



Welcome

Advanced Data Engineering
with Databricks



Learning Objectives

1. Design databases and pipelines optimized for the Databricks Lakehouse Platform.
2. Implement efficient incremental data processing to validate and enrich data driving business decisions and applications.
3. Leverage Databricks-native features for managing access to sensitive data and fulfilling right-to-be-forgotten requests.
4. Manage error troubleshooting, code promotion, task orchestration, and production job monitoring using Databricks tools.



Course Tools

The Zoom logo is displayed in a blue, lowercase, sans-serif font.

Lecture

Breakout

Rooms

The Slack logo features a colorful icon of four rounded squares (cyan, green, yellow, and red) arranged in a cross pattern, followed by the word "slack" in a bold, black, lowercase, sans-serif font.

TA Help + Discussion

Resources

The Databricks logo consists of a red icon of three stacked cubes, followed by the word "databricks" in a bold, dark blue, lowercase, sans-serif font.

Lab Notebooks

Solutions

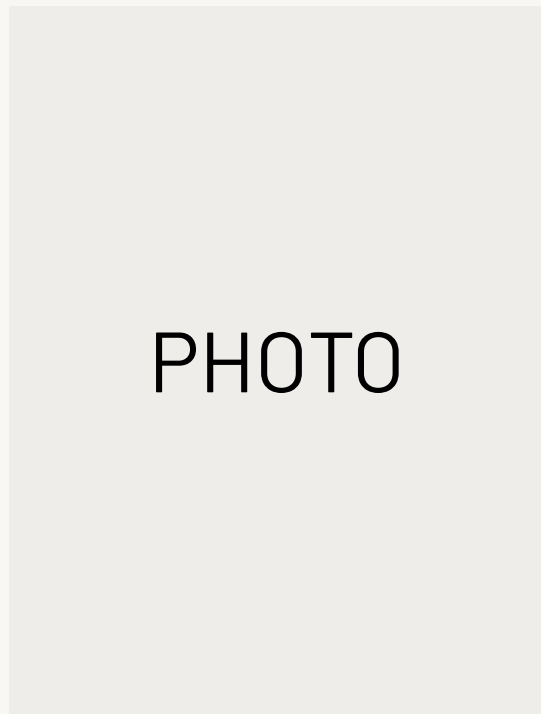


Modules

1. [Architecting for the Lakehouse](#)
2. [Bronze Ingestion Patterns](#)
3. [Promoting to Silver](#)
4. [Gold Query Layer](#)
5. [Storing Data Securely](#)
6. [Propagating Updates and Deletes](#)
7. [Orchestration and Scheduling](#)



Welcome!



Your Instructor – Your Name

-
-
-

 /in/profile



Welcome!

Let's get to know you 

- Name
- Role and team
- Length of experience with Spark and Databricks
- Motivation for attending
- Fun fact or favorite mobile app

