TR Raveendra

Databricks Performance Tuning Steps and Process

Source : Databricks.com

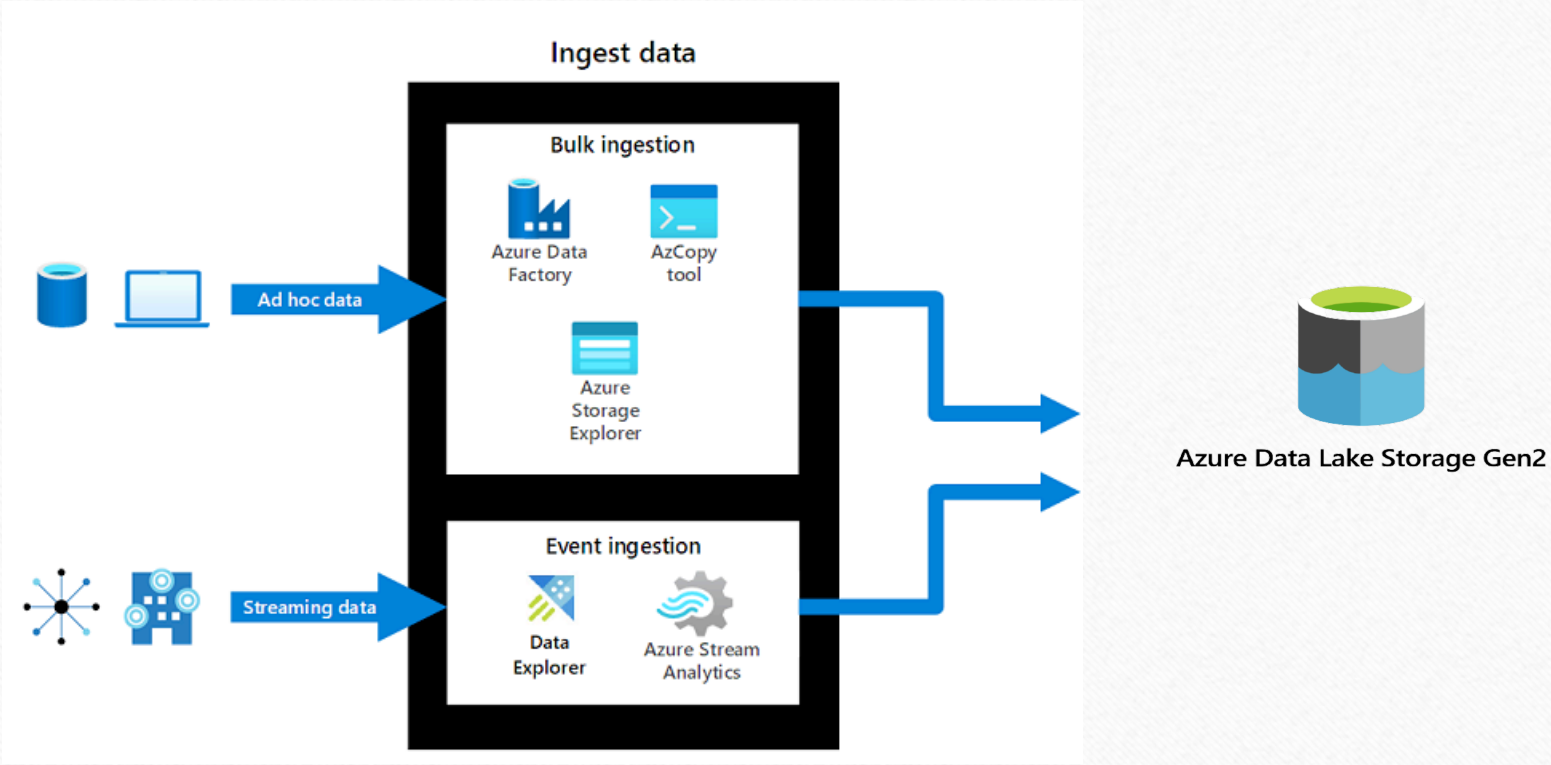
We can divide performance tuning into three categories.

1. **Ingestion**
2. **Transformation**
3. **Extraction for Analytics**

i) Extracting data from different sources and ingesting into data lake.
ii) Transformation layer. Using databricks + pyspark notebooks loading into different stages like Bronze, Silver and Gold
iii) Final tables are using for Analytics. (Delta tables performance for Extraction)

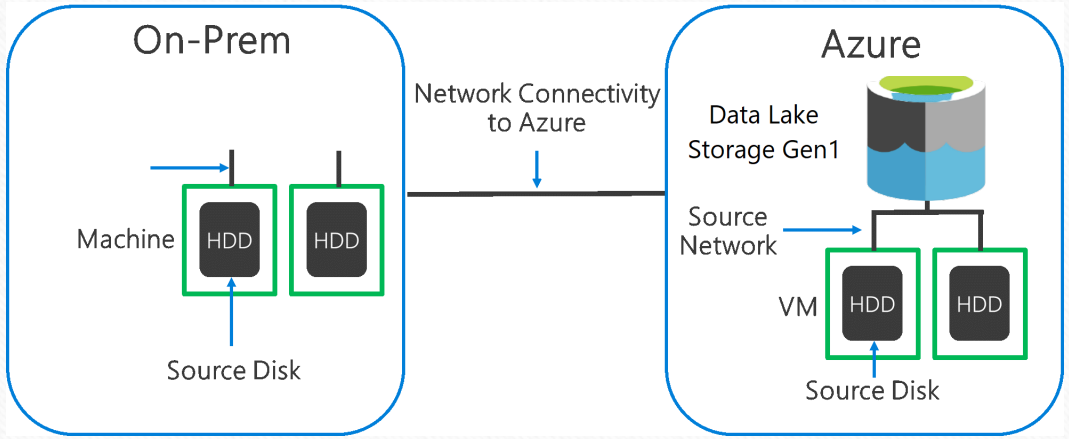
Data ingestion

When ingesting data from a source system to Data Lake Storage Gen2, it's important to consider that the source hardware, source network hardware, and network connectivity to Data Lake Storage Gen2 can be the bottleneck.



Data ingestion

When ingesting data from a source system to Data Lake Storage Gen2, it's important to consider that the source hardware, source network hardware, and network connectivity to Data Lake Storage Gen2 can be the bottleneck.



Batch Data Ingestion To Cloud

1) Historical Data Migration from On-Premises To Cloud
2) Incremental data Migration from On-Premises To Cloud

Historical data will always be a one-time process with massive amounts of data.

ADF is commonly used for migrating data from on premises to cloud.
Follow below approaches for any huge data migration.

1) **use Binary copy in ADF**
2) **Compress larger files and migrate it as-is.**
3) **Split into multiple files if file size is more than 100GB.**
4) **Increase Concurrent Jobs at Self Hosted IR.**

Data will be transferred over the network while migrating from on-premises to cloud storage.
Proceed to the next screen to learn about network transfer statistics.

"Queued" experienced long duration: it means the copy activity waits long in the queue until your Self-hosted IR has resource to execute. Check the IR capacity and usage,
and scale up or out according to your workload.

Data Transfer Over The Network

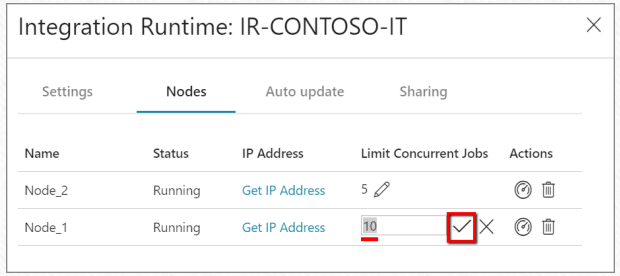
Data size / bandwidth	50 Mbps	100 Mbps	500 Mbps	1 Gbps	5 Gbps	10 Gbps	50 Gbps
1 GB	2.7 min	1.4 min	0.3 min	0.1 min	0.03 min	0.01 min	0.0 min
10 GB	27.3 min	13.7 min	2.7 min	1.3 min	0.3 min	0.1 min	0.03 min
100 GB	4.6 hrs	2.3 hrs	0.5 hrs	0.2 hrs	0.05 hrs	0.02 hrs	0.0 hrs
1 TB	46.6 hrs	23.3 hrs	4.7 hrs	2.3 hrs	0.5 hrs	0.2 hrs	0.05 hrs
10 TB	19.4 days	9.7 days	1.9 days	0.9 days	0.2 days	0.1 days	0.02 days
100 TB	194.2 days	97.1 days	19.4 days	9.7 days	1.9 days	1 day	0.2 days
1 PB	64.7 mo	32.4 mo	6.5 mo	3.2 mo	0.6 mo	0.3 mo	0.06 mo
10 PB	647.3 mo	323.6 mo	64.7 mo	31.6 mo	6.5 mo	3.2 mo	0.6 mo

Verify your network speed using this portal

Check if the Self-hosted IR machine has enough outbound bandwidth to transfer and write the data efficiently.
If your sink data store is in Azure, you can use this tool to check the upload speed.

https://www.azurespeed.com/Azure/UploadLargeFile

Increase Concurrent jobs depends on On-Premises Node Compute Capacity

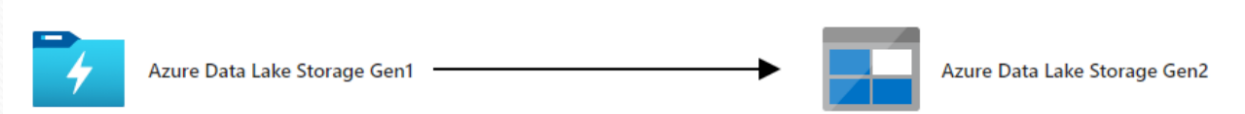


If source is Database:

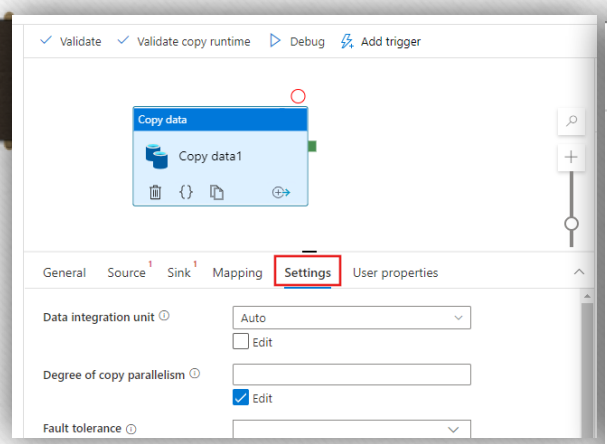
When copying data from Oracle, Netezza, Teradata, SAP HANA, SAP Table, and SAP Open Hub), enable data partition options to copy data in parallel copy.

