Product. This possibility is not reported in the Spark documentation. Thus, summarizing the results obtained in the experiments on the Cartesian Product join Strategy:

- The broadcast threshold influences the choice between a Cartesian Product and a Broadcast Nested Loop: while this is not advertised in Spark code [7], it is shown by the difference between the results returned by the .explain() function concerning the broadcast threshold.

- While the .explain() function returns a plan involving a Cartesian Product, the plan returned by the SparkUI reports a Broadcast Nested Loop join with AQE disabled: this is so because of the broadcast threshold settings. This is also not advertised in the Spark code [7].

- AQE can prompt a dynamic join strategy switch from Cartesian Product to Broadcast Nested Loop join, despite not being quoted as an AQE capability in the documentation [20].

Broadcast Nested Loop Join

According to documentation, the ability of AQE to change the join type dynamically is limited to the Sort Merge join strategy. These experiments are aimed to check whether AQE could have any side effect on Broadcast Nested Loop join strategies.

The settings used were the following:

```
val threshold= 1 * 1024 * 1024

join_conditions= dfA("LAVORO_A")===dfAux("LAVORO_B") or
dfA("AGE")===dfAux("FILLER1")

join_type=full
```

With AQE disabled a Broadcast Nested Loop join is performed. This is so by design: the composite join key prohibits the usage of Sort Merge join and the low threshold prohibits

the usage of Broadcast Hash join. Setting the join type as full also prohibits the usage of a Cartesian Product, thus leaving the Broadcast Nested Loop join as the only possibility. With AQE enabled it can be observed that the join strategy is not changed at runtime (Appendix B, Section 8.1.2) We can thus conclude that, in regards to the Broadcast Nested Loop join strategy, AQE behaves as described in the documentation, not affecting Broadcast Nested Loop joins [20].

Conclusions

In this Section, the experiments explored the interaction between AQE and the various kinds of joins that Spark can perform. The results highlight that AQE strays from the behavior described in the documentation. It can dynamically switch from a Sort Merge join to a Broadcast Hash join only if there is a shuffle stage between the filtering action and the subsequent Sort Merge join (Section 3.1.2), while the documentation suggests otherwise [20]. Furthermore, AQE can dynamically change the join strategy from Cartesian Product to Broadcast Nested Loop (Section 3.1.1), when the table dimensions fall below the broadcast threshold during execution. This possibility is not quoted in the documentation or the Spark code comments [20] [7]. These experiments also outlined a dependence between the join strategy choosing process between Broadcast Nested Loop and Cartesian Product: as seen in Section 3.1.1 if the broadcast threshold is high enough the Broadcast Nested Loop is chosen. This influence of the broadcast threshold is not quoted in the documentation nor the source code [20] [7].

3.1.2. With hints

Given how the usage of hints alters how the Catalyst chooses the join strategy [7], this part will be focused on the interaction between hints and the dynamic join selection performed by AQE, in particular, to check whether the usage of hints could be a substitute for the usage of AQE. The workload used in these experiments is as follows:

```
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 =join_expression
dfResult.count()
```

The specific join expression used and possible alterations to the Spark configuration will be specified in relation to the single experiment.

Broadcast Hash Join

In this Section, the experiments explore the usage of both AQE and hints in a situation in which the Catalyst statically chooses the Broadcast Hash join as join strategy.

For the reasons already explained in the Broadcast Hash join experiments, the experiments in this Section use a different workload:

```
val dfA=spark.read.parquet("Spark-aqe-main/src/main/resources/bigTableA")
val dfK=spark.read.parquet("Spark-aqe-main/src/main/resources/keys")
                .repartition(200)
val dfResult_ =join_expression
```

Control The first experiment conducted acts as a control, checking whether the usage of a broadcast hint could interfere with the action of AQE in a situation in which the Catalyst statically chooses a Broadcast Hash join strategy.

The settings used in this experiment were as follows:

```
threshold = 3 * 1024 * 1024

join_expression=dfA.join(dfK.hint("BROADCAST"),
                       dfA("LAVORO_A")===dfK("LAVORO_KEYS")
```

The experiment was executed with AQE enabled.

As shown by the plans returned by the SparkUI (Appendix B, Section 8.1.3), the presence of a broadcast hint does not alter the execution of the workload. This was to be expected, in the first place because of AQE's documentation [20], because of the results in the experiment 3.1.1 and because of the nature of hints in Spark [21].

Broadcast hint with AQE off This experiment is aimed at checking whether the execution changes in any measurable way when AQE is not active but a broadcast hint is present. The configurations used are the same as in the previous experiment:

```
threshold = 3 * 1024 * 1024

join_expression=dfA.join(dfK.hint("BROADCAST"),
                       dfA("LAVORO_A")===dfAux("LAVORO_KEYS")
```

The workload was executed with AQE disabled.

The SparkUI shows (Appendix B, Section, 8.1.4), as it was reasonable to expect, that the results were the same as in the last experiment. Once again, given AQE's specifications [20], the results obtained in 3.1.1 and the information provided in Section [21], the results were expected.

Sort Merge Join

In this Section, the experiments are aimed to study the interaction between hints and AQE in a situation in which the Catalyst would statically choose a Sort Merge join strategy while the action of AQE would dynamically change it into a Broadcast Hash join.

AQE enabled, Merge hint These experiments are aimed at checking the behavior displayed by AQE in presence of a hint for a worse performing join strategy. In particular, merge hints were introduced in a situation in which a Broadcast Hash join strategy would have been chosen by AQE at runtime. The workload was the same one displayed for the experiment 3.1.1, adding the merge hints first on the smaller table, then on the bigger one:

1. Hint on smaller table:

```
threshold = 1024*1024*1

join_expression=dfA.join(dfAux.hint("merge"),
                         dfA("LAVORO_A")===dfAux("LAVORO_B")
```

2. Hint on bigger table:

```
threshold = 1024*1024*1

join_expression=dfA.hint("merge").join(dfAux,
                                dfA("LAVORO_A")===dfAux("LAVORO_B")
```

As The SparkUI shows (Appendix B, Section 8.1.5), in both cases the result was the same: the join strategy chosen by AQE was the Sort Merge join.

This has to be expected because of the workings of both AQE and the Catalyst [7]. During the shuffle preceding the join of interest the Catalyst is re-engaged and the best join strategy is re-evaluated: in this phase, the Catalyst first checks for hints [7] and, where possible, chooses the join strategy hinted.

It is thus possible for users to override the decisions that AQE would otherwise take: as a consequence the user must be aware of this when using both hints and AQE in order not to hinder AQE capabilities.

AQE disabled, Broadcast hint These experiments are aimed at checking whether the usage of hints could have the same effects as AQE without the overhead stemming from its activation at each shuffle phase.

The workload used was the same used in Section 3.1.1, however adding the `BROADCAST` hint first on the smaller table and then on the bigger one:

1. Hint on smaller table:

```
threshold = 1024*1024*1

join_expression=dfA.join(dfAux.hint("BROADCAST"),
                    dfA("LAVORO_A")===dfAux("LAVORO_B")
```

Physical plans:

```
== Physical Plan ==
* HashAggregate (19)
+- Exchange (18)
   +- * HashAggregate (17)
      +- * Project (16)
         +- * BroadcastHashjoin LeftOuter BuildRight (15)
            :- * ColumnarToRow (2)
            :  +- Scan parquet (1)
            +- BroadcastExchange (14)
               +- Exchange (13)
                  +- * Project (12)
                     +- * BroadcastHashjoin LeftOuter BuildRight (11)
                        :- * Project (6)
                        :  +- * Filter (5)
                        :     +- * ColumnarToRow (4)
                        :        +- Scan parquet (3)
                        +- BroadcastExchange (10)
                           +- * Filter (9)
                              +- * ColumnarToRow (8)
                                 +- Scan parquet (7)
```

2. Hint on bigger table:

```
threshold = 1024*1024*1

join_expression=dfA.hint("BROADCAST").join(dfAux,
                                    dfA("LAVORO_A")===dfAux("LAVORO_B")
```

Physical plan:

```
== Physical Plan ==
* HashAggregate (22)
+- Exchange (21)
   +- * HashAggregate (20)
      +- * Project (19)
         +- SortMergejoin LeftOuter (18)
            :- * Sort (4)
            :  +- Exchange (3)
```

```
:     +- * ColumnarToRow (2)
:        +- Scan parquet (1)
+- * Sort (17)
   +- Exchange (16)
      +- Exchange (15)
         +- * Project (14)
            +- * BroadcastHashjoin LeftOuter BuildRight (13)
               :- * Project (8)
               :  +- * Filter (7)
               :     +- * ColumnarToRow (6)
               :        +- Scan parquet (5)
               +- BroadcastExchange (12)
                  +- * Filter (11)
                     +- * ColumnarToRow (10)
                        +- Scan parquet (9)
```

As the SparkUI shows, if the hint is placed on the bigger table the strategy chosen is a Sort Merge join: the Catalyst evaluates the hint [7], considers it unfeasible and proceeds to choose a Sort Merge join Strategy.

If the hint is instead placed on the smaller table the chosen strategy is a Broadcast Hash join, as it can be observed in the SparkUI. This is unexpected, since the smaller table does not satisfy the threshold set for Spark [14], and shows a behaviour different from the one outlined in Spark's code comments [7]. Further analyzing this behavior, testing for the other combinations of join types and hints placement it is possible to define the following outlines on whether Spark performs a Broadcast Hash join in presence of broadcast hints, reported in Table 3.1. As shown in Table 3.1, the broadcast threshold is taken into account only when it is greater than 0 and the broadcast hint is placed on the outer-side table. Therefore, while the usage of a broadcast hint eliminates AQE's evaluation overhead costs, it also introduces a considerable risk. In most situations it does not take into account the broadcast threshold that has been set, even completely disregarding it when it was set to -1 to disable the broadcast join. This could lead to the driver crashing if the dimensions of the broadcasted tables exceed the driver's memory. As such, the usage of broadcast hints in place of AQE has to be avoided.

.

Cartesian Product

In this Section, the experiments are aimed to study the interaction between hints and AQE in a situation in which the Catalyst would statically choose a Cartesian Product strategy while the action of AQE would either leave it unchanged or dynamically change it into a Broadcast Nested Loop 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

Table 3.1: Effects of broadcast hints on a Sort Merge join

AQE disabled, Broadcast hint Given the results shown in Section 3.1.1, these experiments are aimed to give a baseline for the behavior of the Catalyst when broadcast hints are to be applied to Cartesian Products.The configurations are as follows:

1. Hint on smaller table:

```
threshold = 256 * 1024

join_expression=dfA.join(dfAux.hint("BROADCAST"),
                              dfA("LAVORO_A")===dfAux("LAVORO_B") ||
                              dfA("AGE")===dfAux("FILLER1"), "inner")
```

Physical plan:

```
== Physical Plan ==
* HashAggregate (19)
+- Exchange (18)
   +- * HashAggregate (17)
      +- * Project (16)
         +- BroadcastNestedLoopjoin Inner BuildRight (15)
            :- * ColumnarToRow (2)
            :  +- Scan parquet (1)
            +- BroadcastExchange (14)
               +- Exchange (13)
```

```
+- * Project (12)
   +- * BroadcastHashjoin LeftOuter BuildRight (11)
      :- * Project (6)
      :  +- * Filter (5)
      :     +- * ColumnarToRow (4)
      :        +- Scan parquet (3)
      +- BroadcastExchange (10)
         +- * Filter (9)
            +- * ColumnarToRow (8)
               +- Scan parquet (7)
```

2. Hint on bigger table:

```
threshold = 1 * 1024 * 1024

join_expression=dfA.hint("BROADCAST").join(dfAux,
                                  dfA("LAVORO_A")===dfAux("LAVORO_B") ||
                                  dfA("AGE")===dfAux("FILLER1"), "inner")
```

Physical plan:

```
== Physical Plan ==
* HashAggregate (19)
+- Exchange (18)
   +- * HashAggregate (17)
      +- * Project (16)
         +- BroadcastNestedLoopjoin Inner BuildLeft (15)
            :- BroadcastExchange (3)
            :  +- * ColumnarToRow (2)
            :     +- Scan parquet (1)
            +- Exchange (14)
               +- * Project (13)
                  +- * BroadcastHashjoin LeftOuter BuildRight (12)
                     :- * Project (7)
                     :  +- * Filter (6)
                     :     +- * ColumnarToRow (5)
                     :        +- Scan parquet (4)
                     +- BroadcastExchange (11)
                        +- * Filter (10)
                           +- * ColumnarToRow (9)
                              +- Scan parquet (8)
```

In both cases the join strategy selected by the Catalyst in a Broadcast Nested Loop join (stage 15). It is thus possible to conclude that, in this instance, the usage of hints has the same effects that AQE would have (Section 3.1.1).

AQE enabled, Broadcast hint Given the results shown in Section 3.1.1, these experiments are aimed to further explore AQE interaction with Cartesian Products in presence of hints. The workload used is the same one used in Section 3.1.1 experiment with the added `BROADCAST` hint:

1. Hint on smaller table:

```
threshold = 256 * 1024

join_expression=dfA.join(dfAux.hint("BROADCAST"),
                            dfA("LAVORO_A")===dfAux("LAVORO_B") ||
                            dfA("AGE")===dfAux("FILLER1"), "inner")
```

Initial physical plan:

```
+- == Initial Plan ==
   HashAggregate (unknown)
   +- Exchange (unknown)
      +- HashAggregate (unknown)
         +- Project (unknown)
            +- BroadcastNestedLoopjoin Inner BuildRight (unknown)
               :- Scan parquet (1)
               +- BroadcastExchange (unknown)
                  +- Exchange (unknown)
                     +- Project (unknown)
                        +- BroadcastHashjoin LeftOuter BuildRight (unknown)
                           :- Project (unknown)
                           :  +- Filter (unknown)
                           :     +- Scan parquet (3)
                           +- BroadcastExchange (unknown)
                              +- Filter (unknown)
                                 +- Scan parquet (7)
```

Final Physical plan:

```
== Physical Plan ==
AdaptiveSparkPlan (24)
+- == Final Plan ==
   * HashAggregate (23)
   +- ShuffleQueryStage (22)
      +- Exchange (21)
         +- * HashAggregate (20)
            +- * Project (19)
               +- BroadcastNestedLoopjoin Inner BuildRight (18)
                  :- * ColumnarToRow (2)
                  :  +- Scan parquet (1)
                  +- BroadcastQueryStage (17)
```

```
                  +- BroadcastExchange (16)
                     +- ShuffleQueryStage (15)
                        +- Exchange (14)
                           +- * Project (13)
                              +- * BroadcastHashjoin LeftOuter BuildRight
                                 (12)
                                 :- * Project (6)
                                 :  +- * Filter (5)
                                 :     +- * ColumnarToRow (4)
                                 :        +- Scan parquet (3)
                                 +- BroadcastQueryStage (11)
                                    +- BroadcastExchange (10)
                                       +- * Filter (9)
                                          +- * ColumnarToRow (8)
                                             +- Scan parquet (
```

2. Hint on bigger table:

```
threshold = 1024*256

join_expression=dfA.hint("BROADCAST").join(dfAux,
                               dfA("LAVORO_A")===dfAux("LAVORO_B") ||
                               dfA("AGE")===dfAux("FILLER1"), "inner")
```

Initial Physical plan:

```
+- == Initial Plan ==
   HashAggregate (unknown)
   +- Exchange (unknown)
      +- HashAggregate (unknown)
         +- Project (unknown)
            +- BroadcastNestedLoopjoin Inner BuildLeft (unknown)
               :- BroadcastExchange (unknown)
               :  +- Scan parquet (1)
               +- Exchange (unknown)
                  +- Project (unknown)
                     +- BroadcastHashjoin LeftOuter BuildRight (unknown)
                        :- Project (unknown)
                        :  +- Filter (unknown)
                        :     +- Scan parquet (5)
                        +- BroadcastExchange (unknown)
                           +- Filter (unknown)
                              +- Scan parquet (9)
```

Final Physical plan:

```
== Physical Plan ==
```

```
AdaptiveSparkPlan (24)
+- == Final Plan ==
   * HashAggregate (23)
   +- ShuffleQueryStage (22)
      +- Exchange (21)
         +- * HashAggregate (20)
            +- * Project (19)
               +- BroadcastNestedLoopjoin Inner BuildLeft (18)
                  :- BroadcastQueryStage (4)
                  :  +- BroadcastExchange (3)
                  :     +- * ColumnarToRow (2)
                  :        +- Scan parquet (1)
                  +- ShuffleQueryStage (17)
                     +- Exchange (16)
                        +- * Project (15)
                           +- * BroadcastHashjoin LeftOuter BuildRight (14)
                              :- * Project (8)
                              :  +- * Filter (7)
                              :     +- * ColumnarToRow (6)
                              :        +- Scan parquet (5)
                              +- BroadcastQueryStage (13)
                                 +- BroadcastExchange (12)
                                    +- * Filter (11)
                                       +- * ColumnarToRow (10)
                                          +- Scan parquet (9)
```

As shown by the SparkUI, in both cases the join strategy selected by the Catalyst in a Broadcast Nested Loop join (step 18), as could be expected given the results obtained in the previous experiment 3.1.1 and given the documented behavior of Spark regarding hints [7] [21].

AQE enabled, Cartesian hint Given the behavior shown in previous experiments, these aim to study the interaction between AQE and hints when a Cartesian Product would be chosen by the Catalyst. In particular, given the observed transformations into Broadcast Nested Loop. The workload used is the same one used in Section 3.1.1 with the added `SHUFFLE_REPLICATE_NL` hint, which acts as cartesian hint:

1. Hint on smaller table:

```
threshold = 1024*256

join_expression=dfA.join(dfAux.hint("SHUFFLE_REPLICATE_NL"),
                 dfA("LAVORO_A")===dfAux("LAVORO_B") ||
                 dfA("AGE")===dfAux("FILLER1"), "inner")
```

Physical plans: Initial Plan:

```
+- == Initial Plan ==
   HashAggregate (unknown)
   +- Exchange (unknown)
      +- HashAggregate (unknown)
         +- Project (unknown)
            +- CartesianProduct Inner (unknown)
               :- Scan parquet (1)
               +- Exchange (unknown)
                  +- Project (unknown)
                     +- BroadcastHashjoin LeftOuter BuildRight (unknown)
                        :- Project (unknown)
                        :  +- Filter (unknown)
                        :     +- Scan parquet (3)
                        +- BroadcastExchange (unknown)
                           +- Filter (unknown)
                              +- Scan parquet (7)
```

Final Plan:

```
== Physical Plan ==
AdaptiveSparkPlan (22)
+- == Final Plan ==
   * HashAggregate (21)
   +- ShuffleQueryStage (20)
      +- Exchange (19)
         +- * HashAggregate (18)
            +- * Project (17)
               +- CartesianProduct Inner (16)
                  :- * ColumnarToRow (2)
                  :  +- Scan parquet (1)
                  +- ShuffleQueryStage (15)
                     +- Exchange (14)
                        +- * Project (13)
                           +- * BroadcastHashjoin LeftOuter BuildRight (12)
                              :- * Project (6)
                              :  +- * Filter (5)
                              :     +- * ColumnarToRow (4)
                              :        +- Scan parquet (3)
                              +- BroadcastQueryStage (11)
                                 +- BroadcastExchange (10)
                                    +- * Filter (9)
                                       +- * ColumnarToRow (8)
                                          +- Scan parquet (7)
```

2. Hint on bigger table:

```
  threshold = 1024*256
```

```
join_expression=dfA.hint("SHUFFLE_REPLICATE_NL").join(dfAux,
                 dfA("LAVORO_A")===dfAux("LAVORO_B") ||
                 dfA("AGE")===dfAux("FILLER1"), "inner")
```

Physical plans:

```
+- == Initial Plan ==
   HashAggregate (unknown)
   +- Exchange (unknown)
      +- HashAggregate (unknown)
         +- Project (unknown)
            +- CartesianProduct Inner (unknown)
               :- Scan parquet (1)
               +- Exchange (unknown)
                  +- Project (unknown)
                     +- BroadcastHashjoin LeftOuter BuildRight (unknown)
                        :- Project (unknown)
                        :  +- Filter (unknown)
                        :     +- Scan parquet (3)
                        +- BroadcastExchange (unknown)
                           +- Filter (unknown)
                              +- Scan parquet (7)
```

Final Plan:

```
== Physical Plan ==
AdaptiveSparkPlan (22)
+- == Final Plan ==
   * HashAggregate (21)
   +- ShuffleQueryStage (20)
      +- Exchange (19)
         +- * HashAggregate (18)
            +- * Project (17)
               +- CartesianProduct Inner (16)
                  :- * ColumnarToRow (2)
                  :  +- Scan parquet (1)
                  +- ShuffleQueryStage (15)
                     +- Exchange (14)
                        +- * Project (13)
                           +- * BroadcastHashjoin LeftOuter BuildRight (12)
                              :- * Project (6)
                              :  +- * Filter (5)
                              :     +- * ColumnarToRow (4)
                              :        +- Scan parquet (3)
                              +- BroadcastQueryStage (11)
```

```
                  +- BroadcastExchange (10)
                     +- * Filter (9)
                        +- * ColumnarToRow (8)
                           +- Scan parquet (7)
```

As shown by the SparkUI physical plans, AQE's action is contrasted by the presence of the cartesian hint. In the absence of this hint, as shown in Section 3.1.1, in this situation AQE would have chosen a Broadcast Nested Loop join. The presence of the hint forces the Cartesian Product selection.

Broadcast Hint and Broadcast Threshold without AQE As it has already been done for the Sort Merge join strategy (Section 3.1.2), the workload used in this Section was executed with broadcast hints and with the broadcast threshold set to -1. Experimental results (Appendix B, Section 8.1.6) confirm that the broadcast threshold is not taken into account when selecting the join strategy in presence of broadcast hint. As already stated concerning the Broadcast Hash join (Section 3.1.2), this can lead to otherwise avoidable failures.

Conclusions

The results of these experiments highlight the following insights:

1. The usage of hints can override AQE's optimizations: as shown in Section 3.1.2 and in 3.1.2 an erroneous usage of hints can override the decisions taken by AQE.

