Welcome

Applied Data Engineering with Databricks

databricks

<INSERT SLACK CHANNEL INFORMATION>

Learning Objectives

1. Build relational tables and ELT pipelines designed for the Lakehouse
2. Write Databricks-native code to incrementally process ever-expanding (streaming) data with ease
3. Design pipelines that store and delete personal identifiable information (PII) securely for data governance and compliance
4. Use best practices for developing, troubleshooting, and promoting code on Databricks
5. Implement best practices for balancing costs and latency in data pipelines
6. Schedule, orchestrate, and monitor production Databricks code

<INSERT SLACK CHANNEL INFORMATION>

Welcome!

PHOTO

Your Instructor - Your Name

-
-
-

/in/profile

databricks

http://databricks.link/class-4390-slack

Welcome!

Let’s get to know you

- Name
- Role and team
- Motivation for attending
- Favorite mobile app

Course Tools

zoom

Lecture

Breakout

Rooms

slack

TA Help + Discussion

Resources

databricks

Lab Notebooks

Solutions

http://databricks.link/class-4390-slack

Agenda

databricks

Module 1 - Architecting for the Lakehouse

- Course Intro & Overview
- Adopting the Lakehouse Architecture
- The Lakehouse Medallion Architecture
- Setting Up Tables
- Optimizing Data Storage
- Clone for Development and Disaster Recovery
- Delta Lake Atomicity and Durability
- Delta Lake Isolation with Optimistic Concurrency
- Streaming Design Patterns
- Multiplex vs. Singleplex Design

Module 2 - Managing Data in Motion

- Making Ingestion Easy with Delta Lake
- Auto Load Data to Multiplex Bronze
- Promoting Data to Silver
  - Incremental Processing from Multiplex Bronze
  - Deduplication with Incremental Processing
  - Quality Enforcement
- Slowly Changing Dimension Tables
- Propagating Changes with Delta Lake & Change Data Feed
- Streaming Joins and Statefulness
- Making Data Available for Analytics

Module 3 - Privacy in the Lakehouse

- PII & Regulatory Compliance
- Pseudonymized PII Lookup Table
- Storing PII Securely
- Managing ACLs for the Enterprise Lakehouse
- De-identified PII Access
- Propagating Deletes with Change Data Feed
- Deleting at Partition Boundaries

Module 4 - Troubleshooting Performance Issues

- The Spark UI
    - Assessment
    - Review
- The 5 Most Common Performance Problems - Condensed
    - Storage
    - Serialization
    - Skew
    - Spill
    - Shuffles
- Designing Clusters for High Performance
    - Prescriptions for Cloud-Specific VM Types
    - Cluster Configurations Scenarios

Module 5 - Databricks in Production

- Promoting Code with Databricks Repos
  - Refactoring to a Relative Import
  - Committing, Merging, and Pulling Changes
- Managing Costs and Latency with Incremental Workloads
  - Controlling Latency in Structured Streaming Workloads
  - Efficient Structured Streaming
- Orchestration and Scheduling with Multi-Task Jobs
- Using the CLI
- Monitoring

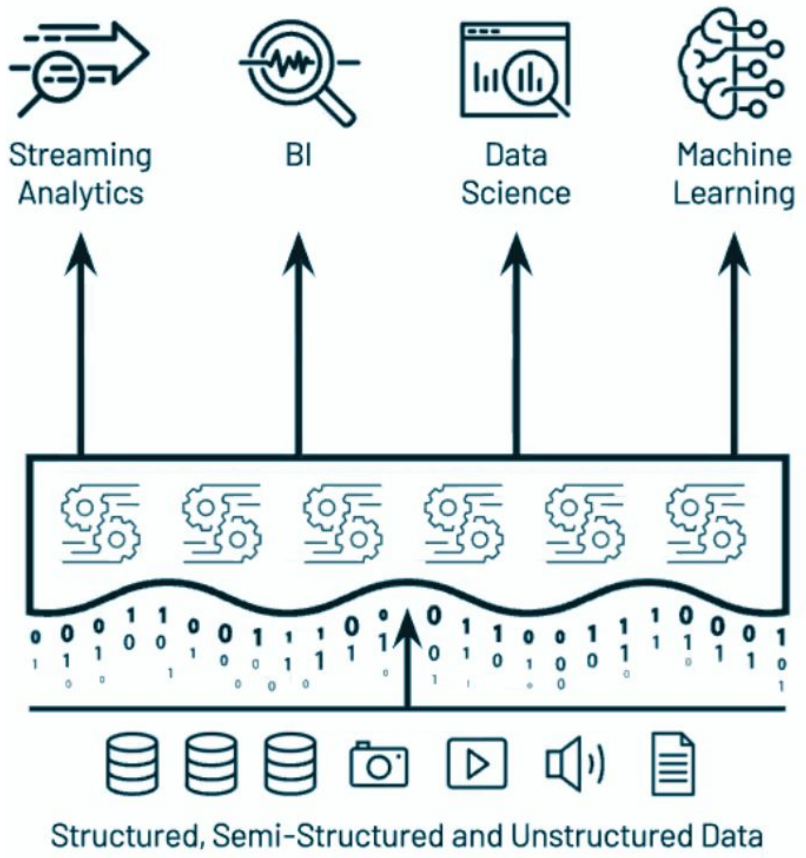
Project Introduction

Design and implement a multi-pipeline multi-hop architecture to enable the Lakehouse paradigm.

databricks

Our Company

moovio



Adopting the Lakehouse Architecture

databricks

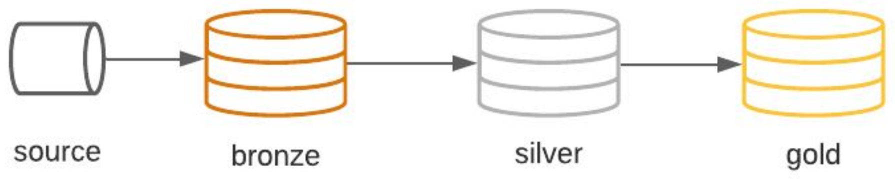
Multi-hop Pipeline

Source:
    Files or integrated systems

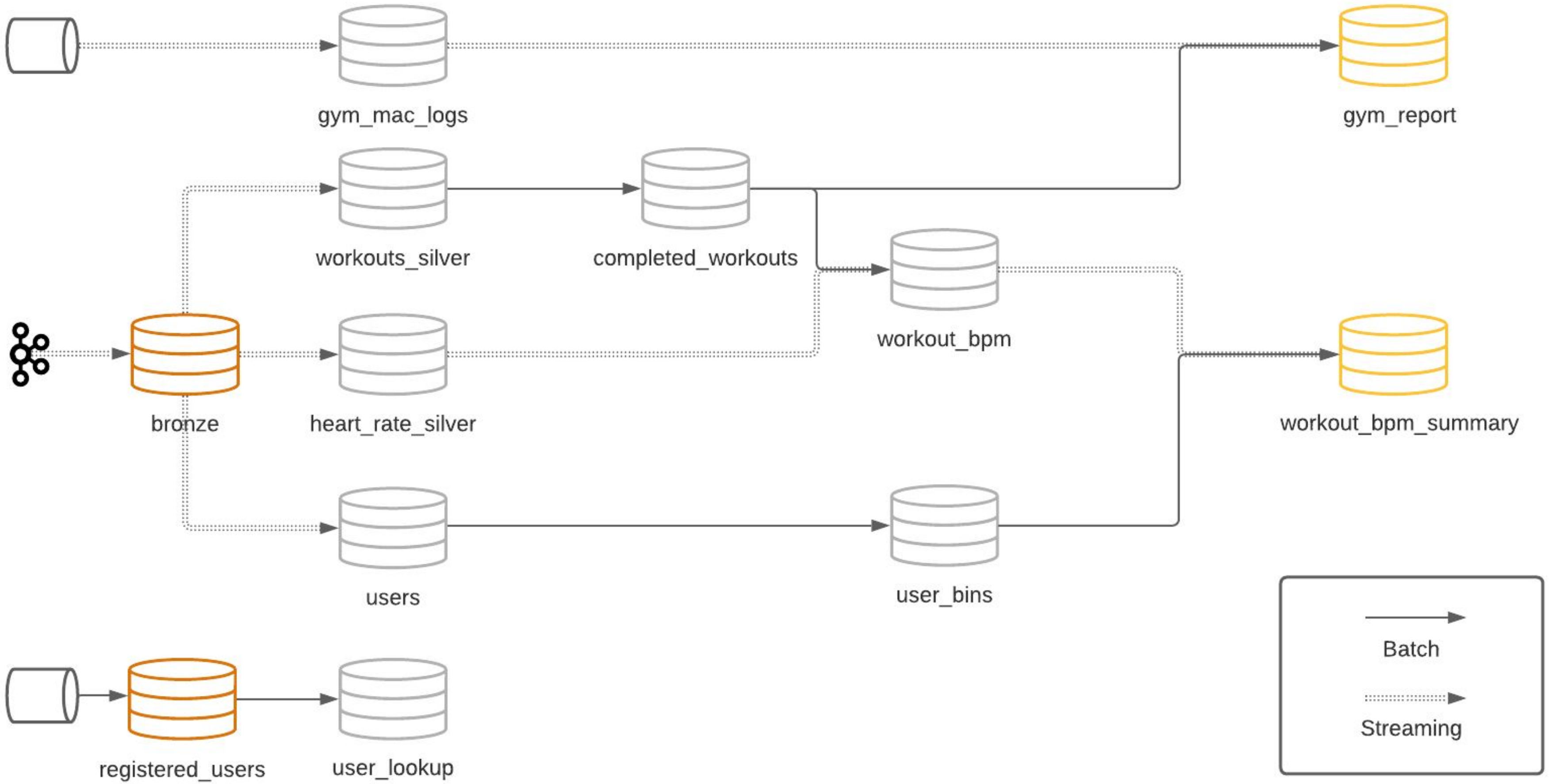
Bronze:
    Raw data and metadata

Silver:
    Validated data with atomic grain

Gold:
    Refined, aggregated data



databricks


gym_mac_logs
gym_report
workouts_silver
completed_workouts
workout_bpm
bronze
heart_rate_silver
workout_bpm_summary
users
user_bins
Batch
Streaming
registered_users
user_lookup

