Systems and Architectures for Big Data - A.A. 2023/24
Project 1: Analyzing monitoring data with Apache Spark*

Emanuele Valzano
University of Rome Tor Vergata
Rome, Italy
emanuele.valzano@gmail.com

Giulio Appetito
University of Rome Tor Vergata
Rome, Italy
giulio.appetito@alumni.uniroma2.eu

Anastasia Brinati
University of Rome Tor Vergata
Rome, Italy
anastasia.brinati@alumni.uniroma2.eu

***Abstract*—The aim of the document is to illustrate the ideas behind the development of a batch processing pipeline and the reasoning on some architectural and analytical aspects.**

I. Introduction

The project involves batch processing on a dataset of real-world telemetry data for hard drive failures, provided by Backblaze and used for DEBS 2024 GC, using Apache Spark as the data processing framework to answer some queries. We are considering a reduced version of the dataset consisting of ∼600MB (∼3M events).
The dataset is about hard disk drives failures, since these are the most frequently replaced hardware components of data centers and the main reason behind server failures. Each tuple includes S.M.A.R.T. telemetry data and some additional attributes added by Backblaze.

II. System Architecture

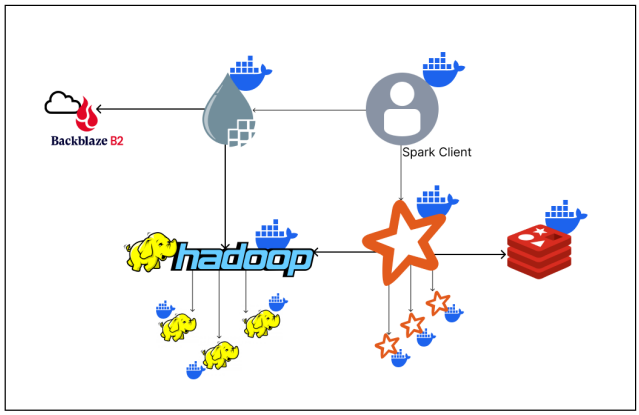


Fig. 1. System architecture scheme

The system architecture is emulated using Docker Compose on a single node. A bridge network facilitates communication between the containers, ensuring seamless data flow across components. The architecture comprises six main components:

- **Spark Client**
  The Spark Client orchestrates the entire workflow. It schedules HDFS ingestion via NiFi and triggers both preprocessing and processing tasks. Once results are computed, they are stored in HDFS and subsequently exported to Redis. The client leverages three key libraries: PySpark, primarily for data processing and for reading and writing results to HDFS; NiPyAPI, for scheduling the NiFi processor group, and the Redis library, to put results in the key-value data store.
- **Apache NiFi**
  Apache NiFi handles data ingestion, configured to pull the dataset from either a local directory or a Backblaze cloud storage bucket. It ingests the dataset into HDFS in both Avro and Parquet formats, enabling efficient storage and retrieval.
- **Apache Hadoop**
  Hadoop serves as the distributed file system for data storage and retrieval. The original dataset is ingested into HDFS in Parquet and Avro formats, only with the column of interest. Additionally, query results are stored in HDFS in CSV format. The default HDFS setup includes one Namenode and one Datanode to optimize resource allocation for Spark workers. However, Datanodes can be dynamically scaled after system startup to accommodate varying workload demands. For our purposes we just used one datanote, because we decided to allocate out additional machine resources to the Spark cluster.
- **Apache Spark**
  Apache Spark is utilized for both preprocessing and processing data. During preprocessing, the DataFrame API is used to validate the dataset by removing invalid entries and casting values to the proper data type. For processing, both RDD and Spark SQL are employed to evaluate and compare query execution times. The Spark

cluster is composed of a master and one or more workers. Spark workers can be scaled dynamically, perhaps we used different configurations during processing.

- **Redis**
  Redis functions as a key-value store for query results. The Spark Client exports these results from HDFS and writes them to Redis in JSON format, using the query name as the key. This approach enables efficient and straightforward retrieval of the data.
- **B2 Cloud Storage**
  We utilize Backblaze B2 Cloud Storage, which comes to rescue when the architecture is deployed with a remote dataset acquisition configuration. An encrypted version of the dataset resides in a B2 bucket. During the ingestion step, NiFi is responsible for acquiring the dataset from the B2 bucket, performing the necessary operations, and then ingesting the results into HDFS.

