databricks

Data + AI Professional Workshop Series

Data Engineering Optimization Best Practices

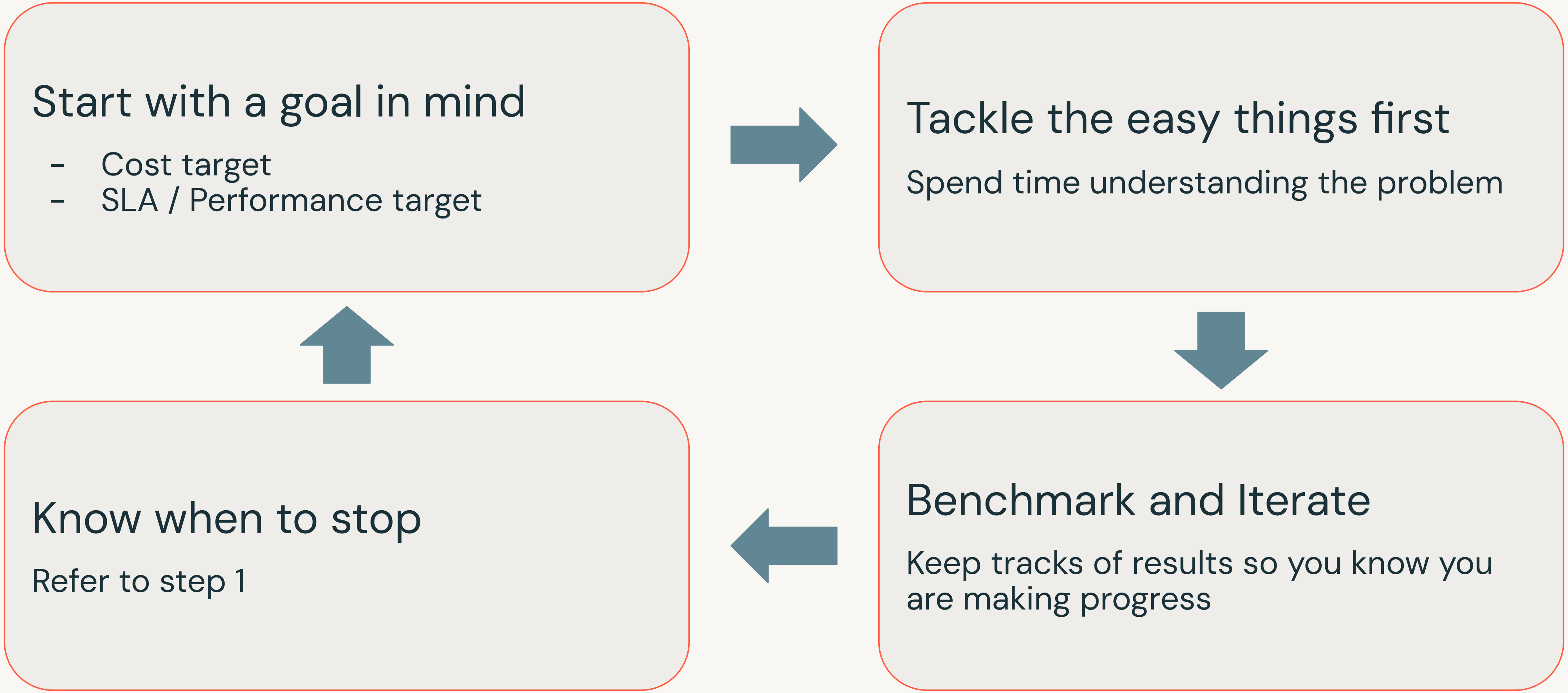
Irfan Elahi – Specialist Solutions Architect

Housekeeping

- This presentation will be recorded and we will share these materials after the session.
- There are no hands-on components so you only need something to take notes.
- Use the Q&A function to ask questions.
- Please fill out the survey at the end of the session so we can improve our future sessions.