Recommendations for Success with Lakehouse

- All writes use Delta Lake
- All tables should be in the metastore
- Each layer in its own database
- Each layer using its own cloud object storage
- Advertise features:
  - Column prefixes: **p_**, **z_**, **b_**, etc.
  - DB & tables suffixes: **_bronze**, **_hist, _etl**, **_t**, etc

Cloud Blockers to Avoid

- Make sure region and cloud preferences are established before writing/migrating production data
- Try to colocate compute to storage
- Identify the necessary quota for operation in each account
- Identify conflicts due to regulatory requirements
- Identify existing infrastructure in undesired regions

The Lakehouse Medallion Architecture

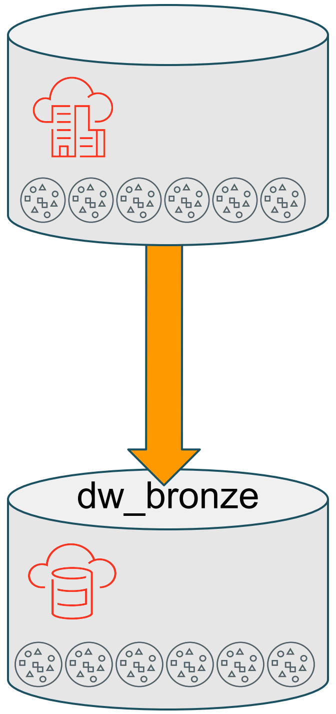
databricks

Bronze Layer

databricks

Ingest Bronze

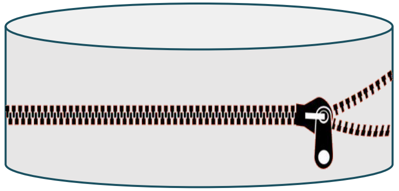
- Typically just a raw copy of ingested data
- Should be append only
- Can be incremental or batch appended
- Can contain additional metadata
- Creates a replayable history of the data



Optimize Bronze

- **`spark.sql("OPTIMIZE table")`**
- Develop an optimized schedule based on
  - Ingest periodicity
  - Downstream requirements
- Consider smaller clusters running in off hours
  - Ingest clusters are often oversized for this task
  - See the Optimize Helpers in the Playbook
- If later Z-Ordering, don't use optimize-writes or auto-compact
- Compute stats, not just for the whole table, but also for key columns

```
analyze table <table name> compute statistics for columns joinKey1, joinKey2
```



databricks

Processing Deletes

- The source_delete != lakehouse_delete
- Soft-deletes are more valuable
- Hard-deletes may be required by regulatory processes

Silver Layer

What is the Silver Layer?

- Preserves grain of original data (no aggregation)
- Eliminates duplicate records
- Production schema enforced
- Data quality checks passed
- Corrupt data quarantined
- Data stored to support production workloads
- Optimized for long-term retention and ad-hoc queries

Why is the Silver Layer important?

- Replaces non-curated “data lake”
  - Data is clean
  - Transactions have ACID guarantees
- Represents full history of business action modeled
  - Each record processed is preserved
  - All records can be efficiently queried
- Reduces data storage complexity, latency, and redundancy
  - Built for both ETL throughput AND analytic query performance

Schema Considerations

- Choose the correct precision for numeric values
- Use proper datetime types
- Flatten nested fields
- Eliminate case-sensitive field names
- **Order matters**

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
- Historic records also have field no-write “appended”
  - Change is to metadata only, managed by Delta Log
  - All previous records will show newly added field as Null

Streaming Deduplication

- `dropDuplicates` operator removes all duplicate rows from a DataFrame
  - For streaming operations, set a watermark to avoid state explosion
- A Delta Lake insert-only merge prevents records with duplicate keys from being processed
- Streaming operations combine this with `foreachBatch`

Delta Lake Constraints

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

```
ALTER TABLE tableName ADD CONSTRAINT constraintName
    CHECK fieldName > '99';
```

Alternative Quality Check Approaches

- Use boolean filters to identify rows that violate constraints
- Quarantine data by filtering non-compliant data to alternate location
- Warn without failing by writing additional fields with constraint check results to Delta tables

Mapping Bronze to Silver

- Consider using a new bucket/storage account
- Start with a new database, e.g. **dw_silver**
- Implement the mappings to transform Bronze to Silver
  - Where the implementation will most likely get behind ⚠️
  - The #1 reason is not having near-perfectly defined mappings ⚠️
- Demonstrate best practices for CDC datasets
  - Only the current data is active in the Silver layer
  - Historical records are properly annotated with a terminal timestamp

Standard optimizations

- Design and implement proper partitioning strategies
  - Based on each persona's specific use cases
  - Validate both file and partition sizes
  - Partition by values with a low cardinality

- Design and implement Z-Ordering & optimization strategies
  - Based on each persona's specific use cases
  - To accept upserts from Bronze
  - To deliver specific resources to Gold
  - Index by values with a high cardinality

Metadata

- Design for Silver & Gold, start to implement now
- Good metadata can aid in developing validations later
- For example, columnar metadata can define an enumerated set of valid values
- Examples of where to store metadata include:
  - Column Comments
  - Table Properties
  - The Unity Catalog ⭐

Gold Layer

databricks

Upon Completing Silver...

- You have a fully documented inventory of each table to be replicated

- A complete design of what the Gold layer will consist of including:
  - Which tables are customer facing
  - Which tables will be created as Type-2
  - How much historical data will be presented
  - Which historical tables will have a "current-only" version of the dataset
  - If the volume of history warrants a history-specific DB such as **dw_gold_hist**
  - How the data will be partitioned and z-ordered to meet specific use cases

Creating the Gold Layer

- Unlike Bronze or Silver, you must use a new bucket/storage account
    - ETL can put heavy load on systems - e.g. throttled by the cloud provider
    - Provides additional security capabilities

- Start with two to four new databases
    - ***_gold** - Only for views
    - ***_gold_t** - Only for final tables upon which views are built
    - ***_gold_etl** - Only for ETL tables used to produced **gold_t** tables
    - ***_gold_hist** - Like, **dw_gold**, this is views only, but of historical tables only

- Employ an intentional table nomenclature:
    - **city_t**, **trx_curr_t**, **trx_hist_t** - for storage tables - not customer facing
    - **city**, **trx_curr**, **trx_hist** - for views - customer facing

Setting up Security

- Create each database with an ACL cluster using SQL statements

- Validate proper ownership of the DB before continuing ⚠️
  - Future tables will inherit ownership
  - Precludes extra work required when tables are created before security is setup

- Consumer groups should be explicitly denied access to the ***_bronze**, ***_silver**, ***_gold_etl**, and ***_gold_t** databases

- Consumer groups should be explicitly granted **select** and **read_metadata** to the ***_gold** and ***_hist_gold** databases

Create the Gold Views

- Create one view for EVERY Gold table
- This should be done in the ***_gold** and ***_gold_hist** databases
- Do not use **select *** when defining the source columns ⚠️
- Table columns must be explicitly mapped to view columns and re-aliased
- Predicates must be replicated in the view definitions to ensure predicates are pushed down to the underlying data sources ⚠️

databricks

Notebook: Setting Up Tables

Notebook: Optimizing Data Storage

databricks

Clone for Development and Disaster Recovery

databricks

Clones:

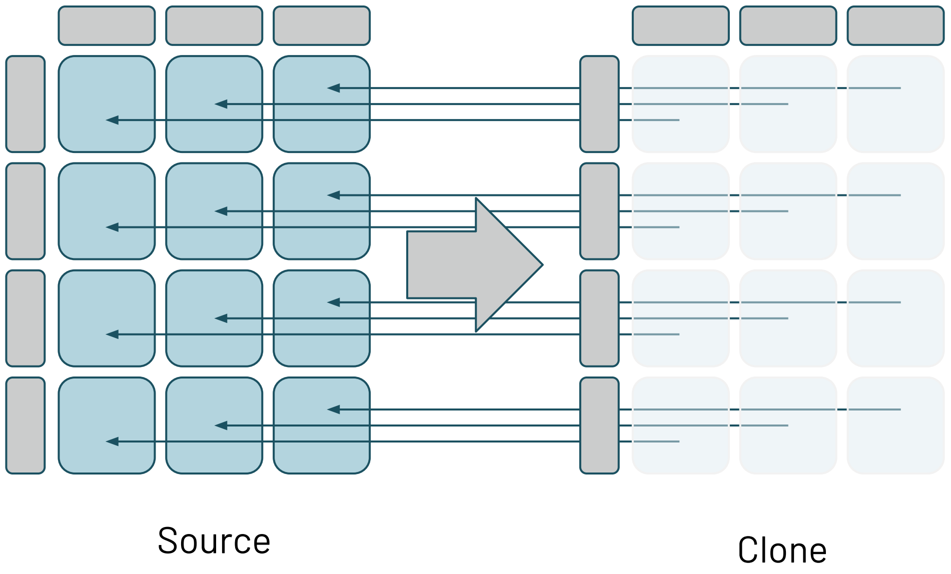
- Create a replica of a target table
- At a point in time
- In a specific destination location

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

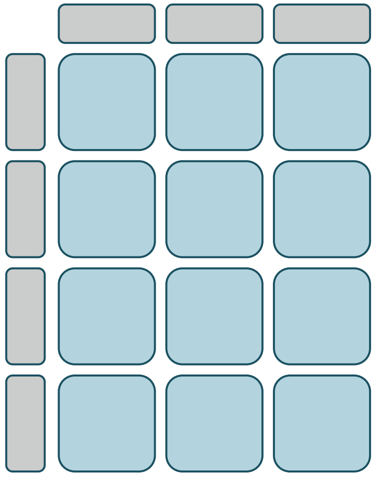
Shallow Clones

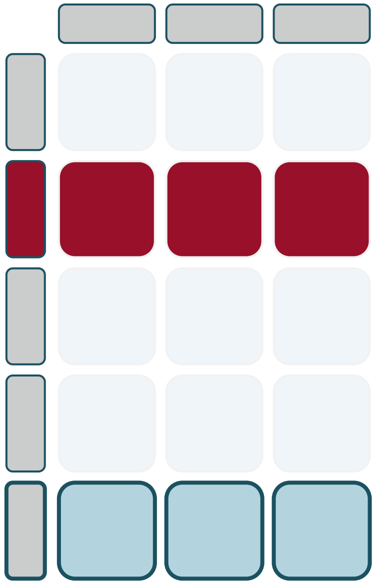
- Zero-copy cloning
  - Only metadata is copied
  - Points to original data files
- Inexpensive and fast
- Not self-contained
  - Depend on sourced data files
  - If source data files are removed, shallow clone may break



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

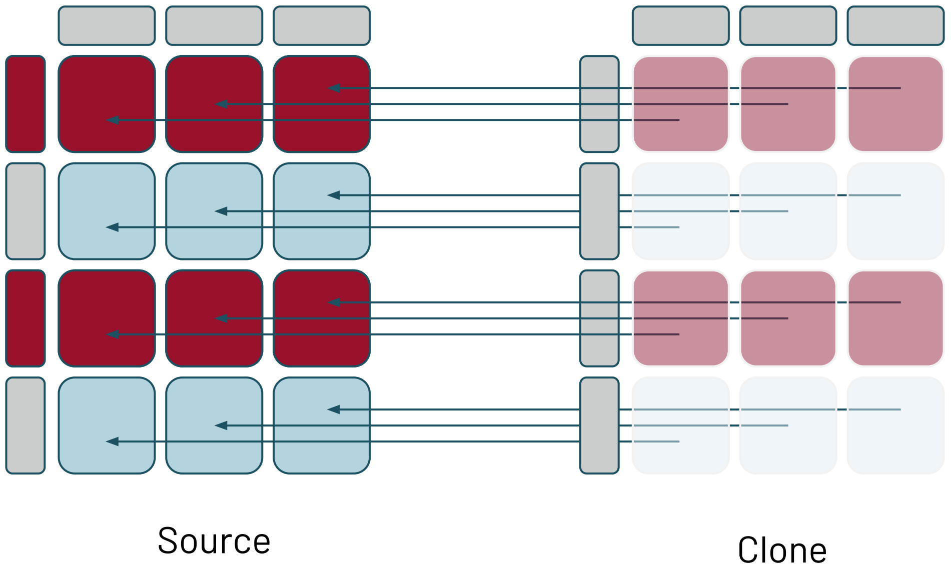

Source


Clone

databricks

Removing Source Files

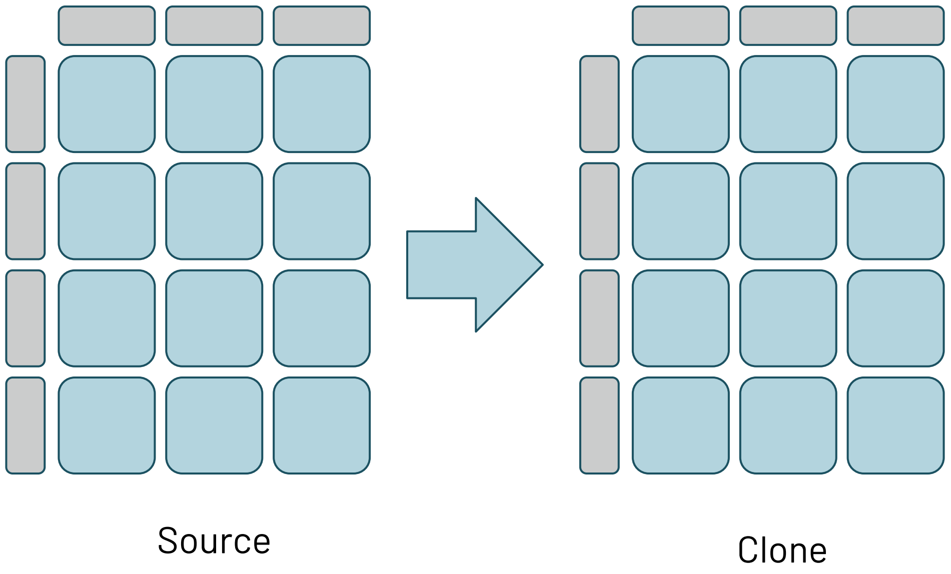
- Changes to the source table mark data files as no longer valid
- Vacuuming the source table will permanently remove these data files
- References to source data files will cause queries on the clone table to fail



databricks

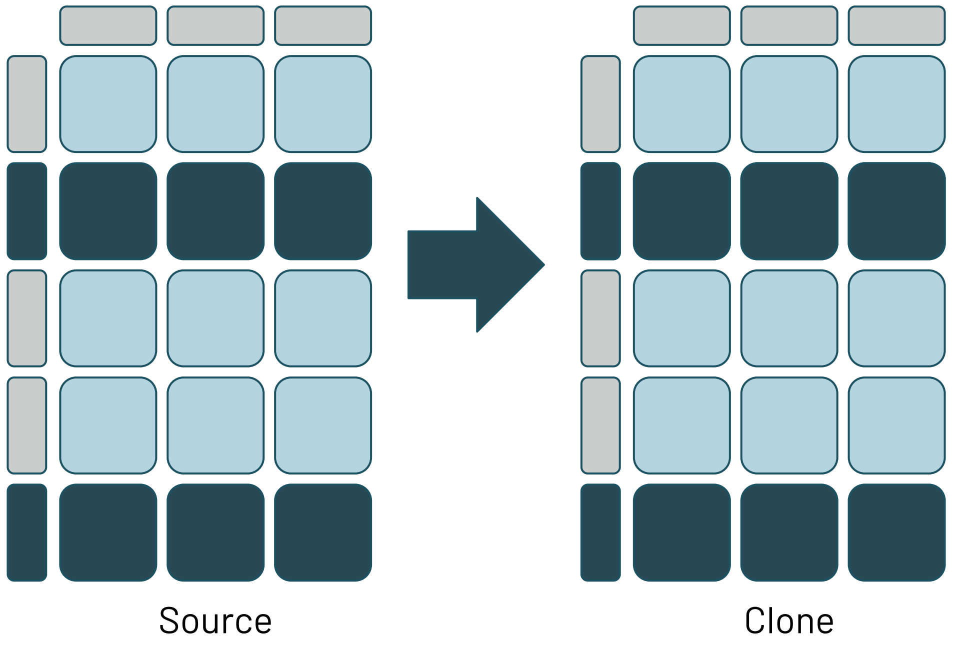
Deep Clones

- 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



databricks

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

Notebook: Using Clone with Delta Lake

databricks

Delta Lake Atomicity and Durability

databricks

Delta Lake brings ACID to object storage

- **A**tomicity
- **C**onsistency
- **I**solation
- **D**urability

Delta Lake provides ACID guarantees scoped to tables

Problems solved by ACID

1. Hard to append data
2. Modification of existing data difficult
3. Jobs failing mid way
4. Real-time operations hard
5. Costly to keep historical data versions