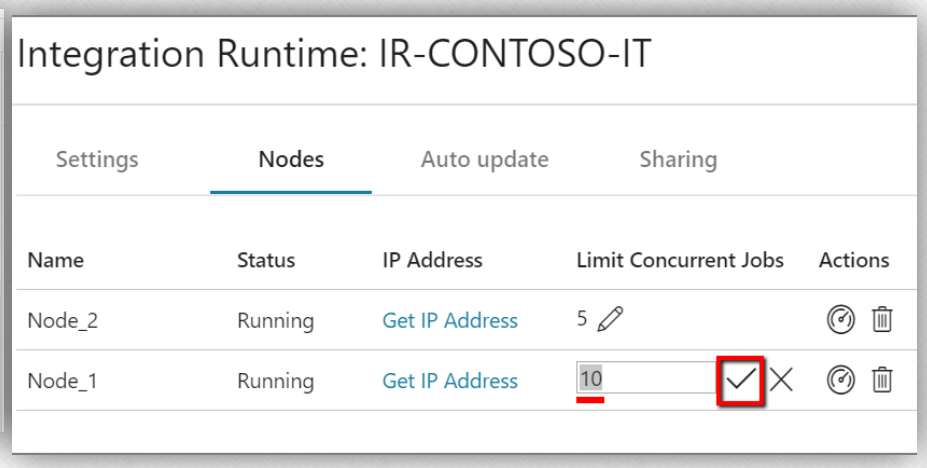
Avoid staged copy while migrating data from Data lake to DB's or Datalake to data lake

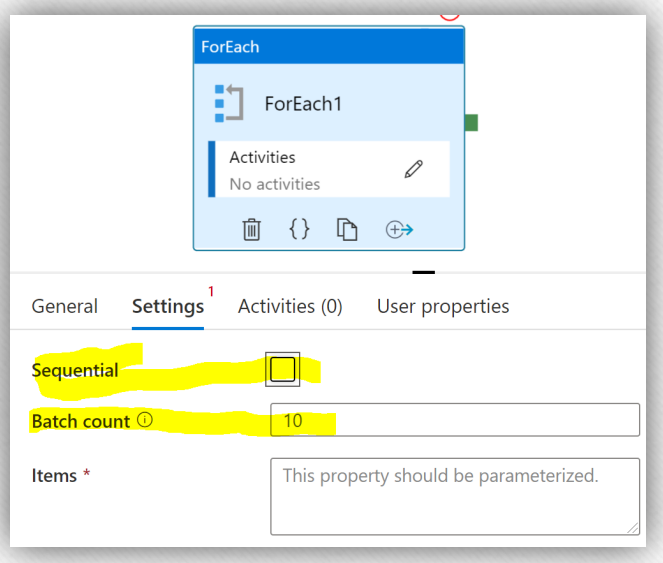
Migrating data from ADLS Gen1 to ADLS Gen2



1. A single copy activity can take advantage of scalable compute resources
2. when using Azure Integration Runtime, you can specify up to **256 data integration units (DIUs)** for each copy activity in a serverless manner
3. when using **self-hosted Integration Runtime**, you can manually scale up the machine or scale out to multiple machines (up to 4 nodes), and a single copy activity will partition its file set across all nodes.
4. A single copy activity reads from and writes to the data store using multiple threads.
5. ADF control flow can start multiple copy activities in parallel, for example using For Each loop.







Transformation Layer

Step 1 : Understand execution plan

Step 2 : Fix and implement below scenarios

1) Understand Metadata (columns and data types)
2) Cache Manager (Cache/Persist)
3) Column Pruning
4) Predicate Push Down
5) Partition Filter
6) Join Strategies
7) Adaptive Query Execution

Step 3 : Delta Table Features

1) Fix Small files issue using optimise
2) Use Portioning with ZOrderBy for more than 1TB Tables
3) Use Cluster By for 10GB to 1TB Tables
4) Vacuum older snapshots
5) Use Auto-Optimized for small tables

Partition Size

Determine Desired Partition Size: Consider the desired partition size based on the available resources, cluster configuration, and the nature of your workload. A common guideline is to aim for partitions that are 128 MB to 1 GB in size, as this range generally balances performance and resource utilization. However, this can vary depending on your specific use case and cluster setup.

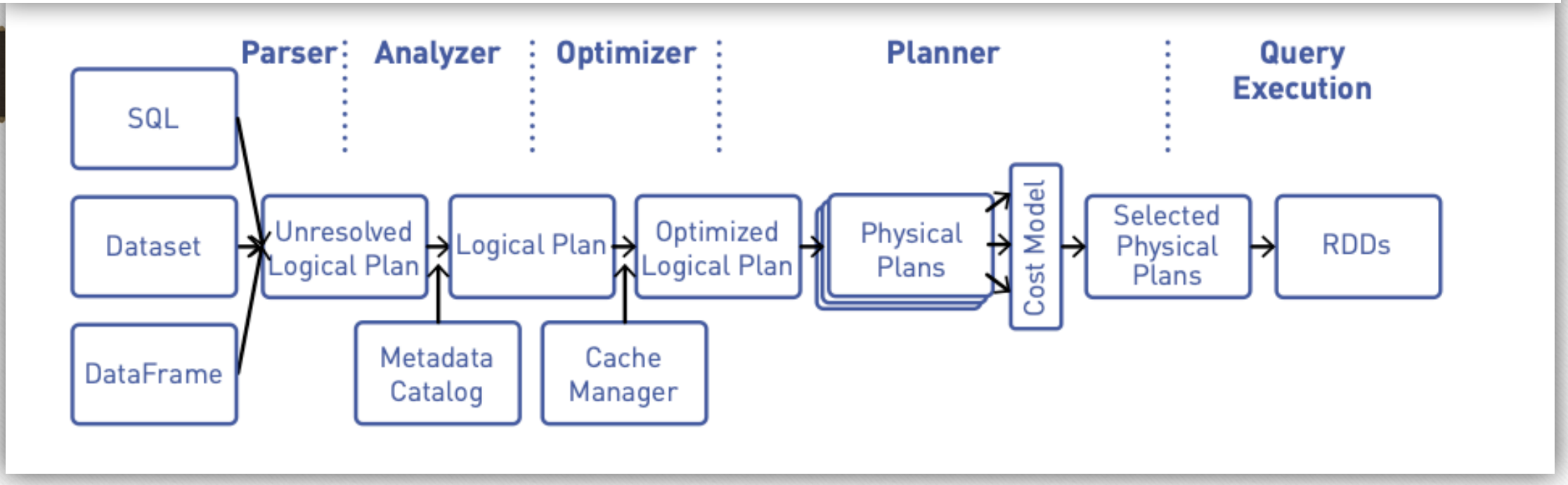
Default partition size parameter value is spark.sql.files.maxPartitionBytes. = 128MB
We can increase or decrease based on incoming file sizes using spark.conf.set("spark.sql.files.maxPartitionBytes","1G")

Table size	Target file size	Approximate number of files in table
10GB	256MB	40
1TB	256MB	4096
2.56TB	256MB	10240
3TB	307MB	12108
5TB	512MB	17339
7TB	716MB	20784
10TB	1GB	24437
20TB	1GB	34437
50TB	1GB	64437
100TB	1GB	114437

Spark Execution Plan

Spark Execution Plan steps

- To illustrate all of these architectural and most relevantly transformations and actions - let's go through a more thorough example, this time using DataFrames and a csv file.
- The DataFrame and SparkSQL work almost exactly as we have described above, we're going to build up a plan for how we're going to access the data and then finally execute that plan with an action. We'll see this process in the diagram below. We go through a process of analyzing the query, building up a plan, comparing them and then finally executing it.



Explain Function SQL & Pyspark

explain(extended=None, mode=None)

- Prints the (logical and physical) plans to the console for debugging purpose.

Parameters

- extended – boolean, default False. If False, prints only the physical plan. When this is a string without specifying the mode, it works as the mode is specified.

mode –

- specifies the expected output format of plans.
- `simple` : Print only a physical plan.
- `extended` : Print both logical and physical plans.
- `codegen` : Print a physical plan and generated codes if they are available.
- `cost` : Print a logical plan and statistics if they are available.
- `formatted` : Split explain output into two sections: a physical plan outline and node details.

```
%sql
EXPLAIN EXTENDED SELECT * FROM sales_table;
```

```
%python
# Create a DataFrame
df = spark.read.csv("data.csv")

# Explain the execution plan of the DataFrame in extended mode
df.explain(extended=True)
```

Spark SQL configurations

Property	Default	Description
spark.sql.inMemoryColumnarStorage.compressed	true	When set to true, Spark SQL automatically selects a compression codec for each column based on statistics of the data.
spark.sql.inMemoryColumnarStorage.batchSize	10000	Controls the size of batches for columnar caching. Larger batch sizes can improve memory utilization and compression, but risk OutOfMemoryErrors (OOMs) when caching data.
spark.sql.files.maxPartitionBytes	134217728 (128 MB)	The maximum number of bytes to pack into a single partition when reading files.
spark.sql.broadcastTimeout	300	Timeout in seconds for the broadcast wait time in broadcast joins.
spark.sql.files.openCostInBytes	10485760 (10 MB)	The estimated cost to open a file, measured by the number of bytes that could be scanned in the same time. This is used when putting multiple files into a partition. It is better to overestimate; that way the partitions with small files will be faster than partitions with bigger files (which is scheduled first).
spark.sql.autoBroadcastJoinThreshold	10485760 (10 MB)	Configures the maximum size in bytes for a table that will be broadcast to all worker nodes when performing a join. You can disable broadcasting by setting this value to -1. Note that currently statistics are supported only for Hive Metastore tables for which the command ANALYZE TABLE COMPUTE STATISTICS noscan has been run
spark.sql.shuffle.partitions	200	Configures the number of partitions to use when shuffling data for joins or aggregations.

1) Adaptive Query Execution

Adaptive Query Execution (AQE) is an optimisation technique in Spark SQL that makes use of the runtime statistics to choose the most efficient query execution plan, which is enabled by default since Apache Spark 3.2.0. Spark SQL can turn on and off AQE by **spark.sql.adaptive.enabled** as an umbrella configuration. As of Spark 3.0, there are three major features in AQE: including coalescing post-shuffle partitions, converting sort-merge join to broadcast join, and skew join optimisation.

1. Coalescing Post Shuffle Partitions
2. Converting sort-merge join to broadcast join
3. Converting sort-merge join to shuffled hash join
4. Optimizing Skew Join


```
%sql
set spark.sql.adaptive.enabled=true
```

Coalescing Post Shuffle Partitions

This feature coalesces the post shuffle partitions based on the map output statistics when both spark.sql.adaptive.enabled and **spark.sql.adaptive.coalescePartitions.enabled** configurations are true.

- ☑ Enable AQE for all of your Spark jobs. This will allow Spark to optimize all of your queries, even if you are not sure which queries will benefit the most from AQE.
- ☑ Monitor the performance of your queries with and without AQE enabled. This will help you to identify which queries are benefiting from AQE and which queries are not.
- ☑ If you find that a particular query is not performing as well with AQE enabled, you can disable AQE for that query by setting the **spark.sql.adaptive.enabled** configuration property to false for that query.

Low Cardinality Vs High Cardinality

Low cardinality columns are columns in a database table that have a small number of distinct values. For example, a column that stores the gender of a customer might have low cardinality, since there are only a few distinct values (e.g., male, female, other).
High cardinality columns are columns in a database table that have a large number of distinct values. For example, a column that stores the email address, customer id of a customer might have high cardinality, since each customer has a unique email address.
Medium cardinality columns are columns in a database table that have a Average number of distinct values. For example, a column that stores the country, city, DOB of a customer might have Medium cardinality, since some customers can have same DOB or City, Country.

