Top 5 Tuning Tips for Delta Lake and Spark 3.0 on Databricks

DELTA LAKE

databricks

Joe Widen, Solutions Architect
Chris Hoshino-Fish, Solutions Architect
Denny Lee, Developer Advocate

databricks

Top 5 Tuning Tips

1. Use the latest version of DBR
2. Pick the best join for your workload
3. Use Spark 3.0 with AQE
4. Partitions, File Sizes, Zorder, Stats and Merges
5. Pick the right machines

Use the latest version of DBR

Keeping up with the Databricks Runtime

- The latest Databricks runtime is almost always faster than the one before it.

- We constantly work with some of the largest users of Apache Spark and Delta Lake to make it easier, faster, and better.
    - DBR 7.3 significantly reduced delta metadata overhead time
    - DBR 7.2 added cloning, upgraded cloud connectors, upgraded AQE
    - DBR 7.1 added convert to delta, MERGE INTO performance improvements
    - DBR 7.0 upgraded to Spark 3.0, Delta Lake Auto Loader and COPY INTO and Dynamic Subquery Reuse

Picking the best join strategy

Physical Join Types - A review in performance

BroadcastHashJoin

BroadcastNestedLoopJoin

ShuffledHashJoin: One side is smaller (3x or more) and a partition of it can fit in memory

(off by default, enable by `spark.sql.join.preferSortMergeJoin = false`)

SortMergeJoin: Scalable - just works

Cartesian: Try not to do this

Physical Join Types - A review in performance

- Broadcast Hash Join / Nested Loop

  SELECT /*+ BROADCAST(a) */ id FROM a JOIN b ON a.key = b.key

Requires one side to be small. No shuffle, no sort, very fast.

- Shuffle Hash Join

  SELECT /*+ SHUFFLE_HASH(a, b) */ id FROM a JOIN b ON a.key = b.key

Needs to shuffle data but no sort. Can handle large tables, but will OOM too if data is skewed. One side is smaller (3x or more) and a partition of it can fit in memory
(enable by `spark.sql.join.preferSortMergeJoin = false`)

- Sort-Merge Join

  SELECT /*+ MERGE(a, b) */ id FROM a JOIN b ON a.key = b.key

Robust. Can handle any data size. Needs to shuffle and sort data, slower in most cases when the table size is small.

- Shuffle Nested Loop Join (Cartesian)

  SELECT /*+ SHUFFLE_REPLICATE_NL(a, b) */ id FROM a JOIN b

Does not require join keys as it is a cartesian product of the tables. Avoid doing this if you can

Lets use hints!

SELECT /*+ BROADCAST(t1) */ * from t1... - Broadcast hint

SELECT /*+ SHUFFLE_HASH(t1) */* from t1... - Shuffle Hash Join

SELECT /*+ SHUFFLE_MERGE(t1) */* from t1... - Sort Merge Join

Use multiple hints?

SELECT /*+ BROADCAST(t1), SHUFFLE_HASH(t2) */* - Broadcast t1, shuffle hash t2

Help Catalyst choose the right Joins

At the cluster level, set these configs:

```
spark.sql.autoBroadcastJoinThreshold 100*1024*1024
spark.sql.join.preferSortMergeJoin false
```

Why do I need the slide before?

You know your data best. For example, you're joining two tables on a partitioned column and a couple others, a sort merge join would make more sense then a full shuffle