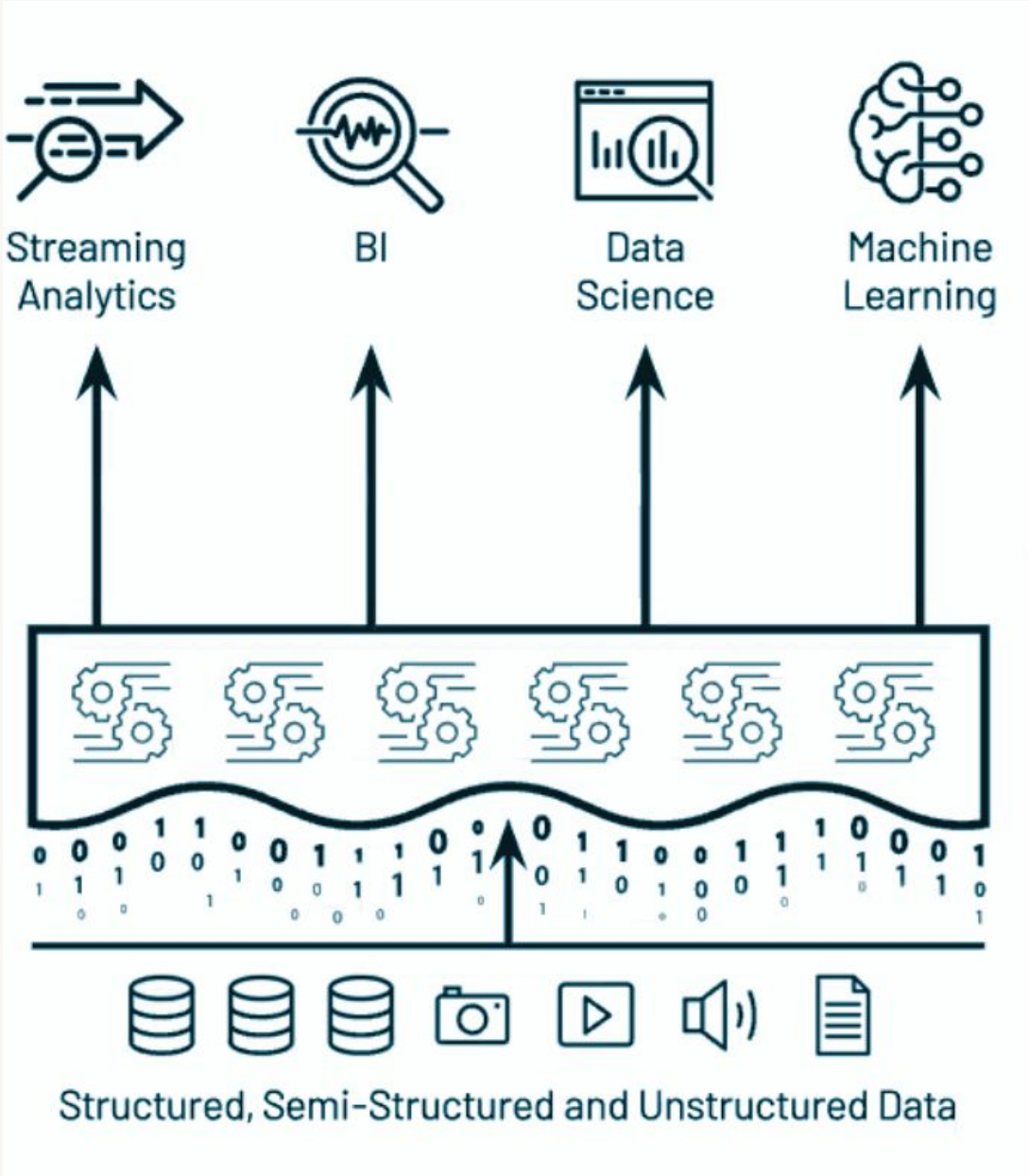


Course Objective

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



Our Company





Architecting for the Lakehouse

Adopting the Lakehouse Architecture
Lakehouse Medallion Architecture
Streaming Design Patterns