Medium Cardinality

High Cardinality

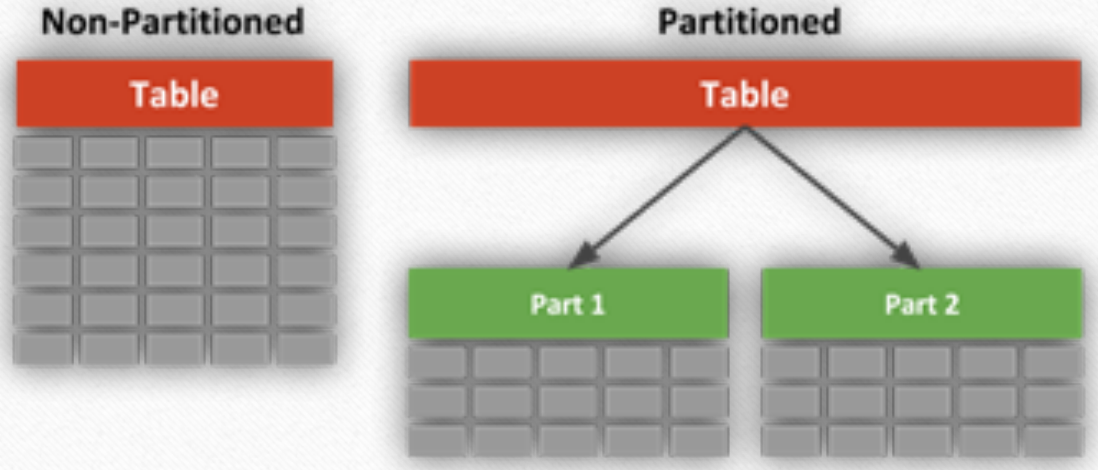
Low Cardinality

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO	GENDER
7369	SMITH	CLERK	7902	17-12-80	800	null	20	MALE
7499	ALLEN	SALESMAN	7698	20-02-81	1600	300	30	MALE
7521	WARD	SALESMAN	7698	22-02-81	1250	500	30	MALE
7566	JONES	MANAGER	7839	02-04-81	2975	null	20	MALE
7654	MARTIN	SALESMAN	7698	28-09-81	1250	1400	30	MALE
7698	SGR	MANAGER	7839	01-05-81	2850	null	30	MALE
7782	RAVI	MANAGER	7839	09-06-81	2450	null	10	MALE
7788	SCOTT	ANALYST	7566	19-04-87	3000	null	20	MALE
7839	KING	PRESIDENT	null	17-11-81	5000	null	10	MALE
7844	TURNER	SALESMAN	7698	08-09-81	1500	0	30	MALE
7876	ADAMS	CLERK	7788	23-05-87	1100	null	20	MALE
7900	JAMES	CLERK	7698	03-12-81	950	null	30	MALE
7902	FORD	ANALYST	7566	03-12-81	3000	null	20	MALE
7934	MILLER	CLERK	7782	23-01-82	1300	null	10	MALE
1234	ANITHA	doctor	7777	null	667	78	80	FEMALE

Partitioning and ZOrder By

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 merge on partition column(s), as it will help Spark to skip a lot of unnecessary data partition (i.e., subfolders) during scan time.

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



Partitioning with Year and Month

Creating Partition using year and month column.

- first we need to add two columns.
- year (yyyy) and month (mm) for creating partitions

Cmd 27

```
1  from pyspark.sql.functions import date_format
2  #creating two YEAR and MONTH new columns based on hiredate date field
3  df_emp_csv = df_emp.withColumn("YEAR",date_format("HIREDATE",'yyyy')).withColumn("MONTH",date_format("HIREDATE",'MM'))
4  df_emp_csv.show()
```

▸ df_emp_csv: pyspark.sql.dataframe.DataFrame = [EMPNO: integer, ENAME: string ... 8 more fields]

```
+-----+------+---------+----+----------+----+----+------+----+-----+
|EMPNO| ENAME|      JOB| MGR|  HIREDATE| SAL|COMM|DEPTNO|YEAR|MONTH|
+-----+------+---------+----+----------+----+----+------+----+-----+
| 7369| SMITH|    CLERK|7902|2080-12-17| 800|null|    20|2080|   12|
| 7499| ALLEN| SALESMAN|7698|2081-02-20|1600| 300|    30|2081|   02|
| 7521|  WARD| SALESMAN|7698|2081-02-22|1250| 500|    30|2081|   02|
| 7566| JONES|  MANAGER|7839|2081-04-02|2975|null|    20|2081|   04|
| 7654|MARTIN| SALESMAN|7698|2081-09-28|1250|1400|    30|2081|   09|
| 7698|   SGR|  MANAGER|7839|2081-05-01|2850|null|    30|2081|   05|
| 7782|  RAVI|  MANAGER|7839|2081-06-09|2450|null|    10|2081|   06|
| 7788| SCOTT|  ANALYST|7566|2087-04-19|3000|null|    20|2087|   04|
| 7839|  KING|PRESIDENT|null|2081-11-17|5000|null|    10|2081|   11|
| 7844|TURNER| SALESMAN|7698|2081-09-08|1500|   0|    30|2081|   09|
| 7876| ADAMS|    CLERK|7788|2087-05-23|1100|null|    20|2087|   05|
| 7900| JAMES|    CLERK|7698|2081-12-03| 950|null|    30|2081|   12|
| 7902|  FORD|  ANALYST|7566|2081-12-03|3000|null|    20|2081|   12|
| 7934|MILLER|    CLERK|7782|2082-01-23|1300|null|    10|2082|   01|
| 1234|SEKHAR|   doctor|7777|      null| 667|  78|    80|null| null|
+-----+------+---------+----+----------+----+----+------+----+-----+
```

```
1  # using PartitionBy with two columns creating partitions
2  df_emp_csv.write.format("delta").partitionBy("YEAR","MONTH").mode("overwrite").saveAsTable("emp_part")
```

Partitioning and ZOrder By

Partitioning divides the data into smaller subsets based on the values of one or more columns. This makes it easier for Delta Lake to find the data that is relevant to a query.

Z-ordering is a technique for co-locating related information in the same set of files. This makes it faster for Delta Lake to read the data that is relevant to a query.

Partitioning and Z-ordering can be used together or independently. Partitioning is typically used for tables that are large and have a lot of data. Z-ordering is typically used for tables that are frequently queried on a small subset of the columns.

Benefits of partitioning and Z-ordering:

1. Improved performance of queries
2. Reduced data scanning
3. Reduced CPU usage
4. Improved scalability

```
1   CREATE TABLE sales_table (
2     id INT,
3     sales_date TIMESTAMP,
4     sales_amount DECIMAL(10,2),
5     country_cd string, -- zorder by for if you are using join or filter
6     YEAR(sales_date) AS year --partitioned column
7   )
8   PARTITIONED BY (year)
```

```
1   optimize sales_table zorder by (country_cd)
```

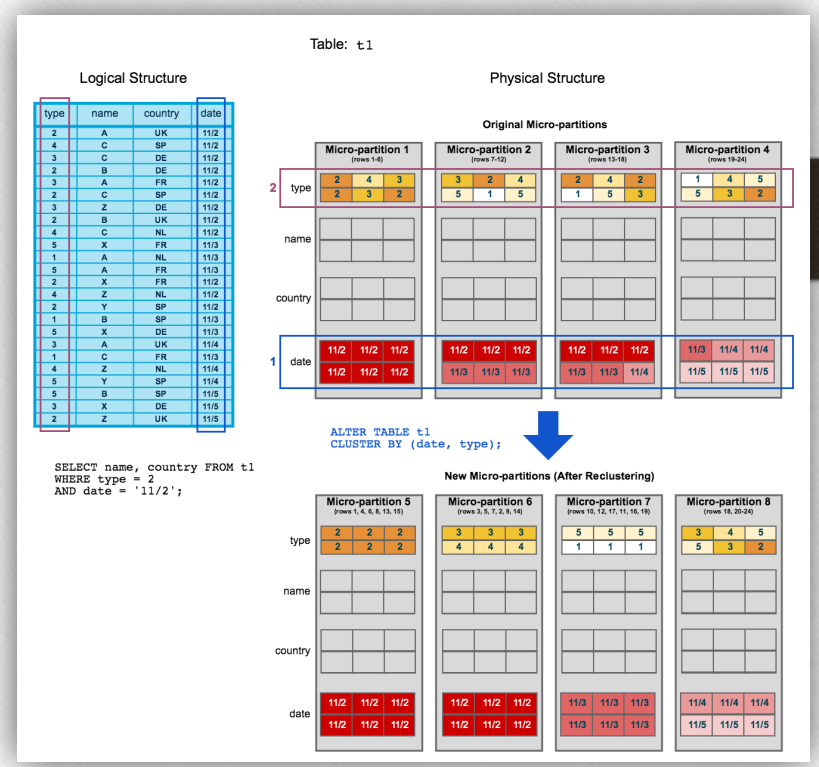
Liquid Clustering

Liquid clustering in Databricks is a new feature that allows you to cluster your Delta Lake tables in a more flexible and efficient way. It is a replacement for traditional table **partitioning and Z-ordering.**

Liquid clustering has a number of advantages over traditional table partitioning and Z-ordering, including:

- ☑ **Flexibility**: You can redefine the clustering keys for your table without having to rewrite the data.
- ☑ **Efficiency**: Liquid clustering incrementally clusters new data as it is written to the table. This means that you don't have to worry about the performance penalty of clustering large datasets.
- ☑ **Performance**: Liquid clustering can improve the performance of queries that filter or aggregate on the clustering keys.

```
-- Create an empty table
CREATE TABLE sales (
  id INT,
  sales_date TIMESTAMP,
  sales_amount DECIMAL(10,2),
  YEAR(sales_date) AS year
)
CLUSTERED BY (year); -- cluster by column
-- Using a CTAS statement
CREATE EXTERNAL TABLE sales CLUSTER BY (year)  --
LOCATION '/mnt/sales/'
AS SELECT * FROM old_sales;

-- Using a LIKE statement to copy configurations
CREATE TABLE sales2 LIKE sales;
```

Once you have created a table with liquid clustering enabled, you can start writing data to it. Liquid clustering will automatically cluster the new data as it is written.

You can also manually cluster the data in your table at any time by running the following SQL statement:

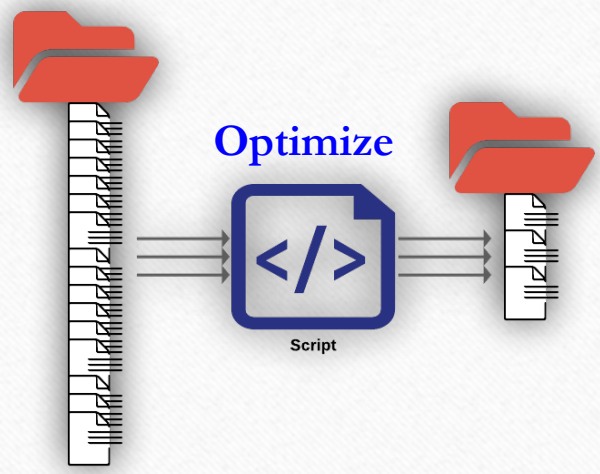
```
optimize table sales
```

What is Small Files Issue?

The small files issue is a common problem in distributed file systems. It occurs when the data is divided into many small files. This can cause performance problems, as Spark needs to read and process each file individually.

There are a few things that can cause the small files issue in Delta Lake, including:

1. **Streaming jobs**: Streaming jobs can generate many small files, as they write data to Delta Lake in micro-batches.
2. **Frequent updates**: If you are frequently updating a Delta table, this can also lead to the small files issue, as Spark will create a new file for each update.
3. **Incorrect partitioning**: If the data in Delta Lake is not partitioned correctly, this can also lead to the small files issue.



How to Solve Small Files Issue?

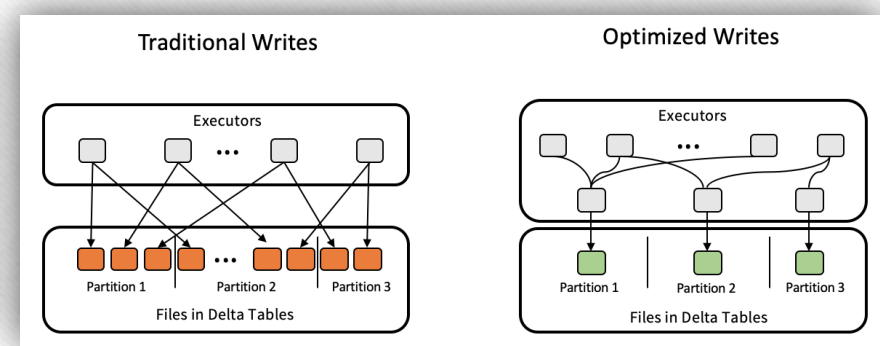
OPTIMIZE compacts the files to get a file size of up to 1GB, which is configurable. This command basically attempts to size the files to the size that you have configured (or 1GB by default if not configured).