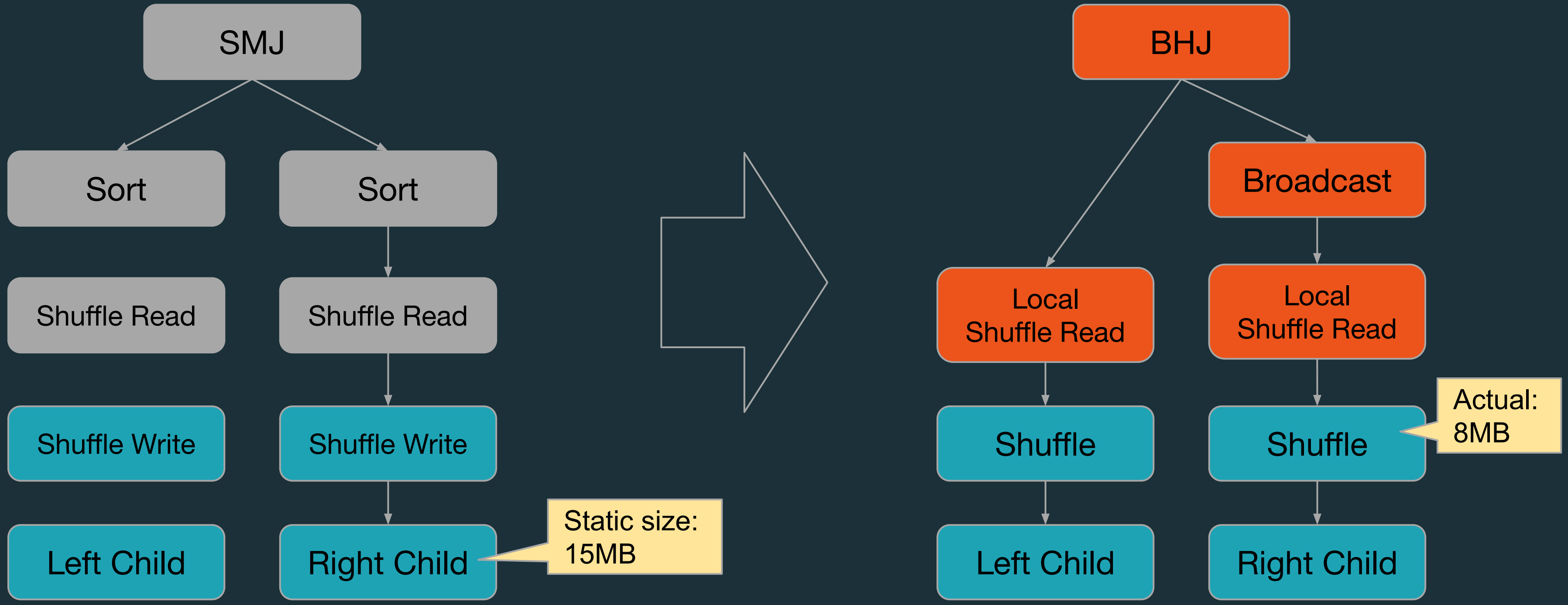
Use Apache Spark 3.0 and AQE

databricks

Spark 3.0 Adaptive Query Execution

- What is it? [#SPARK-23128] A new approach to do adaptive execution in Spark SQL - ASF JIRA
- What does it do?
  - Adapts a query plan automatically at runtime based on accurate metrics (no hint needed)
  - Capabilities in release
    - Sort Merge Join (SMJ) ⟶ Broadcast Hash Join (BHJ)
    - Coalesce shuffle partitions
    - Skew Join

