



DATABRICKS OPTIMIZATION

Document Date: April 2022

Author: Rahul Biswas

ABSTRACT

Databricks is an industry-leading, cloud-based platform for data analytics, data science, and data engineering supporting thousands of organizations across the world in their data journey. It is a fast, easy, and collaborative Apache Spark-based big data analytics platform for data science and data engineering in the cloud.

Although there is a plethora of material on the web regarding Spark or Databricks optimization, there is lack of any single go-to reference document which explores in detail the various optimization approaches possible. This document is an attempt to bridge that gap. It can serve as a field guide and covers how to select the optimal Spark cluster configuration for running big data processing and workloads in Databricks, some very useful optimization techniques for Spark DataFrames, best practices for optimizing Delta Lake, and techniques to optimize Spark jobs through Spark core.



Intended audience

This document is for data engineers, data scientists, and cloud architects who have working knowledge of Spark/Databricks and some basic understanding of data engineering principles. Readers will need to have a working knowledge of Scala, and some experience of Spark SQL is beneficial.

CONTENTS

- INTRODUCTION TO the SPARK UI 5
 - Journey Through the Spark UI Tabs 5
 - Jobs and Stages 5
 - Jobs tab 6
 - Stages tab 7
 - Executors 9
 - Storage 9
 - SQL 10
 - Environment 11
- Ganglia 13
 - Best practices for using Ganglia 14
- SPARK OPTIMIZATION TECHNIQUES 15
 - General best practices 15
 - Partitioning in Spark 16
 - Bucketing in Spark 20
 - Salting 22
- DELTA ENGINE OPTIMIZATION TECHNIQUES 24
 - Data Skipping 24
 - Bin-packing and Z-ordering 24
 - Auto-Optimize and Auto-Compaction 25
 - Delta Caching 26
 - Important settings for Delta Cache 27
 - Dynamic File Pruning (DFP) 27
 - Important settings for Dynamic Pruning 28
 - Bloom Filter Indexing 28
- OTHER OPTIMIZATION TECHNIQUES 29
 - Caching 29
 - Cache 29
 - Persist 29
 - Best Practices for caching 30
 - When to Cache and Persist 30

When Not to Cache and Persist	30
Cache vs. Delta Cache	30
Broadcast Join	31
Adaptive Query Execution	32
Pandas	33
Apache Arrow	33
Vectorized UDF	33
CLUSTER OPTIMIZATION	34
Understanding cluster types	35
Spot Instances	36
Auto-Scaling	36
Pools	36
Creating a pool	37
Best practices for pools	38
Auto-termination	38
Cluster Sizing	39
Scenario based cluster sizing	39
Databricks Runtime Version	40
References	41
Courses	42
Books	42

INTRODUCTION TO THE SPARK UI

At a high level, every Apache Spark application consists of a driver program that launches various parallel operations on executor Java Virtual Machines (JVMs) running either in a cluster or locally on the same machine. In Databricks, the notebook interface is the driver program. This driver program contains the main loop for the program and creates distributed datasets on the cluster, then applies operations (transformations & actions) to those datasets. Driver programs access Apache Spark through a `SparkSession` object regardless of deployment location.

Spark provides an elaborate web UI that allows us to inspect various components of our Apache Spark application. It offers details on memory usage, jobs, stages, and tasks, as well as event timelines, logs, and various metrics and statistics that can give insight into what transpires in Spark applications, both at the Spark driver level and in individual executors. The UI provides a microscopic lens into Spark's internal workings as a tool for debugging and inspecting.

The SPARK UI is accessible in Databricks by going to "Clusters" and then clicking on the "View Spark UI" link for your cluster, it is also available by clicking at the top left of this notebook where you would select the cluster to attach this notebook to. In this option will be a link to the Apache Spark Web UI.

Journey Through the Spark UI Tabs

The Spark UI has eight tabs, as shown below, each providing opportunities for exploration. Let's look at what each tab reveals to us.

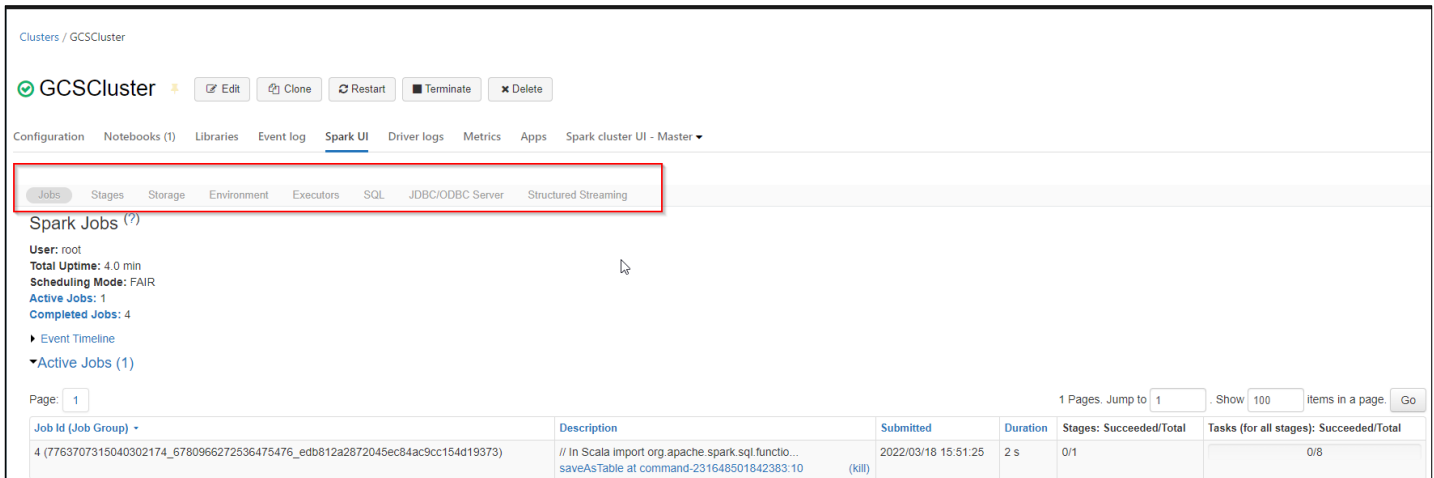
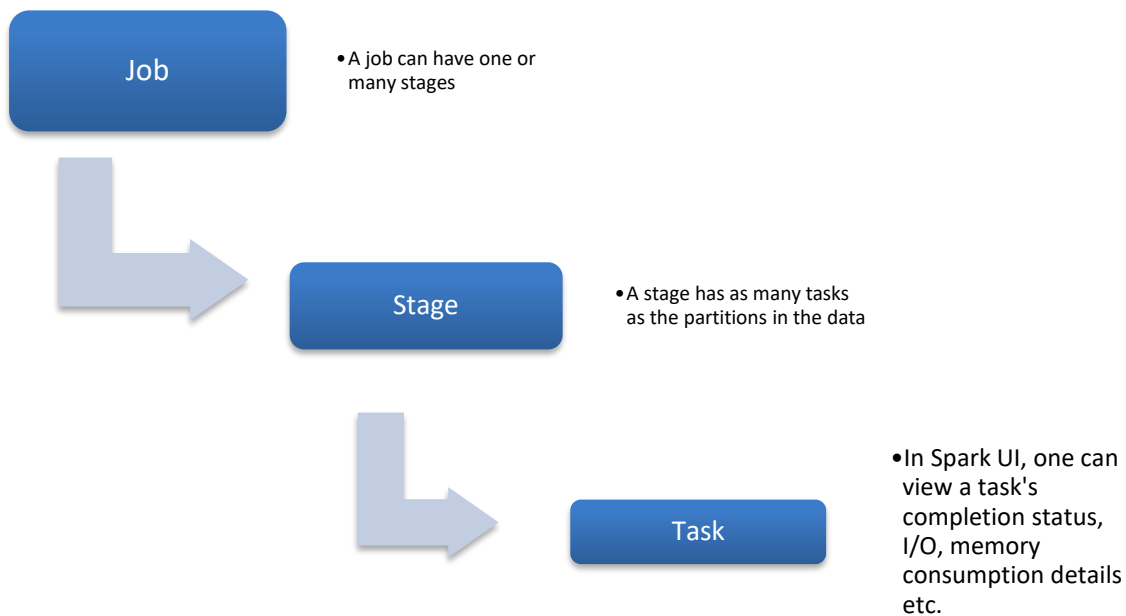


Figure 1 Spark UI tabs

Jobs and Stages

Spark breaks an application down into jobs, stages, and tasks. The Jobs and Stages tabs allow you to navigate through these and drill down to a granular level to examine the details of individual tasks. You can

view their completion status and review metrics related to I/O, memory consumption, duration of execution, etc.



Jobs tab

The following figure shows the Jobs tab.

- A. It has an expanded Event Timeline, showing when executors were added to or removed from the cluster.
- B. It provides a tabular list of all completed jobs in the cluster.
 - The Duration column indicates the time it took for each job (identified by the Job Id in the first column) to finish.
 - If this time is high, it's a good indication that you might want to investigate the stages in that job to see what tasks might be causing delays.
 - From this summary page you can also access a details page for each job, including a DAG visualization and list of completed stages.