III. Pipeline

The data processing pipeline initiates with *data ingestion* using Apache NiFi, which ensures the transfer of data into the Hadoop Distributed File System (HDFS). The dataset is ingested with only the column of interest. Once stored in HDFS, the data undergoes a more fine-grained *pre-processing* using Apache Spark. This pre-processing phase includes tasks such as removing null values, deduplication, type casting, and filtering out invalid rows. The pre-processed data is then stored back in HDFS for checkpointing and will be used for the subsequent processing.
Following pre-processing, the data is subjected to query processing, with the results stored in HDFS and exported to Redis for quick access. These steps are intentionally separated to enable flexible execution of individual stages. For instance, after the initial ingestion, this step does not need to be repeated for each client execution. Clients have the flexibility to opt for either pre-processing or proceed directly to processing, depending on whether the data has already been pre-processed. Furthermore, clients can choose to export results to Redis or not. Additionally, results can be written to the local machine if necessary.

A. Data Ingestion

The ”HDFS_INGESTION” NiFi flow is designed and structured with various processors, each performing specific roles to ensure correct data handling.

- **Configuration Extraction and Source Evaluation**
  The initial processors are responsible for extracting configuration information and determining the source of the dataset. In fact, the process begins with the ExtractConf processor, an instance of the EvaluateJsonPath processor type. This processor is responsible for extracting configuration details from an incoming JSON file, which determines whether the dataset is to be acquired from the local container or remotely from a Backblaze bucket. The configuration

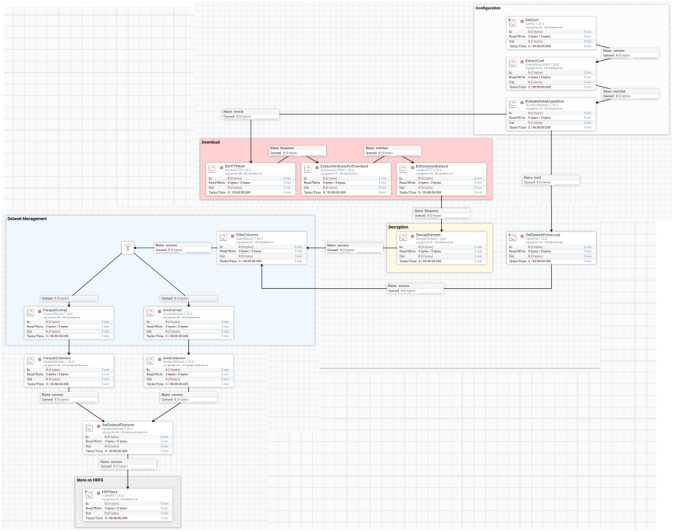


Fig. 2. NiFi flow

file is structured as follows:

```
{
    "acquisition": "local_or_remote",
    "datasetPath": "local_path",
    "b2": {
        "authKey": "base64_key",
        "bucketName": "bucket_name",
        "filename": "dataset_name"
}
}
```

The extracted values are stored as attributes in the flowfile, which are essential for the subsequent data download process. This structured approach ensures that the data ingestion phase is flexible and can adapt to different data sources.
Following the configuration extraction, the flowfile is evaluated and the branch goes to the local fetch or remote download.

- **Local or Remote Dataset Fetching**

  Local Fetching The dataset is fetched using the FetchFile processor if the configuration specifies a local source.

  Remote Downloading If the dataset is to be downloaded from a remote Backblaze bucket, two InvokeHTTP processors are used: The first InvokeHTTP processor performs a GET request to `/b2api/v3/b2_authorize_account`, with the authorization header set to the authorization key, to retrieve the URL and the authentication token needed for the subsequent download.
  The second InvokeHTTP processor uses the retrieved

URL and authentication token to perform the actual download of the dataset.

- **Decryption of the Retrieved Dataset**
  For security reasons, the downloaded dataset is encrypted using the AES-128 GCM algorithm. The EncryptContent processor handles the decryption, with its mode property set to "Decrypt" and a sensitive processor key is provided for the decryption process.