Durability with Delta Lake

- All write operations commit data changes as Parquet files
- Transactions committed using JSON log files
- Stored in a nested directory
- Inherits the durability guarantees of the file system

Cloud-based object storage

- Infinitely scalable
- Affordable
- Availability: > 99.9%
- Durability: > 99.999999999%

databricks

The Anatomy of Delta Lake Atomicity

Data Files

- Parquet files
- Written continuously throughout transaction
- All data modifications create new parquet files

Transaction Log

- JSON files
- Written once as transaction completes
- Indicates added and removed data files, plus metadata

The Anatomy of Delta Lake Atomicity

```
/path/to/table/
 -  _delta_log/
 -  part-0000.snappy.parquet
 -  part-0001.snappy.parquet
 -  ...

/path/to/table/_delta_log/
 -  0000.json
 -  0001.json
 -  0002.json
 -  ...
 -  0010.parquet
```

(with partitions)

```
/path/to/table/
 -  _delta_log/
 -  date=2020-01-01/
 -  date=2020-01-02/
 -  ...

/path/to/table/date=2020-01-01/
 -  part-0000.snappy.parquet
 -  part-0001.snappy.parquet
 -  ...
```

The Transaction Log

Makes every operation transactional

- It either fully succeeds - or it is fully aborted for later retries

```
/path/to/table/_delta_log/
 -  0000.json
 -  0001.json
 -  0002.json
 -  …
 -  0010.parquet
```

The Transaction Log

Makes every operation transactional

- It either fully succeeds - or it is fully aborted for later retries

```
/path/to/table/_delta_log/
 -  0000.json ⎧ Add   part-0001.snappy.parquet
 -  0001.json ⎨ Add   part-0002.snappy.parquet
 -  0002.json ⎩ …
 -  …
 -  0010.parquet
```

The Transaction Log

Makes every operation transactional

- It either fully succeeds - or it is fully aborted for later retries

```
/path/to/table/_delta_log/
 -  0000.json
 -  0001.json  { Remove  part-0001.snappy.parquet
 -  0002.json  { Add     part-0003.snappy.parquet
 -  …          { …
 -  0010.parquet
```

databricks

The Transaction Log

Makes every operation transactional

- It either fully succeeds - or it is fully aborted for later retries

```
/path/to/table/_delta_log/
 -   0000.json
 -   0001.json
 -   0002.json
 -   …
 -   0010.parquet
 -   0010.json
 -   0011.json
```

Inside the Transaction Log

- `txn`
- `add`
- `remove`
- `metaData`
- `protocol`
- `commitInfo`

databricks

Inside the Transaction Log - `add`

- `path`
- `partitionValues`
- `size`
- `modificationTime`
- `stats`

Inside the Transaction Log - `remove`

- `path`
- `deletionTimestamp`

Inside the Transaction Log - `metaData`

- `schemaString`
- `partitionColumns`
- `format`
- `createdTime`

databricks

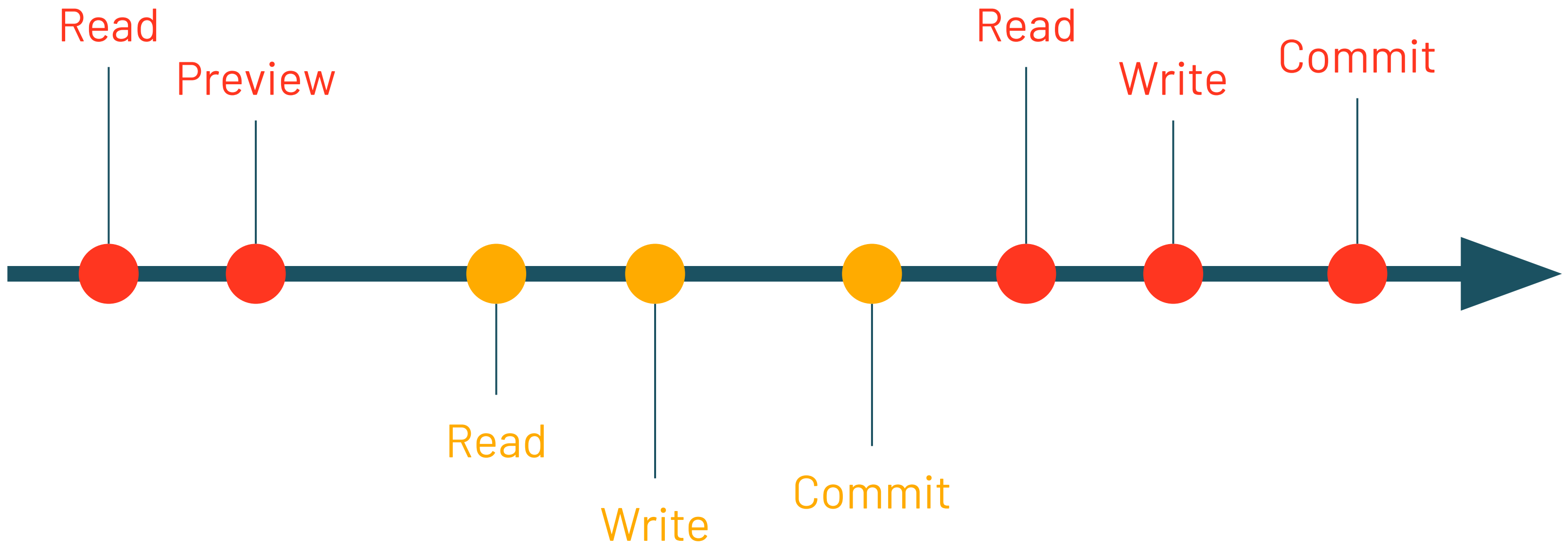
Isolation with Optimistic Concurrency Control

databricks

What is Isolation?

- Determines when and how transactions become visible to other users and systems
- Concerned with concurrency effects
- Exists as a spectrum, referred to as “isolation levels”
- Typically defined at database level
- **Delta Lake defines isolation at the table level**

Concurrent Transactions



Read Phonomena

Dirty reads

Data modified by another concurrent transaction is returned instead of the valid state.

Non-repeatable reads

Multiple results are returned for a single row due to changes made in a concurrent transaction.

Phantom reads

Data added or deleted by a concurrent transaction is also read.

Read Phonomena

Dirty reads

Data modified by another concurrent transaction is returned instead of the valid state.

Non-repeatable reads

Multiple results are returned for a single row due to changes made in a concurrent transaction.

Phantom reads

Data added or deleted by a concurrent transaction is also read.

Lax $\longrightarrow$ Strict

Traditional Isolation Levels

	Read uncommitted	Read committed	Repeatable reads	Serializable
Dirty reads	🟥	✅	✅	✅
Non-repeatable reads	🟥	🟥	✅	✅
Phantoms	🟥	🟥	🟥	✅

databricks

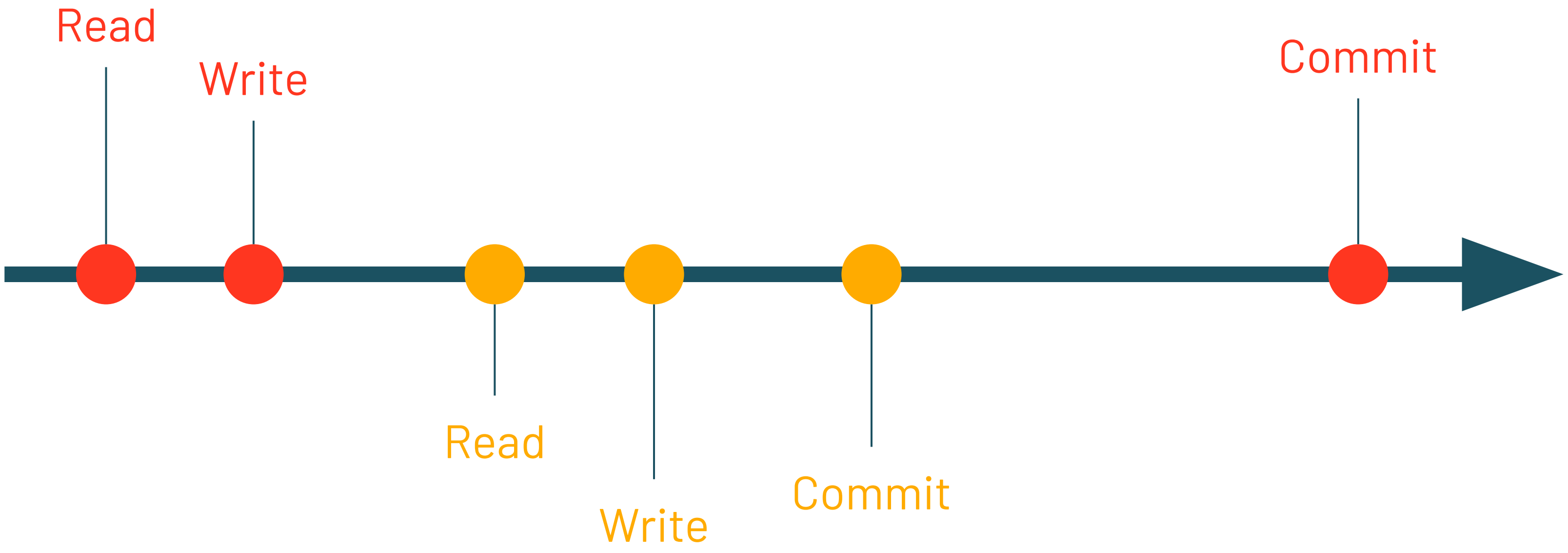
Delta Lake WriteSerializable

- Relaxes strict serial guarantees to increase throughput
- Append operations will never conflict or block concurrent execution
- Default in Delta Lake

