# Introduction

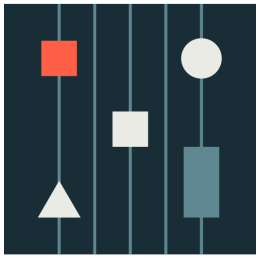
This document aims to compile most (if not all) of the essential Databricks, Apache Spark™, and Delta Lake best practices and optimization techniques in or place. All data engineers and data architects can use it as a guide when designing and developing optimized and cost-effective and efficient data pipelines. ( should not be treated as an afterthought but as one of the most important nonfunctional requirements right at the inception of the project. As a result, all of th practices discussed in this book should be considered throughout the development and productionization of data pipelines. This book is divided into several sections, each focusing on a particular problem statement and deep diving into providing all the possible solutions for it.

# Delta Lake — The Lakehouse Format

Delta Lake is an open format storage layer that delivers reliability, security and performance on your data lake. It enables building a Lakehouse architecture compute engines including Spark, PrestoDB, Flink, Trino, and Hive, and APIs for Python, SQL, Scala, Java, Rust, and Ruby. It has many benefits over other formats like Parquet, Avro, ORC, etc. — for example:

- Delta is a protocol to guarantee ACID transactions, high performance and a ton of other features on top of the open source Parquet format. Delta is ope source and the complete protocol can be found here.
- Delta Lake supports ACID transactions and unifies batch and streaming paradigms, simplifying delete/insert/update transactions on incremental data
- Delta allows you to time travel and to read a point-in-time snapshot version of the table
- Delta also comes with many performance enhancements around efficient data layout, indexation, data skipping and caching, etc.

Therefore, it is highly recommended to use Delta as the default data lake storage format to reap all the benefits. On Databricks DBR 8.x and above, Delta La the default format.



# Underlying Data Layout

Underneath the hood of a Delta table are Parquet files that store the data. It also contains a subdirectory named *_delta_log* that stores the Delta transactio right next to the Parquet files. The size of these Parquet files is really crucial for query performance.

The tiny files problem is a well-known problem in the big data world. When a table has too many underlying tiny files, read latency suffers as a result of the ti spent for just opening and closing those tiny files.

To avoid this problem, it's best to have a file size between 16MB and 1GB, which is configurable on a case-by-case basis based on the workload and the spe use case.

If you are using Databricks Runtime 11.3 and above to create managed Delta tables cataloged in Unity Catalog (Databricks' data catalog), you don't need to about optimizing the underlying file sizes or configuring a target file size for your Delta tables because Databricks will carry out this task automatically in the background as part of the auto-tuning capability. In the future, this will also be available for external tables.

For the remaining cases, you would need to manually compact files together using specific Spark jobs to obtain the appropriate file size, but this isn't necess Delta tables because they come with out-of-the-box bin packing (compaction) features. In Delta, bin packing can be accomplished in two ways, as detailed b

## 1. Optimize & Z-order

OPTIMIZE compacts the files to get a file size of up to 1GB, which is configurable. This command basically attempts to size the files to the size that you have configured (or 1GB by default if not configured). You can also combine the OPTIMIZE command with the ZORDER, which physically sorts or co-locates data chosen column(s).

SQL (AUTO-DETECTED)

```
OPTIMIZE table_name [WHERE predicate]
[ZORDER BY (col_name1 [, ...] ) ]
```

- Always choose high cardinality columns (for example: `customer_id` in an orders table) for Z-ordering. Date columns are usually l cardinality columns, so they should not be used for Z-order — they are a better fit as the partitioning columns (but you don't alway to partition the tables; please refer to the Partitioning section below for more details). For Z-order, choose the columns that are mc frequently used in filter clauses or as join keys in the downstream queries.
- Never use more than 4 columns since too many columns will degrade Z-ordering effectiveness
- Always run **OPTIMIZE** command on a separate job cluster and not as part of the job itself; otherwise, it might impact the correspo job's SLA
- Compute-optimized instance family is recommended for the **OPTIMIZE** command as it's a compute-intensive operation
- **OPTIMIZE** (with or without **ZORDER**) should be done on a regular basis, such as once a day (or weekly or as per your requireme maintain a good file layout for better downstream query performance

# 2. Auto optimize

Auto optimize, as the name suggests, automatically compacts small files during individual writes to a Delta table, and by default, it tries to achieve a file size 128MB. It comes with two features:

### 1. Optimize Write

Optimize Write dynamically optimizes Apache Spark partition sizes based on the actual data, and attempts to write out 128MB files for each table partition. It inside the same Spark job.

### 2. Auto Compact

Following the completion of the Spark job, Auto Compact launches a new job to see if it can further compress files to attain a 128MB file size.

PYTHON (AUTO-DETECTED)

```
-- Table properties
delta.autoOptimize.optimizeWrite = true
delta.autoOptimize.autoCompact = true

-- In Spark session conf for all new tables
set spark.databricks.delta.properties.defaults.autoOptimize.optimizeWrite = true
set spark.databricks.delta.properties.defaults.autoOptimize.autoCompact = true
```

- Always enable *optimizeWrite* table property if you are not already leveraging the manual **OPTIMIZE** (with or without **ZORDER**) co to get a decent file size. Keep in mind that optimized writes require the shuffling of data according to the partitioning structure of th table. This shuffle naturally incurs additional cost. However, the throughput gains during the write may pay off the cost of the shuffl the throughput gains when querying the data should still make this feature worthwhile.
- If your job isn't strictly SLA bound, then in that case you can also enable *autoCompact* table property to further take advantage of bin packing

# 3. Partitioning

Partitioning can speed up your queries if you provide the partition column(s) as filters or join on partition column(s) or aggregate on partition column(s) or me partition column(s), as it will help Spark to skip a lot of unnecessary data partition (i.e., subfolders) during scan time.

- Databricks recommends not to partition tables under 1TB in size and let ingestion time clustering automatically take effect. This feature will cluster the data based on the order the data was ingested by default for all tables.
- You can partition by a column if you expect data in each partition to be at least 1GB
- Always choose a low cardinality column — for example, year, date — as a partition column
- You can also take advantage of Delta's generated columns feature while choosing the partition column. Generated columns ar special type of column whose values are automatically generated based on a user-specified function over other columns in the table.

# 4. File size tuning

In cases where the default file size targeted by Auto-optimize (128MB) or Optimize (1GB) isn't working for you, you can fine-tune it as per your requirement. set the target file size by using `delta.targetFileSize` table property and then Auto-optimize and Optimize will binpack to achieve the specified size instea

```
-- Table properties
delta.targetFileSize = 134217728
```

You can also delegate this task to Databricks if you don't want to manually set the target file size by turning on the `delta.tuneFileSizesForRewrites` tabl property. When this property is set to true, Databricks will automatically tune the file sizes based on workloads. For example, if you do a lot of merges on the table, then the files will automatically be tuned to much smaller sizes than 1GB to accelerate the merge operation.

```
-- Table properties
delta.tuneFileSizesForRewrite = true
```

# Data Shuffling — Why It Happens and How to Control It

Data shuffle occurs as a result of wide transformations such as joins, aggregations and window operations, among others. It is an expensive process since it data over the network between worker nodes. We may use a few optimization approaches to either eliminate or improve the efficiency and speed of shuffles

## 1. Broadcast hash join

To entirely avoid data shuffling, broadcast one of the two tables or DataFrames (the smaller one) that are being joined together. The table is broadcast by the who copies it to all worker nodes.

When executing joins, Spark automatically broadcasts tables less than 10MB; however, we may adjust this threshold to broadcast even larger tables, as demonstrated below:

PYTHON (AUTO-DETECTED)

```
set spark.sql.autoBroadcastJoinThreshold = <size in bytes>
```

When you know that some of the tables in your query are small tables, you can tell Spark to broadcast them explicitly using hints, which is a recommended c

SQL (AUTO-DETECTED)

```
SELECT /*+ BROADCAST(t) */ * FROM <table-name> t
```

Spark 3.0 and above comes with AQE (Adaptive Query Execution), which can also convert the sort-merge join into broadcast hash join (BHJ) when the runti statistics of any join side is smaller than the adaptive broadcast hash join threshold, which is 30MB by default. You can also increase this threshold by chang following configuration:

PYTHON (AUTO-DETECTED)

```
set spark.databricks.adaptive.autoBroadcastJoinThreshold = <size in bytes>
```

- Note that broadcast hash join is not supported for a full outer join. For the right outer join, only the left-side table can be broadcast, a other left joins only the right-side table can be broadcast.
- If you're running a driver with a lot of memory (32GB+), you can safely raise the broadcast thresholds to something like **200MB**

PYTHON (AUTO-DETECTED)

```
set spark.sql.autoBroadcastJoinThreshold = 209715200;
set spark.databricks.adaptive.autoBroadcastJoinThreshold = 209715200;
```

- Always explicitly broadcast smaller tables using hints or PySpark broadcast function
- Why do we need to explicitly broadcast smaller tables if AQE can automatically broadcast smaller tables for us? The reason for this AQE optimizes queries while they are being executed.
  a. Spark needs to shuffle the data on both sides and then only AQE can alter the physical plan based on the statistics of the shuffl and convert to broadcast join

  b. Therefore, if you explicitly broadcast smaller tables using hints, it skips the shuffle altogether and your job will not need to wait f AQE's intervention to optimize the plan
- Never broadcast a table bigger than 1GB because broadcast happens via the driver and a 1GB+ table will either cause OOM on the or make the drive unresponsive due to large GC pauses
- Please take note that the size of a table in disk and memory will never be the same. Delta tables are backed by Parquet files, which have varying levels of compression depending on the data. And Spark might broadcast them based on their size in the disk — howe they might actually be really big (even more than 8GB) in memory after the decompression and conversion from column to row form Spark has a hard limit of 8GB on the table size it can broadcast. As a result, your job may fail with an exception in this circumstance case, the solution is to either disable broadcasting by setting `spark.sql.autoBroadcastJoinThreshold` to -1 and do the explicit broadcast using hints (or the PySpark broadcast function) of the tables that are really small in the disk as well as in memory, or set the `spark.sql.autoBroadcastJoinThreshold` to smaller values like 100MB or 50MB instead of setting the threshold to -1.
- The driver can only collect up to 1GB of data in memory at any given time, and anything more than that will trigger an error in the dri causing the job to fail. However, since we want to broadcast tables larger than 10MB, we risk running into this problem. This problen be solved by increasing the value of the following driver configuration.
  - Please keep in mind that because this is a driver setting; it cannot be altered once the cluster is launched. Therefore, it should b under the cluster's advanced options as a Spark config. Setting this parameter to 8GB for a driver with >32GB memory seems t fine in most circumstances. In certain cases where the broadcast hash join is going to broadcast a very large table, setting this 16GB would also make sense.
  - In Photon, we have the executor-side broadcast. So, you don't have to change the following driver configuration if you use a Da Runtime (DBR) with Photon.