The Optimization Mindset

Optimize Only When Necessary



Start with a goal in mind

- Cost target
- SLA / Performance target

Tackle the easy things first

Spend time understanding the problem

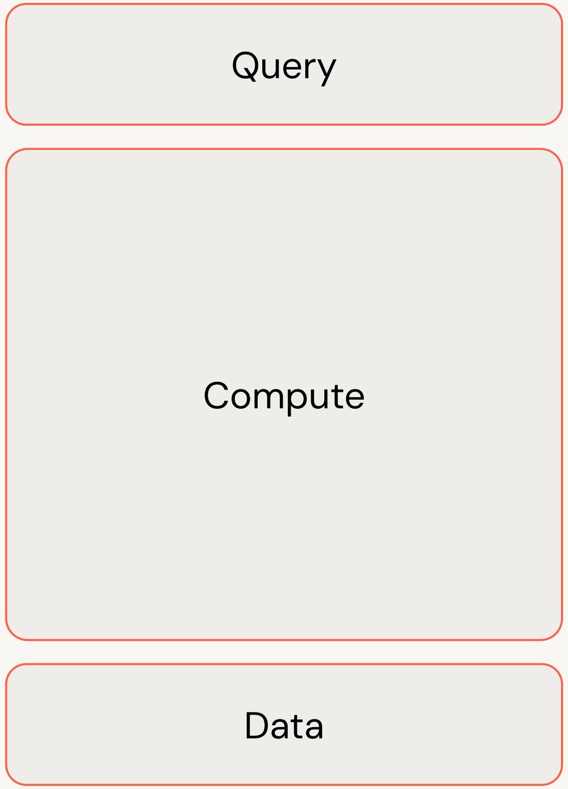
Benchmark and Iterate

Keep tracks of results so you know you are making progress

Know when to stop

Refer to step 1

Understanding the journey of a query

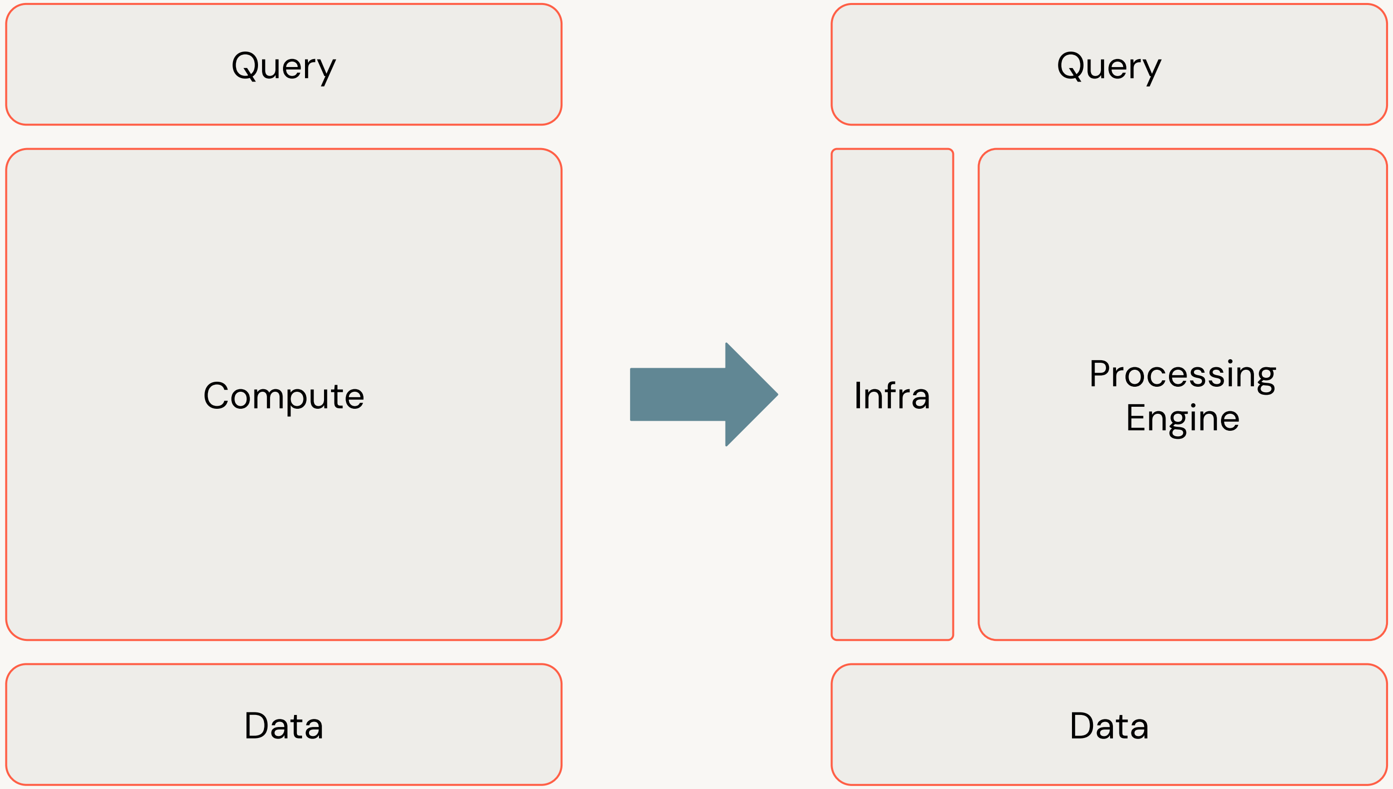


Query

Compute

Data

Understanding the journey of a query



Query

Compute

Data

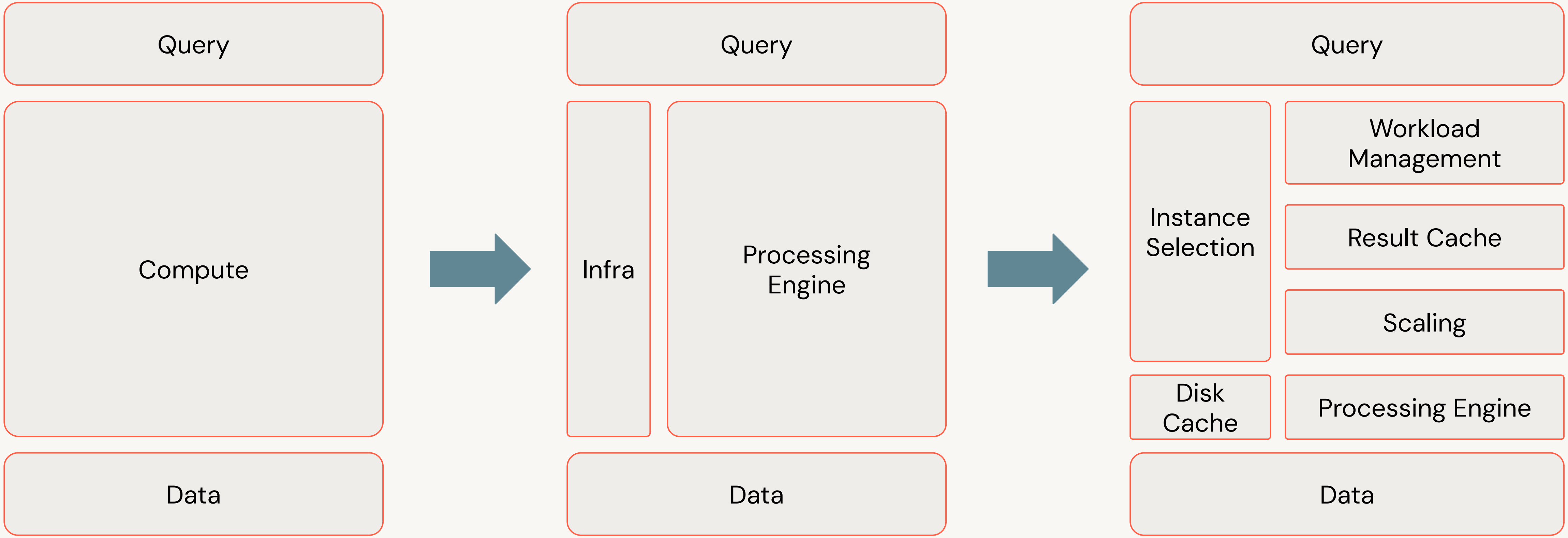
Query

Infra

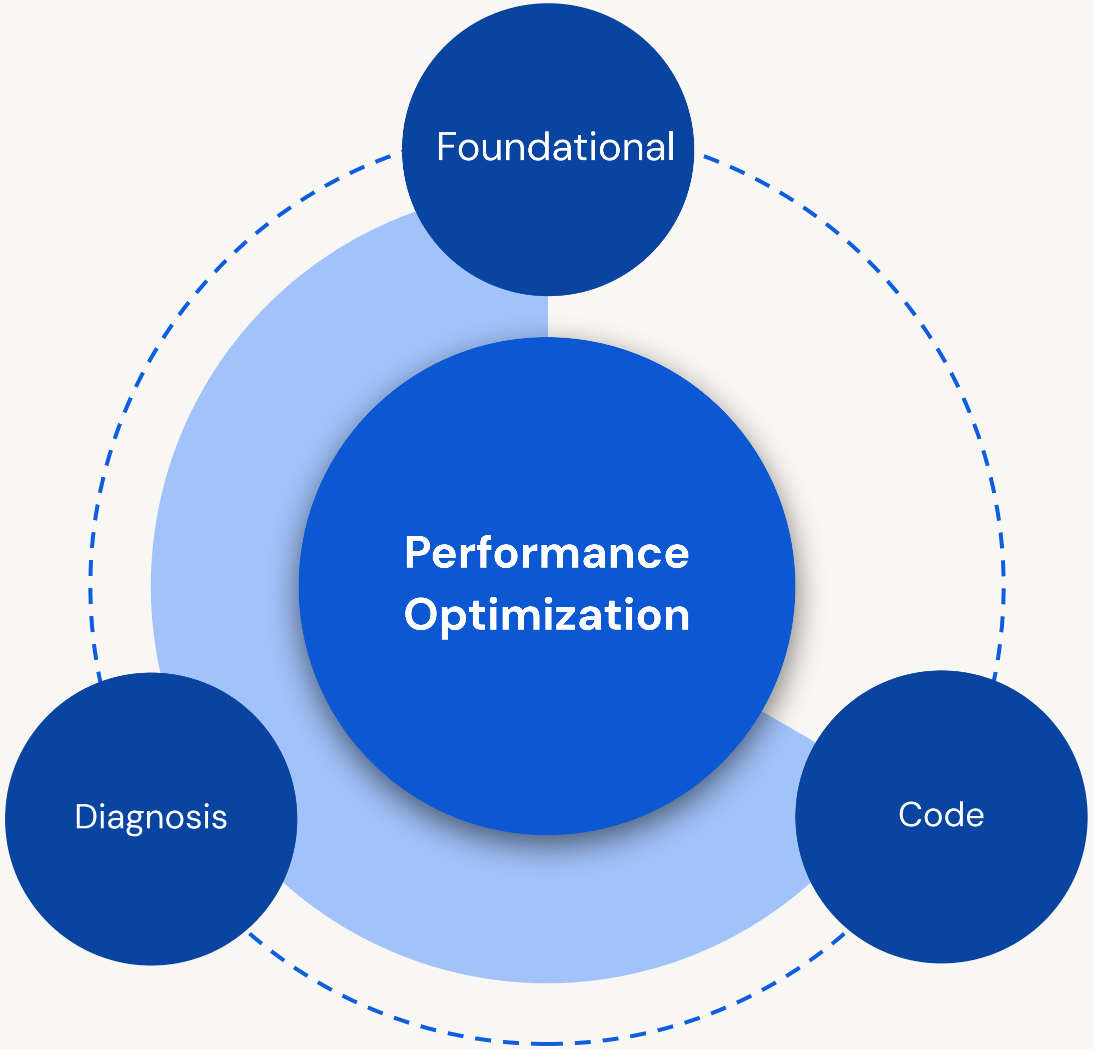
Processing Engine

Data

Understanding the journey of a query

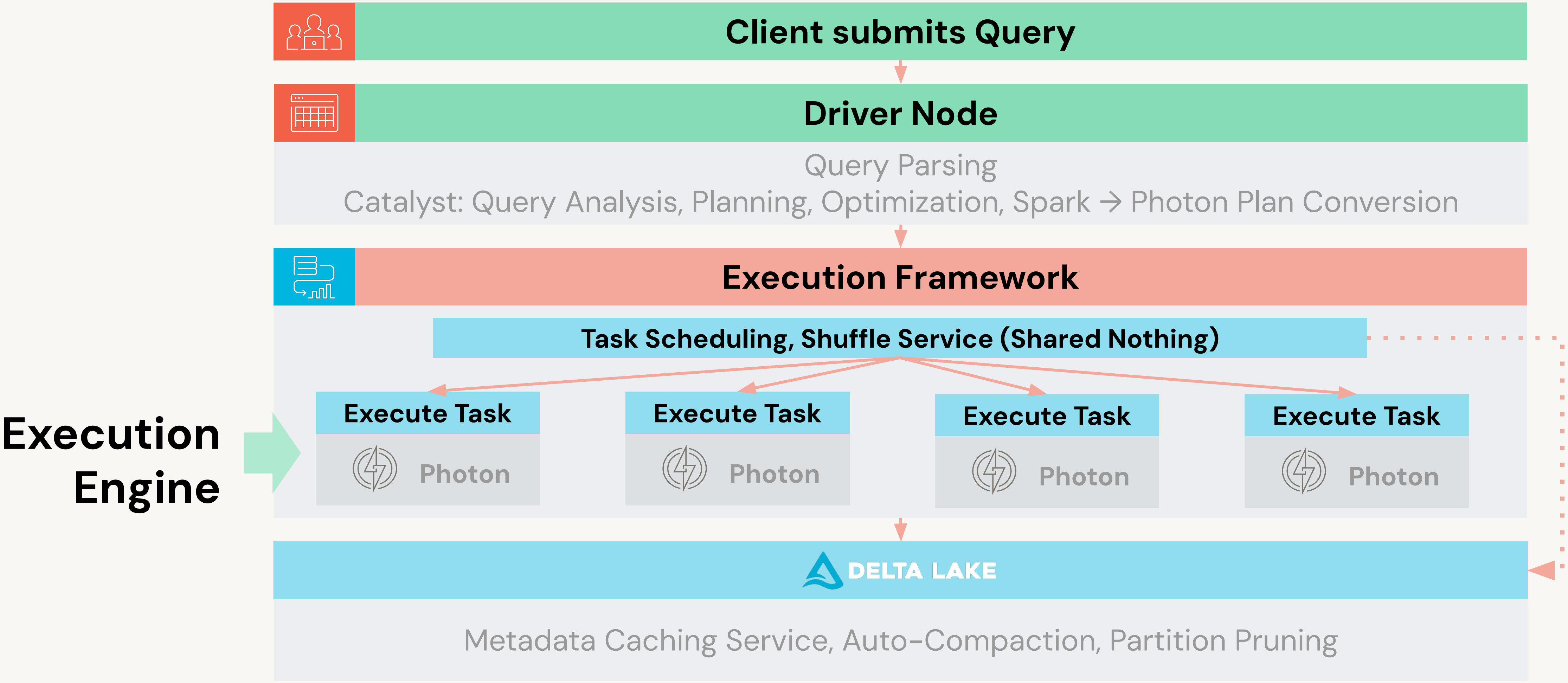


Performance Optimization – Framework



Foundational – Compute

Photon – Speeding up Data Processing


Client submits Query
Driver Node
Query Parsing
Catalyst: Query Analysis, Planning, Optimization, Spark → Photon Plan Conversion
Execution Framework
Task Scheduling, Shuffle Service (Shared Nothing)
Execute Task
Photon
Execute Task
Photon
Execute Task
Photon
Execute Task
Photon
Execution Engine
DELTA LAKE
Metadata Caching Service, Auto-Compaction, Partition Pruning

Where and When to Use Photon

Batch

COPY INTO

Auto Loader

Structured Streaming

Delta Live Tables

✅ Delta Lake
✅ Parquet
✅JSON
✅CSV
✅AVRO
✅XML
✅Binary
✅JDBC/ODBC

Ingestion

APIs and Methods

✅ DataFrame
✅ SQL
✅ SQL UDFs
🚧 Pandas and Python UDFs

We do not support these

❌ RDDs
❌ Typed Datasets
❌ Java/Scala udfs

Workloads that benefit the most

- Joins and aggregation heavy computations
- Delta Lake merge
- Reading/writing wide tables
- Decimal computations
- Delta Live Tables (DLT) and AutoLoader
- Updates and delete workloads via DV’s

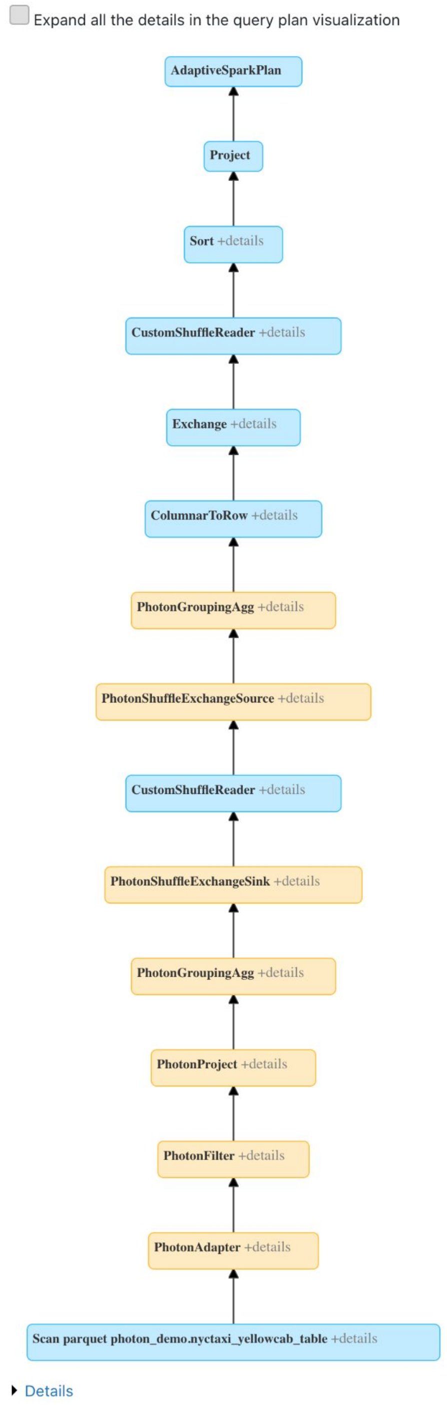
Find Out Why Things Don’t Photonize

- Photon makes everything faster
- If a query falls out of Photon, figure out why!

  collect_set? Use collect_list(distinct)

  UDFs? RDDs? Avoid those wherever possible.