Optimistic Concurrency Control

1. Read
2. Write
3. Validate and commit

Concurrent Transactions

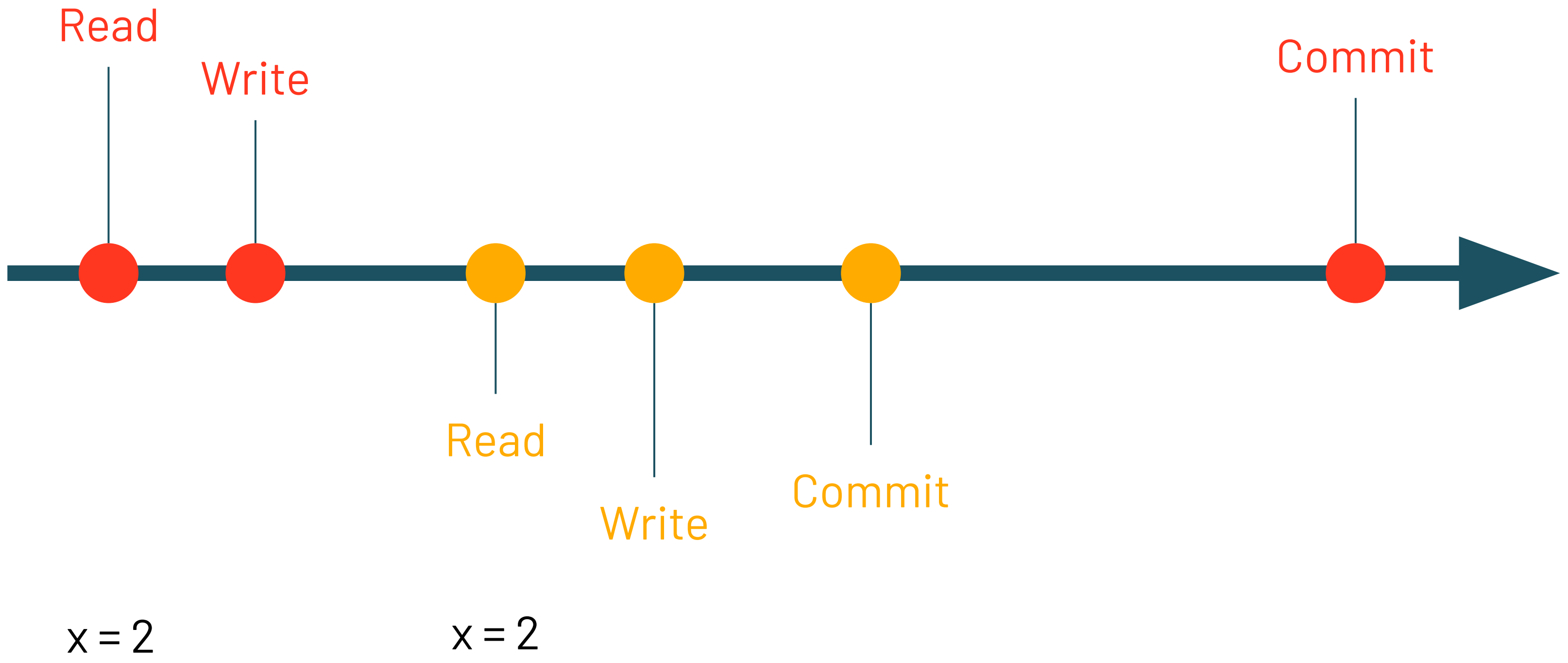


databricks

Write Conflicts

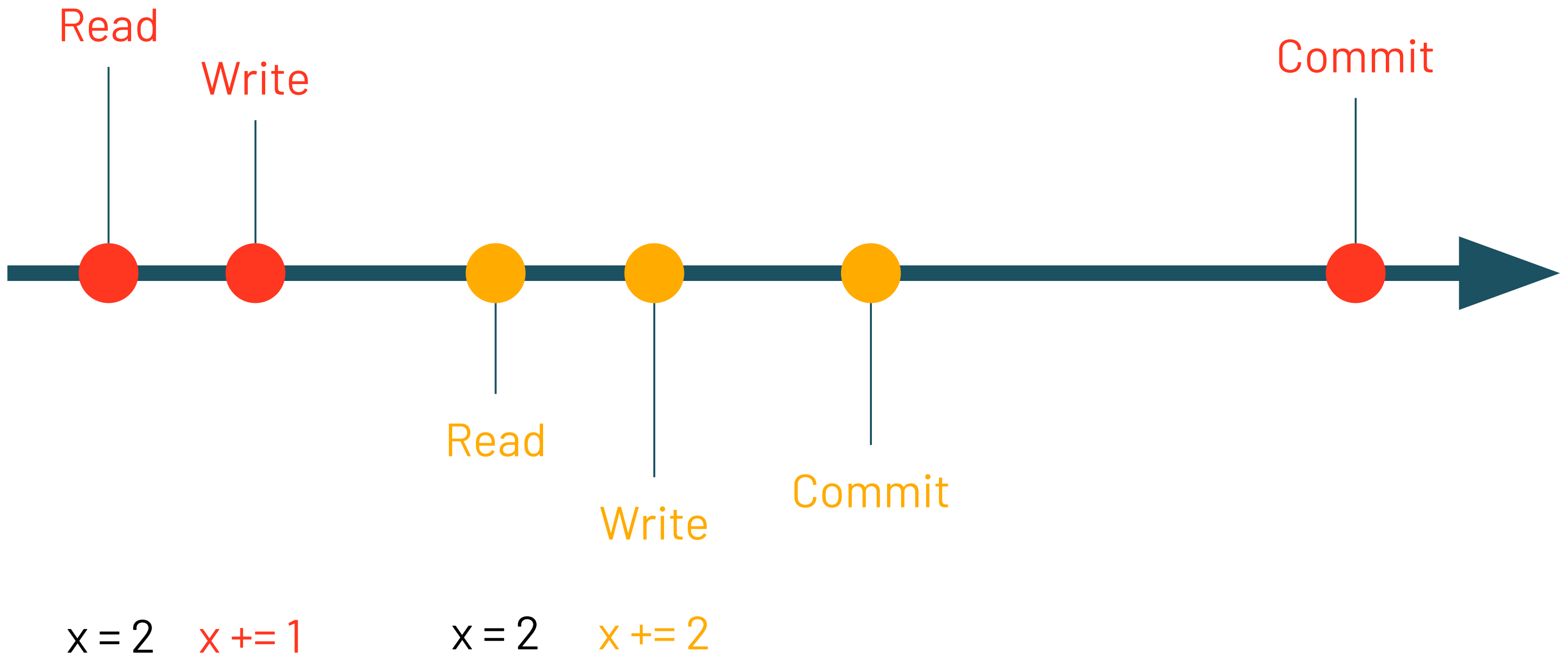
	INSERT	UPDATE, DELETE, MERGE INTO	OPTIMIZE
INSERT	**Cannot conflict**		
UPDATE, DELETE, MERGE INTO	**Cannot conflict**	**Can conflict**	
OPTIMIZE	**Cannot conflict**	**Can conflict**	**Can conflict**

