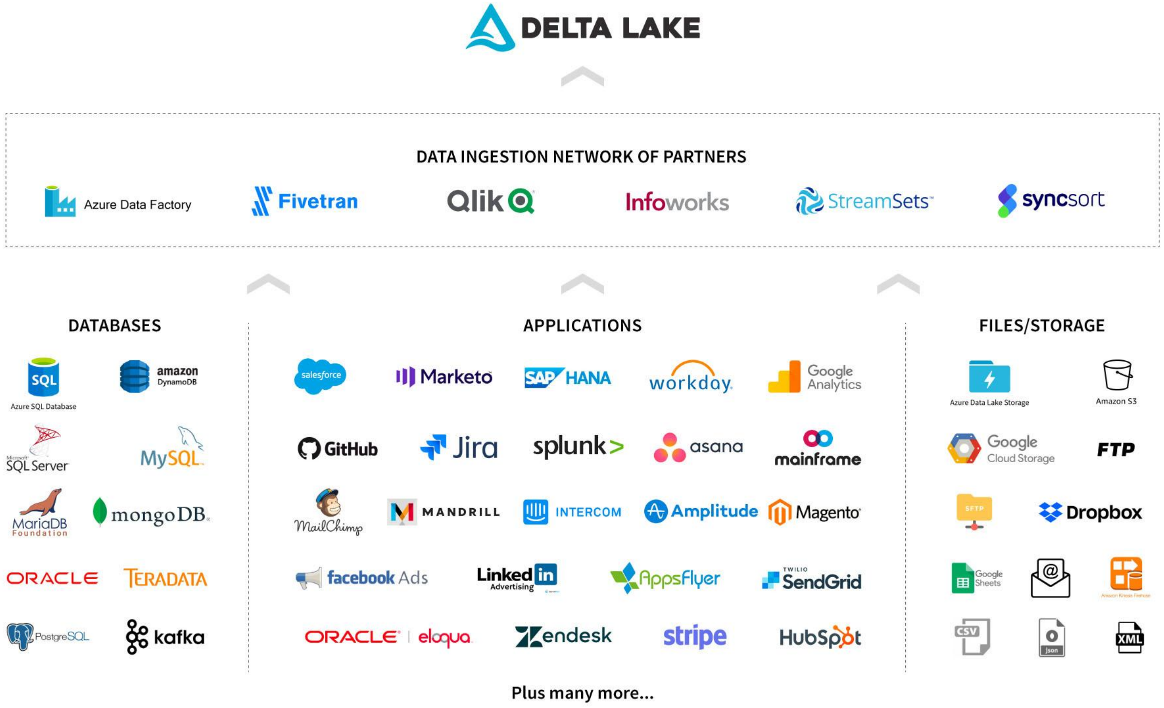
Making Ingestion Easy with Delta Lake

databricks

Data Ingestion Network of Partners

All Your Application, Database, and File Storage Data in your Delta Lake



databricks

Batch Loads using SQL

COPY INTO tableIdentifier

FROM { location | (SELECT identifierList FROM location) }

FILEFORMAT = { CSV | JSON | AVRO | ORC | PARQUET }

[FILES = ('<file_name>' [, '<file_name>'] [, ...])]

[PATTERN = '<regex_pattern>']

[FORMAT_OPTIONS ('dataSourceReaderOption' = 'value', ...)]

[COPY_OPTIONS ('force' = {'false', 'true'})]

Example:

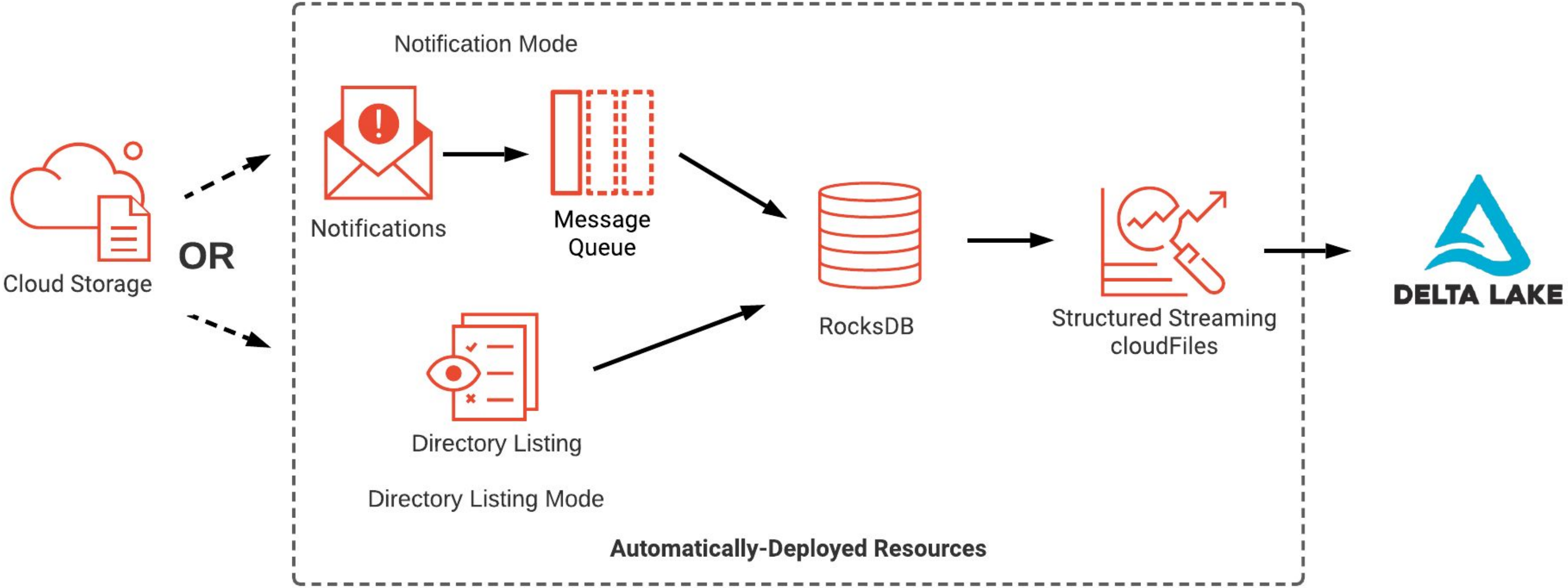
```
COPY INTO delta.`targetPath`
  FROM (SELECT key, textData, 'constant'
FROM 'sourcePath')
  FILEFORMAT = CSV
  PATTERN = '[a-cx-z]/[a-z]g[qwerty].csv'
  FORMAT_OPTIONS('header' = 'true')
```

