Contents

[1. Objective 3](#_Toc107159566)

[1.1. Cookbook will address the below design decisions 3](#_Toc107159567)

[2. Guide To Cookbook 4](#_Toc107159568)

[3. Using Unravel for Visibility and Troubleshooting 5](#_Toc107159569)

[4. Spark Optimizations 8](#_Toc107159570)

[4.1. Workspace and Cluster Configurations 8](#_Toc107159571)

[4.1.1. Optimized Cluster Configurations 10](#_Toc107159572)

[4.1.2. VM Instances and which one to use when 11](#_Toc107159573)

[4.1.3. Delta to Synapse dedicated pools 13](#_Toc107159574)

[4.1.4. Linked Service configuration 14](#_Toc107159575)

[4.2. Optimized data Storage 14](#_Toc107159576)

[4.2.1. Delta 14](#_Toc107159577)

[4.2.2. Small file Problem 15](#_Toc107159578)

[4.2.3. Partitioning 16](#_Toc107159579)

[4.2.4. File Size Tuning 17](#_Toc107159580)

[4.3. Incremental Loads 17](#_Toc107159581)

[4.4. Controlling Data Shuffles and Spills 18](#_Toc107159582)

[4.5. Data Skew and How do you handle it 20](#_Toc107159583)

[4.5.1. Filter Skewed Values 22](#_Toc107159584)

[4.5.2. Skew Hints 22](#_Toc107159585)

[4.5.3. AQE Skew Optimization 22](#_Toc107159586)

[4.5.4. Salting 23](#_Toc107159587)

[4.6. Broadcast Joins 23](#_Toc107159588)

[4.7. Shuffle Hash Joins over Sort Merge Joins 24](#_Toc107159589)

[4.8. Spark Cost Based Optimizer (CBO) 24](#_Toc107159590)

[4.9. Controlling Data Spills by tuning Shuffle Partitions 25](#_Toc107159591)

[4.9.1. Manually Configure Shuffle Partitions 25](#_Toc107159592)

[4.9.2. Auto Tuning using Adaptive Query Execution 27](#_Toc107159593)

[4.10. Data Explode 28](#_Toc107159594)

[4.11. Data Caching 29](#_Toc107159595)

[4.12. Temporary Views 30](#_Toc107159596)

[4.13. Data Skipping & Pruning 31](#_Toc107159597)

[4.14. Delta Merge 32](#_Toc107159598)

[5. Anti-Patterns 33](#_Toc107159599)

# Objective

# Cookbook will address the below design decisions

|  |  |  |
| --- | --- | --- |
| **S No** | **Factors** | **Design Decisions** |
| 1 | Databricks Workspace | * Using Premium and Standard workspaces judicially * Premium workspaces to be used in production * Leveraging governance and security related features * Teams should have separate workspaces |
| 2 | Cluster configurations | * Large clusters being used for development, UT, DDLs, syntax checks and ad-hoc analysis |
| 3 | Datafile formats | * Using optimal data format for analytical operations * Converting to Columnar data format (Delta) before running any window/group by / de duplication / business logic * Using right granularity for folder partitioning * Use of row serialized formats (csv, json, avro) for source of truth layer * Inefficient use of deltaRe-organizing column layout to utilize column statistic collection using delta |
| 4 | Incremental Loads | * Using structured streaming and Delta Merge operations to process incremental data * Using optimal data partitioning strategy to leverage file skipping at partition boundaries |
| 5 | Spark Configs | * Configure spark configurations based on your workload   e.g. – set spark.sql.shuffle.partitions   * Use Spark AQE based optimization configurations * Configure auto broadcast threshold |

# Guide To Cookbook

**Agenda1 : Interactive to Job cluster migration**

Check the section on [Optimize Cluster and Workspace](#_Workspace_and_Cluster_1) and action below points

* Check the auto termination and set it to maximum 10 mins.
* Check number of activities running on the cluster using ADF excel. If your activities are missing, please reach out with ADF name and we will fetch the details.
* Check if the activity start/end time of activities are overlapping.
* If there is a notebook activity with no overlap with other activities, it can be moved to a job cluster.
* If there are multiple notebooks overlapping and don't have any interdependencies with other activity types (Example, Stored Procedure, Copy, send email, etc), they can be moved to job cluster by using orchestrator notebook.
* If there is a foreach activity executing multiple notebooks in sequence/parallel on same job cluster , they can be moved to a single job cluster using orchestrator notebook.
  + If due to some reasons, orchestrator notebook is not feasible, and if there are few notebooks whose execution time is much higher, then these can be moved to job cluster and the config of interactive cluster can be reduced.

**Agenda2: Optimizing Linked Services**

* Ensure there are multiple Linked services in ADF based on execution pattern
* Check ADF excel to identify list of ADB linked services and how many activities use the same linked service
* If LS is auto-scaling job clusters, change to fixed E\*ds\_v4 and test
  + If max autoscaling >16 choose 16 nodes
  + If max autoscaling <16 choose 8 nodes
  + If autoscaling min<>max nodes is high(>20 diff) reduce autoscaling window and if needed create separate interactive cluster for exception notebooks
* Check the section [Linked Service Configurations](#_Linked_Service_configuration)

**Agenda3: optimizing Spark Code**

* For the notebook activity, check [unravel UI](#_Using_Unravel_for) by going to the jobs tab.
* In the free text box, use the activity name as a wild card, \*adfname\*activity\*
* Go to the spark section and look for recommendations
* Check if there is an instance type recommendation or SQL Insights
* If there is an [instance type recommendation](#_Workspace_and_Cluster_1), note it down to change for the next run. Reconcile the config with cookbook. For example, if unravel recommends moving down to E\*as\_v4, use E\*ds\_v4 instead.
* If there is no instance type recommendation but there is a SQL insight, check if there is a data skew or contended driver or GC time insights. If yes, the particular SQL can be optimized by referring cookbook sections on [Optimized Data layout](#_Optimized_data_Storage), [data shuffle](#_Controlling_Data_Shuffles) and [Data skew](#_Data_Skew_and).

# Using Unravel for Visibility and Troubleshooting

Unravel helps us to observe how our data pipelines are performing. It analyses the resource, Spark queries, etc and gives us insights which can help us to optimize our Spark code to improve performance and in turn reduce cost.

Below is how you can navigate through unravel to get insights on notebook activities.

* After logging into Unravel dashboards, click the job cluster
* On the search text box, give a wild card search for the Azure Data Factory (ADF) name and the pipeline you want to check.
* For example
  + ADF instance name is *adfazewpoceanpnl*
  + the pipeline name is *ML\_Delta*,
  + you can give a wildcard search as \*adfazewpoceanpnl\*ML\_Delta \*

We can search for our data factory notebook activities by logging

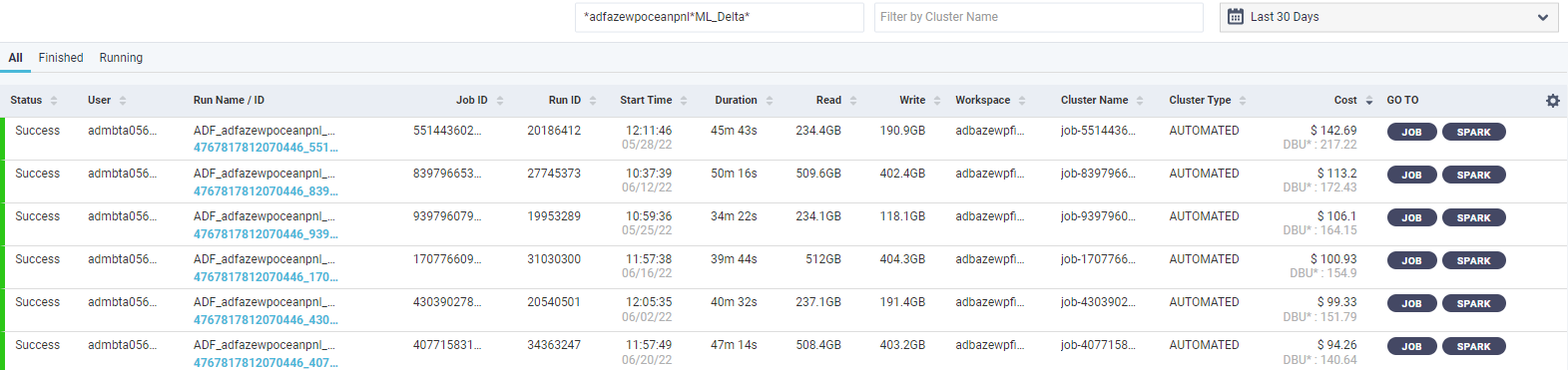
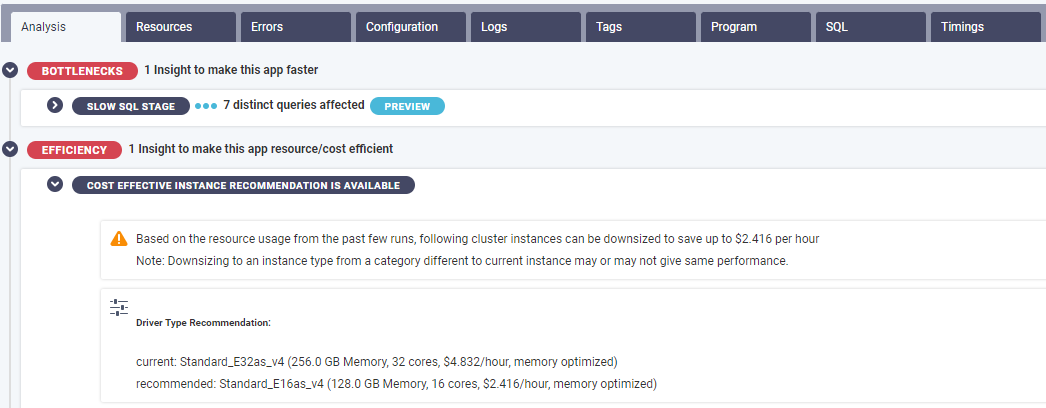