- ☑ Run the OPTIMIZE command periodically to compact small files.

```
OPTIMIZE '/path/to/delta/table' -- Optimizes the path-based Delta Lake table
OPTIMIZE delta_table_name;
OPTIMIZE delta.`/path/to/delta/table`;
-- If you have a large amount of data and only want to optimize a subset of it,
--you can specify an optional partition predicate using `WHERE`:
OPTIMIZE delta_table_name WHERE date >= '2017-01-01'
```

- ☑ 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's done inside the same Spark job.
- ☑ Use auto compaction for streaming jobs and jobs that frequently update Delta tables.

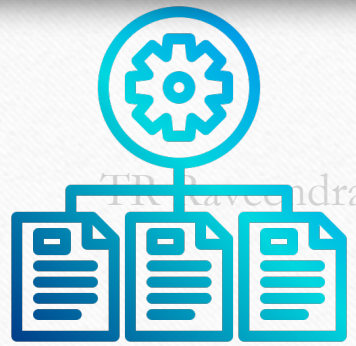
```
1  -- Table properties
2  delta.autoOptimize.optimizeWrite = true
3  delta.autoOptimize.autoCompact = true
4
5  -- In Spark session conf for all new tables
6  set spark.databricks.delta.properties.defaults.autoOptimize.optimizeWrite = true
7  set spark.databricks.delta.properties.defaults.autoOptimize.autoCompact = true
```



File Size Tuning

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. You can set the target file size by using delta.targetFileSize table property and then Auto-optimize and Optimize will binpack to achieve the specified size instead.

```
ALTER TABLE table_name SET TBLPROPERTIES ('delta.targetFileSize' = '100mb');
```



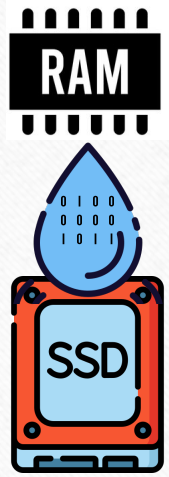
You can also delegate this task to Databricks if you don't want to manually set the target file size by turning on the **delta.tuneFileSizesForRewrites** table property. When this property is set to true, Databricks will automatically tune the file sizes based on workloads. For example, if you do a lot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

```
ALTER TABLE table_name SET TBLPROPERTIES ('delta.tuneFileSizesForRewrite' = 'true');
```

1. It can help to improve the performance of queries by reducing the number of files that need to be scanned.
2. It can help to reduce the cost of storing data in cloud storage by reducing the number of files that need to be stored.
3. It can help to improve the manageability of large tables by making it easier to split and merge tables.

Data Spilling

Why It Happens and How to Get Rid of It



Data spilling in Apache Spark refers to the process of moving data from memory to disk and back again. This occurs when a given partition of data is too large to fit into the memory of the executor, which is the Spark worker that is responsible for processing that partition.

1. Spilling to disk is a costly operation, as it involves data serialization, de-serialization, reading and writing to disk, etc.
2. Spilling needs to be avoided at all costs and in doing so, we must tune the number of shuffle partitions.
3. There are a couple of ways to tune the number of Spark SQL shuffle partitions as discussed below.

There are a few reasons why data spilling might occur:

- ☑ **Spark SQL shuffle partitions is 200 (Default)** : The number of CPU cores used to perform wide transformations such as joins, aggregations and so on which isn't always the best value.
- ☑ **Data skew**: This is when one or more partitions of data are much larger than the others. This can be caused by a variety of factors, such as uneven distribution of data values or inefficient data partitioning.
- ☑ **Memory limitations**: If the executor does not have enough memory to store all of the data in a partition, it will spill some of the data to disk. This can happen even if the data is not skewed, if the partition is simply too large.
- ☑ **Shuffle operations**: Shuffle operations, such as joins and aggregations, can also cause data spilling. This is because shuffles require Spark to repartition the data, which can result in larger partitions.

Fixing Data Spilling Issues

AQE auto-tuning

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

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

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 follows

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

- ☑ The optimal size of data to be processed per task should be 128MB approximately. It works well in most cases. It might not work if there 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 section 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, or better thrice, the number of total worker CPU cores

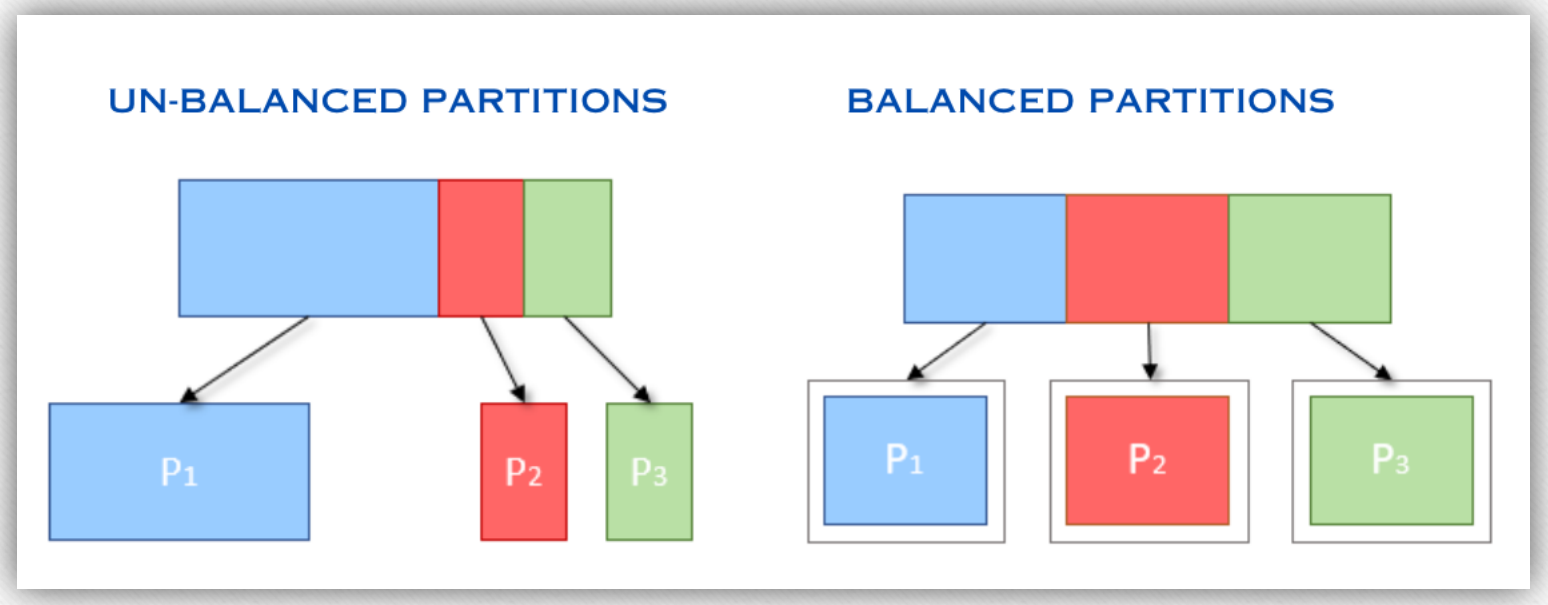
```
-- in SQL
set spark.sql.shuffle.partitions = 2*<number of total worker cores in cluster>
-- in PySpark
spark.conf.set("spark.sql.shuffle.partitions", 2*<number of total worker cores in cluster>)
-- or
spark.conf.set("spark.sql.shuffle.partitions", 2*sc.defaultParallelism)
```

DATA SKEW

Data skew in Spark is when the data is unevenly distributed across the partitions. This can lead to performance problems, as some tasks will take much longer to complete than others.

There are a few things that can cause data skew in Spark, including:

1. **Data joins:** When joining two datasets, the data will be partitioned by the join key. If the join key is not evenly distributed across the data, it can lead to data skew.
2. **Group-by operations**: Group-by operations can also cause data skew, if the group-by key is not evenly distributed across the data.
3. **Data partitioning**: If the data is partitioned by a column that is not evenly distributed across the data, it can lead to data skew.



Salting and other AQE options for fixing data Skew

There are a few ways to fix data skew issues in Spark, including:

1. **Use Adaptive Query Execution (AQE)**: AQE is a feature in Spark that can automatically detect and fix data skew issues.
2. **Remove problematic keys**: If you know which keys are causing the data skew, you can remove them from the join or group-by operation.
3. **Use salting**: Salting is a technique that can be used to evenly distribute the data across the partitions. Salting involves adding a random value to the join key or group-by key.
4. **Repartition the data**: You can repartition the data by a column that is evenly distributed across the data.
5. **Use Broadcast join** : Use broadcast join for broadcast join to avoid shuffling.


```
# Enable Adaptive Query Execution
spark.conf.set("spark.sql.adaptive.enabled", "true")
```

Salting is a technique that can be used to evenly distribute data across partitions. This can help to avoid data skew and improve the performance of Spark jobs.

```
# Read the data
df = spark.read.csv("data.csv")

# Add a salt column to the data
df = df.withColumn("salt", pyspark.sql.functions.rand())

# Repartition the data by the salt column
df = df.repartition(10, "salt")

# Perform the operation
df = df.groupBy("group_by_key").agg({"count": "count"})

# Write the data to a file
df.write.csv("output.csv")
```

How to Solve Data Skew Issues

Repartitioning: Use the repartition() function to redistribute data across partitions. You can manually repartition the data using a different column or use the repartition() function along with a random or semi-random value to shuffle the data evenly across partitions.

```
# Repartition the DataFrame using a random column
repartitioned_df = df.repartition("random_column")
```

Skew Join Optimization: Spark provides a built-in optimization technique called skew join optimization. It handles data skew during joins by redistributing the skewed data across multiple partitions and performing the join in a way that minimizes the impact of skewness.


```
# Enable skew join optimization
spark.conf.set("spark.sql.adaptive.skewJoin.enabled", "true")
```

Broadcast Join: If one side of the join operation is small enough to fit in memory, you can use a broadcast join. This technique broadcasts the smaller DataFrame to all worker nodes, avoiding data shuffling. However, this approach may not be suitable if the skewed key is on the larger DataFrame.

```
from pyspark.sql.functions import broadcast

# Perform a broadcast join
joined_df = df1.join(broadcast(df2), "key_column")
```

Join Strategies

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 driver, 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:

```
%sql
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 option.