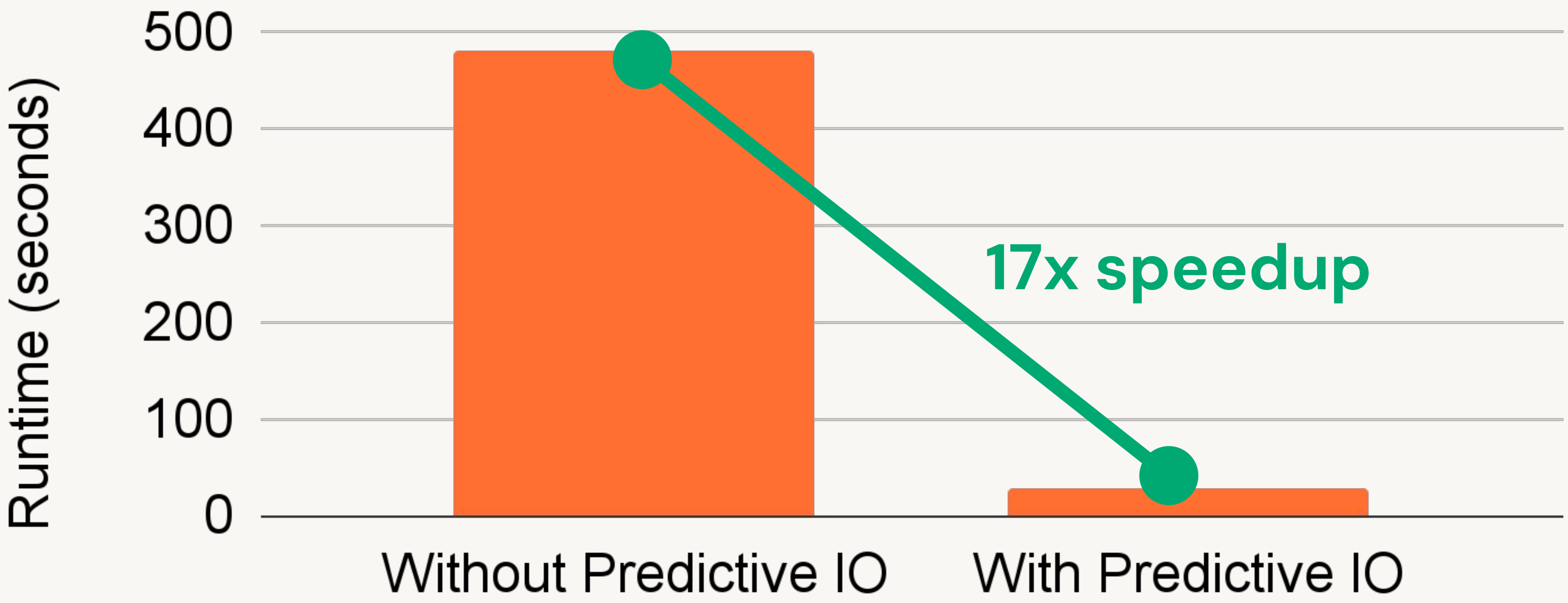
  Non-photonizable source? We may have a preview for that!
- Getting things to run in Photon completely solves most problems


□ Expand all the details in the query plan visualization
AdaptiveSparkPlan
Project
Sort +details
CustomShuffleReader +details
Exchange +details
ColumnarToRow +details
PhotonGroupingAgg +details
PhotonShuffleExchangeSource +details
CustomShuffleReader +details
PhotonShuffleExchangeSink +details
PhotonGroupingAgg +details
PhotonProject +details
PhotonFilter +details
PhotonAdapter +details
Scan parquet photon_demo.nyctaxi_yellowcab_table +details
Details

Predictive I/O – Speeding up point queries

Let the compute engine determine the best way to fetch data

```
SELECT protoBase64 FROM query_profile_protos WHERE id LIKE '204ff749-dc88-4b0a%’
```



17x speedup

Spark 3.0 Adaptive Query Execution

- Adapts a query plan automatically at runtime based on accurate metrics

- Capabilities:
  - Sort Merge Join (SMJ) → Broadcast Hash Join (BHJ)
  - Coalesce shuffle partitions
  - Handles skew

Compute Sizing

A balance between query performance and concurrency requirement

- Vertical Scaling - Cluster Size (2XS ⇔ 4XL)
  - Larger cluster for larger queries and tables
- Horizontal Scaling - No. of Clusters (Min ⇔ Max #)
  - More cluster for more concurrent queries
- Monitor Query History to find the right fit
  - Too many queries in queue = more clusters
  - Queries taking too long = larger clusters

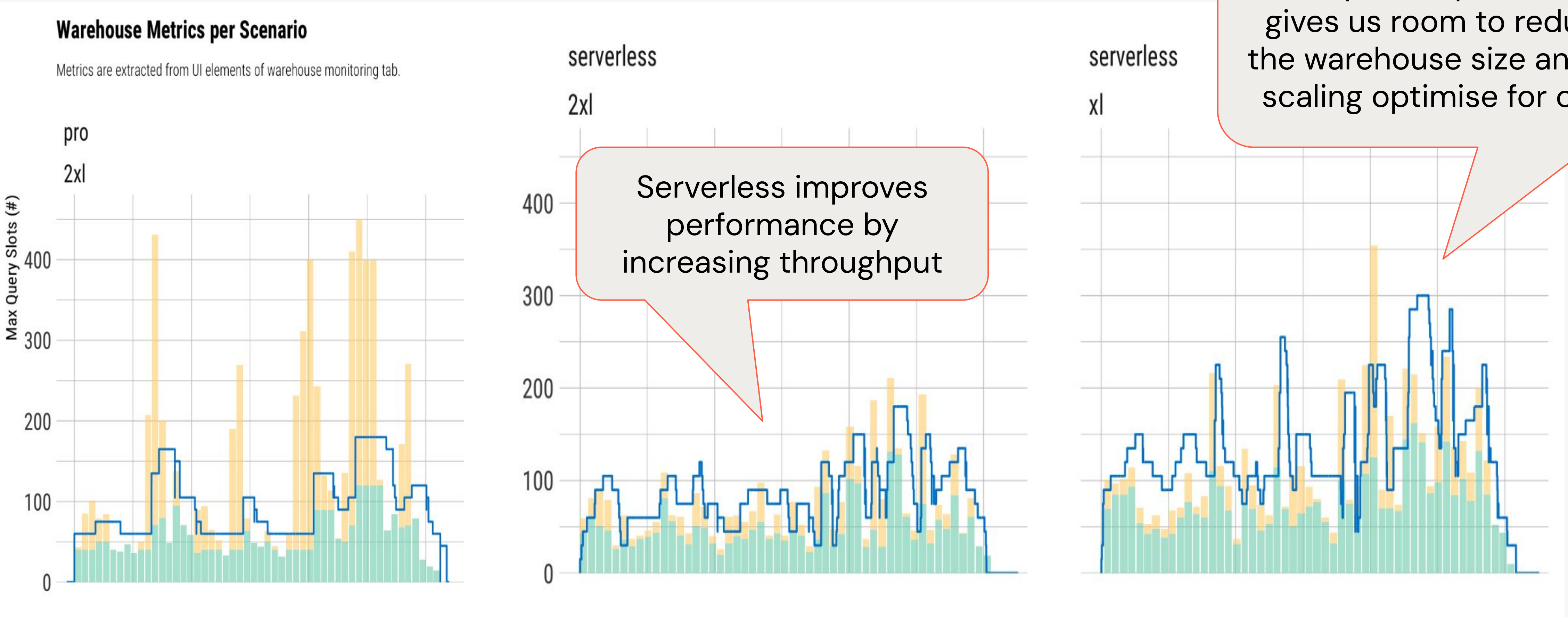
Compute – Architectural Considerations

Isolated Clusters & Warehouses to Avoid Resource Contention

- **Ephemeral Job compute**
    - Jobs – Isolated compute for ingestion + ETL jobs, can be sized/optimized for that workload, run on a schedule
    - Only charged for when the job is running
- **Shared development clusters**
    - All-purpose – Auto-scale, auto-pause to only use when teams are actively developing, only resources needed
    - Recommended to develop and test with a subset of the full dataset
- **Shared SQL warehouse for ad-hoc analysis**
    - SQL warehouse – Auto-scale, auto-pause to only use when teams are actively querying, only resources needed
    - Serverless available for instant startup, shutdown to reduce idle time
- **Separate SQL warehouse for BI reporting**
    - Size appropriately for BI needs, avoid contention with other processes

Serverless – Shifting the paradigm

How to fundamentally move your price-performance profile



Serverless improves performance by increasing throughput

The improved performance gives us room to reduce the warehouse size and let scaling optimise for cost