- **Dataset Filtering**
  Once decrypted (or fetched locally), the plaintext dataset is sent to the QueryRecord processor. This processor filters unnecessary columns by executing the following SQL statement:

```
SELECT
    "date", serial_number,
    model, failure,
    vault_id, s9_power_on_hours
    FROM FLOWFILE
```

  This step ensures that only the columns of interest are retained.

- **Dataset Formatting**
  The resulting flowfile is then sent to two different ConvertRecord processors through a funnel, these are responsible for converting the filtered CSV dataset into two formats: AVRO and PARQUET. These formats are chosen for their efficiency in storage and retrieval:
  - **AVRO**: Known for its compact, fast, and schema-based serialization.
  - **PARQUET**: Optimized for read-heavy operations, offering efficient data compression and encoding.

  To assess performance, we opted to employ both column-oriented and row-oriented formats.

- **Storage on HDFS**
  After formatting, the processors UpdateAttribute adjust the filename properties to correctly identify the datasets. The final flowfiles, representing the dataset in AVRO and PARQUET formats, are then stored on HDFS using the PutHDFS processor, configured with necessary Hadoop settings from 'hdfs-site.xml' and 'core-site.xml' files.

Error handling is also managed, to avoid the disruption of the entire flow. All processors are configured with a TIMER_DRIVEN scheduling strategy and an initial scheduling period of 0 seconds. This setup enables them to process flowfiles immediately upon receipt, ensuring minimal latency. However, it does impose resource demands. Additionally, the execution node is set to ALL, allowing the processors to execute on any node within a NiFi cluster.
In summary, the "HDFS_INGESTION" NiFi flow is a comprehensive data pipeline that consists of extracting necessary configuration details, downloading the dataset from a local or remote source, decrypting if necessary, filtering unnecessary columns, converting them into the required formats, and finally storing computed results in the HDFS.

B. Pre-processing

Although NiFi conducted pre-processing by filtering unnecessary columns, it did not validate the dataset by removing invalid rows or performing useful type conversions. In such scenarios, Spark emerges as an excellent tool for comprehensive data validation and transformation.
The process begins with the removal of rows containing NaN values to uphold data cleanliness and integrity. We operate under the assumption that each value within the column of interest significantly contributes to the representativeness of the entry, since each represents a log message. Moreover, the fraction of incomplete entries was negligible when compared to the dataset size.
Next, duplicate rows are eliminated to avoid redundancy and ensure the uniqueness of records.
All columns of interest are then selected and casted to the appropriate data type, ensuring proper data processing. The casting was made using the datatypes offered by the spark.sql.type module.

- 'date' has been casted to *DateType()*
- *'serial_number' to StringType()*
- *'model' to StringType()*
- *'failure' to BooleanType()*
- *'vault_id' to IntegerType()*
- *'s9_power_on_hours' to DoubleType()*

Lastly, invalid rows where the serial_number and model don't comply to the right regex are filtered out to avoid processing of invalid hard disk events.

- Serial Number Regex:
  `^[A-Z0-9_-]+$`
  This pattern ensures that the 'serial_number' column contains only alphanumeric characters (uppercase), underscores, and hyphens. At least one character must be present.
- Model Regex:
  `^[A-Z0-9 ]+$`
  This pattern ensures that the 'model' column contains only alphanumeric characters (uppercase) and white spaces. There must be at least one character.

Finally, the column 'date' is renamed to 'event_date' since 'date' is an SQL keyword.
After the above steps, the preprocessed DataFrame is saved on HDFS in AVRO or PARQUET format (depending on the format read by the client prior to preprocessing, based on the client's execution parameters), so it can be later analyzed. This stage also represents an important checkpointing step for the dataflow.

C. Processing: Queries

Prior to query execution, the preprocessed dataset is read by HDFS as a DataFrame in AVRO or PARQUET format.