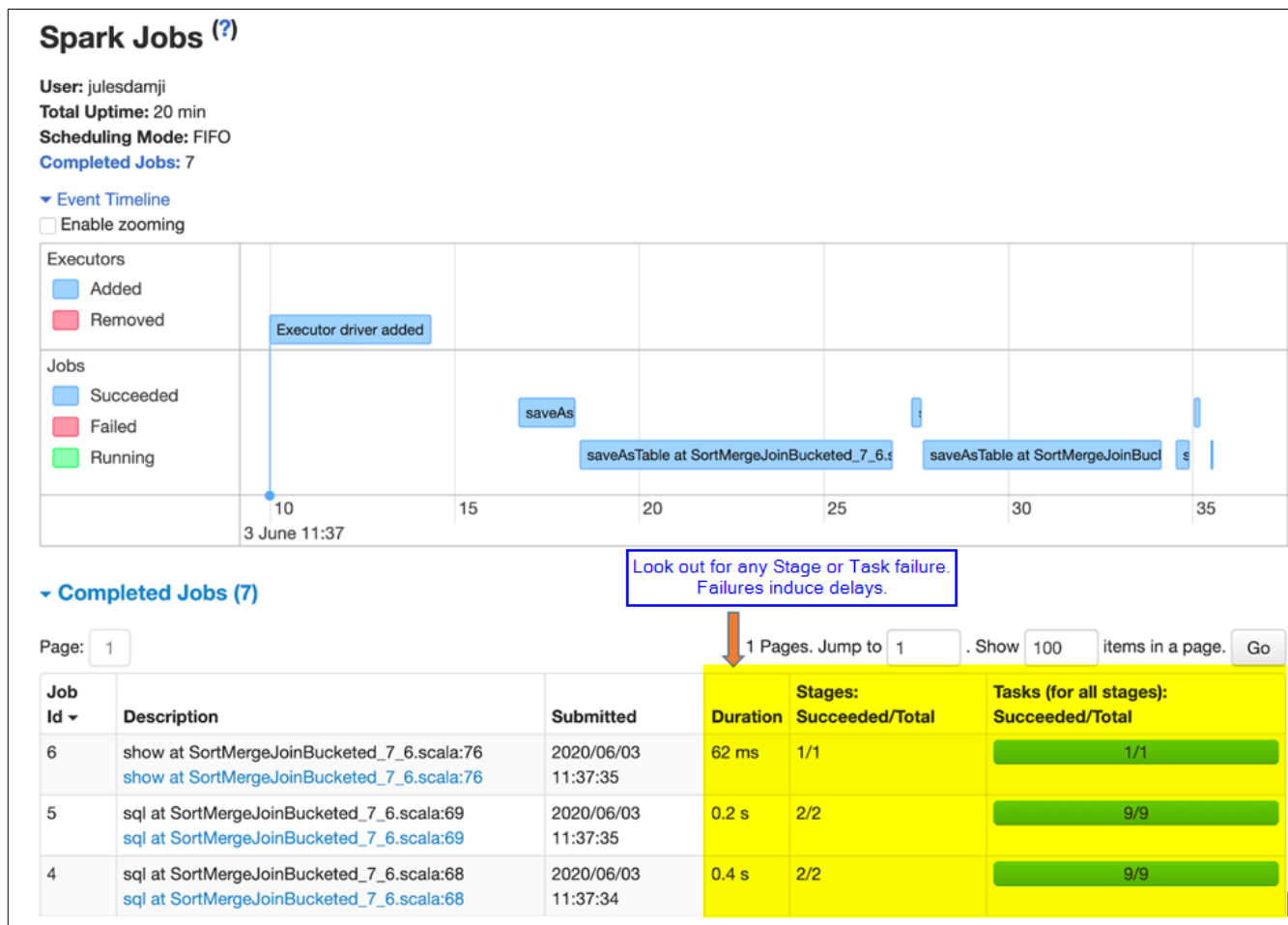


Figure 2 The Jobs tab offers a view of the event timeline and list of all completed jobs

Stages tab

- The Stages tab provides a summary of the current state of all stages of all jobs in the application.
- One can access a details page for each stage, providing a DAG and metrics on its tasks (refer to adjoining figure).
- You can also see aggregated metrics for each executor and a breakdown of the individual tasks on this page.
- Along with other optional statistics, one can see
 - the average duration of each task
 - time spent in garbage collection (GC)
 - number of shuffle bytes/records read

E. Issues to look out for-

- If shuffle data is being read from remote executors, a high Shuffle Read Blocked Time can signal I/O issues.
- A high GC time indicates that executors may be low on memory.
- If a stage's max task time is much larger than the median, this is an indicator of data skew.
- If there are any straggler tasks
- If there are too many or too few partitions (or tasks)

[Show Additional Metrics](#)
[Event Timeline](#)

Summary Metrics for 1 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	48.0 ms	48.0 ms	48.0 ms	48.0 ms	48.0 ms
GC Time	0.0 ms	0.0 ms	0.0 ms	0.0 ms	0.0 ms
Input Size / Records	813.3 KiB / 4	813.3 KiB / 4	813.3 KiB / 4	813.3 KiB / 4	813.3 KiB / 4

Showing 1 to 3 of 3 entries

Aggregated Metrics by Executor

Show entries Search:

Executor ID	Logs	Address	Task Time	Total Tasks	Failed Tasks	Killed Tasks	Succeeded Tasks	Blacklisted	Shuffle Read Size / Records
driver		10.0.1.5:61630	0.2 s	1	0	0	1	false	253.8 KiB / 12411

Showing 1 to 1 of 1 entries [Previous](#) [Next](#)

Tasks (1)

Show entries Search:

Index	Task ID	Attempt	Status	Locality level	Executor ID	Host	Logs	Launch Time	Duration	GC Time
0	24	0	SUCCESS	NODE_LOCAL	driver	10.0.1.5		2020-06-03 20:06:43	0.1 s	

Showing 1 to 1 of 1 entries [Previous](#) [Next](#)

Figure 3 The Stages tab provides details on stages and their tasks

Executors

The Executors tab provides information on the executors created for the application. As you can see in below figure, you can deep dive into the microscopic details about –

- A. resource usage (disk, memory, cores)
- B. time spent in GC
- C. amount of data written and read during shuffle, etc.

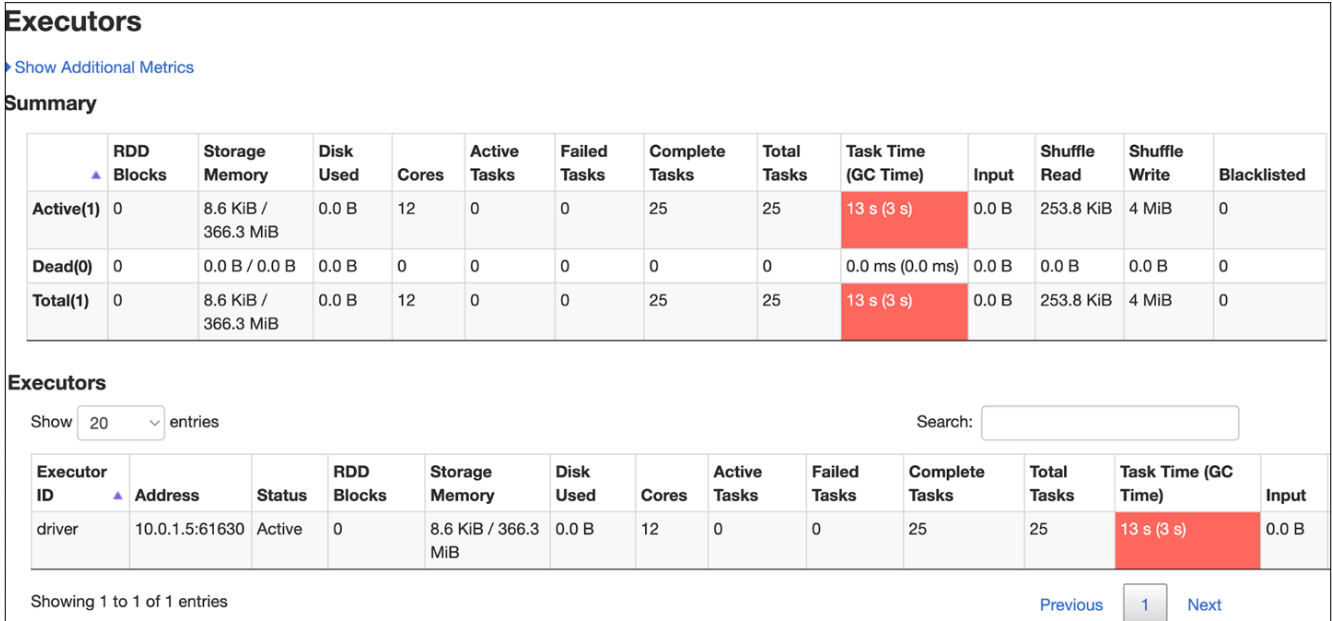


Figure 4 Executors

In addition to the summary statistics, you can view how memory is used by each individual executor, and for what purpose. This also helps to examine resource usage when you have used the `cache()` or `persist()` method on a DataFrame or managed table, which we discuss next.

Storage

The Storage tab, shown below, provides information on any tables or DataFrames cached by the application because of the `cache()` or `persist()` method.

▼ RDDs						
ID	RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size on Disk
2	LocalTableScan [uid#13, login#14, email#15, user_state#16]	Disk Serialized 1x Replicated	12	100%	0.0 B	4.4 MiB
16	LocalTableScan [transaction_id#40, quantity#41, users_id#42, amount#43, state#44, items#45]	Disk Serialized 1x Replicated	12	100%	0.0 B	1771.0 KiB
31	In-memory table `UsersTbl`	Disk Memory Deserialized 1x Replicated	8	100%	4.4 MiB	0.0 B
42	In-memory table `OrdersTbl`	Disk Memory Deserialized 1x Replicated	8	100%	2.0 MiB	0.0 B

Figure 5 Storage tab

SQL

The effects of Spark SQL queries that are executed as part of your Spark application are traceable and viewable through the SQL tab. You can see when the queries were executed and by which jobs, and their duration.

SQL				
Completed Queries: 9				
▼ Completed Queries (9)				
Page: <input type="text" value="1"/> 1 Pages. Jump to <input type="text" value="1"/> . Show <input type="text" value="100"/> items in a page. <input type="button" value="Go"/>				
ID ▼	Description	Submitted	Duration	Job IDs
8	show at SortMergeJoinBucketed_8_6.scala:69 +details	2020/02/27 18:44:34	0.2 s	[6]
7	sql at SortMergeJoinBucketed_8_6.scala:63 +details	2020/02/27 18:44:33	0.4 s	[5]
6	sql at SortMergeJoinBucketed_8_6.scala:63 +details	2020/02/27 18:44:33	0.4 s	
5	sql at SortMergeJoinBucketed_8_6.scala:62 +details	2020/02/27 18:44:33	0.7 s	[4]
4	sql at SortMergeJoinBucketed_8_6.scala:62 +details	2020/02/27 18:44:33	0.8 s	
3	saveAsTable at SortMergeJoinBucketed_8_6.scala:60	2020/02/27 18:44:31	1 s	[2] [3]

Figure 6 The SQL tab shows details on the completed SQL queries

Clicking on the description of a query displays details of the execution plan with all the physical operators, as shown in below figure. Under each physical operator of the plan are SQL metrics. Examples of physical operators, as demonstrated in the diagram are- Scan In-memory table, HashAggregate, and Exchange.

These metrics are useful when we want to inspect the details of a physical operator and discover what transpired: how many rows were scanned, how many shuffle bytes were written, etc.