A Note on Classic Compute

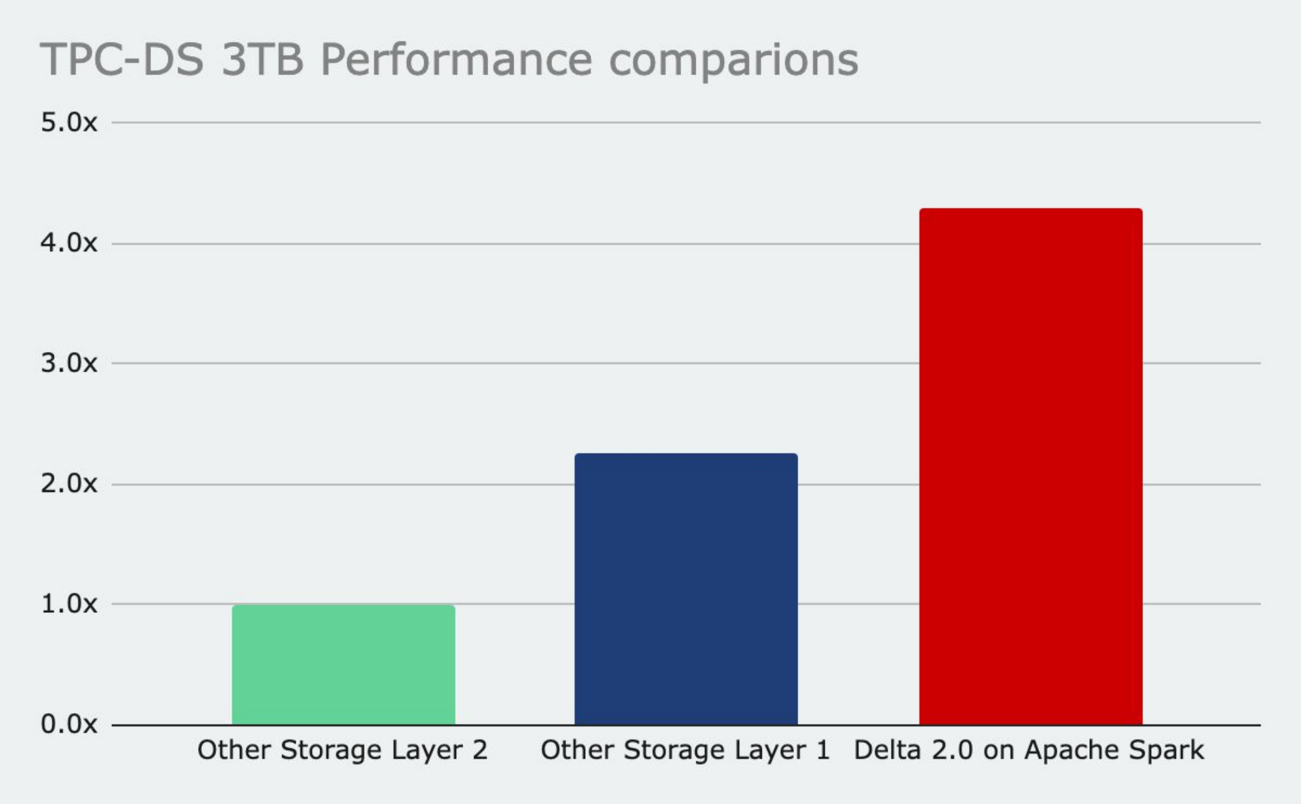
- DBSQL Warehouse automatically take cares of compute instance selection and cluster configurations

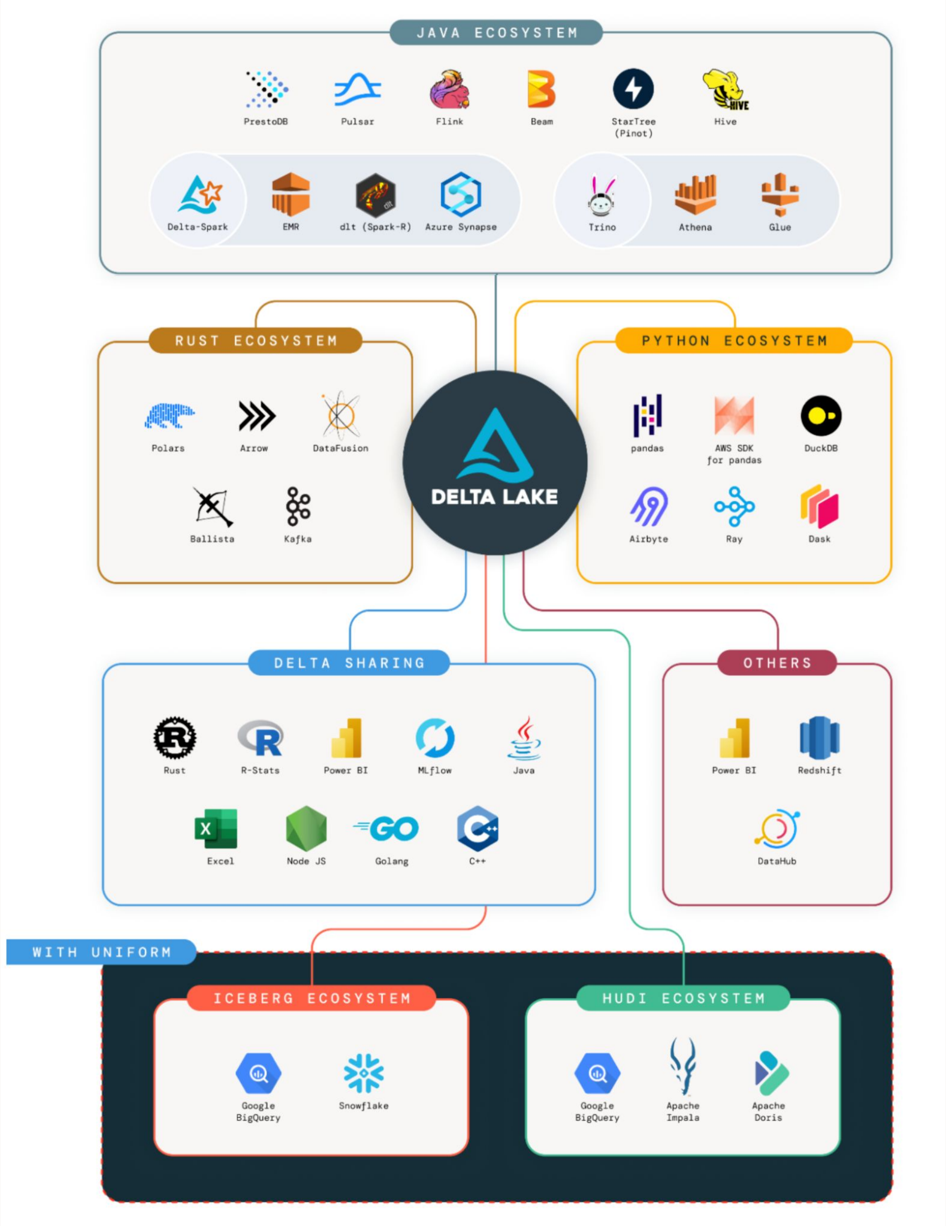
- Classic Compute Clusters (Jobs, Interactive, etc.) still give you the ability to configure instances and here is the TLDR
    - Core:RAM Ratio - Most core given enough memory for the budget
    - Processor Type - ARM based chip can work quite well
    - Local Storage - Disk Cache is useful for repeated data access
    - Driver Size - Don’t over complicate it (4-8 core with 16-32 GB RAM should be enough)
    - Spot Availability - Stability is more important for long running jobs
    - Auto Scaling - Achieve high cluster utilization and reduce overall cost

Foundational - Data

Delta Lake

Most performant modern open data format


TPC-DS 3TB Performance comparions

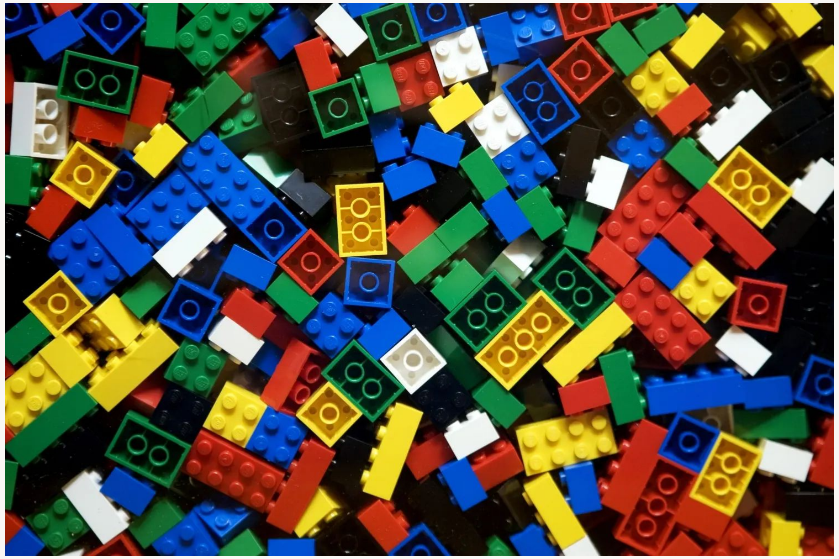


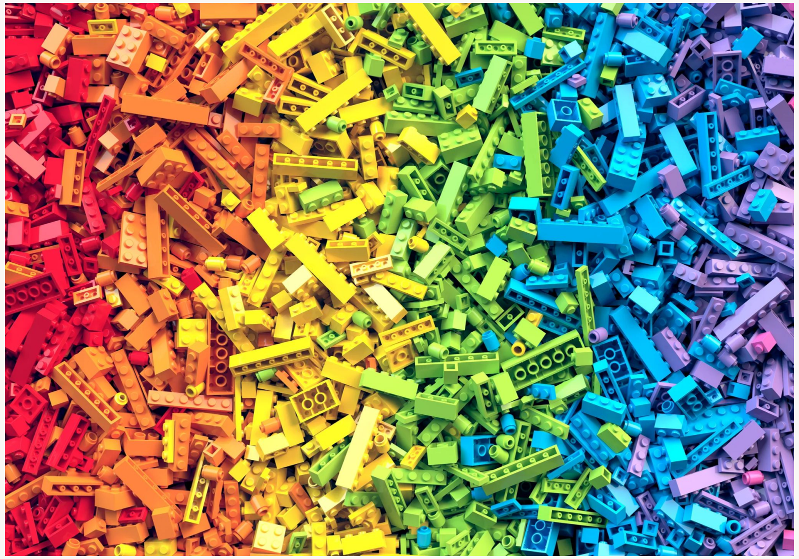
Data Layout matters

Putting the right things together makes life easier

`SELECT COUNT(*) FROM LEGOS WHERE COLOUR = “RED”`

How do you want to store your legos?





What about **`SELECT COUNT(*) FROM LEGOS WHERE SIZE = “SMALL”`**?