The retrieved DataFrame, along with the underlying RDD, is persisted using the *persist()* method. To materialize this persistence, two Spark actions have been called (on both RDD and DataFrame). This persistance enables Spark workers to efficiently access both structures during query execution. Lastly, we created a temporary view on the DataFrame called 'DisksMonitor' for SQL queries.
It is important to note that these steps are not part of the actual query evaluation. Our objective is to perform the actions that trigger a Spark job during query execution as efficiently as possible. In support of this, we introduced an abstraction for query results called *QueryResult*. This abstraction is essentially a list of objects, each representing the result of a Spark action. Each object, termed *SparkActionResult*, contains the result of an action (e.g., *collect()* or *take()*).
By using this approach, each query returns a common interface, allowing results to be easily evaluated and clearly presented. The evaluation of a query involves measuring the time taken to execute each Spark action within the query. Each execution time is stored in a *SparkActionResult* object, which is then encapsulated within a *QueryResult* object. To determine the total execution time of a query, we sum up all the execution times stored in each element of the *QueryResult* list.
For instance, Query 1 and Query 3 each return a *QueryResult* with a single *SparkActionResult*, as they trigger only one job using the *collect()* method. In contrast, Query 2 performs two jobs using the *take()* method, thus returning a *QueryResult* with two *SparkActionResults*.

Query 1. *For each day, for each vault (referencing the vault id field), calculate the total number of failures. Determine the list of vaults that have experienced exactly 4, 3, and 2 failures.*

***Q1 with RDD**:* We decided to apply the following steps:

- First, a preliminary ***filter*** function is applied to reduce the data size, keeping only the elements where the fourth field (failure) is true.
- Next, each element is converted into a key-value pair with the ***map*** function, where the key is a tuple consisting of the event date and the vault ID (event_date, vault_id), and the value is 1, representing one failure.
- The values are then summed up for each pair (event_date, vault_id) using the ***reduceByKey*** function, so the resulting values are the failure counts.
- After the reduction, another ***filter*** function is applied to retain only the pairs where the failure count is 2, 3, or 4.
- Finally, the data is reorganized with another ***map*** function in the following format (event_date, vault_id, failures_count).

***Q1 with Spark SQL**:* For the Spark SQL version of Q1, we wrote a single SQL query consisting of the following steps on the 'DisksMonitor' "table":

- Once the data is read by the table, the rows are filtered keeping only those WHERE the failure column equals 1. This reduces the dataset to include only relevant records.
- Then, a SELECT clause is applied to the columns of interest event_date, vault_id, and the count of the failure column, which is aliased as failures_count.

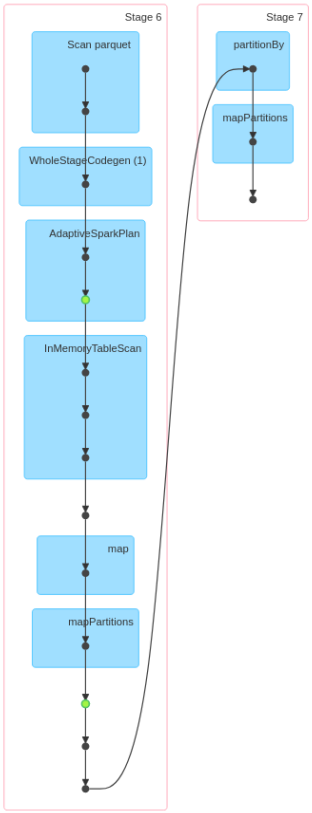


Fig. 3. DAG of Q1 with RDD

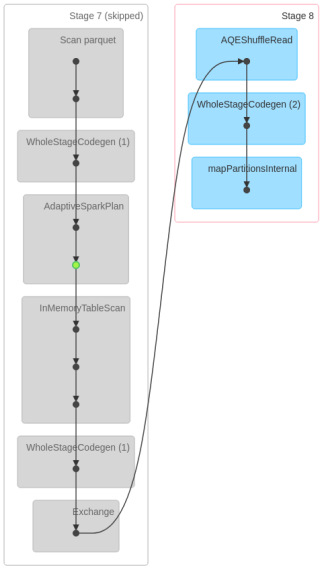


Fig. 4. DAG of Q1 with DataFrame

- The data is then organized by event_date and vault_id with a GROUP BY clause, so that the COUNT of failures is computed for each unique combination of these two fields.
- The HAVING clause is applied to filter the grouped results, retaining only those where the failures_count is exactly 2, 3, or 4.

The final result is a DataFrame containing the event_date, vault_id, and the corresponding failures_count for the specified conditions.