2. The usage of the broadcast hint can substitute AQE: as shown in Sections 3.1.2 and 3.1.2, in particular as exposed in Table 3.1, the usage of a broadcast hint can have the same effects as the usage of AQE, however at the risk of having the driver crashing. As such, the usage of broadcast hints in place of AQE is better to be avoided.

3. While the join strategy selection algorithm presented in [7] proves to be accurate in most situations, there are some situations in which Spark follows a different behavior. As seen in 3.1.2 and 3.1.2, the usage of broadcast hints overrides the definition of the broadcast threshold(Table 3.1), which could lead the driver to crash.

3.2. Automatic Post Shuffle Coalesce of partitions

In this Section the experiments explore AQE's ability of automatically recognise, during shuffle phases [6] [4], situations in which coalescing the dataframe partitions could benefit performances and subsequently perform the coalesces themselves [20]. As already done in the previous experiments, AQE's component of interest has been isolated by disabling skew join handling via Spark's configurations and avoiding by design and by broadcast threshold tuning dynamic join strategy changes. Spark was configured as follows:

- With AQE:

```
val spark= SparkSession
.builder()
.appName("CoalesceAQEOn")
.master("local[*]")
.config("spark.sql.adaptive.enabled","true")
.config("spark.sql.adaptive.coalescePartitions.enabled","true")
.config( "spark.sql.adaptive.skewjoin.enabled","false")
.config("spark.sql.autoBroadcastjoinThreshold", threshold)
.config("Spark.eventLog.enabled", "true")
.config("Spark.eventLog.dir","Spark-aqe-main/src/main/resources
    /historyServer/logs")
.getOrCreate()
```

- Without AQE:

```
val spark= SparkSession
.builder()
.appName("SortMergejoinAQEOff")
.master("local[*]")
.config("spark.sql.adaptive.enabled","false")
.config("Spark.eventLog.enabled", "true")
.config("Spark.eventLog.dir","Spark-aqe-main/src/main/resources
    /historyServer/logs")
.config("spark.sql.autoBroadcastjoinThreshold", threshold)
.getOrCreate()
```

The experiments on the automatic coalesce of partitions use tables A and B as data sources for dfA and dfB

3.2.1. Coalesce

In this part the experiments are aimed at checking whether AQE behaves accordingly with the documentation, automatically performing coalesces where needed [20]. All of the experiments in this Section share the same workload:

```
val dfA=spark.read.parquet("Spark-aqe-main/src/main/resources/bigTableA")
  .sample(false, 0.0001)
  .repartition(200)
val dfB =spark.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()
```

Further modifications will be specified for each experiment.

Broadcast Hash Join

The Broadcast Hash join is considered separately from the Sort Merge join and is used as a representative of the joins other than Sort Merge join because of the absence of a shuffle phase, characteristic of the Sort Merge join [7]. Given the absence of a shuffle phase, AQE will probably not be engaged [6]. In both cases, the broadcast threshold is high enough to guarantee that the Broadcast Hash join strategy is picked by the Catalyst.

Settings:

```
threshold= 1 * 1024 * 1024
```

In both cases, as shown by the SparkUI, the coalesce is not performed. While this has to be expected when AQE is disabled, when AQE is enabled the absence of coalescing is due to the absence of a shuffle phase in which AQE could identify the necessity of a repartition:

1. With AQE:

   - Initial plan:

```
+- == Initial Plan ==
   HashAggregate (unknown)
   +- Exchange (unknown)
      +- HashAggregate (unknown)
         +- Project (unknown)
            +- BroadcastHashjoin LeftOuter BuildRight (unknown)
               :- Sample (unknown)
               :  +- Exchange (unknown)
               :     +- Scan parquet (1)
               +- BroadcastExchange (unknown)
                  +- Exchange (unknown)
                     +- Filter (unknown)
                        +- Sample (unknown)
                           +- Scan parquet (6)
```

   - Final Plan:

```
== Physical Plan ==
AdaptiveSparkPlan (20)
+- == Final Plan ==
   * HashAggregate (19)
   +- ShuffleQueryStage (18)
      +- Exchange (17)
         +- * HashAggregate (16)
```

```
+- * Project (15)
   +- * BroadcastHashjoin LeftOuter BuildRight (14)
      :- * Sample (5)
      :  +- ShuffleQueryStage (4)
      :     +- Exchange (3)
      :        +- * ColumnarToRow (2)
      :           +- Scan parquet (1)
      +- BroadcastQueryStage (13)
         +- BroadcastExchange (12)
            +- ShuffleQueryStage (11)
               +- Exchange (10)
                  +- * Filter (9)
                     +- * Sample (8)
                        +- * ColumnarToRow (7)
                           +- Scan parquet (6)
```

2. Without AQE:

   Physical PLan:

```
== Physical Plan ==
* HashAggregate (15)
+- Exchange (14)
   +- * HashAggregate (13)
      +- * Project (12)
         +- * BroadcastHashjoin LeftOuter BuildRight (11)
            :- Exchange (4)
            :  +- * Sample (3)
            :     +- * ColumnarToRow (2)
            :        +- Scan parquet (1)
            +- BroadcastExchange (10)
               +- Exchange (9)
                  +- * Filter (8)
                     +- * Sample (7)
                        +- * ColumnarToRow (6)
                           +- Scan parquet (5)
```

Sort Merge Join

Given the presence of the sort stage during the execution of the Sort Merge join, AQE is expected to trigger and perform an automatic coalesce of the partitions.

Settings:

```
threshold= -1
```

The following plans and Dag were obtained from the SparkUI:

1. With AQE:

- Initial plan:

```
+- == Initial Plan ==
   HashAggregate (unknown)
   +- Exchange (unknown)
      +- HashAggregate (unknown)
         +- Project (unknown)
            +- SortMergejoin LeftOuter (unknown)
               :- Sort (unknown)
               :  +- Exchange (unknown)
               :     +- Sample (unknown)
               :        +- Exchange (unknown)
               :           +- Scan parquet (1)
               +- Sort (unknown)
                  +- Exchange (unknown)
                     +- Exchange (unknown)
                        +- Filter (unknown)
                           +- Sample (unknown)
                              +- Scan parquet (10)
```

- Final Plan:

```
== Physical Plan ==
AdaptiveSparkPlan (26)
+- == Final Plan ==
   * HashAggregate (25)
   +- ShuffleQueryStage (24)
      +- Exchange (23)
         +- * HashAggregate (22)
            +- * Project (21)
               +- SortMergejoin LeftOuter (20)
                  :- * Sort (9)
                  :  +- CustomShuffleReader (8)
                  :     +- ShuffleQueryStage (7)
                  :        +- Exchange (6)
                  :           +- * Sample (5)
                  :              +- ShuffleQueryStage (4)
                  :                 +- Exchange (3)
                  :                    +- * ColumnarToRow (2)
                  :                       +- Scan parquet (1)
                  +- * Sort (19)
                     +- CustomShuffleReader (18)
                        +- ShuffleQueryStage (17)
                           +- Exchange (16)
```

```
                              +- ShuffleQueryStage (15)
                                 +- Exchange (14)
                                    +- * Filter (13)
                                       +- * Sample (12)
                                          +- * ColumnarToRow (11)
                                             +- Scan parquet (10)


(8) CustomShuffleReader
Input [1]: [LAVORO_A#2]
Arguments: coalesced

(18) CustomShuffleReader
Input [1]: [LAVORO_B#10]
Arguments: coalesced
```

2. Without AQE:

   Physical PLan:

```
== Physical Plan ==
* HashAggregate (18)
+- Exchange (17)
   +- * HashAggregate (16)
      +- * Project (15)
         +- SortMergejoin LeftOuter (14)
            :- * Sort (6)
            :  +- Exchange (5)
            :     +- Exchange (4)
            :        +- * Sample (3)
            :           +- * ColumnarToRow (2)
            :              +- Scan parquet (1)
            +- * Sort (13)
               +- Exchange (12)
                  +- Exchange (11)
                     +- * Filter (10)
                        +- * Sample (9)
                           +- * ColumnarToRow (8)
                              +- Scan parquet (7)
```

While without AQE no coalesce was performed, with AQE enabled the shuffle phase before the sort part of the Sort Merge join strategy is replaced by a custom shuffle reader, that coalesces both dataframes into 11 partitions each, as shown in Figure 3.1.

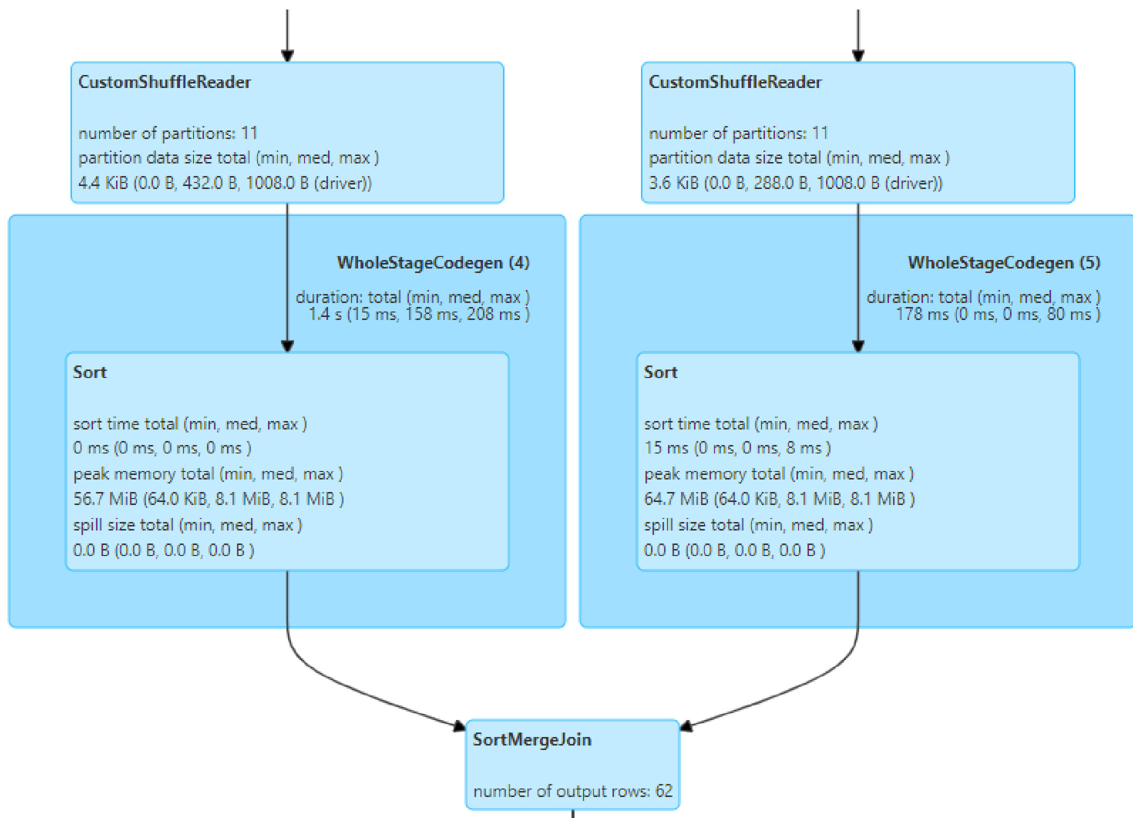

Figure 3.1: DAG illustrating a coalesce

Minimum number of partitions

This experiment aimed at checking whether the user had control over the number of partitions in which AQE coalesces the dataframes. While the workload remains unchanged, the Spark configurations have been altered as follows:

```
threshold= -1
.config("spark.sql.adaptive.coalescePartitions.minPartitionNum", 50)
```

The following plans and Dag 3.2 were obtained from the SparkUI:

- Initial physical plan:

```
     +- == Initial Plan ==
HashAggregate (unknown)
+- Exchange (unknown)
   +- HashAggregate (unknown)
      +- Project (unknown)
         +- SortMergejoin LeftOuter (unknown)
            :- Sort (unknown)
            :  +- Exchange (unknown)
            :     +- Sample (unknown)
            :        +- Exchange (unknown)
```

```
:           +- Scan parquet (1)
+- Sort (unknown)
   +- Exchange (unknown)
      +- Exchange (unknown)
         +- Filter (unknown)
            +- Sample (unknown)
               +- Scan parquet (10)
```

- Final physical plan:

```
       == Physical Plan ==
AdaptiveSparkPlan (26)
+- == Final Plan ==
   * HashAggregate (25)
   +- ShuffleQueryStage (24)
      +- Exchange (23)
         +- * HashAggregate (22)
            +- * Project (21)
               +- SortMergejoin LeftOuter (20)
                  :- * Sort (9)
                  :  +- CustomShuffleReader (8)
                  :     +- ShuffleQueryStage (7)
                  :        +- Exchange (6)
                  :           +- * Sample (5)
                  :              +- ShuffleQueryStage (4)
                  :                 +- Exchange (3)
                  :                    +- * ColumnarToRow (2)
                  :                       +- Scan parquet (1)
                  +- * Sort (19)
                     +- CustomShuffleReader (18)
                        +- ShuffleQueryStage (17)
                           +- Exchange (16)
                              +- ShuffleQueryStage (15)
                                 +- Exchange (14)
                                    +- * Filter (13)
                                       +- * Sample (12)
                                          +- * ColumnarToRow (11)
                                             +- Scan parquet (10)

(8) CustomShuffleReader
Input [1]: [LAVORO_A#2]
Arguments: coalesced

(18) CustomShuffleReader
Input [1]: [LAVORO_B#10]
Arguments: coalesced
```

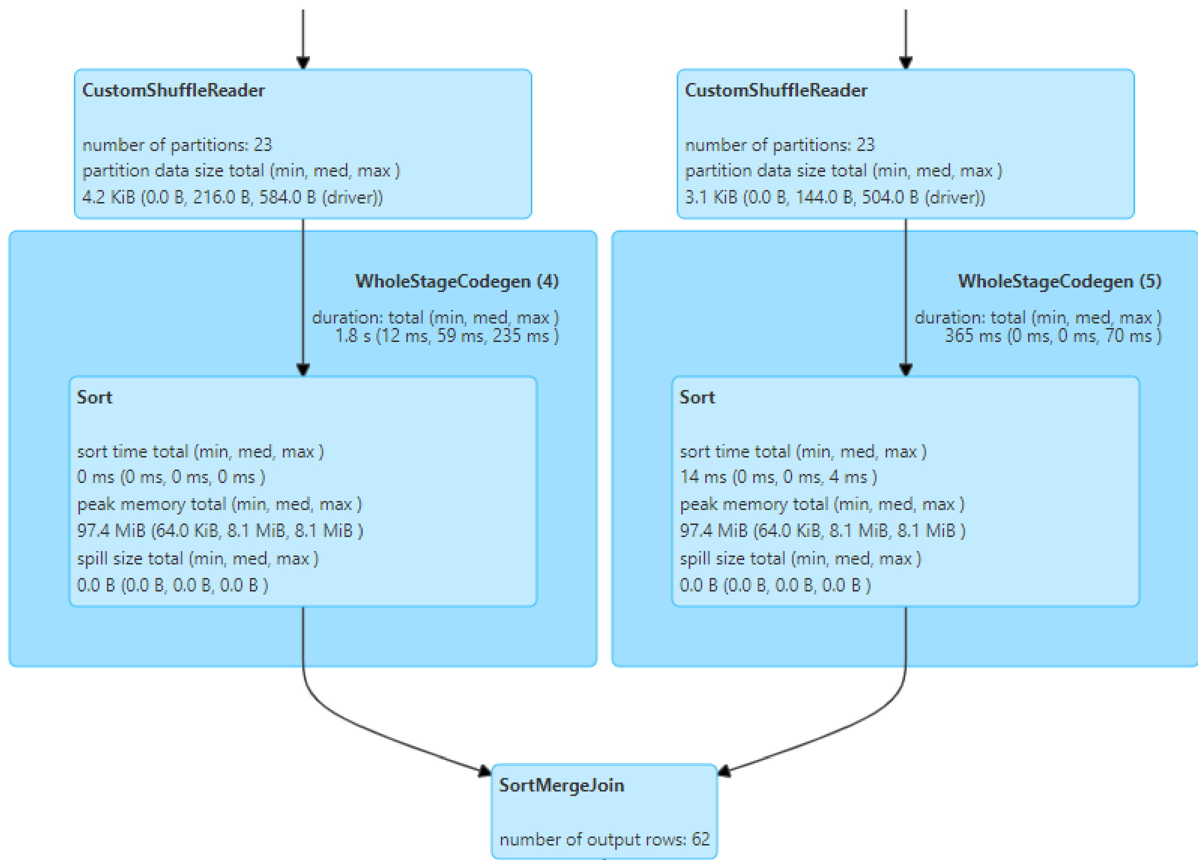

Figure 3.2: DAG illustrating a coalesce after the minimum number of partitions has been tuned

As shown by the SparkUI (image 3.2), while the minimum number of partitions was set at 50, both dataframes are coalesced into 23 partitions each.

Thus the configuration `spark.sql.adaptive.coalescePartitions.minPartitionNum` [20], while altering the number of partitions from 11 to 23, does not directly translate into the actual minimum number of partitions, as shown in Figure 3.2

Conclusions

As shown in the previous experiments and as stated in the developer's comments, the automatic coalesce is performed during shuffle phases via the usage of the CustomShuffleReader [6]. The user should, however, keep in mind that the minimum number of partitions set while influencing the number of partitions obtained after the coalesce, does not directly translate into the number of partitions per dataframe after the coalesce.

3.2.2. Repartition

Given AQE's ability to check whether a coalesce is needed, these experiments aim to check whether it is also capable of automatically performing repartitions, incrementing

the number of partitions. The workload used in these experiments was the following:

```
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()
```

Broadcast Hash Join

As seen in the Section 3.2.1 and as reported in Spark's code [6] [4], the presence of a shuffle phase influences the engagement of AQE. As such, as already done in Section 3.2.1, the Broadcast Hash join strategy has been treated separately.

Spark was configured as follows:

```
threshold= 1 * 1024 * 1024
```

In both cases, the threshold was high enough to guarantee that a Broadcast Hash join strategy would be the one chosen by the Catalyst.

As the SparkUI shows (Appendix B, 8.2.1), no repartition was performed both with AQE enabled and disabled, which was to be expected given the documentation [20]

Sort Merge Join

Spark was configured as follows:

```
threshold= -1
```

As the SparkUI shows (Appendix B, Section 8.2.2), no repartition was performed both with AQE enabled and disabled, which was to be expected given that in Spark documentation nothing is said about AQE performing repartitions aimed at raising the number of partitions [20].

Conclusions

As shown in the experiments, AQE is not able to dynamically repartition dataframes incrementing the number of partitions. This is probably such since coalescing is far more efficient than repartitioning; thus the performance penalty for an unneeded coalesce operation is lower than repartitioning. This fact, united with that some of the situations in which a repartition could be beneficial (skewness in data) are handled by other AQE features, reduces the advantages that an automatic repartition feature could have.

3.3. Skewed Join Handling

In this part, the experiments explore AQE's capabilities to handle skew joins. According to documentation, AQE should be able to recognize where there is skewness in data leading to skewness in task execution time during joins and intervene to obtain better performances [20]. As already seen in the Section 3.2, for this effort to take place, there must be a shuffle phase for AQE to activate [6] [4].