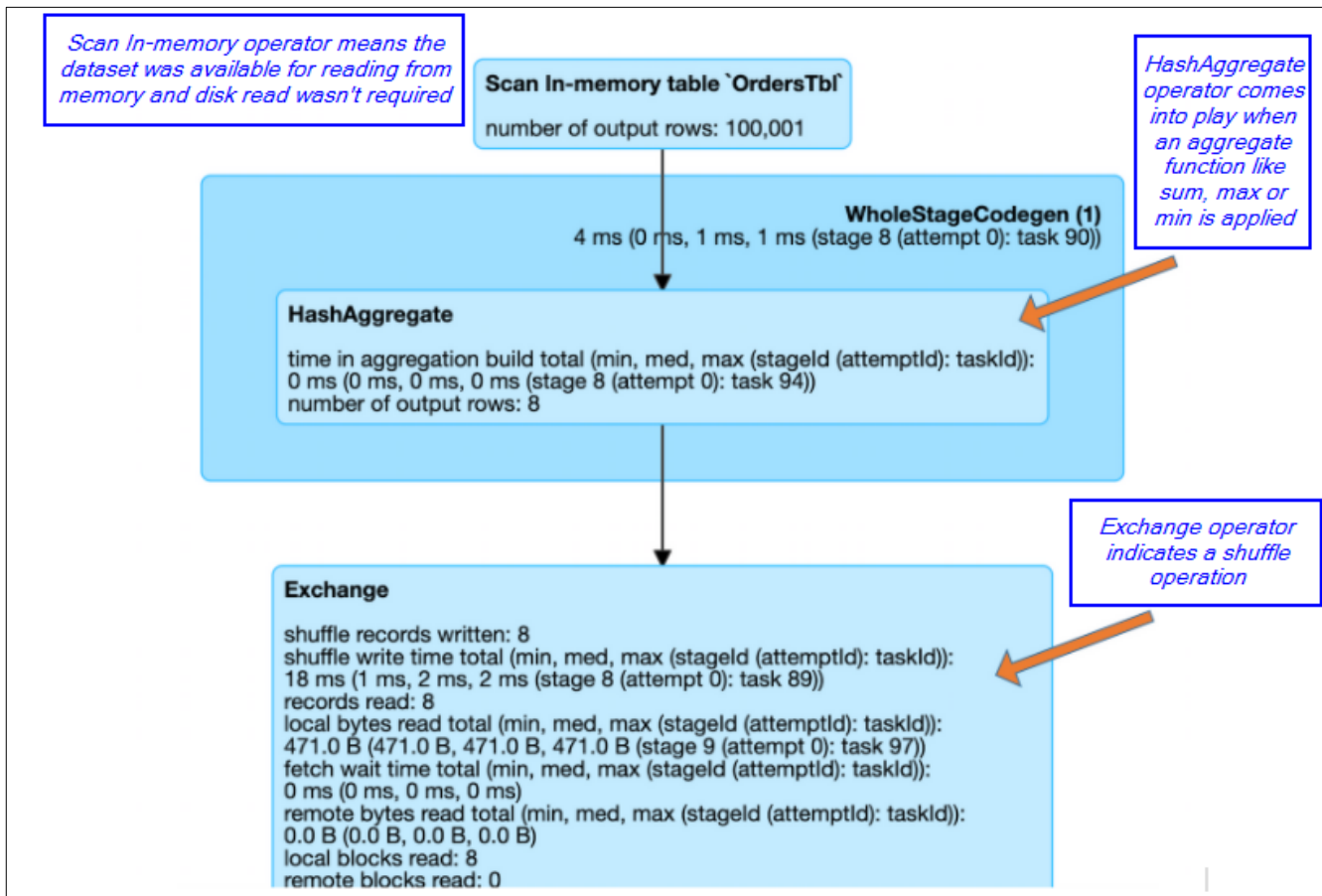


Figure 7 Spark UI showing detailed statistics on a SQL query

Environment

The Environment tab allows one to know –

- A. Environment variables values
- B. Included jars
- C. Spark properties settings

D. System properties settings

E. what runtime environment (such as JVM or Java version) is used, etc.

All these read-only details are very helpful for investigative efforts should you notice any abnormal behavior in your Spark application.

Environment	
▼ Runtime Information	
Name	Value
Java Home	/Library/Java/JavaVirtualMachines/jdk1.8.0_241.jdk/Contents/Home/jre
Java Version	1.8.0_241 (Oracle Corporation)
Scala Version	version 2.12.10
▼ Spark Properties	
Name	Value
spark.app.id	local-1591215877337
spark.app.name	SortMergeJoinBucketed
spark.driver.host	10.0.1.5
spark.driver.port	61781
spark.executor.id	driver
spark.jars	file:/Users/julesdamji/gits/LearningSparkV2/chapter7/scala/jars/scala-chapter7_2.12-1.0.jar
▶ Hadoop Properties	
▶ System Properties	
▼ Classpath Entries	
Resource	Source
/Users/julesdamji/spark/spark-3.0.0-preview2-bin-hadoop2.7/conf/	System Classpath
/Users/julesdamji/spark/spark-3.0.0-preview2-bin-hadoop2.7/jars/HikariCP-2.5.1.jar	System Classpath
/Users/julesdamji/spark/spark-3.0.0-preview2-bin-hadoop2.7/jars/JLargeArrays-1.5.jar	System Classpath

Figure 8 The Environment tab shows the runtime properties of your Spark cluster

GANGLIA

To understand how the machines from the cluster are working, we can look at the Ganglia dashboard, a monitoring system of high-performance computing where you can check metrics related to CPU, memory usage, network, etc. By analyzing the Ganglia, we can understand if our cluster configuration is under-provisioned, over-provisioned or provisioned appropriately. It's largely complementary to Databricks UI, though it also works at the cluster and at the node level. We can see node level metrics such as CPU consumption, memory consumption, disk usage, network-level IO – all node level factors that can affect the stability and performance of our job. This can allow us to see if your nodes are configured appropriately, to institute manual scaling or auto-scaling, or to change instance types.

NOTE: Ganglia does not have job-specific insights, neither does it work with pipelines. There is a lack of good output options; we are not left with many options besides taking a screen snapshot to get a JPEG or PNG image of the current status.

We can find the Ganglia at *Databricks Clusters > Metrics*, and it is shown below.

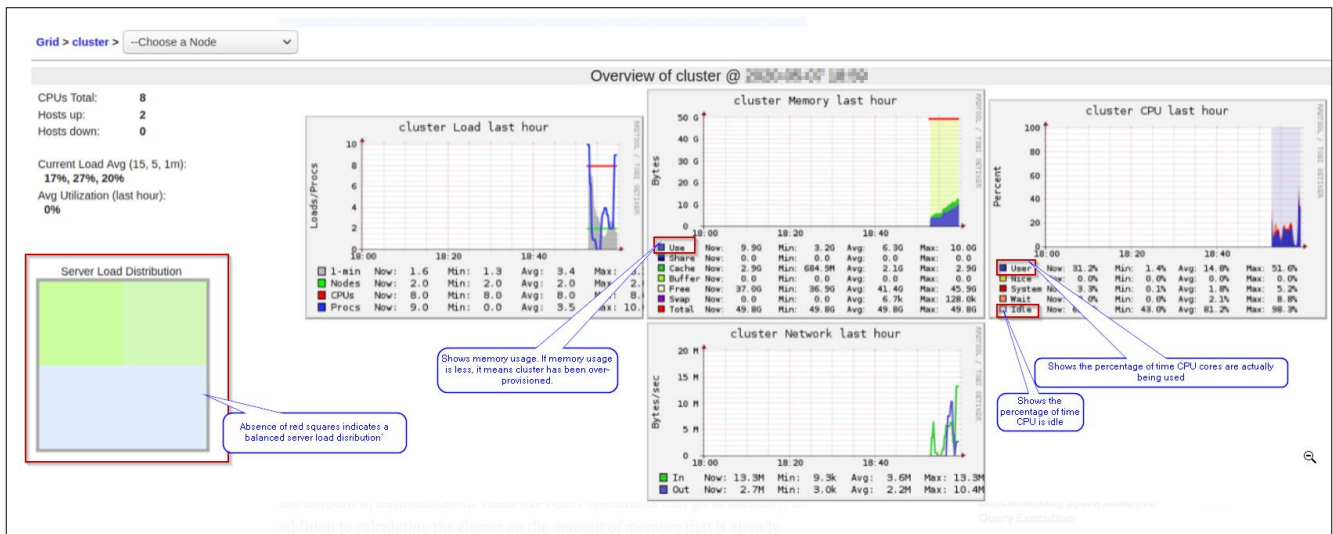


Figure 9 Ganglia metrics and their interpretation

The above diagram shows an example of a balanced server load distribution. The below shows an example of an unbalanced server load distribution. Look out for the red squares. Those indicate the hot spots where load is more.

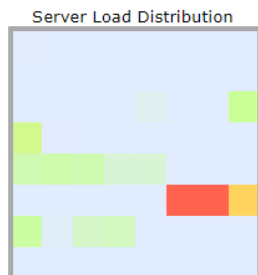


Figure 10 Example of a disbalanced server load distribution

Another thing we need to be on the lookout for is memory swapping as this indicates pressure on RAM. The below diagram highlights an example.

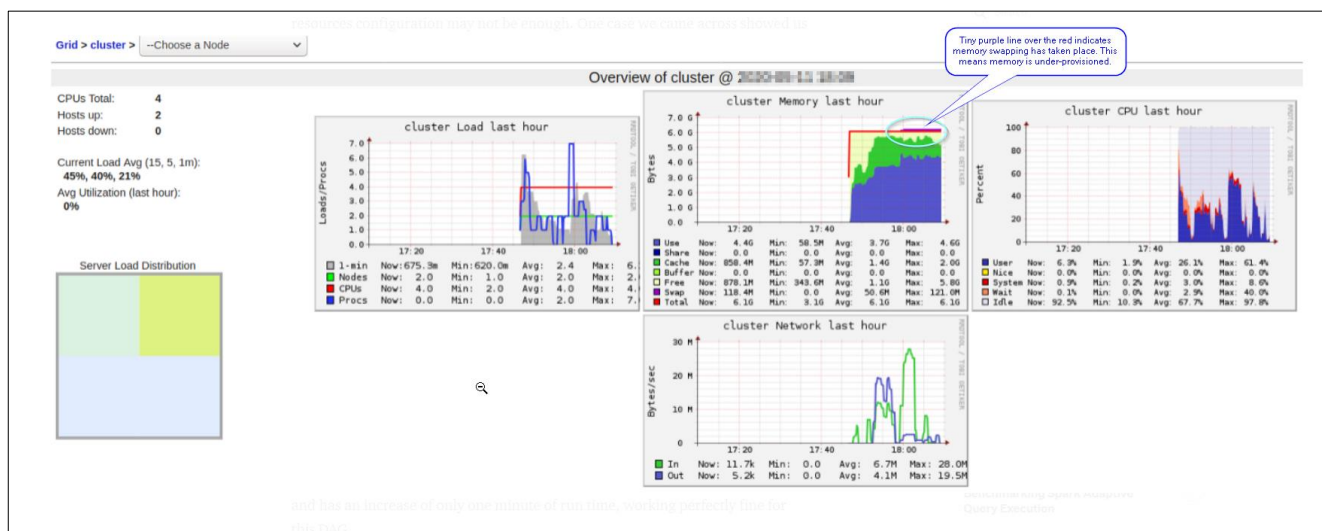


Figure 11 Indicator of RAM pressure

Best practices for using Ganglia

- Consider the *cluster Memory last hour* and *cluster CPU last hour* dashboards. Look for usage and idle times.
- Consider the *Server Load Distribution* map to get an idea of how balanced the workload across the nodes is. Absence of red squares indicates a well-balanced load.
- Be on the lookout for Memory Swapping. It can be detected in *cluster Memory last hour* dashboard by seeing a small purple line over a red line as indicated in the above diagram. This indicates memory pressure.