Auto Loader

- No Custom Bookkeeping
- Scalable
- Easy to use

Auto Loader Under the Hood

```
.option("cloudFiles.useNotifications","true")
```



New File Detection Modes

Directory Listing Mode

- Default mode
- Easily stream files from object storage without configuration
- Creates file queue through parallel listing of input directory
- Good for smaller source directories

File Notification Mode

- Requires some security permissions to other cloud services
- Uses cloud storage queue service and event notifications to track files
- Configurations handled automatically by Databricks
- Scales well as data grows

Current Challenges

- Schema is unknown when initial configuring a pipeline
- New fields appear in source data
- Data types are updated or incorrectly enforced by source systems
- Some changes lead to silently dropping data
- Other changes completely break ingestion scripts

How Auto Loader Helps

Identify schema on stream initialization

Auto-detect changes and evolve schema to capture new fields

Add type hints for enforcement when schema is known

Rescue data that does not meet expectations

Notebook: Auto Loader

databricks

Notebook: Auto Load into Multiplex Bronze

databricks

Notebook: Streaming from Multiplex Bronze

databricks

Notebook: Streaming Deduplication

databricks

Notebook: Quality Enforcement

Notebook: Promoting to Silver

databricks

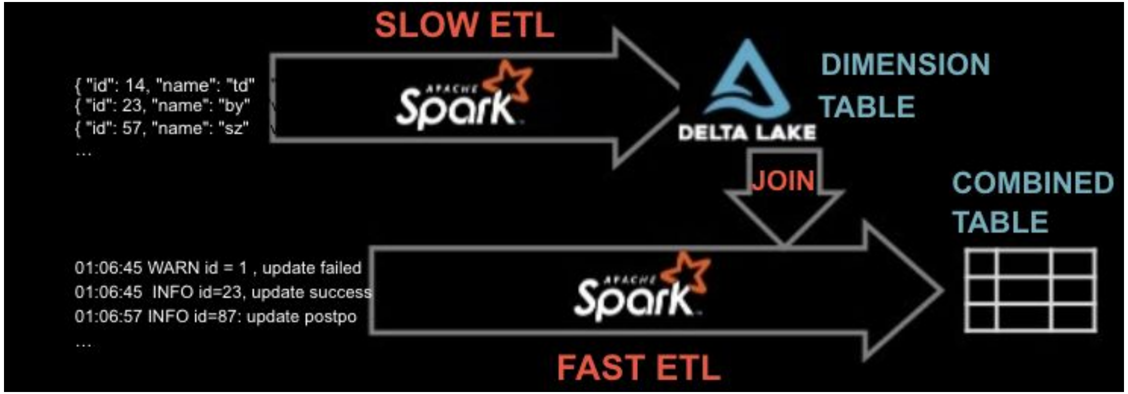
Slowly Changing Dimensions in the Lakehouse

Fact table fits streaming nicely

- The data often is a time series.
- No intermediate aggregations are done that overwrite output.
- You are only appending to the table.

Using Dimension Tables in Streaming

- Each micro-batch captures the most recent state of joined Delta table
- Allows modification of dimension while maintaining downstream composability



Slowly Changing Dimensions (SCD)

- **Type 0: No changes allowed (static/append only)**

  E.g. static lookup table

- **Type 1: Overwrite (no history retained)**

  E.g. do not care about historic comparisons other than quite recent (use Delta Time Travel)

- **Type 2: Adding a new row for each change and marking the old as obsolete**

  E.g. Able to record product price changes over time, integral to business logic.

Slowly Changing Dimensions (SCD)

Dimension Table (Type 0/1)

user_id	street	name
1	A	John
2	C	Doe

Dimension Table (Type 2)

user_id	street	name	**effective_from**	current
1	A	John	2020-01-01 00:00:00	Y
2	B	Doe	2020-01-01 00:00:00	N
2	C	Doe	2020-03-03 15:00:00	Y

SCD principles can be applied to facts

- **Type 0-A: No changes allowed (append only) - no history**

  E.g. stream of orders being placed

- **Type 0-AH: No changes allowed (append only) + history**

  E.g. stream of orders being placed, and keeping record of the effective time of the data to allow for corrections to be made without impacting the stream composability of the pipeline.

Fast Changing Facts (FCF)

event_id	order_id	user_id	occurred_at	action
456	123	1	2021-10-01 10:00:00	ORDER_PLACED
457	234	1	2021-10-01 10:05:00	ORDER_CANCELLED

event_id	order_id	user_id	occurred_at	action	effective_from
456	123	1	2021-10-01 10:00:00	ORDER_PLACED	2021-10-01 10:00:00
457	234	1	2021-10-01 10:05:00	ORDER_CANCELLED	2021-10-01 10:05:00
458	123	1	2021-10-02 01:00:00	ORDER_PLACED	2021-10-02 01:00:00
459	234	1	2021-10-02 01:00:00	ORDER_CANCELLED	2021-10-02 01:02:00