As already seen in the Sections 3.1 and 3.2, the component of interest was isolated to analyze it on its own. The automatic coalesce of partitions was disabled via Spark's configurations while the dynamic selection of join strategy was avoided by properly setting the broadcast threshold and by workload design. Spark was configured as follows:

- Without AQE:

```
val spark=SparkSession
.builder()
.appName("SortMergejoinAQEOff")
.master("local[*]")
.config("spark.sql.adaptive.enabled","false")
.config("Spark.eventLog.enabled", "true")
.config("Spark.eventLog.dir","Spark-aqe-main/src/main/resources
    /historyServer/logs")
.config("spark.sql.shuffle.partitions", 50)
.config("spark.sql.autoBroadcastjoinThreshold", threshold)
.getOrCreate()
```

- With AQE:

```
val spark= SparkSession
.builder()
.appName("SortMergejoinAQEOn")
.master("local[*]")
.config("spark.sql.adaptive.enabled","true")
.config("spark.sql.adaptive.coalescePartitions.enabled","false")
.config( "spark.sql.adaptive.skewjoin.enabled","true")
.config("spark.sql.autoBroadcastjoinThreshold", threshold)
.config("Spark.eventLog.enabled", "true")
.config("Spark.eventLog.dir","Spark-aqe-main/src/main/resources
    /historyServer/logs")
.config("spark.sql.adaptive.coalescePartitions.minPartitionNum", 50)
.config("spark.sql.adaptive.advisoryPartitionSizeInBytes", 32 * 1024 )
.config("spark.sql.adaptive.skewjoin.skewedPartitionThresholdInBytes",
    64*1024)
.config("spark.sql.adaptive.skewjoin.skewedPartitionFactor", 1)
.config("spark.sql.shuffle.partitions", 50)
.getOrCreate()
```

3.3.1. Sort Merge Join

In this Section the experiments explore the AQE's actions in presence of a skewed Sort Merge join. The workload used in this Section is the following:

```
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 as data sources are Big and BigSkewed for, respectively, dfA and dfB

AQE disabled

This experiment shows the usual Spark behavior in presence of a skewed join. The experiment configurations are as follows:

```
threshold= -1

join_expression=dfA("skewedjoinKey")===dfB("joinKey")

join_type= left
```

As expected, Figures 3.3 and 3.4 highlight skewness in the tasks, contributing to bad performances and long execution times. In particular, it is possible to observe in Figure 3.3 that the task that takes the most to complete takes more the nine minutes, while the one following takes 0,7 seconds. This highlights a heavy skewness among the tasks.

AQE enabled

These results will serve as a baseline for AQE's optimization

Left Join This experiment presents the following configurations:

```
threshold= -1

join_expression=dfA("skewedjoinKey")===dfB("joinKey")

join_type= left
```

As the figures 3.5, 3.6 and 3.7 highlight, skewness is identified in the left table. This required a fine tuning of both indexes used to determine the skewness: AQE considers a partition skewed if it satisfies two requirements [20]:

1. Its dimension is above the threshold set by `spark.sql.adaptive.skewjoin.skewedPartitionThresholdInBytes`

Duration	GC Time	Shuffle Write Size / Records	Shuffle Read Size / Records
9.1 min	0.2 s	59 B / 1	105.3 KiB / 1019404
0.7 s	44.0 ms	59 B / 1	157.2 KiB / 59611
0.7 s	44.0 ms	59 B / 1	80 KiB / 40215
0.6 s	44.0 ms	59 B / 1	80.1 KiB / 40000
0.6 s		59 B / 1	2.6 KiB / 20132
0.6 s		59 B / 1	2.7 KiB / 20232
0.5 s		59 B / 1	2.7 KiB / 19981
0.5 s		58 B / 1	2.6 KiB / 20008
0.4 s	44.0 ms	59 B / 1	79.6 KiB / 39966
0.4 s	44.0 ms	59 B / 1	2.6 KiB / 19975
0.3 s		59 B / 1	158.8 KiB / 59862
0.3 s		59 B / 1	81.1 KiB / 40604
0.3 s		59 B / 1	79.8 KiB / 39975
0.3 s		59 B / 1	80 KiB / 39771
0.3 s	44.0 ms	59 B / 1	2.6 KiB / 20010
0.2 s		59 B / 1	2.6 KiB / 20152
0.2 s		59 B / 1	80.6 KiB / 39983
0.2 s		59 B / 1	79.4 KiB / 39902
0.2 s		59 B / 1	2.6 KiB / 19894
0.2 s	44.0 ms	59 B / 1	2.6 KiB / 20019

Figure 3.3: Table showing the statistics of the 20 tasks that took the most time to complete

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	21.0 ms	79.0 ms	0.2 s	0.3 s	9.1 min
GC Time	0.0 ms	0.0 ms	0.0 ms	0.0 ms	0.2 s
Shuffle Read Size / Records	0.0 B / 0	0.0 B / 0	2.6 KiB / 20019	79.8 KiB / 39902	158.8 KiB / 1019404
Shuffle Write Size / Records	56 B / 1	56 B / 1	59 B / 1	59 B / 1	59 B / 1

Figure 3.4: Tasks statistics in a skewed Sort Merge join with AQE disabled

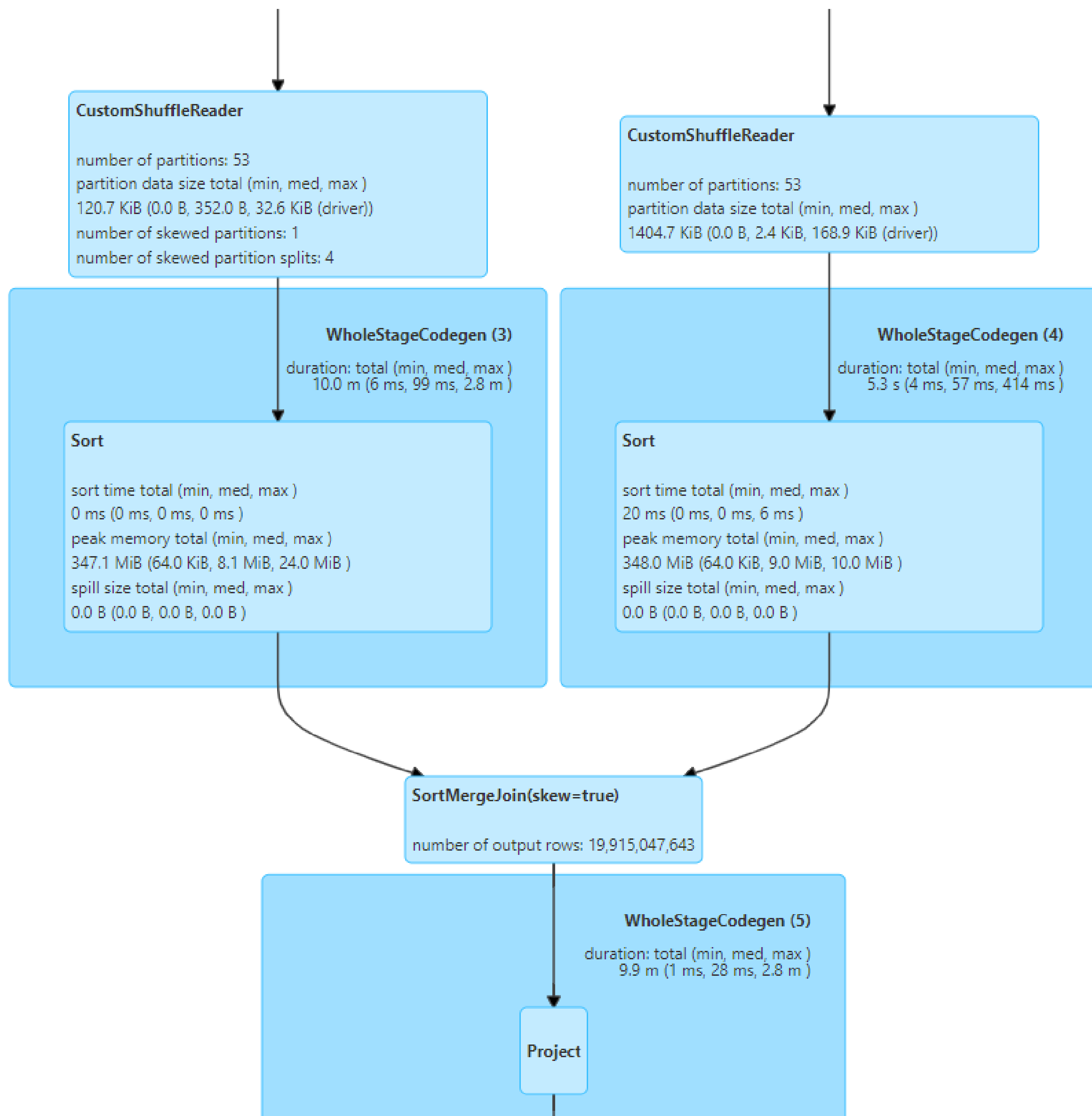

Figure 3.5: DAG of a skewed Sort Merge join with AQE enabled

2. Its dimension is above the median of the partition dimension multiplied by the factor defined as `spark.sql.adaptive.skewjoin.skewedPartitionFactor`

As an additional rule, the Spark documentation suggests to set `spark.sql.adaptive.skewjoin.skewedPartitionThresholdInBytes` to a value at least double of the one set for `spark.sql.adaptive.advisoryPartitionSizeInBytes` [20].

Right Join This experiment presents the following configurations:

```
threshold=  -1
```

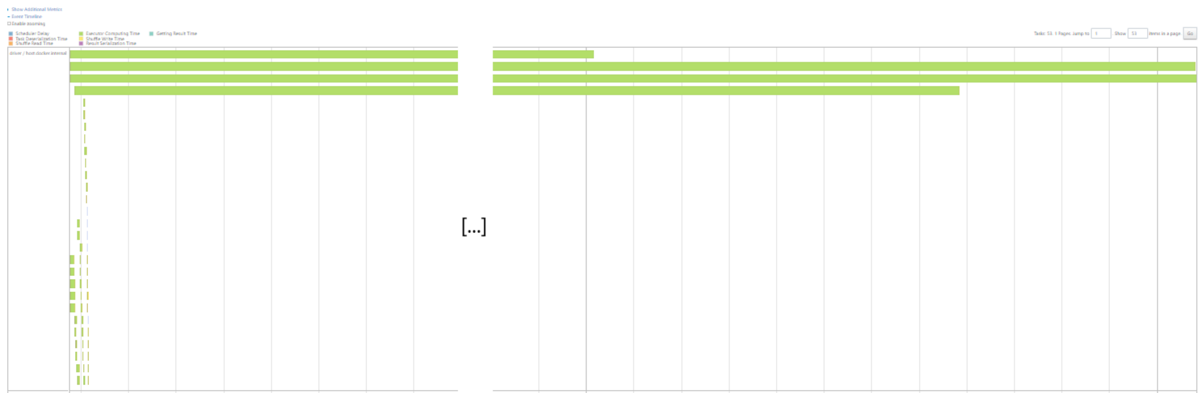

Figure 3.6: Tasks duration in a skew Sort Merge join with AQE enabled

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	17.0 ms	60.0 ms	0.2 s	0.3 s	7.0 min
GC Time	0.0 ms	0.0 ms	0.0 ms	0.0 ms	0.2 s
Shuffle Read Size / Records	0.0 B / 0	0.0 B / 0	2.6 KiB / 20050	79.4 KiB / 39975	158.8 KiB / 320629
Shuffle Write Size / Records	56 B / 1	56 B / 1	59 B / 1	59 B / 1	59 B / 1

Figure 3.7: Tasks statistics in a skewed Sort Merge join with AQE enabled

```
join_expression=dfA("skewedjoinKey")===dfB("joinKey")

join_type= right
```

As the Figures 8.2, 8.3 and 8.4 in Appendix B, Section 8.3 highlight, skewness is identified in the right table, where there should not be given the statistics of the aforementioned table, due to type of join used and the Spark configurations. Since the join is defined as right the result will depend on the contents of the right table, as such AQE will look for skewness only on the right-hand table. Given the low thresholds set for the classification as skewed, will find skewed partitions and over-correct the issue.

Inner Join This experiments presents the following configurations:

```
threshold= -1

join_expression=dfA("skewedjoinKey")===dfB("joinKey")

join_type= inner
```

As the Figures 8.5, 8.6 and 8.7 in Appendix B, Section 8.3 highlight, skewness is identified in both tables. This is due, as in the previous experiments, to the type of join chosen and the low thresholds: given that the join results depends on the contents of both tables, AQE will search for skewness in both of them and, given the low thresholds, will find skewed partitions on both sides.

Smaller partitions Given the results obtained in the experiment regarding the setting of the minimum number of partitions, this experiment checked whether it is possible to define the dimension of the partitions in which the skewed one is repartitioned. This experiments presents the following configurations:

```
threshold= -1

join_expression=dfA("skewedjoinKey")===dfB("joinKey")

join_type= left

.config("spark.sql.adaptive.advisoryPartitionSizeInBytes", 16* 1024)
```

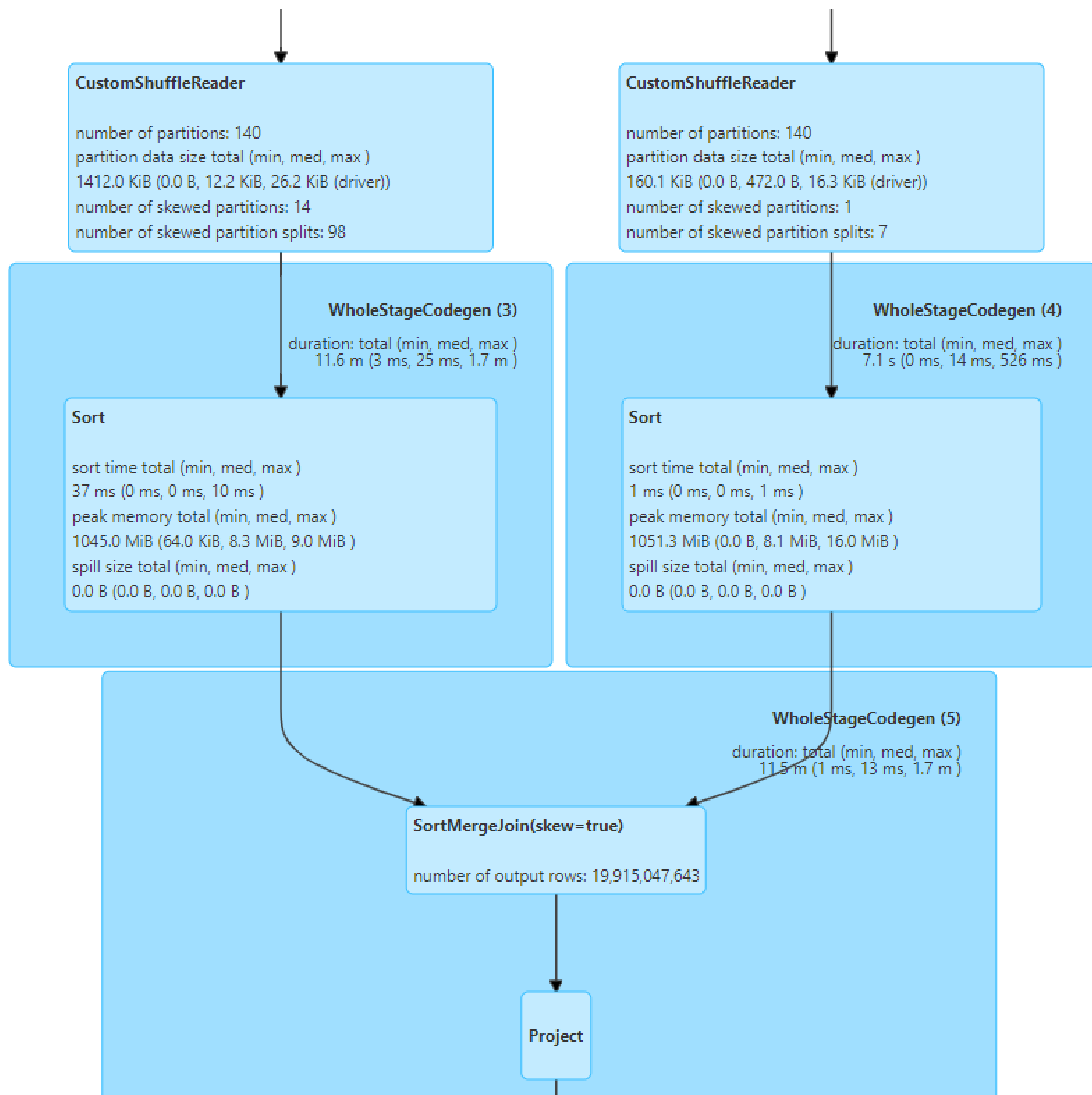

Figure 3.8: DAG of a skewed Sort Merge join with AQE enabled, the advisory size of partitions has been decreased

As the Figures 3.8, 3.9 and 3.10 highlight, tuning `spark.sql.adaptive.advisoryPartitionSizeInBytes` grants the user direct control over the dimension of the new partitions generated by AQE.

Higher partition factor This experiment explores whether the usage of the skewed partition factor can be used effectively to avoid repartitioning partitions that are not skewed enough to necessitate repartitioning.

The configurations used are as follows:

```
threshold=  -1
```

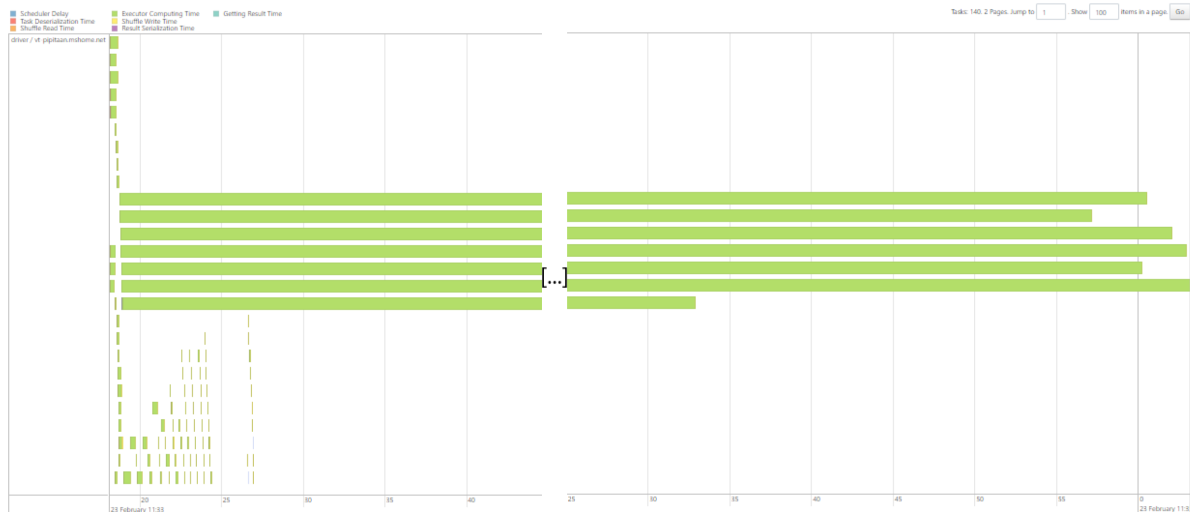

Figure 3.9: Tasks duration in a skewed Sort Merge join with AQE enabled, the advisory size of partitions has been decreased

Summary Metrics for 140 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	14.0 ms	32.0 ms	47.0 ms	0.1 s	1.7 min
GC Time	0.0 ms	0.0 ms	0.0 ms	0.0 ms	0.3 s
Shuffle Read Size / Records	0.0 B / 0	8 KiB / 5938	12.4 KiB / 6048	12.7 KiB / 9010	24.6 KiB / 170286
Shuffle Write Size / Records	56 B / 1	59 B / 1	59 B / 1	59 B / 1	59 B / 1

Figure 3.10: Tasks statistics in a skewed Sort Merge join with AQE enabled, the advisory size of partitions has been decreased

```
join_expression=dfA("skewedjoinKey")===dfB("joinKey")

join_type= inner

.config("spark.sql.adaptive.skewedPartitionFactor", 100)
```

As the Figures 3.11, 3.12 and 3.13 highlight, the skewed partitions are now only found in the left-hand side of the join. It is thus possible to use `spark.sql.adaptive.skewedPartitionFactor` as a relative unit of measure to make sure that AQE does not repartition partitions that are only slightly skewed.

Conclusions From the previous experiments, it is possible to conclude that AQE behaves as described in the documentation when interacting with a skewed Sort Merge join, with the added functionality of searching for skewness according to the type of join performed. This feature can lead to problems in cases in which the skewness is present in the table that is not referred to by the join type. Furthermore, as an operational note, the `spark.sql.adaptive.skewedPartitionFactor` configuration is of great relevance:

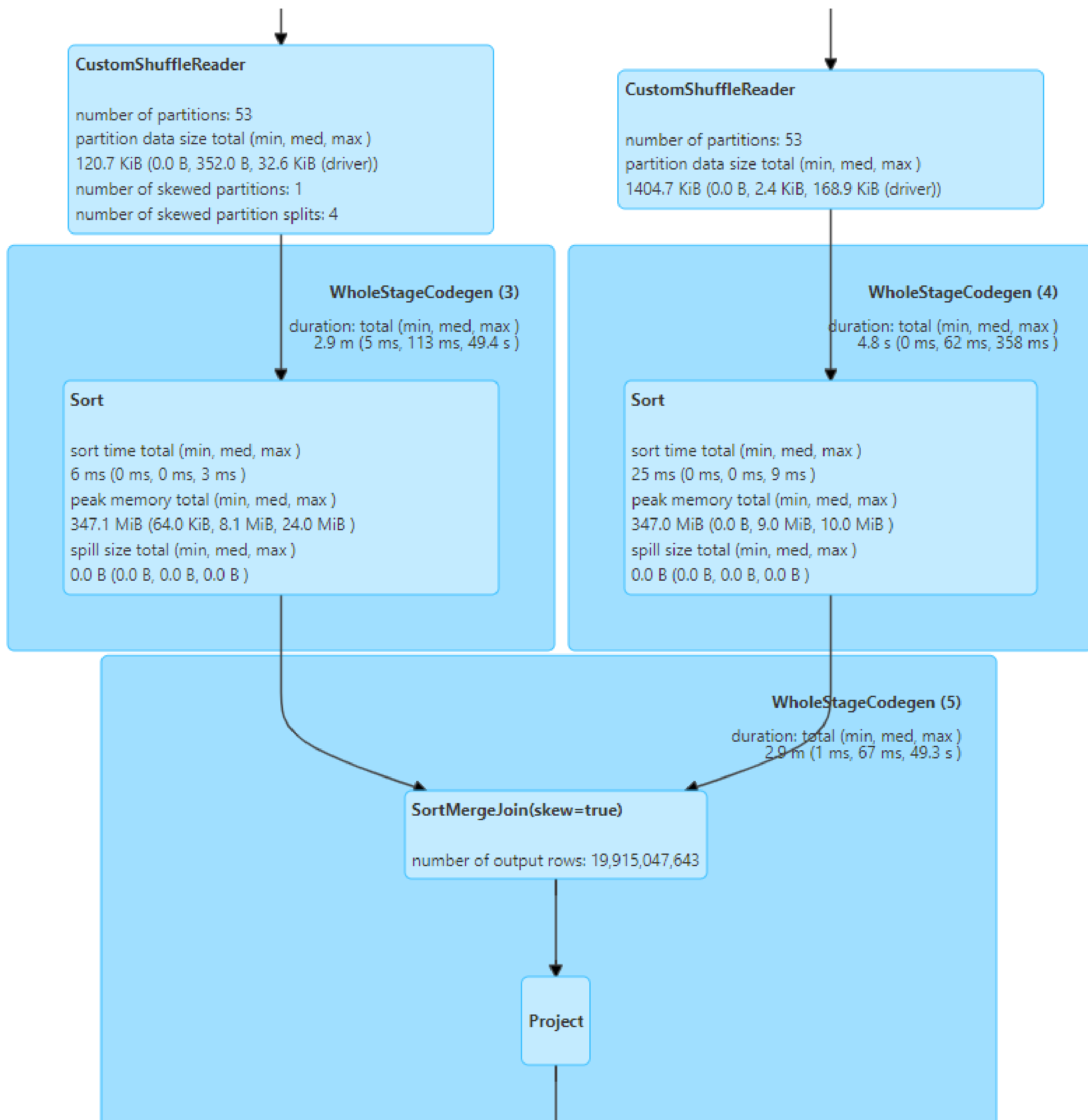

Figure 3.11: DAG

while it is possible to set an absolute threshold for skewed partitions, the possibility of configuring a threshold relative to the median partition dimension lends itself to more flexible configurations.

3.3.2. Broadcast Hash Join

This experiment aims to check whether AQE can recognize skewness in a Broadcast Hash join. The workload was executed both with AQE enabled and disabled. These experiments use the following configurations:

```
threshold=  1 * 1024 * 1024
```

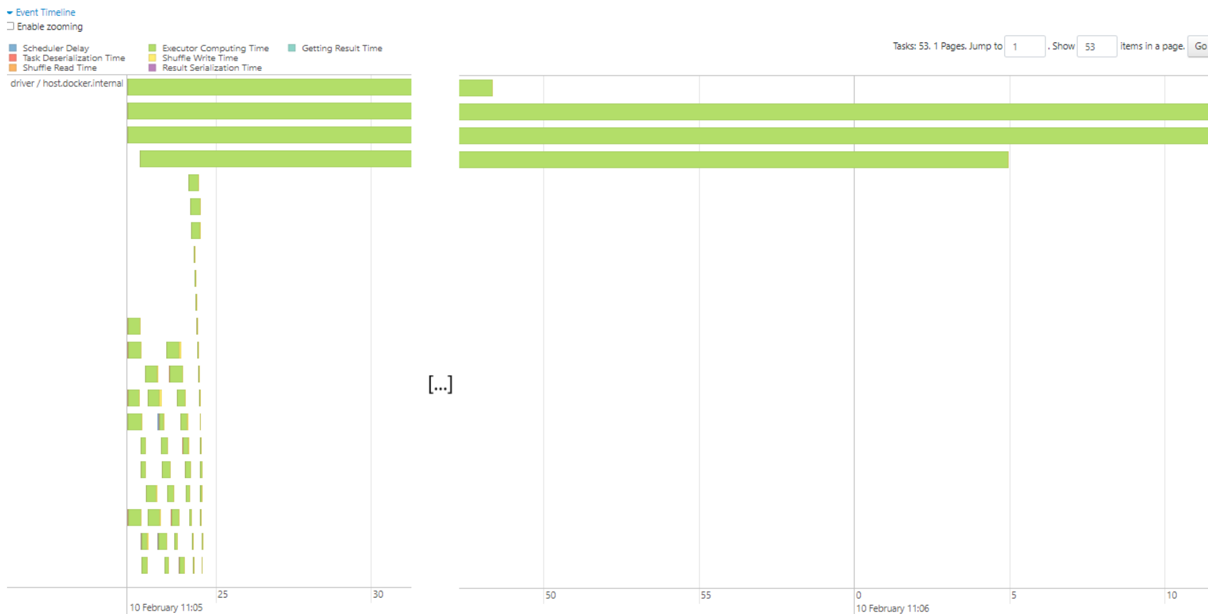

Figure 3.12: Tasks duration

Summary Metrics for 53 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	16.0 ms	77.0 ms	0.2 s	0.3 s	55 s
GC Time	0.0 ms	0.0 ms	0.0 ms	0.0 ms	0.2 s
Shuffle Read Size / Records	0.0 B / 0	0.0 B / 0	2.6 KiB / 20050	79.4 KiB / 39975	158.8 KiB / 320629
Shuffle Write Size / Records	56 B / 1	56 B / 1	59 B / 1	59 B / 1	59 B / 1

Figure 3.13: Tasks statistics

```
join_expression=dfA("skewedjoinKey")===dfB("joinKey")

join_type= left
```

The data sources are the same that were used in the Sort Merge join Section As the SparkUI (Appendix B, Section 8.3, Images 8.9, 8.10, 8.8, 8.9) highlights, there are no differences between the execution with and without AQE. This is so because in a Broadcast Hash join there is no shuffle phase: as such the custom shuffle reader is not engaged and, thus, the skewness is not identified.

While this behavior is expected, given the need for a shuffle phase for AQE to enact changes to the query plan [6] [4], is not in line with the documentation [20].

3.3.3. Cartesian Product

This experiment aims to check whether AQE can recognize skewness in a Cartesian Product join. The workload was executed both with AQE enabled and disabled. These exper-

iments use the following configurations:

```
threshold= -1

val dfA = spark.read.parquet("Spark-aqe-main/src/main/resources/skewReduced")
val dfB =
   spark.read.parquet("Spark-aqe-main/src/main/resources/notSkewReduced")

join_expression=dfA("skewedjoinKey") >dfB("joinKey")

join_type= inner
```

The data sources used in this experiment are the tables skewReduced and notSkewReduced, respectively for dfA and dfB. The tables have been changed from the previous experiments on skewed joins since both the Cartesian Product and the Broadcast Nested Loop join strategies are too resource-intensive to be executed on tables as big as the tables Big and BigSkewed. The tables used here are smaller than Big and BigSkewed while maintaining a similar distribution of values (Appendix A, Section 7). Experimental results prove that these tables are still linked to a high degree of skewness in the tasks part of a Sort Merge join (Appendix B, Section, 8.1). dfA is obtained by reading the skewReduced table, while dfB from the notSkewReduced table As the SparkUI highlights(Appendix B, Section 8.3, Figures 8.13, 8.12, 8.14, 8.15), there is no evidence of skewness in the tasks and AQE has no effect on the results .

3.3.4. Broadcast Nested Loop Join

This experiment aims to check whether AQE can recognize skewness in a Broadcast Nested Loop join. The workload was executed both with AQE enabled and disabled. These experiments use the following configurations:

```
threshold= -1

val dfA = spark.read.parquet("Spark-aqe-main/src/main/resources/skewReduced")
val dfB =
   spark.read.parquet("Spark-aqe-main/src/main/resources/notSkewReduced")

join_expression=dfA("skewedjoinKey") > dfB("joinKey")

join_type= left
```

The tables used in this experiment are the same used in the previous experiment (Section 3.3.3) for the reasons highlighted in that same experiment As the SparkUI highlights (Appendix B, Section 8.3, Figures 8.16, 8.17, 8.18 and 8.19),The results are the same as the ones obtained in the previous experiment (Section 3.3.3): while an inner Sort Merge join executed between the same tables on `dfA("skewedjoinKey")===dfB("joinKey")` highlights otherwise (Appendix B, Section, 8.1), and AQE has no effect on the results.

3.3.5. Conclusions

The results of the experiments provide four relevant insights regarding how AQE handles skew joins, the first one being that, if the join strategy requires a shuffle phase, through the usage of the custom shuffle reader, the partitions identified as skewed are repartitioned in smaller ones and the other table is modified accordingly for the join to be possible. The second insight that can be gathered from these experiments is that the effectiveness of AQE's component that handles skew joins is strictly linked to the tuning of the parameters `spark.sql.adaptive.skewjoin.skewedPartitionThresholdInBytes` and `spark.sql.adaptive.skewjoin.skewedPartitionFactor`: without tuning them in an appropriate fashion AQE will identify skewness where data is not skewed or will not recognize skewness where it is present. The third insight is that the join type determines what side of the join will be analyzed while searching for skewness. The fourth and last insight regards the join strategies that do not make use of shuffle phases, such as Cartesian Product, Broadcast Nested Loop, and Broadcast Hash join: when these strategies are chosen skewness in the join column does not translate into skewness in the join tasks. Thus, AQE is not needed and, if it were, it would not be able to correct the join execution, given that a shuffle phase is needed to engage the CustomShuffleReader, responsible for handling the skewed joins.

4 | Workload Tests

Chapter 3 has analyzed AQE in a curated environment, making use of workloads suit-tailored for AQE's capabilities. This Chapter will instead provide an analysis of AQE's effects on real-world scenarios making use of two workloads currently in use in production ETL pipelines, executed on a distributed cluster. The workloads used will be referred to as Workload A and Workload B. Workload A is responsible for updates on delta tables. Workload B instead executes a query presenting long DAGs with frequent joins, statically scheduled as Sort Merge joins, and highly filtering statements. On both the workloads both the effects of the automatic post-shuffle coalesce and the dynamic join selection will be tested. An important thing to note is that, prior to these experiemnts, both workloads had the broadcast threshold set at -1, thus disabling the Broadcast Hash join.

4.1. Workload A

Workload A is responsible for updating delta tables of various dimensions: as such it involves short DAGs with joins scheduled as sort-merge join. This opens up the possibility to use the automatic post-shuffle coalesce to improve the Sort Merge join performances. The usage of post-shuffle coalesce could also impact the operations on delta tables, as the number of files involved is dictated by the number of partitions in which the data to be written is divided. Another possibility is enabling the dynamic join strategy selection in order to schedule as many Broadcast Hash joins as possible in place of the Sort Merge joins. Given the nature of the workload, AQE's impact both on Spark's performances and on the delta tables' update times will be analyzed. Workload A is executed by a Driver with 14GiB of memory, memory overhead included, and 1 core, and 4 executors, each with 4 cores and 27GiB of memory, overhead included.

4.1.1. Experiment 1

The first experiment had the duration of one week and made use of the following settings for AQE:

- `spark.sql.adaptive.enabled`: True , in order to enable AQE
- `spark.sql.adaptive.coalescePartitions.enabled`: True in order to enable the feature studied in this phase

- `spark.sql.adaptive.skewjoin.enabled`: False
- `spark.sql.autoBroadcastjoinThreshold`: -1, as such both for reasons related to the workload and to disable the dynamic join selection
- `spark.sql.adaptive.coalescePartitions.minPartitionNum`: dafault value [20]
- `spark.sql.adaptive.advisoryPartitionSizeInBytes`: 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
- `spark.sql.adaptive.coalescePartitions.initialPartitionNum`: left to the default value of `none`, as it was considered to be irrelevant to this experiment.

The results were collected over the duration of a week and were compared to the results of the previous 2 weeks of execution, which did not make use of AQE.

While the SparkUI shows that multiple coalesce are performed, thus confirming the results obtained in chapter 3, Section 3.2.1, to better analyze AQE's impact the following paragraphs will provide a more in-depth analysis of the results. Data collected on the duration of the experiments are reported in Figure 4.1. As shown, while the average changes greatly from one week to the following one, it is possible to observe how the results of the first week of testing are aligned with the overall data from the previous two weeks.

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

Figure 4.1: Table describing the data regarding the first week of tests and older executions. The values express the duration of the workload execution in minutes

To better compare the average execution time in the first week to the results from the previous two weeks a t-test was performed between them, testing the null hypothesis that the two samples have the same mean. The test was set to take into account

the different variances, yielding t-statistic of -0.03304265493099191 and a p-value of 0.9736490854405545, thus confirming that it is reasonable to consider the two mean values to be equal. This suggests that the usage of the automatic post-shuffle coalesce has no measurable impact on performances. Another relevant finding to point out is that both AQE and Spark as a whole acted as they did while executing locally.

4.1.2. Experiment 2

While the experiment reported in 4.1.1 focused on testing the automatic post-shuffle coalesce capabilities of AQE, this experiment will also introduce the dynamic join strategy selection module while tuning the automatic post-shuffle coalesce. This phase makes use of the following configurations, as far as AQE is concerned:

- `spark.sql.adaptive.enabled`: True
- `spark.sql.adaptive.advisoryPartitionSizeInBytes`: 256 MiB
- `spark.sql.adaptive.skewjoin.enabled`: False
- `spark.sql.adaptive.coalescePartitions.enabled`: True
- `spark.sql.adaptive.coalescePartitions.minPartitionNum`: 16
- `spark.sql.autoBroadcastjoinThreshold`: 20MiB

In this instance, the AQE components have not been isolated: by enabling both the dynamic join selection and the automatic post shuffle coalesce this experiment will also explore the interaction between the two components. As shown in Section 3.2.1 there is the need for a shuffle phase to engage the automatic coalesce mechanism: the elimination of a Sort Merge join in favor of a Broadcast Hash join, be it static or dynamic, removes the possibility to perform a coalesce of the partitions. In this experiment, it will be possible to study how this interaction affects the performances both on Spark and on the delta merge operations.

The comparison of execution times between the two weeks before experiment one and the week of experiment two show a significant change in the average execution times: when compared, as shown in Figure 4.2, execution times have been reduced by about one half. This result is supported by a t-test between the samples from the second experiment and the samples taken from the two weeks prior to the start of the experiments: the previously mentioned t-test, executed with the same settings as the ones used for the t-test quoted in 4.1.1. The t-test returned a t-statistic value of -3.481360458255661 and a p-value of 0.0005245685385066565, thus confirming that, with a good amount of significance, the averages are different and the usage of AQE improved Spark's performances

This can be considered an effect of the introduction of statically-scheduled hash broadcast joins that, substituting the otherwise selected Sort Merge joins, noticeably reduce

	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 4.2: Results of both the first and second experiments over Workload A. The values express the duration of the workload execution in minutes

execution times. It has to be noted that no dynamic join strategy change was performed. Because of the absence of shuffle stages between filter and joins, and because of the data involved, Broadcasts Hash joins were only scheduled statically. Another relevant finding to be highlighted is that the dynamic selection of join strategy, when used in a cluster, behaves in the same way it behaved while executing locally. It was also found that the automatic post shuffle coalesce does not respect the minimum number of partitions set via Spark's configurations. This confirms the findings in Section 3.2.1.

4.1.3. Data From Delta Logs

Other than only analyzing the execution times returned by the SparkUI, analyzing the delta logs returned further information on the effects of AQE. As Table 8.1 shows, there is a significant decrease in the percentages of files appended and deleted in both weeks of experimentation. This is due to the reduced number of partitions, which in turn stems from the automatic post-shuffle coalesce. While this should translate in lower read and write times, results in the same table show little to no improvement in rewrite times and a small increase in the scan times. This highlights that the bottleneck in delta merges could be the transmission of data over internet. It is well-known that data transfer over the web takes significantly more time than accessing caches, main memory or storage memory on the same device. These data transfer times act as a bottleneck over the update of delta tables and hinder the optimizations otherwise seen in Spark's performances.

The estimation of the distributions for the averages of files added (Figure 4.3, Appendix B, Section 8.4), write times (Figure 4.4) and read times (Figure 4.5), computed over the mean value of the aforementioned metrics for each table involved over the weeks, further show that, while there is a significant difference between the number of files added, there is no difference in the scan and write times.

These findings regarding the behaviour of delta update operations highlight that the number of files added (which means the number of partitions the data to be written is divided in) is not a bottleneck. Data shows that, while the average number of files involved

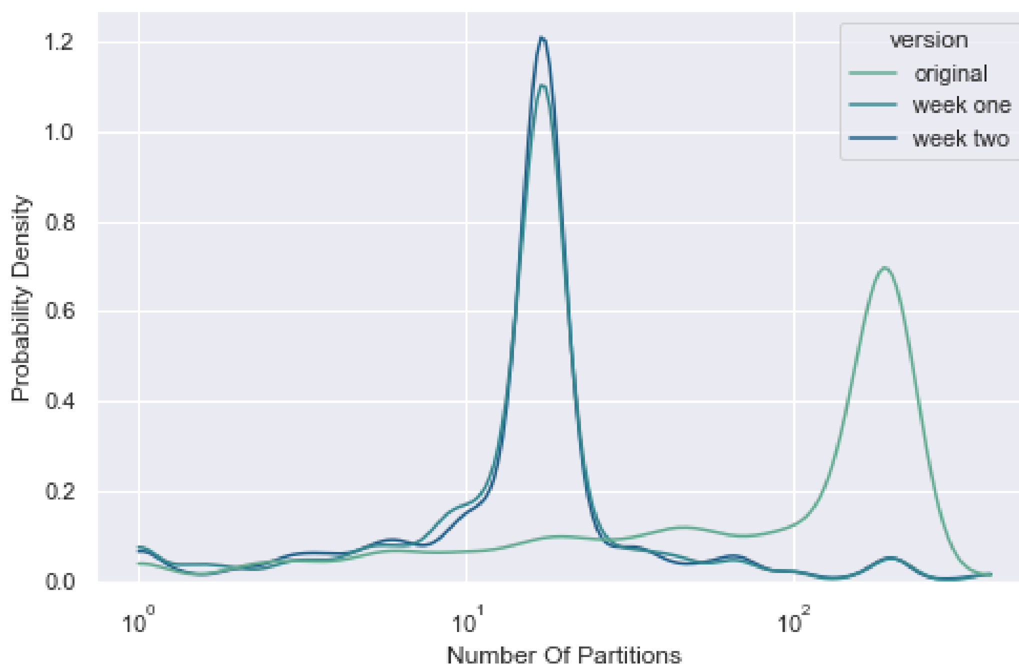

Figure 4.3: Estimated distribution of the per-table average of the number of files to be added to the delta tables, axes in logarithmic scale, base 10

in update operation diminishes, execution times do not.

4.2. Workload B

Workload B was chosen because of the complexity of the DAGs and the high selectivity of some of the filtering operations involved: after multiple filtering operations that reduce the size of some tables involved under 20 MiB the joins involving these tables are scheduled as Sort Merge joins. This opens up two possible optimization routes: introducing the dynamic selection of join strategy, thus using AQE to switch from Sort Merge join to Broadcast Hash join, and introducing the automatic post shuffle coalesce of partitions, thus improving execution times. Workload B is executed by a Driver, 1 cpu core and 14 GiB of memory, overhead included, and two executors, each having 4 cpu cores and 43GiB of memory, overhead included.

4.2.1. Experiment 1

The first experiment had the duration eight days, saw the enabling of the dynamic join strategy selection and made use of the following settings for AQE:

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

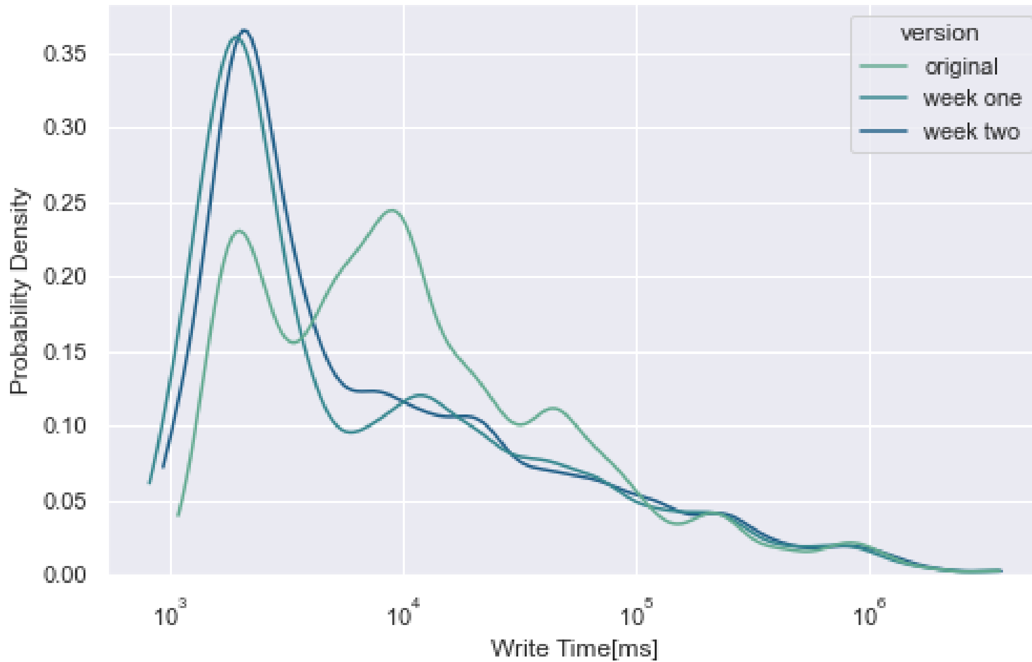

Figure 4.4: Estimated distribution of the per-table average of the delta rewrite times, axes in logarithmic scale, base 10

- `spark.sql.adaptive.coalescePartitions.enabled`: False in order to disable this feature
- `spark.sql.adaptive.skewjoin.enabled`: False
- `spark.sql.autoBroadcastjoinThreshold`: 20 MiB, since the inspection of previous executions of Workload B revealed that this value should guarantee the dynamic selection of Broadcast Hash joins.

While the configurations were set in order to guarantee the dynamic selection of the Broadcast Hash join, the SparkUI highlights that the Broadcast Hash join is never selected dynamically. This is due to the absence of shuffling phases between the filtering stages and the Sort Merge joins: without an intermediate shuffle stage the Catalyst has no visibility over the changes to the table dimensions and, thus, it is not possible to adapt the execution plan. This is to be expected, given the results obtained in Section 3.1 and the information available regarding when AQE performs optimizations [11]: to be able to switch dynamically the join strategy, AQE needs a shuffle phase between the filtering action and the subsequent Sort Merge join. The results, however, show a reduction over execution times of 20,79%. This is due to the static scheduling of Broadcast Hash joins in place of the Sort Merge joins scheduled previously. This shows that, while enabling AQE's dynamic join selection when no dynamic optimization can be applied to the DAG may introduce unneeded overhead costs, the advantages stemming from the enabling of the

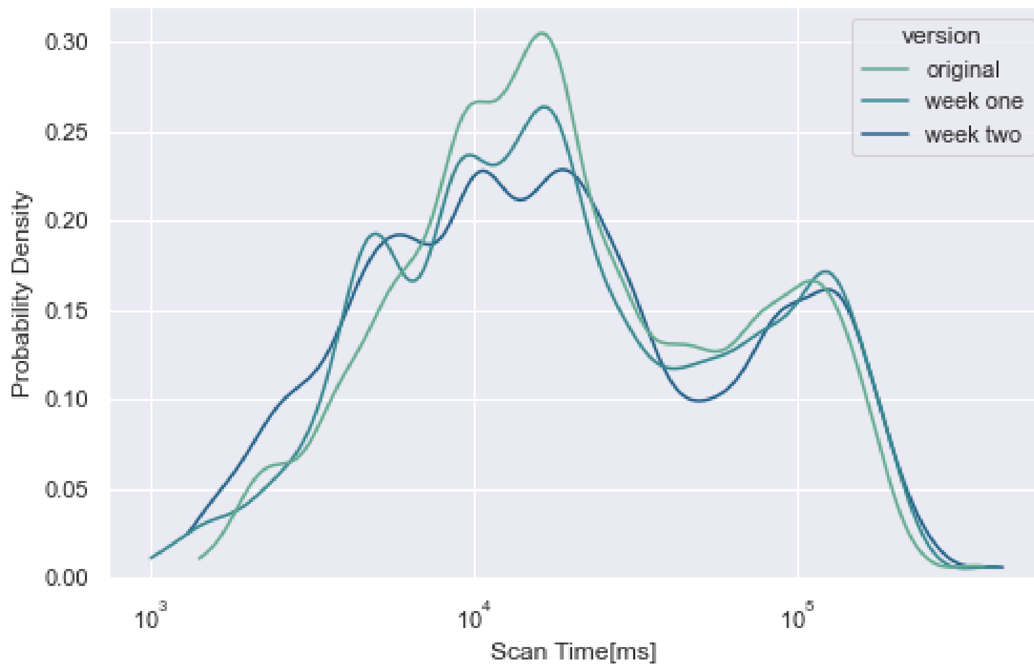

Figure 4.5: Estimated distribution of the per-table average of the delta scan times, axes in logarithmic scale, base 10

	AQE enabled, week 1	AQE disabled, week 1	AQE disabled, week 2	AQE disabled
count	15.000000	17.000000	12.000000	29.000000
mean	56.160000	63.165020	78.050000	69.324322
std	57.147138	58.433245	73.643768	64.322606
min	8.800000	0.005333	9.800000	0.005333
25%	13.500000	16.000000	14.750000	16.000000
50%	24.000000	23.000000	33.500000	31.000000
75%	126.000000	120.000000	157.500000	144.000000
max	144.000000	156.000000	174.000000	174.000000

Figure 4.6: Table describing the data regarding the first week of tests and older executions. The values express the duration of the workload execution in minutes

Broadcast Hash join can easily mask them and improve the overall performances. While these results appear to be clean-cut, it has to be noted that these comparisons have little statistical significance. A t-test was performed to compare the averages from the two weeks prior to the experiment and the results obtained during the eight days of experimentation, which took into account the independence of the samples and the different variances and

tested the null hypothesis that the two samples have the same mean, returned a t-statistic of -0.6934465336367015 and a p-value of 0.49309864314589724. This is due to the small sample size and the high variance among the samples, thus prompting the need for further experimentation to either verify or disprove these results.

Another finding to highlight is that the dynamic join selection behaves in the same way it did during local testing.

4.2.2. Experiment 2

In the second experiment, of the duration of 8 days, the effects of the automatic post shuffle coalesce of partitions were tested on a workload that involves a high number of Sort Merge joins in long DAGs. As far as AQE is concerned, the Spark configurations were as follows:

- `spark.sql.adaptive.advisoryPartitionSizeInBytes`: 256 MiB
- `spark.sql.adaptive.skewjoin.enabled`: False
- `spark.sql.adaptive.coalescePartitions.enabled`: True
- `spark.sql.adaptive.coalescePartitions.minPartitionNum`: 8
- `spark.sql.autoBroadcastjoinThreshold`: -1

Evidence (Image 4.7) shows that there is a significant decrease in execution times: this is due to the significant number of Sort Merge joins scheduled in the DAG and the effects that proper partitions number tuning has on them. By setting the number of minimum partitions to the number of total CPU cores available to the executors in all of those situations in which the number of partitions would be inferior Spark is still using all the degrees of parallelism available, thus improving the performances. Where instead the number of partitions is above 8 there is still optimization deriving from the reduced number of partitions in place of the default 200. While this improvement on performances appears clean-cut, it has to be noted that, as already observed in Section 4.2.1, the statistical significance is little to none. Performing a t-test, configured as the one quoted in Section 4.2.1, has returned a t-statistic of -1.0891514092468852 and a p-value of 0.28242440162286875, thus prompting the need for further testing to either confirm or disprove the results obtained during this test.

Another finding to be highlighted is that AQE's behaviour is in contrast with the one observed in Sections 3.2 and 4.1.2, as it respects the minimum number of partitions. In particular, the number of partitions appears to be grater than the threshold set in the Spark configurations. It was possible to observe, as an example, an instance in which 8.5 KiB of data were coalesced into 9 partitions. Given the advisory size of 256 MiB per partition, 8 would have sufficed.

	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 4.7: Table describing the data regarding both the first and second experiments and older executions. The values express the duration of the workload execution in minutes

5 | Conclusions and Future Work

The experiments executed in the previous two Chapters provide a detailed overview of AQE's capabilities. Chapter 3 investigated AQE's capabilities in a controlled environment, yielding multiple findings. In the first place, differently from what is outlined in the documentation [20] and the developers' comments on the source code [6], AQE can recognize when a scheduled Sort Merge join can be transformed into a Broadcast Hash join dynamically only when there is a shuffle stage between the filtering action and the subsequent Sort Merge join(Section 3.1.1). The reason behind this can be found in how AQE performs its optimization procedures [11]: AQE is only able to optimize future stages. Being unable to optimize the current stages it is unable to change the join strategy while performing the shuffle needed in the Sort Merge join. Another thing to note is that, given that the choice between Cartesian Product and Broadcast Nested Loop join is influenced by the broadcast threshold, AQE can dynamically switch from Cartesian Product to Broadcast Nested Loop join. Even if not described in the documentation, this choice is influenced by the broadcast threshold both with and without AQE(Section 3.1.1). Another relevant finding which is not only related to AQE, but the Catalyst, is that it does not always follow the algorithm for join strategy selection outlined in [7]. In particular, the choice between Cartesian Product and Broadcast Nested Loop join is influenced by the broadcast threshold (Section 3.1.1). Another variation from the defined behavior is that the usage of a broadcast hint can override the selection of a Sort Merge join in favor of a Broadcast Hash join in situations in which a Broadcast Hash join should not be feasible (Sections 3.1.2, and 3.1.2). The usage of the broadcast hint overrides the broadcast threshold: this is quite dangerous since it can lead to driver failure. The Catalyst's behavior in presence of a broadcast hint where a Sort Merge join would be scheduled is explained in-depth in the Table 3.1. This behavior entails that the usage of a broadcast hint is not a good substitute for AQE's Dynamic join Selection: while it would eliminate AQE's overhead costs, it could also bring the driver to crash, a situation in which fault recovery is not guaranteed by Spark, thus requiring the computation to be started over. Furthermore, if AQE is enabled and the broadcast threshold allows for it, AQE will switch from Cartesian Product to Broadcast Nested Loop join (Section 3.1.2. On the other hand, the presence of Sort Merge and Cartesian hints hinder AQE's capabilities to perform the switch from Sort Merge join to Broadcast Hash join and from Cartesian Products to Broadcast Nested Loop join. Given these findings regarding AQE's capabilities of dynamically changing join strategy, it seems that hints cannot safely substitute AQE and their usage together with AQE should be generally avoided unless the user has an extremely clear picture of the workload and data involved and wants to perform some

extremely accurate and punctual manipulations of the DAG.

Tests over AQE's ability to identify whether it could be beneficial to the performances to perform an automatic coalesce over a dataframe (Section 3.2) show that AQE behaves as described in the documentation, except for the setting regarding the number of partitions. On the other hand, as expected from the documentation, AQE is not capable of performing automatic repartitions over dataframes (Section 3.2.2). Given the results shown in Section 3.3 it seems, however, that AQE can increase the number of partitions. This is not due to its ability to perform repartitions: it is instead because, while handling skewed joins, partitions that are to be joined with the partitions obtained from splitting the skewed one are replicated. Given these results, it is possible to conclude that the automatic post-shuffle coalesces work, for the most part, as described in the documentation, except for the minimum number of partitions.

The experiments regarding the skewness handling by AQE have shown that AQE is capable, as described in the documentation, to identify skewness in join operations and correct them accordingly, however with two main caveats. The first caveat is that tuning AQE for skewness recognition can be quite challenging: the join type influences the join side in which AQE looks for skewness, completely disregarding the other one even in situations in which there is evident proof of skewness. Furthermore, the variables involved in the definition of skewness are complex to set properly. As per documentation, the variables directly involved are `spark.sql.adaptive.skewjoin.skewedPartitionThresholdInBytes` and `spark.sql.adaptive.skewjoin.skewedPartitionFactor`, respectively acting as an absolute threshold and a threshold relative to the median size of partitions involved in the join, heavily influence AQE's ability to recognize skewness and, if not set properly, could lead AQE to identify skewness where it is not present. The other caveat is that skewness identification and handling depends on the CustomShuffleReader component [4], which is only engaged in shuffle phases. This entails that it cannot be used in join strategies that do not involve shuffle phases. This, however, could not be tested here, as skewness in the join key fields does not translate directly in skewness in tasks, as shown in the experiments (Section 3.3). The results obtained regarding skew join handling show that AQE can be extremely effective while handling skewness, however needing proper tuning and the usage of a Sort Merge join as join strategy. These results make up a baseline evaluation of AQE's functionalities and shed light on the physical optimization applied by the Catalyst to the query execution plans. To evaluate AQE's impact on performances, the workload Section has analyzed AQE's impact on real-world scenarios.

Tests on the production workloads have, first and foremost, confirmed that triggering the automatic join selection can prove to be extremely difficult (Sections 4.2.1 and 4.1.2), as it is triggered only if there is a shuffle stage between the filtering action and the join stage, otherwise the Catalyst is not aware of the change in dimensions of the tables involved in the join. The performances can, however, still improve due to the static scheduling of Broadcast Hash joins, as it is not possible to enable this feature without enabling the static scheduling of Broadcast Hash joins too [14] [20]. Another interesting finding is that, according to the experiments performed (Sections 4.1.1, 4.1.2 and 4.2.2), the usage of automatic post-shuffle coalesce can improve performances where sort-merge

join are involved when the number of minimum partitions is set to take full advantages of the worker's parallelism, which means to set it to the total number of cores available across the workers. This is observable when comparing the results of experiment 4.1.1 to 4.1.2 and 4.2.2. It has to be noted, however, that the comparison is made spurious by the introduction of the Broadcast Hash join in 4.1.2 and the small statistical significance in both experiments 4.2.1 and 4.2.2. The experiments in Chapter 4 have also confirmed the findings in Section 3.2 regarding the minimum number of partitions. This was observed, in particular, in Section 4.1.2. In Section 4.2.2 it was instead possible to observe that the configuration was respected, with the minimum number of partitions observed exceeding it also when the data involved could fit in 8 partitions of the size of 256MiB. Given this behavior, it will be necessary to further study how AQE selects the target number of partitions for the coalesce.

Another relevant finding is that Spark and AQE behaved in the same way both in local execution and executing on a cluster. Overall, the experiments presented here brought a great amount of insight into both AQE and Spark's optimizer as a whole. There are, however, some points that could benefit from further testing. In the first place, further tuning the coalesce options could lead to a deeper understanding of the automatic coalesce, while further testing on skewness handling in a situation in which skewness is present when the join strategy is not a Sort Merge join could highlight more AQE functionalities. Also regarding the workload tests, further testing on more workloads would contribute to better understanding when the usage of AQE is beneficial.

Despite the great amount of insight brought by the experiments performed in this thesis, there is still a lot to test regarding AQE's capabilities. First and foremost, AQE's ability to handle skewness in joins was not tested in a production workload. To fully understand AQE's capabilities and to observe how this capability interacts with the others further, while also comparing its behavior on a cluster with the one it has locally, further testing in this direction is needed. Given the findings in Sections 3.2, 4.1.2 and 4.2.2 regarding the inconsistencies of the minimum number of partitions when automatically coalescing, further tests aimed at understanding the underlying mechanisms will be needed.

Furthermore, AQE's interaction with index usage has to be explored, especially with Hyperspace's indexing system. Hyperspace's indexes usage is integrated with Spark's Catalyst optimization procedures (as shown in 1); as such the usage of indexes might lead to avoiding shuffle phases. While, on its own, this optimizes the workloads' execution, when used combined with AQE this could lead to AQE being unable to recognize optimization possibilities. This interaction introduces, thus, a tradeoff between the two optimizations procedure that, when studied, could lead to interesting findings. Furthermore, as experiments in both Sections 3 and 4 have shown, properly tuning AQE's settings can make a great difference in terms of optimization. As such, further testing is needed to properly tune AQE and obtain the best results possible from its usage. Further testing is also needed, as already stated in Sections 4.2.1 and 4.2.2, to increase the number of samples to be analyzed and thus provide the data needed for results to be statistically significant.

6 | Bibliography

[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] *BigQuery technical whitepaper*. 2012. URL: https://cloud.google.com/files/BigQueryTechnicalWP.pdf.

[3] Biswapesh Chattopadhyay et al. "Tenzing a sql implementation on the mapreduce framework". In: *Proceedings of the VLDB Endowment* 4.12 (2011), pp. 1318–1327.

[4] *Coalesce need and skeweness in data identification code*. URL: https://github.com/apache/spark/blob/branch-3.1/sql/core/src/main/scala/org/apache/spark/sql/execution/adaptive/CustomShuffleReaderExec.scala.

[5] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[6] *Developers' comments on how and AQE's optimization is engaged*. URL: https://github.com/apache/spark/blob/branch-3.1/sql/core/src/main/scala/org/apache/spark/sql/execution/adaptive/AdaptiveSparkPlanExec.scala#L49.

[7] *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.

[8] Yuan Yu Michael Isard Dennis Fetterly et al. "DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language". In: *Proc. LSDS-IR* 8 (2009).

[9] Goetz Graefe. "The cascades framework for query optimization". In: *IEEE Data Eng. Bull.* 18.3 (1995), pp. 19–29.

[10] Goetz Graefe and David J DeWitt. "The EXODUS optimizer generator". In: *Proceedings of the 1987 ACM SIGMOD international conference on Management of data*. 1987, pp. 160–172.

[11] *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.

[12] Qifa Ke, Michael Isard, and Yuan Yu. "Optimus: a dynamic rewriting framework for data-parallel execution plans". In: *Proceedings of the 8th ACM European Conference on Computer Systems*. 2013, pp. 15–28.

[13] Sergey Melnik et al. "Dremel: interactive analysis of web-scale datasets". In: *Proceedings of the VLDB Endowment* 3.1-2 (2010), pp. 330–339.

[14] *Official documentation for SparkSQL configurations*. URL: https://spark.apache.org/docs/3.1.2/configuration.html#spark-sql.

[15] *Oracle Database SQL Tuning Guide, 21c*. 2021. URL: https://docs.oracle.com/en/database/oracle/oracle-database/21/tgsql/sql-tuning-guide.pdf.

[16] Rahul Potharaju et al. "Hyperspace: the indexing subsystem of azure synapse". In: *Proceedings of the VLDB Endowment* 14.12 (2021), pp. 3043–3055.

[17] Russell Power and Jinyang Li. "Piccolo: Building fast, distributed programs with partitioned tables". In: *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*. 2010.

[18] Raghav Sethi et al. "Presto: SQL on everything". In: *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE. 2019, pp. 1802–1813.

[19] Konstantin Shvachko et al. "The hadoop distributed file system". In: *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. Ieee. 2010, pp. 1–10.

[20] *Spark SQL Guide*. URL: https://spark.apache.org/docs/3.1.2/sql-performance-tuning.html.

[21] *Spark SQL guide on Hints*. URL: https://spark.apache.org/docs/3.1.2/sql-ref-syntax-qry-select-hints.html.

[22] *Spark SQL guide on Parquet datasources*. URL: https://spark.apache.org/docs/3.1.2/sql-data-sources-parquet.html.

[23] Ashish Thusoo et al. "Hive-a petabyte scale data warehouse using hadoop". In: *2010 IEEE 26th international conference on data engineering (ICDE 2010)*. IEEE. 2010, pp. 996–1005.

[24] Gary Valentin et al. "DB2 advisor: An optimizer smart enough to recommend its own indexes". In: *Proceedings of 16th International Conference on Data Engineering (Cat. No. 00CB37073)*. IEEE. 2000, pp. 101–110.

[25] Reynold S Xin et al. "Shark: SQL and rich analytics at scale". In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of data*. 2013, pp. 13–24.

[26] 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.

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

7 | Appendix A - Table statistics

This section provides insight on the statistics of the tables used in the aforementioned experiments.

7.1. Table A

Used in Sections 3.1 and 3.2. The joins are performed on the column LAVORO_A, which presents 13 evenly-distributed distinct values.Occupies 1,33MB when compressed in memory as a parquet file, 1356.4 KiB during execution according to the SparkUI. Value count for LAVORO_A column are reported in Table 7.1.