Conflicting Update



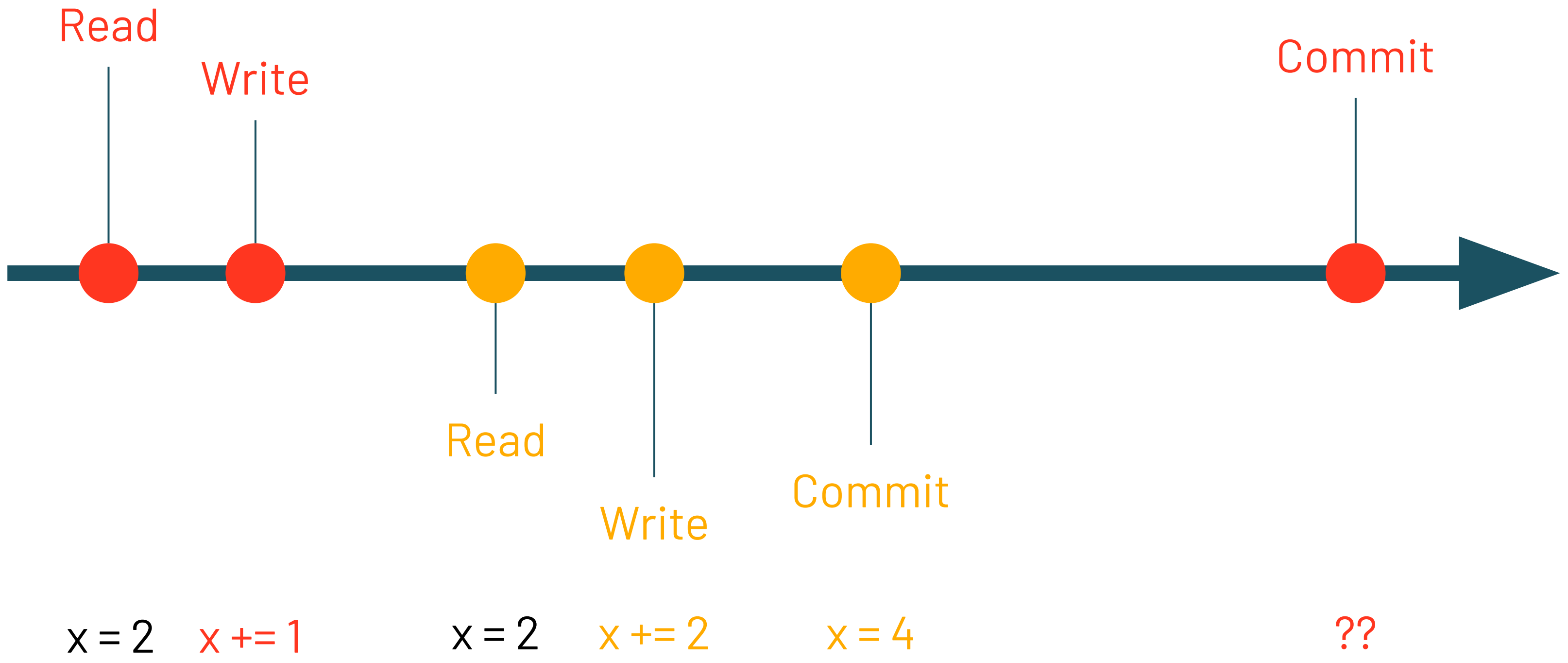
databricks

Conflicting Update



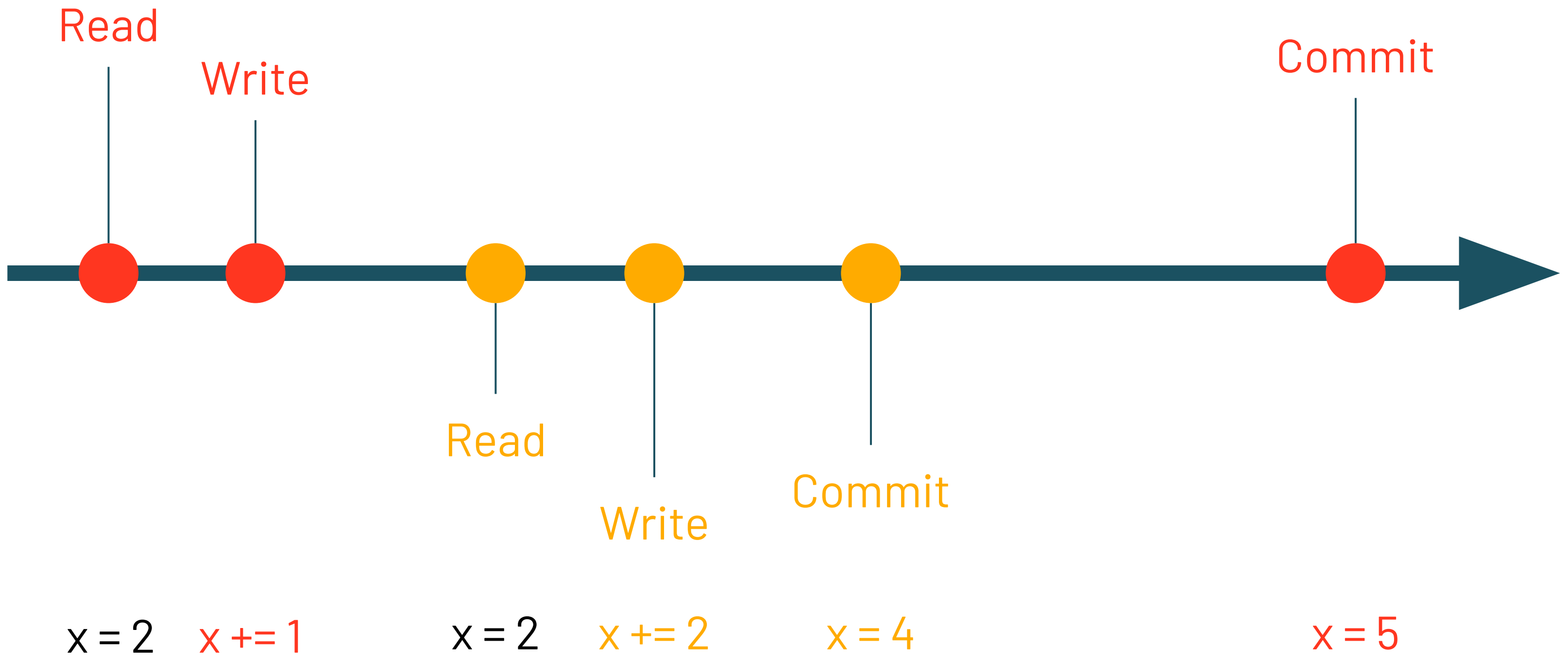
databricks

Conflicting Update



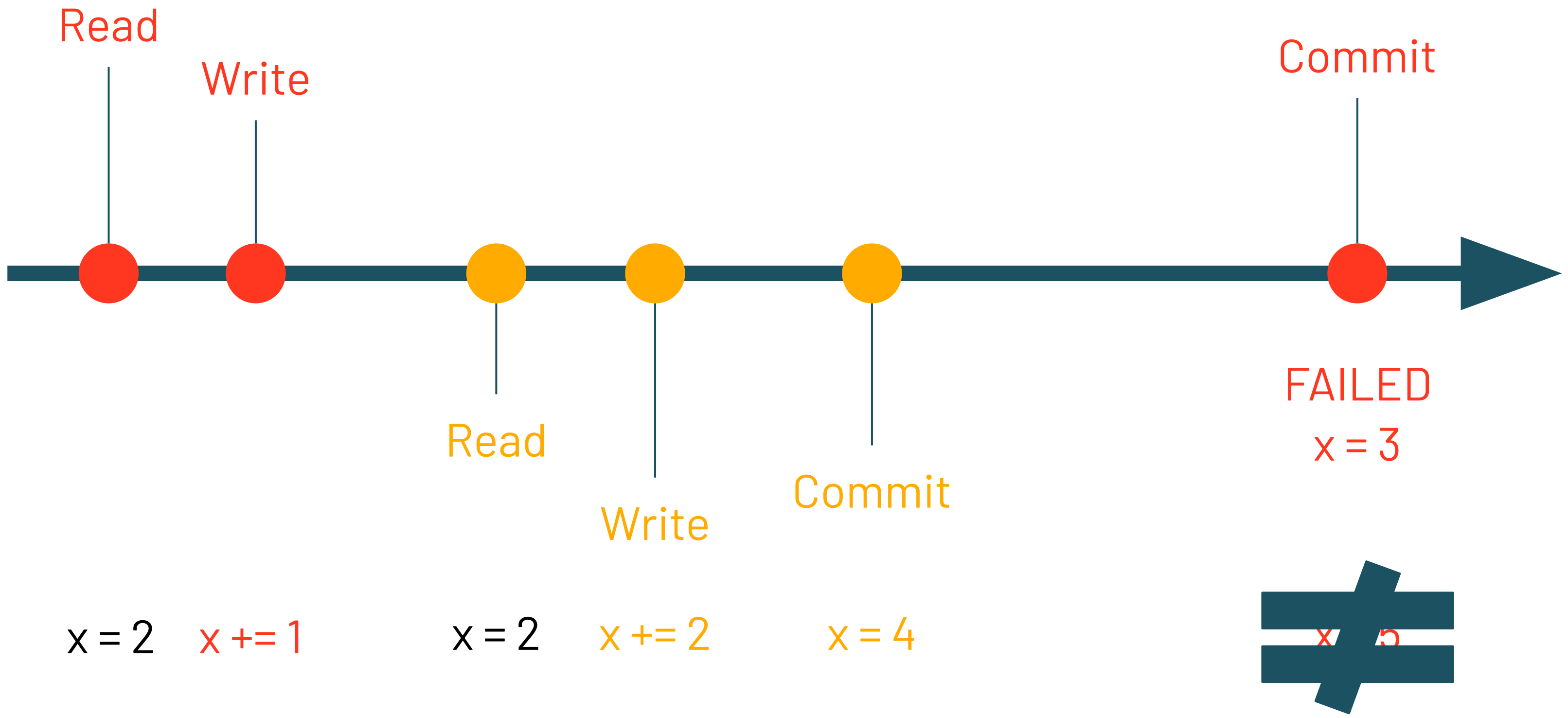
databricks

Conflicting Update

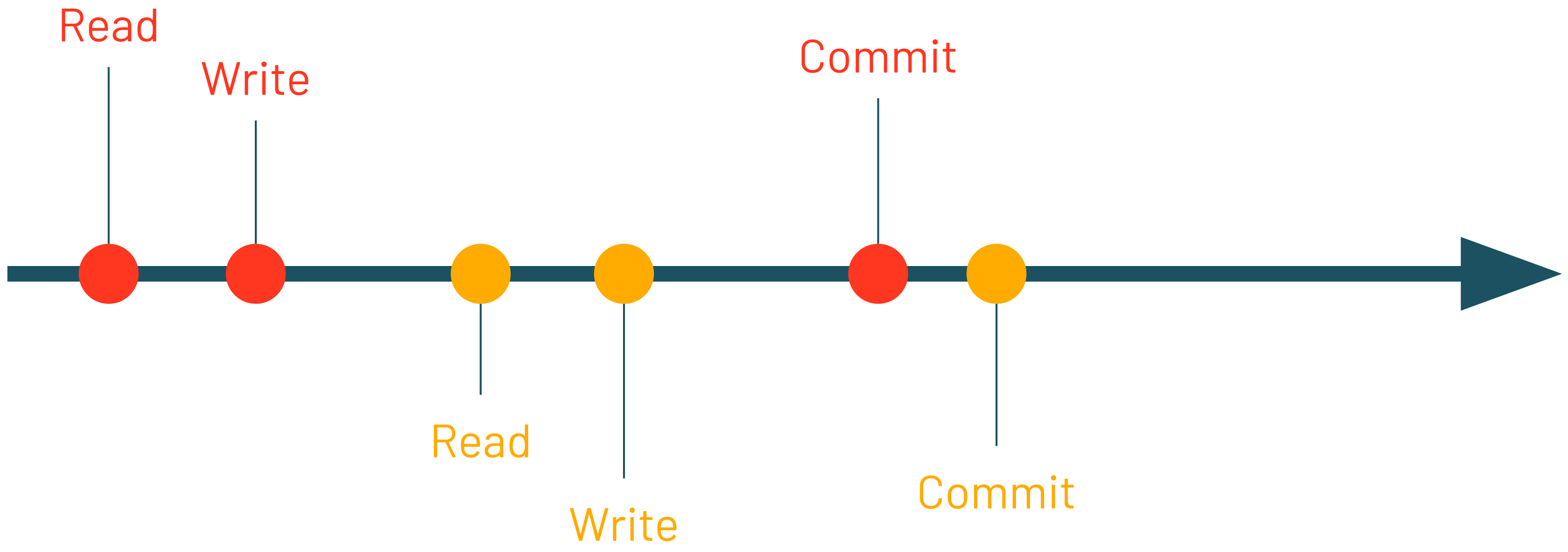


databricks

Conflicting Update



Conflicting Update