SPARK OPTIMIZATION TECHNIQUES

General best practices

- A. Avoid `collect()` method on a DataFrame. It may cause out of memory issues on the driver node if data volume is too big to fit into driver memory. If it must be used, apply appropriate filtering or aggregation on the original DataFrame and then use `collect()`.
- B. Where performance is critical and one is dealing with voluminous source files, consider keeping the `inferSchema` option as false. This is the default and in this case every field is read in as a string. In case you need to enforce a schema, supply the schema explicitly to the read operation, using the `schema` option.
- C. Where possible, use parquet file format instead of csv. Parquet file formats have greatly reduced storage and data scanning requirements. Also, they work very well in partitions and help to leverage Spark's parallelism. A 1 TB CSV file, when converted to Parquet, can bring down the storage requirements to as low as 100 GB in cloud storage. This helps Spark to scan less data and speed up jobs in Databricks.
- D. Avoid user defined functions (UDF) and try to use native Spark functions as much as possible. User defined functions are opaque to the Catalyst optimizer, and it cannot take any steps to optimize them. On top of this, there is the added load of serialization of code. When we write any Spark code, it must be serialized, sent to the executors, and then deserialized before any output can be obtained. Python UDFs require moving data from the executor's JVM to a Python interpreter, which induce even more slowness. In some cases, if UDF is entirely necessary to implement a business logic, consider using Scala to define the UDF instead of Python. PySpark UDFs are much slower and more memory-intensive than Scala and Java UDFs are. Spark is written in Scala and runs on the Java Virtual Machine (JVM), so Scala and Java UDFs run better compared to PySpark UDF.
- E. Employ **Column Predicate Pushdown** as much as possible.
 - When reading data from databases, for example, consider limiting the data read using appropriate filter conditions in the source SQL itself.
 - To speed up filter queries on a DataFrame-
- I. Write DataFrames to disk, and partition it by columns which are most **frequently used for filtering queries**, using `partitionBy` method.

Tip→ Use low cardinality columns for partitionBy. Otherwise, a lot many very small partitions would be generated, and this would increase disk I/O and bring down performance. This is known as the "**small file problem**"—many small partition files, introducing an inordinate amount of disk I/O and performance degradation thanks to filesystem operations such as opening, closing, and listing directories, which on a distributed filesystem can be slow.

- II. For subsequent read operations which employ filtering on the *columns chosen for partitioning*, read from the partitioned files outputted to disk in step A using *spark.read* method.
- III. The amount of data scanned by Spark engine in step B reduces because only relevant partitions based on filter conditions are scanned and this speeds up read queries. For instance, if partitionBy columns were Year and Month in step A, then any query in step B which relies on filter conditions like Year = 2022 and/or Month = 03 shall benefit because only the partitions corresponding to Year = 2022 and Month = 03 need to be scanned by Spark engine. This is called **partition pruning**.
- IV. The benefit of Partition Columns: Spark supports partition pruning which skips scanning of non-needed partition files when filtering on partition columns. However, notice that partition columns do not help much on joining in Spark.
- V. When to Use Partition Columns:
 - Table size is big (tens of GBs and above).
 - The table has low cardinality columns which are frequently used in filtering conditions.
 - Amount of data in each partition: You can partition by a column if you expect data in that partition to be at least 1 GB. Partitioning is not required for smaller tables.
- VI. How to Choose Partition Columns:
 - Choose low cardinality columns as partition columns (since an HDFS directory will be created for each partition value combination). The total number of partition combinations should ideally be less than 50 thousand.
 - The columns are used frequently in filtering conditions.
 - Use at most 2 partition columns as each partition column creates a new layer of the directory.

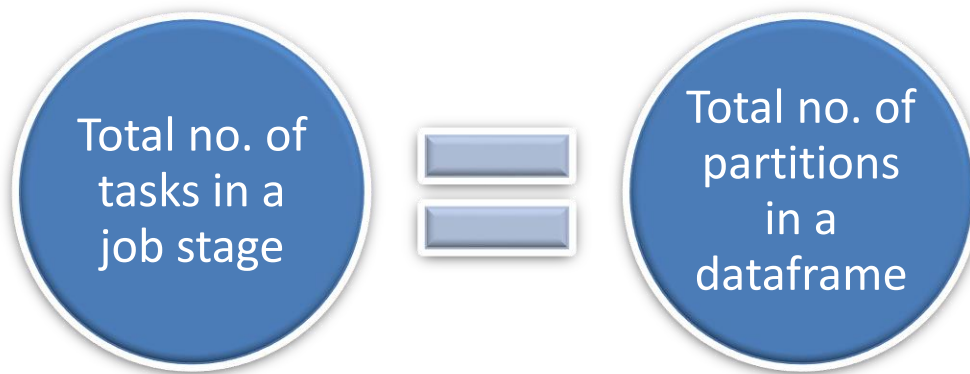
F. Employ Column Pruning as much as possible. Select only the fields in your DataFrame which you need for your requirement and scrap the rest.

Partitioning in Spark

Whenever Spark processes data in memory, it *breaks that data down into parts*, and these parts are processed in the cores of the executors. These are the Spark **partitions**. Let's examine how to examine the partitions for any given DataFrame:

- A. To check the Spark partitions of a given DataFrame, we use the following syntax:
dataframe.rdd.getNumPartitions. Also, remember that the total number of tasks doing work on a

Spark DataFrame is equal to the total number of partitions of that DataFrame.



- B. Next, we will learn how to check the number of records in each Spark partition. We will begin with creating the airlines DataFrame. We are using Scala so that we can use the [mapPartitionsWithIndex](#) function. Run the following code block to create the DataFrame:

```
1 %scala
2 // Read csv files to create Spark dataframe
3 val airlines = (
4   spark
5   .read
6   .option("header",true)
7   .option("delimiter",",")
8   .option("inferSchema",true)
9   .csv("dbfs:/databricks-datasets/asa/airlines/*")
10 )
11
```

▼ (2) Spark Jobs

- ▼ Job 23 View (Stages: 1/1)
Stage 32: 1/1
- ▼ Job 24 View (Stages: 1/1)
Stage 33: 93/93

airlines: org.apache.spark.sql.DataFrame = [Year: integer, Month: integer ... 27 more fields]
airlines: org.apache.spark.sql.DataFrame = [Year: int, Month: int ... 27 more fields]

93 tasks in a Stage indicates data has been distributed into 93 partitions

- C. Next, we will check the number of partitions in the DataFrame using the following code: [airlines.rdd.getNumPartitions](#). Our DataFrame has 93 partitions.

```
1 %scala
2 airlines.rdd.getNumPartitions
```

res3: Int = 93

D. Now we will execute Scala code that displays the number of records in each partition:

Cmd 37





```
1 %scala
2 display(airlines
3   .rdd
4   .mapPartitionsWithIndex{case (i,rows) => Iterator((i,rows.size))}
5   .toDF("partition_identifier","records")
6 )
```

► (4) Spark Jobs

Table Data Profile

	partition_identifier ▲	records ▲
1	0	1398172
2	1	1397695
3	2	1394085
4	3	1395041
5	4	1394619
6	5	1390018
7	6	1394208

Showing all 93 rows.

Here, we get a DataFrame consisting of two columns:

- a) partition_identifier → This gives a unique ID to every partition of the DataFrame, starting from 0.
- b) records → This indicates the number of records in a particular partition.

We can even create a bar chart from this DataFrame to understand data skew in the DataFrame. Data skewing simply refers to the phenomenon where some partitions of the DataFrame are heavily loaded with records while others are not. In other words, DataFrames that have data evenly spread across partitions are not skewed in nature. In our DataFrame, most of the partitions consist of 1.3 million to 1.5 million records.

The Spark configuration `spark.sql.files.maxPartitionBytes` can play a major role in influencing the `partition size`. It does this by limiting the maximum size of a Spark partition when **reading** from source files. By default, the limit is set to 128 MB. The syntax is as follows:

```
spark.conf.set('spark.sql.files.maxPartitionBytes', <put desired value in bytes here>)
```

Two other configurations which influence partitioning-

spark.sql.shuffle.partitions

Number of partitions to use by default when shuffling data for joins or aggregations

Default: 200

spark.sql.files.maxRecordsPerFile

Maximum number of records to write out to a single file. If this value is 0 or negative, there is no limit.

Default: 0

Additionally, there are also two very useful functions to manage Spark partitions:

repartition(): This function is used to increase or decrease the number of Spark partitions for a DataFrame. It induces a shuffle when called and is often very helpful to remove skew from a DataFrame. The syntax is `dataframe.repartition(number)`. Here, number designates the new partition count of the DataFrame. The repartition function leads to roughly equally sized partitions.

There is a difference between `partitionBy()` and `repartition()`. Both `repartition()` and `partitionBy` can be used to "partition data based on DataFrame column", but `repartition()` partitions the data in memory and `partitionBy` partitions the data on disk. As such, `partitionBy` is only available when one uses the `DataFrame.write` method i.e., when one is writing a DataFrame to disk.

coalesce(): This function is used to decrease the number of partitions and is extremely helpful when the partition count needs to be drastically reduced. *Also, note that it does not lead to shuffling.* The syntax is `dataframe.coalesce(number)`. Here, number designates the new partition count of the DataFrame. The coalesce function can often lead to skew in partitions.

If we have more cores and fewer partitions and fewer tasks being performed, then it is better to increase the partitions of the DataFrame so that all the cores get engaged. This can be very useful with the delta file format, as even though the result of such a job might spit out small files, these could be later compacted using various techniques like OPTIMIZE, auto-optimize and auto-compaction.

Tip → In Azure Databricks, to check the number of cores in a cluster programmatically, we can use `spark.sparkContext.defaultParallelism`.

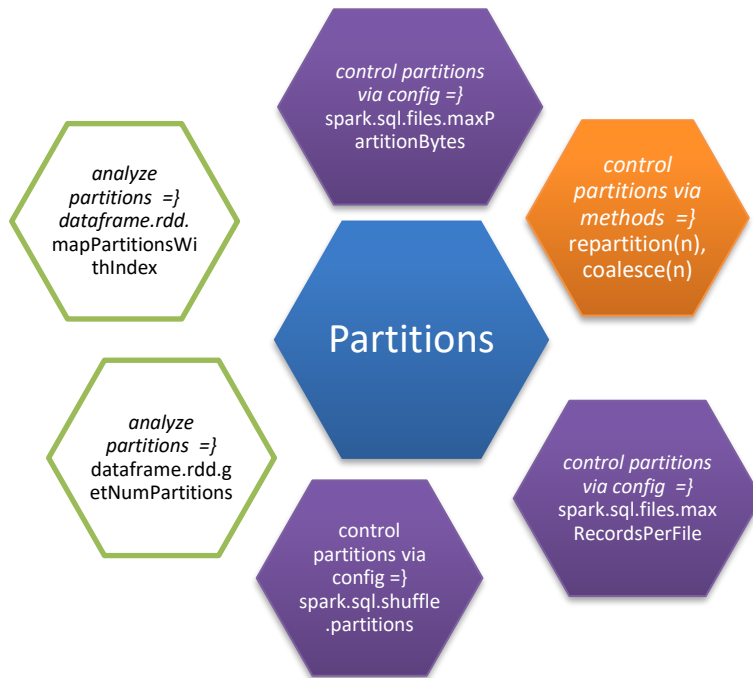


Figure 12 Concepts related to Spark Partition optimization

Bucketing in Spark

Bucketing is an optimization technique that helps to prevent shuffling and sorting of data during compute-heavy operations such as joins. Based on the bucketing columns we specify data is collected in several *bins*. Bucketing is like partitioning, but in the case of partitioning, we create directories for each partition. In bucketing, we create equal-sized buckets, and data is distributed across these buckets by a hash on the value of the bucket. Partitioning is helpful when filtering data, whereas bucketing is more helpful during joins. Queries with sort-merge join or shuffle-hash join and aggregations or window functions require the data to be repartitioned by the joining/grouping keys. More specifically, all rows that have the same value of the joining/grouping key must be in the same partition. To satisfy this requirement Spark must repartition the data, and to achieve that, Spark must physically move the data from one executor to another — also known as shuffling.