Spark 3.0 Adaptive Query Execution



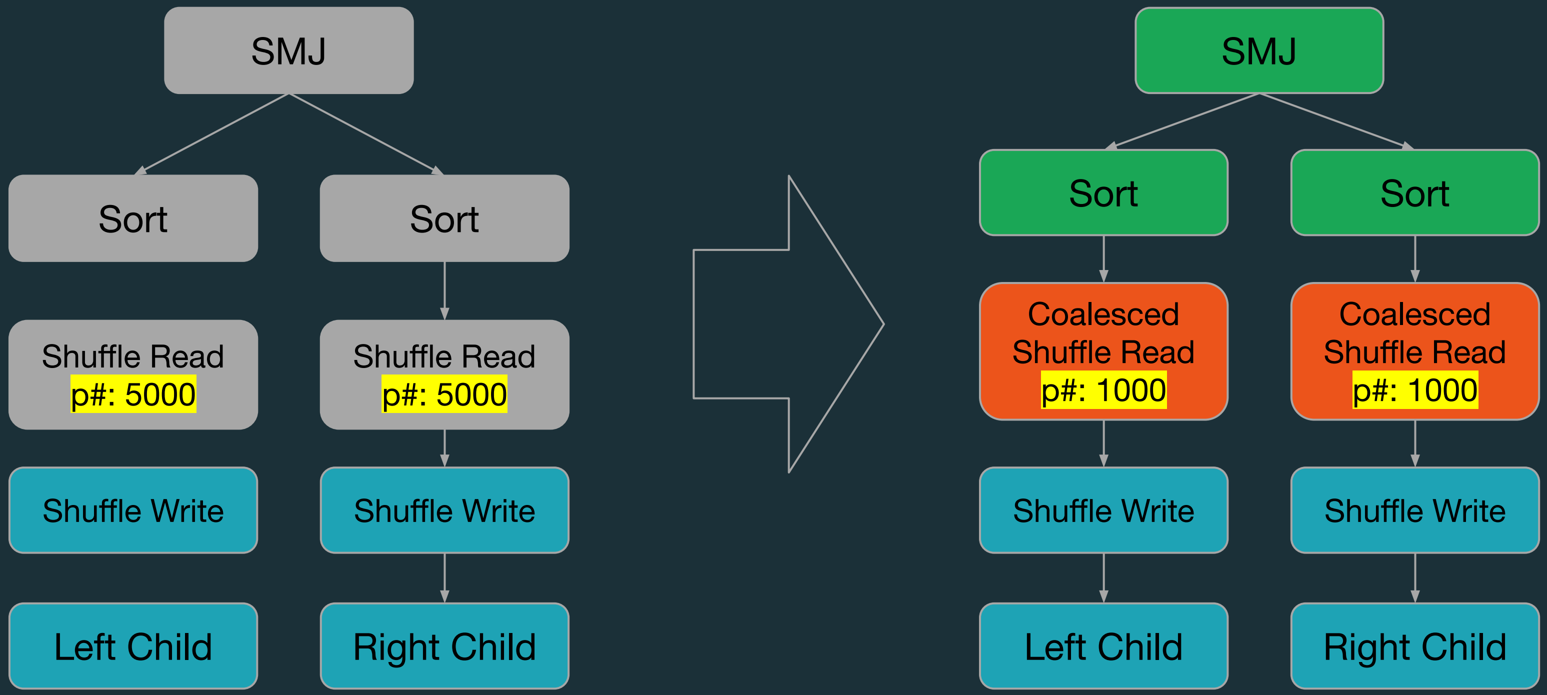
Static size: 15MB

Actual: 8MB

databricks

⬜ Not Started 🟦 Done 🟧 New Plan - Changed 🟩 New Plan - Unchanged

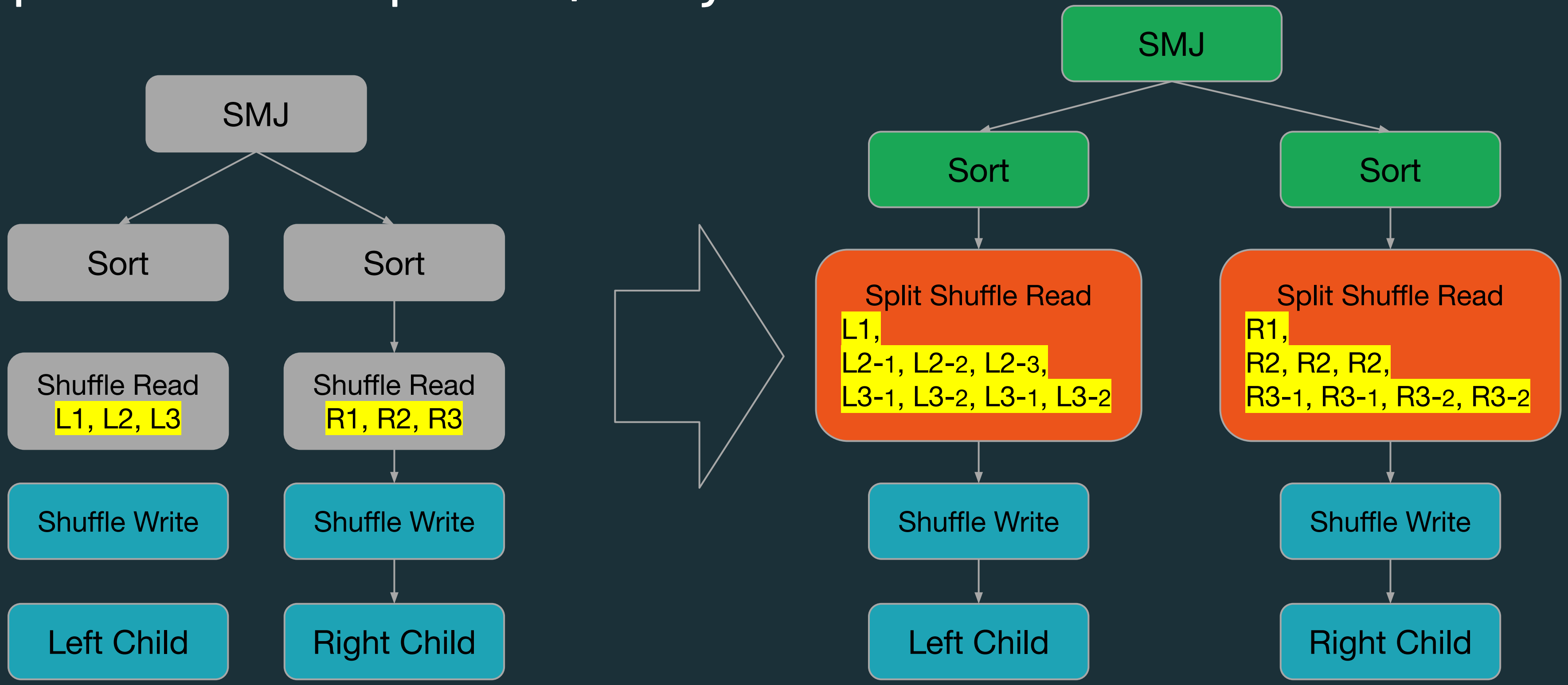
Spark 3.0 Adaptive Query Execution



⬜ Not Started 🟦 Done 🟥 New Plan - Changed 🟩 New Plan - Unchanged

databricks

Spark 3.0 Adaptive Query Execution



SMJ

Sort

Sort

Shuffle Read
L1, L2, L3

Shuffle Read
R1, R2, R3

Shuffle Write

Shuffle Write

Left Child

Right Child

SMJ

Sort

Sort

Split Shuffle Read
L1,
L2-1, L2-2, L2-3,
L3-1, L3-2, L3-1, L3-2

Split Shuffle Read
R1,
R2, R2, R2,
R3-1, R3-1, R3-2, R3-2

Shuffle Write

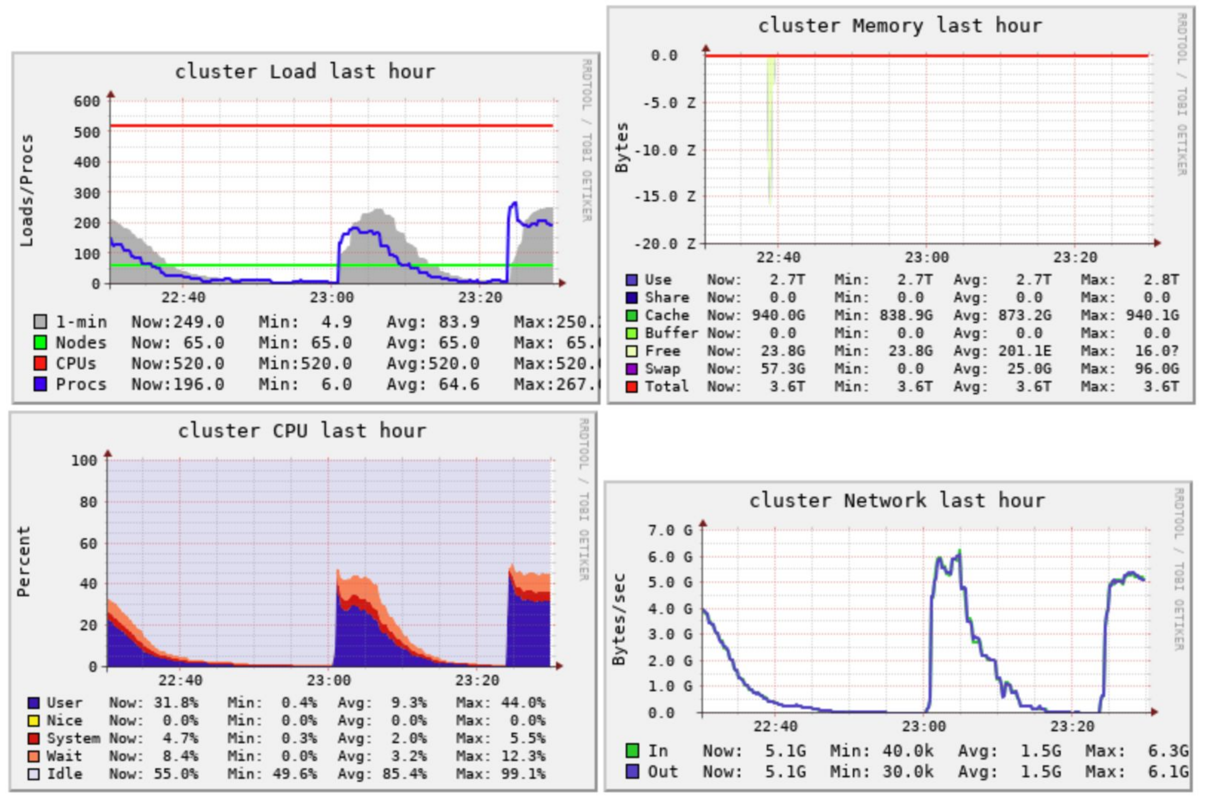
Shuffle Write

Left Child

Right Child

▨ Not Started 🟦 Done 🟧 New Plan - Changed 🟩 New Plan - Unchanged

