Managing Data in Motion

databricks

Module Objectives

1. Use clone and Auto Loader for efficient copies and incremental ingestion
2. Use native features in Spark and Delta Lake to deduplicate, enrich, and validate incremental data
3. Combine traditional data modeling techniques with Databricks features to propagate updates through the Lakehouse
4. Implement solutions that correctly manage dependencies between incremental datasets

Agenda

- Clone for Development and Data Backup
- Auto Loader
- Bronze Ingestion Patterns
- Promoting Bronze to Silver
- Streaming Deduplication
- Quality Enforcement
- Slowly Changing Dimensions in the Lakehouse
- Streaming Joins and Statefulness
- Stream-static Joins

Clone for Development and Data Backup

databricks

Clones:

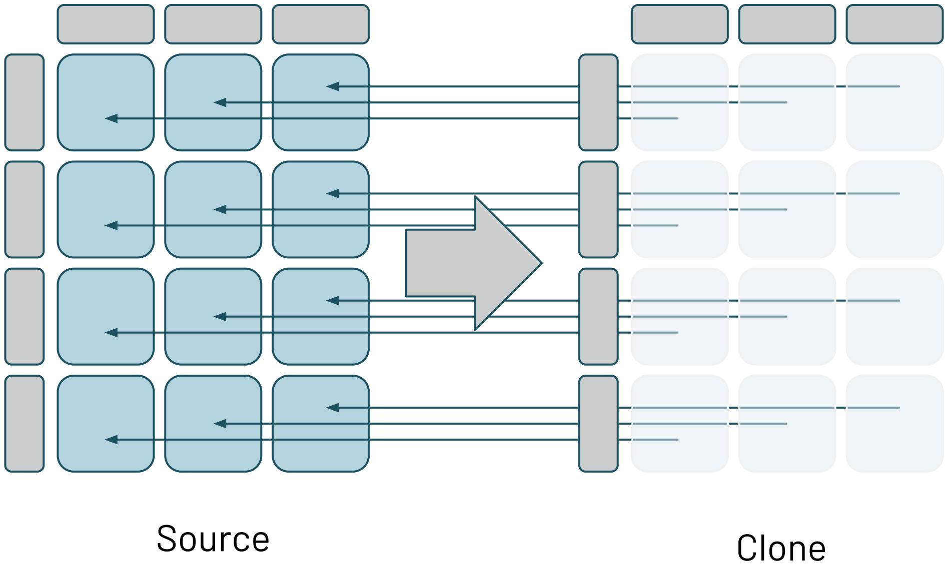
- Create a replica of a target table
- At a point in time
- In a specific destination location

Basic Details

- Metadata is replicated
  - Schema
  - Constraints
  - Column descriptions
  - Statistics
  - Partitioning
- Clones have separate lineage
  - Changes to cloned table due not affect the source
  - Changes to the source during or after cloning are not reflected in the clone

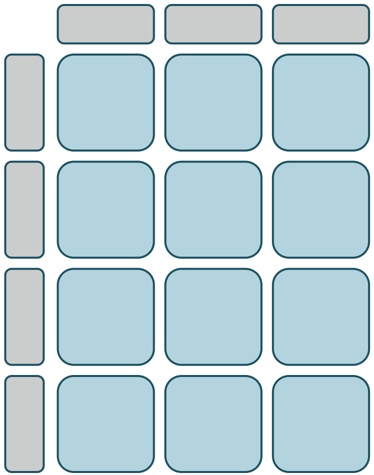
Shallow Clones

- Zero-copy cloning
  - Only metadata is copied
  - Points to original data files
- Inexpensive and fast
- Not self-contained
  - Depend on sourced data files
  - If source data files are removed, shallow clone may break

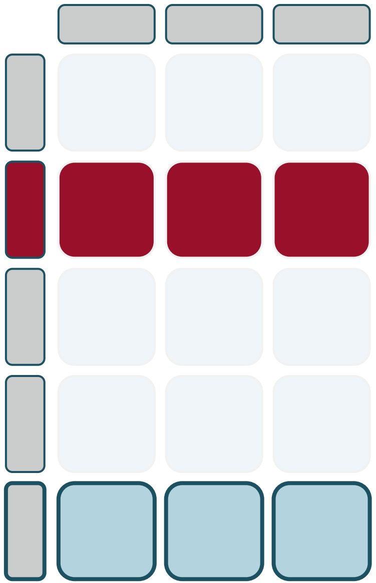


Making Changes to the Shallow Clone

- Inserts to the cloned table write new data files
  - Files are recorded in the cloned table directory
- Updates, deletes, and optimizations also write new data files
  - Allows for easy testing without risking prod data