```
%sql
-- BROADCAST  or MAPJOIN or BROADCASTJOIN
select /*+ MAPJOIN(e)  */ * from emp as e inner join dept as d on e.deptno=d.deptno where e.deptno=10
```

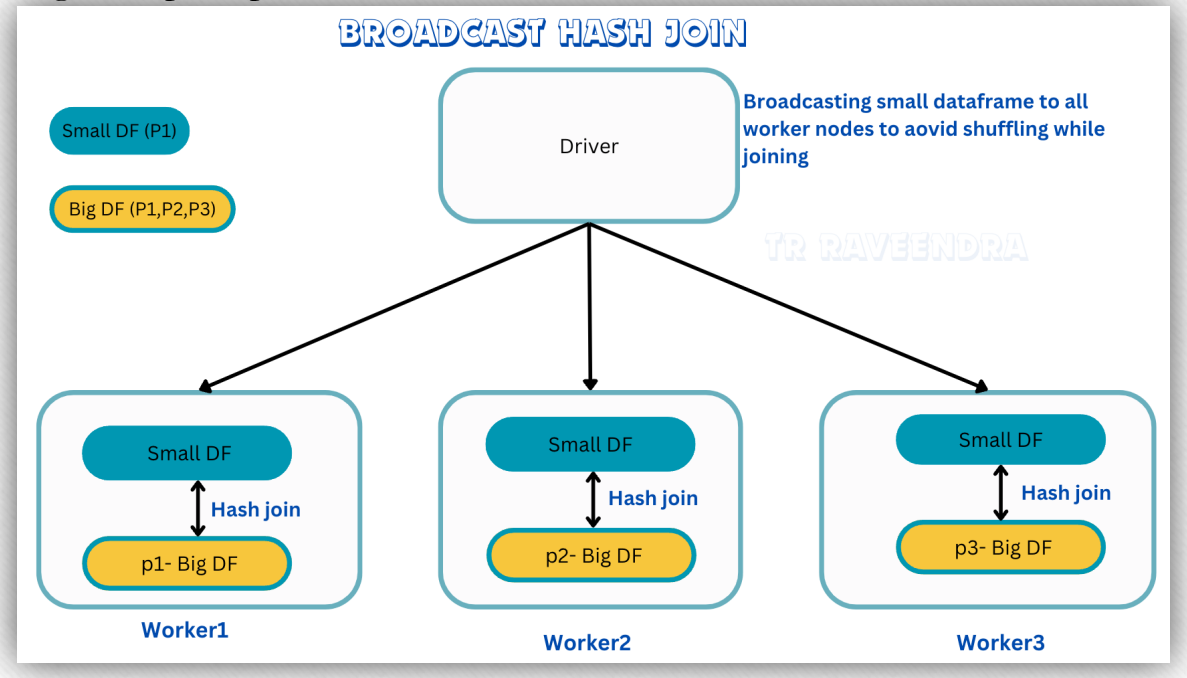
Spark 3.0 and above comes with AQE (Adaptive Query Execution), which can also convert the sort-merge join into broadcast hash join (BHJ) when the runtime statistics of any join side is smaller than the adaptive broadcast hash join threshold, which is 30MB by default. You can also increase this threshold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

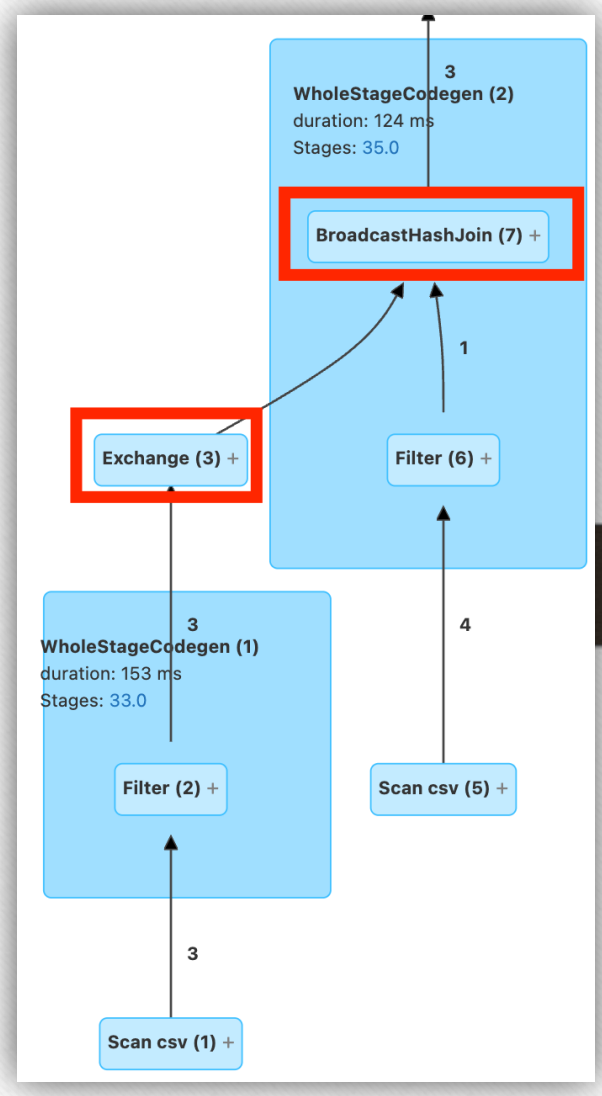
```
%sql
set spark.databricks.adaptive.autoBroadcastJoinThreshold = <size in bytes>
```

Broadcast Hash Join

```
1   %sql
2   -- BROADCAST  or MAPJOIN or BROADCASTJOIN
3   select /*+ MAPJOIN(e)  */ * from emp as e inner join dept as d on e.deptno=d.deptno where e.deptno=10
```

- The smaller join table must be able to fit in the memory of each executor. If the table is too large, the join will fail.
- Broadcast hash join cannot be used when joining two large DataFrames. This is because broadcasting a large DataFrame to all executors can be expensive and can lead to performance bottlenecks.
- Broadcast hash join is not suitable for joins with high data skewness. This is because the broadcast DataFrame will be replicated on all executors, even if the data is skewed. This can lead to inefficient memory usage and poor performance.





Sort-Merge Join

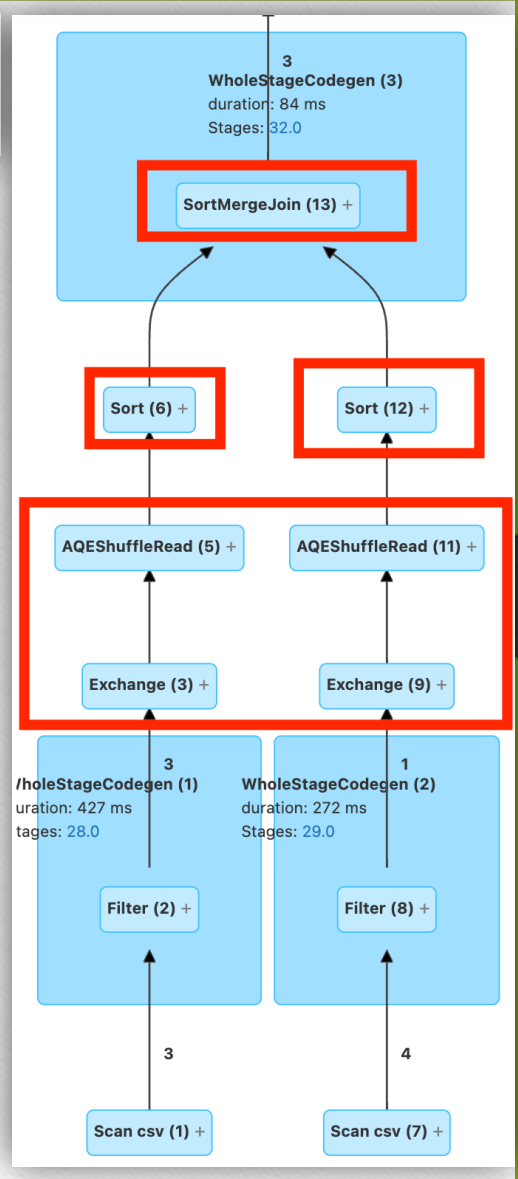
```
1   %sql
2   -- SHUFFLE_MERGE or MERGEJOIN  or MERGE
3   select /*+ MERGEJOIN(e)  */ * from emp as e inner join dept as d on e.deptno=d.deptno where e.deptno=10
```

Sort-merge join (SMJ) is a type of join that is used when both of the join tables are large and cannot be broadcast to all executors. SMJ works by first sorting both tables on the join key, and then merging the sorted tables to perform the join operation.

To perform a SMJ in PySpark, you can use the following steps:

1. Sort both join tables on the join key using the orderBy() function.
2. Join the two sorted tables using the join() function.
3. Specify the how parameter of the join() function to the type of join you want to perform (e.g., inner, left, right).
4. Specify the on parameter of the join() function to the join condition.

SMJ is a more general join algorithm than broadcast hash join, and it can be used to join any two DataFrames, regardless of their size. However, SMJ can be less efficient than broadcast hash join when the smaller join table is small enough to be broadcast.



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)** has been 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 you to 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 mean that Spark will always choose SHJ over SMJ. We are simply defining your preference for this option.

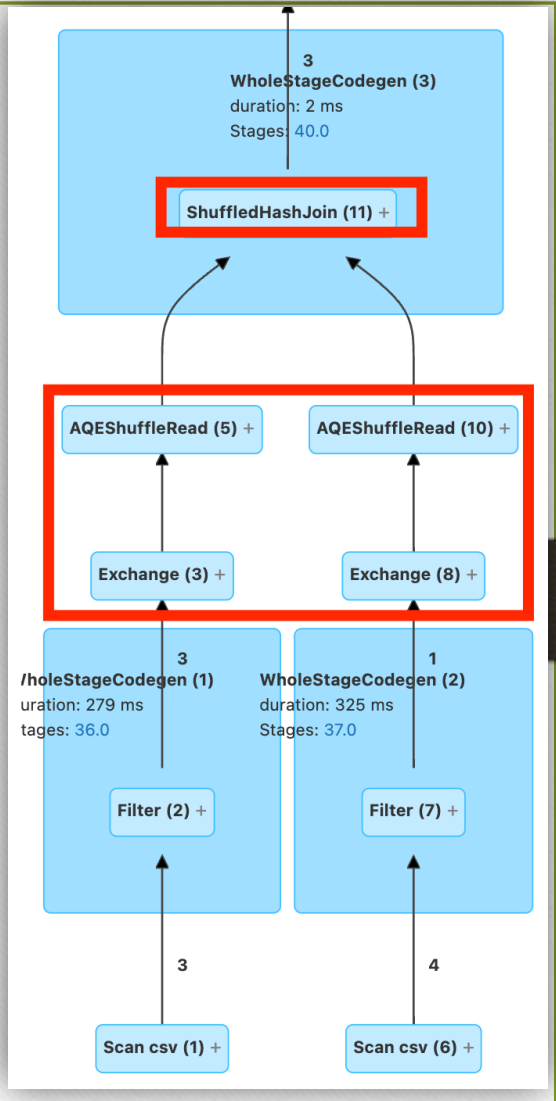
```
%sql
set spark.sql.join.preferSortMergeJoin = false
set spark.sql.join.preferShuffleHashJoin=true
```


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 concerned job, 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, then you can use this option and change the preferSortMergeJoin value to false

Use the **spark.sql.join.preferShuffleHashJoin** configuration property to control which join algorithm Spark uses. By setting this property to true, you can force Spark to use shuffle hash join, even if broadcast hash join or sort-merge join would be more efficient.