Does it work? This ran in 6h 47m



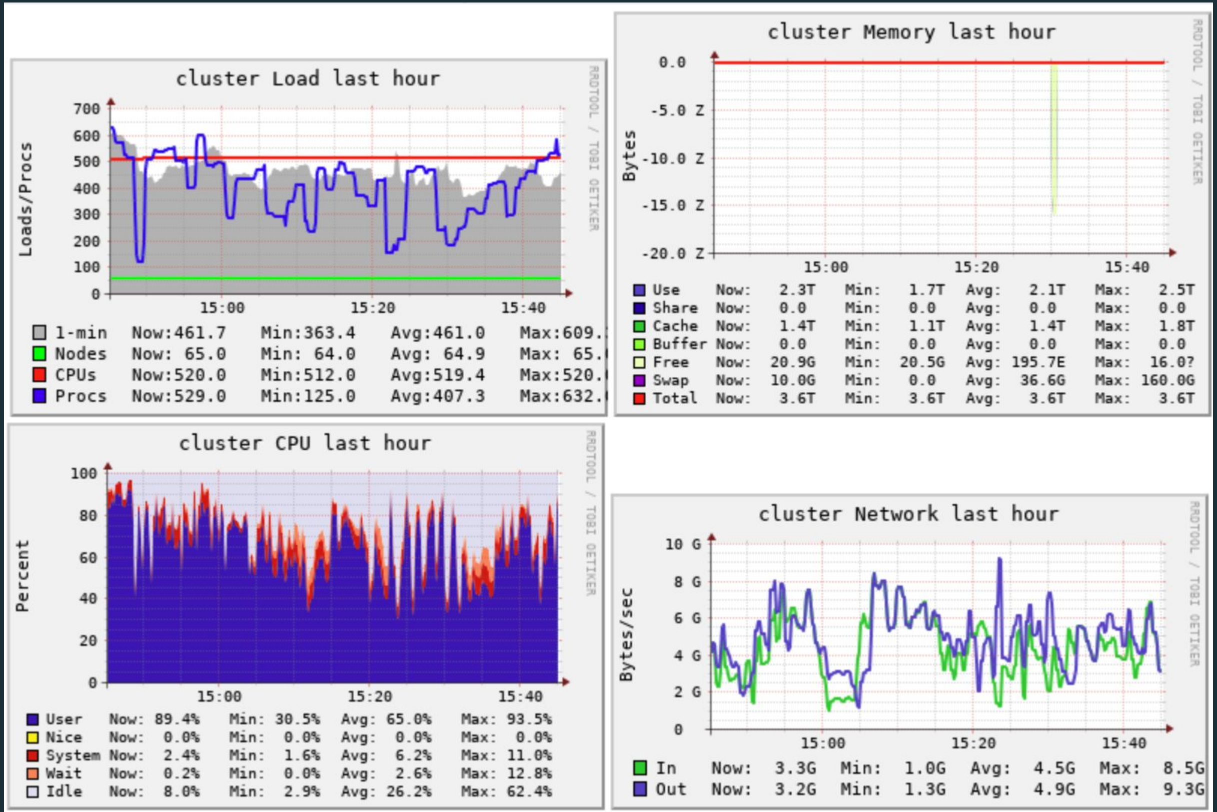
```
□ 1-min   Now:249.0   Min:  4.9   Avg: 83.9   Max:250.
🟩 Nodes   Now: 65.0   Min: 65.0   Avg: 65.0   Max: 65.
🟥 CPUs    Now:520.0   Min:520.0   Avg:520.0   Max:520.
■ Procs   Now:196.0   Min:  6.0   Avg: 64.6   Max:267.
```

```
■ Use    Now:   2.7T   Min:   2.7T   Avg:   2.7T   Max:   2.8T
■ Share  Now:   0.0    Min:   0.0    Avg:   0.0    Max:   0.0
🟩 Cache  Now: 940.0G   Min: 838.9G   Avg: 873.2G   Max: 940.1G
🟩 Buffer Now:   0.0    Min:   0.0    Avg:   0.0    Max:   0.0
□ Free   Now:  23.8G   Min:  23.8G   Avg: 201.1E   Max:  16.0?
■ Swap   Now:  57.3G   Min:   0.0    Avg:  25.0G   Max:  96.0G
🟥 Total  Now:   3.6T   Min:   3.6T   Avg:   3.6T   Max:   3.6T
```