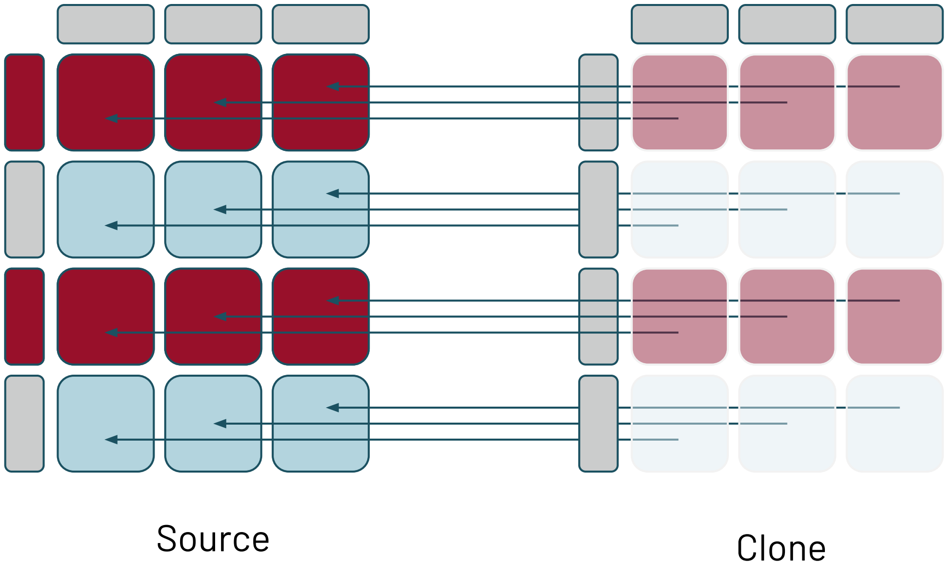
Source



Clone

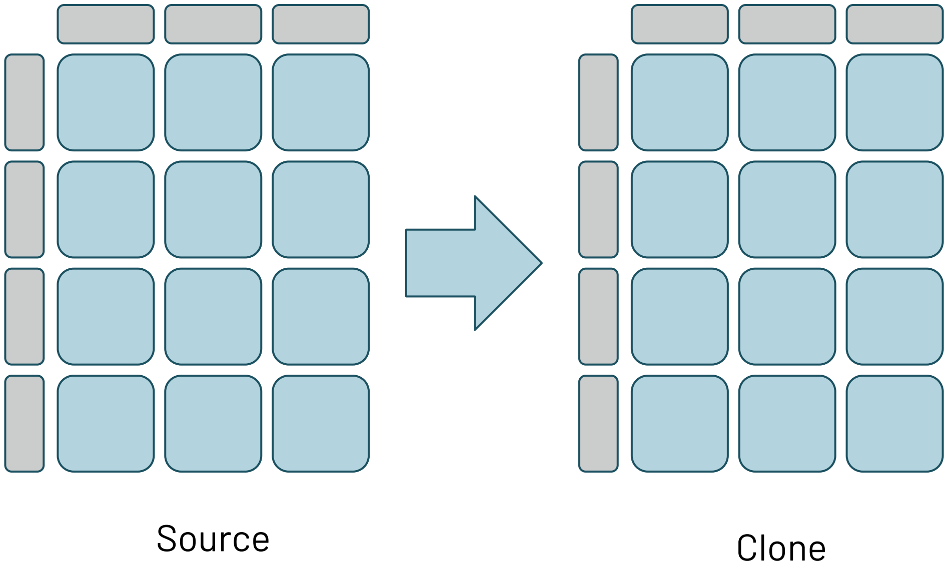
Removing Source Files

- Changes to the source table mark data files as no longer valid
- Vacuuming the source table will permanently remove these data files
- References to source data files will cause queries on the clone table to fail



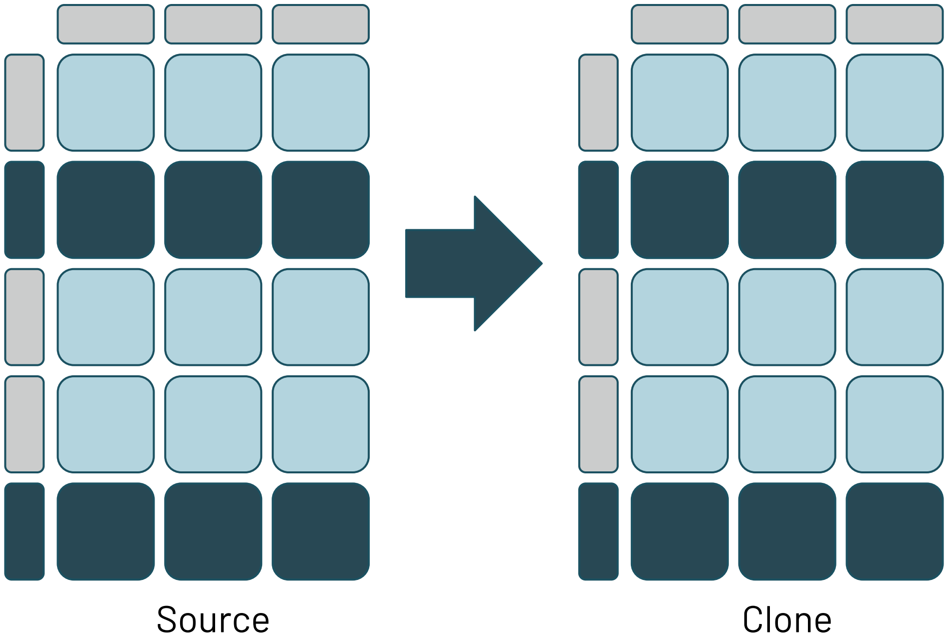
Deep Clones

- Data is copied alongside metadata
- Copy is optimized, transactional, and robust
- Incrementally copies data files



Incremental Cloning

- Only newly written data files are copied
- Updates, deletes, and appends are automatically applied
- Data files will be identical in both tables after cloning



History and Time Travel

- Clones have separate versioning
  - History begins at version 0
  - New version recorded with updates (including incremental clone)
  - Metadata tracks source table version
- Clones can have separate retention settings
  - Delta Lake default settings are tuned for performance
  - Increase log retention and deleted file retention for archiving
  - Clone copies source table properties, so will need to reset after each incremental clone
  - If you vacuum the source, a shallow clone is impacted but a deep clone is not.
  - If you vacuum a clone, the original is not impacted.