PYTHON (AUTO-DETECTED)

```
spark.driver.maxResultSize 16g
```

## 2. Shuffle hash join over sort-merge join

In most cases Spark chooses sort-merge join (SMJ) when it can't broadcast tables. Sort-merge joins are the most expensive ones. Shuffle-hash join (SHJ) h found to be faster in some circumstances (but not all) than sort-merge since it does not require an extra sorting step like SMJ. There is a setting that allows y advise Spark that you would prefer SHJ over SMJ, and with that Spark will try to use SHJ instead of SMJ wherever possible. Please note that this does not r that Spark will always choose SHJ over SMJ. We are simply defining your preference for this option.

PYTHON (AUTO-DETECTED)

```
set spark.sql.join.preferSortMergeJoin = false
```

Databricks Photon engine also replaces sort-merge join with shuffle hash join to boost the query performance.

- Setting the `preferSortMergeJoin` config option to false for each job is not necessary. For the first execution of a concernec you can leave this value to default (which is true).
- If the job in question performs a lot of joins, involving a lot of data shuffling and making it difficult to meet the desired SLA, th can use this option and change the `preferSortMergeJoin` value to false

# 3. Leverage cost-based optimizer (CBO)

Spark SQL can use a Cost-based optimizer (CBO) to improve query plans. This is especially useful for queries with multiple joins. The CBO is enabled by de You disable the CBO by changing the following configuration:

PYTHON (AUTO-DETECTED)

```
set spark.sql.cbo.enabled = false
```

For CBO to work, it is critical to collect table and column statistics and keep them up to date. Based on the stats, CBO chooses the most economical join stra So you will have to run the following SQL command on the tables to compute stats. The stats will be stored in the Hive metastore.

SQL (AUTO-DETECTED)

```
ANALYZE TABLE table_name COMPUTE STATISTICS FROM COLUMNS col1, col2, ...;
```

## Join reorder

For faster query execution CBO can also use the stats calculated by ANALYZE TABLE command to find the optimal order in which the tables should be joine instance, joining smaller tables first would significantly improve the performance). Join reordering works only for INNER and CROSS joins. To leverage this f set the following configurations:

PYTHON (AUTO-DETECTED)

```
set spark.sql.cbo.enabled = true
set spark.sql.cbo.joinReorder.enabled = true
set spark.sql.statistics.histogram.enabled = true
-- CostBasedJoinReorder requires statistics on the table row count at the very least and its accuracy is improved by having
ANALYZE TABLE table_name COMPUTE STATISTICS FOR COLUMNS col1, col2..;
-- The maximum number of tables in a query for which this joinReorder can be used (default is 12)
set spark.sql.cbo.joinReorder.dp.threshold = <number of tables>
```

- To properly leverage CBO optimizations, ANALYZE TABLE command needs to be executed regularly (preferably once per day or data has mutated by more than 10%, whichever happens first)
- When Delta tables are recreated or overwritten on a daily basis, the ANALYZE TABLE command should be executed immediately the table is overwritten as part of the same job or pipeline. This will have an impact on your pipeline's overall SLA. As a result, in c like this, there is a trade-off between better downstream performance and the current job's execution time. If you don't want CBO optimization to affect the current job's SLA, you can turn it off.
- Never run ANALYZE TABLE command as part of your job. It should be run as a separate job on a separate job cluster. For examp can be run inside the same nightly notebook running commands like Optimize, Z-order and Vacuum.
- Leverage join reorder where a lot of inner-joins and/or cross-joins are being performed in a single query
- Spark's Adaptive Query Execution (AQE), which changes the query plan on the fly during runtime to a better one, also takes adva the statistics calculated by ANALYZE TABLE command. Therefore, it's recommended to run ANALYZE TABLE command regularly keep the table statistics updated.

# Data Spilling — Why It Happens and How to Get Rid of It

The default setting for the number of Spark SQL shuffle partitions (i.e., the number of CPU cores used to perform wide transformations such as joins, aggreg and so on) is 200, which isn't always the best value. As a result, each Spark task (or CPU core) is given a large amount of data to process, and if the memor available to each core is insufficient to fit all of that data, some of it is spilled to disk. Spilling to disk is a costly operation, as it involves data serialization, de-serialization, reading and writing to disk, etc. Spilling needs to be avoided at all costs and in doing so, we must tune the number of shuffle partitions. There a couple of ways to tune the number of Spark SQL shuffle partitions as discussed below.

## 1. AQE auto-tuning

Spark AQE has a feature called `autoOptimizeShuffle` (AOS), which can automatically find the right number of shuffle partitions. Set the following configura enable auto-tuning:

PYTHON (AUTO-DETECTED)

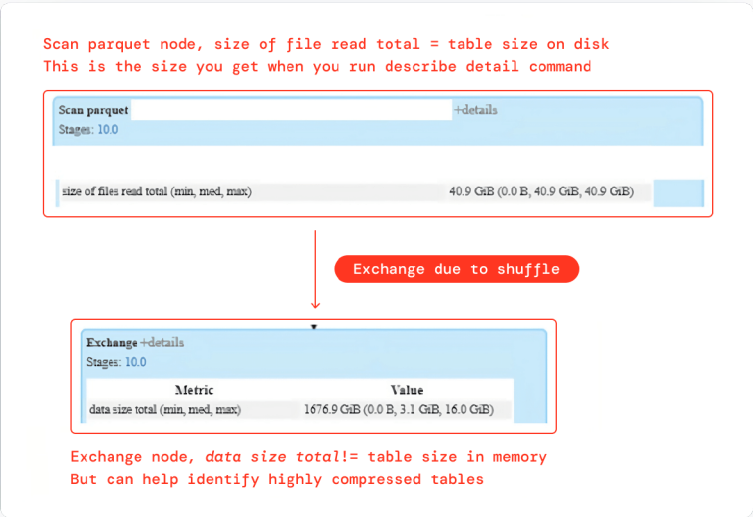
```
set spark.sql.shuffle.partitions=auto
```

### Caveat: unusually high compression

There are certain limitations to AOS. AOS may not be able to estimate the correct number of shuffle partitions in some circumstances where source tables ha unusually high compression ratio (20x to 40x).

There are two ways you can identify the highly compressed tables:

**1. Spark UI SQL DAG**




Although "data size total" metrics in the Exchange node don't provide the exact size of a table in memory, it can definitely help identify the highly compressed Scan Parquet node provides the precise size of a table in the disk. The Exchange node data size in the aforementioned case is 40x larger than the size on th indicating that the table is probably heavily compressed on the disk.

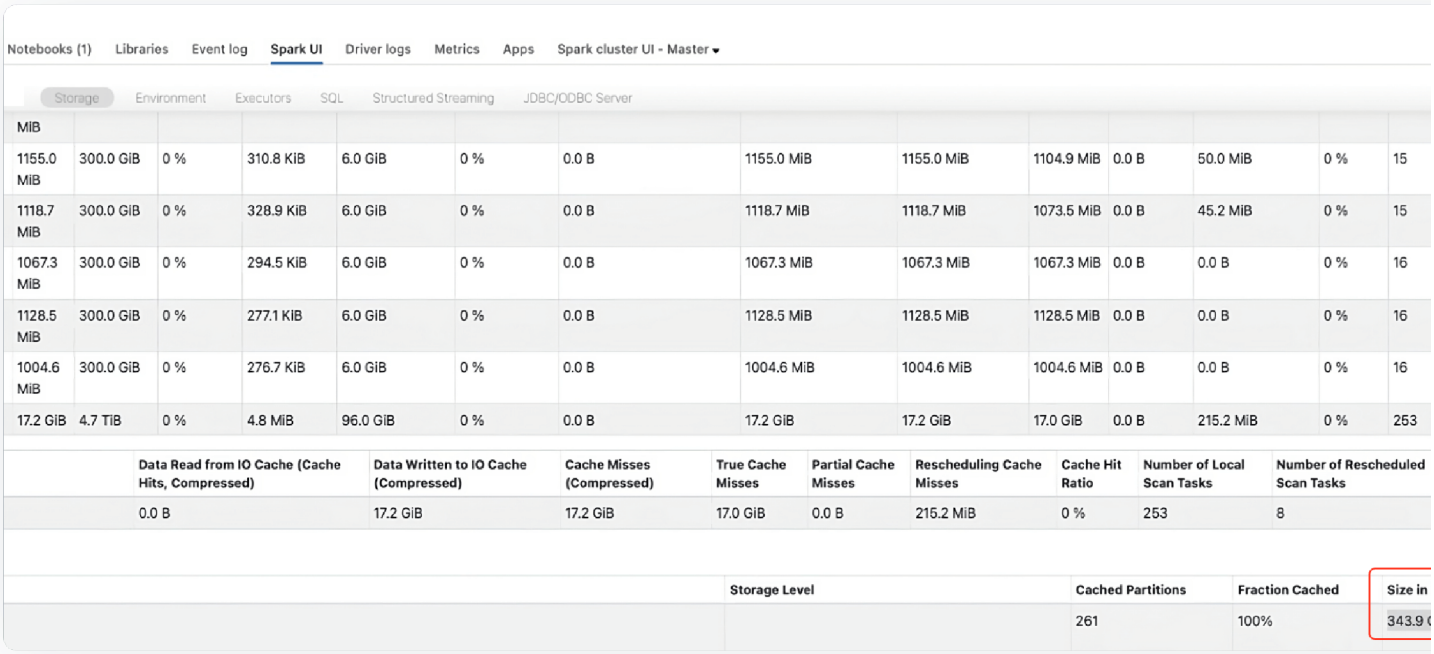
**2. Cache the table**

A table can be cached in memory to figure out its actual size in memory. Here's how to go about it:

PYTHON (AUTO-DETECTED)

```
-- count here is forcing the cache materialization
spark.table("table").cache().count()
```

Refer to the storage tab of Spark UI to find the size of the table in memory after the command above has been completed:




**Solution:** To counter this effect, reduce the value of the per partition size used by AQE to determine the initial shuffle partition number (default 128MB) as fo

PYTHON (AUTO-DETECTED)

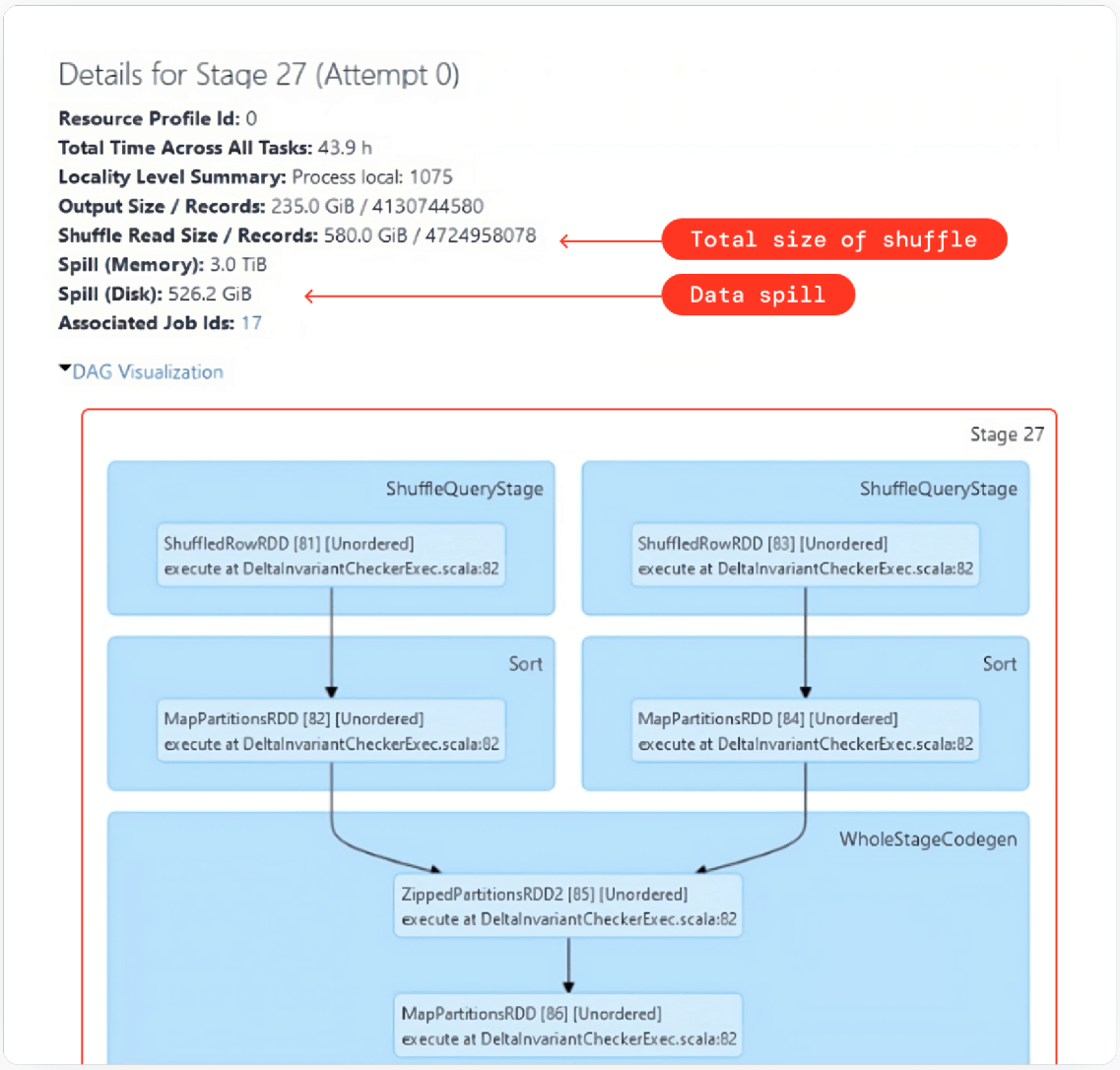
```
-- setting to 16MB for example
set spark.databricks.adaptive.autoOptimizeShuffle.preshufflePartitionSizeInBytes = 16777216
```

After lowering the `preshufflePartitionSizeInBytes` value to 16MB, if AOS is still calculating the incorrect number of partitions and you are still experienc large data spills, you should further lower the `preshufflePartitionSizeInBytes` value to 8MB. If this still doesn't resolve your spill issue, it is best to disat and manually tune the number of shuffle partitions as explained in the next section.

## 2. Manually fine-tune

To manually fine-tune the number of shuffle partitions we need:

1. The total amount of shuffled data. To do so, run the Spark query once, and then use the Spark UI to retrieve this value, as demonstrated in the example




2. As a rule of thumb, we need to make sure that after tuning the number of shuffle partitions, each task should approximately be processing **128MB** to **20** data. You can see this value in the summary metrics for the shuffle stage in Spark UI, as shown in the example below:


So here is the formula to compute the right number of shuffle partitions:

PYTHON (AUTO-DETECTED)

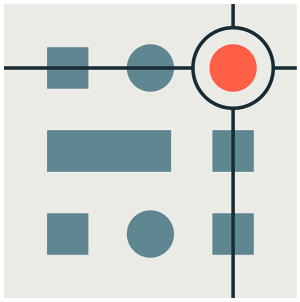
```
Let's assume that:
Total number of total worker cores in cluster = T
Total amount of data being shuffled in shuffle stage (in megabytes) = B
Optimal size of data to be processed per task (in megabytes) = 128
Hence the multiplication factor (M): M = ceiling(B / 128 / T)
And the number of shuffle partitions (N): N = M x T
Note that we have used the ceiling function here to ensure that all the cluster cores are fully engaged till the very last
```

- The optimal size of data to be processed per task should be 128MB approximately. It works well in most cases. It might not work is some sort of data explosion happening in your query. You might need to choose a smaller value in that case. We will have a s on data explosion later in this document.
- If you are neither using auto-tune (AOS) nor manually fine-tuning the shuffle partitions, then as a rule of thumb set this to twice, thrice, the number of total worker CPU cores

PYTHON (AUTO-DETECTED)

```
-- in SQL
set spark.sql.shuffle.partitions = 2*<number of total worker cores in cluster>
-- in PySpark
spark.conf.set("spark.sql.shuffle.partitions", 2*<number of total worker cores in cluster>)
-- or
spark.conf.set("spark.sql.shuffle.partitions", 2*sc.defaultParallelism)
```

- Because there may be multiple Spark SQL queries in a single notebook, fine-tuning the number of shuffle partitions for each que time-consuming task. So, our advice is to fine-tune it for the largest query with the greatest number for the total amount of data b shuffled for a shuffle stage and then set that value once for the entire notebook.
- If there is skewness in data, then fine-tuning the shuffle partitions will not help with data spills. In that case, you should first get ri data skew. Please refer to the next section on data skew for more details.

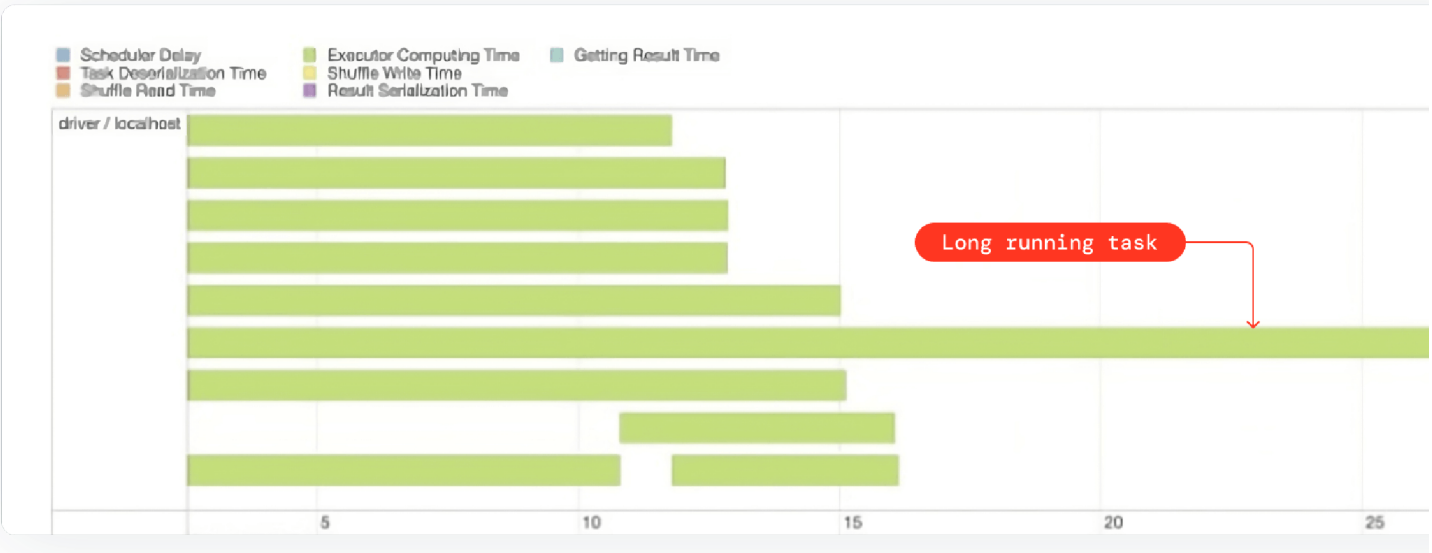


# Data Skewness — Identification and Remediation

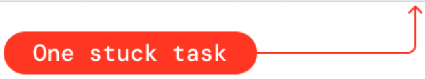
Data skew is the situation where only a few CPU cores wind up processing a huge amount of data due to uneven data distribution. For example, when you j aggregate using the columns(s) around which data is not uniformly distributed, then you will end up with a skewed shuffle stage that will take a lot of time to (might actually fail as well after several attempts).