Data Layout Rationale

Different ways to organise your data so that we don't need to read too much unnecessary files

```
Select * from deltalake_table where part = 2 and col = 6
```

Partition Pruning

```
/path/to/deltalake_table/
                  part=1/part_00001.parquet
                  part=1/part_00002.parquet
                  part=1/part_00003.parquet
                  part=2/part_00001.parquet
                  part=2/part_00002.parquet
                  part=2/part_00003.parquet
```

File Skipping

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

Data Layout – To partition or not partition

DO NOT partition unless you know why you are partitioning

- **Over-partition is worse than no partition at all**
  - small files kill performance
- Reasons to partition
  - Table size > 100TB
  - Isolating data for separate schemas (i.e. multiplexing)
  - Governance use cases where you commonly delete entire partitions of data
  - Physical boundary to isolate data is required
- Partition best practices
  - Keep partition size between 1GB and 1TB
  - Combine partition with Z-order

Data Skipping and Delta Lake 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 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`

Data layout – Clustering

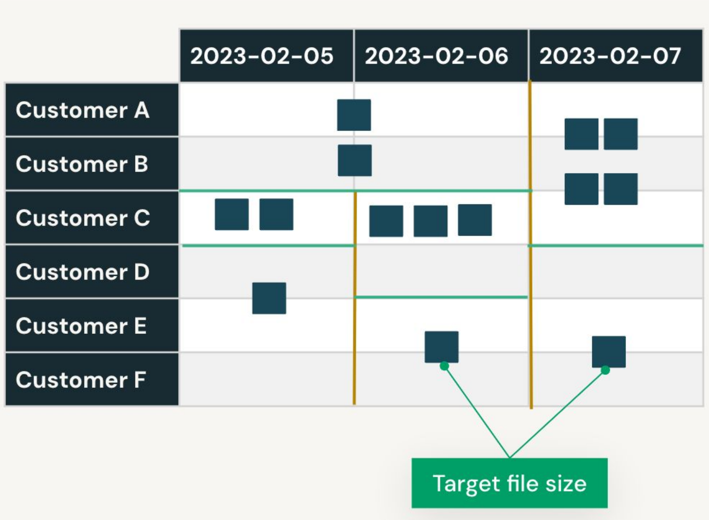
Sort the data in ways that you will need it

Z-Ordering

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

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

Liquid Clustering



Liquid Clustering – No More partitions

- Fast
  - Faster writes and similar reads vs. well-tuned partitioned tables
- Self-tuning
  - Avoids over- and under-partitioning
- Incremental
  - Automatic partial clustering of new data
- Skew-resistant
  - Produces consistent file sizes and low write amplification
- Flexible
  - Want to change the clustering columns? No problem!

Scenarios Benefiting from Liquid Clustering

- Tables often filtered by high cardinality columns.
- Tables with significant skew in data distribution.
- Tables that grow quickly and require maintenance and tuning effort.
- Tables with concurrent write requirements.
- Tables with access patterns that change over time.
- Tables where a typical partition key could leave the table with too many or too few partitions.

Data Layout – File Sizes

When it comes to performance, file size matters

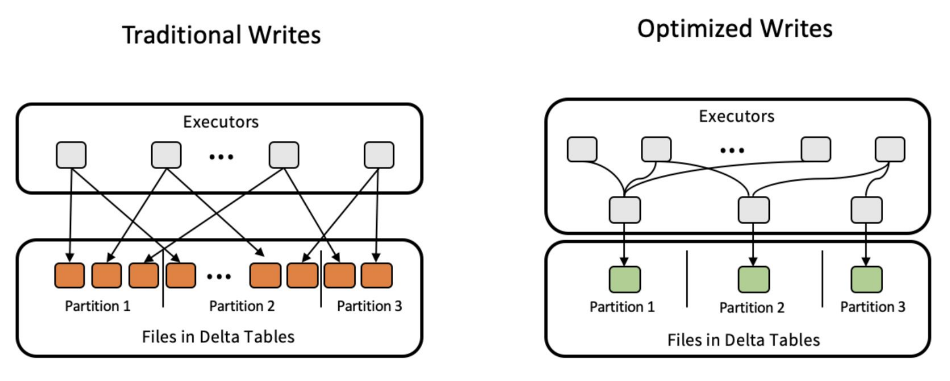
Small File Size

- Less data to read
- More files
- Rewrite is cheaper

Large File Size

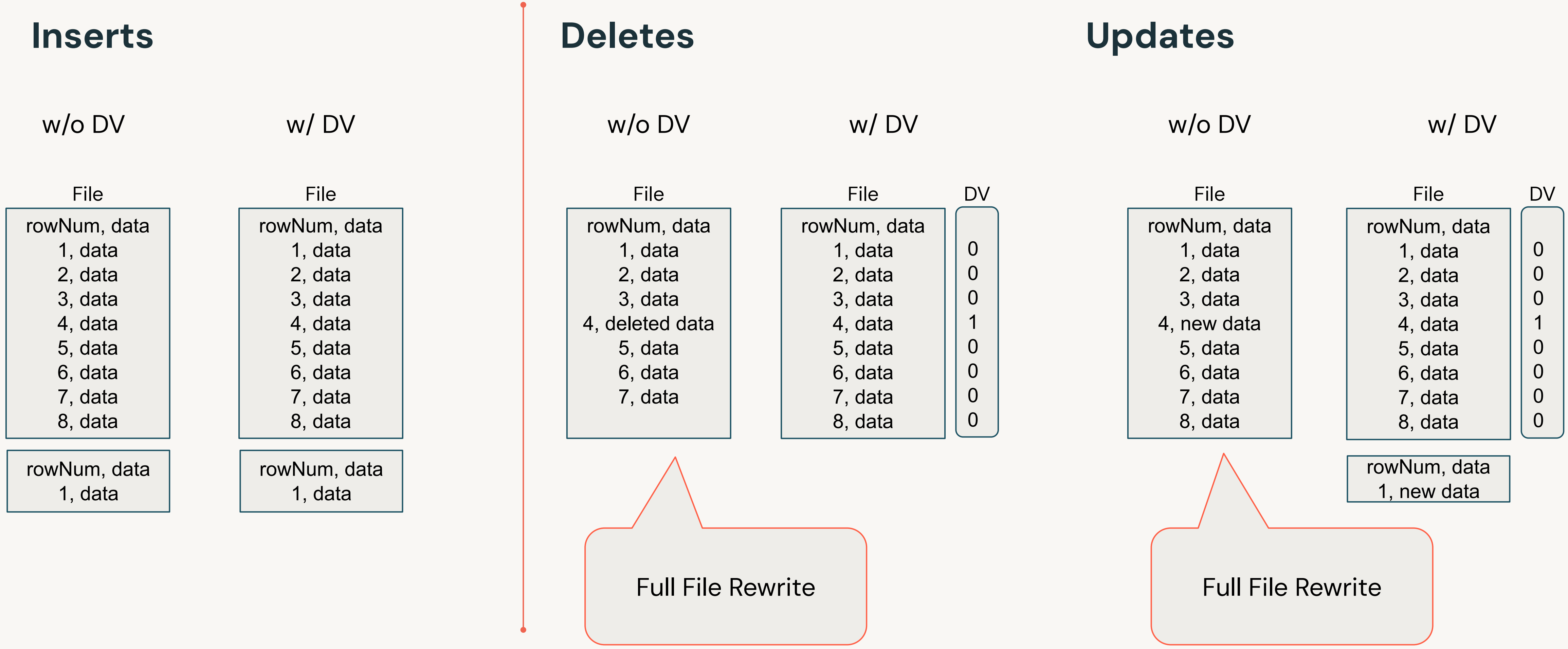
- More data to read
- Less files
- Rewrite is expensive

`delta.tuneFileSizesForRewrites`



Deletion Vector

Amortisation of rewrite costs



Inserts

w/o DV

File

rowNum, data
1, data
2, data
3, data
4, data
5, data
6, data
7, data
8, data

rowNum, data
1, data

w/ DV

File

rowNum, data
1, data
2, data
3, data
4, data
5, data
6, data
7, data
8, data

rowNum, data
1, data

Deletes

w/o DV

File

rowNum, data
1, data
2, data
3, data
4, deleted data
5, data
6, data
7, data

Full File Rewrite

w/ DV

File	DV
rowNum, data	
1, data	0
2, data	0
3, data	0
4, data	1
5, data	0
6, data	0
7, data	0
8, data	0

Updates

w/o DV

File

rowNum, data
1, data
2, data
3, data
4, new data
5, data
6, data
7, data
8, data

Full File Rewrite

w/ DV

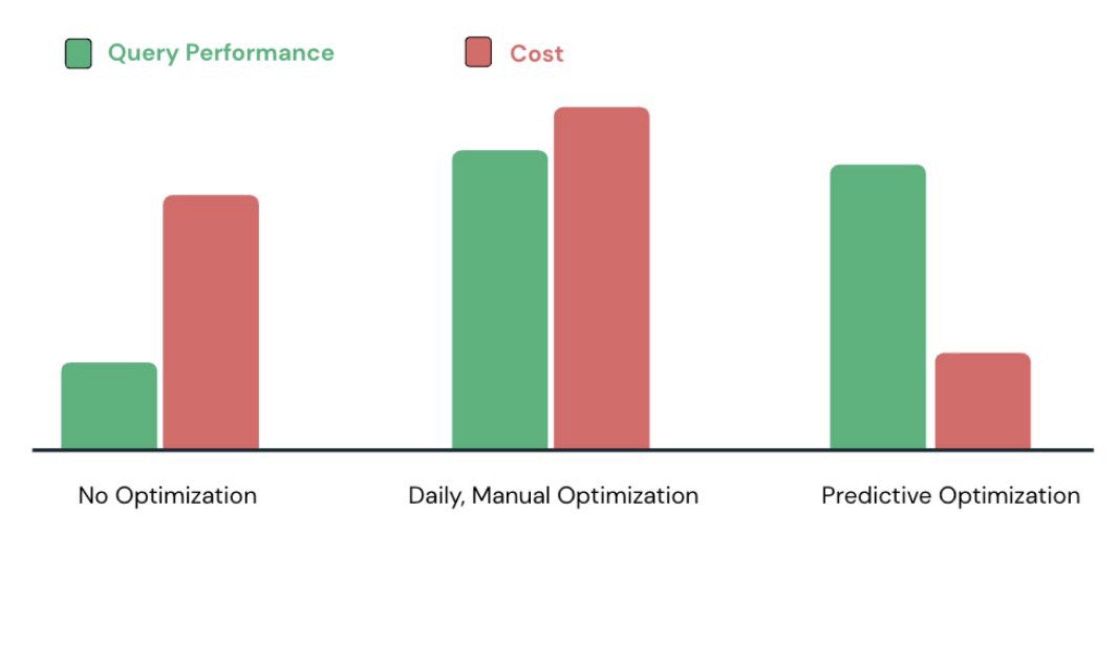
File	DV
rowNum, data	
1, data	0
2, data	0
3, data	0
4, data	1
5, data	0
6, data	0
7, data	0
8, data	0

rowNum, data
1, new data

Predictive Optimization

Bringing it all together automatically

- Scheduling of these optimization can be tricky
- Mistakes can be made if users forget to set up these process
- Predictive Optimization automatically determines which operations to execute based on usage
- Prioritise high return operations based on expected benefits



Code Optimization

Basics

1. In production jobs, avoid operations that trigger an action besides reading and writing files. These include count(), display(), collect().
2. Avoid operations that will force all computation into the driver node such as using single threaded python/pandas/Scala. Use Pandas API on Spark instead to distribute pandas functions.
3. Avoid python UDFs which execute row-by-row. Instead use native pyspark functions or Pandas UDFs for vectorized UDFs.
4. Use Dataframes or Datasets instead of RDDs. RDDs cannot take advantage of the cost-based optimizer.