```
■ User   Now: 31.8%   Min:  0.4%   Avg:  9.3%   Max: 44.0%
□ Nice   Now:  0.0%   Min:  0.0%   Avg:  0.0%   Max:  0.0%
🟥 System Now:  4.7%   Min:  0.3%   Avg:  2.0%   Max:  5.5%
■ Wait   Now:  8.4%   Min:  0.0%   Avg:  3.2%   Max: 12.3%
□ Idle   Now: 55.0%   Min: 49.6%   Avg: 85.4%   Max: 99.1%
```

```
🟩 In    Now:   5.1G   Min: 40.0k   Avg:   1.5G   Max:   6.3G
■ Out   Now:   5.1G   Min: 30.0k   Avg:   1.5G   Max:   6.1G
```

You tell me. This ran in 2h 28m



```
□ 1-min   Now:461.7   Min:363.4   Avg:461.0   Max:609.
🟩 Nodes   Now: 65.0   Min: 64.0   Avg: 64.9   Max: 65.
🟥 CPUs    Now:520.0   Min:512.0   Avg:519.4   Max:520.
🟦 Procs   Now:529.0   Min:125.0   Avg:407.3   Max:632.
```

```
🟦 Use    Now:   2.3T   Min:   1.7T   Avg:   2.1T   Max:   2.5T