```
%sql
-- SHUFFLE_HASH
select /*+ SHUFFLE_HASH(e)  */ * from emp as e inner join dept as d on e.deptno=d.deptno where e.deptno=10
```



Nested Loop Join

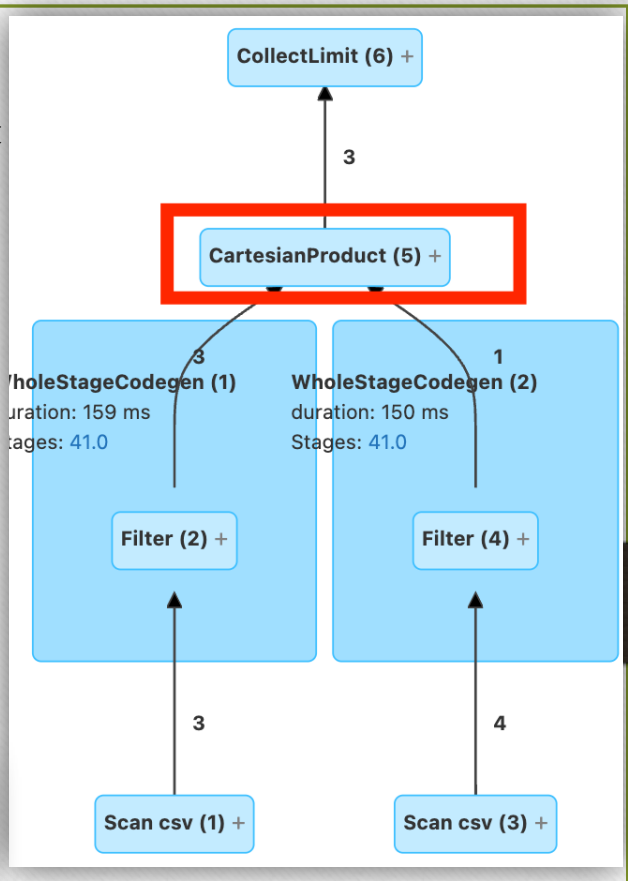
A **nested loop** join in PySpark is a type of join that works by iterating over every row in one table and comparing it to every row in the other table. This can be a very slow and inefficient join algorithm, especially when the join tables are large.

However, a nested loop join can be useful for joining two small DataFrames, or for joining two DataFrames with a complex join condition.

To perform a nested loop join in PySpark, you can use the following steps:

- ☑ Create a new DataFrame to store the joined results.
- ☑ Iterate over every row in the first DataFrame.
- ☑ For each row in the first DataFrame, iterate over every row in the second DataFrame.
- ☑ If the two rows match the join condition, add them to the new DataFrame

```
1  %sql
2  -- SHUFFLE_REPLICATE_NL
3  select /*+ SHUFFLE_REPLICATE_NL(e)  */ * from emp as e inner join dept as d on e.deptno=d.deptno where e.deptno=10
```



Compression Types

`GZIP` compression uses more CPU resources than `Snappy` or `LZO`, but provides a higher compression ratio.
`GZip` is often a good choice for `cold data`, which is accessed infrequently. `Snappy` or `LZO` are a better choice for hot data, which is accessed frequently for analytics.
`BZip2` can also produce more compression than GZip for some types of files, at the cost of some speed when compressing and decompressing.
`HBase` does not support `BZip2` compression.
`Snappy` often performs better than `LZO`. It is worth running tests to see if you detect a significant difference.

CSV Compressions..

- bzip2, deflate, lz4, gzip, snappy,

Cmd 21

```
df_airlines.coalesce(1).write.format("csv").mode("overwrite").option("compression","gzip").save("/bigdata/csv_gz/",header=True)
df_airlines.coalesce(1).write.format("csv").mode("overwrite").option("compression","snappy").save("/bigdata/csv_snappy/",header=True)
df_airlines.coalesce(1).write.format("csv").mode("overwrite").option("compression","deflate").save("/bigdata/csv_deflate/",header=True)
df_airlines.coalesce(1).write.format("csv").mode("overwrite").option("compression","lz4").save("/bigdata/csv_lz4/",header=True)
df_airlines.coalesce(1).write.format("csv").mode("overwrite").option("compression","bzip2").save("/bigdata/csv_bz2/",header=True)
```

Parquet Compressions

- brotli, uncompressed, lz4, gzip, lzo, snappy, none, zstd

Cmd 26

This lzo library is not available by default in databricks. need to install manually if its required.

Cmd 27

```
df_airlines.coalesce(1).write.format("parquet").mode("overwrite").option("compression","snappy").save("/bigdata/parquet_snappy/",header=True)
df_airlines.coalesce(1).write.format("parquet").mode("overwrite").option("compression","gzip").save("/bigdata/parquet_gz/",header=True)
df_airlines.coalesce(1).write.format("parquet").mode("overwrite").option("compression","zstd").save("/bigdata/parquet_zstd/",header=True)
df_airlines.coalesce(1).write.format("parquet").mode("overwrite").option("compression","lz4").save("/bigdata/parquet_lz4/",header=True)
df_airlines.coalesce(1).write.format("parquet").mode("overwrite").option("compression","lzo").save("/bigdata/parquet_lzo/",header=True)
```

Property	CSV	Json	Parquet	Avro	ORC
Human Readable	YES	YES	NO	NO	NO
Compressable	YES	YES	YES	YES	YES
Splittable	YES*	YES*	YES	YES	YES
Complex data structure	NO	YES	YES	YES	YES
Schema evolution	NO	NO	YES	YES	YES
Columnar	NO	NO	YES	NO	YES

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 default. You disable the CBO by changing the following configuration:

```
%sql
set 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 strategy. 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
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 joined (for instance, joining smaller tables first would significantly improve the performance). Join reordering works only for INNER and CROSS joins. To leverage this feature, set the following configurations:

```
%sql
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 statistics on the columns that are being used as join keys and filters.
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>
```

Leverage cost-based optimizer (CBO)

- To properly leverage CBO optimizations, **ANALYZE TABLE** command needs to be executed regularly (preferably once per day or when 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 after 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 cases 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 example, it 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 advantage of the statistics calculated by **ANALYZE TABLE** command. Therefore, it's recommended to run ANALYZE TABLE command regularly to keep the table statistics updated.

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 table. 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:

```
%sql
-- table property
ALTER TABLE table_name SET TBLPROPERTIES ("delta.dataSkippingNumIndexedCols"= val1)
```

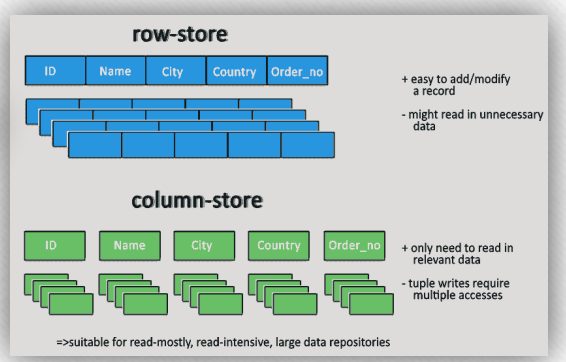

Collecting statistics on long strings is an expensive operation. To avoid collecting statistics on long strings, you can either configure the table property delta.dataSkippingNumIndexedCols to avoid columns containing long strings or move columns containing long strings to a column greater than delta.dataSkippingNumIndexedCols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

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

Column pruning

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 are truly part of the workload computation and are needed by downstream queries. Only those columns should be selected from the source database. This has the potential to have a significant impact on query performance.

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

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



Selecting only the required columns, also known as column pruning, offers several advantages in data processing, particularly in the context of large datasets and analytical workloads. Here are some key benefits of column pruning:

Reduced I/O Operations: By selecting only the columns needed for a specific task or analysis, you can significantly reduce the amount of data that needs to be read from disk, minimizing I/O operations and improving overall performance.
Reduced Memory Consumption: Loading only the required columns into memory reduces the memory footprint of the data, allowing for more efficient processing of large datasets. This is particularly beneficial when working with limited memory resources.
Faster Data Processing: By reducing the amount of data to be processed, column pruning can significantly speed up data processing tasks, especially for complex analyses involving multiple joins and aggregations.
Optimized Data Transfer: Column pruning minimizes the amount of data transferred between different stages of a data processing pipeline, reducing network overhead and improving overall efficiency.
Reduced Processing Overhead: By excluding irrelevant columns, you can reduce the amount of unnecessary processing, such as filtering and type conversions, which can consume CPU resources and slow down the processing pipeline.
Improved Cache Efficiency: Column pruning ensures that only relevant data is cached in memory, making the cache more efficient and reducing the likelihood of cache misses.
Optimized Data Storage: Column pruning can also be used to optimize data storage, as you only need to store the columns that are actually used for analysis or processing. This can save storage space and reduce costs.
Enhanced Data Privacy: By limiting access to only the required columns, you can protect sensitive or confidential data from unauthorized access or exposure.

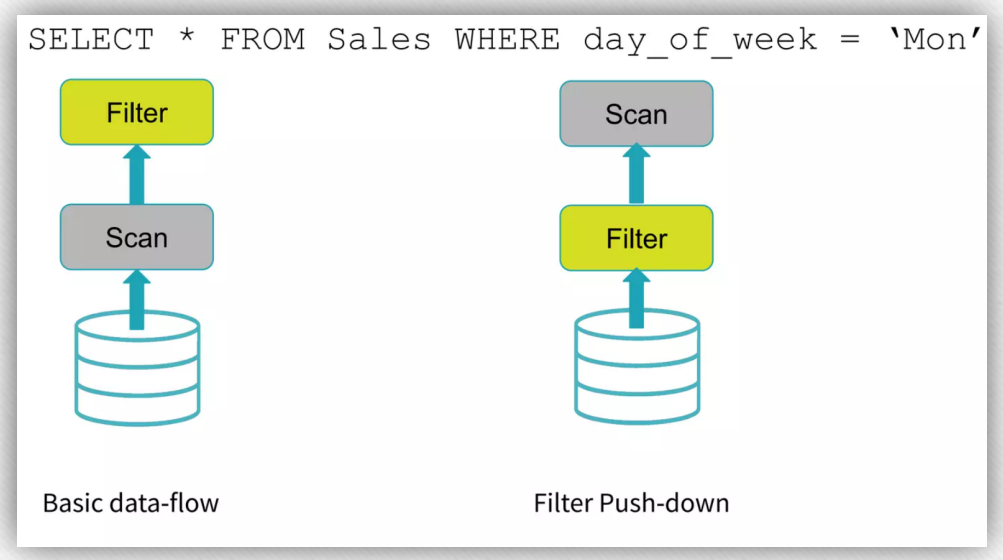
Predicate pushdown

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 dependent. It works for data sources like Parquet, Delta, Cassandra, JDBC, etc., but it will not work for data sources like text, JSON, XML, etc.

```
1   -- SQL
2   SELECT col1, col2 .. coln FROM table WHERE col1 = <value>
3
4   -- PySpark
5   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.



Partition pruning

Delta Lake on Databricks Performance Tuning with Pruning

In addition to eliminating data at partition granularity, Delta Lake on Databricks dynamically skips unnecessary files when possible. This can be achieved because Delta Lake automatically collects metadata about data files managed by Delta Lake and so, data can be skipped without data file access. Prior to Dynamic File Pruning, file pruning only took place when queries contained a literal value in the predicate but now this works for both literal filters as well as join filters. This means that Dynamic File Pruning now allows star schema queries to take advantage of data skipping at file granularity.

1. Static (based on filters) On Both Partititons and Files
2. Dynamic (based on joins) On Both Parttiions and files

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

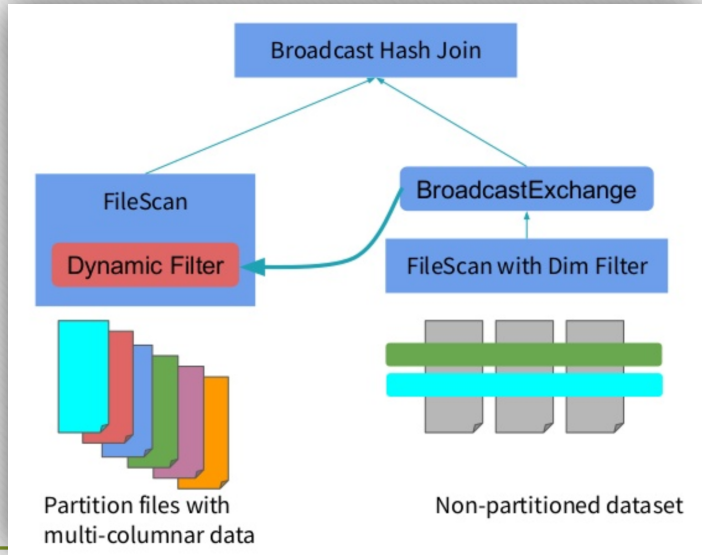
```
-- SQL
SELECT * FROM table WHERE partition_col = <value>

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

Dynamic partition pruning (DPP)

In Apache Spark 3.0+, a new optimization called Dynamic Partition Pruning (DPP) is implemented.
Dynamic partition pruning optimization is performed based on the type and selectivity of the join operation. During query optimization, we insert a predicate on the partitioned table using the filter from the other side of the join and a custom wrapper called DynamicPruning.