With bucketing, we can shuffle the data in advance and save it in this pre-shuffled state. After reading the data back from the storage system, Spark will be aware of this distribution and will not have to shuffle it again.

In Spark API there is a function **bucketBy** that can be used for this purpose:

```
(
  df.write
    .mode(saving_mode) // append/overwrite
    .bucketBy(n, field1, field2, ...)
    .sortBy(field1, field2, ...)
    .option("path", output_path)
    .saveAsTable(table_name)
)
```

Important points and best practices with respect to bucketing:

- A. We need to save the data as a table for bucketing to work.
- B. The **n** in the code snippet indicates the number of buckets.
- C. Together with `bucketBy`, we can call also `sortBy`, this will sort each bucket by the specified fields. Calling `sortBy` is optional, bucketing will work also without the sorting.
- D. The first argument of the `bucketBy` is the number of buckets that should be created. Choosing the correct number should be done by considering the overall size of the dataset and the number and size of the created files.
- E. Indiscriminate usage of the `bucketBy` function can lead to the creation of too many files and custom repartition of the DataFrame might be needed before the actual write.
- F. Benefits of Bucket Columns:
 - Spark supports **bucket pruning** which skips scanning of non-needed bucket files when filtering on bucket columns.
 - Bucket join will be leveraged when the 2 joining tables are both bucketed by joining keys of the same data type and bucket numbers of the 2 tables are equal or one is a multiple of the other (e.g., 500 vs 1000).
 - The number of buckets helps guide the Spark engine on parallel execution levels.
- G. When to Use Bucket Columns:
 - Voluminous tables (tens of GBs and above) are involved.
 - The table has high cardinality columns which are frequently used as filtering and/or joining keys.
 - If table is medium sized, but it is mainly used to join with a huge bucketized table, it is still beneficial to bucketize it.
 - The sort-merge join (without bucketing) is slow due to shuffling and not due to data skew
- H. How to Configure Bucket Columns:
 - Choose high cardinality columns as bucket columns.
 - Try to avoid data skew.

- Depending on the queries that you will run on the data, 150-200 MB per bucket might be a reasonable choice and if you know the total size of the dataset, you can compute from this how many buckets to create.
- Sorting buckets are optional but strongly recommended.

Salting

In group by or join operations, sometimes the group by or joining keys experience data skew. A particular set of values in the keys dominate the other possible values and this causes system instability in the form of long running tasks or Out of memory exceptions.

To combat this problem, developers often fall back on a technique called Salting. The idea is like below-

- A. Identify the key with the skew problem. Let's call this field key1.
- B. Key 1 has data skew. Determine a number N such that
 - if every distinct value in key 1 is divided into N chunks,
 - and then we consider the chunked-up version of the key values for joining/grouping,
 - the data distribution shall become more uniform for the chunked-up key.
 - For e.g., if dataset has 100 million rows and 10% of the distinct values of key 1 occupy 80% of the rows, you may determine that if every distinct value was carved up into 20 chunks, then the distribution of the chunked-up version becomes more uniform and manageable. Here, N= 20. Why would more uniformity be possible? It's because, on an average, every distinct value of key 1 is now splitting into 20 new unique values, and thus, volume of records for each new key is $1/20^{\text{th}}$ of the original key.
- C. Use a random function to generate one among N possible values.
- D. Concatenate the original key 1 with the random function output. Let's call this concatenated version our salted key.
- E. Use this salted key for group by operations. Then do another group by on the output using the original key 1. This approach works well for SUM, MIN, and MAX. It shall work for COUNT as well. In first group by, COUNT function must be applied. In the second, SUM must be applied.
- F. This idea is summed up in the pseudo-code snippet below-


```
// n is the no. of chunks you'd like to have
df.withColumn("salt_random_column", (rand * n).cast(IntegerType))
  .groupBy(groupByFields, "salt_random_column") //first aggregation
  .agg(aggFields)
  .groupBy(groupByFields) //second aggregation
  .agg(aggFields)
```

- G. In case of a join, an additional step needs to be performed besides the concatenation with random output. The second table with which the joining happens – each record in that must be cross-joined with the set of all possible values the random function can generate. For e.g., this could be integers 1 to 20. Each record in the second table shall then have 20 versions of the same record, with only the joining key varying among them. This exploded version of the second table shall then be used to join with the primary table. The joining key shall be the salted key. This idea is summed up in the code snippet below which joins a fact table with data skew with a dimension table-

```
// Let's create salt dataframe
val N = 7
val saltDF = spark.range(N).toDF()
val partitions = Math.ceil(865 / 128d).toInt
val statesSaltedDF = spark
  .read.format("delta").load(statesPath)
  .repartition(partitions)
  .crossJoin(saltDF)
  .withColumn("salted_state_id", concat($"state_id", lit("_"),
    $"salt"))
  .drop("salt")
val trxSaltedDF = spark
  .read.format("delta").load(trxPath)
  .withColumn("salt", (lit(N) * rand()).cast("int"))
  .withColumn("salted_state_id",
    concat(col("state_id"), lit("_"), col("salt")))
  .drop("salt")
// Join salted tables
trxSaltedDF
  .join(statesSaltedDF, statesSaltedDF("salted_state_id") ===
    trxSaltedDF("salted_state_id"))
  .write.format("noop").mode("overwrite").save()
```

Note that for smaller data the performance benefit from salting will be marginal.

DELTA ENGINE OPTIMIZATION TECHNIQUES

Data Skipping

As new data arrives, Delta Lake will keep track of file-level statistics related to the **input/output (I/O)** granularity of the data. These statistics are obtained automatically, storing information on minimum and maximum values at query time. The point of this is to identify columns that have a certain level of correlation, plan queries accordingly, and avoid unnecessary I/O operations. During lookup queries, the Azure Databricks Data Skipping feature will leverage the obtained statistics to skip files that are not necessary to read in this operation. There is no need to configure Data Skipping because it is activated by default and applied whenever possible, meaning also that the performance of this operation is bound to the underlying structure of your data.

Bin-packing and Z-ordering

Delta Engine allows improved management of files in Delta Lake, yielding better query speed, thanks to optimization in the layout of the stored data. Delta Lake does this by using two types of algorithms: **bin-packing** and **Z-Ordering**. The first algorithm is useful when merging small files into larger ones and is more efficient in handling the larger ones. The second algorithm is borrowed from mathematical analysis and is applied to the underlying structure of the data to map multiple dimensions into one dimension while preserving the locality of the data points. Z-Ordering is a feature of Delta Lake that seeks to allocate related data in the same location. This can improve the performance of our queries on the table in which we run this command because Azure Databricks can leverage it with data-skipping methods to read and partition files more efficiently.

The syntax for both is as follows:

- A. Bin-packing: `OPTIMIZE delta.'delta_file_path'` or `OPTIMIZE delta_table`. Here, `delta_file_path` is the location of the Delta Lake file and `delta_table` is the name of the delta table.
- B. Z-Ordering: `ZORDER BY (columns)`. Here, columns indicate the names of the columns based on which we are Z-Ordering.

Option 1:

```
OPTIMIZE <table>
ZORDER BY (<field1>, <field2>)
```

Option 2:

```
OPTIMIZE delta.'path_to_data'
ZORDER BY (event_type)
```

As a best practice, columns that have many unique values, as well as columns used to filter by very frequently, are the most recommended columns to optimize by running ZORDER. But *Z-Ordering* on too many columns can also degrade performance. Hence, the columns to Z-Order on should be chosen wisely.

Bin-packing should always be used when different transactions such as inserts, deletes, or updates are being executed on a delta table. Also, it is an idempotent process, meaning that if the **OPTIMIZE** command is run twice on a table, the second run will have no effect.

Optimize is analogous to Index Rebuild in SQL Server. It takes all the partitions and rewrites them in the order we specify (business key). This will reduce the number of partitions and make the Merge statement much faster because the data is stored in required key order and not arbitrarily as the data came in.

Auto-Optimize and Auto-Compaction

Auto Optimize is a feature that helps us automatically compact small files while an individual writes to a delta table. Unlike bin-packing, we do not need to run a command every time Auto Optimize is executed. It consists of two components:

- A. Optimized Writes: Databricks dynamically optimizes Spark partition sizes to write 128 MB chunks of table partitions.
- B. Auto Compaction: Here, Databricks runs an optimized job when the data writing process has been completed and compacts small files. It tries to coalesce small files into 128 MB files. This works on data that has the greatest number of small files. Auto Optimize keeps frequently modified tables optimized, which is very useful when dealing with—for example—low-latency streams of data or frequent table merges.

To enable this option on existing tables, we can use the TBLPROPERTIES SQL command to set those properties to true, as in the following example:

```
ALTER TABLE our_table SET TBLPROPERTIES
(delta.autoOptimize.optimizeWrite = true,
delta.autoOptimize.autoCompact = true)
```

We can also define these options as true by default on all new Delta tables, using the following SQL configuration:

```
set spark.databricks.delta.properties.defaults.autoOptimize.optimizeWrite =
true

set spark.databricks.delta.properties.defaults.autoOptimize.autoCompact = true
```

If we would like to apply this only to our working session, we can use the following code to set the following Spark configurations:

```
spark.databricks.delta.optimizeWrite.enabled
```

```
spark.databricks.delta.autoCompact.enabled
```

One reason to have to enable these options manually is that if we have many concurrent DML operations on the same table from different users, this may cause conflicts in the transaction log, which can be dangerous because this will not cause any failure or retry. Another thing to keep in mind is that on big tables, having Auto Optimize enabled on a table can be used in combination with OPTIMIZE and does not replace the running of ZORDER.

Delta Caching

Delta caching is an optimization technique that helps speed up queries by storing the data in the cluster node's local storage. The Delta cache automatically makes copies of remote Parquet files into the local node storage to accelerate its processing. The remote locations that it can access are Databricks File System (DBFS), Hadoop Distributed File System (HDFS), Azure Blob storage, Azure Data Lake Storage Gen1, and Azure Data Lake Storage Gen2. It can operate fast thanks to optimized decompression and an output format consistent with processing requirements. Data can be preloaded using a CACHE statement. Data is then stored on the local disk and can be read quickly thanks to a solid-state drive (SSD). The easiest way to use delta caching is to provision a cluster with **Standard_L** series worker types (**Delta Cache Accelerated**).