🟦 Share  Now:   0.0    Min:   0.0    Avg:   0.0    Max:   0.0
🟩 Cache  Now:   1.4T   Min:   1.1T   Avg:   1.4T   Max:   1.8T
🟩 Buffer Now:   0.0    Min:   0.0    Avg:   0.0    Max:   0.0
□ Free   Now:  20.9G   Min:  20.5G   Avg: 195.7E   Max:  16.0?
🟪 Swap   Now:  10.0G   Min:   0.0    Avg:  36.6G   Max: 160.0G
🟥 Total  Now:   3.6T   Min:   3.6T   Avg:   3.6T   Max:   3.6T
```

```
🟦 User   Now: 89.4%   Min: 30.5%   Avg: 65.0%   Max: 93.5%
🟨 Nice   Now:  0.0%   Min:  0.0%   Avg:  0.0%   Max:  0.0%
🟥 System Now:  2.4%   Min:  1.6%   Avg:  6.2%   Max: 11.0%
🟧 Wait   Now:  0.2%   Min:  0.0%   Avg:  2.6%   Max: 12.8%
□ Idle   Now:  8.0%   Min:  2.9%   Avg: 26.2%   Max: 62.4%
```

```
🟩 In   Now: 3.3G   Min: 1.0G   Avg: 4.5G   Max: 8.5G
🟦 Out  Now: 3.2G   Min: 1.3G   Avg: 4.9G   Max: 9.3G
```