After performing transformations on both RDD and DataFrame in Spark SQL, a job was triggered to obtain the query result using the collect method for both RDD and DataFrame. Subsequently, the results and evaluations were saved in the QueryResult object. The results of both the query with RDD and Spark SQL are identical. This consistency gives us confidence in the correctness of the query operations.

Query 2. *Calculate the ranking of the 10 hard disk models that have suffered the greatest number of failures. The ranking must report the hard disk model and the total number of failures suffered by the hard disks of that specific model. Next, calculate a second ranking of the 10 vaults that recorded the most failures. For each vault, report the number of failures and the list (without repetitions) of hard disk models subject to at least one failure.*

Q2 with RDD*:* We opted to process the RDD by initially computing an intermediate RDD, termed *partial_rdd*. This serves as the foundational element for two additional RDDs: one pertaining to the "first" part of the query concerning the models, and the other concerning the vaults. Essentially, the partial RDD functions as a parent node in a tree structure, with two leaf nodes representing the other RDDs. This partial RDD is computed following these steps:

- First, a ***filter*** is applied to the RDD in order to reduce the data size, keeping only the elements where the failure field is *true*.
- Next, a ***map*** is applied to convert each element into a key-value pair *((vault_id, model), 1)*.
- Finally, a ***reduceByKey*** is applied in order to sum the failures for each (vault_id, model) pair.

This partial RDD is cached to prevent redundant computations and take advantage of its presence in memory. Essentially, when the first action is performed, the partial RDD is cached, enabling the second action to benefit from its cached state.

Next, we computed the RDD for model failures through the following transformations:

- Starting from the *partial_rdd*, we applied a ***map*** function, in order to convert each element of the RDD into a key-value pair (model, failures_count), where failures_count counts the number of failures for each model inside of each different vault.

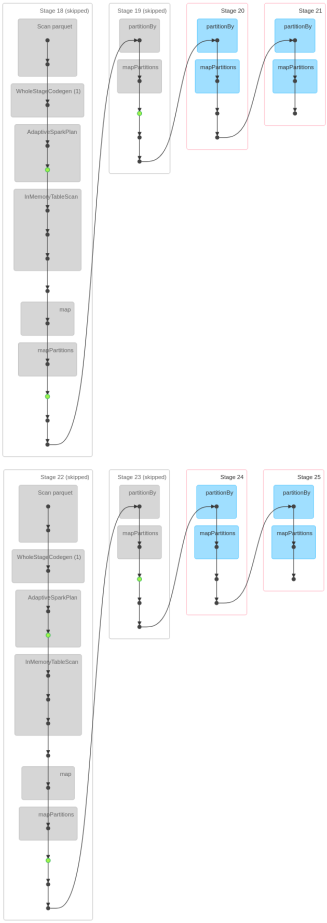


Fig. 5. DAGs of Q2 with RDD

- We then applied a ***reduceByKey***, that sums the number of failures for each model over all the different vaults.
- Lastly, we applied a ***sortBy*** transformation to obtain an ordering based on the number of failures for each model (descending order).

Similarly, we computed the RDD for vaults failures following these transformations:

- Beginning with the *partial_rdd*, we utilized a ***map*** transformation to convert each element into a key-value pair (vault_id, (failures_count, [model])). Here, the second element of each pair's value, denoted as the model that failed under that vault, was "grouped" into a list for subsequent aggregation.
- Next, we applied a ***reduceByKey*** transformation to sum failures for each vault and compute the list of models, without duplicates. This was achieved by employing a combination of list and set operations in Python.
- We then applied a ***sortBy*** to obtain a new RDD sorted by the number of failures for each vault (descending order).
- Lastly, we employed a ***map*** operation to convert each element into the following tuple: (vault_id, failure, list_of_models), where the list of models is represented as a string with each model separated by a comma. This

was achieved using the join operation in Python.

To obtain the first 10 records from both computed RDDs, we performed two take(10) actions. We chose not to construct a single RDD using join operations because caching the partial result and materializing it with the first action can significantly improve the performance of the second take action. Additionally, this approach also allows for more fine-grained actions, each handling less data. Moreover, join operations introduce extra overhead. The trade-off is that Spark must execute two different jobs, thus preparing and executing separate execution plans, which adds some overhead. However, using the caching mechanism and avoiding join operations and reorganizing results for each part of the query is more efficient overall. Finally, after the take operations, we computed a QueryResult object containing the results of these two actions.