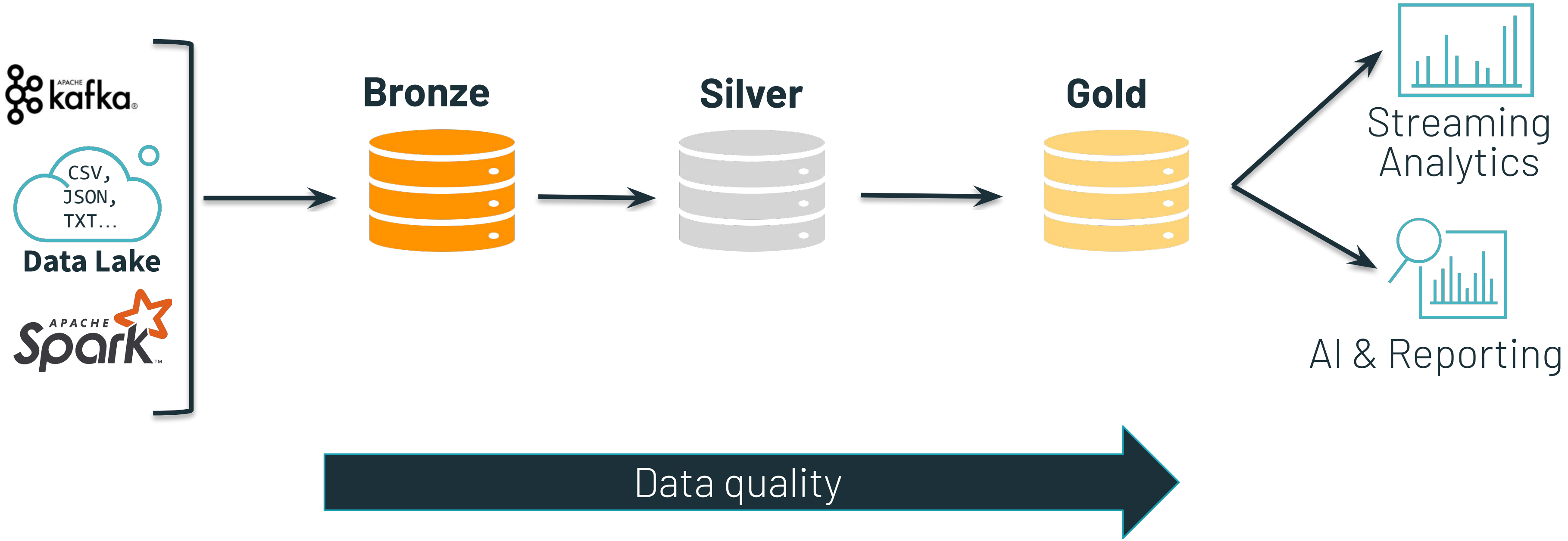
databricks

Notebook: Type 2 SCD

databricks

Propagating Changes with Delta Change Data Feed

Multi-Hop in the Lakehouse



databricks

What is Stream Composability?

- Structured Streaming expects append-only sources
- Delta tables are composable if new streams can be initiated from them

Operations that break stream composability

- Complete aggregations
- Delete
- UPDATE/MERGE

Data is changed in place, breaking append-only expectations

Workarounds for Deleting Data

ignoreDeletes

- Allows deletion of full partitions
- No new data files are written with full partition removal

ignoreChanges

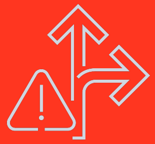
- Allows stream to be executed against Delta table with upstream changes
- Must implement logic to avoid processing duplicate records
- Subsumes ignoreDeletes

Current Challenges

Identifying Changes

Updates in ETL struggle to find changes in the data from version to version in large tables

Without information regarding the specific changes to be made, all data must be compared



Updating BI & Analytics Data

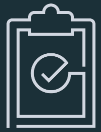
Real-time updates to BI and analytics require additional processing as changes arrive

Recalculating full datasets causes downtime to users incompatible with real-time needs

Producing an Audit Trail

Audits of records, en masse or individually, demand the ability to readily construct data as it was at any or every point in time

Digging through all versions is impractical yet required to meet compliance requirements



What Delta Change Data Feed Does for You

Improve ETL pipelines

Process less data during ETL to increase efficiency of your pipelines

Unify batch and streaming

Common change format for batch and streaming updates, appends, and deletes

BI on your data lake

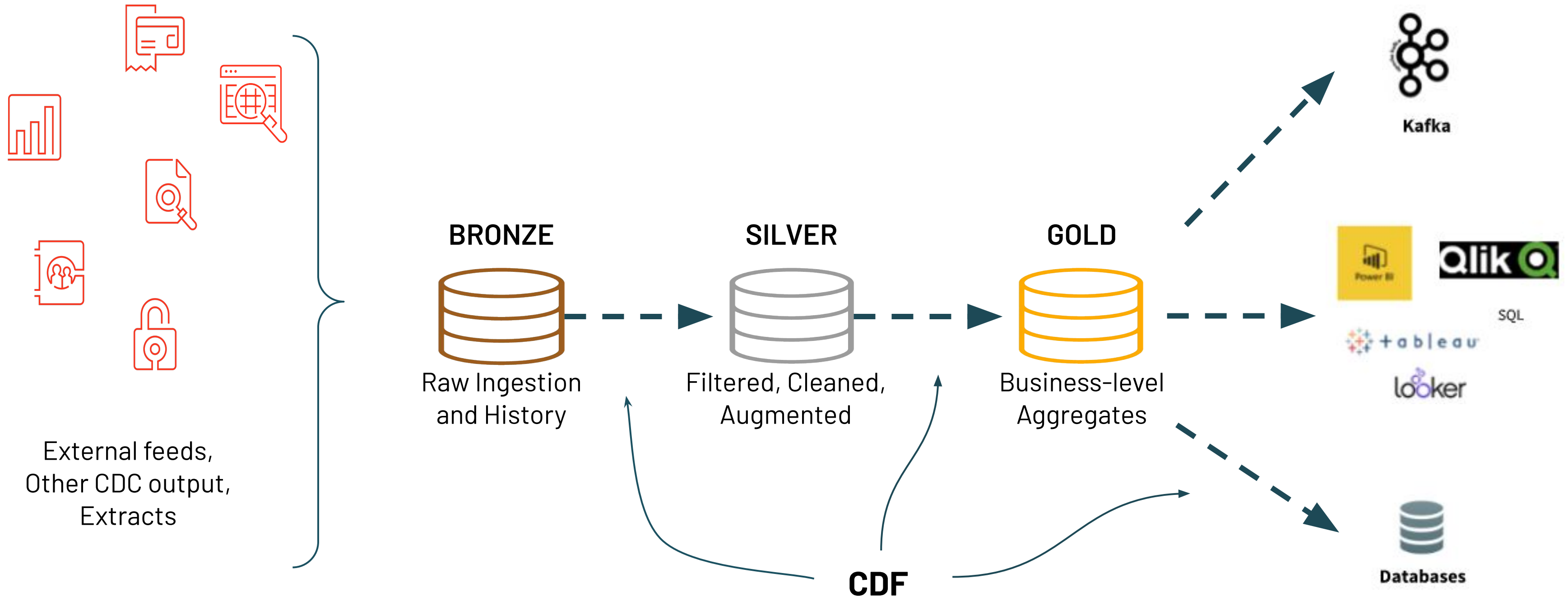
Incrementally update the data supporting your BI tool of choice

Meet regulatory needs

Full history available of changes made to the data, including deleted information

Delta Change Data Feed

Where Delta Change Data Feed Applies



How Does Delta Change Data Feed Work?