```
dfA
root
 |-- NOME: string (nullable = true)
 |-- COGNOME: string (nullable = true)
 |-- LAVORO_A: string (nullable = true)
 |-- BIRTHPLACE: string (nullable = true)
 |-- AGE: double (nullable = true)
```

LAVORO_A	count
H	38418
C	38264
J	38348
G	38398
L	38589
M	38366
K	38416
I	38666
B	38584
A	38569
F	38663
E	38338
D	38381

Table 7.1: Count of values in column Lavoro_A in table A

7.2. Table B

Used in the 3.1 and 3.2 Sections. The joins are performed on the column LAVORO_B, while the other fields are used to filter its rows in the Section 3.1 experiments in order to bring the dimension of Table B below the broadcast threshold. Occupies 1,51MB in memory when compressed as a parquet file, 1535.7 KiB in memory. Summary statistics for columns FILLER1, FILLER2 and FILLER3 are reported in Table 7.2, while value count for column LAVORO_B is reported in Table 7.3.

```
dfB
root
 |-- LAVORO_B: string (nullable = true)
 |-- FILLER1: double (nullable = true)
 |-- FILLER2: double (nullable = true)
 |-- FILLER3: double (nullable = true)
```

summary	FILLER1	FILLER2	FILLER3
count	500000	500000	500000
mean	49.52	49.445224	49.489928
stddev	28.848103419610442	28.85861052144784	28.874386331404185
min	0.0	0.0	0.0
25%	25.0	24.0	24.0
50%	50.0	49.0	49.0
75%	75.0	74.0	75.0
max	99.0	99.0	99.0

Table 7.2: Summary of columns FILLER1, FILLER2 and FILLER3 in table B

7.3. Table K

This table contains only the distinct values of the LAVORO_A and LAVORO_B keys, it is used in the 3.1 Section joined to table B in order to obfuscate the dimensions of table B to the Catalyst. Occupies 496 B when compressed as parquet file, 476B when uncompressed. The reduction in dimension is due to the fact that the SparkUI does not take into account parquet logs. The count of the values for column LAVORO_KEYS is reported in Table 7.4.

```
dfK
root
 |-- LAVORO_KEYS: string (nullable = true)
```

7.4. Table Big

This table contains two filler fields and one acting as join key, assuming 50 different values with equal probability. This table us used in the 3.3 Section. Occupies 2,36 MB when compressed in Parquet file, 15.3 MiB in memory according to the SparkUI. Count of values and summary statistics for column `joinKey` are reported in Tables 7.6 and 7.5.

```
dfBig
root
 |-- filler1: string (nullable = true)
 |-- filler2: string (nullable = true)
 |-- joinKey: integer (nullable = true)
```

7.5. Table BigSkewed

This table contains two filler fields and one acting as join key, assuming 50 different values with a high probability of being equal to 49. This table us used in the 3.3 Section as skewed dataset. Occupies 1,64MB when compressed as Parquet file, 15.3 MiB in memory according to the SparkUI. Count of values and summary statistics for column `skewedJoinKey` are reported in Tables 7.8 and 7.7.

```
dfBigSkewed
root
 |-- filler1S: string (nullable = true)
 |-- filler2S: string (nullable = true)
 |-- skewedjoinKey: integer (nullable = true)
```

7.6. Table SkewReduced

This table is a reduced version of BigTableSkewed, used in Section 3.3 for the tests involcing Cartesian Product and Broadcast Nested Loop join strategies in order to make the computation less intensive. Occupies 642KB when compressed as a Parquet file, 6.1 MiB in memory according to the SparkUI. Count of values and summary statistics for column `skewedJoinKey` are reported in Tables 7.10 and 7.9.

```
dfSkewedReduced
root
|-- filler1S: string (nullable = true)
|-- filler2S: string (nullable = true)
|-- skewedjoinKey: integer (nullable = true)
```

7.7. Table NotSkewReduced

This table is a reduced version of BigTable, used in Section 3.3 for the tests involving Cartesian Product and Broadcast Nested Loop join strategies in order to make the computation less intensive. Occupies 40,5KB when compressed as Parquet file, 7.8 KiB in memory according to the SparkUI. Count of values and summary statistics for column `joinKey` are reported in Tables 7.12 and 7.11.