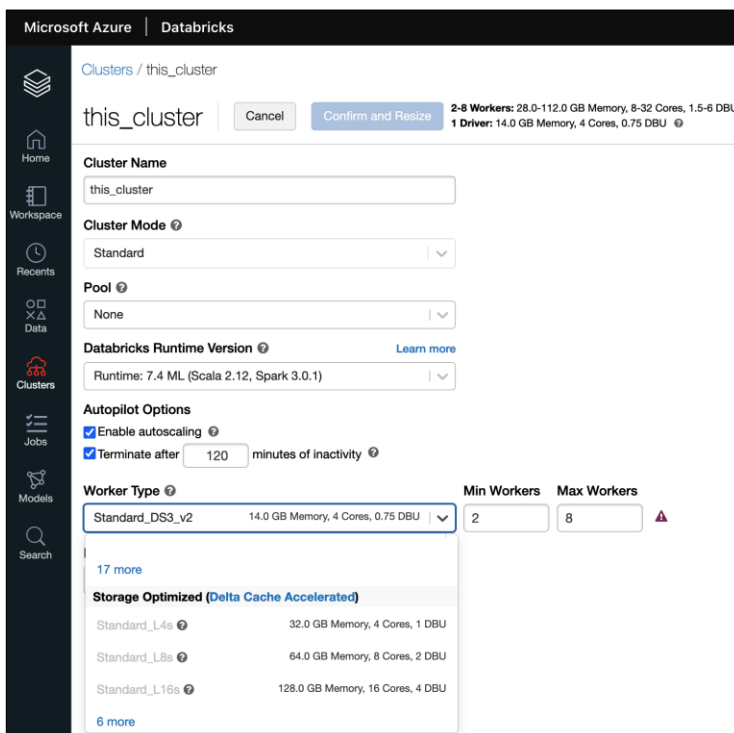


Figure 13 Delta Cache Accelerated Series of VMs

Important settings for Delta Cache

Delta cache-accelerated instances are already configured to work with the Delta cache as default, and you can manually configure its parameters using Apache Spark options during the session creation:

- To establish a maximum disk usage, which is set by default to half of the worker local storage, you can use the `maxDiskUsage` option, measured in bytes, as shown in the following example, where we set it to 30g:

```
spark.databricks.io.cache.maxDiskUsage 30g
```

- We can also disable the compression, which is enabled by default, as follows:

```
spark.databricks.io.cache.compression.enabled false
```

- To disable the Delta cache, we can use the following Scala command:

```
spark.conf.set("spark.databricks.io.cache.enabled", "false")
```

Disabling delta caching does not lead to dropping data from local storage. It only prevents queries from writing data to the cache or reading from it.

Dynamic File Pruning (DFP)

The following kinds of queries, whereby we join two or more tables, are very common when using data models based on a star schema:

```
SELECT col1, col2, col3
FROM fact_table src
JOIN dim_table trg
ON src.id_col=trg.id_col
WHERE col1=our_condition
```

In this case, the condition acts on the dimension table and not on the fact table. Dynamic File Pruning shall reorder the execution plan of the query to be able to check in advance the condition on both tables, which gives the possibility of skipping files on both tables and increasing the performance of the query. Using this technique, we can prune the partitions of a fact table during the join to a dimension table. This is made possible when the filter that's applied to a dimension table to prune its partitions is dynamically applied to the fact table.

This prevents Delta Lake having to scan all the files on the fact table while performing the join.

Dynamic File Pruning is **enabled by default** in Azure Databricks, and it is applied to queries based on the following rules:

- A. The join strategy is BROADCAST HASH JOIN

- B. The join type is INNER or LEFT-SEMI
- C. The inner table is a Delta table
- D. The number of files in the inner table is big enough to justify applying Dynamic File Pruning

Note: The hash join has two inputs like every other join: the build input (outer table) and the probe input (inner table). The query optimizer assigns these roles so that the smaller of the two inputs is the build input and the larger is the probe input.

Important settings for Dynamic Pruning

- A. `spark.databricks.optimizer.dynamicFilePruning` (default is `true`): The main flag that directs the optimizer to push down DFP filters. When set to `false`, DFP will not be in effect.
- B. `spark.databricks.optimizer.deltaTableSizeThreshold` (default is `10,000,000,000 bytes (10 GB)`): Represents the minimum size (in bytes) of the Delta table on the probe side of the join required to trigger Dynamic Pruning. If the probe side is not very large, it is probably not worthwhile to push down the filters and we can just simply scan the whole table. You can find the size of a Delta table by running the `DESCRIBE DETAIL table_name` command and then looking at the `sizeInBytes` column.
- C. `spark.databricks.optimizer.deltaTableFilesThreshold` (default is `10` in Databricks Runtime 8.4 and above, `1000` in Databricks Runtime 8.3 and below): Represents the number of files of the Delta table on the probe side of the join required to trigger dynamic file pruning. When the probe side table contains fewer files than the threshold value, dynamic file pruning is not triggered. If a table has only a few files, it is probably not worthwhile to enable dynamic file pruning. You can find the size of a Delta table by running the `DESCRIBE DETAIL table_name` command and then looking at the `numFiles` column

Bloom Filter Indexing

Bloom filters are a way of efficiently filtering records in a database based on a condition. They have a probabilistic nature and are used to test the membership of an element in a set. We can encounter false positives but not false negatives. These filters were developed as a mathematical construct, to be applied when the amount of data to scan is impractical to be read and are based on hashing techniques. A **bloom filter index** is a data structure that provides data skipping on columns, especially on fields containing arbitrary text. The filter works by either stating that certain data is not in a file or that it is probably in the file, which is defined by a **false positive probability (FPP)**.

We can enable or disable Bloom filters by setting the `spark.databricks.io.skipping.bloomFilter.enabled` session configuration to `true` or `false`. This option is already enabled by default.

We can create Bloom filters using the `CREATE BLOOMFILTER INDEX` command. This applies to all columns on a table or only a subset of them.

The syntax to create a Bloom filter is shown in the following code snippet:

```
CREATE BLOOMFILTER INDEX ON TABLE our_table
FOR COLUMNS(col1 OPTIONS(fpp=0.1), col2 OPTIONS(fpp=0.2))]
```

The lower the fpp the more accurate the index will be.

We can also drop all Bloom filters from a table using the `DROP BLOOMFILTER INDEX` command or by running `VACUUM` on a table.

OTHER OPTIMIZATION TECHNIQUES

Caching

Every time we perform an action on a Spark DataFrame, Spark must re-read the data from the source, run jobs, and provide an output as the result. If a certain DataFrame needs to be queried repeatedly, Spark will have to re-do the same work every time. In such cases, Spark caching proves to be highly useful. Spark *caching* means that we store data in the cluster's memory. Spark reserves a section of memory for cached DataFrames. Every time a DataFrame is cached in memory, it is stored in the cluster's memory, and Spark does not have to re-read it from the source to perform computations on the same DataFrame.

Cache

`cache()` will store as many of the partitions read in memory across Spark executors as memory allows. If not all your partitions are cached, when you want to access the data again, the partitions that are not cached will have to be recomputed, slowing down your Spark job.

Persist

`persist(StorageLevel.LEVEL)` offers more control over how your data is cached via `StorageLevel`.

StorageLevel	Description
MEMORY_ONLY	Data is stored directly as objects and stored only in memory.
MEMORY_ONLY_SER	Data is serialized as compact byte array representation and stored only in memory. To use it, it has to be deserialized at a cost.

StorageLevel	Description
MEMORY_AND_DISK	Data is stored directly as objects in memory, but if there's insufficient memory the rest is serialized and stored on disk.
DISK_ONLY	Data is serialized and stored on disk.
OFF_HEAP	Data is stored off-heap. Off-heap memory is used in Spark for storage and query execution.
MEMORY_AND_DISK_SER	Like MEMORY_AND_DISK, but data is serialized when stored in memory. (Data is always serialized when stored on disk.)

Each `StorageLevel` (except `OFF_HEAP`) has an equivalent `LEVEL_NAME_2`, which means replicate twice on two different Spark executors: `MEMORY_ONLY_2`, `MEMORY_AND_DISK_SER_2`, etc. While this option is expensive, it allows data locality in two places, providing fault tolerance and giving Spark the option to schedule a task local to a copy of the data.

Note: Spark caching or persisting are transformations and therefore are evaluated *lazily*. To enforce a cache on a `DataFrame`, we need to call an *action*.

Best Practices for caching

When to Cache and Persist

Common use cases for caching are scenarios where we want to access a large data set repeatedly for queries or transformations. For instance- `DataFrames` commonly used during iterative machine learning training or accessed commonly for doing frequent transformations during ETL or building data pipelines.

When Not to Cache and Persist

Some scenarios where caching is not a good idea are:

- A. `DataFrames` that are too big to fit in memory
- B. An inexpensive transformation on a `DataFrame` not requiring frequent use, regardless of size
- C. Rule of thumb is to use memory caching judiciously, as it can incur resource costs in serializing and deserializing, depending on the `StorageLevel` used.

Cache vs. Delta Cache

Spark Caching and Delta Caching are two different concepts although they do share some common themes. The following diagram summarizes the key differences between Delta and Apache Spark caching:

Feature	Delta cache	Apache Spark cache
Storage	Local files on a worker node. The Delta cache accelerates data reads by creating copies of remote files in nodes' local storage using a fast intermediate data format. The data is cached automatically whenever a file must be fetched from a remote location. Successive reads of the same data are then performed locally, which results in significantly improved reading speed.	In-memory blocks, but it depends on storage level.
Applicability	Any Parquet table stored on WASB and other file systems. The Delta cache works for all Parquet files and is not limited to Delta Lake format files. The Delta cache supports reading Parquet files in DBFS, HDFS, Azure Blob storage, Azure Data Lake Storage Gen1, and Azure Data Lake Storage Gen2. It does not support other storage formats such as CSV, JSON, and ORC.	Any DataFrame or RDD.
When is it triggered	Automatically, on the first read (if cache is enabled).	Manually, requires code changes.
How is the evaluation performed	Lazily.	Lazily.
How to proactively do it	CACHE SELECT command	.cache + any action to materialize the cache and .persist.
Availability	Can be enabled or disabled with configuration flags, disabled on certain node types.	Always available.
What is the eviction strategy	Automatically in LRU fashion or on any file change, manually when restarting a cluster.	Automatically in LRU fashion, manually with unpersist.

Broadcast Join

The broadcast hash join is employed when two data sets, one small (fitting in the driver's and executor's memory) and another large enough to ideally be spared from movement, need to be joined over certain conditions or columns. Using a Spark broadcast variable, the smaller data set is broadcasted by the driver to all Spark executors, and subsequently joined with the larger data set on each executor.

This strategy avoids the large exchange.

By default, Spark will use a broadcast join if the smaller data set is less than 10 MB. This configuration is set in `spark.sql.autoBroadcastJoinThreshold`; we can decrease or increase the size depending on how much memory we have on each executor and in the driver. If we have enough memory, we can use a broadcast join with DataFrames larger than 10 MB (even up to 100 MB).