Original Table (v1)

PK	B
A1	B1
A2	B2
A3	B3

Change data (Merged as v2)

PK	B
A2	Z2
A3	B3
A4	B4

Change Data Feed Output

PK	B	Change Type	Time	Version
A2	B2	Preimage	12:00:00	2
A2	Z2	Postimage	12:00:00	2
A3	B3	Delete	12:00:00	2
A4	B4	Insert	12:00:00	2

A1 record did not receive an update or delete. So it will not be output by CDF.

Typical Use Cases

Silver & Gold Tables

Improve Delta performance by processing only changes following initial MERGE comparison to accelerate and simplify ETL/ELT operations

Materialized Views

Create up-to-date, aggregated views of information for use in BI and analytics without having to reprocess the full underlying tables, instead updating only where changes have come through

Transmit Changes

Send Change Data Feed to downstream systems such as Kafka or RDBMS that can use it to incrementally process in later stages of data pipelines

Audit Trail Table

Capturing Change Data Feed outputs as a Delta table provides perpetual storage and efficient query capability to see all changes over time, including when deletes occur and what updates were made

When to Use Delta Change Data Feed

✓	✗
Delta changes include updates and/or deletes	Delta changes are append only
Small fraction of records updated in each batch	Most records in the table updated in each batch
Data received from external sources is in CDC format	
	Data received comprises destructive loads
Send data changes to downstream application	Find and ingest data outside of the Lakehouse

Notebook: Processing Records from Change Data Feed

databricks

Streaming Joins and Statefulness

databricks

When in processing time are results materialized?

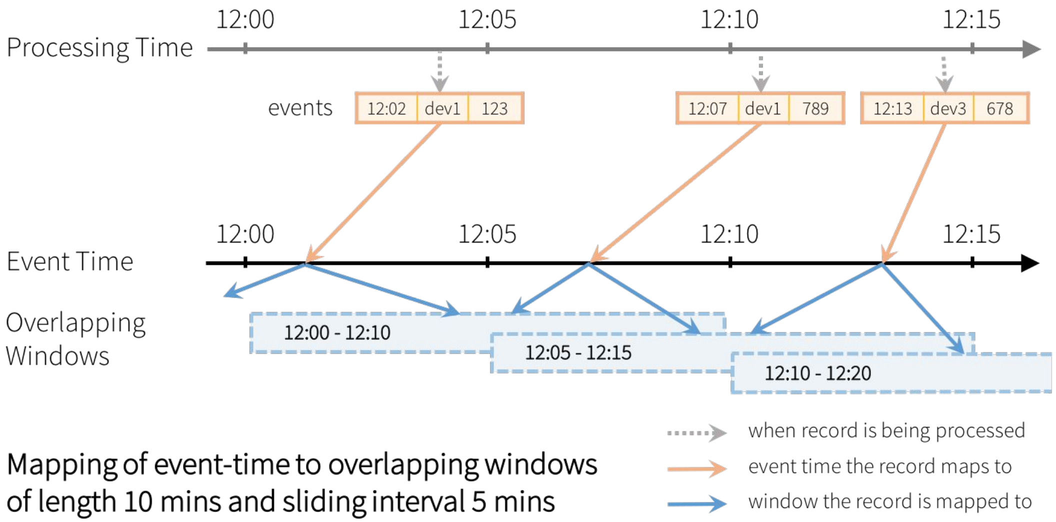
Triggers

- Unspecified (default)
  - start a new mini-batch after the last one completed
- Fixed interval
  - start a new mini-batch after the last one completed and x time has passed
- One-time (trigger once)
  - run a mini-batch once and then shut down the stream

When in processing time are results materialized?

Event Time Windowing + Trigger Example

```
windowedAvgSignalDF =
  (eventsDF
   .groupBy(window("eventTime",
                   "10 minutes",
                   "5 minutes"))
   .count()
   .writeStream
   .trigger(processingTime="5 minutes")
   )
```



Mapping of event-time to overlapping windows of length 10 mins and sliding interval 5 mins

databricks

When in processing time are results materialized?

How do we know when we can no longer expect new data for a certain event time window? How do we handle late data?

We need Watermarks

Mark the most recent event time seen, and discard late data after a configurable lateness threshold

Note: In Structured Streaming there is only a one-sided guarantee to not drop data within the lateness threshold. Late data might still be processed based on when the watermark is advanced.

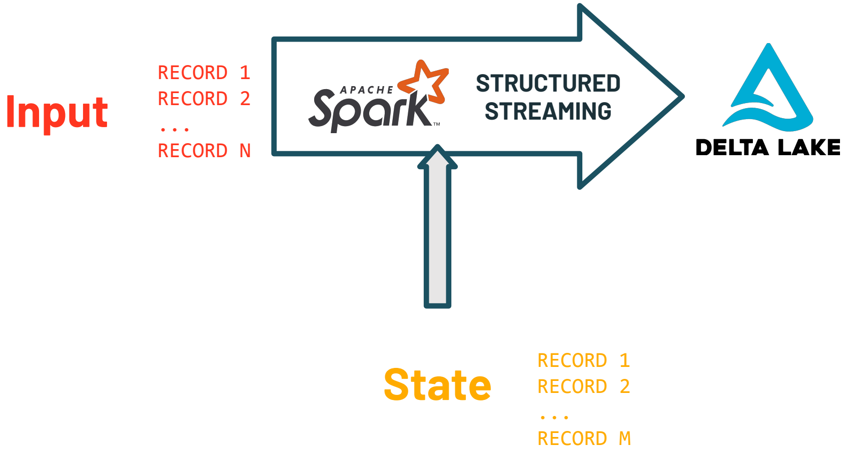
When in processing time are results materialized?