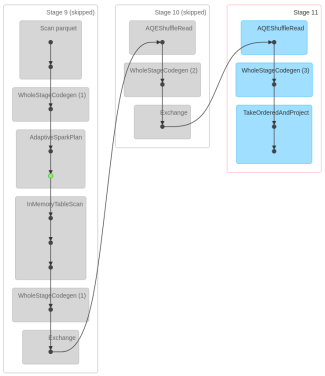


Fig. 6. DAG of Q2.1 with DataFrame

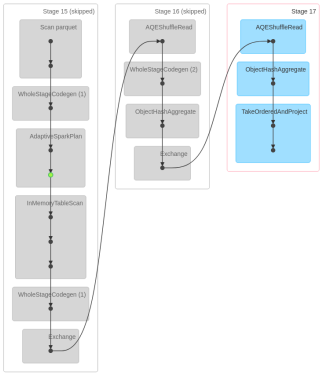


Fig. 7. DAG of Q2.2 with DataFrame

Q2 with Spark SQL: For the Spark SQL version of Q2, we followed a similar schema to the RDD version, by writing three different "sub-queries", with one of them conveying to a partial DataFrame (analogous to the partial RDD in the RDD version).

For the partial dataframe, we followed these steps:

- We start from the *DiskMonitor* "table", filtering the rows according to a WHERE clause, maintaining only rows with failure=1.
- We then organize the data with a GROUP_BY (vault_id, model) so that we can sum up the total failures for each (vault, model) pair.
- Finally, we SELECT vault_id, model and failures_count.

Having this partial DataFrame, we then compute the DataFrame for models' failures according to these steps:

- Starting from the table resulting from the partial DataFrame (VaultModelFailures), we apply a GROUP_BY model to organize the data according to each different model.
- We then apply an ORDER_BY the number of failures for each model.
- Afterwards, we filter only the first ten rows, using LIMIT.
- Finally, we SELECT the model and the number of failures per model (we used SUM()).

Similarly, we compute the DataFrame for vaults failures:

- Starting from the table resulting from the partial DataFrame, we apply a GROUP_BY vault_id, in order to organize the data according to each vault.
- We then apply an ORDER_BY each vault's number of failures.
- Then, we filter only the first ten rows using the previously cited LIMIT.
- Finally, we SELECT the vault_id, the number of failures per vault (using SUM()), and the list of models for each vault. This was achieved by using a combination of CONCAT_WS and COLLECT_SET to create a comma-separated list of models without repetitions.

In the Spark SQL solution, we did not perform a take action because Spark SQL includes a LIMIT clause, allowing the Spark SQL engine to optimize the execution plan as needed. Consequently, we used collect rather than take on the DataFrame.
The results of the queries using RDDs and Spark SQL were almost identical, with two justified differences. The less significant difference is that some models in the list of vault failures appear in a different order. The more notable difference is that the last entry in both the vaults failures and model failures queries differs between the RDD and SQL approaches. This discrepancy occurs because there are multiple entries with the same number of failures as the last one, leading the two queries to return different entries. This variation illustrates how the overall process and execution between the two frameworks can differ and highlights the distributed nature of the computations.

Query 3. *Calculate the minimum, 25th, 50th, 75th percentile and maximum operating hours (field s9_power_on_hours) of the hard disks that have suffered failures and of the hard disks that have not suffered failures. Pay attention, the s9 power on hours field reports a cumulative value, therefore the statistics*

requested by the query must refer to the last available day of detection for each specific hard disk (consider the use of the serial number field). In the output also indicate the total number of events used to calculate the statistics.