Controlled Batch Size For Streaming

- Defaults are large and non-deterministic:
  - 1000 files per micro-batch
  - No limit to input batch size

- Optimal mini-batch size → Optimal cluster usage

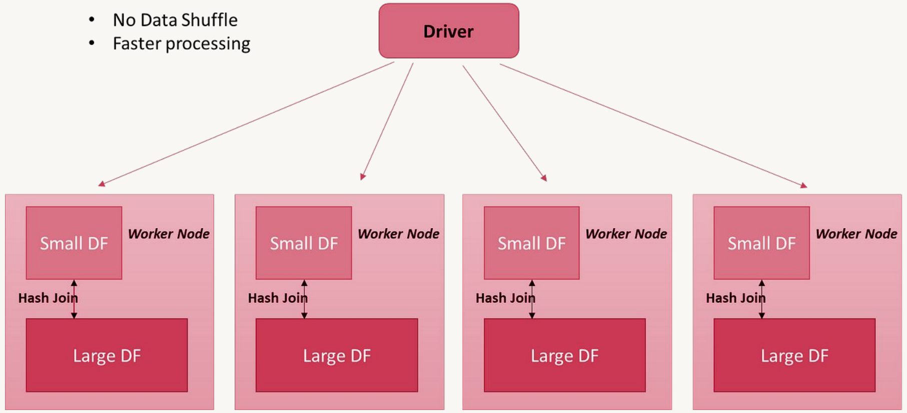
- Suboptimal mini-batch size → Performance cliff
-

  Per Trigger Settings:
  - Kafka
    - maxOffsetsPerTrigger

  - Delta Lake and Auto Loader
    - maxFilesPerTrigger
    - maxBytesPerTrigger

Broadcast Join

- The most performant type of join
- Distributes a small dataset across all worker nodes to minimize shuffling and speed up query execution.
- Triggered when smaller table/data-frame is lesser than **spark.sql.autoBroadcastJoinThreshold** (10 MB by default)
- Control micro-batch size to trigger it when joining with target table (e.g. during merge) in Structured Streaming
- Prerequisite for Dynamic File Pruning (DFP)

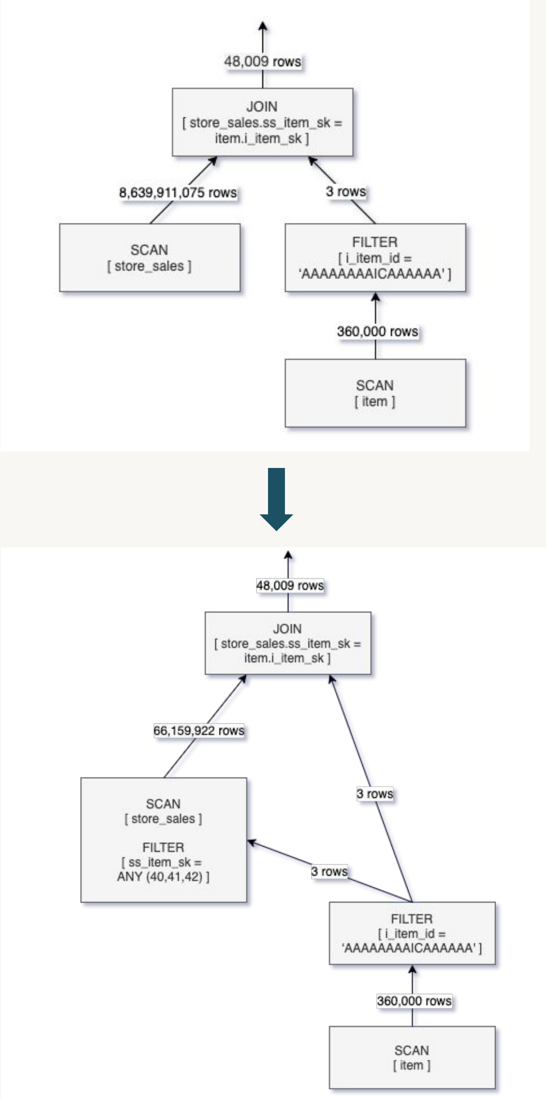


- No Data Shuffle
- Faster processing

Dynamic File Pruning (DFP)

- Intelligently skipping non-relevant data files during selective joins, achieving up to 8x faster performance

- Key Prerequisites:
    - The join strategy is BROADCAST JOIN
    - The join type is INNER or LEFT-SEMI


48,009 rows
JOIN [store_sales.ss_item_sk = item.i_item_sk]
8,639,911,075 rows
3 rows
SCAN [store_sales]
FILTER [i_item_id = 'AAAAAAAAICAAAAAA']
360,000 rows
SCAN [item]
48,009 rows
JOIN [store_sales.ss_item_sk = item.i_item_sk]
66,159,922 rows
3 rows
SCAN [store_sales]
FILTER [ss_item_sk = ANY (40,41,42)]
3 rows
FILTER [i_item_id = 'AAAAAAAAICAAAAAA']
360,000 rows
SCAN [item]

Incremental Processing via Streaming

- Consider streaming for all of your workloads:
  - Incremental processing resulting in reduced latency and quick time to insight
  - Built-in Checkpointing for exactly-once guarantees and fault tolerance
  - Efficient resource utilization
- Move to a CDC architecture pattern where you are only processing change data will greatly reduce overall processing time

DLT vs Structured Streaming

Feature	Structured Streaming	Delta Live Tables
Autoloader	✅	✅
Trigger Options	✅	✅
Workflow Support	✅	✅
Schema Evolution	✅	✅
CDC with SCD Type 1 and 2	❌	✅
Data Quality Constraints + Monitoring	❌	✅
Automatic Orchestration	❌	✅
Concurrent Streaming Jobs on a Cluster	❌ (not recommended)	✅
Pipeline Observability	❌	✅
Simplified Deployment + UI	❌	✅
Enhanced Autoscaling	❌	✅
Enzyme Runtime Engine	❌	✅

Simple query is usually a fast query

Get to the results with the least amount of data and transformation

1. Predicates are pushed as far up as possible
   a. Select the least amount of columns and rows that you need
   b. Align data layout with commonly used predicates (ZORDER, LIQUID)
   c. Make sure data is right sized (OPTIMIZE)
2. Simplify how you join your tables
   a. Join the smallest tables first / collect statistics for the optimizer to do it for you
   b. Provide join hints if you can
   c. Reduce unnecessary data movement, i.e. If you know your data layout and join keys you can use the right join strategy (sort-merge v. shuffle hash vs broadcast)
3. Simplify operations
   a. Be careful about expensive operations (distinct, sort, window)
   b. UDFs are powerful but they are not fast, try to use native functions as much as possible

Diagnosis

Performance 5S’s

Skew | Spill | Shuffle | Storage (small files) | Serialization

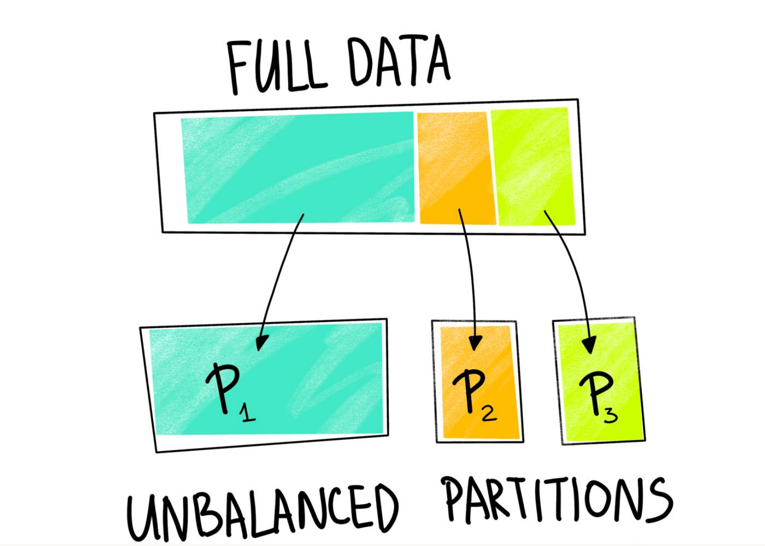
Common Performance Bottlenecks

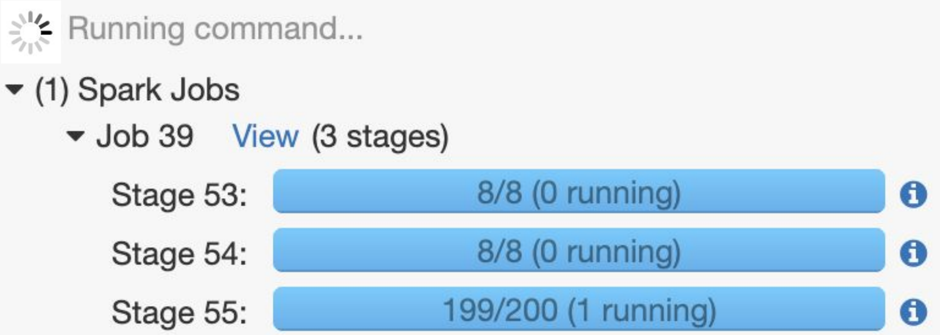
Encountered with any big data or MPP system

Symptom	Details
Skew	An imbalance in the size of partitions
Spill	The writing of temp files to disk due to a lack of memory
Shuffle	The act of moving data between executors
Small Files	A set of problems indicative of high overhead due to tiny files
Serialization	The distribution of code segments across the cluster

Skew – Mitigation

- Repartition Data (comes with caveats)
- Enable Adaptive Query Execution (AQE) in Spark 3 (enabled by default from DBR 7.3+)
- Employ skew hints
- Salting





```
Running command...
▾ (1) Spark Jobs
   ▾ Job 39   View (3 stages)
      Stage 53:   8/8 (0 running)
      Stage 54:   8/8 (0 running)
      Stage 55:   199/200 (1 running)
```

Spill – Mitigation

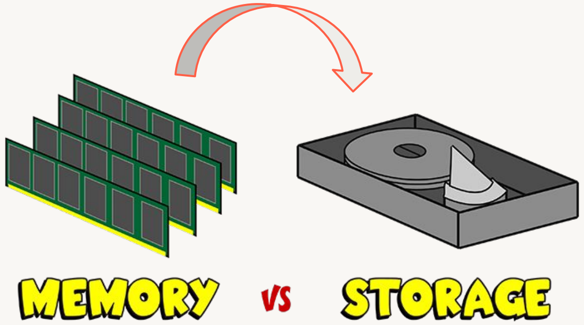
Option #1 – Allocate a cluster with more memory per worker

Tip: Larger, fewer nodes > Smaller, more nodes

Option #2 – In the case of *skew*, address that root cause first.