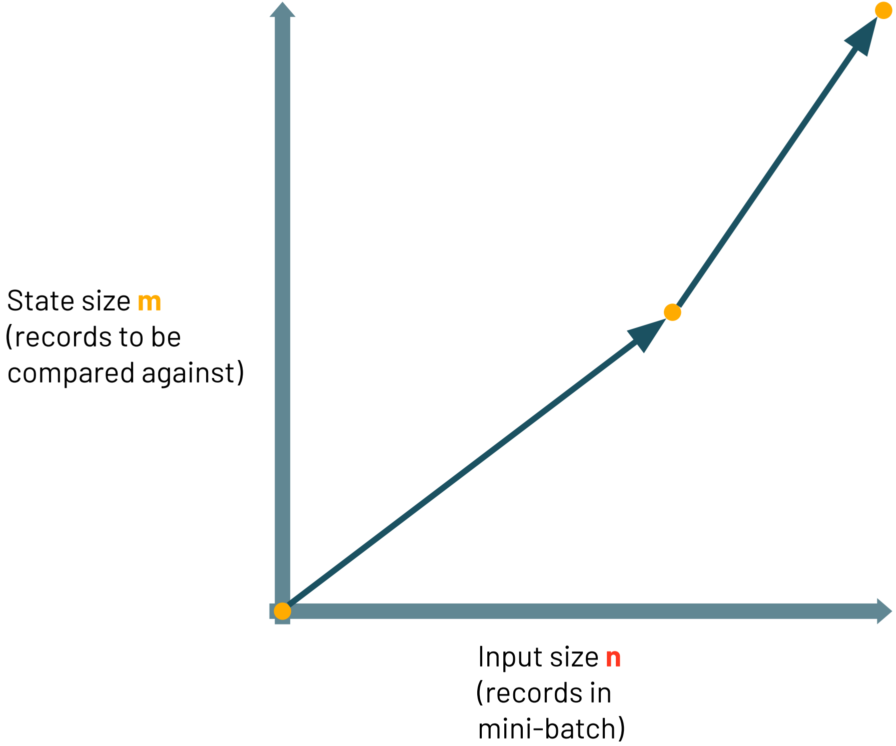
What and when do we exactly output to the sink of the stream?

We need to choose the appropriate Output Mode.

- Complete (materializes every trigger, outputs complete table)
- Update (materializes every trigger, outputs only new values)
- Append (only materializes after watermark + lateness passed)

Scale dimensions

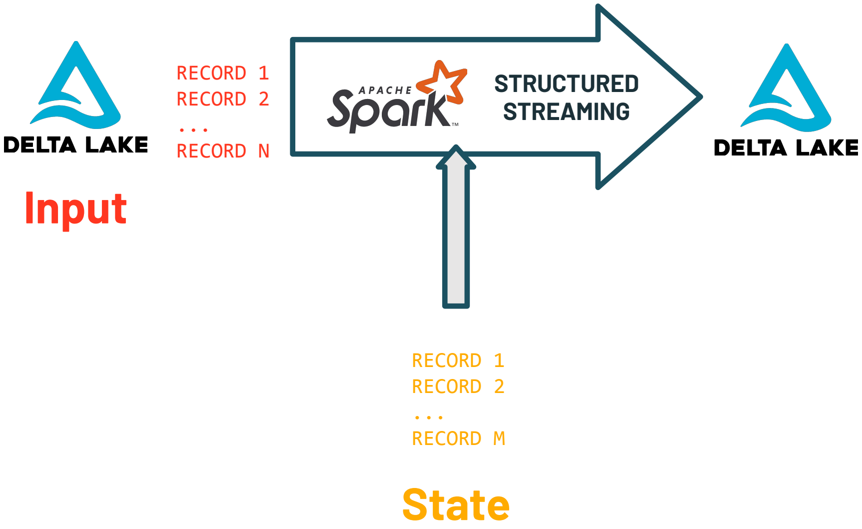




How do we correctly tune this?

databricks

Let’s use this example!



1. Main input stream

```
salesSDF = (
  spark
    .readStream
    .format("delta")
    .table("sales")
)
```

2. Join item category lookup

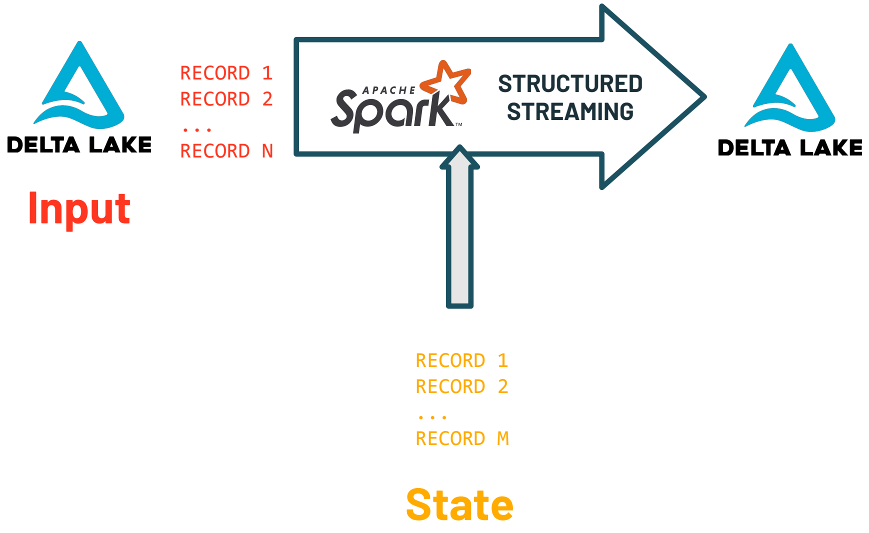
```
itemSalesSDF = (
  salesSDF
    .join( spark.table(“items”), “item_id”)
)
```