## Identification of skew

- If all the Spark tasks for the shuffle stage are finished and just one or two of them are hanging for a long time, that's an indication of skew. You can also information from Spark UI.



| | Submitted | Duration | Tasks: Succeeded/Total | Input | Output |
|---|---|---|---|---|---|
| +details | 2018/02/18 15:12:11 | 47 h | 199/200 | | |




- In the tasks summary metrics, if you see a huge difference between the min and max shuffle read size, that's also an indication of data skewness

| Metric | Min | 25th percentile | Median | 75th percentile | Max |
|---|---|---|---|---|---|
| Duration | 29 s | 1.2 min | 1.5 min | 1.7 min | 4.0 min |
| GC Time | 77.0 ms | 0.8 s | 1 s | 1 s | 3 s |
| Spill (memory) | 2.8 GiB | 6.9 GiB | 8.5 GiB | 10.4 GiB | 19.2 GiB |
| Spill (disk) | 388.5 MiB | 1.2 GiB | 1.6 GiB | 2.2 GiB | 4.3 GiB |
| Shuffle Read Size / Records | 479.6 MiB / 5349999 | 749.9 MiB / 6923474 | 1015 MiB / 8417393 | 1.3 GiB / 9910309 | 2.3 GiB / 15914688 |
| Shuffle Write Size / Records | 477.2 MiB / 5124404 | 746.3 MiB / 6697251 | 1008.6 MiB / 8191021 | 1.2 GiB / 9683993 | 2.3 GiB / 15688494 |
| Scheduler Delay | 4.0 ms | 6.0 ms | 21.0 ms | 29.0 ms | 0.7 s |
| Task Deserialization Time | 4.0 ms | 7.0 ms | 9.0 ms | 11.0 ms | 29.0 ms |
| Shuffle Read Blocked Time | 0.0 ms | 18 s | 32 s | 51 s | 3.1 min |
| Shuffle Remote Reads | 23 MiB | 746.1 MiB | 1012.4 MiB | 1.3 GiB | 2.3 GiB |
| Result Serialization Time | 0.0 ms | 0.0 ms | 0.0 ms | 0.0 ms | 0.0 ms |
| Getting Result Time | 0.0 ms | 0.0 ms | 0.0 ms | 0.0 ms | 0.0 ms |
| Peak Execution Memory | 6 GiB | 6.1 GiB | 6.3 GiB | 7.1 GiB | 16.1 GiB |


- Even after fine-tuning the number of shuffle partitions, if there are a lot of data spills, then this might actually be because of skewness
- Lastly, you can just simply count the number of rows while grouping by join or aggregation columns. If there is an enormous disparity between the row c then it's a definite skew.

SQL (AUTO-DETECTED)

```sql
SELECT COLUMN_NAME,
COUNT(*)
FROM TABLE
GROUP BY COLUMN_NAME
```

## Skew remediation

1. **Filter skewed values**
   If it's possible to filter out the values around which there is a skew, then that will easily solve the issue. If you join using a column with a lot of null values example, you'll have data skew. In this scenario, filtering out the null values will resolve the issue.
2. **Skew hints**
   In the case where you are able to identify the table, the column, and preferably also the values that are causing data skew, then you can explicitly tell Sp about it using skew hints so that Spark can try to resolve it for you.

SQL (AUTO-DETECTED)

```sql
SELECT /*+ SKEW('table', 'column_name', (value1, value2)) */ * FROM table
```

3. **AQE skew optimization**
   Spark 3.0+'s AQE can also dynamically solve the data skew for you. It's by default enabled, but if you want to disable it, then set the following configurat false:

PYTHON (AUTO-DETECTED)

```
set spark.sql.adaptive.skewJoin.enabled = false
```

By default any partition that has at least 256MB of data and is at least 5 times bigger in size than the average partition size will be considered as a skew partition by AQE. You can also change these values to fine-tune default AQE behavior:

PYTHON (AUTO-DETECTED)

```
-- default is 5
set spark.sql.adaptive.skewJoin.skewedPartitionFactor = <value>
-- default is 256MB
set spark.sql.adaptive.skewJoin.skewedPartitionThresholdInBytes = <size in bytes>
```

4. **Salting**
   If none of the above-mentioned options work for you, the only other option is to do salting. It's a strategy for breaking a large skewed partition into smalle partitions by appending random integers as suffixes to skewed column values. Please see this example notebook (from cmd7 onwards) to fix the data s problem using the salting technique. This example shows how to fix data skew in a join query.

   Now let's imagine you have a skewed aggregation query. We can still use salting to remedy the data skew, but in this situation, we would only need to e partial salting for the value(s) that are contributing to the skewness. Let's assume that for an e-commerce organization, we want to find out the last trans date for all its customers and the data is skewed due to uneven distribution of the *customer_id* value *xyz*. So to fix this skewness, we will perform the pa salting as follows:

```
Select
   Case
      When
         salted_key like '%_salt'
      Then
         'xyz'
      Else
         salted_key
   End
   As customer_id, max(transaction_date) as last_transaction_date
From
   (
      Select
         Case
            When
               customer_id is 'xyz'
            Then
               concat(cast( 1 * FLOOR(RAND(12345)*500) AS string), '_', 'salt')
            Else
               customer_id
         End
         As salted_key, max(transaction_date) as transaction_date
      From
         customer_sales_table
      Group by
         salted_key
   )
Group by customer_id
```

Final Aggregation

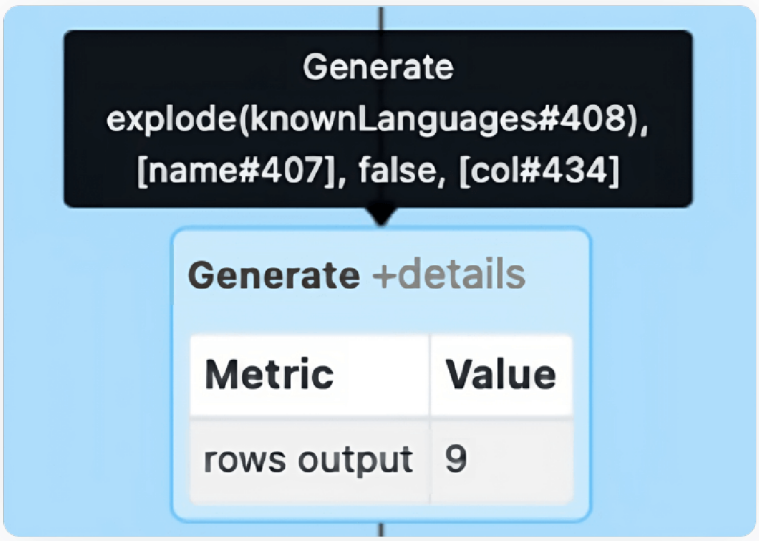
Intermediate Aggregation

Try these solutions in the order they are mentioned above. That means salting should be the last choice, not the first, as it requir changes. Hints and AQE solutions are much simpler to implement.

# Data Explosion — Identification, Consequences and Solutions

During a Spark job execution after a certain transformation step, you may see an unusually big rise in data volume, which is considered a data explosion. Th
execution is significantly slowed as a result of this. The following are some of the most prevalent transformations that can result in a data explosion:

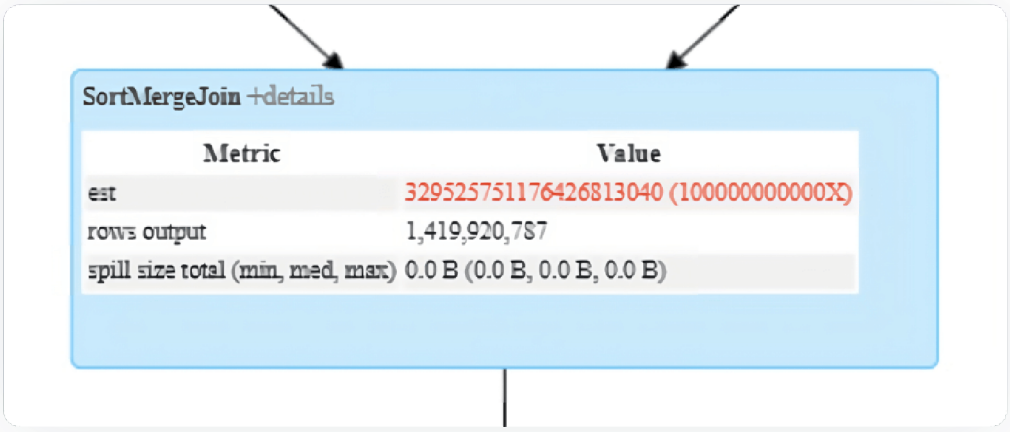
## 1. Explode function




While working with structured files with the formats like JSON, Parquet, Delta and XML, we often get data in collections like arrays, lists and maps. In such cases, `the explode()` function is useful to convert collection columns to rows in order to process in Spark effectively. This explode operation can significant increase the data volume. The explode operation is represented by the *Generate* node in Spark UI as shown in the image above.

## 2. Join operation

A common cause of slow queries is joins that produce significantly more rows than anticipated. This is often referred to as a “row explosion.” Refer to the *rov output* metric in the *SortMergeJoin* or *ShuffleHashJoin* node of Spark UI to identify a potential row explosion.




### Impact

While reading input data from sources like Parquet, Delta, etc., Spark approximately reads 128MB per task per core, which is a very decent partition size to f the memory available to each CPU core. But due to data explosion, the 128MB data might get converted into a significantly high volume (for example, a cou GBs), which is problematic as a single CPU core might not have enough memory to fit that exploded partition. As a consequence, for subsequent wide transformations, a lot of data might get spilled into the disk, which can significantly impact the query performance.

Solutions

- Decrease the size of input partitions, i.e., `spark.sql.files.maxPartitionBytes` (default 128MB), to create smaller input partitions in order to counter effect of `explode()` function. Instead of 128MB, you can choose a much smaller partition size like 16MB or 32MB, for example.