Example code for broadcast join is below:

```
peopleDF.join(  
    broadcast(citiesDF),  
    peopleDF("city") <=> citiesDF("city")  
) .show()
```

Adaptive Query Execution

Adaptive Query Execution (AQE) goes one step ahead of the Catalyst Optimizer and adapts the physical query plan by continuously analysing the run time statistics of data. Essentially, it is query re-optimization that occurs during query execution. The motivation for runtime re-optimization is that Azure Databricks has the most up-to-date accurate statistics at the end of a shuffle and broadcast exchange (referred to as a query stage in AQE). As a result, Azure Databricks can opt for a better physical strategy, pick an optimal post-shuffle partition size and number, or do optimizations that used to require hints, for example, skew join handling.

This can be very useful when statistics collection is not turned on or when statistics are stale. It is also useful in places where statically derived statistics are inaccurate, such as in the middle of a complicated query, or after the occurrence of data skew.

In Databricks Runtime 7.3 LTS and above, AQE is enabled by default. It has 4 major features:

- A. Dynamically changes sort merge join into broadcast hash join.
- B. Dynamically coalesces partitions (combine small partitions into reasonably sized partitions) after shuffle exchange. Very small tasks have worse I/O throughput and tend to suffer more from scheduling overhead and task setup overhead. Combining small tasks saves resources and improves cluster throughput.
- C. Dynamically handles skew in sort merge join and shuffle hash join by splitting (and replicating if needed) skewed tasks into roughly evenly sized tasks.
- D. Dynamically detects and propagates empty relations.

AQE applies to all queries that are:

- E. Non-streaming

- F. Contain at least one exchange (usually when there's a join, aggregate, or window), one sub-query, or both.

Not all AQE-applied queries are necessarily re-optimized. The re-optimization might or might not come up with a different query plan than the one statically compiled.

Pandas

Apache Arrow

Apache Arrow is an in-memory columnar data format that helps to efficiently store data between clustered **Java Virtual Machines (JVMs)** and Python processes. This is highly beneficial for data scientists working with Pandas and **NumPy** in Databricks. Apache Arrow does not produce different results in terms of the data. It is helpful when we are converting Spark DataFrames to Pandas DataFrames, and vice versa. Apache Arrow minimizes inefficiencies which arise due to serialization and deserialization processes pressurizing memory and CPU resources (example - during converting a Spark DataFrame to a Pandas DataFrame).

We turn it on with the following setting:

```
spark.conf.set('spark.sql.execution.arrow.enabled', 'true')
```

Vectorized UDF

Pandas UDFs are user defined functions that are executed by Spark using Arrow to transfer data and Pandas to work with the data, which allows vectorized operations. A Pandas UDF is defined using the `pandas_udf()` as a decorator or to wrap the function, and no additional configuration is required. A Pandas UDF behaves as a regular PySpark function API in general.

Before Spark 3.0, Pandas UDFs used to be defined with `pyspark.sql.functions.PandasUDFType`. From Spark 3.0 with Python 3.6+, you can also use Python type hints. Using Python type hints is preferred and using `pyspark.sql.functions.PandasUDFType` will be deprecated in the future release.

Vectorized UDFs are more efficient compared to normal UDFs. If UDF must be used, then consider using Vectorized UDF as an alternative.

CLUSTER OPTIMIZATION

Choosing the configurations for a Databricks cluster involves considering several factors. Spark clusters in Databricks can be designed using the *Compute* section. Determining the right cluster configuration is very important for managing costs and data for different types of workloads. A cluster that's used concurrently by several data analysts shall have different requirements from a cluster which is meant for real-time streaming or machine learning workloads. Before we decide on a Spark cluster configuration, we need to have the below considerations checked off:

- A. Who will be the primary user of the cluster? It could be a data engineer, data scientist, data analyst, or machine learning engineer.
- B. What kind of workloads run on the cluster? It could be an **Extract, Transform, and Load (ETL)** process for a data engineer or exploratory data analysis for a data scientist. An ETL process could also be further divided into batch and streaming workloads.
- C. What is the **service-level agreement (SLA)** that needs to be met for the organization using Databricks to build an enterprise data platform?
- D. What are the constraints on budget and cost for the workloads?

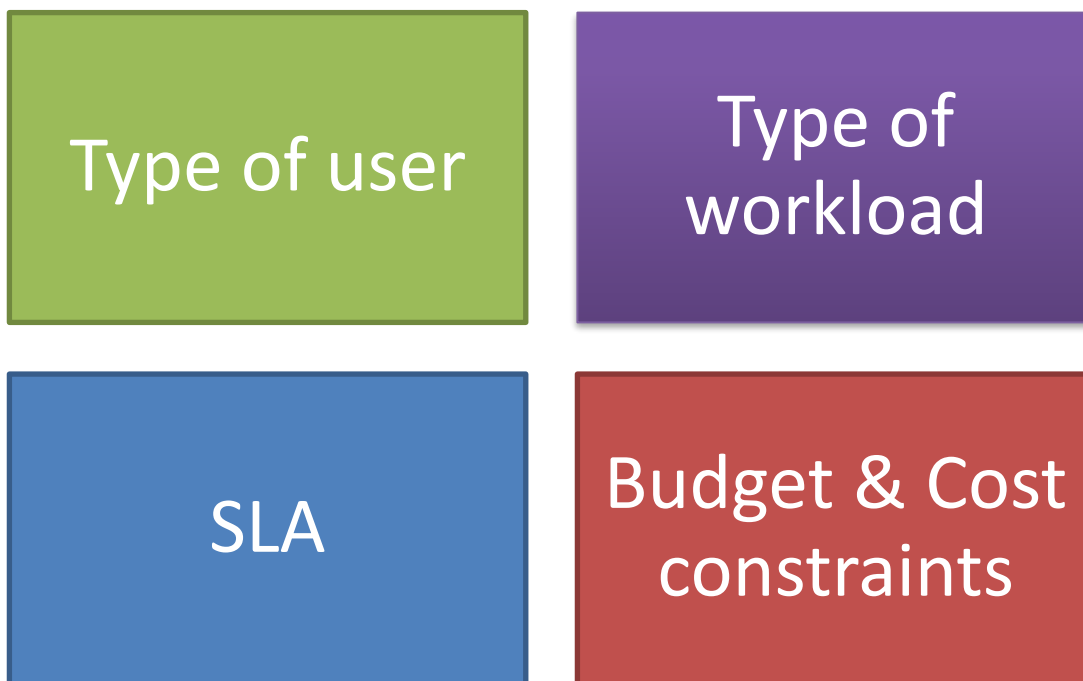


Figure 14 Cluster design considerations

Understanding cluster types

Spark clusters are broadly categorized into **all-purpose** and **job clusters**. All-purpose clusters can be used for several purposes such as **ad hoc analysis**, **data engineering development**, and **data exploration**. On the other hand, job clusters are spawned to perform a particular workload (an ETL process and more) and are automatically terminated as soon as the job finishes.

The best practice here is to do the development work using an all-purpose cluster and when the development needs to be run in production, a job cluster should be used. Doing this ensures that a production job does not unnecessarily keep a Spark cluster active, thereby reducing usage and cost.

Regarding cluster modes, there are three types available for all-purpose clusters:

- A. **Standard**: This mode is the most frequently used by users. They work well to process big data in parallel but are not well suited for sharing with many users concurrently.
- B. **Single Node**: Here, a cluster with only a driver and no workers is spawned. A single-node cluster is used for smaller workloads or use cases, wherein data needs to be processed in a non-distributed fashion.
- C. **High Concurrency**: As the name suggests, a high concurrency cluster is ideal for use among many users at the same time. It is ideal for ad hoc analytical use cases. It is also recommended to enable autoscaling when using a high concurrency cluster.

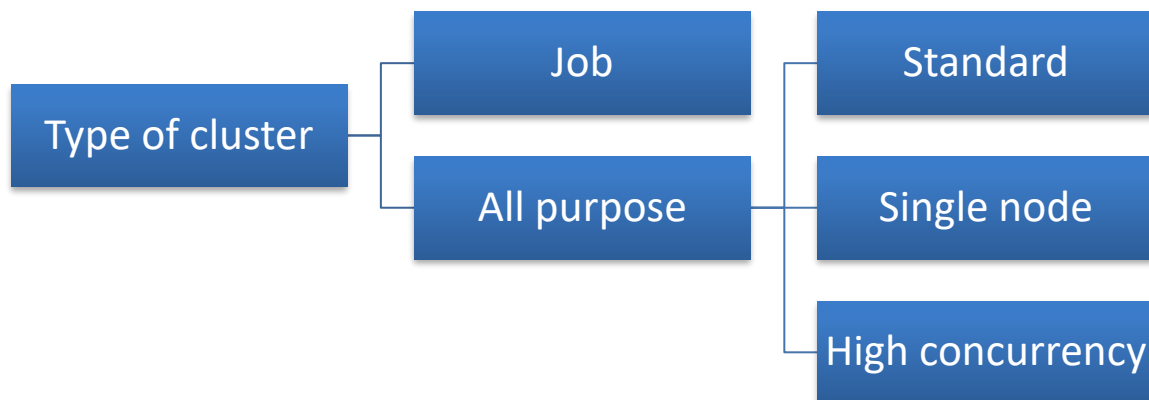


Figure 15 Types of clusters

Spot Instances

Spot instances in Azure Databricks leverage **Azure Spot VMs**.

Azure Spot VMs allow customers to purchase VMs from a pool of unused spare capacity at a significantly lower price than on-demand instances. The costs can be up to 90% less. The lower cost comes with the provision that these Azure spot instances can be taken away with minimal warning if demand for capacity increases or instances are needed to service reserved instances or pay-as-you-go customers.

Nevertheless, this does not disrupt the existing Spark jobs as the taken away spot instances are automatically replaced by on-demand instances. Hence, spot instances are ideal for workloads in Databricks that can tolerate interruptions (preferably batch processes) and latency in compute.

When creating a new cluster, there is a checkbox for **Spot Instances** in the creation page, which must be enabled in order to use spot VMs.

Auto-Scaling

Autoscaling means automatically scaling cluster worker nodes via Databricks. Autoscaling helps (increase or decrease worker nodes) based on workloads. For example, say a cluster with autoscaling enabled has been instantiated with a minimum of two and a maximum of five worker nodes. Naturally, the cluster will start functioning with two worker nodes and as the processing load on the cluster increases, it will automatically scale up to increase the number of worker nodes. Although it takes some time for the cluster to scale up, autoscaling is greatly beneficial for lowering costs:

- A. Compared to a fixed-size cluster (no autoscaling), a cluster with autoscaling enabled only brings on new node instances where there is an actual need. This helps reduce costs.
- B. At times, it is difficult to decide on the cluster's size before running Spark jobs. Hence, there are chances that a cluster can get under-provisioned. Scaling up such a cluster can take more time and manual effort as somebody needs to edit the cluster configuration and then restart it. In such scenarios, autoscaling helps save a lot of time and effort. Similarly, for long-running Spark jobs, autoscaling can be beneficial as there might be a time where the cluster might remain idle or may be under-utilized. At such times, autoscaling can help reduce a cluster's size, thereby lowering costs.
- C. There is also support for *autoscaling local storage*. This means that Databricks keeps an eye on disk space utilization across the cluster worker nodes. When a worker starts running low on the amount of free disk space, Databricks automatically provides an additional managed disk.