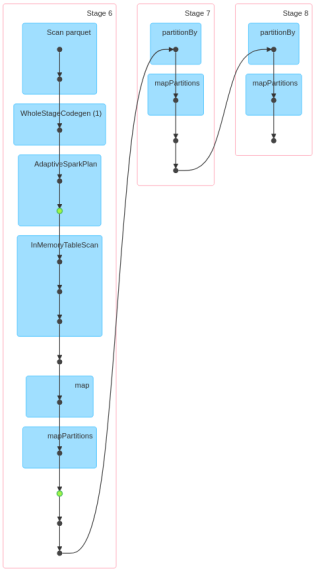


Fig. 8. DAG of Q3 with RDD

Q3 with RDD: We decided to process the RDD following these steps:

- First, a ***map*** transformation is applied to convert each element into the following key-value pair *(serial_number, (failure, s9_power_on_hours))*.
- Next, a ***reduceByKey*** transformation is applied to combine records by *serial_number*. This step computes whether the disk has ever failed and determines the most recent *s9_power_on_hours*. This is achieved by using a logical OR operation between a boolean accumulator and the current failure boolean, and by taking the maximum value between the power on hours accumulator and the current one.
- Another ***map*** transformation is then applied to convert each element into a key-value pair *(failure, s9_power_on_hours)*, where *failure* indicates the failure status for the disk, and *s9_power_on_hours* the maximum hours the disk was powered.
- This is followed by a ***groupByKey*** transformation to group the *s9_power_on_hours* by their *failure* status.
- The values (i.e., lists of *s9_power_on_hours*) are then sorted within each group using a ***mapValues*** transformation.
- Finally, statistics are calculated for each group using a ***map*** transformation, including the minimum, 25th percentile, median (50th percentile), 75th percentile, maximum, and count of *s9_power_on_hours* for both failed and non-failed disks. The minimum was determined by retrieving the first element of the list, while the maximum was obtained by taking the last element.

For the Spark SQL version of the query, we wrote a single SQL query consisting of the following steps on the 'DisksMonitor' table:

- First, the data is aggregated by *serial_number*, computing the maximum *s9_power_on_hours* and the overall *failure* status using the MAX function. This ensures that for each *serial_number*, we get the latest power-on hours and whether the disk has ever failed.
- The results of this aggregation are used as a subquery, aliased as LatestS9.
- Then, an outer query groups the data by *failure* status. For each group, it computes the following statistics:
  - Minimum *s9_power_on_hours* using the MIN function.
  - 25th, 50th (median), and 75th percentiles of *s9_power_on_hours* using the *percentile_approx* function.
  - Maximum *s9_power_on_hours* using the MAX function.
  - Count of records using the COUNT function.

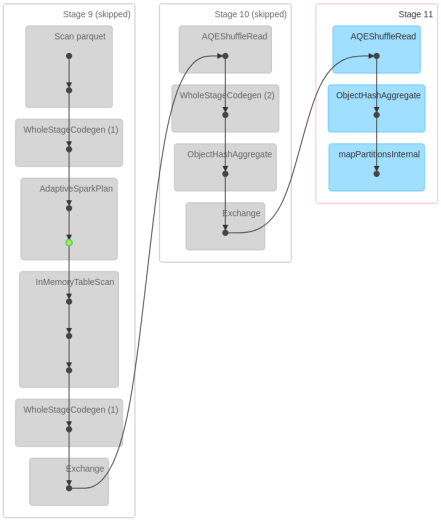


Fig. 9. DAG of Q3 with DataFrame

The final result is a DataFrame containing the failure status, minimum, 25th percentile, median, 75th percentile, maximum *s9_power_on_hours*, and the count of disks for each failure status.

For both the RDD query and SQL query, we used the collect() method to obtain the results.

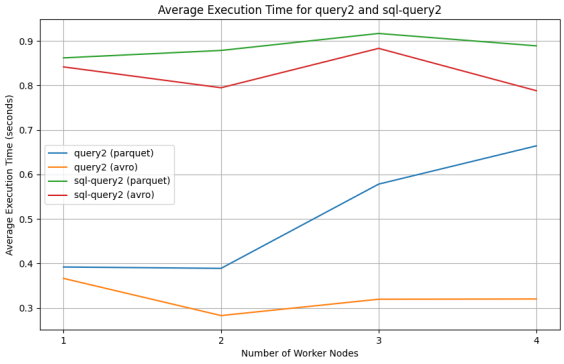
IV. Empirical Results

Queries' execution times have been measured using the *time* library in Python; more specifically, we measured the execution time for each query by taking the time before and