Option #3 – Decrease the size of each partition by increasing the number of partitions

- By managing spark.sql.shuffle.partitions
- By explicitly repartitioning
- By managing spark.sql.files.maxPartitionBytes



Shuffle – Mitigation

TIP: Don’t get hung up on trying to remove every shuffle

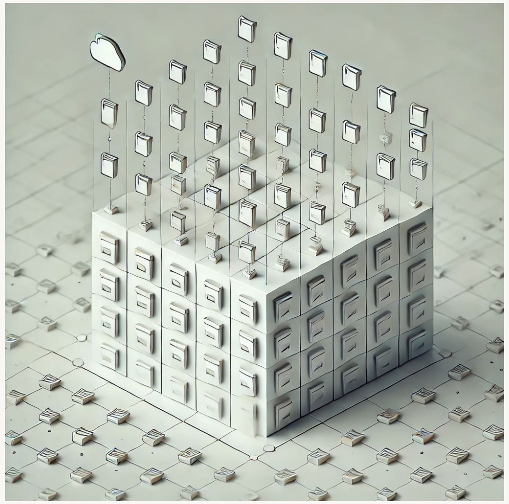
- Shuffles are often a necessary evil. Focus on the [more] expensive operations instead. Many shuffle operations are actually quite fast.
- Reduce network IO by using **fewer and larger workers**
- Reduce the **amount of data** being shuffled
  - Narrow your columns
  - Preemptively filter out unnecessary records

Storage (Tiny Files) – Mitigation

Make sure you constantly **optimize** and **vacuum** your delta tables

- OPTIMIZE will compact and Z-order/Cluster your files
- Vacuum will delete old versions and cleanup the metadata

(Predictive Optimization automates it)



Enable **auto-optimize** in all your tables unless there is a good reason for not doing so (e.g. streaming low latency requirements)

spark.databricks.delta.optimizeWrite.enabled = true

spark.databricks.delta.autoCompact.enabled = true

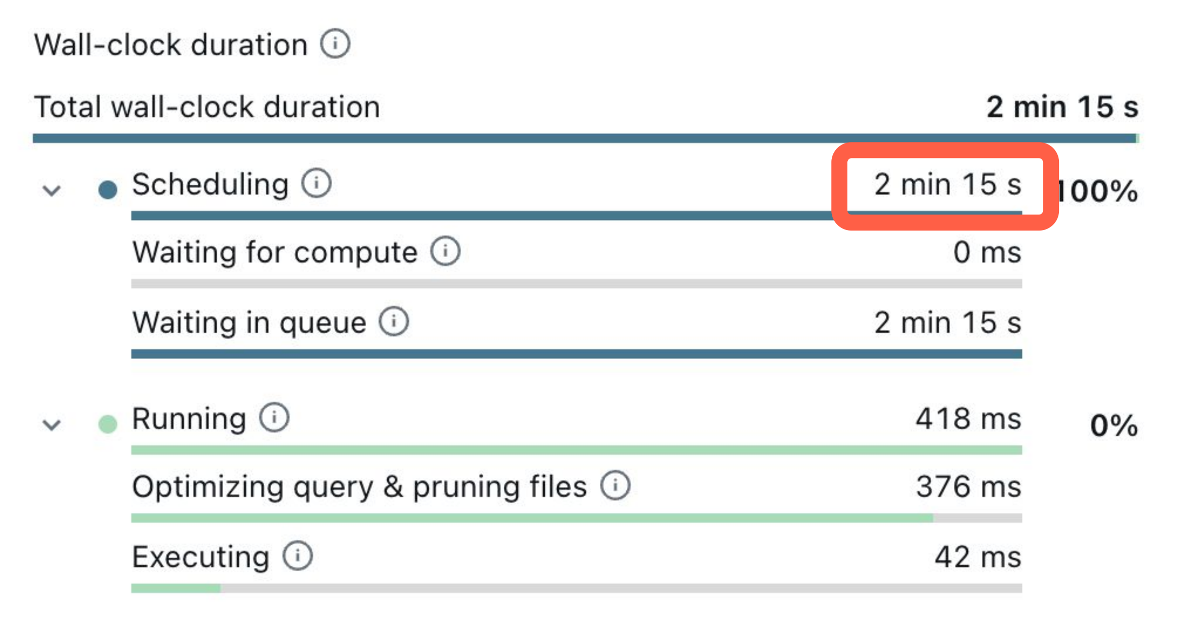
Serialization - Mitigation

- PySpark UDFs have significant serialization overhead between JVM and Python interpreter
- Act like “black box” and cannot be optimized by Spark's Catalyst optimizer, leading to suboptimal execution
- Thus, wherever possible, don’t use UDFs!
- The native and SQL higher-order functions are very robust
- But if you have to...
  - Use Arrow Optimized Python UDFs
  - Use Vectorized UDFs aka Pandas UDFs

Diagnosing performance issues

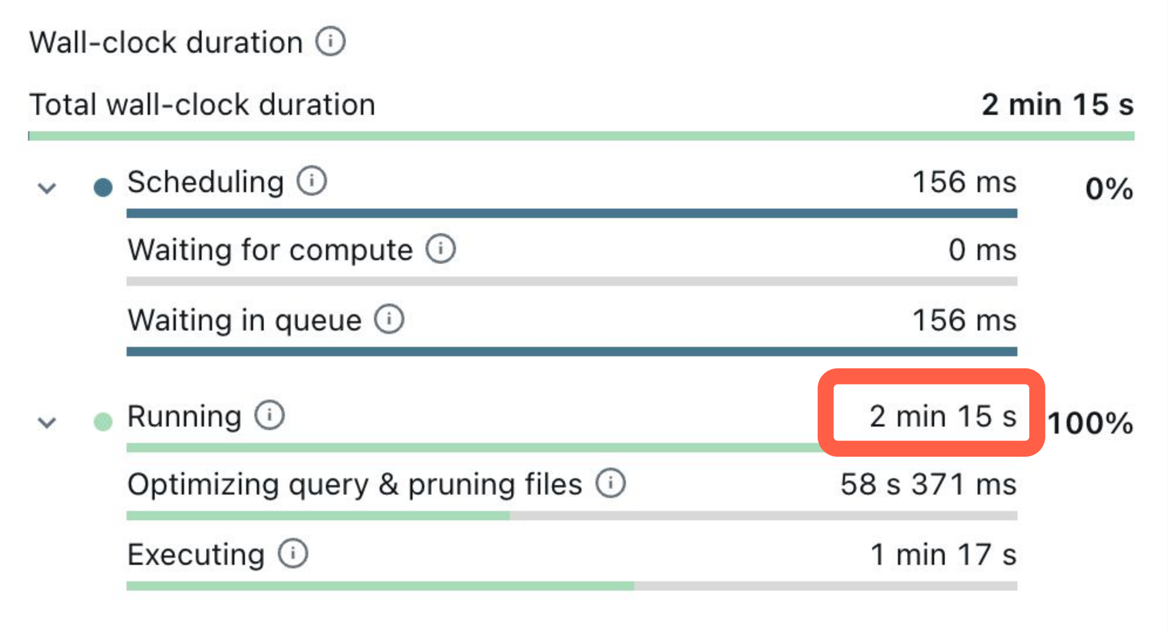
Scheduling Time v. Running Time

- Same wall clock time != same performance
- Scheduling time has nothing to do with your code
  - Waiting for compute = Need running compute = Serverless / pre-started cluster
  - Waiting in queue = We need more concurrency = Increase max # of clusters



Wall-clock duration ⓘ

Total wall-clock duration	**2 min 15 s**	
● Scheduling ⓘ	2 min 15 s	**100%**
Waiting for compute ⓘ	0 ms	
Waiting in queue ⓘ	2 min 15 s	
● Running ⓘ	418 ms	**0%**
Optimizing query & pruning files ⓘ	376 ms	
Executing ⓘ	42 ms	



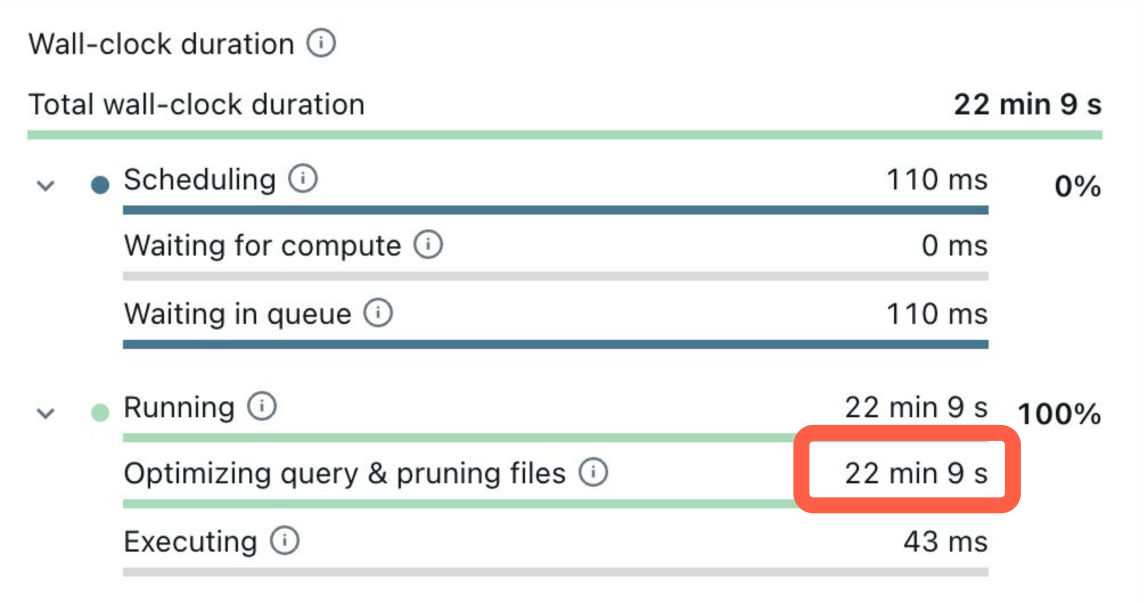
Wall-clock duration ⓘ

Total wall-clock duration	**2 min 15 s**	
● Scheduling ⓘ	156 ms	**0%**
Waiting for compute ⓘ	0 ms	
Waiting in queue ⓘ	156 ms	
● Running ⓘ	2 min 15 s	**100%**
Optimizing query & pruning files ⓘ	58 s 371 ms	
Executing ⓘ	1 min 17 s	

Diagnosing performance issues

Running Time Breakdown

- Same running time != Same time spent on execution
- Long Optimizing Query & Pruning Files Time = better stats collection