Figure 1 - Unravel Job Page UI

* You can use the drop down on top right to filter execution details for up to last 30 days.
* Click on the SPARK button to get the insights from unravel. These insights are broken into two categories
  + SQL Insights
  + Cluster recommendations
* Cluster recommendations are straight forward and can be used to change the cluster configs based [Optimized Cluster Configuration](#_Optimized_Cluster_Configurations) and test the code immediately.
* SQL insights do not offer any recommendations; however, they give us a starting point on what we need to work on to improve the performance of our code.



* The second dashboard of interest is inefficient tab under the cloud clusters page which gives us a bunch of inefficient events for all our workloads.

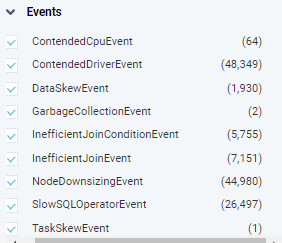


Figure 2- Inefficient events in unravel UI

* Below table gives the recommendations for all the inefficient events which are available in unravel.

|  |  |  |
| --- | --- | --- |
| **Unravel Event** | **Reasons** | **Recommendation/Guidance** |
| DataSkewEvent  TaskSkewEvent | Uneven distribution of data causing skew during joins and aggregation | [Check Data Skew Section](#_Data_Skew_and) |
| ContendedDriverEvent | Too much load on the driver, it can happen when   * Too many jobs running on the same cluster with a small driver * Using Pandas * Collecting too much data on driver | * Don’t use Pandas as it only runs on the driver instead use Pyspark or SQL on spark. * Don’t call collect() function to collect data on the driver. * Don’t share a cluster with a small driver with too many jobs, instead start separate job clusters. * Use a bigger driver (32GB+ RAM) if doing a lot of broadcasts. * Refer [Cluster Tuning](#_Workspace_and_Cluster_1) section |
| SlowSqlEvent  InefficientJoinEvent | Join operation can be really slow when:   * Inefficient join strategies * Sub optimized shuffle partitions * Scanning unnecessary data * Caching not leveraged * Persisting work/staging tables * Cluster is under provisioned | Refer the following cookbook sections:   * Check [Control Data Shuffle and Spills](#_Controlling_Data_Shuffles) * Check [Cluster Tuning](#_Workspace_and_Cluster_1) |
| NodeDownsizingEvent | Overprovisioned cluster | Refer [Cluster Tuning](#_Workspace_and_Cluster_1) section |
| GarbageCollectionEvent | Long GC pauses can slow down or even fail the job. It can happen when:   * Using a huge driver or workers (>= 256GB RAM) * Collecting data on the driver * Using Spark Cache to cache a dataframe in memory | * Don’t call collect() function to collect data on the driver. * Don’t create a driver or workers with more than 128GB RAM. * If none of the above solutions help, use [G1GC](https://www.oracle.com/technical-resources/articles/java/g1gc.html#:~:text=The%20Garbage%20First%20Garbage%20Collector,and%20to%20maintain%20good%20throughput.) garbage collector by setting spark.executor.defaultJavaOptions and spark.executor.extraJavaOptions to -XX:+UseG1GC under Advance cluster options |

# 

# Spark Optimizations

# Workspace and Cluster Configurations

**Problem Statement:** Significant spend due to inefficient cluster/VM sizing, leading to sub-optimal design patterns [cost, run-time, etc..]

Patterns to look for:

1. Underutilized cluster.
2. Prod/Dev workloads on executing on interactive clusters [instead of job clusters]
3. Cluster idle while loading data into Synapse dedicated pools/SQL DW.
4. Default config for Auto-terminate in Interactive clusters is high (default 120 mins)
5. ADF linked services with auto-scale job clusters

**Troubleshoot:**

|  |  |
| --- | --- |
| **Where to check** | **What to check** |
| Autoscaling Linked Services | LS configurations or cluster configs |
| Unravel UI | Unravel UI shows cluster downsizing recommendation |
| Ganglia Metrics | Shows unutilized cluster |

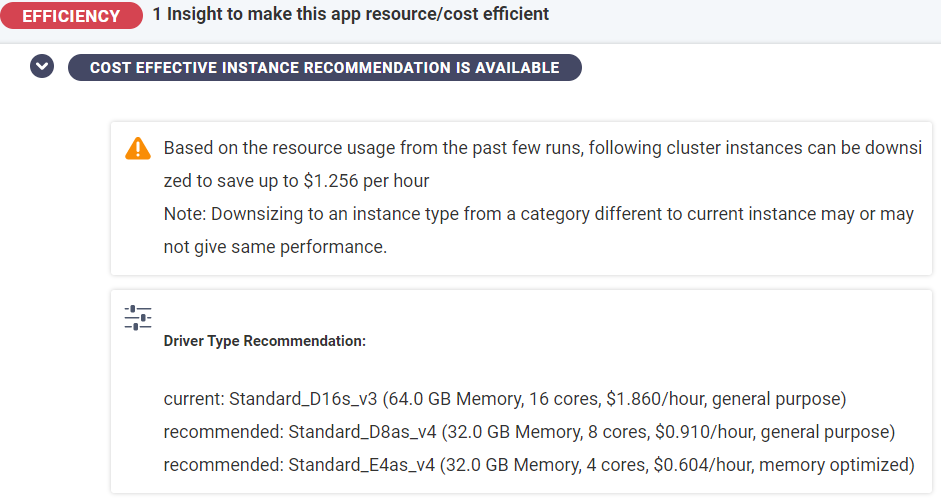
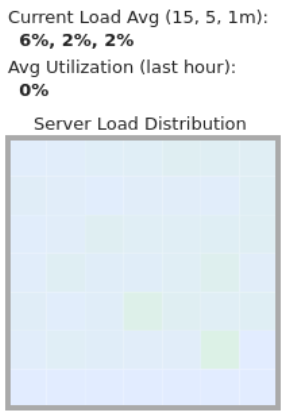


Figure 3-Unravel UI giving cluster recommendation



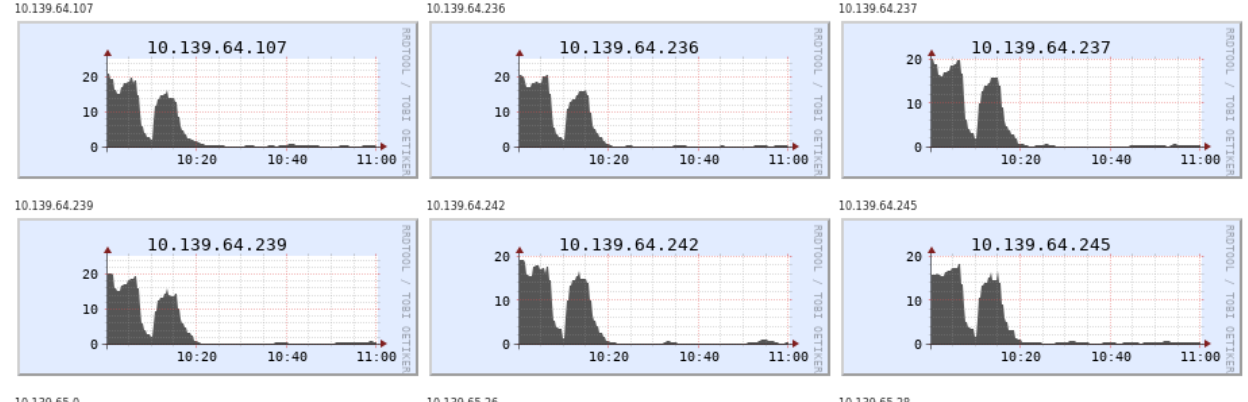


Figure 4 - Underutilized cluster

**Guidance:**

|  |  |  |
| --- | --- | --- |
| **Problem** | **Guidance** | **Additional Guidance** |
| Underutilized Cluster | * Change the cluster config based on Unravel recommendation. * Change cluster config based on OMS recommendation * [Check Cluster Recommendation](#_Workspace_and_Cluster) * [Check Which VM instance to use](#_VM_Instances_and) | * For all compute heavy recommendation with multiple joins, aggregations, etc, start with a **E16ds\_v4 or E8ds\_v4** VM instance with 1-16 worker nodes and increase or decrease the nodes or VM type based on cluster utilization. * Use fixed node job clusters as much as possible for production workloads |
| Most workloads executing on interactive clusters | * Use [interactive clusters](#_Workspace_and_Cluster_1) only for one or two rounds on UT and then switch to job clusters. * Use fixed node job clusters for production workloads | * If the notebooks are executed in an iteration, interactive cluster is the way to go. |
| Start time of job cluster is time consuming | * Use instance pools * [Check Cluster recommendations](#_Workspace_and_Cluster) | * Set the idle time of instance pool based on orchestration. Recommendation is to leave it at max 5 mins * Leave the max and min idle instances as optional |
| Cluster is idle while writing into SQL DW/Synapse dedicated pools | [Check Delta to Synapse](#_Delta_to_Synapse) | * Always use auto-scaling job clusters to copy data into synapse. |
| Autoscaling Linked Services | [Check Linked Service Configuration](#_Linked_Service_configuration)  [Check Optimized Cluster Configuration](#_Optimized_Cluster_Configurations) |  |