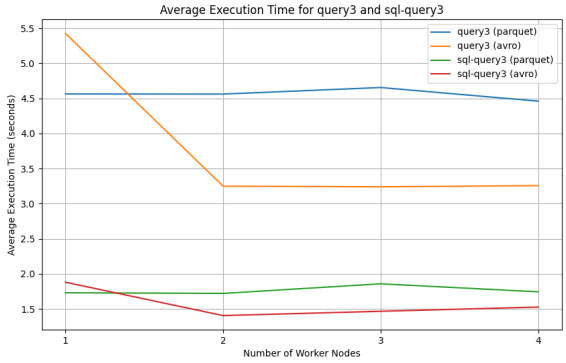
after a Spark action (i.e. a *collect* and *take()*) on both RDDs and DataFrames. We evaluated the processing of all queries 10 times for each combination of data format and number of worker nodes. The data formats used were Avro and Parquet. The number of worker nodes ranged from 1 to 4, constrained by running on a single machine. Increasing the number of worker nodes beyond 4 led to errors in some iterations due to resource limitations. The system architecture was restarted with each scale-up, incrementing the number of worker nodes by 1, to ensure no influences between different executions. After executing all queries once for a given data format and worker nodes configuration, we closed the Spark session to release resources before the next iteration.
Each Spark worker node runs with the following configuration:

- $SPARK\_WORKER\_MEMORY = 1G$
- $SPARK\_WORKER\_CORES = 1$

Then, we computed the mean execution time of the 10 runs for each query configuration, using the previously computed results.



To analyze and explain these data, we break down the analysis into a few key sections: differences between data formats, the effect of the number of worker nodes, and a comparison between queries executed using the Spark Core framework (using RDDs) and Spark SQL (using DataFrames).



A. Differences Between Data Formats (Avro vs. Parquet)

We compared Apache's Avro and Parquet formats, both efficient data serialization formats developed as part of the Hadoop project. This allowed us to contrast row-based and column-based approaches. Generally, Avro outperformed Parquet for most queries, with some exceptions:

- Query 1 (RDD version) with one worker node: In this specific case, Parquet performs slightly better than Avro, although the difference is almost negligible.
- Query 3 (both RDD and SQL versions): For this query, particularly the RDD version, Avro's performance is inferior to Parquet when utilizing only one worker node.

The preference for Avro's performance can be attributed to the transformations that frequently access different fields within a record. These transformations often involve filtering and mapping to new tuples, following a row-based pattern. Avro tends to excel in such scenarios, particularly with the 2 worker nodes configuration, where operations can be parallelized effectively.
However, it's important to note that in the specific case of using a single worker node, which results in partitioning on a single node, we observed that Parquet performs almost identically to Avro. In some cases, Parquet even outperforms Avro. This discrepancy may be attributed to two main reasons: the inability to parallelize mapping and filtering operations effectively (especially beneficial for AVRO), and Parquet's better behaviour for aggregations through its columnar format. This can be observed, for example, when employing operations like groupByKey in query 3, as well as the reduceByKey operation in query 1.

B. Effect of the Number of Worker Nodes

We observe that for most queries, the most significant improvements in average execution times occur when increasing from one to two worker nodes, especially for queries using the Avro format. This is because transformations can be better parallelized and there are more resource allocated. However, as the number of worker nodes increases beyond two, execution times tends to increase. This is likely due to the shuffling caused by aggregation transformations and the fact that resources are limited to a single machine.

Therefore, we cannot make general assumptions about our results regarding the number of Spark workers, both due to resource constraints of our machine and the presence of almost zero latency. In a real-world scenario with distributed nodes, network latency would also play a role, which is not a factor in our case.

C. Comparison Between Spark Core (using RDD) and Spark SQL (DataFrame)

When comparing the SQL versions of the three queries with their RDD counterparts, we observed the following:

- *Query 1 and Query 3* both show better performance in the SQL version.
- *Query 2* shows the opposite trend, with the SQL version of the query performing worse than its RDD counterpart.

Spark SQL, which involves the use of DataFrames, leverages the Catalyst Optimizer and generally tends to perform better. However, the performance also depends on the granularity at which a query operates. RDDs have a lower level of abstraction and, when implemented efficiently, can sometimes lead to better performance. For instance, Query 2 benefits from the more granular control by caching a partial RDD and reusing it in subsequent transformations, followed by performing two take actions that meet the query's needs. This approach is more efficient with RDDs, which provide lower-level control compared to DataFrames in Spark SQL.