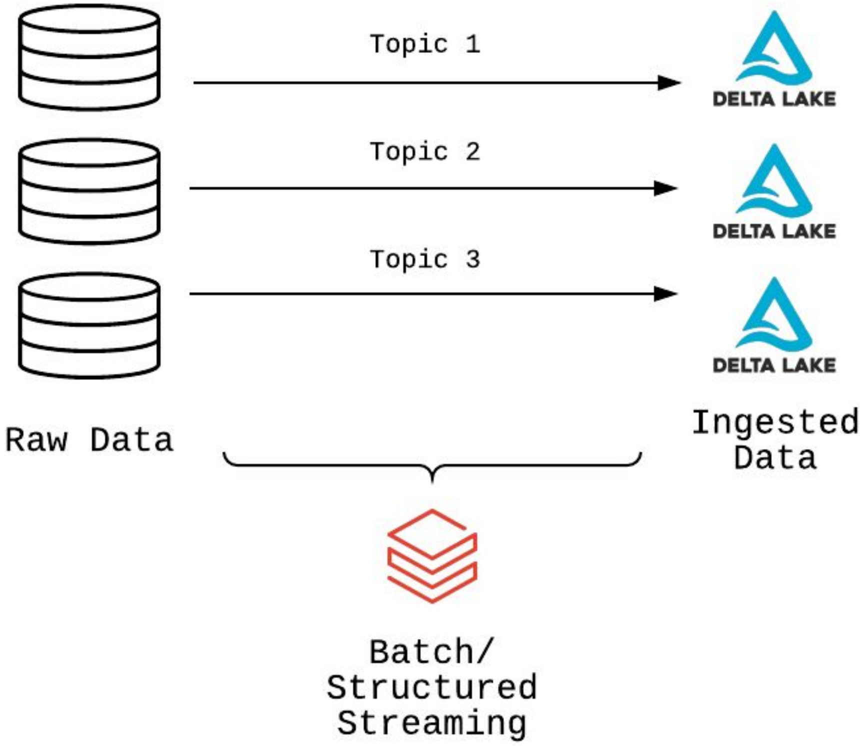
Notebook: Using Clone with Delta Lake

Notebook: Auto Loader

databricks

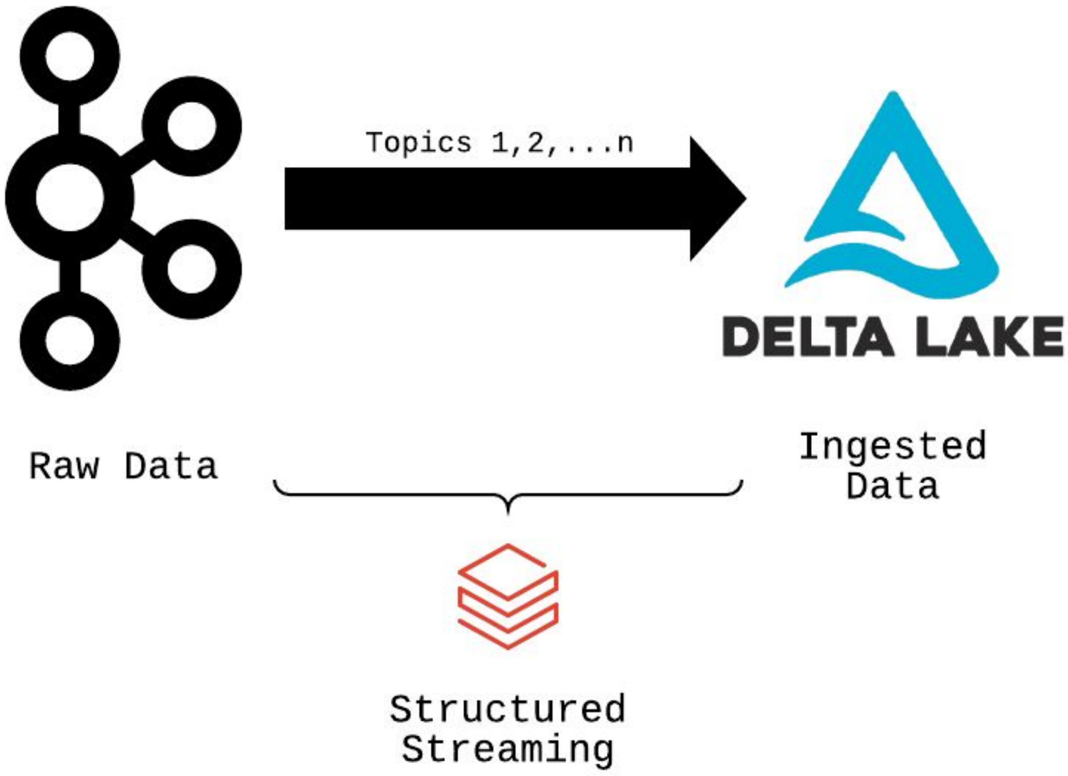
Bronze Ingestion Patterns

Singleplex Ingestion



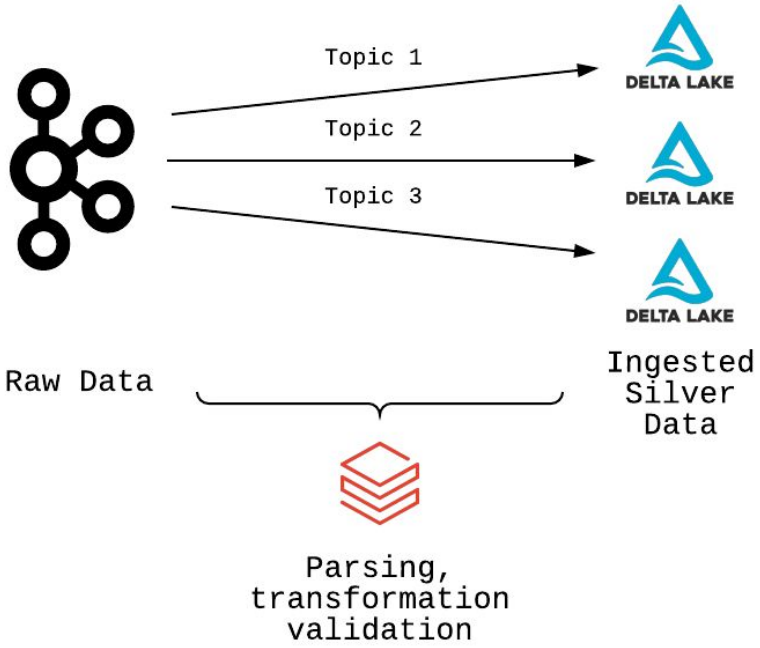
databricks

Multiplex Ingestion

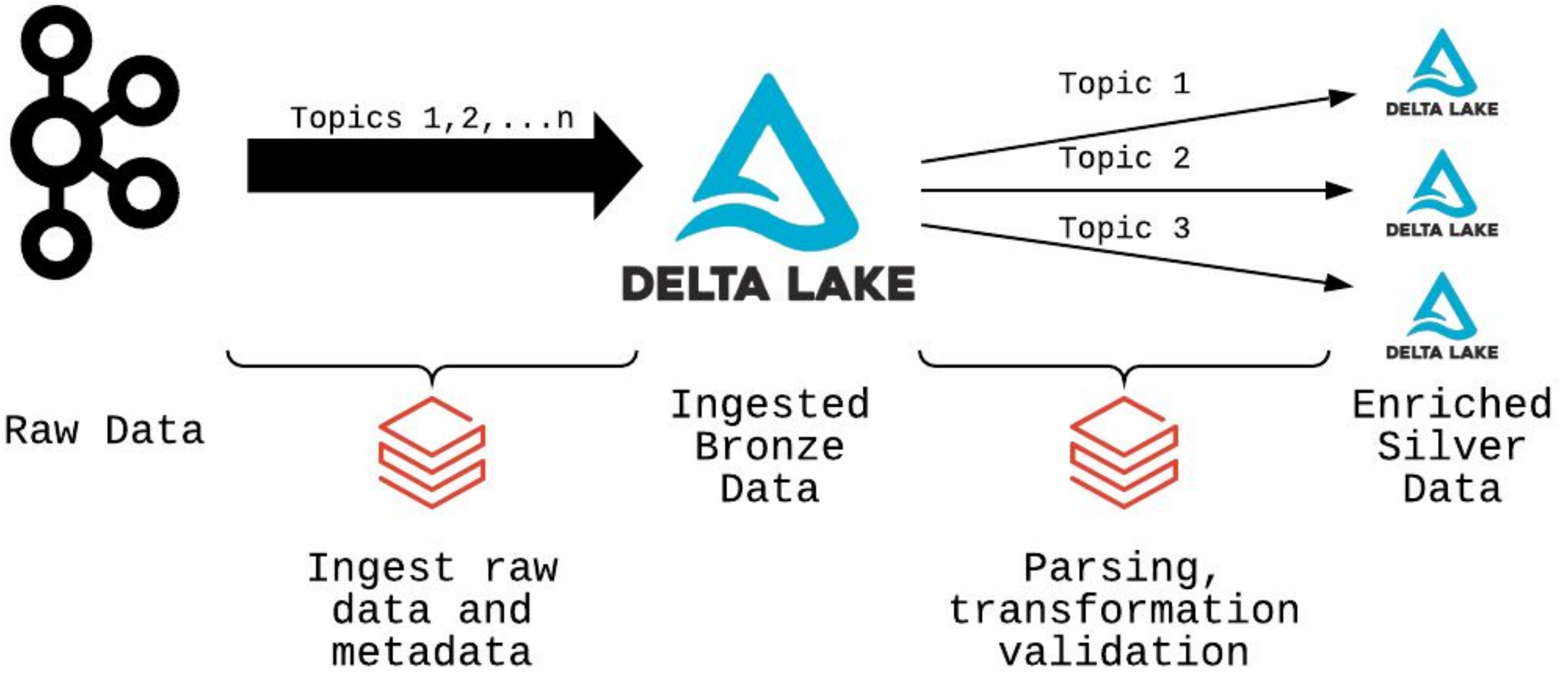


Don't Use Kafka as Bronze

- Data retention limited by Kafka; expensive to keep full history
- All processing happens on ingest
- If stream gets too far behind, data is lost
- Cannot recover data (no history to replay)



Delta Lake Bronze



Notebook: Auto Load into Multiplex Bronze

databricks

Promoting Bronze to Silver

Silver Layer Objectives

- Validate data quality and schema
- Enrich and transform data
- Optimize data layout and storage for downstream queries
- Provide single source of truth for analytics

Schema Enforcement & Evolution

- Enforcement prevents bad records from entering table
  - Mismatch in type or field name
- Evolution allows new fields to be added
  - Useful when schema changes in production/new fields added to nested data
  - **Cannot** use evolution to remove fields
  - All previous records will show newly added field as Null