PYTHON (AUTO-DETECTED)

```
set spark.sql.files.maxPartitionBytes = <size in bytes>
```

- You can explicitly call the repartition() function right after the read statement to increase the total number of partitions. This will allow you to reduce the s each partition.
- When the explosion is happening due to a join operation, a simple solution would be to increase the number of shuffle partitions, which will decrease the the partition to much less than 128MB. Refer to the manual shuffle partitions tuning section for more information.

# Data Skipping and Pruning

The amount of data to process has a direct relationship with query performance. Therefore, it's extremely important to read only the required data and skip a unnecessary data. There are a couple of data skipping and pruning techniques that you can apply with Spark and Delta.

## 1. Delta data skipping

Delta data skipping automatically collects the stats (min, max, etc.) for the first 32 columns for each underlying Parquet file when you write data into a Delta t Databricks takes advantage of this information (minimum and maximum values) at query time to skip unnecessary files in order to speed up the queries.

To collect stats for more than the 32 first columns, you can set the following Delta property:

```
-- table property
delta.dataSkippingNumIndexedCols = <value>
```

Collecting statistics on long strings is an expensive operation. To avoid collecting statistics on long strings, you can either configure the table property `delta.dataSkippingNumIndexedCols` to avoid columns containing long strings or move columns containing long strings to a column greater than `delta.dataSkippingNumIndexedCols` using ALTER TABLE as shown below:

SQL (AUTO-DETECTED)

```
ALTER TABLE table_name ALTER [COLUMN] col_name col_name data_type [COMMENT col_comment] [FIRST|AFTER colA_name]
```

## 2. Column pruning

When reading a table, we generally choose all of the columns, but this is inefficient. To avoid scanning extraneous data, always inquire about what columns part of the workload computation and are needed by downstream queries. Only those columns should be selected from the source database. This has the p to have a significant impact on query performance.

PYTHON (AUTO-DETECTED)

```
-- SQL
SELECT col1, col2, .. coln FROM table

-- PySpark
dataframe = spark.table("table").select("col1", "col2", ... "coln")
```

## 3. Predicate pushdown

This aims at pushing down the filtering to the "bare metal" — i.e., a data source engine. That is to increase the performance of queries since the filtering is performed at a very low level rather than dealing with the entire data set after it has been loaded to Spark's memory.

To leverage the predicate pushdown, all you need to do is add filters when reading data from source tables. Predicate pushdown is data source engine depe It works for data sources like Parquet, Delta, Cassandra, JDBC, etc., but it will not work for data sources like text, JSON, XML, etc.

SQL (AUTO-DETECTED)

```
-- SQL
SELECT col1, col2 .. coln FROM table WHERE col1 = <value>

-- PySpark
dataframe = spark.table("table").select("col1", "col2", ... "coln").filter(col("col1") = <value>)
```

In cases where you are performing join operations, apply filters before joins. As a rule of thumb, apply filter(s) right after the table read statement.

## 4. Partition pruning

The partition elimination technique allows optimizing performance when reading folders from the corresponding file system so that the desired files only in th specified partition can be read. It will address shifting the filtering of data as close to the source as possible to prevent keeping unnecessary data in memory aim of reducing disk I/O.

To leverage partition pruning, all you have to do is provide a filter on the column(s) being used as table partition(s).

SQL (AUTO-DETECTED)

```
-- SQL
SELECT * FROM table WHERE partition_col = <value>

-- PySpark
dataframe = spark.table("table").filter(col("partition_col") = <value>)
```

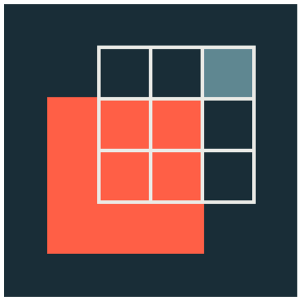
In cases where you are performing join operations, apply partition filter(s) before joins. As a rule of thumb, apply partition filter(s) right after the table read sta

## 5. Dynamic partition pruning (DPP)

In Apache Spark 3.0+, a new optimization called Dynamic Partition Pruning (DPP) is implemented. DPP occurs when the optimizer is unable to identify at pa the partitions it has to eliminate. In particular, we consider a star schema that consists of one or multiple fact tables referencing any number of dimension tab such join operations, we can prune the partitions the join reads from a fact table by identifying those partitions that result from filtering the dimension tables. leverage this feature, no configuration is required. It's enabled by default on Spark 3.0+.

## 6. Dynamic file pruning (DFP)

Dynamic File Pruning (DFP) is available on Databricks Runtime and is by default enabled on all recent runtimes. As the name suggests, it works in a similar as DPP, but it performs dynamic pruning at the file level instead of the partition level to further speed up your queries. To leverage this feature, no configurati required. DFP is automatically enabled in Databricks Runtime 6.1 and higher.



# Data Caching — Leverage Caching to Speed Up the Workload

Delta cache and Spark cache are the two different types of caching that you can leverage to make your workloads faster.

## 1. Delta cache

The Delta cache accelerates data reads by creating copies of remote files in nodes' local storage (SSD drives) using a fast intermediate data format.

- Use Accelerated instances with Delta cache enabled by default (such as *`Standard_E16ds_v4`* in the memory-optimized category and *`Standard_L4s`* in storage-optimized category in Azure cloud)
- Delta cache can be enabled on the workers of other instance families with SSD drives (such as *`Fsv2-series`* in the compute-optimized category in Azu cloud). To explicitly enable the Delta caching, set the following configuration:

PYTHON (AUTO-DETECTED)

```
set spark.databricks.io.cache.enabled = true
```

## 2. Spark cache

Using `cache()` and `persist()` methods, Spark provides an optimization mechanism to cache the intermediate computation of a Spark DataFrame so they c reused in subsequent actions. Similarly, you can also cache a table using the CACHE TABLE command. There are different cache modes that allow you to c where to store the cached data (in the memory, in the disk, in the memory and the disk, with or without serialization, etc.).

- Spark caching is only useful when more than one Spark action (for instance, count, saveAsTable, write, etc.) is being executed o same DataFrame
- Databricks recommends using Delta caching instead of Spark caching, as Delta caching provides better performance outcomes. data stored in the disk cache can be read and operated on faster than the data in the Spark cache. This is because the disk cach efficient decompression algorithms and outputs data in the optimal format for further processing using whole-stage code generati
- Any compute-heavy workload (with a lot of wide transformations like joins and aggregations) would benefit from Delta caching, es when reading from the same tables over and over. Always use Delta cache-enabled instances for those workloads.

# Intermediate Results — When to Persist Them

In large pipelines spanning multiple SQL queries, we often tend to create one or many intermediate working (or temp or staging) Delta tables. This allows us break a big query into small ones for more readability and maintainability. This strategy, however, has a negative impact on the execution duration of our job

- We write the staging tables to the cloud storage, which takes time
- Then we read back the staging tables from cloud storage in subsequent steps, which also takes additional time

Therefore, the better approach would be to create temp views instead of materialized Delta tables since temp views are lazily evaluated and are actually not materialized.

SQL (AUTO-DETECTED)

```
CREATE OR REPLACE TEMP VIEW <view-name> AS SELECT col1, col2,... FROM <table-name>
```

- Always turn an intermediate table into a temporary view if it is used only once in the same Spark job
- If an intermediate table is being used more than once in the same Spark job, you should leave it as a Delta table rather than turni a temporary view since using a temporary view can cause Spark to execute the related query in part more than once. It is strongl advised to use Delta cache-enabled worker instances in this situation so that the temporary table can be automatically cached in SSD drives to speed up the scanning.

# Delta Merge — Let's Speed It Up

You can upsert data from a source table, view or DataFrame into a target Delta table by using the MERGE operation. Delta Lake supports inserts, updates a deletes in the MERGE command.

Instead of overwriting and inserting the entire Delta table on a daily basis, it is recommended that we use an incremental load strategy wherever possible. To achieve the incremental load, Delta MERGE is very essential. Delta merge can also be used to create SCD Type 2 tables and change data capture (CDC) u cases. Here are a couple of examples of how you can use Delta merge: SCD Type 2 using Merge, Change data capture using Merge.

## Merge internals

Behind-the-scenes merge operation is performed in two steps:

1. Inner-join between source and target using the condition in the ON clause to return a list of files in target that contain matching rows to the driver.
2. This step comprises two possibilities — depending on the conditions, one of the following will be executed:
   a. If there were no matching rows in the target in step 1, an append-only write is performed to append the source to target
   b. Else a full outer join is performed to consolidate the changes between the source being merged and the matching files produced in step 1.

## Merge challenges

There are certain performance issues that can be encountered because of the following reasons:

- If we do not have a specific enough condition to match in the ON clause of the merge operation, we will end up rewriting a lot of data. This can slow dow merge.
- If we rewrite a large amount of data after each merge, the Z-order sorting will be messed up, and we may need to execute Z-order after each merge to r the sorting

## Merge optimizations

We can use the following optimization techniques to resolve the above-mentioned challenges:

### Target table data layout

If the target table contains large files (for example, 500MB-1GB), many of those files will be returned to the drive during step 1 of the merge operation, as the the file, the greater the chance of finding at least one matching row. And because of this, a lot of data will be rewritten. Therefore, for the merge-heavy tables recommended to have smaller file sizes from 16MB to 64MB, varying on a case-by-case basis and workload. Refer to the section on file size tuning for more

### Partition pruning

Provide the partition filters in the ON clause of the merge operation to discard the irrelevant partitions. Refer to this example for more details.

### File pruning

Provide Z-order columns (if any) as filters in the ON clause of the merge operation to discard the irrelevant files. You can add multiple conditions using the AN operator in the ON clause.

### Broadcast join

Since Delta merge performs joins behind the scene, you can speed it up by explicitly broadcasting the source DataFrame to be merged in the target Delta ta the source DataFrame is small enough (<= 200MB). Refer to the broadcast join section for more information.

### Low shuffle merge

- This is a new MERGE algorithm that aims to maintain the existing data organization (including Z-order clustering) for unmodified data, while simultaneou improving performance
- With this "low shuffle" MERGE, only updated rows will be reorganized by the operation, while unchanged rows remain in the same order and file groupin were in before the operation

Low shuffle merge is enabled by default in Databricks Runtime 10.4 and above. In earlier supported Databricks Runtime versions, it can be enabled by settir configuration `spark.databricks.delta.merge.enableLowShuffle` to true.