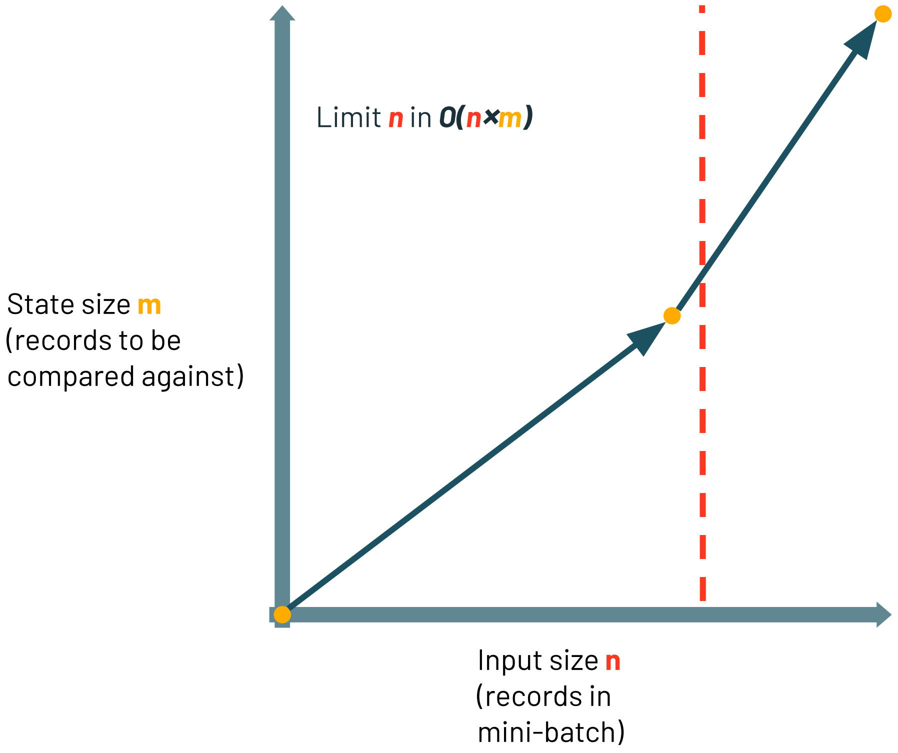
3. Aggregate sales per item category per hour

```
itemSalesPerHourSDF = (
  itemSalesSDF
    .groupBy(window(..., “1 hour”),
             “item_category”)
    .sum(“revenue”)
)
```

Input Parameters

Limiting the input dimension





Why are input parameters important?

- Allows you to control the mini-batch size.

- Optimal mini-batch size → Optimal cluster usage.

- Suboptimal mini-batch size → performance cliff.
    - Shuffle Spill
    - Different Query Plan (Sort Merge Join vs Broadcast Join)

What input parameters are we talking about?

File Source

- Any: maxFilesPerTrigger
- Delta Lake:

  +maxBytesPerTrigger

Message Source

- Kafka: maxOffsetsPerTrigger
- Kinesis: fetchBufferSize
- EventHubs:

  maxEventsPerTrigger

- Controls the size of each mini-batch
- Especially important in relation to shuffle partitions

Input Parameters Example: Stream-Static Join

What is a Stream-Static join?

- Joining a streaming df to a static df
- Induces a shuffling step.

1. Main input stream

```
salesSDF = (
  spark
    .readStream
    .format("delta")
    .table("sales")
)
```

2. Join item category lookup

```
itemSalesSDF = (
  salesSDF
    .join( spark.table(“items”), “item_id”)
)
```

Input Parameters: Not tuning maxFilesPerTrigger

What will happen when not setting maxFilesPerTrigger?

- For Delta: Default option is 1000 files. Each file is ~200 MB.
  - For Message and other File based input: Default option is unlimited.
- Leads to a massive mini-batch!
- When you have shuffle operations → Spill.

Input Parameters: Tuning maxFilesPerTrigger

Base it on shuffle partition size

- **Rule of thumb 1:** Optimal shuffle partition size ~100-200 MB
- **Rule of thumb 2:** Set shuffle partitions equal to # of cores = 20.
- Use Spark UI to tune `maxFilesPerTrigger` until you get ~100-200 MB per partition.
- Note: Size on disk is **not** a good proxy for size in memory
  - Reason is that file size is different from the size in cluster memory

Sort Merge Join vs Broadcast Hash Join

We are not done yet!

- Currently we use a Sort Merge Join.
- Our static DF is small enough to **broadcast** it.
- Leads to 70% increased throughput!
- Can also increase `maxFilesPerTrigger`
- Because of no more risk of Shuffle Spill (shuffles were removed)