# Optimized Cluster Configurations

This section gives an overview of Standard and Premium workspace, job and interactive cluster and helps us decide on which VM instance we should use for which data loads.

|  |  |  |
| --- | --- | --- |
| **Problem** | **Guidance** | **Remarks** |
| Workspace Type | Use Standard workspace in development environment instead of premium | Savings of up to 17% |
| Interactive Cluster | * Single node cluster for development, DDL, syntax checks and ad-hoc analysis * Use Spot Instances * Do not use instance pools * Auto Termination time of max 10 mins * Do not use for more than 2 or 3 UT runs. * Use right instance types based on workload pattern * Use interactive clusters for pipelines which have parallel iterations or use an orchestration notebook on job cluster, if possible. | * Single node clusters help in significant cost savings * Spot instances provide unused resources at heavy discounts * Idle instances in Instance pools incur cost * Idle cluster with auto termination more than 10 minutes unnecessarily incurs cost * Switch to a job cluster once the code and cluster configuration is baselined |
| Job cluster | * Use job clusters mandatorily for production workloads unless there are too many iterations in parallel * Use instance pools * Use fixed node clusters * Use delta cache enabled machines where possible * Do not use job clusters for pipelines which have parallel iterations. * Use job clusters using instance pools for sequential iterations. If there are too many iterations and the pipeline is SLA bound, move to interactive clusters. * Use fixed node job clusters. * For any parallel iterations, prefer an orchestration notebook using a single job cluster | * Instance pools help in reducing the cluster start times * Cluster does not spend lot of time in autoscaling and enables faster data processing * Delta cache enables to fetch dataset from memory, reducing Disk I/Os * Creating a job cluster for each parallel iteration will be much more costly than using an interactive cluster. * Autoscaling is an expensive operation and its better to benchmark the number of nodes required during Unit Testing. |
| Instance Pools | * The Databricks runtime in job cluster and Pool should be same * Mark the max and minimum idle instances as optional * Auto termination of max 5 mins |  |
| Copying data from delta to Synapse dedicated pools | * [Check Delta to Synapse dedicated pools](#_Delta_to_Synapse) |  |

**For queries with many transformations, use a E\*ds\_v4** **machines**

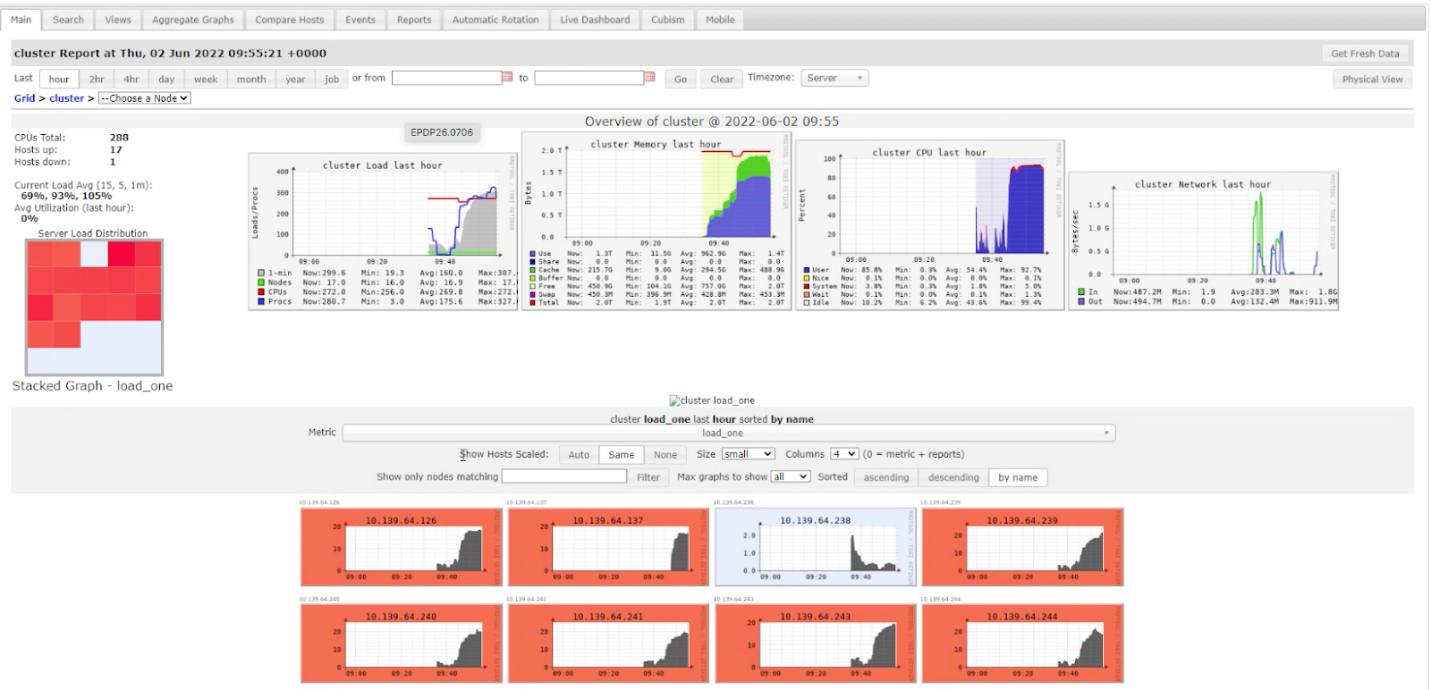
# VM Instances and which one to use when

Below table depicts which instance types should be used when. As a thumb rule, always start with a compute optimized cluster(F4) for simple workloads and work your way up to other instances

|  |  |
| --- | --- |
| **VM Categories** | **When to use** |
| Memory Optimized (Esds\_v4 delta cache enabled VMs, example E16ds\_v4) | * ML workloads with data caching * DE workloads with multiple joins and aggregates. * When shuffle spill is a problem due to joins * When caching is a requirement |
| Compute Optimized (Fs\_v2 series machines) | * ETL with full file scans and no data reuse. * Apt for structured streaming * Reading from and writing to disk |
| Storage Optimized (L Series machines) | * By default, delta cache enabled machines * Last resort when Shuffle spill is still a problem. * When caching is a requirement |
| General Purpose (D Series machines) | * Extremely simple workloads * No other requirements like joins or aggregations |

**Recommendation for Worker Nodes**

|  |  |
| --- | --- |
| **Phase** | **Recommendation** |
| Development | Use a single node cluster or local machine for development  Use single node cluster for DDL, syntax checks, adhoc analysis |
| Unit Testing | First UT run end to end execution on an interactive cluster based on data volume and size  Use a job cluster for remaining runs |
| Performance Testing | Execute the notebook with a fixed 8 node E\*ds\_v4 cluster and set the shuffle partition to twice the number of cores   1. If SLA being met and cluster utilization is greater than 80% and, use this configuration for the job cluster. 2. If SLA is being met and the cluster utilization is less than 80% and, reduce the number of nodes by offset of 4 and test again. 3. If SLA is not being met and the cluster utilization is less than 80%, reduce the number of nodes by offset of 4 and check sections [data layout](#_Optimized_data_Storage), [Data Spills](#_Controlling_Data_Shuffles) and [Data Skew](#_Data_Skew_and) 4. If SLA is not being met and the cluster utilization is greater than 80%, check sections [Data Spills](#_Controlling_Data_Shuffles) and [Data Skew](#_Data_Skew_and). If the SLA is still not being met, increase the number of nodes by offset of 4 and repeat the process. |

*Figure 5 - Ganglia UI of a well utilized cluster(Load should always be greater than 80-90%)*

**Recommendation for Driver Node**

1. Driver node can be of the same instance as of worker node or even smaller depending on the worker load.
2. Driver node can also be of a larger config if there are large datasets which are being returned/collected or if there are large broadcast joins or there are many jobs/notebooks running on the same cluster.