# Data Purging — What to Do With Stale Data

Delta comes with a VACUUM feature to purge unused older data files. It removes all files from the table directory that are not managed by Delta, as well as c files that are no longer in the latest state of the transaction log for the table and are older than a retention threshold. The default threshold is 7 days.

```
VACUUM table_name
```

You can change the default retention threshold if needed as follows:

```
-- table properties
deltaTable.deletedFileRetentionDuration  = "interval 15 days"
```

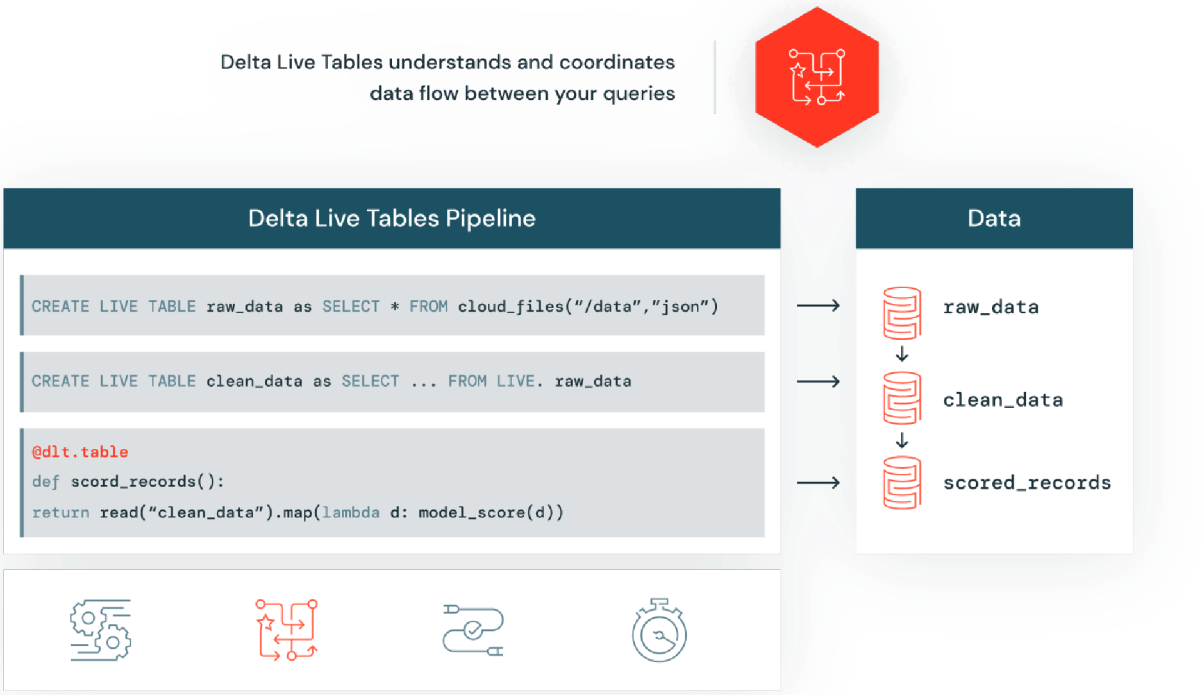
VACUUM will skip all directories that begin with an underscore (_), which includes the *_delta_log*. Log files are deleted automatically and asynchronously aft checkpoint operations. The default retention period of log files is 30 days, configurable through the `delta.logRetentionDuration`.

- It is recommended that you set a retention interval to be at least 7 days because old snapshots and uncommitted files can still b by concurrent readers or writers to the table. If VACUUM cleans up active files, concurrent readers can fail, or, worse, tables car corrupted when VACUUM deletes files that have not yet been committed.
- Set `deltaTable.deletedFileRetentionDuration` and `delta.logRetentionDuration` to the same value to have the same r for data and transaction history
- Run the VACUUM command on a daily/weekly basis depending on the frequency of transactions applied on the Delta tables
- Never run this command as part of your job but run it as a separate job on a dedicated job cluster, usually clubbed with the OPT command
- VACUUM is not a very intensive operation (the main task is file listing, which is done in parallel on the workers; the actual file de handled by the driver), so a small autoscaling cluster of 1 to 4 workers is sufficient

# Delta Live Tables (DLT)

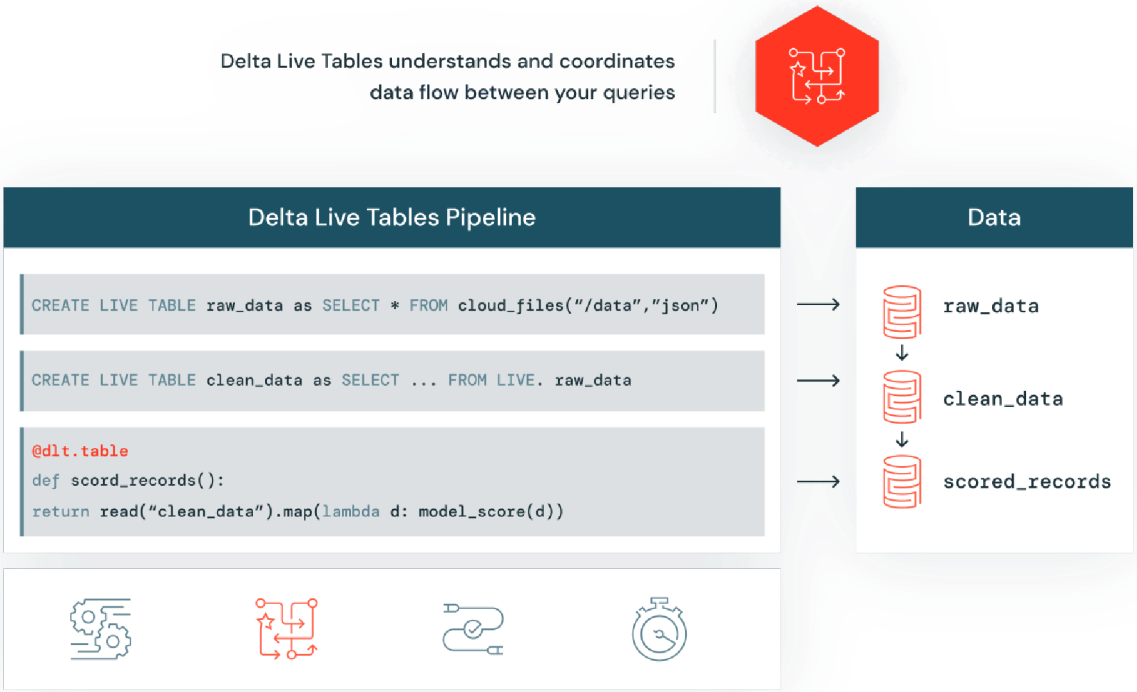
Delta Live Tables (DLT) makes it easy to build and manage reliable batch and streaming data pipelines that deliver high-quality data on the Databricks Lakeh Platform.

DLT helps data engineering teams simplify ETL development and management with declarative pipeline development, automatic data testing, and deep visit monitoring and recovery.



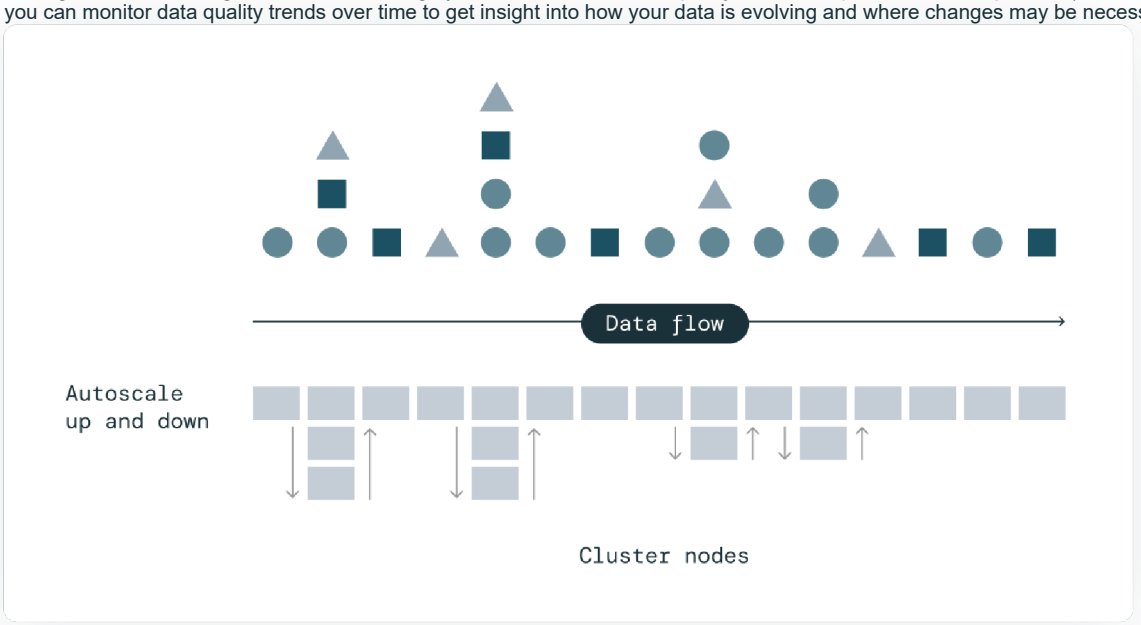

## Easily build and maintain data pipelines

With Delta Live Tables, easily define end-to-end data pipelines in SQL or Python. Simply specify the data source, the transformation logic and the destinatior of the data — instead of manually stitching together siloed data processing jobs. Automatically maintain all data dependencies across the pipeline and reuse pipelines with environment-independent data management. Run in batch or streaming mode and specify incremental or complete computation for each table