The basic mechanism for DPP inserts a duplicated subquery with the filter from the other side, when the following conditions are met: (1) the table to prune is partitioned by the JOIN key (2) the join operation is one of the following types: INNER, LEFT SEMI (partitioned on left), LEFT OUTER (partitioned on right), or RIGHT OUTER (partitioned on left)

In order to enable partition pruning directly in broadcasts, we use a custom DynamicPruning clause that incorporates the In clause with the subquery and the benefit estimation. During query planning, when the join type is known, we use the following mechanism: (1) if the join is a broadcast hash join, we replace the duplicated subquery with the reused results of the broadcast, (2) else if the estimated benefit of partition pruning outweighs the overhead of running the subquery query twice, we keep the duplicated subquery (3) otherwise, we drop the subquery.



Dynamic file pruning (DFP)

Dynamic file pruning (DFP), can significantly improve the performance of many queries on Delta tables. DFP is especially efficient for non-partitioned tables, or for joins on non-partitioned columns. The performance impact of DFP is often correlated to the clustering of data so consider using Z-Ordering to maximize the benefit of DFP

Enabling Dynamic File Pruning

DFP is automatically enabled in Databricks Runtime 6.1 and higher, and applies if a query meets the following criteria:

1. The inner table (probe side) being joined is in Delta Lake format
2. The join type is INNER or LEFT-SEMI
3. The join strategy is BROADCAST HASH JOIN
4. The join type is INNER or LEFT-SEMI

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.

```
1   %sql
2   set spark.databricks.optimizer.dynamicFilePruning=true
```

spark.databricks.optimizer.deltaTableSizeThreshold (default is 10,000,000,000 bytes (10 GB)):
Represents the minimum size (10GB) of the Delta table on the probe side of the join required to trigger DFP. 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

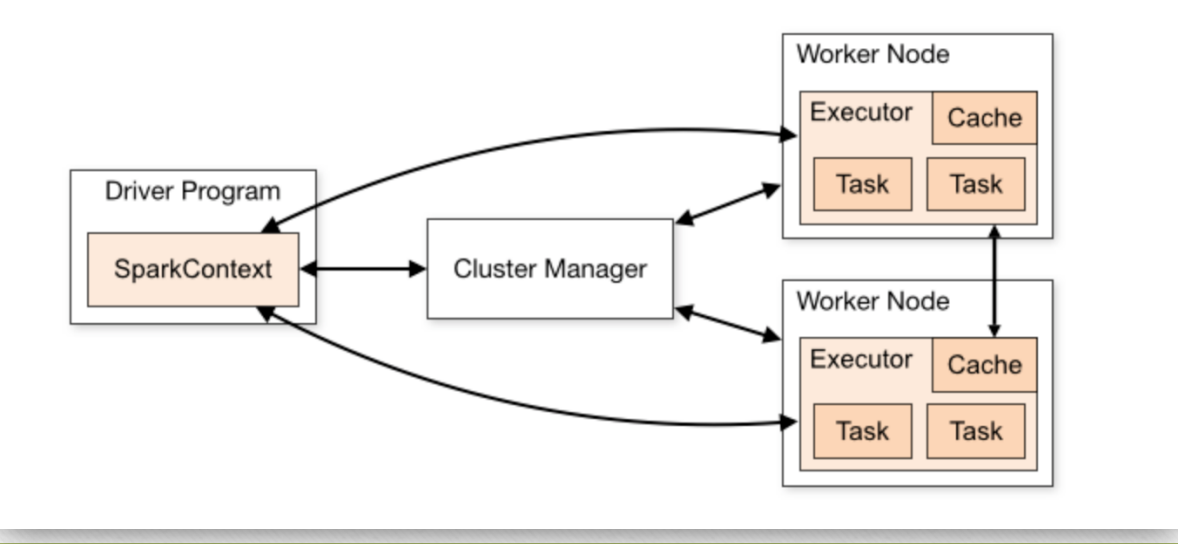
```
1   spark.conf.get("spark.databricks.optimizer.deltaTableSizeThreshold")

'10000000000'
```

Spark cache

Using **cache**() and **persist**() methods, Spark provides an optimization mechanism to cache the intermediate computation of a Spark DataFrame so they can be 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 choose 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 on the same DataFrame

Databricks recommends using Delta caching instead of Spark caching, as Delta caching provides better performance outcomes. The 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 cache uses efficient decompression algorithms and outputs data in the optimal format for further processing using whole-stage code generation. Any compute-heavy workload (with a lot of wide transformations like joins and aggregations) would benefit from Delta caching, especially when reading from the same tables over and over. Always use Delta cache-enabled instances for those workloads.



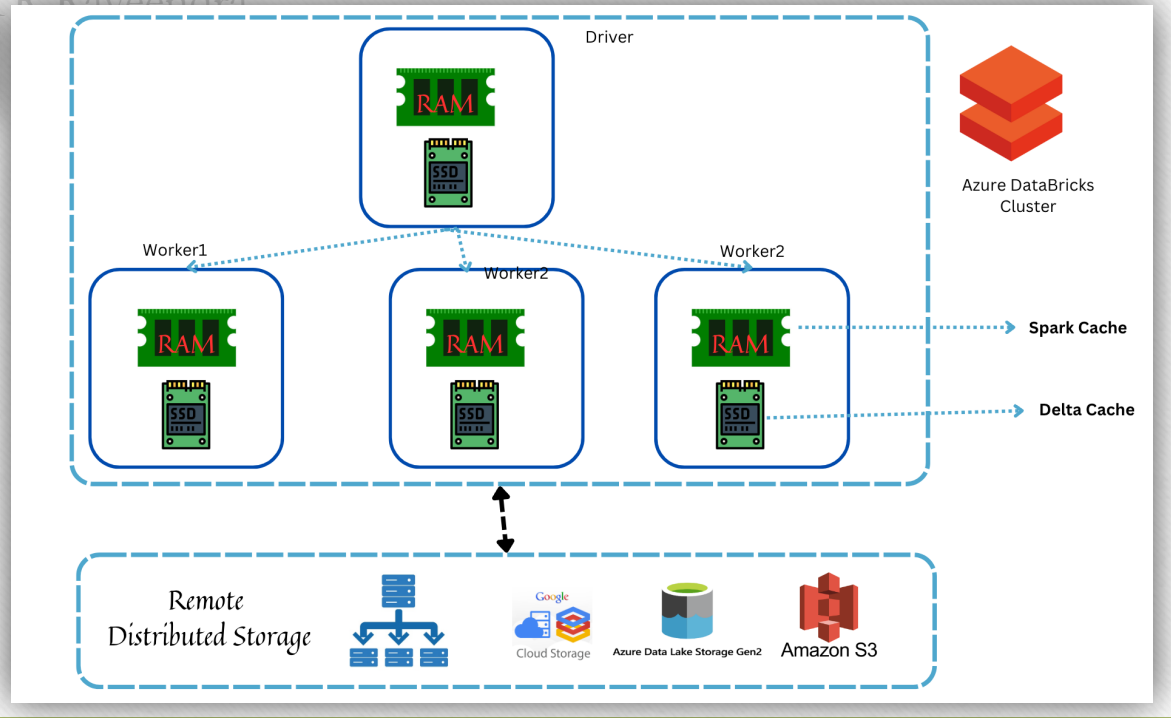
Data Caching

Data Caching — Leverage Caching to Speed Up the Workloads
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 the 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 Azure cloud). To explicitly enable the Delta caching, set the following configuration:

```
1   %sql
2   set spark.databricks.io.cache.enabled = true
```



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 larger 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, it's 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 details.

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 AND 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 table if 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 simultaneously 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 grouping they 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 setting the configuration **spark.databricks.delta.merge.enableLowShuffle** to true.