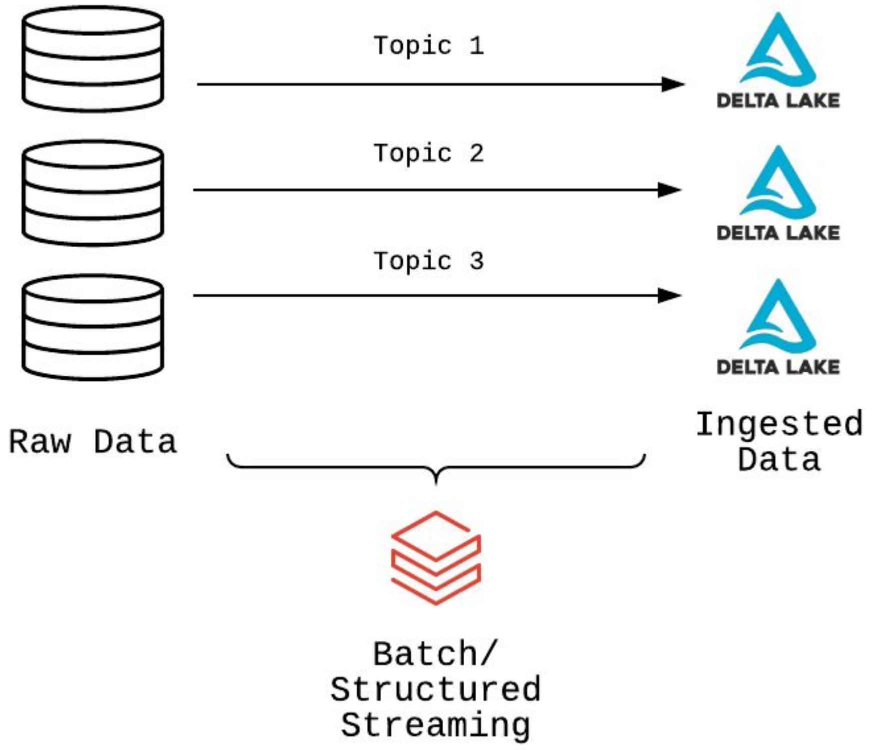
databricks

Notebook: Streaming Design Patterns

Multiplex vs. Singleplex Design

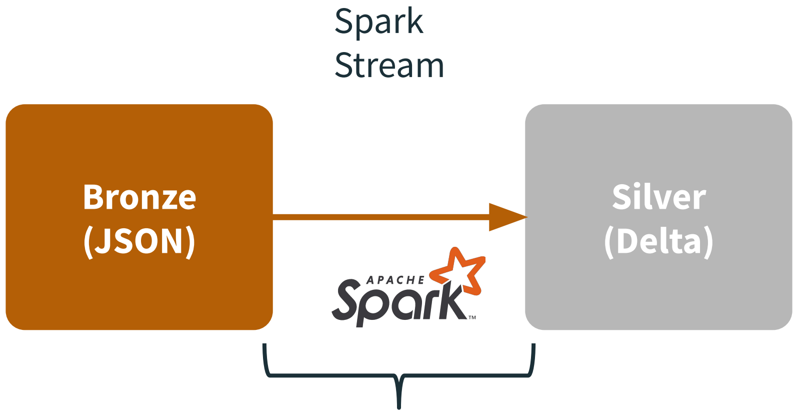
databricks

Singleplex Ingestion



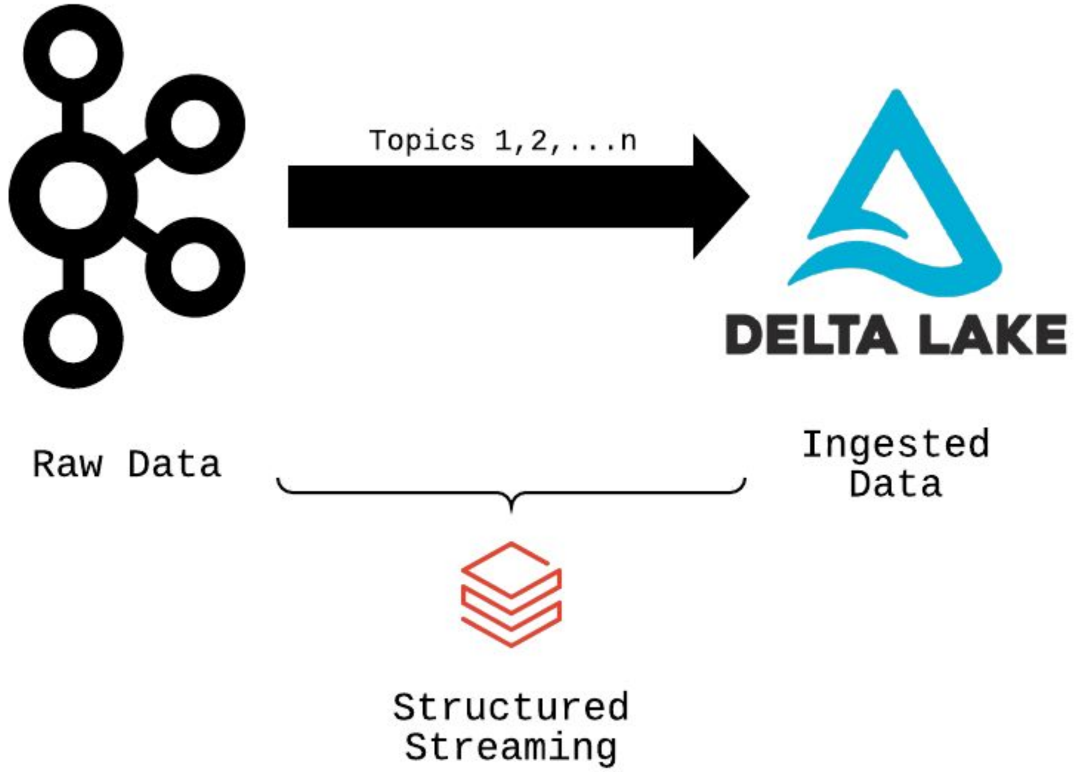
Singleplex streaming

- Multiple streams on one cluster, one per source event type / topic e.g.:



Deploy x of these streams on a single cluster, reading from x queues (1 per event type)

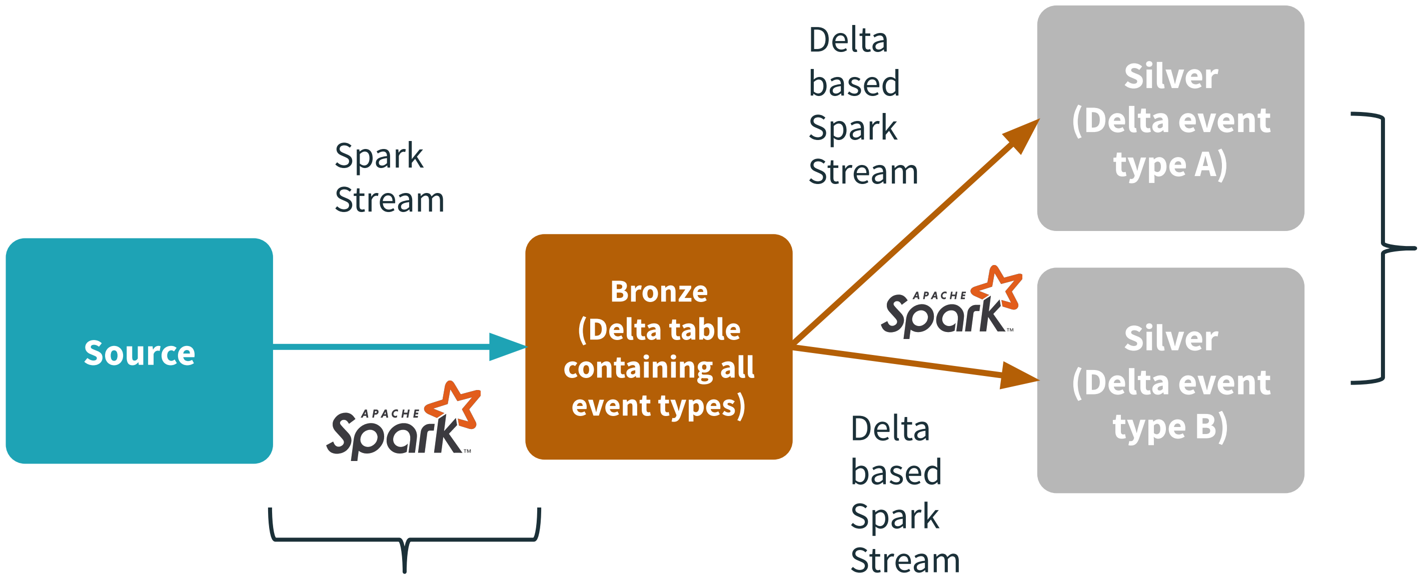
Multiplex Ingestion



Demultiplexing

- Data streamed into pub/sub with heterogeneous subjects
- Simplifies source stream architecture
- Data consumed from heterogeneous topics into raw Delta Table (in the "Bronze" layer), maintaining source message schema
- Delta table is then "demuxed", separating messages by a category (like event type), schemas applied, then written to a target Delta table (in the "Silver" layer)

Multiplex streaming (push Delta up)


Spark Stream
Source
Bronze (Delta table containing all event types)
Delta based Spark Stream
Silver (Delta event type A)
Silver (Delta event type B)
Delta based Spark Stream
APACHE Spark

Branch from this Delta table with multiple streams, on demand (trigger once), potential separate clusters

Multiplex all event types through an initial stream that converts to Delta with limited schema (1 stream on dedicated cluster)

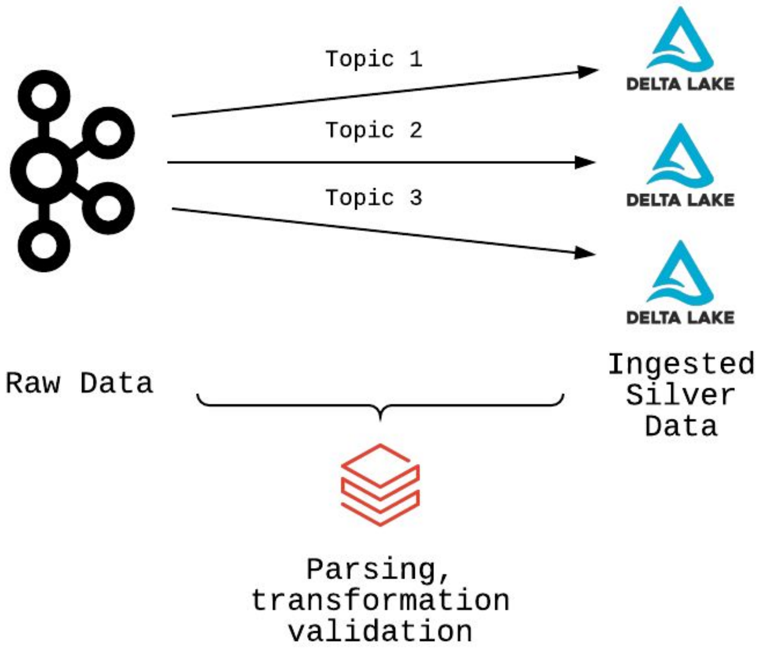
databricks

Why use a bronze layer?

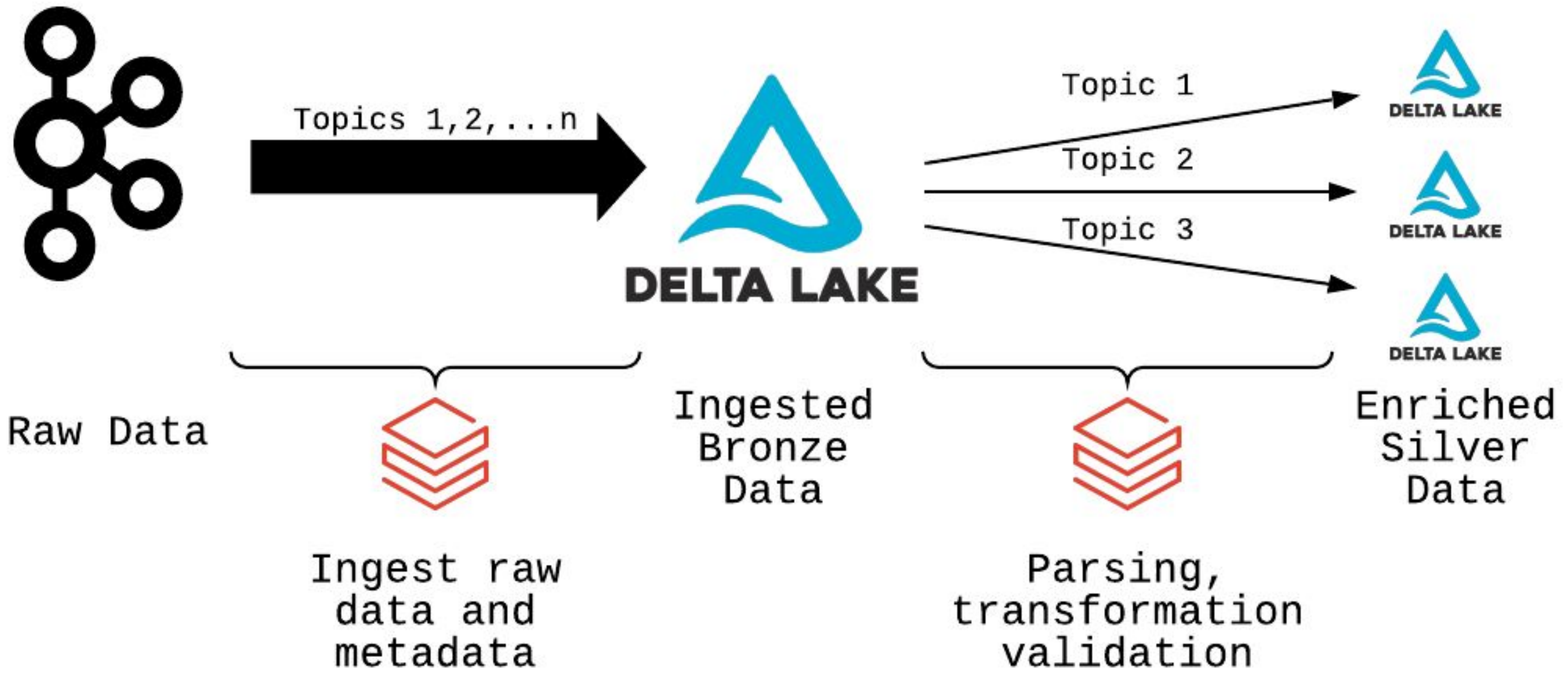
databricks

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



Is multiplex better than singleplex?

Not necessarily

- Domain Driven Design
- Less driver resource contention

databricks