    - For previously written records, the underlying file isn't modified.
    - The additional field is simply defined in the metadata and dynamically read as null

Delta Lake Constraints

- Check `NOT NULL` or arbitrary boolean condition
- Throws exception on failure

```
ALTER TABLE tableName ADD CONSTRAINT constraintName
    CHECK heartRate >= 0;
```

Alternative Quality Check Approaches

- Add a “validation” field that captures any validation errors and a null value means validation passed.
- Quarantine data by filtering non-compliant data to alternate location
- Warn without failing by writing additional fields with constraint check results to Delta tables

Notebook: Streaming from Multiplex Bronze

databricks

Notebook: Streaming Deduplication

Notebook: Quality Enforcement

Notebook: Promoting to Silver

Slowly Changing Dimensions in the Lakehouse

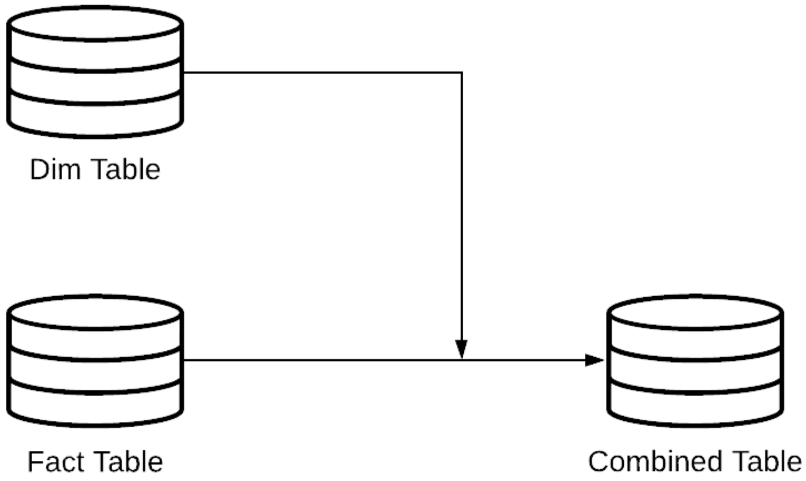
databricks

Fact Tables as Incremental Data

- Often is a time series
- No intermediate aggregations
- No overwrite/update/delete operations
- Append-only operations

Using Dimension Tables in Incremental Updates

- Delta Lake enables stream-static joins
- 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.