Partitions, Files, Zorder, Stats and Merge

Partition Pruning

```
/path/to/deltalake_table/
                  part=1/part_00001.parquet
                  part=1/part_00002.parquet
                  part=2/part_00001.parquet
                  part=2/part_00001.parquet
```

Partition Pruning

```
/path/to/deltalake_table/
                  part=1/part_00001.parquet
                  part=1/part_00002.parquet
                  part=2/part_00001.parquet
                  part=2/part_00001.parquet
```

Select * from deltalake_table where part = 2

Data Skipping

Simple, well-known I/O pruning technique

- Track file-level stats like min & max
- Leverage them to avoid scanning irrelevant files

```
SELECT input_file_name() as “file_name”,
       min(col) AS “col_min”,
       max(col) AS “col_max”
FROM table
GROUP BY input_file_name()
```

file_name	col_min	col_max
1.parquet	6	8
2.parquet	3	10
3.parquet	1	4

Data Skipping

Simple, well-known I/O pruning technique

- Track file-level stats like min & max
- Leverage them to avoid scanning irrelevant files

```
SELECT * FROM TABLE WHERE col < 5

                  ⇩

SELECT file_name FROM index
WHERE col_min < 5
```

file_name	col_min	col_max
1.parquet	6	8
2.parquet	3	10
3.parquet	1	4

Data Skipping

Simple, well-known I/O pruning technique

- Track file-level stats like min & max
- Leverage them to avoid scanning irrelevant files

```
SELECT * FROM TABLE WHERE col < 5

                 ⇩

SELECT file_name FROM index
WHERE col_min < 5 AND col_max >= 5
```

file_name	col_min	col_max
1.parquet	6	8
2.parquet	3	10
3.parquet	1	4

Z-Ordering

```
optimize table zorder by col
```

Old Layout

file_name	col_min	col_max
1.parquet	6	8
2.parquet	3	10
3.parquet	1	4

New Layout

file_name	col_min	col_max
1.parquet	1	3
2.parquet	4	7
3.parquet	8	10

Z-Ordering

```
Select * from table where col = 7
```

Old Layout

file_name	col_min	col_max
1.parquet	6	8
2.parquet	3	10
3.parquet	1	4

New Layout

file_name	col_min	col_max
1.parquet	1	3
2.parquet	4	7
3.parquet	8	10

Z-Ordering Tips

- Z-Ordering is pretty effective on up to 3-4 columns
  - New Hilbert based Z-Ordering will allow for 5-6 in the future
- Z-Ordering creates 1 GB files (by default)
  - If your partitions are less than 1 GB, default Z-Ordering won't help
  - You can change the `zorder` file size
    - 16MB -> 128MB seem to be ideal, choose based on your average partition size
    - The file size will be auto-tuned in the not so distant future
    - Delta doesn’t care about the number of partitions, only the number of files
- Z-Order on columns commonly used in filters/where predicates

Databricks Delta Lake and Stats

- Databricks Delta Lake collects stats about the first N columns
  - `dataSkippingNumIndexedCols = 32`

- These stats are used in queries
  - Metadata only queries: `select max(col) from table`
    - Queries just the Delta Log, doesn't need to look at the files if `col` has stats
  - Allows us to skip files
    - Partition Pruning, **Data Filters**, Pushed Filters apply in that order
  - TimeStamp and String types aren't always very useful
    - Precision/Truncation prevent exact matches, have to fall back to files sometimes