# Delta to Synapse dedicated pools

* Data is copied from Data Lake to Synapse dedicated pools in two steps:
  + 1. Data is copied to a staging blob storage
    2. Data is copied from staging to synapse tables
* Though, the first step requires a bit of compute power, the second step uses the compute power of synapse, and the Spark cluster sits idle until the copy is finished.

**Recommendations**

* Always use auto-scaling job clusters to copy the data into synapse dedicated pools
* Don’t copy the data into synapse in the same notebook where the compute is being done.
* Have a separate notebook which copies the delta tables to synapse.
* Orchestrate the pipelines in a way that the data copy to synapse dedicated pool starts once all the transformations for a batch is complete. Schedule the copy to synapse using a single job cluster in parallel. This will help reduce the overall batch execution time and in turn, give significant cost savings.

# Linked Service configuration

* Ensure that there are multiple Linked Services based on execution pattern
* If Linked Service is an autoscaling linked service,
  + If the notebooks using the Linked Service are complex and have multiple joins, aggregations, etc, change the Linked Service to use E\*ds\_v4 Machines
  + For simple workloads which have max autoscaling nodes less than 16, choose 8 nodes fixed cluster.
  + For workloads with max autoscaling greater than 16, choose E\*ds\_v4 with fixed 16 nodes cluster
    - For SLA bound notebooks, it can be increased to 20 or 24 fixed nodes clusters as well.
  + For autoscaling greater than 32 nodes, choose 32 nodes fixed Linked Service.

# Optimized data Storage

# Delta

**Problem Statement:** Long running query due to underlying format or number of files such as many small csv or avro files, or scanning unnecessary files

**Guidance**: Convert row serialized source files (csv, json, avro) or columnar ORC from cleansed layer, as-is, into an optimized Delta format before performing any further operations.

* Delta is an optimized file format for Spark, especially in the Databricks ecosystem.
* Under the hood Delta uses versioned Parquet files to store your data, and snapshots of your data. In addition, Delta stores metadata about your data: which files belong to which snapshot, transaction logs for upserts. Apart from the versions, Delta Lake also stores a transaction log to keep track of all the commits made to the table or blob store directory to provide ACID transactions.
* Delta is extremely useful for Spark Structured Streaming, simplifying delete/insert/update transactions on incremental data loads.
* Delta allows you to time travel and view a point-in-time snapshot version of the data.
* Delta has built-in features that allow you to optimize the physical data layout. Reducing scanning of many files and managing, enable file skipping and more advanced page skipping in parquet files.

# Small file Problem

One of the most common performance bottlenecks in Big Data world is smaller file problem as each task creates its own output file. During any transformation, reading too many files results in high latency and hence, decreased performance of the transformation. Hence, it is always recommended to have a file size varying between 32 MB to 1 GB depending on the use case (usually 32MB to 256MB works just fine)

Delta comes with couple of compaction features (bin-packing) out of the box.

|  |  |  |
| --- | --- | --- |
| **Feature** | **Syntax** | **Guidance** |
| Optimize and Zorder By | OPTIMIZE table\_name [WHERE predicate] [ZORDER BY (col\_name1 [, ...] ) ] | * Compact smaller files into large files of size up to 1 GB * Zorder on frequently used columns in transformations to enable spark to read only required files. * Don’t Zorder on more than 3 or 4 columns * Don’t Zorder on dates or long string columns * For large tables or tables which undergo multiple merges choose a relatively smaller file size by using   Set delta.targetFileSize = <size in bytes>   * Use a storage or compute optimized cluster. * Do not execute it as a part of the batch execution. Should be executed as a part of housekeeping activities. |
| Auto Optimize | * ALTER TABLE [table\_name | delta.`<table-path>`] SET TBLPROPERTIES (delta.autoOptimize.optimizeWrite = true, delta.autoOptimize.autoCompact = true) * CREATE TABLE <table-name> (columns) TBLPROPERTIES (delta.autoOptimize.optimizeWrite = true, delta.autoOptimize.autoCompact = true) | * Compacts smaller while writing into delta table and tries to achieve a file size of 128MB * **Optimized Writes** dynamically optimizes Apache Spark partition sizes based on the actual data and attempts to write out 128 MB files for each table partition * **Auto compact** tries to further compacts the files to achieve a file size of 128 MB. It triggers a fresh job once the write to delta is complete. * Do not use auto optimize if data size is in Terabytes and cluster VMs are not storage optimized. * If job is SLA bound, make auto compact as optional and use only optimized write to reduce the execution time * For tables which undergo multiple merges choose a relatively smaller file size by using   Set delta.targetFileSize = <size in bytes> |
| Vacuum | * + - VACUUM table\_name     - VACUUM delta.`delta-path`   Use the below syntax to override the default setting of retention to 7 days   * + - SET spark.databricks.delta.retentionDurationCheck.enabled to false     - VACUUM table\_name RETAIN 16 HOURS | * Removes uncommitted/unreferenced files saving storage cost and also reducing the file scans during transformation. * Default retention is 7 days, however, having two versions should be enough. * Do not execute it as a part of the batch execution. Should be executed as a part of housekeeping activities. |

# Partitioning

* + Partitioning your dataset is a great strategy to enable data skipping and reducing data read for downstream processing if data size is huge
  + Avoid applying partitions on a small table as it can have negative impact due to over partitioning leading to small file read problem
  + Partition the datasets on low cardinality columns which are used as filters/join condition/ group by functions.
  + Logical date columns related to key business events rolled up at month/year level granularity can also be used
  + System dates may not be best choice as they are usually not correlated with downstream access patterns and can have data from backdates.
  + Do not create partitions on skewed columns as it can have an adverse impact on performance.

# File Size Tuning

* + The default file size which Spark uses when auto optimize is enabled is 128 MB and for Optimize it is 1GB.
  + However, these file sizes can be controlled by setting the delta table properties to much lesser size.
  + [Target File size](https://docs.databricks.com/delta/optimizations/file-mgmt.html#tune-file-size) can be set by using the delta.targetFileSize table property and any optimize or auto optimize on the table will try to achieve this file size.
  + This task can also be delegated to databricks by setting the delta table property delta.tuneFileSizesForRewrites. When this config is set, Databricks tunes the file size based on the workloads. For example, if there are lot of merge operations on the table, databricks will set the file size to much lower when compared to a 1GB file size, when optimize command is executed.

-- Table properties  
delta.targetFileSize = <size in bytes>  
delta.tuneFileSizesForRewrite = true

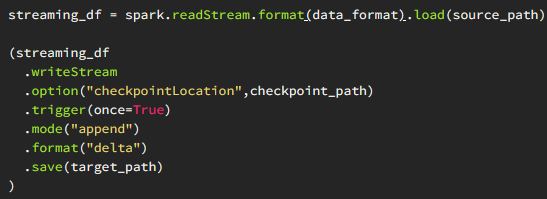
# Incremental Loads

**Problem with Full Loads:** Full loads are usually time consuming and require heavy resources, especially when the source and target tables are huge.

* + Processing entire dataset every time is expensive and increases over time and if not archived, historical data adds on to storage and performance.
  + Processing Incremental data reduces Ingress, Egress, and transaction cost on underlying storage.
  + Implementing and maintaining change data capture pipeline is complex and maintenance overhead with CDC pipelines is high as backdated data needs to be reprocessed manually.
  + Additional database/compute cost is incurred for maintaining metadata and   
    bookkeeping tables

**Guidance:** Implement incremental ETL pipelines using Structured streaming processing only incremental data across all layers.

* + Metadata management and bookkeeping for CDC is provided by Spark, out of box!
  + Incremental records are processed as micro-batches and you can leverage your existing batch codebase to incremental micro-batch based execution by just changing it to a deterministic function and changing trigger parameter to trigger=Once



* + Code can be converted to a near-real time by changing trigger to continuous
  + Structured Streaming, open source or using databricks autoloader, implements parallel listing of input directories to identify new/ overwritten/ backdated files to be processed since last execution, storing metadata information in checkpoint location
  + Check [Delta Merge Statement](#_Delta_Merge)

# Controlling Data Shuffles and Spills

* **Problem Statement:** Long running query with many transformations like joins, aggregations, window functions, distinct, etc
* **Impact**: Shuffle operations are usually costly as they case the data movement between the worker nodes. Shuffle is one of the primary reasons for Spark code not performing well.
* **Troubleshoot**

|  |  |
| --- | --- |
| **Where to check** | **What to check** |
| Unravel UI | Check Unravel for Bad joins and/or slow SQL stage |
| Spark UI | Check for Shuffle Read size and Data Spills |