Type 0 and Type 1

user_id	street	name
1	123 Oak Ave	Sam
2	99 Jump St	Abhi
3	1000 Rodeo Dr	Kasey

Type 2

user_id	street	name	valid_from	current
1	123 Oak Ave	Sam	2020-01-01	true
2	99 Jump St	Abhi	2020-01-01	false
3	1000 Rodeo Dr	Kasey	2020-01-01	false
2	430 River Rd	Abhi	2021-10-10	true
3	1000 Rodeo Dr	Casey	2021-10-10	true

Applying SCD Principles to Facts

- Fact table usually append-only (Type 0)
- Can leverage event and processing times for append-only history

order_id	user_id	occurred_at	action	processed_time
123	1	2021-10-01 10:05:00	ORDER_CANCELLED	2021-10-01 10:05:30
123	1	2021-10-01 10:00:00	ORDER_PLACED	2021-10-01 10:06:30

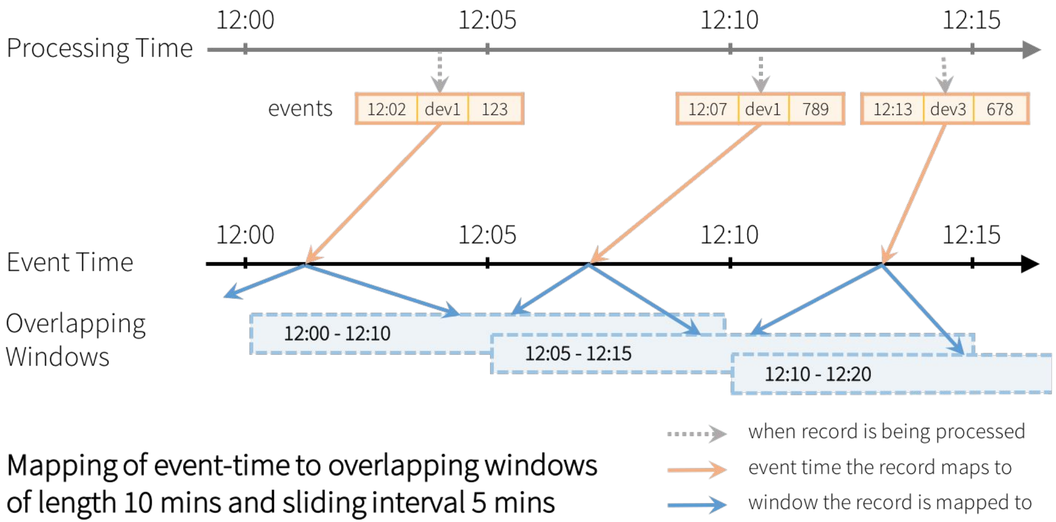
Notebook: Type 2 SCD

databricks

Streaming Joins and Statefulness

The Components of a Stateful Stream

```
windowedDF =
  (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

Output Modes

Mode	When Stateful Results Materialize
Append (default)	Only materialize after watermark + lateness passed
Complete	Materialize every trigger, outputs complete table
Update	Materialize every trigger, outputs only new values

Statefulness vs. Query Progress

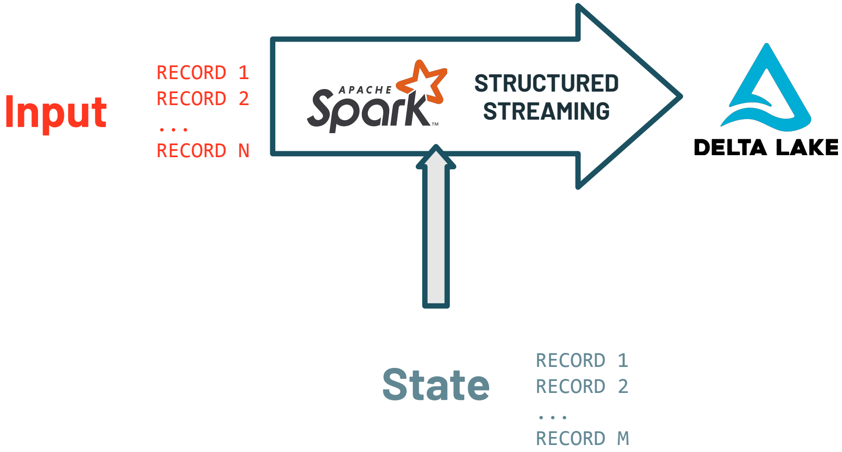