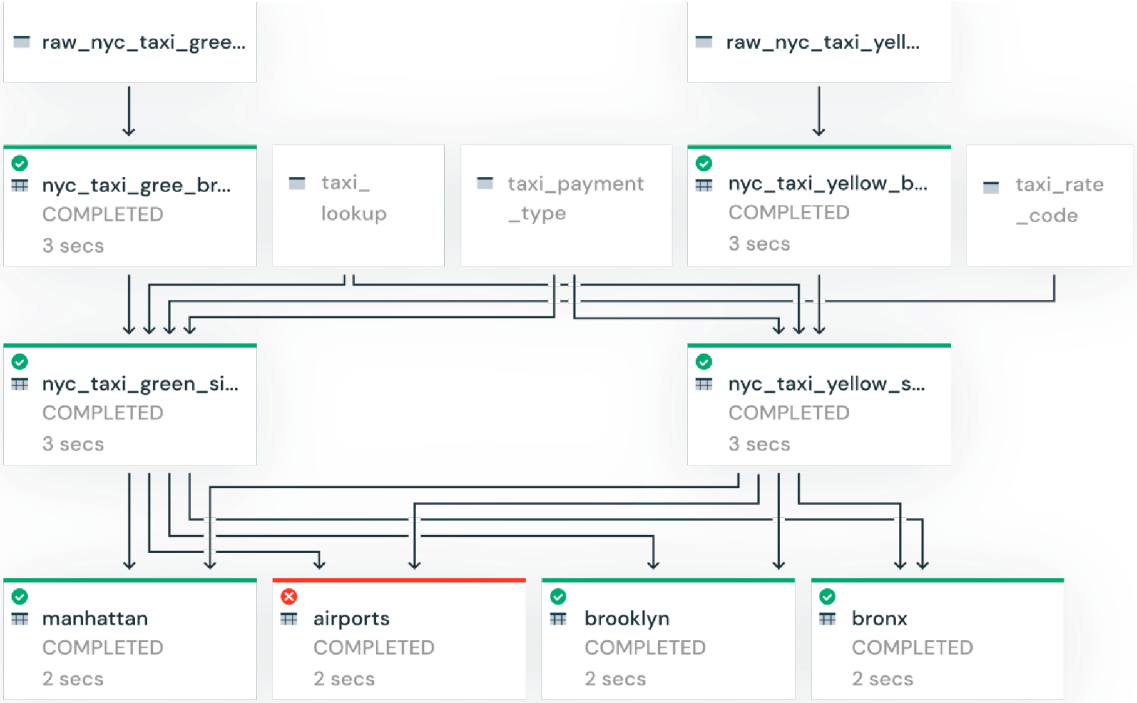

## Automatic data quality testing

Delta Live Tables helps to ensure accurate and useful BI, data science and machine learning with high-quality data for downstream users. Prevent bad data flowing into tables through validation and integrity checks and avoid data quality errors with predefined error policies (fail, drop, alert or quarantine data). In a you can monitor data quality trends over time to get insight into how your data is evolving and where changes may be necessary.




## Cost-effective streaming through efficient compute autoscaling

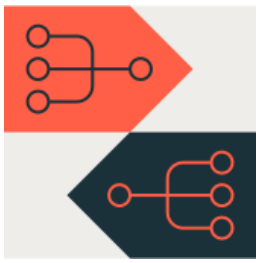
Delta Live Tables Enhanced Autoscaling is designed to handle streaming workloads that are spiky and unpredictable. It optimizes cluster utilization by only s up to the necessary number of nodes while maintaining end-to-end SLAs and gracefully shuts down nodes when utilization is low to avoid unnecessary sper




## Deep visibility for pipeline monitoring and observability

Gain deep visibility into pipeline operations with tools to visually track operational stats and data lineage. Reduce downtime with automatic error handling an replay. Speed up maintenance with single-click deployment and upgrades.

- DLT is a managed ETL framework that frees users to concentrate on writing business-related code while DLT handles operational including clusters, autoscaling, retries on failures, and other configuration tweaking in the background. Therefore, it is advised to u for data engineering workloads as much as possible so that users are spared from worrying about Spark tuning, cluster configurat tuning (see the following part), Delta maintenance tasks, etc.



# Databricks Cluster Configuration and Tuning

## All-purpose clusters vs. automated clusters

- All-purpose clusters should only be used for ad hoc query execution and interactive notebook execution during the development and/or testing phases
- Never use an all-purpose cluster for an automated job; instead, use ephemeral (also called automated) job clusters for jobs
- This is the single biggest cost optimization impact: you could reduce the DBU cost by opting for an automated cluster instead of an all-purpose cluster
- You can also leverage SQL warehouses to run your SQL queries. A SQL warehouse is a compute resource that lets you run SQL commands on data ob within Databricks SQL. SQL warehouses are also available in serverless flavor, offering you access to an instant compute.

## Autoscaling

Databricks provides a unique feature of cluster autoscaling. Here are some guidelines on when and how to leverage autoscaling:

- Always use autoscaling when running ad hoc queries, interactive notebook execution, and developing/testing pipelines using interactive clusters with mi workers set to 1. The combination of both can bring good cost savings.
- For an autoscaling job cluster in production, don't set the minimum instances to 1 if the job certainly needs more resources than 1 worker. Instead, set it bigger value based on the minimum compute requirements. It will save you some scale-up time.

- You don't necessarily need to employ autoscaling if you've fine-tuned the Spark shuffle partitions to use all of the worker cores for a given job and that jo its own job cluster. This is the best option because it allows complete cost control. However, you can still use autoscaling if you wish to have a computat resource buffer in case of unexpected data surges. For this choose the minimum number of workers based on the projected daily data load and required and simply keep a couple of additional workers (say, 2 to 4) as a buffer to be added by autoscaling if and when needed.
- Workflows in Databricks allow you to share a job cluster with many tasks (jobs) that are all part of the same pipeline. If many jobs are executing in parall shared job cluster, autoscaling for that job cluster should be enabled to allow it to scale up and supply resources to all of the parallel jobs.
- Autoscaling should not be used for Spark Structured Streaming workloads because once the cluster scales up, it is tough to determine when to scale it d There were some advances made in this aspect in Delta Live Tables, and autoscaling works pretty well for steaming workloads in Delta Live Tables than the feature called Enhanced Autoscaling.

## Instance types based on workloads

General guidelines to choose instance family based on the workload type:

| VM Category | Workload Type |
|---|---|
| Memory Optimized | • For ML workloads<br>• Where a lot of shuffle and spills are involved<br>• When Spark caching is needed |
| Compute Optimized | • Structured Streaming jobs<br>• ELT with full scan and no data reuse<br>• To run OPTIMIZE and Z-order Delta commands |
| Storage Optimized | • To leverage Delta caching<br>• ML and DL workloads with data caching<br>• For ad hoc and interactive data analysis |
| GPU Optimized | • ML and DL workloads with an exceptionally high memory requirement |
| General Purpose | • Used in absence of specific requirements<br>• To run VACUUM Delta command |

## Number of workers

Choosing the right number of workers requires some trials and iterations to figure out the compute and memory needs of a Spark job. Here are some guideli help you start:

- Never choose a single worker for a production job, as it will be the single point for failure
- Start with 2-4 workers for small workloads (for example, a job with no wide transformations like joins and aggregations)
- Start with 8-10 workers for medium to big workloads that involve wide transformations like joins and aggregations, then scale up if necessary
- Fine-tune the shuffle partitions when applicable to fully engage all cluster cores
- The approximate execution time can be determined after a few first executions. If this violates the SLA, you should add more workers.
- Remember that if shuffle partitions are fine-tuned and data skew and spill issues are addressed, adding additional workers will not result in a higher cost situation, adding more workers will proportionally reduce the total execution time, resulting in a cost that is more or less the same.

## Spot instances

- Use spot instances to use spare VM instances for a below-market rate
- Great for interactive ad hoc/shared clusters
- Can use them for production jobs without SLAs
- Not recommended for production jobs with tight SLAs

- Never use spot instances for the driver

## Auto-termination

- Terminates a cluster after a specified inactivity period in minutes for cost savings
- Enable auto-termination for all-purpose clusters to prevent idle clusters from running overnight or on weekends
- Clusters are configured to auto-terminate in 120 minutes by default, but you can change this to a much lower value, such as 10-15 minutes, to further sa

## Cluster usage

We need to make sure we're getting the most out of our cluster. A job may run many iterations on a cluster to complete depending on the number of shuffle partitions, so the rule of thumb is to ensure that all worker cores are actively occupied and not idle in any of the iterations. The only way to be absolutely cert always set the number of shuffle partitions as a multiplication of the total number of worker cores.

PYTHON (AUTO-DETECTED)

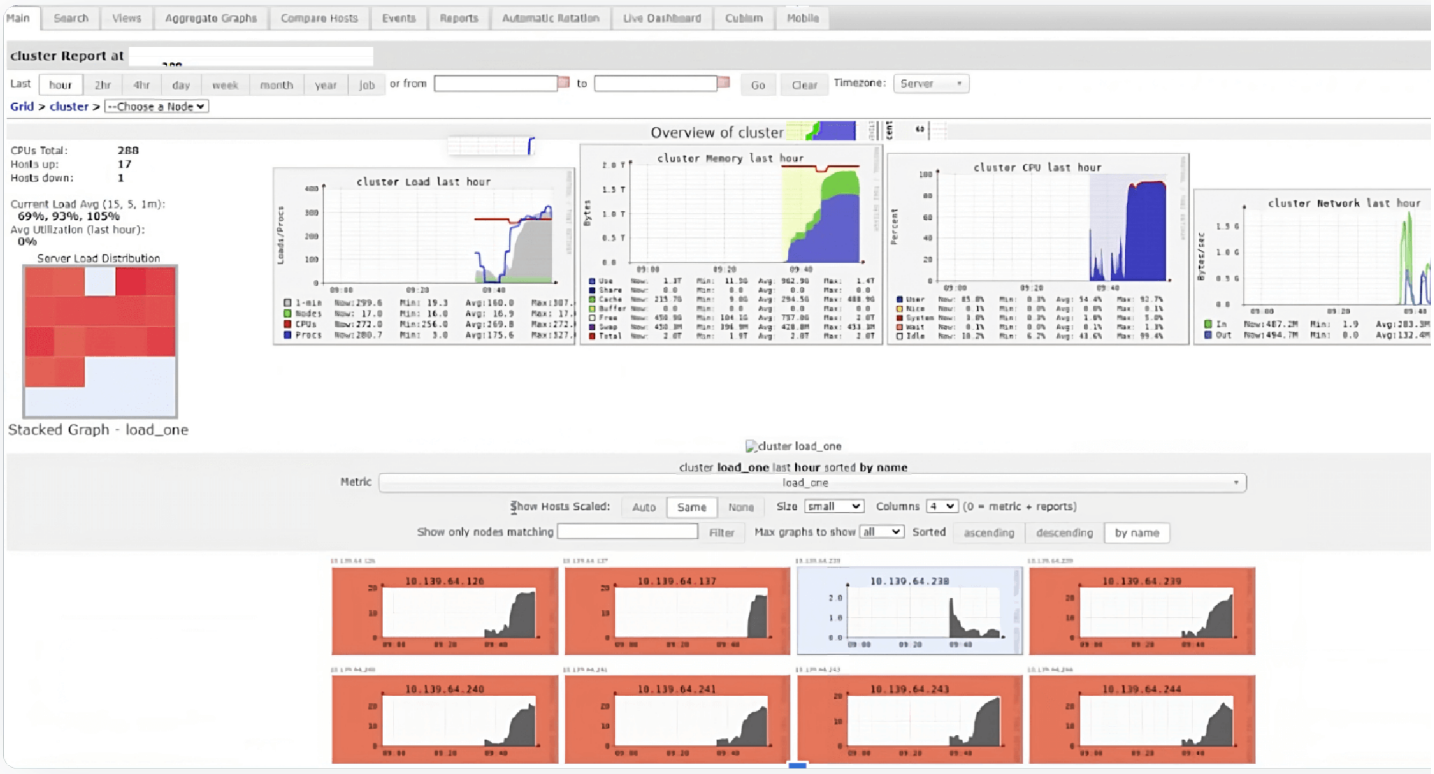
```
-- in SQL
set spark.sql.shuffle.partitions = M*<number of total cores in cluster>

-- in PySpark
spark.conf.set("spark.sql.shuffle.partitions", M*sc.defaultParallelism)

-- M is a multiplier here, pls refer manual shuffle partition tuning section above for more details

-- In the absence of manual shuffle partition tuning set M to 2 or 3 as a rule of thumb
```