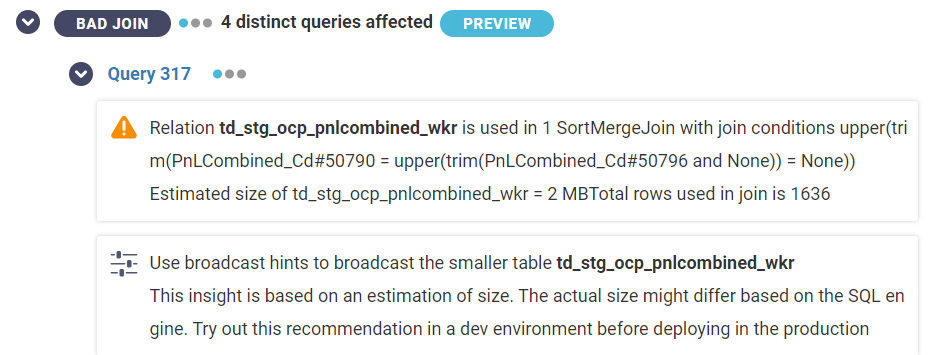


Figure 6 Unravel UI showing Bad Joins

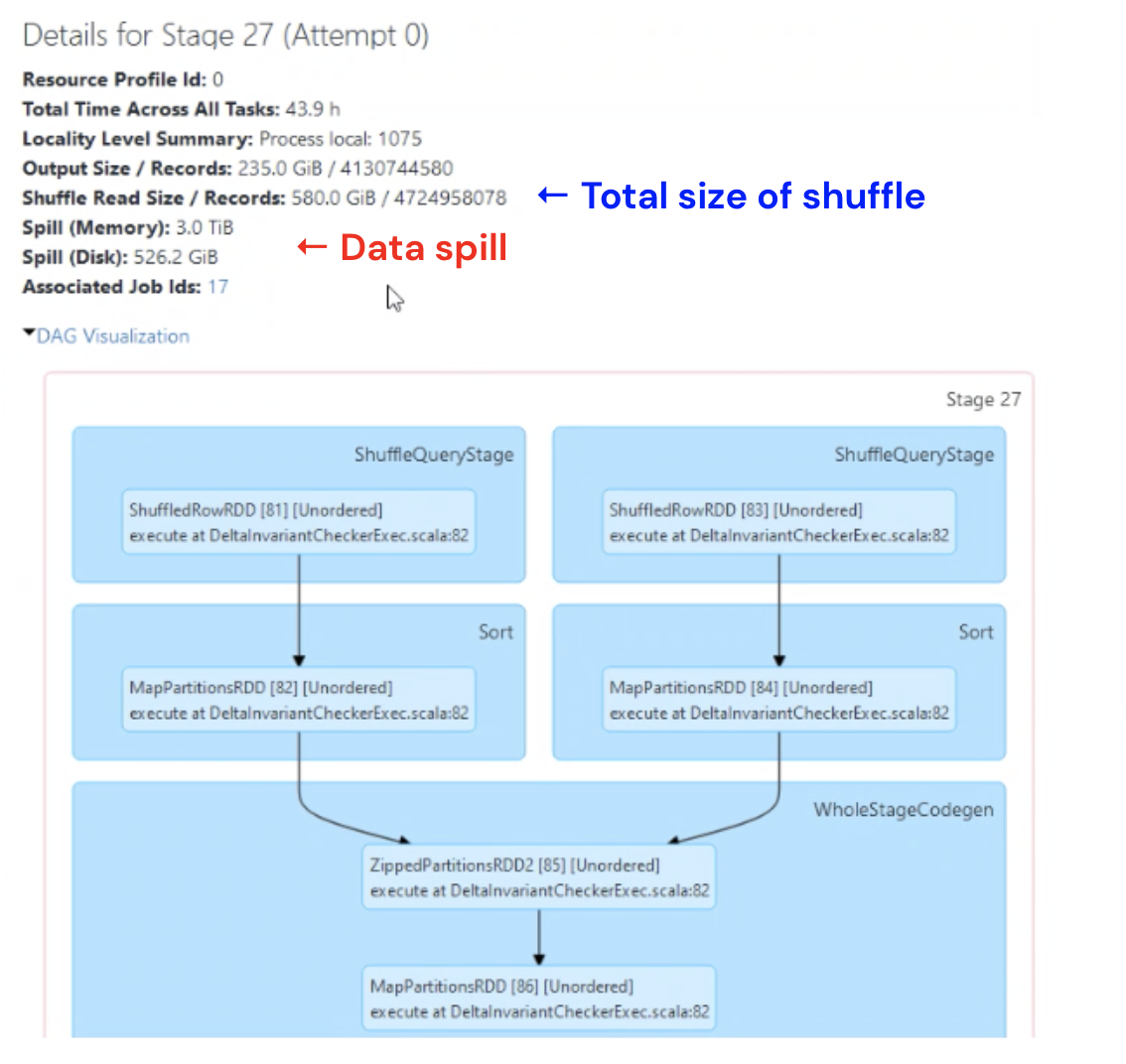


Figure 7 - Spark UI showing Shuffle Read and Data Spill

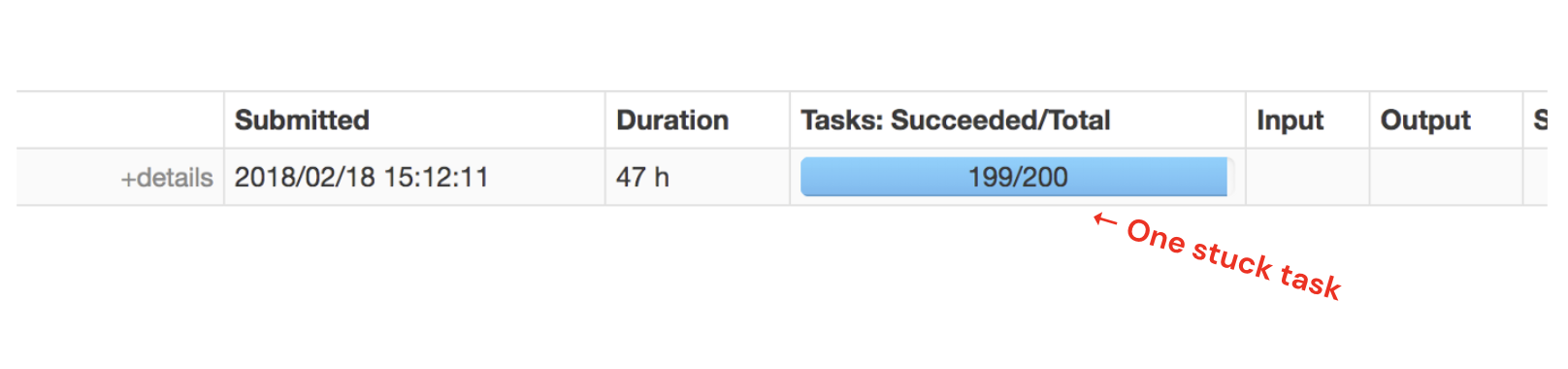
* **Guidance**

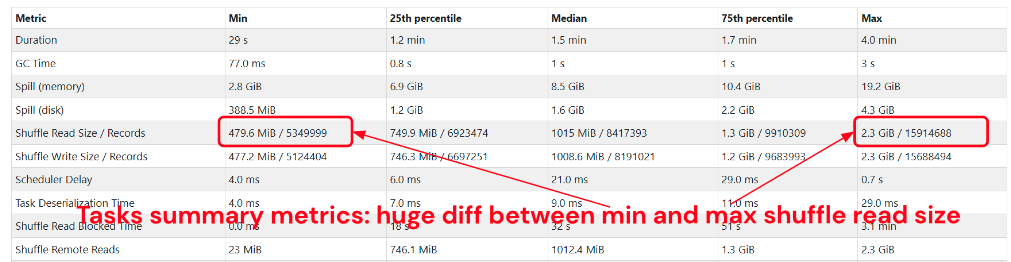
|  |  |  |
| --- | --- | --- |
| **Problem** | **Guidance** | **Additional Guidance** |
| Spark Read Stages running longer | Refer [Data Skipping](#_Data_Skipping_&) | Query performance depends on the amount of data to be processed. Limiting the amount of data will always have a positive impact on the performance |
| Too Many staging tables in a single notebook | [Use Temporary Views](#_Temporary_Views) |  |
| One of the tables in join is relatively smaller and causing Slow SQL stages | [Use broadcast join](#_Broadcast_Joins) | * In a Left Outer Join, if the right side is smaller and can fit in memory, broadcast it. * Spark by default broadcasts all datasets which are <=10MB. This value can be overridden by changing Broadcast Threshold value. |
| Sort Merge Joins causing delays | [Evaluate shuffle hash Joins](#_Shuffle_Hash_Joins) | * Shuffle hash joins may or may not give the desired performance in terms of reducing execution time |
| Improve SQL performance by having an efficient Spark SQL plan | [Use Spark Cost Based Optimizer](#_Spark_Cost_Based) | * Do not use the configs in the actual code notebook. Should be used as a part of housekeeping jobs or before the actual job triggers |
| Spark UI shows a lot of Data Spill | * [Control Data Spills by tuning shuffle Partitions](#_Controlling_Data_Spills). * [Manually Tune shuffle Partitions](#_Manually_Fine_Tune). * [Auto Tune Shuffle Partitions](#_Auto_Tuning_using) * Data Explode | * If there are too many SQLs, configure the partitions for the longest running SQL * Auto Tune shuffle partitions may not always give correct value; hence recommendation is to tune manually. * If there is a data spill even after tuning the shuffle partitions, investigate [Data Skew](#_Data_Skew_and). |
| Same dataset/table being used multiple times in a notebook/code | * [Use Caching](#_Data_Caching) | * While using databricks, it is recommended to use instances which are delta cache enabled and/or use delta cache instead of Spark cache. |