Purging Old Deleted Data using VACUUM

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 data 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.**

```
1   %sql
2   VACUUM table_name
3   -- table properties
4   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 after 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 be in use by concurrent readers or writers to the table. If VACUUM cleans up active files, concurrent readers can fail, or, worse, tables can be 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 retention 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 OPTIMIZE 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 deletion is handled by the driver), so a small autoscaling cluster of 1 to 4 workers is sufficient

Databricks Cluster Configuration and Tuning

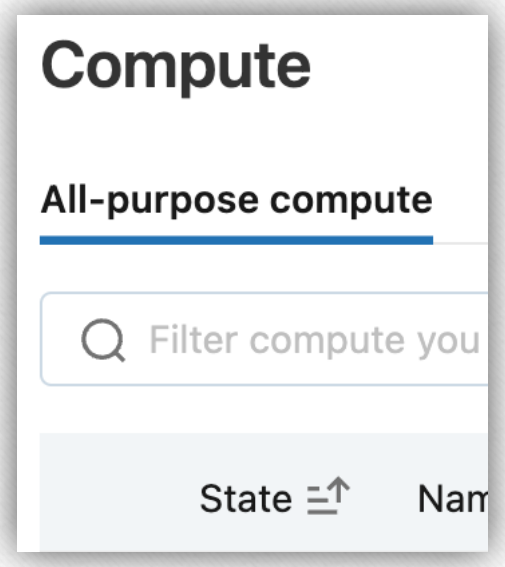
All-purpose clusters vs. automated clusters

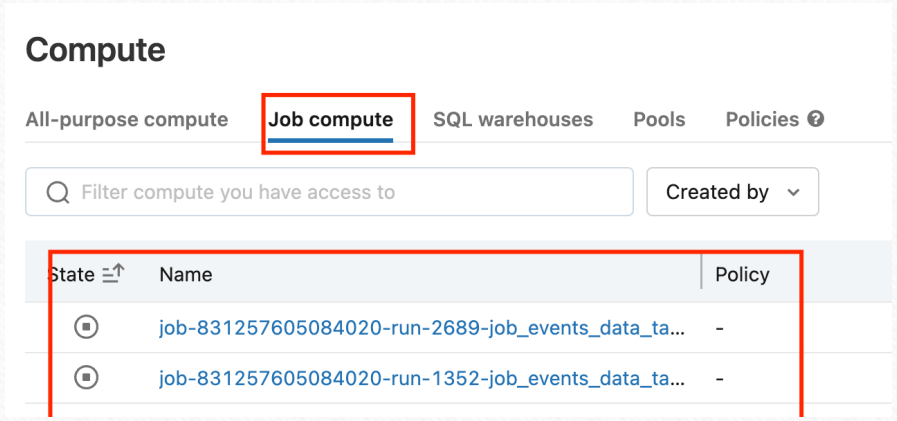
All-purpose clusters should only be used for ad hoc query execution and interactive notebook execution during the development and/or testing phases

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 objects within Databricks SQL. SQL warehouses are also available in serverless flavor, offering you access to an instant compute.

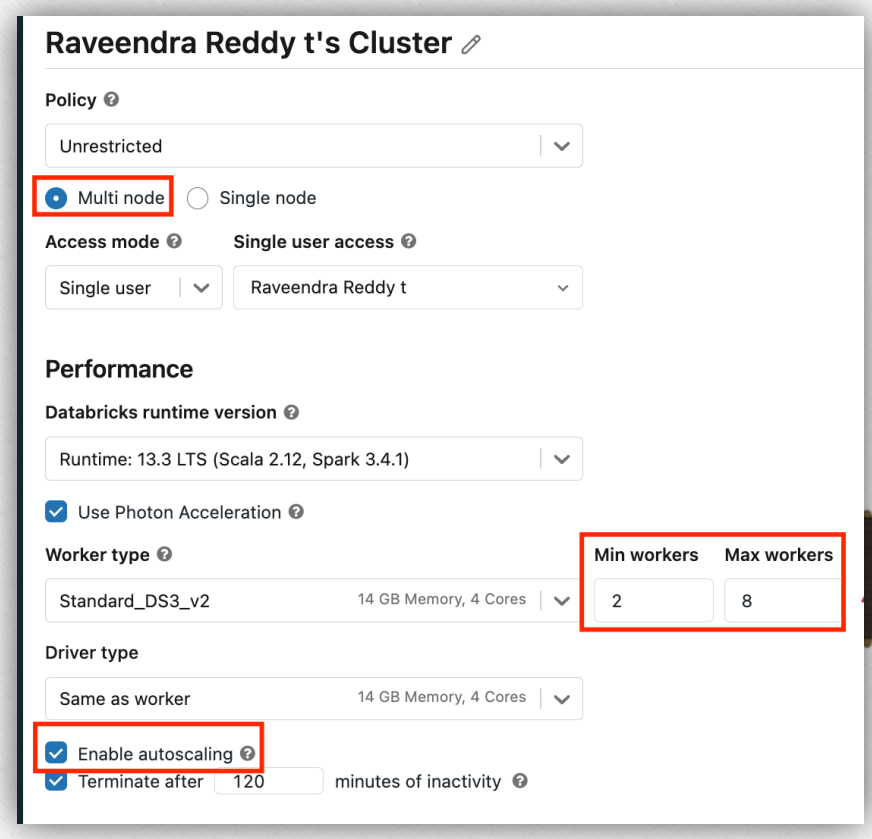




Clusters Autoscaling Option

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

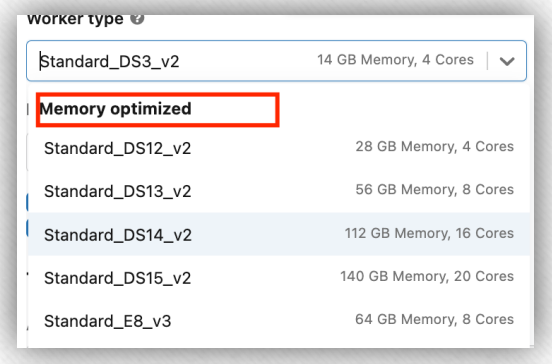
- ☑ Always use autoscaling when running ad hoc queries, interactive notebook execution, and developing/testing pipelines using interactive clusters with minimum workers set to 1. The combination of both can bring good cost savings.
- ☑ For an autoscaling job cluster in production, don't set the minimum instances to 1 if the job certainly needs more resources than 1 worker. Instead, set it to a 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 job has 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 computational 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 SLA 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 parallel on a 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 down. There were some advances made in this aspect in Delta Live Tables, and autoscaling works pretty well for steaming workloads in Delta Live Tables thanks to the feature called Enhanced Autoscaling.

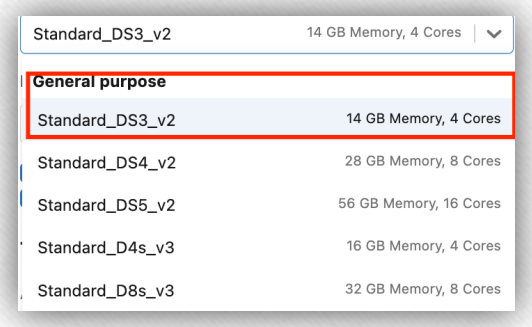


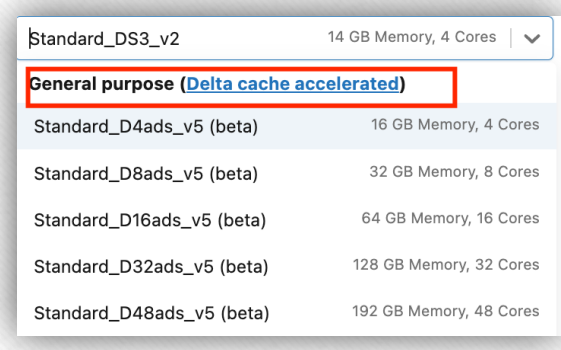
Instance types based on workloads

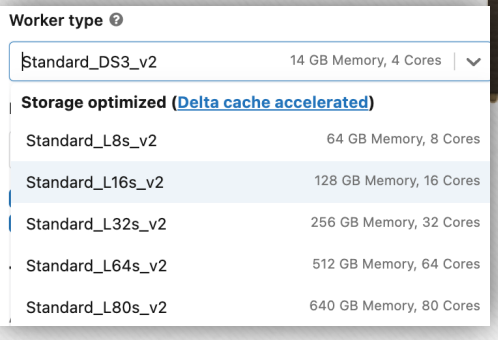
General guidelines to choose instance family based on the workload type:

VM Category	Workload Type
Memory Optimized	• For ML workloads • Where a lot of shuffle and spills are involved • When Spark caching is needed
Compute Optimized	• Structured Streaming jobs • ELT with full scan and no data reuse • To run OPTIMIZE and Z-order Delta commands
Storage Optimized	• To leverage Delta caching • ML and DL workloads with data caching • 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 • 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 guidelines to 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. In this 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 save cost

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 certain is to always set the number of shuffle partitions as a multiplication of the total number of worker cores.

```
-- 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
```

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), indicating 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.

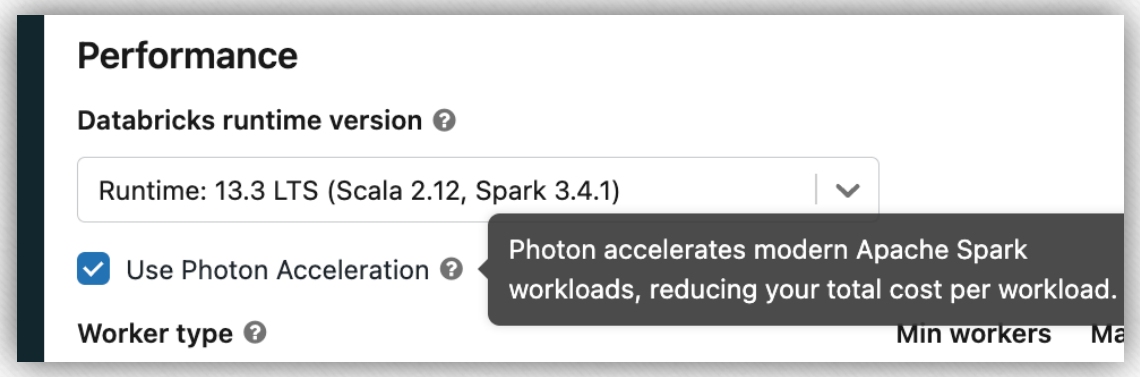
Photon Engine in Clusters

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



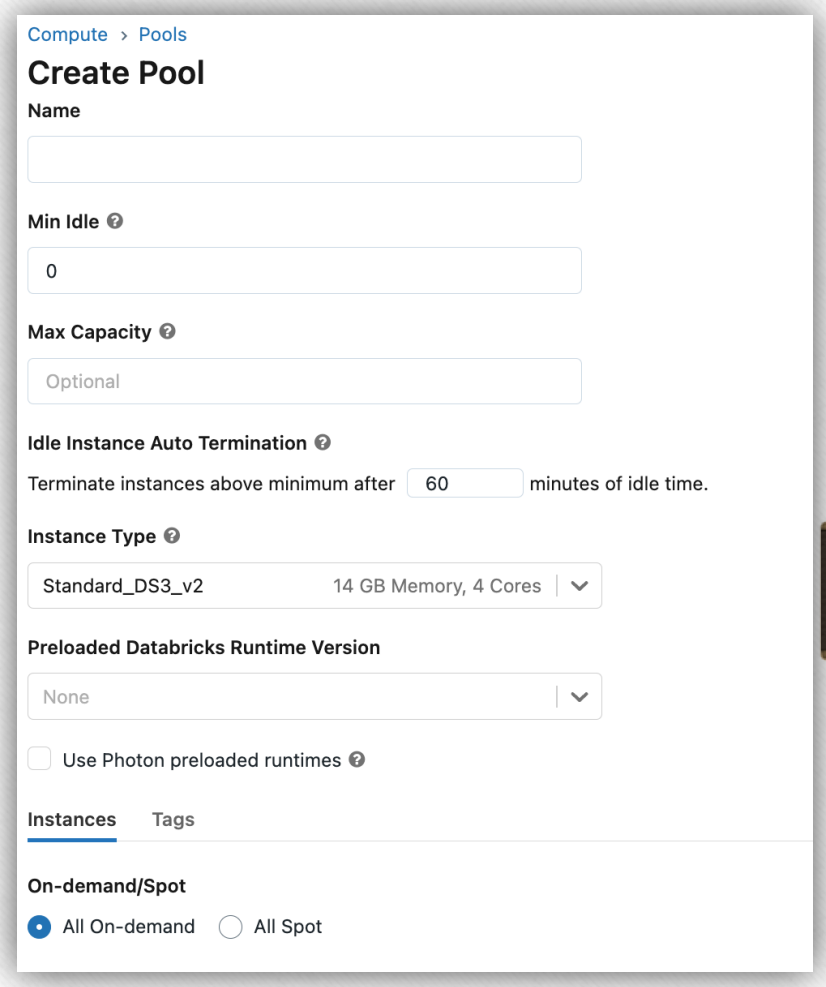
Cluster policy and Instance Pools

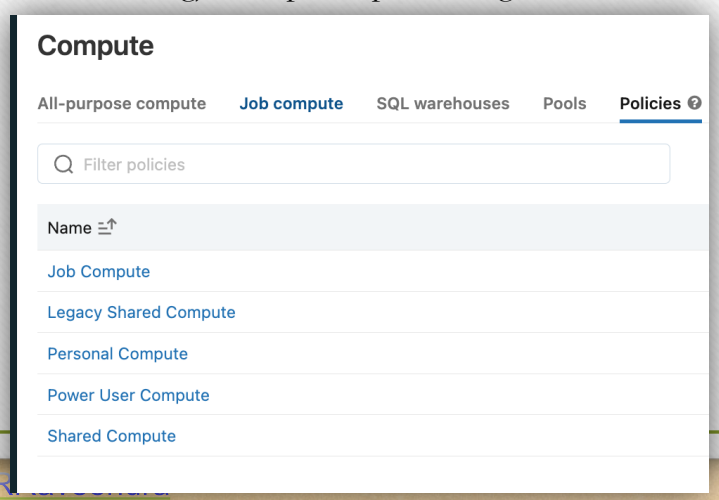
Cluster policy
A cluster policy limits the ability to configure clusters based on a set of rules. Effective use of cluster policies allows administrators to:

- ☑ 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 nodes 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.





Things to follow

- 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 jobs
- 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 in the 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 underlying 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 between 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 collection, which negatively impacts the performance and in some cases can cause job failure. Therefore, it's not recommended to choose workers with more than 128GB 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 Advance cluster options
-

TR Raveendra

THANK YOU

YouTube: www.youtube.com/@TRRaveendra

LinkedIn
@TRRaveendra