Pools

Databricks pools reduce cluster start and auto-scaling times by maintaining a set of idle, ready-to-use instances. When a cluster is attached to a pool, cluster nodes are created using the pool's idle instances. If the pool has no idle instances, the pool expands by allocating a new instance from the instance provider to accommodate the cluster's request. When a cluster releases an instance, it returns to the pool and is free for another cluster to use. Only clusters attached to a pool can use that pool's idle instances.

Databricks does not charge for DBUs when the instances are idle in the pool. Charges are only applied by the instance provider; that is, Azure. Pools are a great way to minimize processing time as cluster start up time reduces drastically.

Creating a pool

To create a pool, head over to the Databricks workspace. Then, click on **Compute**, select **Pools**, and click on **+ Create Pool**. This will open a page where we need to define the pool's configuration, as shown in the following screenshot:

The screenshot shows the 'Create Pool' configuration page in Azure Databricks. The page has a breadcrumb 'Clusters / Pools / Create Pool' and a title 'Create Pool' with 'Cancel' and 'Create' buttons. The configuration fields are as follows:

- Name:** A text input field. Callout: 'A user provided name for the pool'.
- Min Idle:** A text input field with the value '0'. Callout: 'Minimum number of idle instances that will be contained in the pool at any given time. These instances do not terminate and when consumed by a cluster, they will be replaced by another set of idle instances'.
- Max Capacity:** A text input field with the value 'Optional'. Callout: 'Maximum number of instances that can be present in the pool at any given time. It is an optional field but when set, it considers both the idle and used instances.'
- Idle Instance Auto Termination:** A section with a sub-label 'Terminate instances above minimum after' and a text input field with the value '60', followed by the text 'minutes of idle time.'. Callout: 'The time after which the instances in the pool (above the Min Idle limit) will get terminated by the pool when idle'.
- Instance Type:** A dropdown menu showing 'Standard_DS3_v2' and '14 GB Memory, 4 Cores'. Callout: 'The type of instance nodes that will be provisioned in the pool. The instance type remains the same for both drivers and workers'.
- Preloaded Databricks Runtime Version:** A dropdown menu showing 'None'. Callout: 'This helps speed up the cluster's launch time using idle instances from the pool. When a cluster is created with the same DBR version as the preloaded DBR, the cluster launches even quicker.'
- Instances:** A section with a sub-label 'Tags' and a text input field. Callout: 'Pool tags help monitor the costs of the different resources that are part of the Azure environment. These tags are applied as key-value pairs of Azure's resources, such as virtual machines and disks.'
- On-demand/Spot:** A section with two radio buttons: 'All On-demand' (selected) and 'All Spot'. Callout: 'To save costs, pool instances can be defined as spot instances as opposed to the default on-demand instances. In this case, the driver and worker instances from the pool will be provisioned as spot instances in the cluster.'

Figure 16 Creating a pool in Azure Databricks

When we create a cluster, we can configure it so that the driver and worker nodes are picked up from the pool of our choice. If the worker nodes and driver nodes have different node type requirements, then separate Pools must be set up.

Best practices for pools

- A. It is advisable to set the **Min Idle** configuration to **0** to avoid paying instance provider charges to Azure for idle instances. However, this can lead to an increase in cluster start up time.
- B. If a cluster using the pool requests for more instances from the pool, that request will automatically fail. The error that will be thrown will be `INSTANCE_POOL_MAX_CAPACITY_FAILURE`. Azure Databricks recommends only setting a value for **Max Capacity** when the cost being incurred from the Pools must be capped.
- C. Try to set the **Min Idle** and **Idle Instance Auto Termination** configurations in such a way that even if the minimum idle instances are set to **0**, the warmed instances remain in the pool long enough so that they can be consumed by a cluster. This will ensure effective usage and optimized costs for the workloads.
- D. Pools must be configured with on-demand instances when there are short execution times expected with strict SLAs.
- E. Pools must be configured with spot instances when cost savings are a priority over execution times and job reliability.
- F. Use proper **Preloaded Databricks Runtime Version**. This helps speed up the cluster's launch time using idle instances from the pool. When a cluster is created with the same DBR version as the preloaded DBR, the cluster launches even quicker.

Auto-termination

While creating a new cluster in Databricks, we get the option to specify the auto-termination time limit. This time limit ensures that the cluster automatically terminates due to inactivity. By default, this limit is set to *120 minutes*. Decreasing this limit can help reduce costs as DBUs keep getting charged even when the cluster is inactive before being terminated.

Cluster Sizing

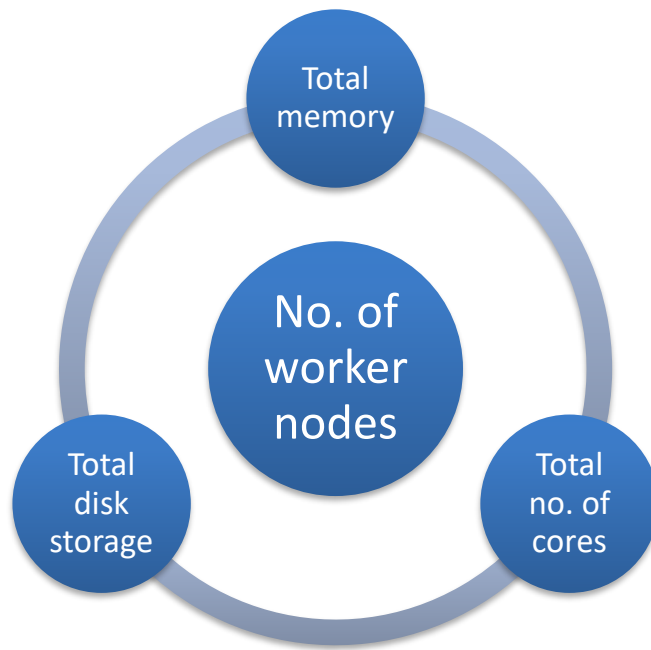


Figure 17 Cluster sizing aspects

Deciding on a **cluster size** involves determining:

- A. *The total number of cores in the cluster:* This is the maximum level of parallelism that a cluster can achieve.
- B. *The total memory or RAM in the cluster:* Since a lot of data is cached while processing Spark jobs, considering memory is also very important while designing a cluster.
- C. *Total executor local storage:* Local storage in executors store data when it gets spilled due to shuffling or caching.
- D. *Total number of worked nodes:* The number of workers for a workload.

Scenario based cluster sizing

General principles for determining the cluster size based on the workload nature:

- A. A shuffle operation is involved when the data is moved across the nodes, and this happens when we are using transformations such as average and sum. These transformations need to collect data from various executors and send that to the driver node. When a considerable amount of shuffling is anticipated, try to go for compute-optimized cluster and fewer number of nodes. For e.g., data analytics where slicing and dicing, aggregations, joins will be the main flavour, or complex ETL batch processes, where costly joins and group by operations are anticipated. Compute-optimized clusters provide larger number of CPU cores and thus more parallelism. Fewer nodes ensure network and disk I/O is minimized from shuffling.

- B. Data analytics workloads benefit from high concurrency clusters, where several data analysts can work on data at the same time.
- C. Job clusters should be used to productionize batch and streaming ETL workloads.
- D. Machine learning workloads require large datasets to be stored in memory for model training purpose. So go for memory-optimized clusters for this scenario. GPU based clusters can be considered if compute and memory requirements are considerably high.
- E. Pools should be considered because of 2 benefits they confer in case of all-purpose clusters-
 - o They ensure consistent cluster configurations.
 - o They help reduce the total job runtime as the cluster startup time is decreased.
- F. Auto-scaling and auto-termination should be configured correctly for all-purpose clusters.

SCENARIO	DATA ANALYTICS	SIMPLE BATCH ETL	COMPLEX BATCH ETL	STREAMING ETL	MACHINE LEARNING
FEATURE					
COMPUTE/MEMORY OPTIMIZED	Compute-optimized cluster				Memory-optimized cluster
TYPE OF CLUSTER	High concurrency cluster	Job cluster for productionizing All-purpose for development, exploration, debugging			All-purpose
NUMBER OF NODES	Fewer nodes		Fewer nodes		Fewer nodes
POOL	G. Ensures consistent cluster configurations. H. Reduces the total job runtime as the cluster startup time is decreased.				

Databricks Runtime Version

- A. Use latest runtime version for all-purpose clusters to get the benefits of latest features and optimizations.
- B. For job clusters, prefer the Long-Term Support (LTS) versions for their robustness and reliability
- C. For ML, Deep Learning, Neural Network scenarios, go for the ML optimized runtime versions.

REFERENCES

Understanding Query Plans and Spark UIs - Xiao Li Databricks

<https://youtu.be/YgQgJceojJY>

Apache Spark Core – Practical Optimization Daniel Tomes (Databricks)

<https://www.youtube.com/watch?v=ArCesEIWp8>

Advancing Spark - Understanding the Spark UI

<https://www.youtube.com/watch?v=rNpzrkB5KQQ>

Databricks related YouTube playlist

https://www.youtube.com/playlist?list=PLHN2ijxAWBaMY4_1xMhOoeRWiZ6_d8F9s

Reading DAGs

<https://dzone.com/articles/reading-spark-dags>

Bucketing

<https://blog.clairvoyantsoft.com/bucketing-in-spark-878d2e02140f>

Apache Spark 3.0 Optimizations

[Introducing Apache Spark 3.0 - The Databricks Blog](#)

The Internals of Spark SQL online book

<https://jaceklaskowski.gitbooks.io/mastering-spark-sql/content/>

Salting

<https://stackoverflow.com/questions/58110207/spark-how-does-salting-work-in-dealing-with-skewed-data>

Cache and Persist

<https://sparkbyexamples.com/spark/spark-difference-between-cache-and-persist/#:~:text=Spark%20Cache%20vs%20Persist&text=Both%20caching%20and%20persisting%20are,the%20user%2Ddefined%20storage%20level.>

Partition

<https://docs.databricks.com/delta/quick-start.html#partition-data&language-sql>

Optimize, Data skipping and Z-Ordering. This page gives guidelines on which column can be used for ZORDER BY. FAQ at bottom of this website tells how often we should run OPTIMIZE

<https://docs.databricks.com/delta/optimizations/file-mgmt.html>

Guidelines on how to choose the right partition column

<https://docs.databricks.com/delta/best-practices.html>

Courses

Coursera course on Apache Spark

[Distributed Computing with Spark SQL | Coursera](#)

Books

Spark: The Definitive Guide, by Bill Chambers, Matei Zaharia, Released February 2018, Publisher(s): O'Reilly Media, Inc., ISBN: 9781491912218

Beginning Apache Spark Using Azure Databricks: Unleashing Large Cluster Analytics in the Cloud, by Robert Ilijason, Released June 2020, Publisher(s): Apress, ISBN: 9781484257814