- Many operations as specifically stateful (stream-stream joins, deduplication, aggregation)
- Some operations just need to store incremental query progress and are not stateful (appends with simple transformations, stream-static joins, merge)
- Progress and state are stored in checkpoints and managed by driver during query processing

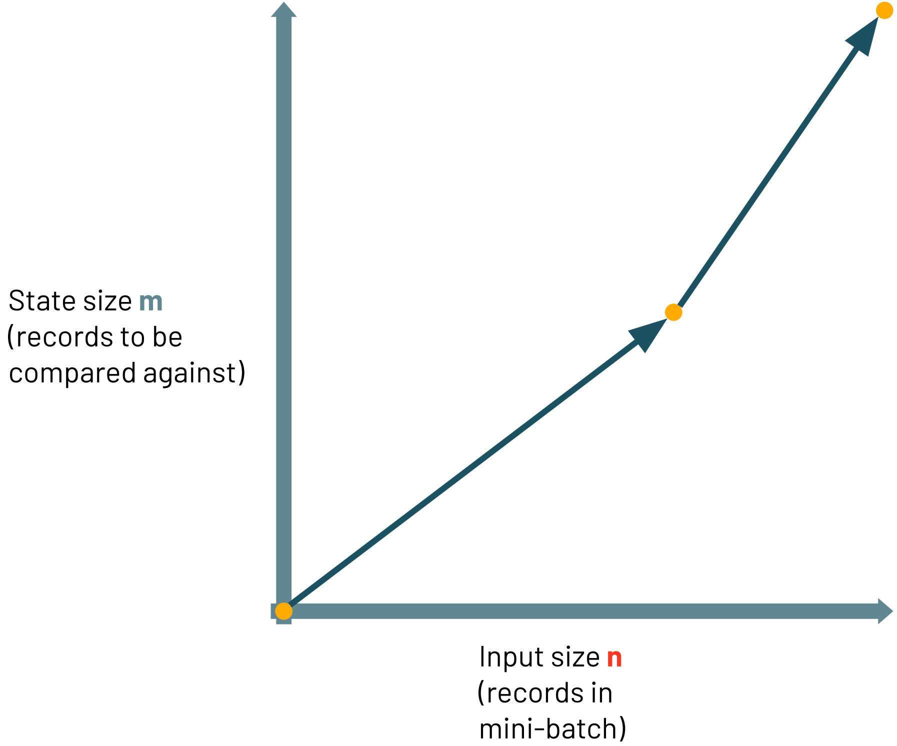
Managing Stream Parameters

GOAL: Balance parameters for sustainable, optimized throughput

- Input Parameters
  - Control amount of data in each micro-batch
- State Parameters
  - Control amount of data required to calculate query results
- Output Parameters
  - Control number and size of files written

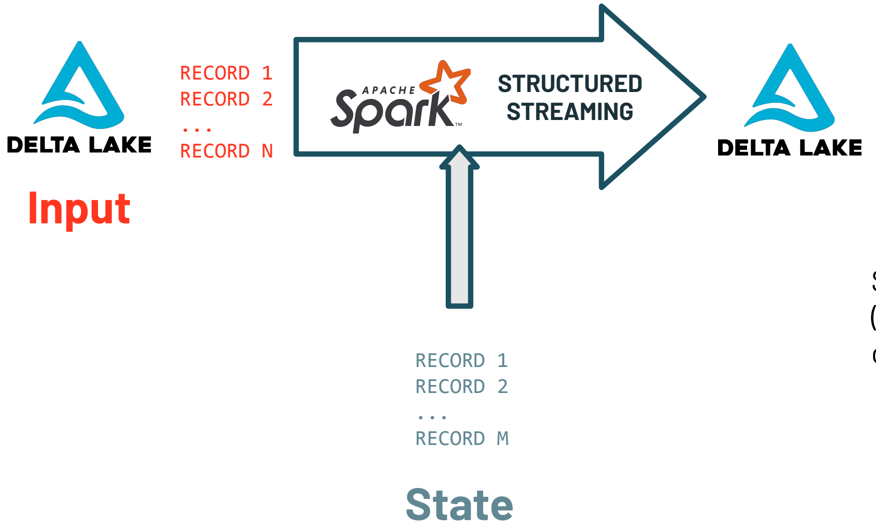
Reasoning about Stream Dimensions

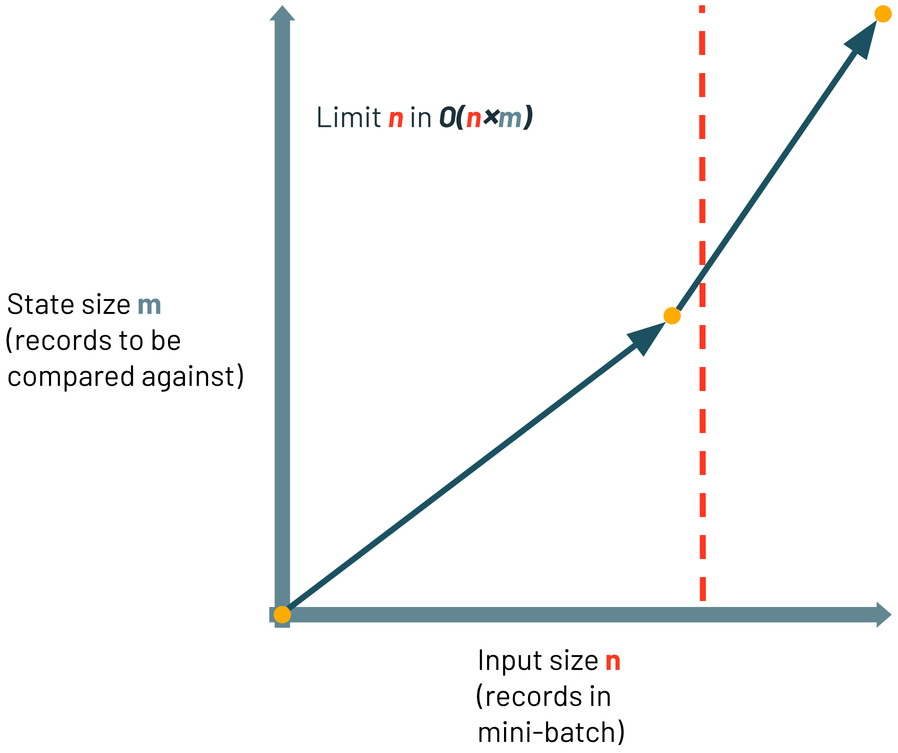




Input Parameters

Limiting the input dimension





Why are input parameters important?

- Allows you to control the mini-batch size
- Defaults are large
  - Delta Lake: 1000 files per micro-batch
  - Pub/Sub & files: No limit to input batch size
- Optimal mini-batch size → Optimal cluster usage
- Suboptimal mini-batch size → performance cliff
  - Shuffle Spill

Per Trigger Settings

- File Source
  - maxFilesPerTrigger