```
NotSkewReduced
root
 |-- filler1: string (nullable = true)
 |-- filler2: string (nullable = true)
 |-- joinKey: integer (nullable = true)
```

LAVORO_B	count
K	37974
F	38601
E	38475
B	38085
M	38698
L	38252
D	38735
C	38506
J	38568
A	38558
G	38568
I	38467
H	38513

Table 7.3: Count of values in column LAVORO_B in table B

LAVORO_KEYS	count
K	1
F	1
E	1
B	1
L	1
M	1
D	1
C	1
J	1
A	1
G	1
I	1
H	1

Table 7.4: Count of values in column LAVORO_KEYS in table K

summary	joinKey
count	1000000
mean	24.480694
stddev	14.431315516654458
min	0
25%	12
50%	24
75%	37
max	49

Table 7.5: Summary of column joinKey in table Big

joinKey	count
0	20087
1	20117
2	19985
3	19966
4	20302
5	20002
6	19981
7	20000
8	19855
9	19979
10	20071
11	20050
12	19899
13	20023
14	19869
15	20208
16	20146
17	19971
18	20130
19	19933
20	19755
21	20135
22	20062
23	19909
24	20205
25	20040
26	19938
27	19964
28	19961
29	20056
30	19905
31	19894
32	20216
33	19902
34	20120
35	19842
36	19938
37	20077
38	19906
39	20073
40	19956
41	20050
42	20076
43	19541
44	20140
45	19878
46	19586
47	20381
48	20005
49	19915

Table 7.6: Count of values in column `joinKey` in table `Big`

summary	skewedjoinKey
count	1000000
mean	48.987106
stddev	0.6539144992796644
min	0
25%	49
50%	49
75%	49
max	49

Table 7.7: Summary of column skewedjoinKey in table BigSkewed

skewedjoinKey	count
0	11
1	9
2	6
3	15
4	12
5	8
6	11
7	8
8	18
9	12
10	14
11	10
12	12
13	8
14	8
15	14
16	6
17	13
18	5
19	8
20	11
21	14
22	9
23	9
24	11
25	10
26	11
27	8
28	14
29	10
30	16
31	7
32	16
33	4
34	12
35	4
36	14
37	9
38	11
39	9
40	16
41	5
42	12
43	8
44	12
45	16
46	4
47	7
48	14
49	999489

Table 7.8: Count of values in column `skewedjoinKey` in table `BigSkewed`

summary	skewedjoinKey
count	399412
mean	48.9856939701361
stddev	0.6957682559531335
min	0
25%	49
50%	49
75%	49
max	49

Table 7.9: Summary of column skewedjoinKey in table SkewReduced

skewedjoinKey	count
0	7
1	4
2	6
3	7
4	5
5	5
6	5
7	6
8	10
9	1
10	2
11	3
12	3
13	4
14	4
15	7
16	3
17	6
18	1
19	3
20	2
21	9
22	1
23	6
24	4
25	2
26	7
27	4
28	6
29	1
30	9
31	3
32	7
33	3
34	5
35	3
36	8
37	2
38	4
39	3
40	10
41	2
42	6
43	2
44	4
45	8
46	1
47	4
48	6
49	399188

Table 7.10: Count of values in column `skewedjoinKey` in table `SkewReduced`

summary	joinKey
count	498
mean	24.35542168674699
stddev	13.724369956655
min	0
25%	13
50%	24
75%	36
max	49

Table 7.11: Summary of column `joinKey` in table NotSkewReduced

joinKey	count
0	6
1	10
2	13
3	8
4	8
5	7
6	7
7	9
8	9
9	11
10	6
11	8
12	11
13	14
14	12
15	13
16	8
17	7
18	16
19	7
20	21
21	14
22	9
23	10
24	13
25	11
26	11
27	10
28	11
29	8
30	12
31	7
32	13
33	12
34	8
35	9
36	15
37	10
38	9
39	6
40	14
41	9
42	5
43	7
44	12
45	8
46	6
47	8
48	14
49	6

Table 7.12: Count of values in column `skewedjoinKey` in table `NotSkewReduced`

8 | Appendix B - Experimental results

8.1. Dynamic join strategy selection

8.1.1. Broadcast Hash Join without hints execution result

- Physical plan with AQE:
  - Initial plan:

```
  - == Initial Plan ==
HashAggregate (unknown)
+- Exchange (unknown)
   +- HashAggregate (unknown)
      +- Project (unknown)
         +- BroadcastHashjoin LeftOuter BuildRight (unknown)
            :- Scan parquet (1)
            +- BroadcastExchange (unknown)
               +- Exchange (unknown)
                  +- Filter (unknown)
                     +- Scan parquet (3)
```

  - Final plan:

```
    == Physical Plan ==
AdaptiveSparkPlan (16)
+- == Final Plan ==
   * HashAggregate (15)
   +- ShuffleQueryStage (14)
      +- Exchange (13)
         +- * HashAggregate (12)
            +- * Project (11)
               +- * BroadcastHashjoin LeftOuter BuildRight (10)
                  :- * ColumnarToRow (2)
                  :  +- Scan parquet (1)
                  +- BroadcastQueryStage (9)
```

```
+- BroadcastExchange (8)
   +- ShuffleQueryStage (7)
      +- Exchange (6)
         +- * Filter (5)
            +- * ColumnarToRow (4)
               +- Scan parquet (3)
```

- Physical plan without AQE:

```
== Physical Plan ==
* HashAggregate (12)
+- Exchange (11)
   +- * HashAggregate (10)
      +- * Project (9)
         +- * BroadcastHashjoin LeftOuter BuildRight (8)
            :- * ColumnarToRow (2)
            :  +- Scan parquet (1)
            +- BroadcastExchange (7)
               +- Exchange (6)
                  +- * Filter (5)
                     +- * ColumnarToRow (4)
                        +- Scan parquet (3))
```

join of interest: stage 8 without AQE, stage 10 with aqe

8.1.2. Broadcast Nested Loop Join without hints execution results

- Physical plan with AQE:

  – Initial plan:

```
+- == Initial Plan ==
   HashAggregate (unknown)
   +- Exchange (unknown)
      +- HashAggregate (unknown)
         +- Project (unknown)
            +- BroadcastNestedLoopjoin LeftOuter BuildLeft (unknown)
               :- BroadcastExchange (unknown)
               :  +- Scan parquet (1)
               +- Exchange (unknown)
                  +- Project (unknown)
                     +- BroadcastHashjoin LeftOuter BuildRight
                        (unknown)
                        :- Project (unknown)
                        :  +- Filter (unknown)
                        :     +- Scan parquet (5)
```

```
                  +- BroadcastExchange (unknown)
                     +- Filter (unknown)
                        +- Scan parquet (9)
```

– Final plan:

```
== Physical Plan ==
AdaptiveSparkPlan (24)
+- == Final Plan ==
   * HashAggregate (23)
   +- ShuffleQueryStage (22)
      +- Exchange (21)
         +- * HashAggregate (20)
            +- * Project (19)
               +- BroadcastNestedLoopjoin LeftOuter BuildLeft (18)
                  :- BroadcastQueryStage (4)
                  :  +- BroadcastExchange (3)
                  :     +- * ColumnarToRow (2)
                  :        +- Scan parquet (1)
                  +- ShuffleQueryStage (17)
                     +- Exchange (16)
                        +- * Project (15)
                           +- * BroadcastHashjoin LeftOuter BuildRight
                               (14)
                              :- * Project (8)
                              :  +- * Filter (7)
                              :     +- * ColumnarToRow (6)
                              :        +- Scan parquet (5)
                              +- BroadcastQueryStage (13)
                                 +- BroadcastExchange (12)
                                    +- * Filter (11)
                                       +- * ColumnarToRow (10)
                                          +- Scan parquet (9)
```

- Physical plan without AQE:

```
== Physical Plan ==
* HashAggregate (19)
+- Exchange (18)
   +- * HashAggregate (17)
      +- * Project (16)
         +- BroadcastNestedLoopjoin LeftOuter BuildLeft (15)
            :- BroadcastExchange (3)
            :  +- * ColumnarToRow (2)
            :     +- Scan parquet (1)
            +- Exchange (14)
               +- * Project (13)
```

```
+- * BroadcastHashjoin LeftOuter BuildRight (12)
   :- * Project (7)
   :  +- * Filter (6)
   :     +- * ColumnarToRow (5)
   :        +- Scan parquet (4)
   +- BroadcastExchange (11)
      +- * Filter (10)
         +- * ColumnarToRow (9)
            +- Scan parquet (8)
```

join of interest: stage 18 of the final physical plan with AQE, stage 15 of the physical plan without AQE.

8.1.3. Broadcast Hash Join with hints control experiment execution results

- Initial plan:

```
+- == Initial Plan ==
   HashAggregate (unknown)
   +- Exchange (unknown)
      +- HashAggregate (unknown)
         +- Project (unknown)
            +- BroadcastHashjoin LeftOuter BuildRight (unknown)
               :- Scan parquet (1)
               +- BroadcastExchange (unknown)
                  +- Exchange (unknown)
                     +- Filter (unknown)
                        +- Scan parquet (3)
```

- Final plan:

```
== Physical Plan ==
AdaptiveSparkPlan (16)
+- == Final Plan ==
   * HashAggregate (15)
   +- ShuffleQueryStage (14)
      +- Exchange (13)
         +- * HashAggregate (12)
            +- * Project (11)
               +- * BroadcastHashjoin LeftOuter BuildRight (10)
                  :- * ColumnarToRow (2)
                  :  +- Scan parquet (1)
                  +- BroadcastQueryStage (9)
                     +- BroadcastExchange (8)
                        +- ShuffleQueryStage (7)
                           +- Exchange (6)
```

```
            +- * Filter (5)
               +- * ColumnarToRow (4)
                  +- Scan parquet (3)
```

8.1.4. Broadcast Hash Join, Broadcast hint with AQE off experiment execution results

```
== Physical Plan ==
* HashAggregate (12)
+- Exchange (11)
   +- * HashAggregate (10)
      +- * Project (9)
         +- * BroadcastHashjoin LeftOuter BuildRight (8)
            :- * ColumnarToRow (2)
            :  +- Scan parquet (1)
            +- BroadcastExchange (7)
               +- Exchange (6)
                  +- * Filter (5)
                     +- * ColumnarToRow (4)
                        +- Scan parquet (3)
```

8.1.5. Sort Merge Join, Merge hint with AQE on experiment execution results

1. Hint on smaller table

   (a) Initial plan

```
+- == Initial Plan ==
   HashAggregate (unknown)
   +- Exchange (unknown)
      +- HashAggregate (unknown)
         +- Project (unknown)
            +- SortMergejoin LeftOuter (unknown)
               :- Sort (unknown)
               :  +- Exchange (unknown)
               :     +- Scan parquet (1)
               +- Sort (unknown)
                  +- Exchange (unknown)
                     +- Exchange (unknown)
                        +- Project (unknown)
                           +- BroadcastHashjoin LeftOuter BuildRight
                              (unknown)
                              :- Project (unknown)
                              :  +- Filter (unknown)
                              :     +- Scan parquet (6)
```

```
                        +- BroadcastExchange (unknown)
                           +- Filter (unknown)
                              +- Scan parquet (10)
```

(b) Final plan

```
== Physical Plan ==
AdaptiveSparkPlan (28)
+- == Final Plan ==
   * HashAggregate (27)
   +- ShuffleQueryStage (26)
      +- Exchange (25)
         +- * HashAggregate (24)
            +- * Project (23)
               +- SortMergejoin LeftOuter (22)
                  :- * Sort (5)
                  :  +- ShuffleQueryStage (4)
                  :     +- Exchange (3)
                  :        +- * ColumnarToRow (2)
                  :           +- Scan parquet (1)
                  +- * Sort (21)
                     +- ShuffleQueryStage (20)
                        +- Exchange (19)
                           +- ShuffleQueryStage (18)
                              +- Exchange (17)
                                 +- * Project (16)
                                    +- * BroadcastHashjoin LeftOuter
                                       BuildRight (15)
                                       :- * Project (9)
                                       :  +- * Filter (8)
                                       :     +- * ColumnarToRow (7)
                                       :        +- Scan parquet (6)
                                       +- BroadcastQueryStage (14)
                                          +- BroadcastExchange (13)
                                             +- * Filter (12)
                                                +- * ColumnarToRow (11)
                                                   +- Scan parquet (10)
```

2. Hint on big table

(a) Initial plan

```
+- == Initial Plan ==
   HashAggregate (unknown)
   +- Exchange (unknown)
      +- HashAggregate (unknown)
         +- Project (unknown)
```

```
            +- SortMergejoin LeftOuter (unknown)
               :- Sort (unknown)
               :  +- Exchange (unknown)
               :     +- Scan parquet (1)
               +- Sort (unknown)
                  +- Exchange (unknown)
                     +- Exchange (unknown)
                        +- Project (unknown)
                           +- BroadcastHashjoin LeftOuter BuildRight
                               (unknown)
                              :- Project (unknown)
                              :  +- Filter (unknown)
                              :     +- Scan parquet (6)
                              +- BroadcastExchange (unknown)
                                 +- Filter (unknown)
                                    +- Scan parquet (10)
```

(b) Final plan

```
== Physical Plan ==
AdaptiveSparkPlan (28)
+- == Final Plan ==
   * HashAggregate (27)
   +- ShuffleQueryStage (26)
      +- Exchange (25)
         +- * HashAggregate (24)
            +- * Project (23)
               +- SortMergejoin LeftOuter (22)
                  :- * Sort (5)
                  :  +- ShuffleQueryStage (4)
                  :     +- Exchange (3)
                  :        +- * ColumnarToRow (2)
                  :           +- Scan parquet (1)
                  +- * Sort (21)
                     +- ShuffleQueryStage (20)
                        +- Exchange (19)
                           +- ShuffleQueryStage (18)
                              +- Exchange (17)
                                 +- * Project (16)
                                    +- * BroadcastHashjoin LeftOuter
                                        BuildRight (15)
                                       :- * Project (9)
                                       :  +- * Filter (8)
                                       :     +- * ColumnarToRow (7)
                                       :        +- Scan parquet (6)
                                       +- BroadcastQueryStage (14)
                                          +- BroadcastExchange (13)
                                             +- * Filter (12)
```

```
                                    +- * ColumnarToRow (11)
                                       +- Scan parquet (10)
```

In both cases the join of interest is at the step 22

8.1.6. Cartesian Product, Broadcast Hint and Broadcast Threshold without AQE

1. Hint on smaller table

```
    == Physical Plan ==
* HashAggregate (22)
+- Exchange (21)
   +- * HashAggregate (20)
      +- * Project (19)
         +- BroadcastNestedLoopjoin Inner BuildRight (18)
            :- * ColumnarToRow (2)
            :  +- Scan parquet (1)
            +- BroadcastExchange (17)
               +- Exchange (16)
                  +- * Project (15)
                     +- SortMergejoin LeftOuter (14)
                        :- * Sort (8)
                        :  +- Exchange (7)
                        :     +- * Project (6)
                        :        +- * Filter (5)
                        :           +- * ColumnarToRow (4)
                        :              +- Scan parquet (3)
                        +- * Sort (13)
                           +- Exchange (12)
                              +- * Filter (11)
                                 +- * ColumnarToRow (10)
                                    +- Scan parquet (9)
```

2. Hint on bigger table

```
    == Physical Plan ==
* HashAggregate (22)
+- Exchange (21)
   +- * HashAggregate (20)
      +- * Project (19)
         +- BroadcastNestedLoopjoin Inner BuildLeft (18)
            :- BroadcastExchange (3)
            :  +- * ColumnarToRow (2)
            :     +- Scan parquet (1)
            +- Exchange (17)
               +- * Project (16)
```

```
+- SortMergejoin LeftOuter (15)
   :- * Sort (9)
   :  +- Exchange (8)
   :     +- * Project (7)
   :        +- * Filter (6)
   :           +- * ColumnarToRow (5)
   :              +- Scan parquet (4)
   +- * Sort (14)
      +- Exchange (13)
         +- * Filter (12)
            +- * ColumnarToRow (11)
               +- Scan parquet (10)
```

In both cases the join of interest is at the step 18

8.2. Automatic Coalesce of partitions Results

8.2.1. Broadcast Hash Join automatic post shuffle repartition test results

1. With AQE:

- Initial plan:

```
+- == Initial Plan ==
   HashAggregate (unknown)
   +- HashAggregate (unknown)
      +- Project (unknown)
         +- BroadcastHashjoin LeftOuter BuildRight (unknown)
            :- Coalesce (unknown)
            :  +- Scan parquet (1)
            +- BroadcastExchange (unknown)
               +- Coalesce (unknown)
                  +- Filter (unknown)
                     +- Scan parquet (4)
```

- Final plan:

```
== Physical Plan ==
AdaptiveSparkPlan (14)
+- == Final Plan ==
   * HashAggregate (13)
   +- * HashAggregate (12)
      +- * Project (11)
         +- * BroadcastHashjoin LeftOuter BuildRight (10)
            :- Coalesce (3)
```

```
            :  +- * ColumnarToRow (2)
            :     +- Scan parquet (1)
            +- BroadcastQueryStage (9)
               +- BroadcastExchange (8)
                  +- Coalesce (7)
                     +- * Filter (6)
                        +- * ColumnarToRow (5)
                           +- Scan parquet (4)
```

2. Without AQE:

   Physical PLan:

```
== Physical Plan ==
* HashAggregate (12)
+- * HashAggregate (11)
   +- * Project (10)
      +- * BroadcastHashjoin LeftOuter BuildRight (9)
         :- Coalesce (3)
         :  +- * ColumnarToRow (2)
         :     +- Scan parquet (1)
         +- BroadcastExchange (8)
            +- Coalesce (7)
               +- * Filter (6)
                  +- * ColumnarToRow (5)
                     +- Scan parquet (4)
```

8.2.2. Sort Merge Join automatic post shuffle repartition test results

1. With AQE:

   - Initial plan:

```
+- == Initial Plan ==
   HashAggregate (unknown)
   +- HashAggregate (unknown)
      +- Project (unknown)
         +- SortMergejoin LeftOuter (unknown)
            :- Sort (unknown)
            :  +- Coalesce (unknown)
            :     +- Scan parquet (1)
            +- Sort (unknown)
               +- Coalesce (unknown)
                  +- Filter (unknown)
                     +- Scan parquet (5)
```

- Final Plan:

```
== Physical Plan ==
AdaptiveSparkPlan (14)
+- == Final Plan ==
   * HashAggregate (13)
   +- * HashAggregate (12)
      +- * Project (11)
         +- SortMergejoin LeftOuter (10)
            :- * Sort (4)
            :  +- Coalesce (3)
            :     +- * ColumnarToRow (2)
            :        +- Scan parquet (1)
            +- * Sort (9)
               +- Coalesce (8)
                  +- * Filter (7)
                     +- * ColumnarToRow (6)
                        +- Scan parquet (5)
```

2. Without AQE:

   Physical PLan:

```
== Physical Plan ==
* HashAggregate (13)
+- * HashAggregate (12)
   +- * Project (11)
      +- SortMergejoin LeftOuter (10)
         :- * Sort (4)
         :  +- Coalesce (3)
         :     +- * ColumnarToRow (2)
         :        +- Scan parquet (1)
         +- * Sort (9)
            +- Coalesce (8)
               +- * Filter (7)
                  +- * ColumnarToRow (6)
                     +- Scan parquet (5)
```

8.3. Skew Join Handling Results

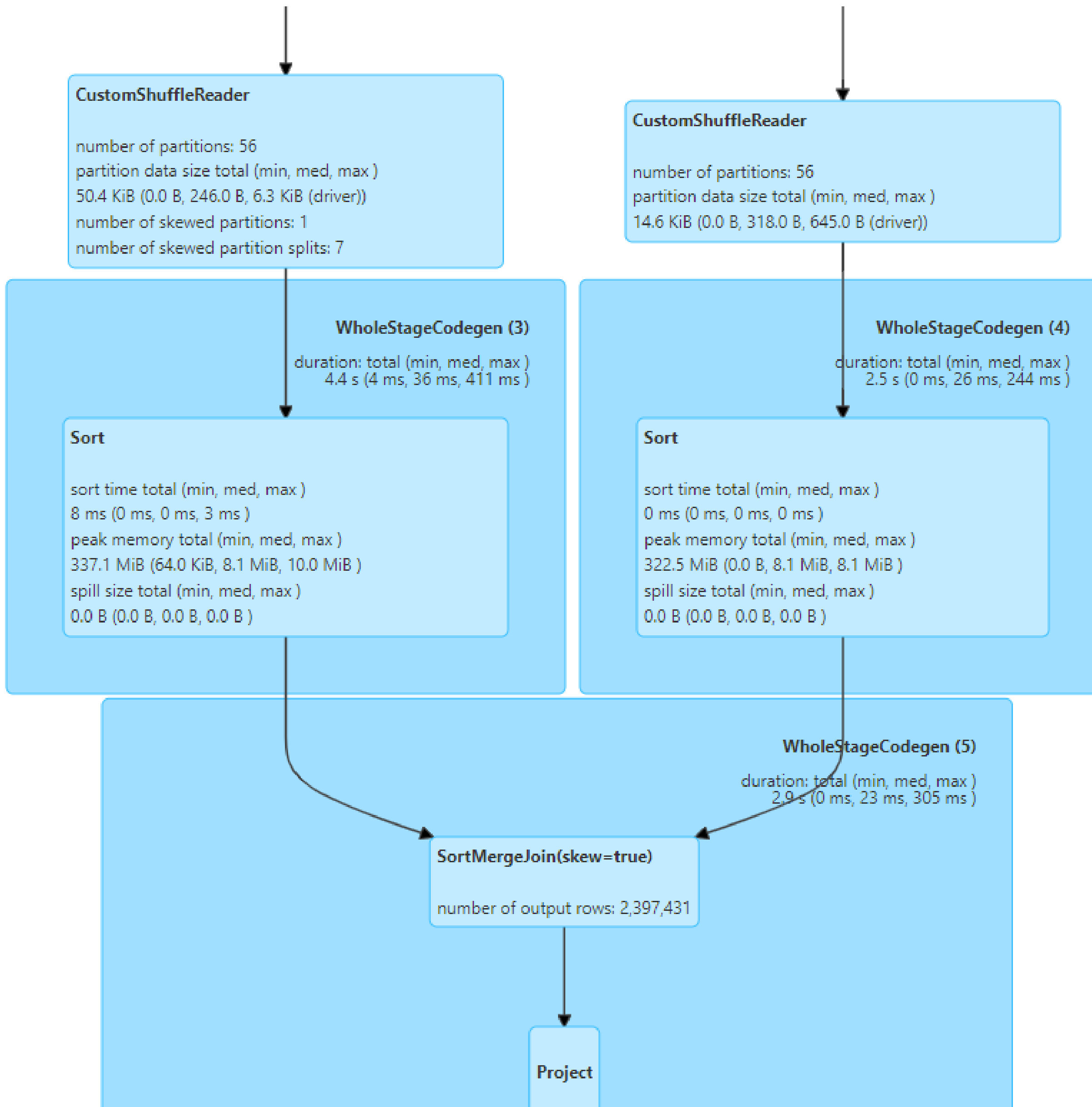

Figure 8.1: Test join between skewReduced and notSkewReduced

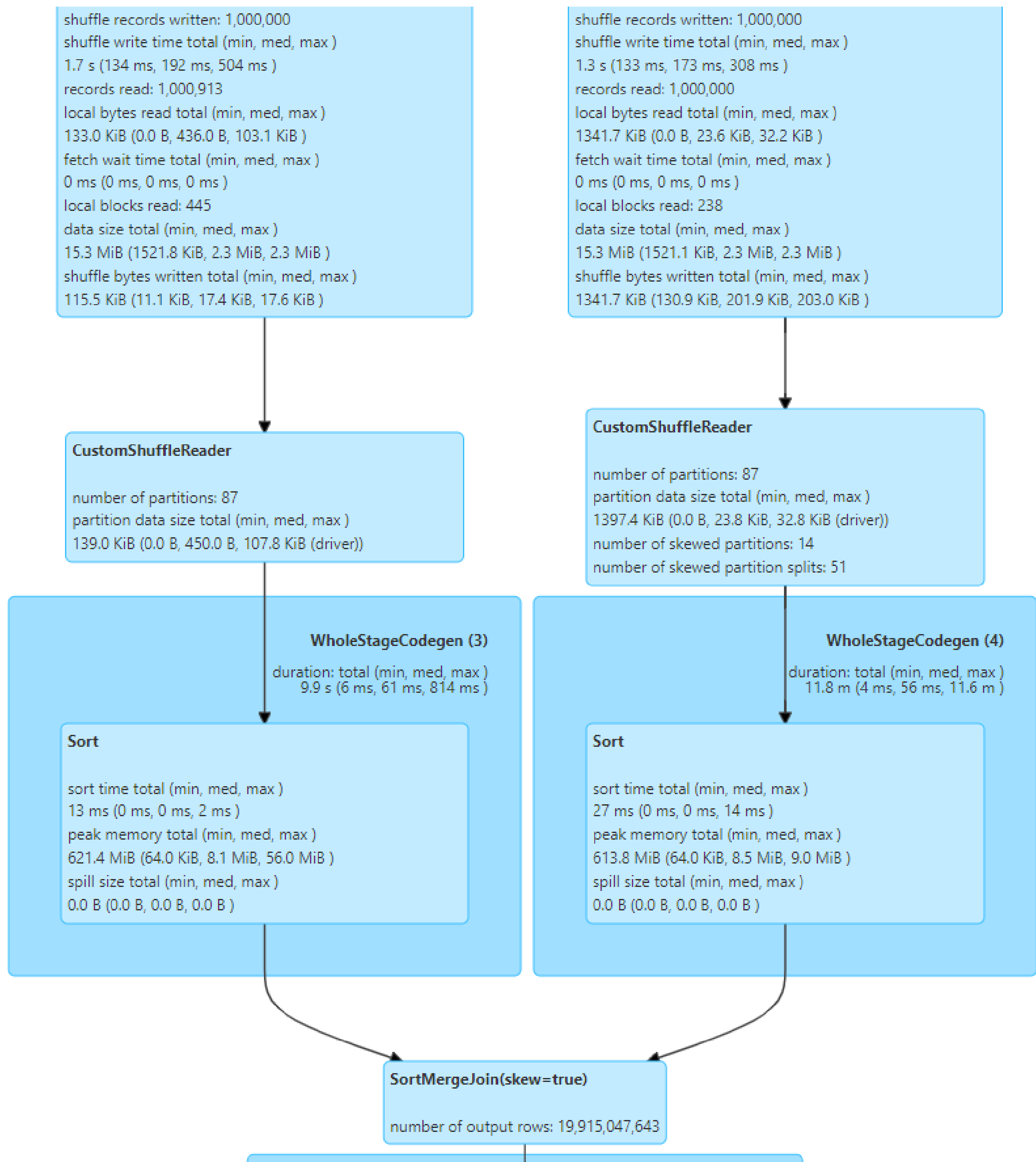

Figure 8.2: DAG for skewed Sort Merge join, AQE enabled, Right join

Duration	GC Time	Shuffle Write Size / Records	Shuffle Read Size / Records
12 min	0.4 s	59 B / 1	105.3 KiB / 1019404
0.7 s		59 B / 1	2.7 KiB / 20232
0.6 s		59 B / 1	2.7 KiB / 19981
0.6 s	0.4 s	59 B / 1	31.6 KiB / 12195
0.5 s	0.4 s	59 B / 1	31.5 KiB / 12119
0.5 s	0.4 s	59 B / 1	24.6 KiB / 12137
0.5 s	0.4 s	59 B / 1	31.9 KiB / 15894
0.5 s	0.4 s	59 B / 1	2.5 KiB / 20135
0.5 s		59 B / 1	24.1 KiB / 9010
0.5 s		59 B / 1	23.5 KiB / 8900
0.5 s	0.4 s	59 B / 1	15.8 KiB / 6003
0.5 s	0.4 s	59 B / 1	26 KiB / 9910
0.5 s		59 B / 1	24.4 KiB / 12142
0.4 s		59 B / 1	32 KiB / 16034
0.4 s		59 B / 1	24.5 KiB / 12077
0.4 s		59 B / 1	24.3 KiB / 8917
0.2 s		59 B / 1	2.6 KiB / 19975
0.2 s		59 B / 1	24.4 KiB / 11967
0.2 s		59 B / 1	24.4 KiB / 12037
0.2 s		59 B / 1	2.6 KiB / 20010

Figure 8.3: Tasks duration of the 20 longest tasks for skewed Sort Merge join, AQE enabled, Right join

Summary Metrics for 87 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	19.0 ms	91.0 ms	0.1 s	0.2 s	12 min
GC Time	0.0 ms	0.0 ms	0.0 ms	0.0 ms	0.4 s
Shuffle Read Size / Records	0.0 B / 0	2.6 KiB / 8989	24.1 KiB / 12077	24.6 KiB / 16035	105.3 KiB / 1019404
Shuffle Write Size / Records	56 B / 1	59 B / 1	59 B / 1	59 B / 1	59 B / 1

Figure 8.4: Tasks statistics for skewed Sort Merge join, AQE enabled, Right join

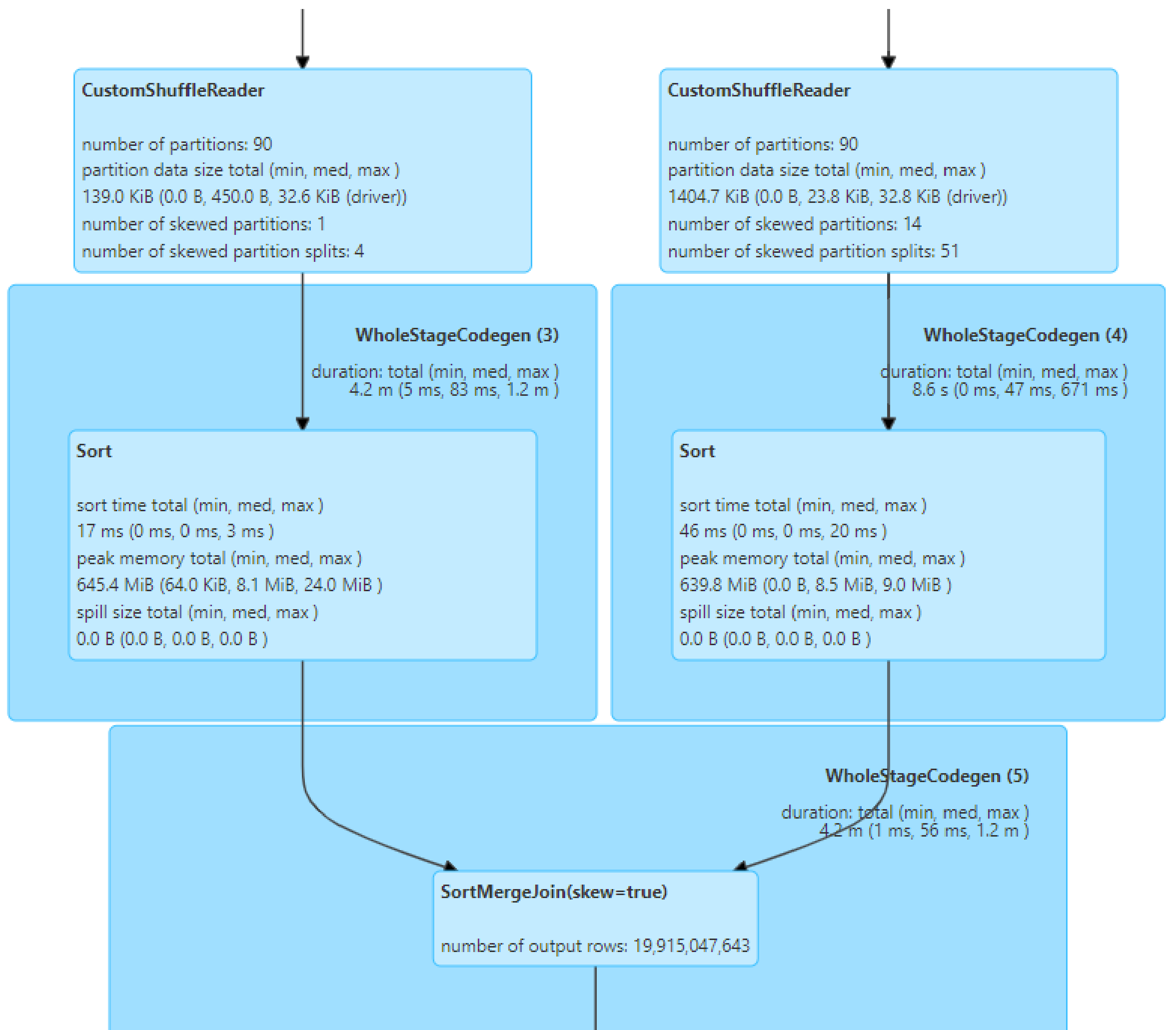

Figure 8.5: DAG for skewed Sort Merge join, AQE enabled, Inner join

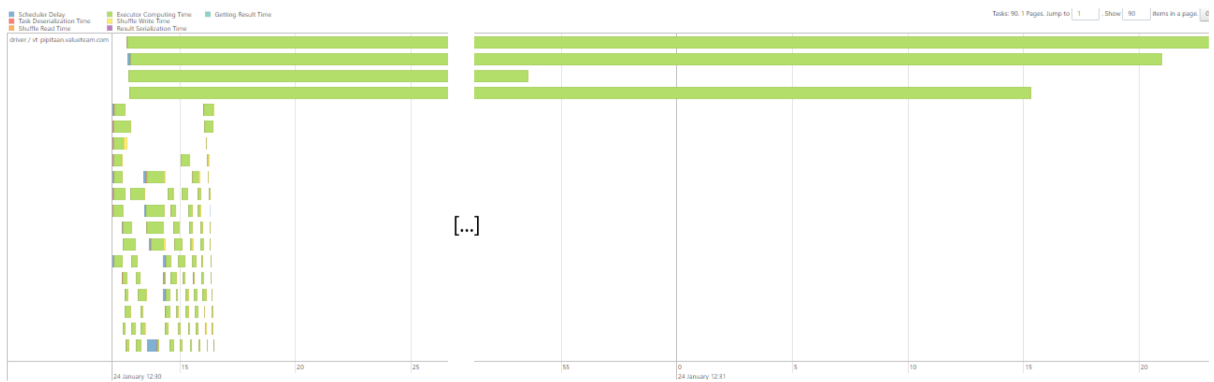

Figure 8.6: Tasks duration for skewed Sort Merge join, AQE enabled, Inner join

Summary Metrics for 90 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	18.0 ms	70.0 ms	0.1 s	0.2 s	53 s
GC Time	0.0 ms	0.0 ms	0.0 ms	0.0 ms	0.1 s
Shuffle Read Size / Records	0.0 B / 0	2.6 KiB / 9010	24.1 KiB / 12119	24.7 KiB / 19846	33.3 KiB / 320629
Shuffle Write Size / Records	56 B / 1	59 B / 1	59 B / 1	59 B / 1	59 B / 1

Figure 8.7: Tasks statistics for skewed Sort Merge join, AQE enabled, Inner join

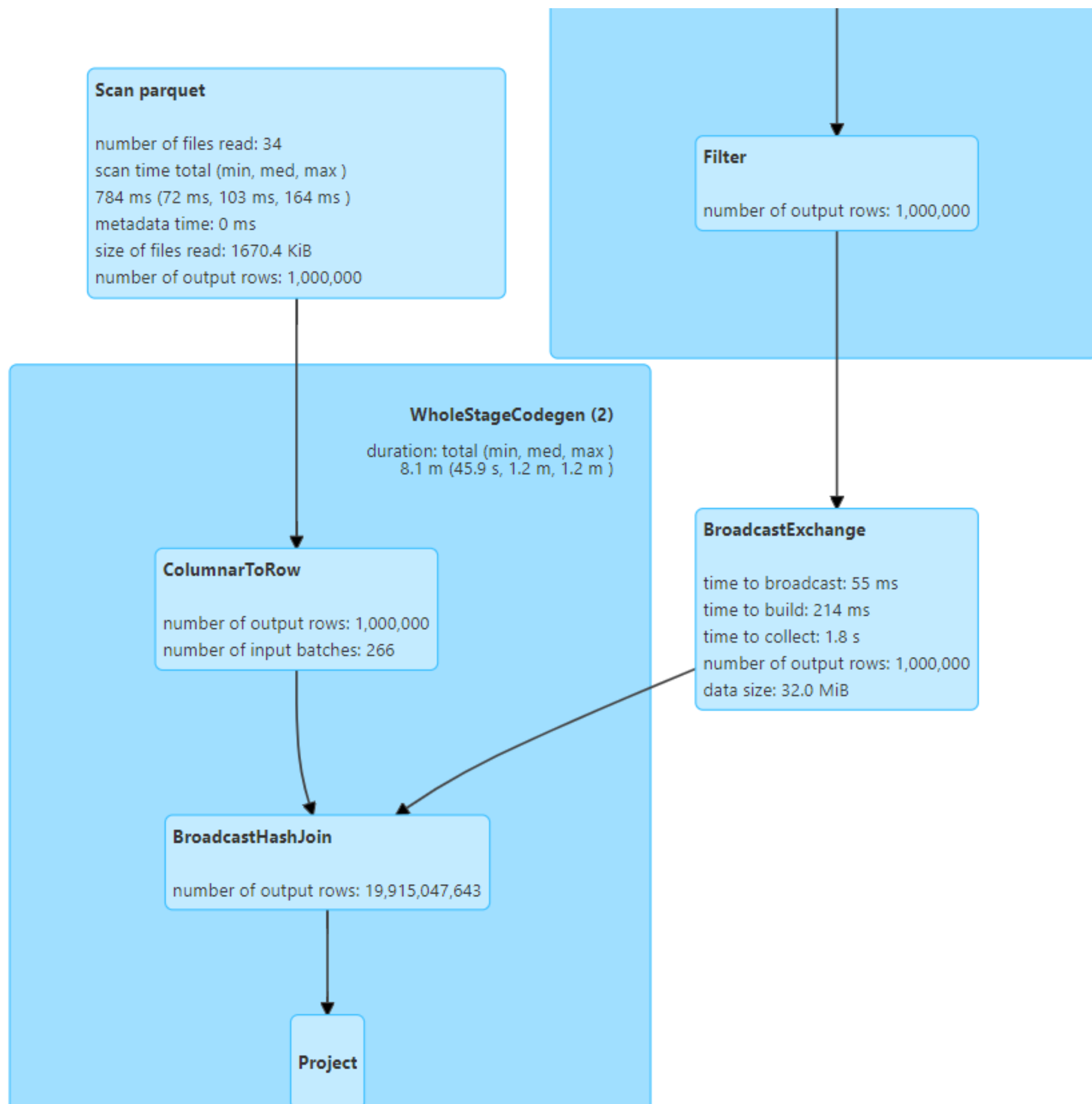

Figure 8.8: DAG of the Broadcast Hash join strategy on skewed data with AQE disabled

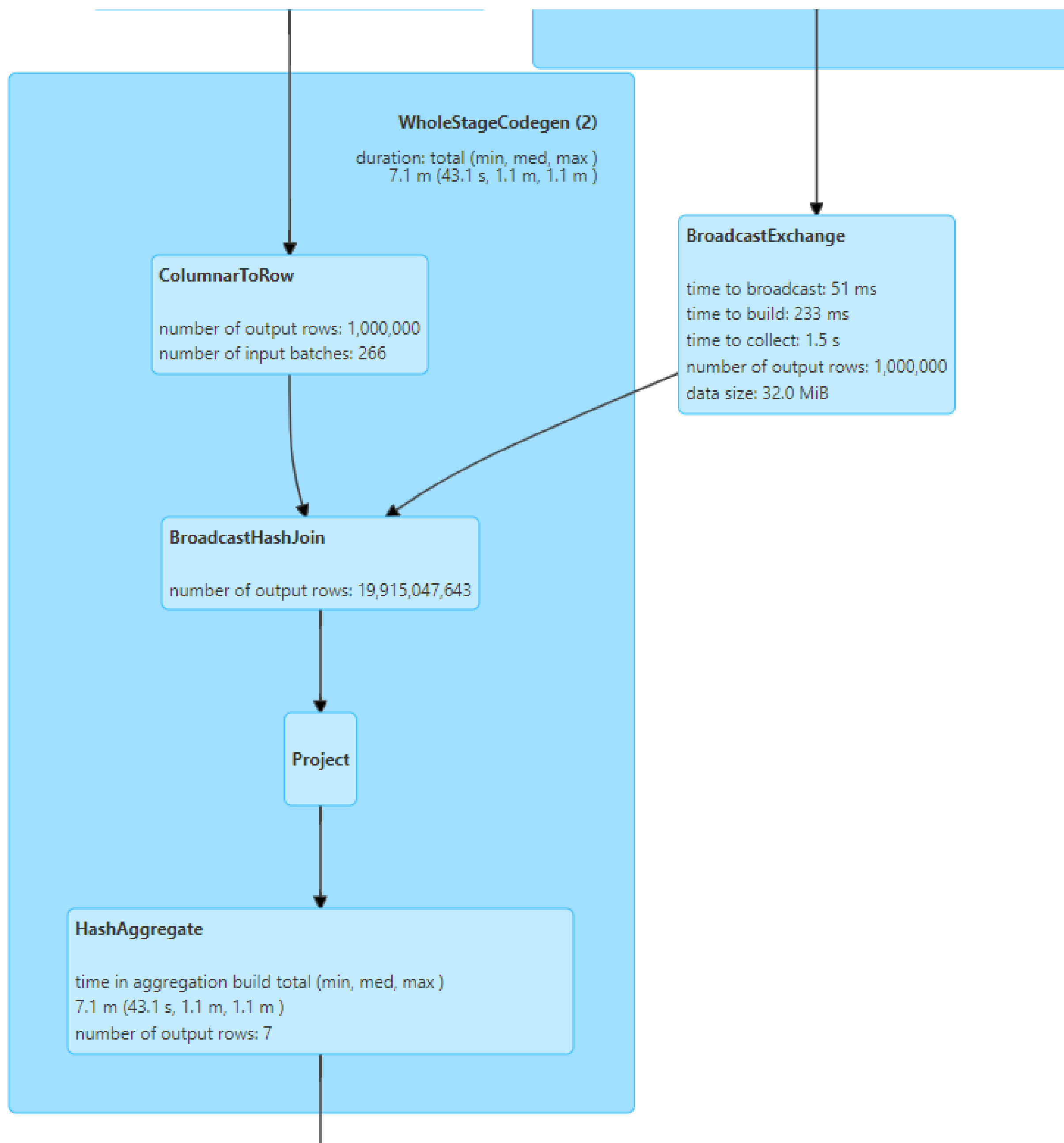