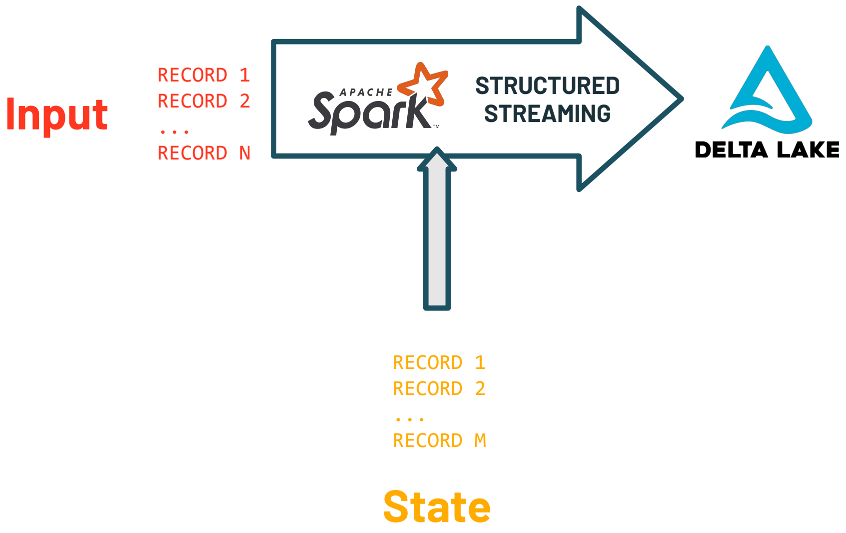
Input Parameters: Summary

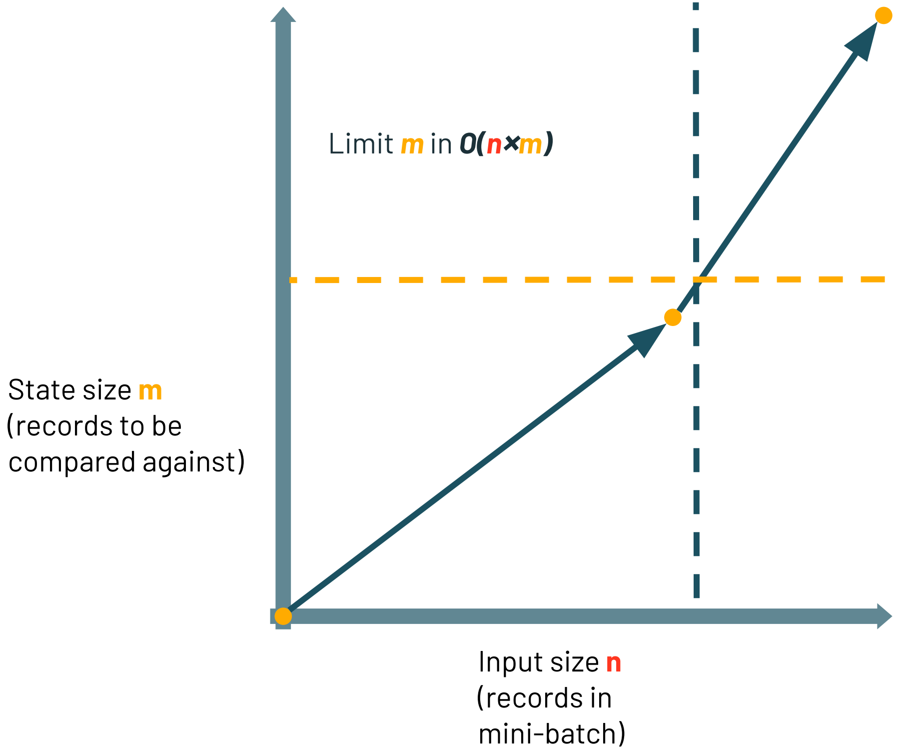
Main takeaways

- Set shuffle partitions to # Cores (assuming no skew)
- Tune `maxFilesPerTrigger` so you end up with 150-200 MB / Shuffle Partition
- Try to make use of broadcasting whenever possible

State Parameters

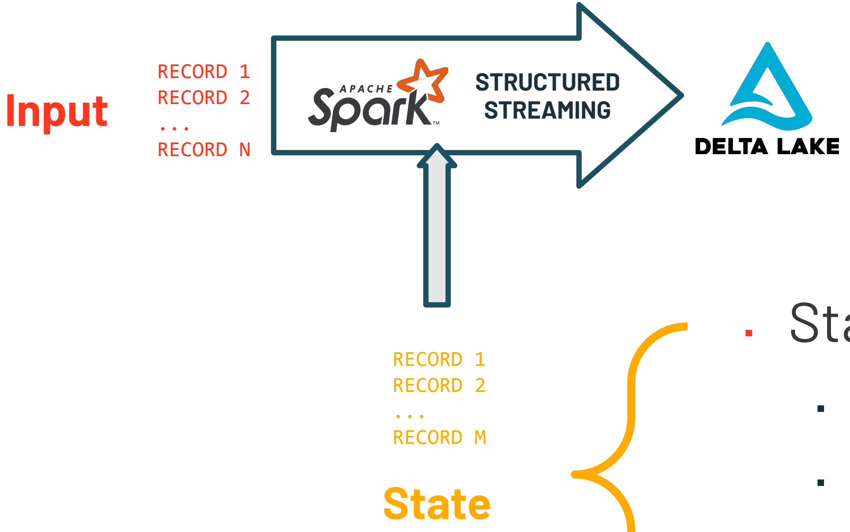
Limiting the state dimension





Limiting the state dimension

What we mean by **state**



- State Store backed operations
  - Stateful (windowed) aggregations
  - Drop duplicates
  - Stream-Stream Joins
- Delta Lake table or external system
  - Stream-Static Join / MERGE

Why are state parameters important?

- Optimal parameters → Optimal cluster usage

- If not controlled, state explosion can occur
  - Slower stream performance over time
  - Heavy shuffle spill (Joins/MERGE)
  - Out of memory errors (State Store backed operations)

What parameters are we talking about?

State Store specific

- How much history to compare against (watermarking)
- What state store backend to use (RocksDB / Default)

State Store agnostic

(Stream-Static Join / MERGE)

- How much history to compare against (query predicate)

databricks

State parameters example

- Extending the earlier code sample with stateful aggregation
- E.g. Calculating the number of sales per item category per hour
- Two types of state dimension here:
  a. Static side of the stream-static join (items)
  b. State Store backed operation (windowed stateful aggregation)

1. Main input stream

```
salesSDF = (
  spark
    .readStream
    .format("delta")
    .table("sales")
)
```

2. Join item category lookup

```
itemSalesSDF = (
  salesSDF
    .join( spark.table("items"), "item_id")
)
```

3. Aggregate sales per item per hour

```
itemSalesPerHourSDF = (
  itemSalesSDF
    .groupBy(window(..., "1 hour"),
             "item_category")
    .sum("revenue")
)
```

databricks

State Parameters: Summary

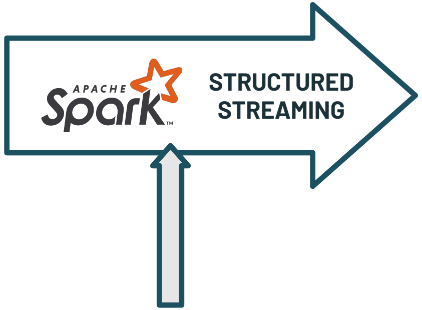
- Limit state accumulation with appropriate watermark
- The more granular the aggregate key / window, the more state
- Delta Backed State might provide more flexibility at cost of latency

Output Parameters

How output parameters influence the scale dimensions

Input

RECORD 1
RECORD 2
...
RECORD N

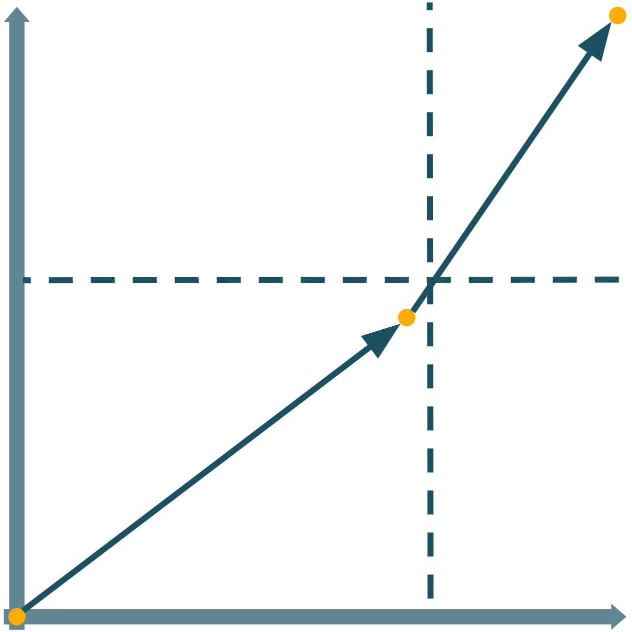


DELTA LAKE

RECORD 1
RECORD 2
...
RECORD M

State

State size **m**
(records to be compared against)