# Data Skew and How do you handle it

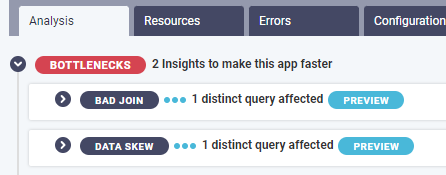
* **Problem Statement**: If one task within a stage is running longer or if Unravel UI shows a Data Skew or a Task Skew event.
* **Impact:** Any transformations on skewed data will run longer and use only a percentage of total CPU cores resulting in underutilization of cluster. In case of a fixed node cluster, the cost impact is much higher since you are billed for all the unused worker nodes as well.
* **Troubleshoot**: Below snapshots are an easy identification of Data Skew:

**SparkUI**





**Unravel UI**



* **Guidance**

|  |  |  |
| --- | --- | --- |
| **Problem** | **Guidance** | **Additional Guidance** |
| Data skew for unbalanced partition | Add another key to the partition key | * Do not partition on columns with high cardinality * Date columns are good candidates for partitioning |
| Data Skew on a join key | * Add another column in the join or filter to ensure even distribution * [Use Skew Hints](#_Skew_Hints) * [Use AQE skew optimization](#_AQE_Skew_Optimization) * [Use salting](#_Salting) | * Try the option in order as mentioned in the Guidance column. * If using Skew hints and AQE skew optimization doesn’t work, then go for Salting |
| One of the tables in join is relatively smaller | [Use broadcast join](#_Broadcast_Joins) | * In a Left Outer Join, if the right side is smaller and can fit in memory, broadcast it. * Spark by default broadcasts all datasets which are <=10MB. This value can be overridden by changing Broadcast Threshold value. |
| High Cardinality joins (usually happens with nulls and -1) | * [Use Skew Hints](#_Skew_Hints) * [Use AQE skew optimization](#_AQE_Skew_Optimization) * [Use salting](#_Salting) | * Try with Skew hints and AQE optimization first. If both don’t work, go for manual salting. |

# Filter Skewed Values

Biggest challenge to handle a skew is to identify the column and values causing the skew. Often nulls and -1 are the biggest contributor to Skews.

Though there are many methods to identify the Skew columns, you can use the below SQL to determine the cardinality of a key column and of it is causing Skew

**SELECT** <keycolumn>, count(1) as skewcount from <table-name> group by <key-column> **ORDER BY** skew count **DESC**;

If there is a huge difference between the count columns and/or uneven distribution, the key column is a candidate for Skew. Post this, go for [Skew Hints](#_Skew_Hints), [AQE Optimization](#_AQE_Skew_Optimization) or [Salting](#_Salting)

# Skew Hints

If possible [Identify](#_Filter_Skewed_Values) the table, column and the column value which is causing Data Skew. Post this, set the Skew hint explicitly with these values and instruct Spark to handle Skew.

**SELECT** /\*+ SKEW('table', 'column', (value1, value2)) \*/ \* **FROM** table

Reference*:* <https://docs.microsoft.com/en-us/azure/databricks/delta/join-performance/skew-join>

# AQE Skew Optimization

In some cases, [AQE](https://docs.databricks.com/spark/latest/spark-sql/aqe.html#dynamically-handle-skew-join) (Adaptive Query Execution) can also dynamically handle Skew using the below configurations

**SET** spark.sql.adaptive.skewJoin.enabled = **true**

By default, any partition that has at least 256 MB of data and is at least 5 times bigger in size than the average partition size will be considered as a skewed partition by AQE. This can be overridden by using the below configurations:

**-- default is 5**

**SET** spark.sql.adaptive.skewJoin.skewedPartitionFactor = <value>

**-- default is 256MB**

**SET** spark.sql.adaptive.skewJoin.skewedPartitionThresholdInBytes = <size in bytes>

# Salting

* [Salting](https://towardsdatascience.com/skewed-data-in-spark-add-salt-to-compensate-16d44404088b) is a strategy for breaking a large, skewed partition into smaller chunks by appending random integers as suffix to the skewed column value.
* An example notebook (from command 7 onwards) to solve skew using salting can be accessed [here](https://www.databricks.training/spark-ui-simulator/experiment-1596/v002-S/index.html)
* If the code is mostly written in SQL and there is a skew on null or -1 value, it can be handled by using random function as shown in below syntax

SELECT a.\*, B.\*, C.\* FROM

(SELECT \*,CASE WHEN Col1 = '-1' Then cast(-1 \* FLOOR(RAND(12345)\*1024) AS int) ELSE Col1 END as **salted\_col1**,

CASE WHEN col2 is null Then cast(-1 \* FLOOR(RAND(12345)\*1024) AS string) ELSE Col2 END as **salted\_col2** from table-name) a

LEFT JOIN B on a.**salted\_col1** = b.col1

LEFT JOIN C on a.**salted\_col2** = b.col2

# Broadcast Joins

* Broadcasting is like replication where in the broadcasted dataset is copied to each worker node and it comes in handy for most of the DWH workloads.
* While doing a left outer join between two tables, if the right side of the join is smaller, then it can be broadcasted. The driver copies the dataset to all the worker nodes and this in turn helps to minimize the data shuffles since each worker already has a copy of the data.
* By default, Spark automatically broadcasts the datasets which are smaller than 10MB in size, however this can be overridden by using the below configuration

**SET** spark.sql.autoBroadcastJoinThreshold = <size in bytes>

* Spark 3.x use [Adaptive Query execution](https://docs.microsoft.com/en-us/azure/databricks/spark/latest/spark-sql/aqe), which can also convert the sort-merge join into broadcast join when the runtime statistics is smaller than the adaptive broadcast hash join threshold (30MB by default). You can also increase this threshold by changing the below configuration

**SET** spark.databricks.adaptive.autoBroadcastJoinThreshold = <size in bytes>

**Guidance**

* Though AQE automatically Broadcasts the smaller tables, it is recommended to Broadcast these explicitly using Broadcast hints. AQE will broadcast the table during the query execution and depends on the physical plan to change sort merge joins to broadcast joins, and, hence may take a bit more time than explicitly broadcasting the smaller tables.

**SELECT** /\*+ **BROADCAST**(t2) \*/ t1.\*,t2.\* **FROM** t1 **LEFT OUTER JOIN** t2 ON..

* While using a bigger driver (>=32GB), it is safe to increase the threshold value for spark and AQE both to 200 MB.

**SET** spark.sql.autoBroadcastJoinThreshold = **209715200**

**SET** spark.databricks.adaptive.autoBroadcastJoinThreshold = **209715200**

* Do not broadcast any table bigger than 1 GB. Broadcast happens via driver and 1GB+ tables will either cause OOM on the driver or make drive unresponsive due to large GC pauses.
* While broadcasting datasets which are more than 10 MB in size, it is also recommended to increase the maxresult size config on the driver.

**SET** spark.driver.maxResultSize **8g**

# Shuffle Hash Joins over Sort Merge Joins

When Spark can’t broadcast the tables , it uses Sort Merge Joins, which are the most expensive joins. Shuffle Hash Joins have proven to be a bit faster when compared to Sort Merge Joins as these do not need to Sort the data, as in the case of Sort Merge Joins. However, using Shuffle Has Joins over Sort Merge Joins may always not guarantee the best of performance and hence, it totally depends on Spark to choose between the two depending on the use case. You can suggest Spark to choose Shuffle Hash Join over Sort Merge Join using the below configuration.

**SET** spark.sql.join.preferSortMergeJoin = **false**

*Note: This configuration doesn’t guarantee that Spark will always use Shuffle Hash join. It will still use Sort Merge Join, where it seems fit*.

# Spark Cost Based Optimizer (CBO)

* Spark SQL can use a [cost-based optimizer](https://docs.databricks.com/spark/latest/spark-sql/cbo.html) (CBO) to improve query plans. This is especially useful for queries with multiple joins. By default, CBO is off and can be turned on using the below config

**SET** spark.sql.cbo.enabled = **true**

* For CBO to work it is critical to collect table and column statistics and keep them up to date. Based on these stats, CBO chooses the most inexpensive join strategy. The following SQL command can be executed on tables to compute stats. The stats will be stored in Hive metastore.

**ANALYZE TABLE** table\_name **COMPUTE STATISTICS** FROM COLUMNS col1, col2, ... ;

* CBO can also use the stats calculated by ANALYZE TABLE command to find the optimal order in which the tables should be joined. To leverage this feature set the following configurations in addition to the above two configurations:

**SET** spark.sql.cbo.joinReorder.enabled = **true**

**SET** spark.sql.statistics.histogram.enabled = **true**

* The above steps should not be performed within the Spark code. It is recommended to run these configs as a part of house keeping jobs, which will ensure that the stats are collected and available when the query executes.