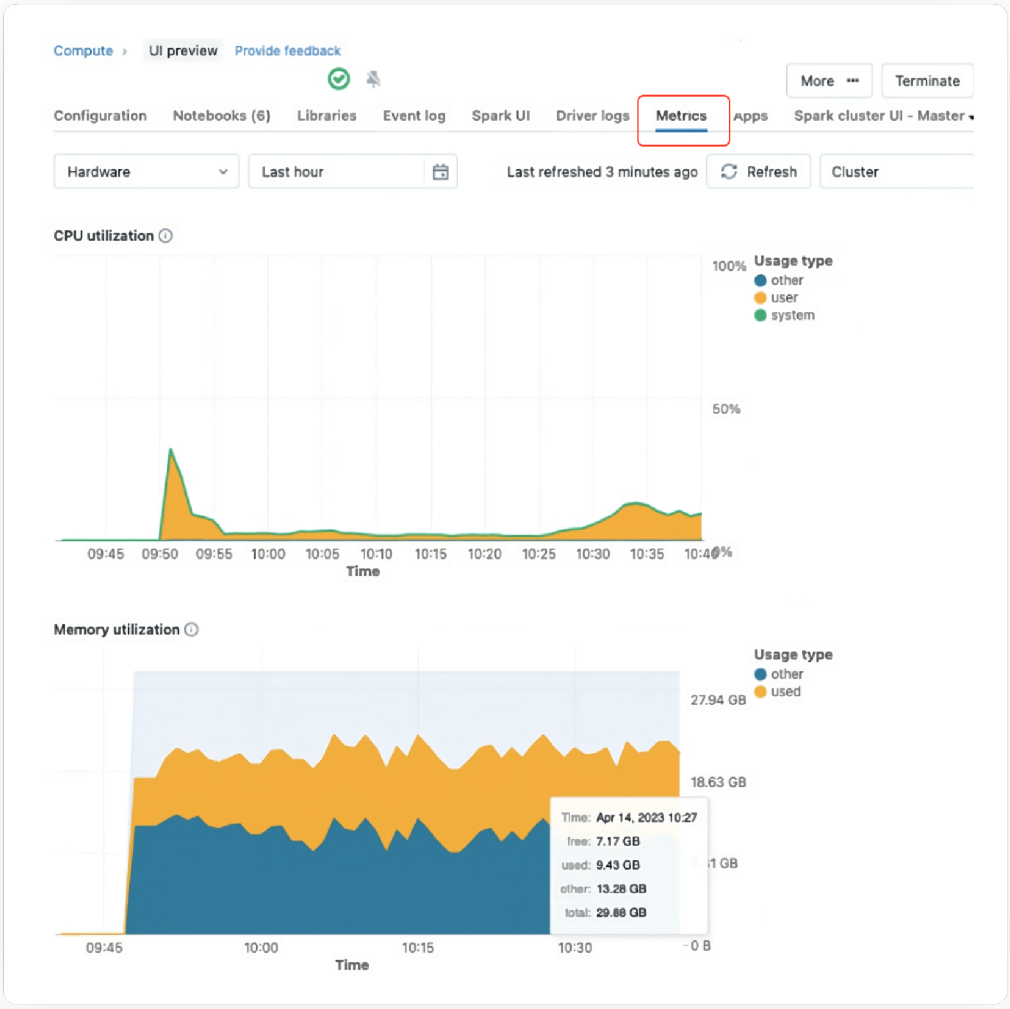
Ganglia UI is your friend to make sure that all the cores are fully engaged. Ganglia UI can be accessed right from the cluster UI pinned under the metrics tab




Things to pay attention to in the Ganglia UI:

- The average cluster load should always be greater than 80%
- During query execution, all squares in the cluster load distribution graph (on the left in the UI) should be red (with the exception of the driver node), indic that all worker cores are fully engaged
- The use of cluster memory should be maximized (at least 60%-70%, or even more)
- Ganglia metrics are only available for Databricks Runtime 12.2 and below

Ganglia has been replaced in Databricks Runtime 13 and above by a new cluster metrics UI that is more secure and powerful, and provides a simple user experience with a clean design. Therefore, in DBR 13 and above, you can leverage this new UI to check out the cluster utilization.




## Cluster policy

A cluster policy limits the ability to configure clusters based on a set of rules. Effective use of cluster policies allows administrators to:

- Limit users to create clusters with prescribed settings
- Control cost by limiting per-cluster maximum cost (by setting limits on attributes whose values contribute to hourly price)
- Simplify the user interface and enable more users to create their own clusters (by fixing and hiding some values)

Refer to the cluster policies best practice guide to plan and ensure a successful cluster governance rollout.

## Instance pools

Databricks pools reduce cluster start and autoscaling times by maintaining a set of idle, ready-to-use instances. When a cluster is attached to a pool, cluster are created using the pool's idle instances. When instances are idle in the pool, they only incur Azure VM costs and no DBU costs.

Pools are recommended for workloads with tight SLAs, where fast access to additional compute resources is a requirement (i.e., fast autoscaling) to improve processing time while minimizing cost.

## Photon

Photon is the next-generation engine on the Databricks Lakehouse Platform that provides extremely fast query performance at a low cost. Photon is compat Apache Spark APIs, works out of the box with existing Spark code, and provides significant performance benefits over the standard Databricks Runtime. Fol are some of the advantages of Photon:

- Accelerated queries that process a significant amount of data (> 100GB) and include aggregations and joins
- Faster performance when data is accessed repeatedly from the Delta cache
- More robust scan/read performance on tables with many columns and many small files
- Faster Delta writing using UPDATE, DELETE, MERGE INTO, INSERT, and CREATE TABLE AS SELECT
- Join improvements

It is recommended to enable Photon (after evaluation and performance testing) for workloads with the following characteristics:

- ETL pipelines consisting of Delta MERGE operations

- Writing large volumes of data to cloud storage (Delta/Parquet)
- Scans of large data sets, joins, aggregations and decimal computations
- Auto Loader to incrementally and efficiently process new data arriving in storage
- Interactive/ad hoc queries using SQL

## Others

- Use the latest LTS version of Databricks Runtime (DBR), as the latest Databricks Runtime is almost always faster than the one before it
- Restart all-purpose clusters periodically, at least once per week (or even daily on busy clusters), as some older DBRs might have stuck/zombie Spark jo
- Configure a large driver node (4-8 cores and 16-32GB is mostly enough) only if:
  - Large data sets are being returned/collected (`spark.driver.maxResultSize` is typically increased also)
  - Large broadcast joins are being performed
  - Many (50+) streams or concurrent jobs/notebooks on the same cluster:
    - Delta Live Tables might be a better fit for such workloads, as DLT determines the complete DAG of the pipeline and then goes about running it most optimized way possible
- For workloads using single-node libraries (e.g., Pandas), it is recommended to use single-node data science clusters as such libraries do not utilize the resources of a cluster in a distributed manner
- Cluster tags can be associated with Databricks clusters to attribute costs to specific teams/departments as they are automatically propagated to underly cloud resources (VMs, storage and network, etc.)
- For such shuffle-heavy workloads, it is recommended to use fewer larger node sizes, which will help with reducing network I/O when transferring data be nodes
- Choosing workers with a large amount of RAM (>128GB) can help jobs perform more efficiently but can also lead to long pauses during **garbage collec** which negatively impacts the performance and in some cases can cause job failure. Therefore, it's not recommended to choose workers with more than RAM. Apart from worker memory, there are some other ways to avoid long garbage collection (GC) pauses:
  - Don't call `collect()` functions to collect data on the driver
    - Applies to the toPandas() function as well
  - Prefer Delta cache over Spark cache
  - If none of the above solutions help, use G1GC garbage collector instead by setting `spark.executor.defaultJavaOptions` and `spark.executor.extraJavaOptions` to `-XX:+UseG1GC` value in Spark config section under A cluster options
- Please refer to the Cluster sizing examples in Databricks documentation for specific cluster sizing examples

# Conclusion

We have covered many Spark, Delta and Databricks optimization strategies and tactics in this book. The purpose of this book is to make readers aware of di challenges that they could encounter when developing big data workloads to run on distributed compute, as well as the methods for identifying these difficult solutions for them. A guide like this may truly assist data engineers and data architects in taking control of the cost of engineering and SLAs and making the use of cloud resources in the present economic climate, where every enterprise is attempting to cut costs and do more with less.