Wall-clock duration ⓘ

Total wall-clock duration 22 min 9 s

Scheduling ⓘ 110 ms 0%

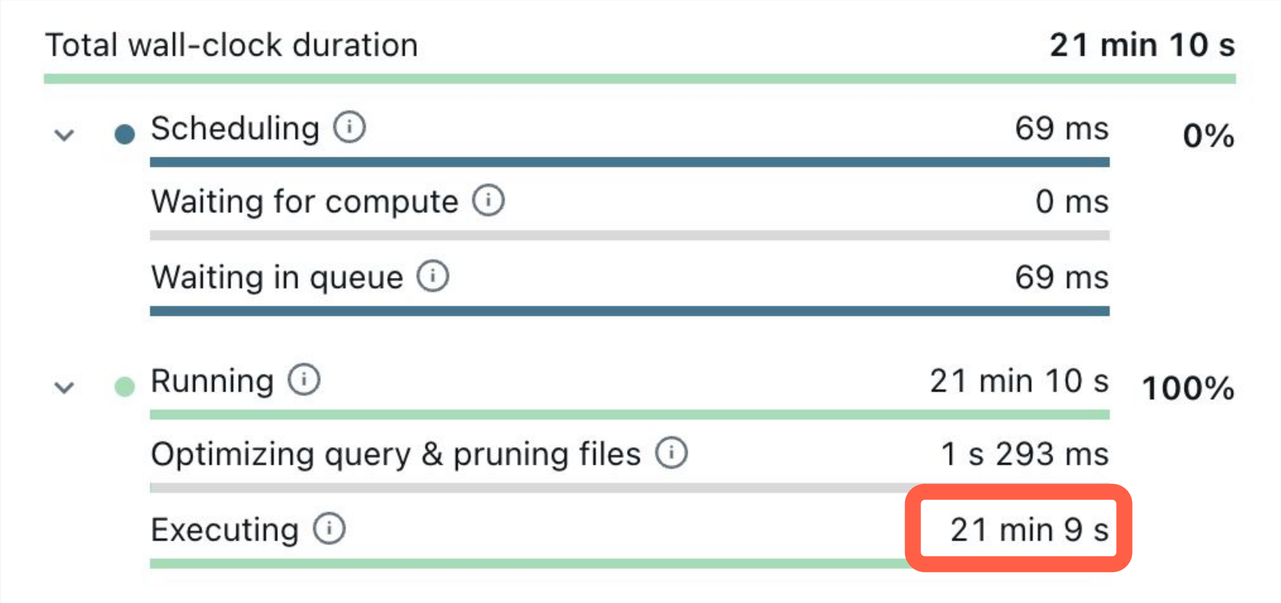
Waiting for compute ⓘ 0 ms

Waiting in queue ⓘ 110 ms

Running ⓘ 22 min 9 s 100%

Optimizing query & pruning files ⓘ 22 min 9 s

Executing ⓘ 43 ms



Total wall-clock duration 21 min 10 s

Scheduling ⓘ 69 ms 0%

Waiting for compute ⓘ 0 ms

Waiting in queue ⓘ 69 ms

Running ⓘ 21 min 10 s 100%

Optimizing query & pruning files ⓘ 1 s 293 ms

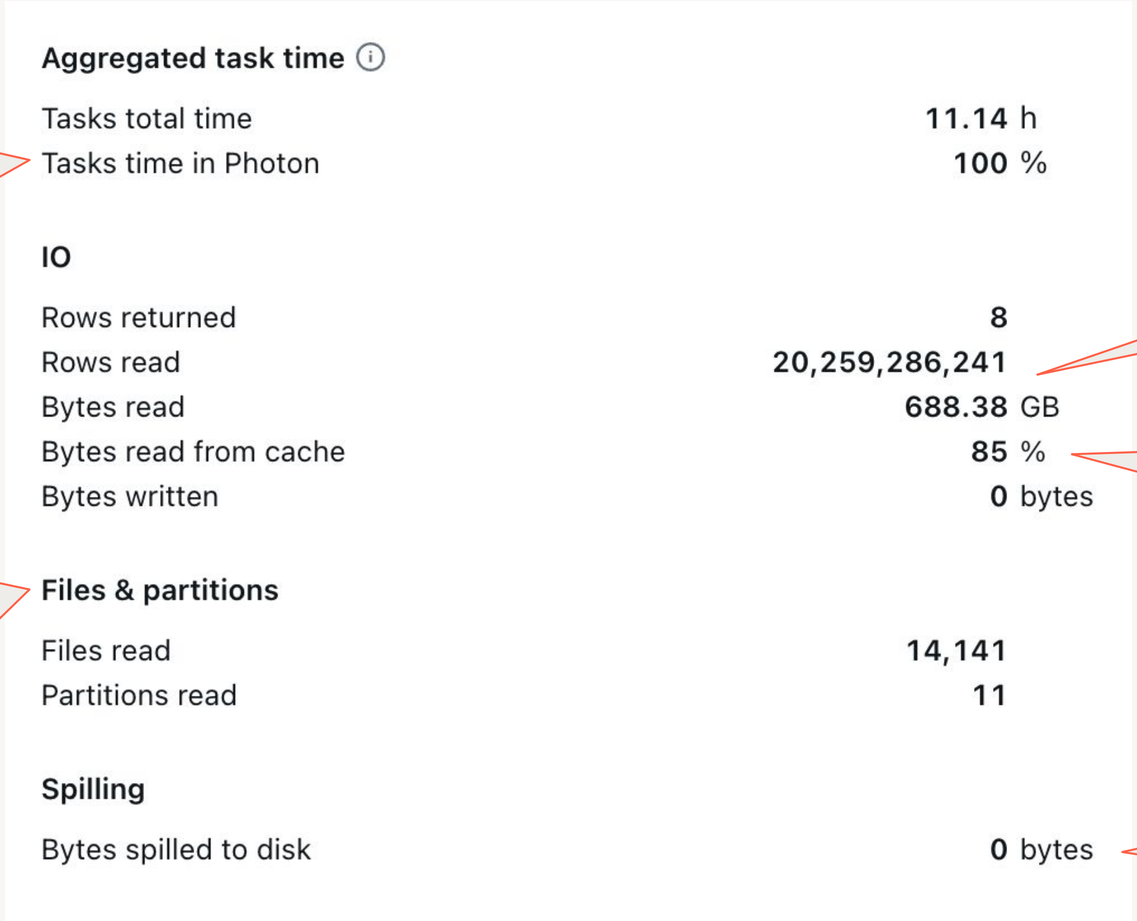
Executing ⓘ 21 min 9 s

Diagnosing performance issues

Execution Details

Make sure photon is close to 100%

If your data read is correct, are you reading too many files or partitions?



Aggregated task time ⓘ

Tasks total time	**11.14** h
Tasks time in Photon	**100** %

IO

Rows returned	**8**
Rows read	**20,259,286,241**
Bytes read	**688.38** GB
Bytes read from cache	**85** %
Bytes written	**0** bytes

Files & partitions

Files read	**14,141**
Partitions read	**11**

Spilling

Bytes spilled to disk	**0** bytes

Does the number of rows read make sense? Did you read too much data?

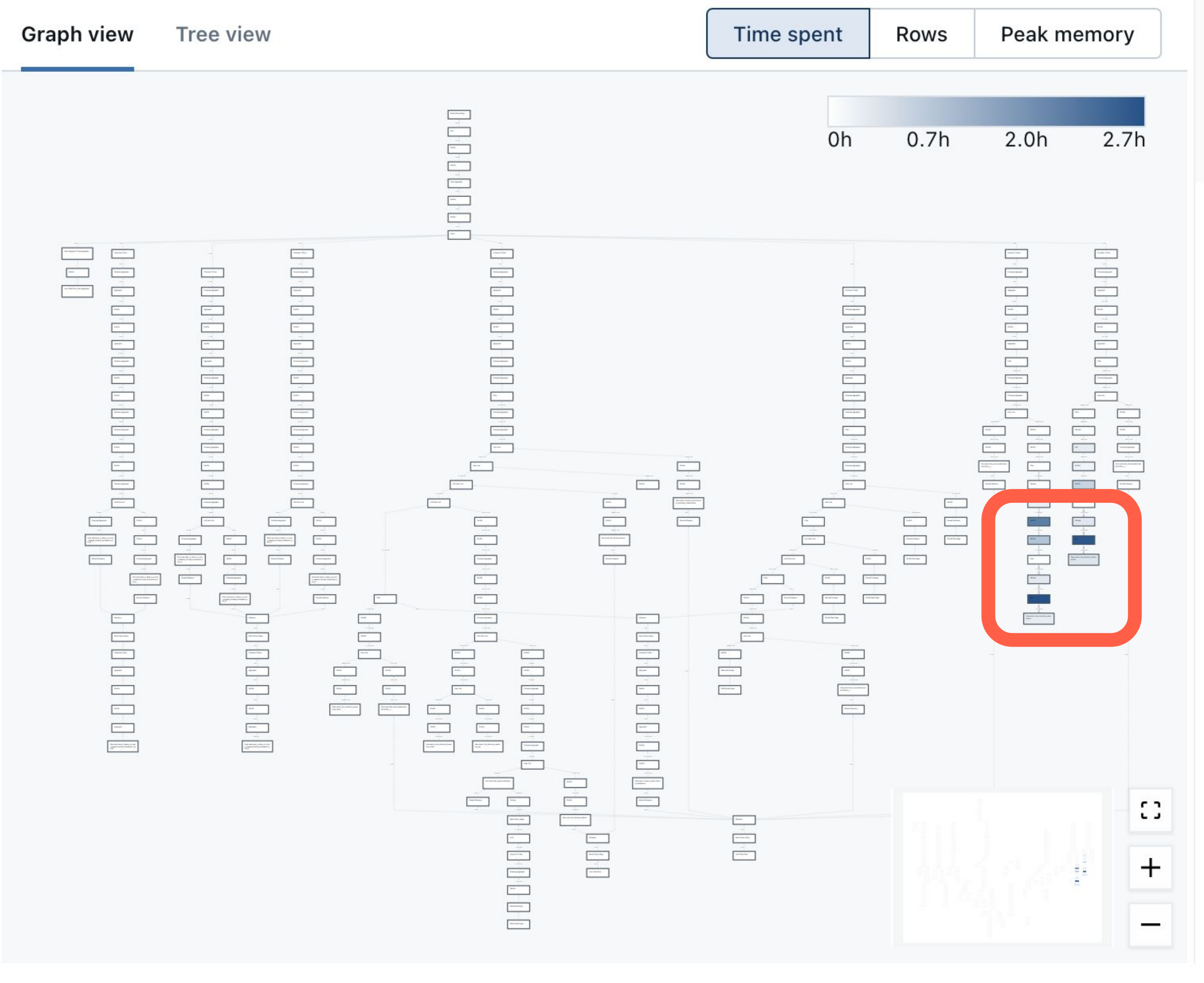
Disk cache meant your data is already cached on local storage

Spill meant your warehouse/cluster is too small, i.e. Not enough RAM

Diagnosing performance issues

Understanding Query Profile

- Execution can be broken down to individual operations within your query
- The most time spent operation is likely where you need to start
- It should tell you which part of the query is causing problem
- Knowing what is the problem doesn't mean it is an easy fix



What now?

Key takeaways

Optimize only when necessary

- Know what you are optimizing towards
- Focus on the easy things first (Platform -> Data -> Query)
- Leverage latest compute and features (i.e. Serverless Compute, Photon, Predictive Optimization, Liquid Clustering etc)
- Scale vertically or horizontally based on indicators (complexity, concurrency)
- Pivot to end-to-end incremental processing, via streaming, wherever possible
- Extensively use observability and monitoring tools to guide optimization efforts
- Know when to stop optimizing and start building more useful things

Thank you