# Controlling Data Spills by tuning Shuffle Partitions

* Spark achieves parallelism by splitting the data into multiple partitions and executes transformations on these partitions parallelly.
* Shuffle partitions are partitions which are created in shuffle stages by many transformations like join, aggregates, group by, window functions, etc.
* [Data spill](https://medium.com/road-to-data-engineering/spark-performance-optimization-series-2-spill-685126e9d21f) happens when a task cannot process the data within the partition size. Data spills is extremely costly as it involves data serialization, deserialization, writing to and reading from the disk.
* Shuffle partitions value is usually controlled by spark configuration spark.sql.shuffle.partitions.
* The default value for shuffle partitions is 200. In most cases, this value is incorrect and is one of the primary reasons for data spills.
* As a rule of thumb, always set the shuffle partitions to twice the number of cores in the cluster. For example, if you have a 16 node machine where each node has 16 cores, the total number of cores is 16\*16, which is 256. Set the shuffle partition to 256\*2 = 512 to start with

# Manually Configure Shuffle Partitions

* Set the shuffle partitions to twice the number of cores in the cluster. Run the Spark SQL to be tuned using a small interactive cluster(preferably fixed node).
* For each stage which gets executed, open the Spark UI to check if there is a Shuffle Read and Data Spill. Usually the stages which perform transformations like Joins, Group By, Window functions, Unions, etc will show Shuffle Read and Data Spills.

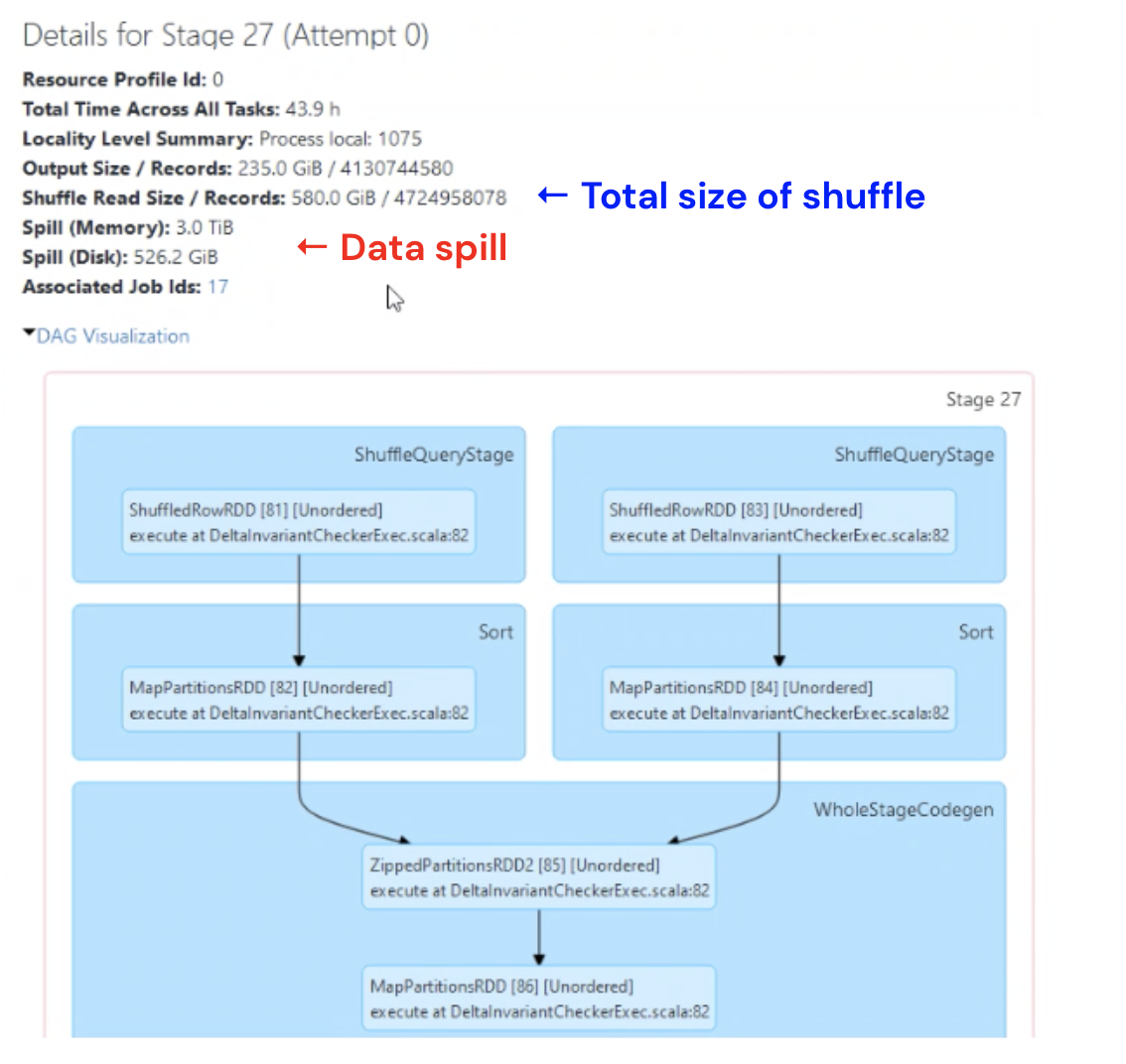


Figure 8 - Retrieve Shuffle Read and Data Spill

* Spark performs extremely well when each partition is allocated 128 MB memory. The objective of configuring shuffle partitions is always to reach a value near 128 MB. Below figure shows how the metrics look when shuffle partitions are inappropriately set

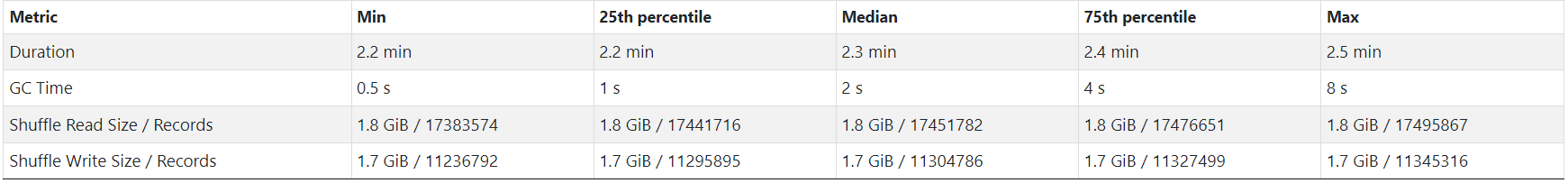


Figure 9Inappropriate Shuffle partition resulting in more memory per partition than it can handle

* The below formula will help to calculate the shuffle partitions appropriately:

Assuming,

C= total number of worker cores in cluster

M= memory needed in each partition = 128

S=Shuffle data size (in MB)

Factor = ceiling(S/C/M)

Shuffle Partition Number = C\*Factor

For Example

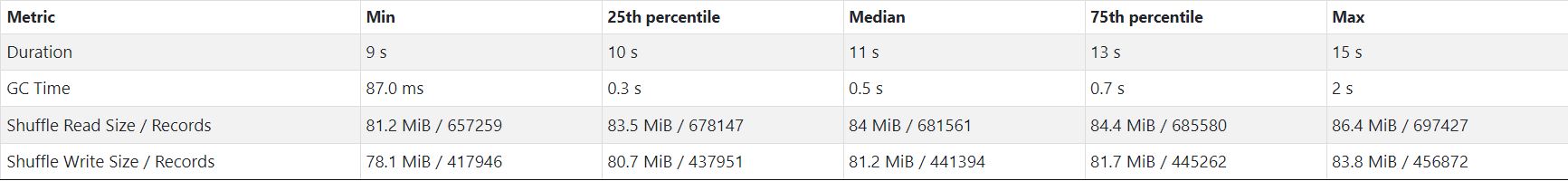
C = 256 (16 worker node cluster with 16 cores each)

M= 128 (Memory needed in each partition)

S = 120 GB = 122880 MB (Shuffle Size in MB)

Factor = Ceiling(122880/128/256) = 4

Shuffle Partition Number = 256\*4=1024



**Points to Remember**

* If there are multiple SQLs in a notebook, same shuffle partition value may or may not work well for all. In such cases, either tune shuffle partition individually before each Spark SQL or set it using the greatest number of data shuffle value, for example if query 1 has Data shuffle as 100 GB, query 2 has 200 GB and query 3 300GB, use 300GB shuffle as the basis for tuning the shuffle partitions and set it only once in the beginning of notebook
* If there is a data skew in query, then fine tuning the shuffle partitions will not help in controlling data spills. Recommendation is to fix [Data Skew](#_Data_Skew_and).

# Auto Tuning using Adaptive Query Execution

* 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:

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

* AOS may not be able to estimate the correct number of shuffle partitions in some circumstances where source tables have an unusually high compression ratio (30x to 40x). To counter this effect, reduce the value of the per partition size used to determine the initial shuffle partition number (default 128MB). 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:

**set** spark.databricks.adaptive.autoOptimizeShuffle.preshufflePartitionSizeInBytes = **16777216**

**Points to Remember**

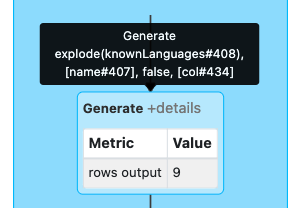
* Even after using AOS to set the shuffle partitions and there is no data skew but still there are data spills, it’s best to configure the shuffle partitions manually.