- Delta Lake and Auto Loader
  - maxFilesPerTrigger
  - maxBytesPerTrigger
- Kafka
  - maxOffsetsPerTrigger

Shuffle Partitions with Structured Streaming

- Should match the number of cores in the largest cluster size that might be used in production
  - Number of shuffle partitions == max parallelism
- Cannot be changed without new checkpoint
  - Will lose query progress and state information
- Higher shuffle partitions == more files written
- Best practice: use Delta Live Tables for streaming jobs with variable volume

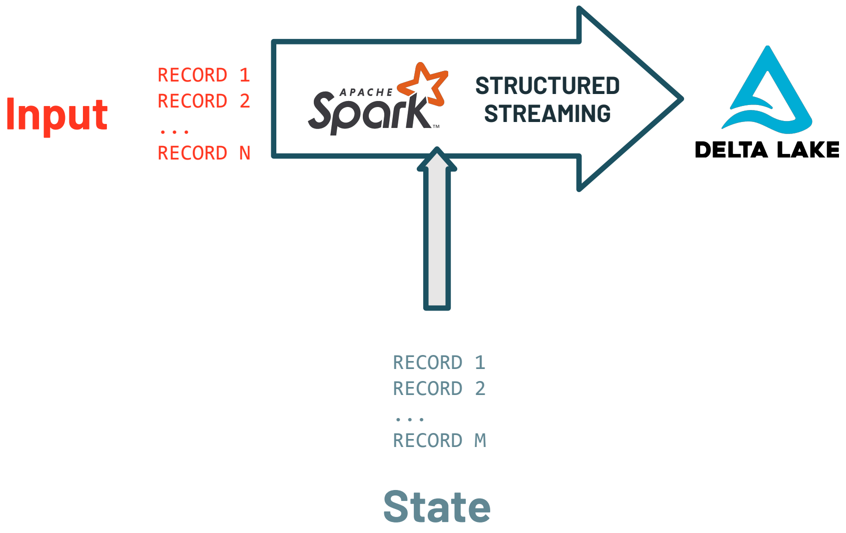
Tuning maxFilesPerTrigger

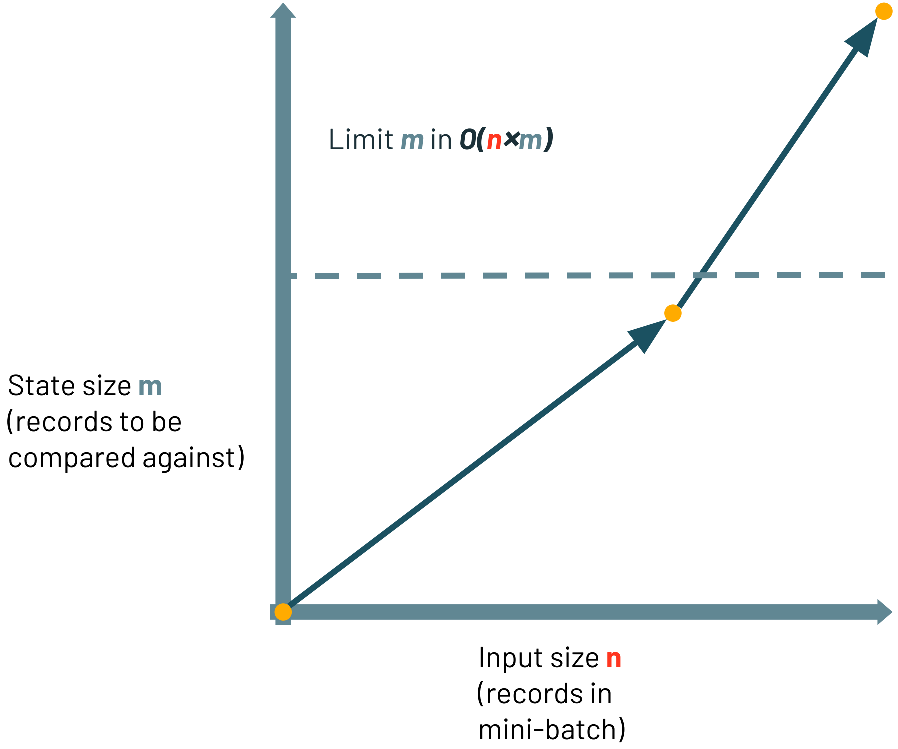
Base it on shuffle partition size

- **Rule of thumb 1:** Optimal shuffle partition size ~100-200 MB
- **Rule of thumb 2:** Set shuffle partitions equal to # of cores
- 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

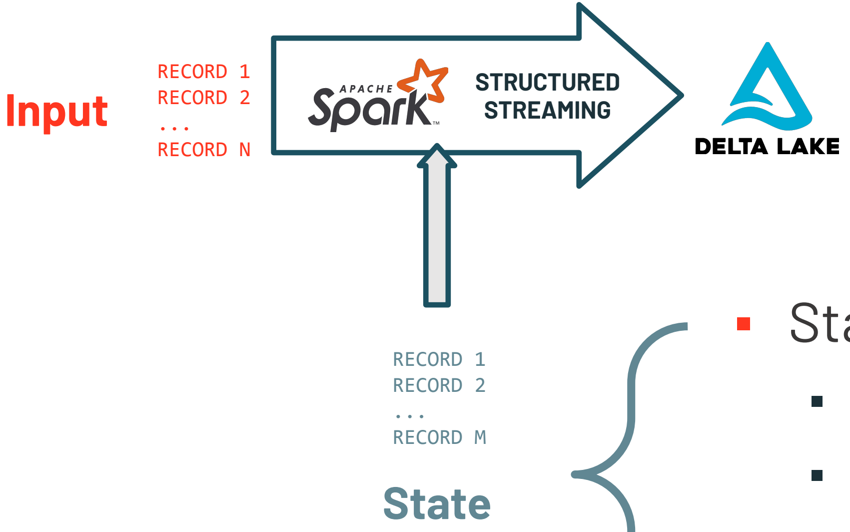
State Parameters

Limiting the state dimension





Limiting the state dimension



- 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)

Example Query

- Static Delta Lake table used in stream-static join
- State Store-backed 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”)
)
```