Figure 8.9: DAG of the Broadcast Hash join strategy on skewed data with AQE enabled

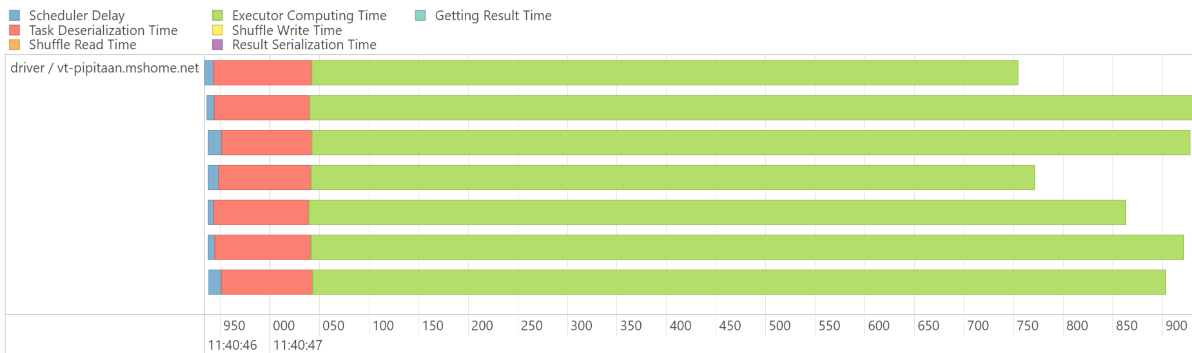

Figure 8.10: Tasks duration with skewed data in the Broadcast Hash join strategy, AQE enabled

Figure 8.11: Tasks duration with skewed data in the Broadcast Hash join strategy, AQE disabled

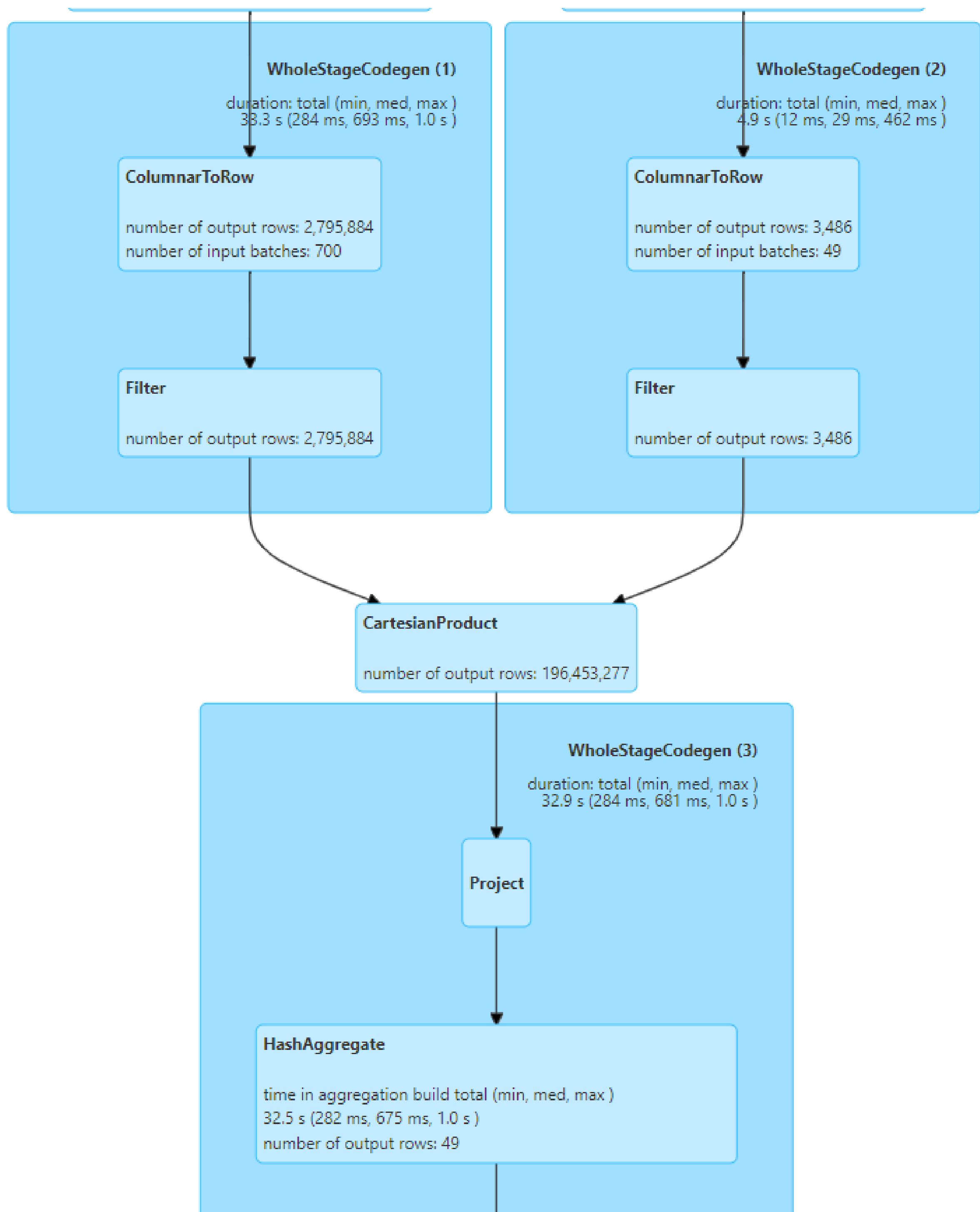

Figure 8.12: DAG of the Cartesian Product join strategy on skewed data with AQE disabled

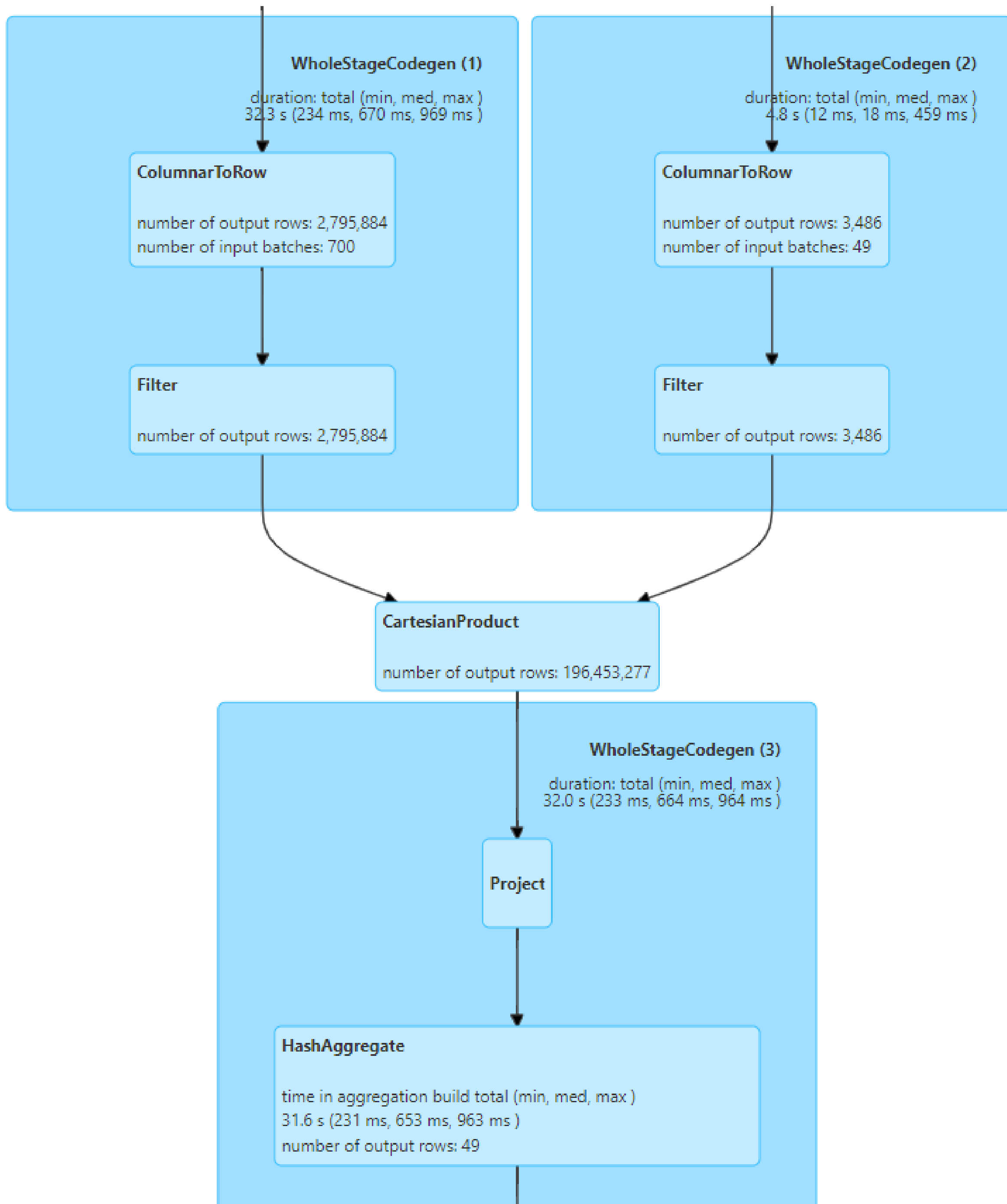

Figure 8.13: DAG of the Cartesian Product join strategy on skewed data with AQE enabled

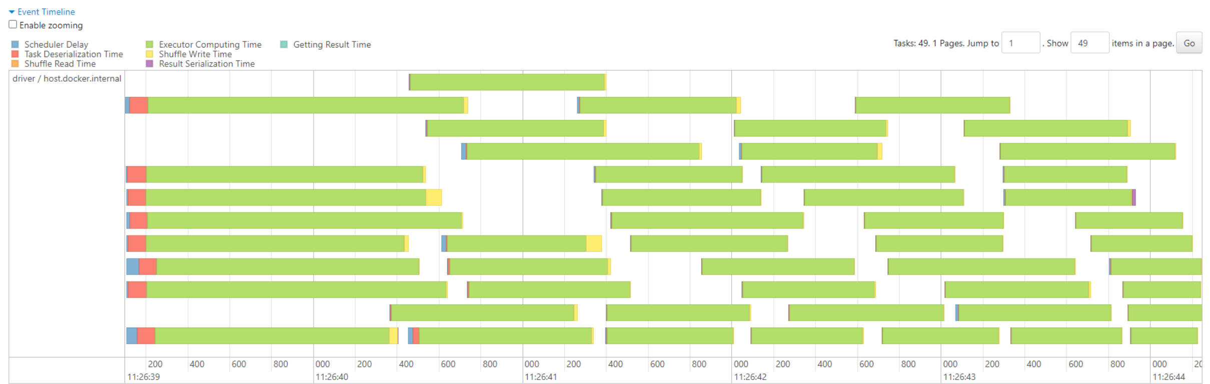

Figure 8.14: Tasks duration with skewed data in the Cartesian Product join strategy, AQE disabled

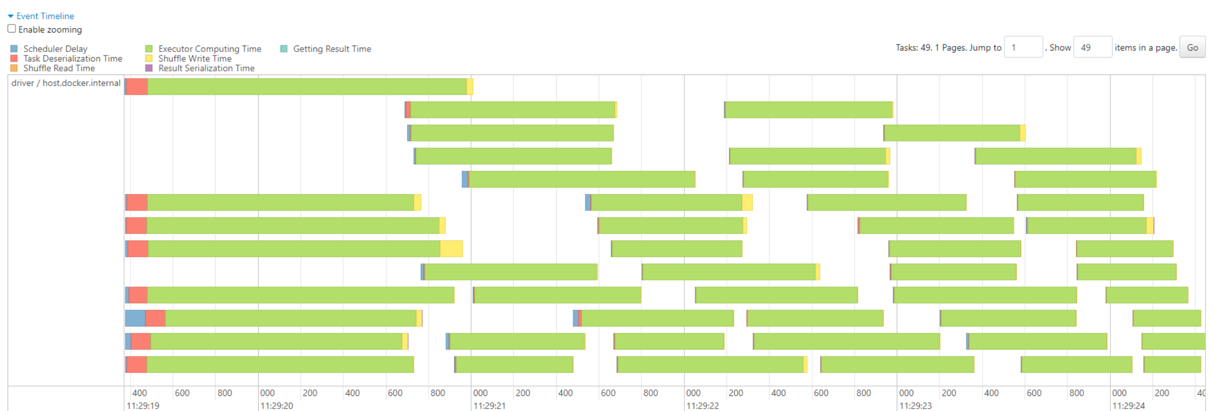

Figure 8.15: Tasks duration with skewed data in the Cartesian Product join strategy, AQE enabled

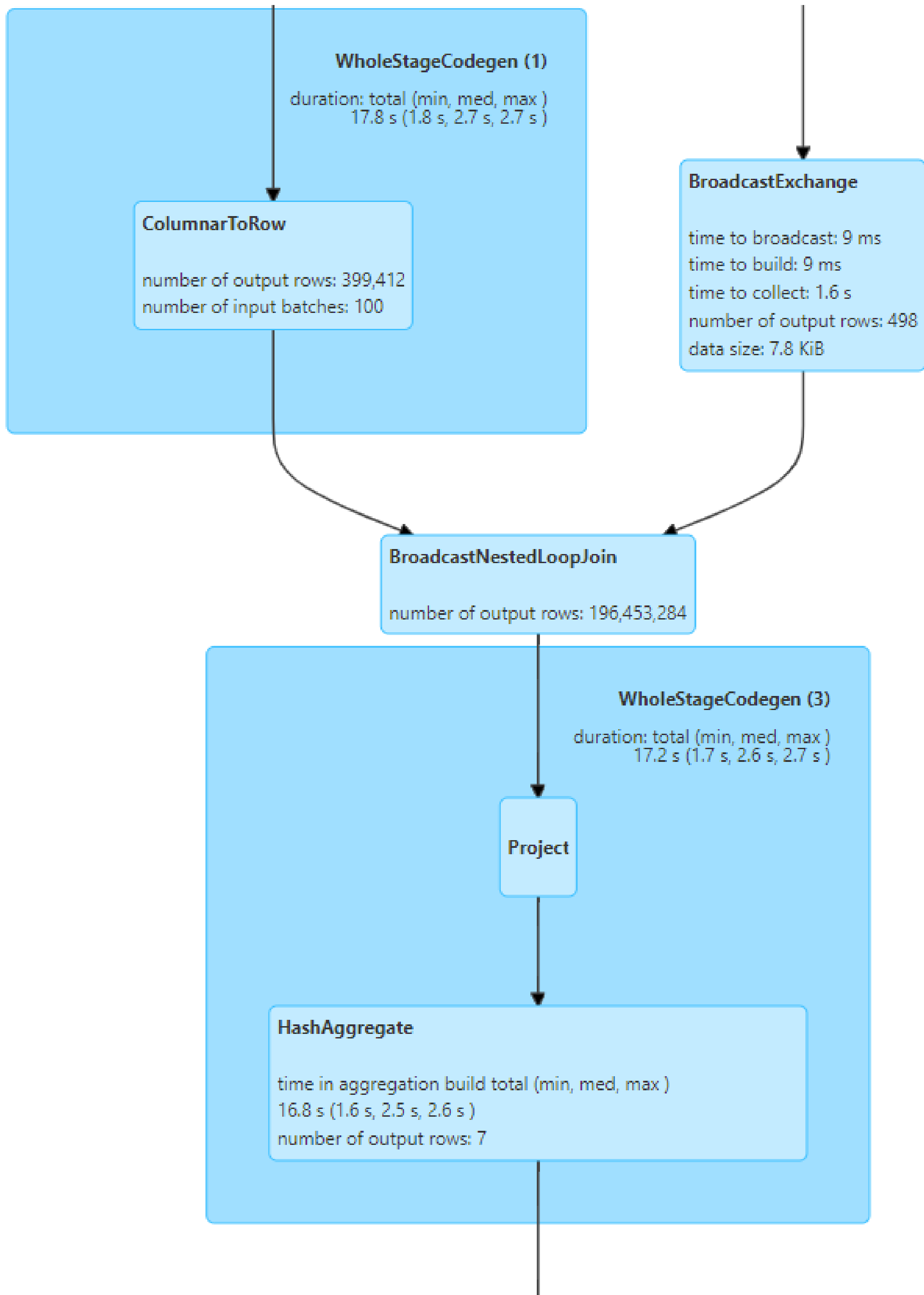

Figure 8.16: DAG of the Broadcast Nested Loop join strategy on skewed data with AQE disabled

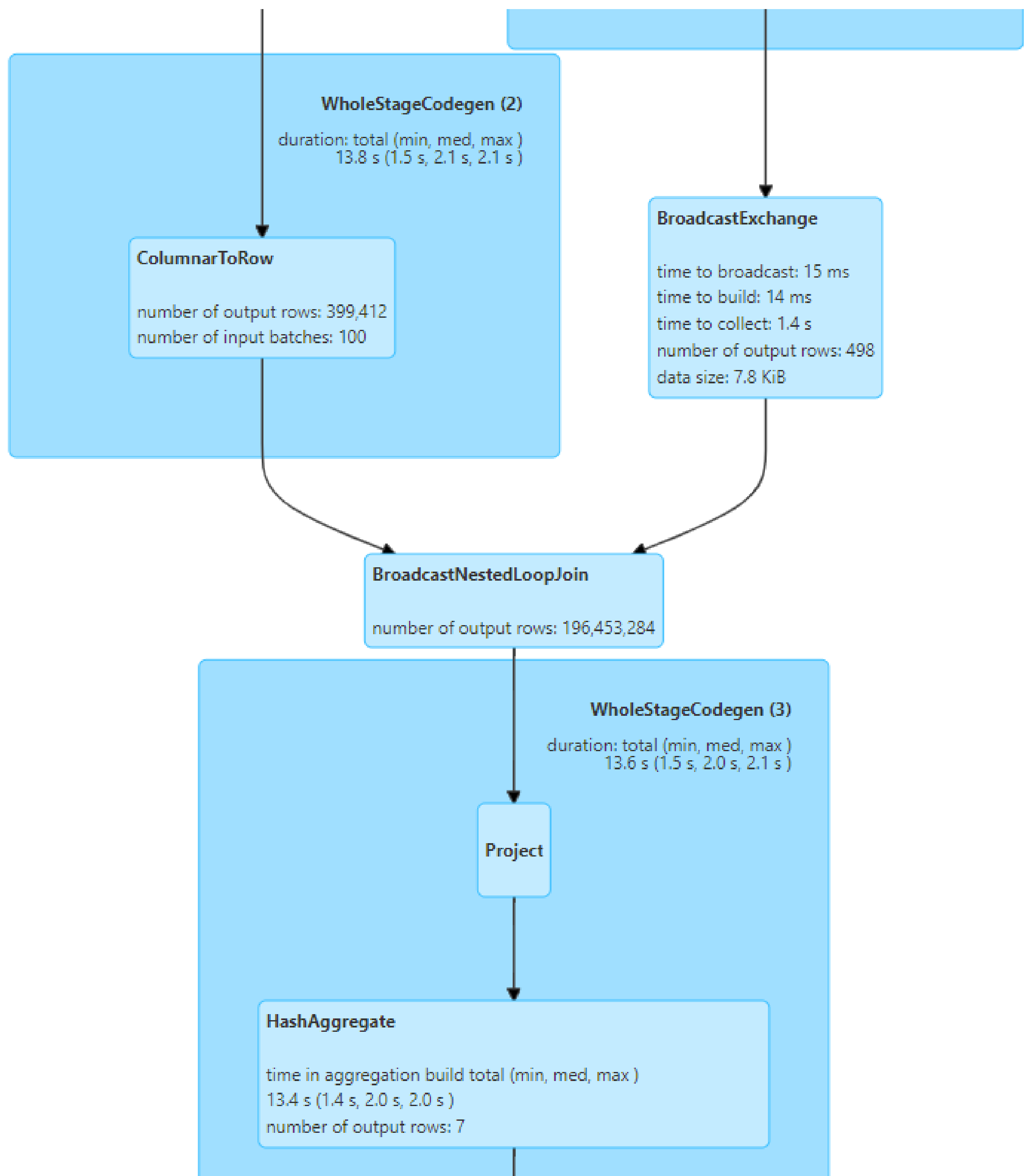

Figure 8.17: DAG of the Broadcast Nested Loop join strategy on skewed data with AQE enabled

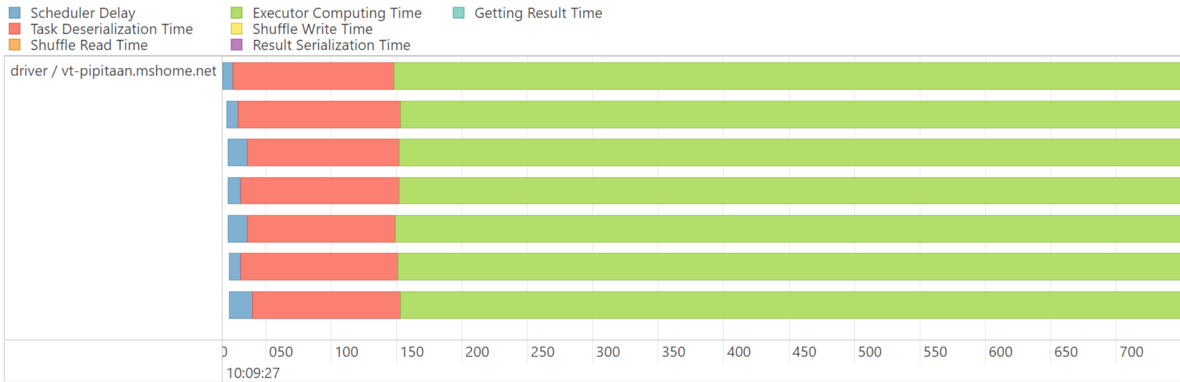

Figure 8.18: Tasks duration with skewed data in the Broadcast Nested Loop join strategy, AQE disabled

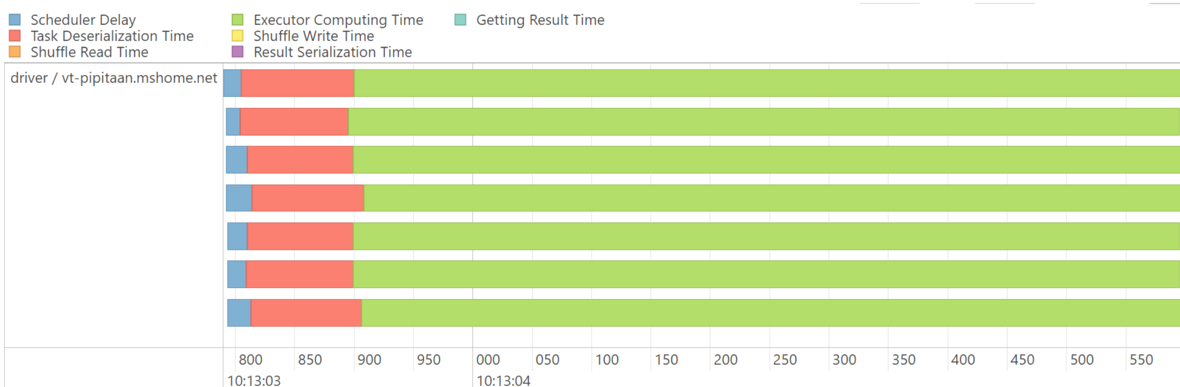

Figure 8.19: Tasks duration with skewed data in the Broadcast Nested Loop join strategy, AQE enabled

8.4. Data From Delta Logs

Metric	Percentage variation at week one	Percentage variation at week two
Files added	-79.28%	-78.57%
Files deleted	-53.58%	-65.36%
Rewrite time	-6.08%	-2.06%
Scan time	7.75%	4.12%
Rows updated	-64.05%	-24.05%
Rows inserted	43.25%	14.79%
Source rows	-44.73%	-17.61%
Rows copied	1.17%	6.56%
Output rows	1.00%	6.48%

Table 8.1: Percentage variations from delta logs between the two weeks without AQE and the two experiments performed

Acknowledgements

I would like to thank Professor Emanuele Della Valle, advisor on this thesis, for the information and knowledge provided both during the Unstructured and Streaming Data Engineering course and the work on this thesis. I would also like to thank Andrea Picasso Ratto, co-advisor on this thesis, for the invaluable insights provided on Spark as a whole and, more generally, on the Data Engineering field; as well as for the invaluable support provided to this thesis. I am also grateful to Alberto Oreste Dell'era, for the insights provided regarding query optimizers.

I would also like to show gratitude to Corrado Palumbi and all my colleagues at NTT Data for the incredible support I received while working on this thesis.

I also thank my friends and my family for their unwavering support, both during the work on this thesis and during my studies as a whole.