- Avoid collecting stats on long strings
  - Put them outside first 32 columns or collect stats on fewer columns
    - `alter table change column col after col32`
    - `set spark.databricks.delta.properties.defaults.dataSkippingNumIndexedCols = 3`

Easy Patterns you can follow

- Don’t collect stats on your Bronze tables

  -
    ```
    alter table bronze set
    TBLPROPERTIES(delta.dataSkippingNumIndexedCols = 0)
    ```

- Only collect stats on useful columns in Silver and Gold level tables
  - Join keys
  - Common WHERE clauses
  - INSERT/UPDATE/DELETE/MERGE predicates
  - Don’t collect stats on long strings
  - If using lots of *equals* filters on TimeStamps, stats are useful, not so much for *greater* or *less than*
  - Z-Order on join keys, and commonly used columns in where clauses

MERGE Basics

```
MERGE INTO my_table d USING my_view s ON condition
WHEN MATCHED AND d.status = 'CLOSED' THEN DELETE
WHEN MATCHED THEN UPDATE SET d.status = 'CLOSED'
WHEN NOT MATCHED THEN INSERT *
```

- Leverage data skipping (partition + stats) and figure out which files may match *predicate*
- Perform inner join and collect input_file_name for all rows that match *condition.* Ensure 1-1 mapping
- Rewrite files found above by updating the rows that match the *conditions* using a full outer join

FYI on Merge Functionality

- Merges rewrite EVERY file that have a matching record in the ON clause, regardless of what is in the WHEN [NOT] MATCHED

```
MERGE INTO myTable
USING myChanges
ON table.id = myChanges.id
WHEN MATCHED AND 1=2
THEN UPDATE *
```

`WHEN MATCHED AND 1=2` is never true, so it shouldn't do anything, merge still scans the entire table myTable, and rewrites every file with matching id

Optimizing Merges - Reduce the scan

- Addition additional information into the ON clause will reduce the amount of data to be scanned

```
MERGE INTO myTable
USING myChanges
ON myTable.id = myChanges.id AND myTable.date > current_day() -
interval 7 days
THEN UPDATE *
```

This leverages partition pruning on the destination table, resulting in smaller data to be scanned during the join of `table.id = myChanges.id`

Optimizing Merges - Rewrite less data

- Use smaller files for merges that can rewrite a large number of files
  - `set spark.databricks.delta.optimize.maxFileSize = 32mb`
  - File sizes between 16mb and 128mb are good

- Z-Order on the columns in your ON predicate

- Use the Hilbert Curve Z-Order (in beta in DBR 7.3, see Tip #1!)

Optimizing Merges - Write fewer files

- Use optimized writes (aka auto optimize) to prevent writing a lot of small files after the merge

```
alter table tablename set tblproperties
(delta.autoOptimize.optimizeWrite = true)
```

Pick good instance types

What happens when you only change the instance type?

Example:

```
Select * from table where partCol = this and otherCol = 1
```

- Dataset was in Delta Lake, 2.2TB, 3k files
- Cluster configs:
  - 6 nodes, 16 cores/node, 32GB ram
  - 6 nodes, 16 cores/node, 128GB ram
  - 3 nodes, 32 cores/node, 256GB ram

What happens when you only change the instance type?

DBR Version	Cache	VM	Duration
6.6	cold	16 core 32 GB RAM	1.82 minutes
6.6	cold	16 core, 128 GB RAM	2.29 minutes
6.6	warm	16 core, 128 GB RAM	1.95 minutes
5.5 LTS	cold	16 core, 128 GB RAM	2.46 minutes
5.5 LTS	warm	16 core, 128 GB RAM	1.92 minutes
6.6	cold	32 cores, 256 GB RAM	4.18 minutes
6.6	warm	32 cores, 256GB RAM	3.72 minutes

Instance Types

- Rules of thumb
  - Use 32 to 128GB machines
  - Cores are cheap, you and your employee's time aren't
  - Not all processors/cores are created equal
- Doing ETL?
  - Use a compute optimized instance
- Running interactive queries?
  - Use an instance with a NVMe SSD
- Building machine learning models?
  - Use an instance with a NVMe SSD

Thank you for attending
For more information go to delta.io and join the Delta Lake slack channels

databricks