State Store Parameters

- Watermarking
  - How much history to compare against
- Granularity
  - The more granular the aggregate key / window, the more state
- State store backend
  - RocksDB / Default

Stream-Static Join & Merge

- Join driven by streaming data
- Join triggers shuffle
- Join itself is stateless
- Control state information with predicate
- Goal is to broadcast static table to streaming data
- Broadcasting puts all data on each node

1. Main input stream

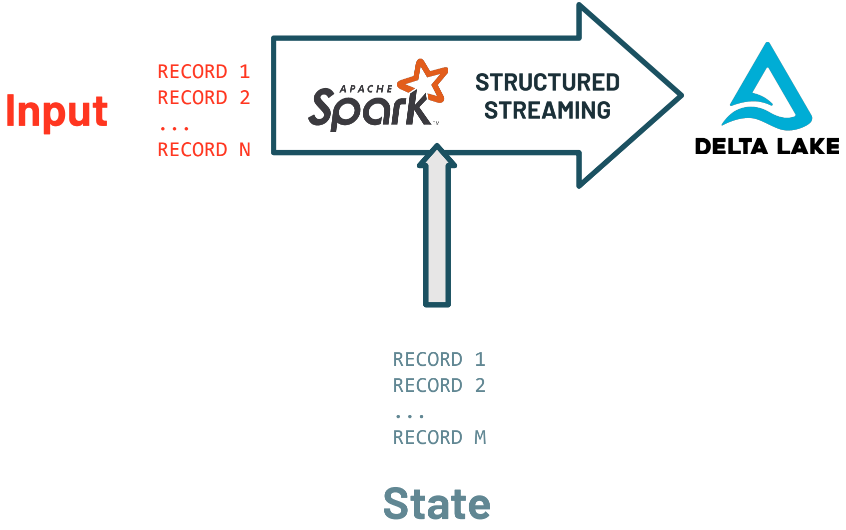
```
salesSDF = (
  spark
    .readStream
    .format("delta")
    .table("sales")
)
```

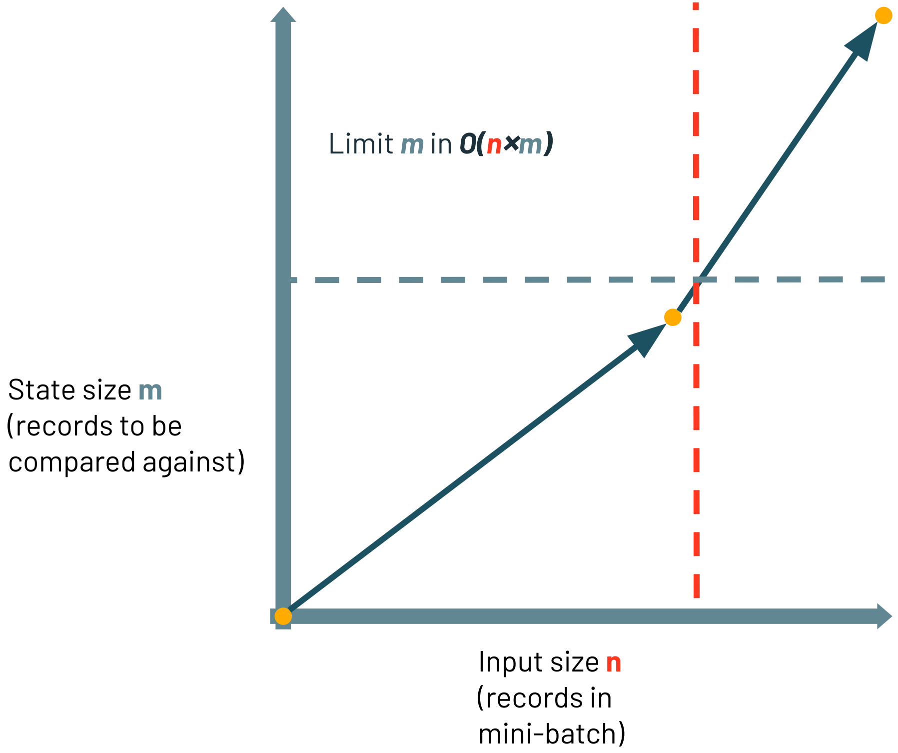
2. Join item category lookup

```
itemSalesSDF = (
  salesSDF
    .join(
      spark.table(“items”)
       .filter(“category=’Food’), # Predicate
      on=[“item_id”]
    )
)
```

Output Parameters

Limiting the output dimension





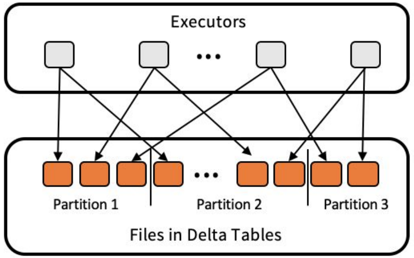
Why are output parameters important?

- Streaming jobs tend to create many small files
  - Reading a folder with many small files is slow
  - Poor performance for downstream jobs, self-joins, and merge
- Output type can impact state information retained
- Merge statements with full table scans increase state

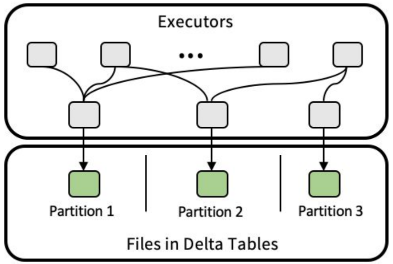
Delta Lake Output Optimizations

- Optimized Writes
- Auto Compaction
- delta.tuneFileSizesForRewrites
- Insert-only merge

Traditional Writes



Optimized Writes



Notebook: Stream Static Joins

databricks

Module Recap

1. Use clone and Auto Loader for efficient copies and incremental ingestion
2. Use native features in Spark and Delta Lake to deduplicate, enrich, and validate incremental data
3. Combine traditional data modeling techniques with Databricks features to propagate updates through the Lakehouse
4. Implement solutions that correctly manage dependencies between incremental datasets

databricks