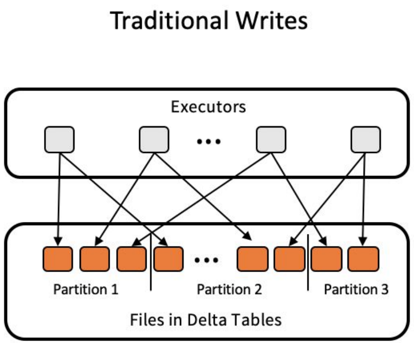
Input size **n**
(records in mini-batch)

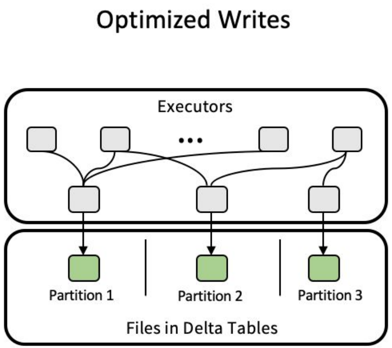
Why are output parameters important?

- Streaming jobs tend to create many small files
- Reading a folder with many small files is slow
- Degrading performance for downstream jobs / self-joins

What Output parameters are we talking about?

- Manually using repartition
- Delta Lake: Auto-Optimize





https://docs.databricks.com/delta/optimizations/auto-optimize.html

Output Parameters: Summary

Main takeaways

- High number of files impact performance
- **10x** speed difference can easily be demonstrated

Notebook: Stream Static Joins

databricks

Lakehouse and the Query Layer

databricks

What is the Query Layer?

- Stores refined datasets for use by data scientists
- Serves results for pre-computed ML models
- Contains enriched, aggregated views for use by analysts
- Powers data-driven applications, dashboards, and reports

Also called the serving layer; gold tables exist at this level.

Tables and Views in the Query Layer

- Gold tables
- Saved views
- Databricks SQL saved queries
- Tables in RDS/NoSQL database

Gold Tables

- Refined, typically aggregated views of data saved to memory using Delta Lake
- Can be updated with batch or stream processing
- Configured and scheduled as part of ETL workloads
- Results computed on write
- Read is simple deserialization; additional filters can be applied with pushdowns

Saved Views

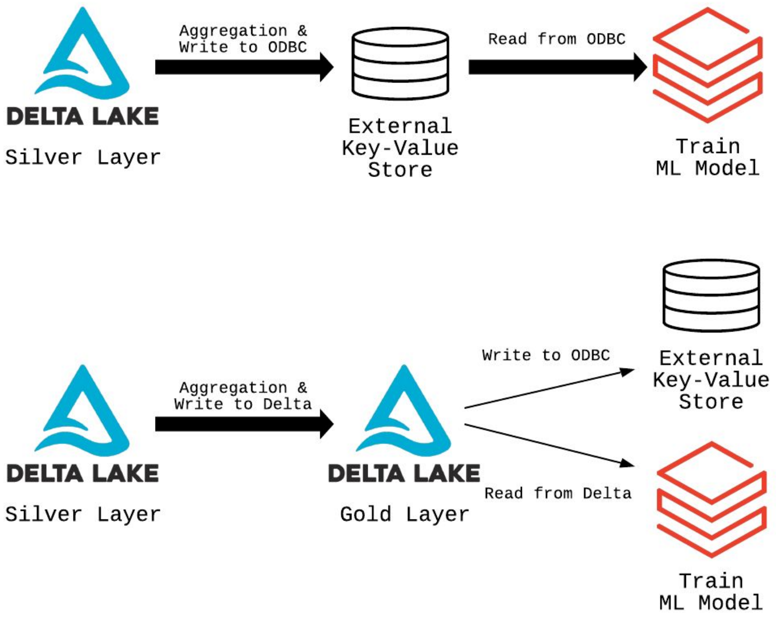
- Views can be registered to databases and made available to users using ACLs
- Views are logical queries against source tables
- Logic is executed each time a view is queried
- Views registered against Delta tables will always query the most current valid version of the table

Databricks SQL Saved Queries

- Similar to saved views in when logic is executed
- Auto-detect changes in upstream Delta tables
- Uses new feature Query Result Cache
- Caching allows reviewing dashboards and downloading CSVs without an active SQL endpoint
- Easy to identify data sources (SQL present in saved query)
- Can be scheduled using Databricks SQL functionality
- Can automatically refresh Databricks SQL dashboards

Tables in External Systems

- Many downstream applications may require refined data in a different system
  - NoSQL databases
  - RDS
  - Pub/sub messaging
- Must decide where single source of truth lives



Recommendations

- Use saved views when filtering silver tables

```
CREATE VIEW sales_florida_2020 AS
  SELECT *
  FROM sales
  WHERE state = ‘FL’ and year = 2020;
```

Recommendations

- Use Delta tables for common partial aggregates

```
CREATE TABLE store_item_sales AS
  SELECT store_id, item_id, department, date,
    city, state, region, country,
    SUM(quantity) AS items_sold,
    SUM(price) AS revenue
  FROM sales
    INNER JOIN stores ON sales.store_id = stores.store_id
    INNER JOIN items ON sales.item_id = items.item_id
  GROUP BY store_id, item_id, department, date,
    city, state, region, country
```

Recommendations

- Share Databricks SQL queries and dashboards within teams

```
SELECT date, hour, SUM(quantity * price) hourly_sales
  FROM sales
  WHERE store_id = 42
  AND date > date_sub(current_date(), 14)
  GROUP BY date, hour;
```

Recommendations

- Analyze query history to identify new candidate gold tables
  - Admins can access Databricks SQL query history
  - Running analytics against query history can help identify
    - Queries that are long-running
    - Queries that are run regularly
    - Queries that use common datasets
- Transitioning these queries to gold tables and scheduling as engineering jobs may reduce total operating costs
- Query history can also be useful for identifying predicates used most frequently; useful for ZORDER indexing during optimization

Notebook: Stored Views

Notebook: Materialized Views

databricks

databricks