# Data Explode

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. The query execution is significantly slowed because of this. The following are some of the most prevalent transformations that can result in a data explosion:

**Explode function with Structured files and arrays**

* While working with structured files like JSON, Parquet, Delta, and XML, we often get data in collections like arrays, lists, and maps. In such cases, explode() functions are useful to convert collection columns to rows to process in Spark effectively.
* The [explode](https://sparkbyexamples.com/pyspark/pyspark-explode-array-and-map-columns-to-rows/) operation can significantly increase the data volume. The explode operation is represented by the *Generate* node in Spark UI as shown below:



* While reading data from files, Spark reads approximately 128 MB per task per core. Due to exploding the data size increases in-proportionally and a single CPU core may not have enough memory to fit it into the partition. As a result, data gets spilled to disk

Recommendation:

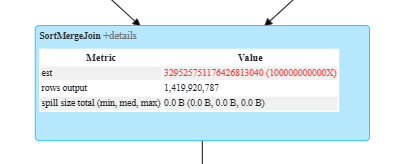
* Decrease the size of input partition i.e. spark.sql.files.maxPartitionBytes (default 128MB) to create smaller input partitions in order to counter the effect of explode() function. Instead of 128MB, you can choose a much smaller partition size like 16MB or 32MB for example

SET spark.sql.files.maxPartitionBytes =<size in bytes>

* Call the [repartition()](https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.sql.DataFrame.repartition.html) function right after the read statement to increase the total number of partitions. This will allow you to reduce the size of each partition

**Data Explosion due to Joins**

In some scenarios, you will notice that there is a data spill even when the shuffle partitions are perfectly tuned and there is not data skew. These corner cases are often called as row explosion and are usually a result of a cartesian or cross joins



**Recommendations:**

* Simple solution would be to increase the number of shuffle partitions. In such case, since the number of shuffle partition increases, the size of each partition decreases to a value much lower than 128 MB and each partition processes the data faster and avoids spilling to a significant level.
* Increase the number of nodes in the cluster or choose a node with higher memory.

# Data Caching

When a table and/or a dataset is being reused within the same notebook or a code, we can take advantage of caching to cache the data and reduce the overall IO of the query.

[Delta cache and Spark cache](https://docs.databricks.com/delta/optimizations/delta-cache.html#delta-and-apache-spark-caching) are two types of caching methods which we can use to make the workloads faster.

* **Delta Cache**

[Delta Cache](https://docs.microsoft.com/en-us/azure/databricks/delta/optimizations/delta-cache) accelerates the data reads by creating copies or remote files in node’s local storage (SSD drives) using a fast intermediate data format.

* While using databricks, there are instances which are accelerated with Delta Cache (example E\*ds\_v4 instances and Lsv2 instances).
* Delta cache can also be enabled on the workers of instances with SSD drives (example Fsv2 series). To explicitly enable the delta cache, set the following configuration in the code:

**SET** spark.databricks.io.cache.enabled = **true**

* **Spark Cache**

Using cache() and persist() methods, Spark provides an optimization mechanism to [cache](https://medium.com/pecan-ai/caching-spark-dataframe-how-when-79a8c13254c0) the intermediate computation of a spark dataframe. These can then be reused in subsequent actions/ similarly, a table can be cached using the [cache table](https://spark.apache.org/docs/3.0.0-preview/sql-ref-syntax-aux-cache-cache-table.html) command

There are different cache modes to choose from when you want to store the cached data (in memory, in disk or in both memory and disk)

# Temporary Views

**Problem Statement:** A complex Spark code where a big query/DAG is split into multiple smaller queries.

**Approach Followed:** For each smaller query, a temporary staging table/dataset is created

**Advantages:** Readability, Maintainability, Reusability

**Disadvantages:** Writing to and reading from storage is time consuming

**Recommendation:** Use Temporary or Global Views. It expedites the query run time and still keeps the code readable and maintainable.

**Python Syntax**

stgDF = spark.read.format(“Delta/parquet/orc”).load(“path”)

stgDF.createOrReplaceTempView(“ViewName”)

stgDF.createOrReplaceGlobalView(“ViewName”)

**SQL Syntax:**

CREATE OR REPLACE TEMP VIEW *ViewName*  AS [Select \* from…..]

# Data Skipping & Pruning

Problem Statement: Read stages within a Spark query running longer.

**Recommendations:**

* **Column Pruning**

Choose only the required columns from reference dataset which are required in the target dataset. Don’t take the whole dataset in the join, especially if the size o the reference dataset runs in hundreds of GB. The smaller the dataset, the easier it would be to Broadcast it.

* **Predicate Pushdown**

Predicate Pushdown involves pushing down the filter to Source. This helps in optimizing the query performance as the data is filtered at a very low level than reading the entire dataset after it’s been loaded to Spark’s memory.

--SQL—

**SELECT** col1, col2, …,coln from table **where** col1 =<value>

--PySpark—

**readDF** = spark.read.format(‘delta’).load(‘path’).select(“col1”,”col2”).filter(col(“col1”) = <value>)

While using any join operations, filters should be applied before the join statement and immediately after read statement

* **Partition Pruning**

Partition pruning further optimizes the performance. If the table or dataset is appropriately partitioned, then filter can be applied on the partition columns in addition to the predicate pushdown predicates.

* **Delta Data Skipping**

Delta data skipping collects the stats (min, max, etc) for the first 32 columns of each dataset when the data is written into a delta dataset/table. Databricks uses this information at query run time to skip the unnecessary files to speed up the queries

To collect stats on more than first 32 columns, set the following:

delta.dataskippingNumIndexedCols = <value>

Collecting statistics on long strings is an expensive operation and should be avoided.

# Delta Merge

Merge statement comes in handy for incremental loads wherein you can perform an upsert using a single statement instead of writing different inert and update statements.

To achieve the incremental load, Delta MERGE is very essential. Delta merge can also be used to create SCD type2 tables and change data capture (CDC) use cases.

**Performance issues with Merge Statement:**

* Not enough condition in “ON” clause results in writing more data than required
* Writing too much data with merge can spoil the distribution done by Zorder. As a result, OPTIMIZE ZORDER BY has to be performed after each merge.
* Merge statements on large tables and/or tables which undergo frequent merge run longer

**Merge Recommendations:**

* Limit the file size in large tables or the tables which have frequent merge to be in the size of 16 MB to 128 MB. This will reduce the amount of data which will be sent to driver when the join happens.
* Use partition and/or Zorder columns as join keys in merge statement. Usually these should be the first predicates for faster filtering.

Databricks Low shuffle Merge algorithm:

* 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](https://docs.microsoft.com/en-us/azure/databricks/delta/optimizations/low-shuffle-merge)” 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. For other versions, it can be enabled by using the config spark.databricks.delta.merge.enableLowShuffle to true

# Anti-Patterns

* Do not overprovision clusters - neither the instance type nor the number of worker nodes. Fine-tune these for each use case and workload with the help of Ganglia and Spark UI metrics.
* Do not use collect in Dev or Prod.
  + With collect, the entire DataFrame distributed across the multiple Worker nodes is materialized to the Driver node, which will most likely OOM as the Driver node will almost always have less memory.
* Do not ever use .toPandas for the exact same reason as above.
* Also for the precise reason that Pandas DataFrames are not distributed, Pandas should not be used for processing anything more than a GB of data.
* Do not use UDFs with Pyspark because Spark can’t optimize the code inside UDFs.
  + If UDFs are the only way to go for your specific use case, leverage [Pandas UDFs](https://databricks.com/blog/2020/05/20/new-pandas-udfs-and-python-type-hints-in-the-upcoming-release-of-apache-spark-3-0.html).
* Remove all show, display, and print statements from the code when transitioning to Prod.
* Never leave spark.sql.shuffle.partitions value to default which is 200. Set this value to “auto” or either fine-tune this value for your workload or set it to 2-3x the number of total worker cores.
* Smaller tables should not be partitioned. Partitioning is only advisable for tables with roughly 1GB of data per partition or more.
* The [REFRESH](https://docs.cloudera.com/runtime/7.2.10/impala-sql-reference/topics/impala-refresh.html) command or [table.purge](https://docs.cloudera.com/HDPDocuments/HDP3/HDP-3.1.4/using-hiveql/content/hive_drop_external_table_data.html) table property should not be used anywhere in the code. These are impala/hive commands, ain’t required for Delta tables.
* Do not run OPTIMIZE, ZORDER, and VACUUM as part of your job. They should be run as separate jobs on a dedicated job cluster.
* Do not materialize the intermediate working tables as delta tables but instead store them as temporary views to leverage Spark lazy evaluation and also to avoid unnecessary network and disk I/O.
* Do not overwrite & insert the whole table on daily basis but instead adopt an incremental load approach wherever possible.