Adopting the Lakehouse Architecture





**Data
Lake**

Lakehouse




One platform to unify all of
your data, analytics, and AI
workloads

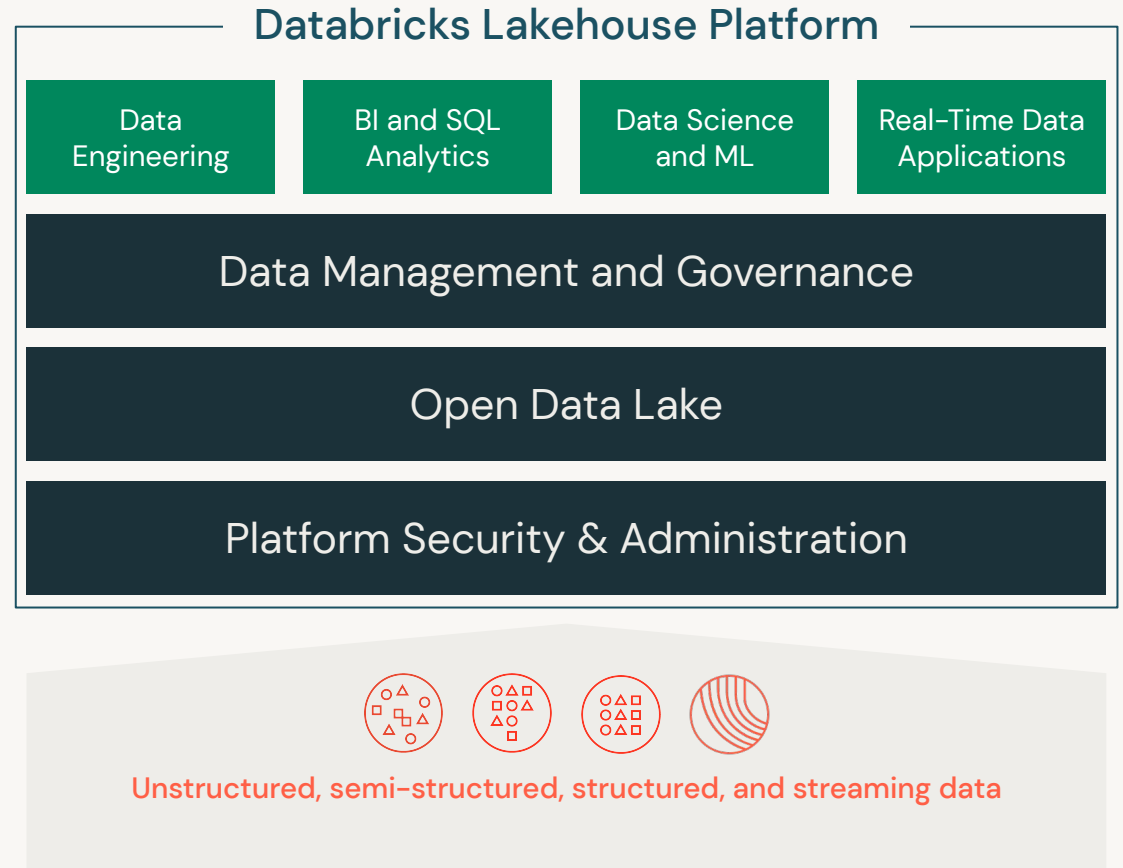


**Data
Warehouse**



The Databricks Lakehouse Platform

-  Simple
-  Open
-  Collaborative





Data Lake



DELTA LAKE

An open approach to bringing
**data management and
governance to data lakes**

Better reliability with transactions

48x faster data processing with indexing

Data governance at scale with
fine-grained access control lists



Data Warehouse



Delta Lake brings ACID to object storage

- Atomicity
- Consistency
- Isolation
- Durability



**Delta Lake provides ACID
guarantees scoped to tables**



Problems solved by ACID transactions in Delta

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



The Lakehouse Medallion Architecture



Multi-hop Pipeline

Source:

Files or integrated systems

Bronze:

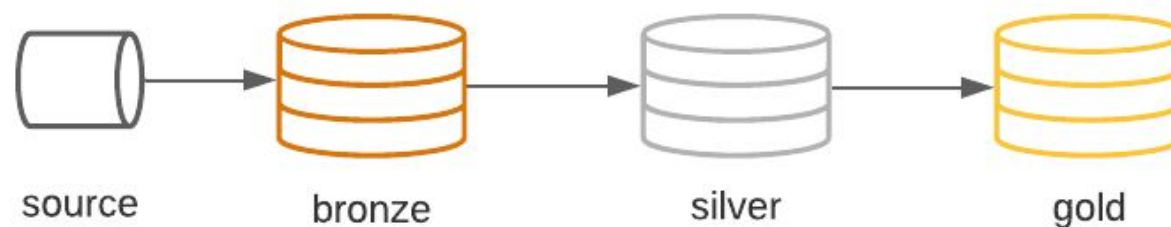
Raw data and metadata

Silver:

Validated data with atomic grain

Gold:

Refined, aggregated data



Bronze Layer



Why is the Bronze Layer Important?

- Bronze layer replaces the traditional data lake
- Represents the full, unprocessed history of the data
- Captures the provenance (what, when, and from where) of data loaded into the lakehouse
- Data is stored efficiently using Delta Lake
- If downstream layers discover later they need to ingest more, they can come back to the Bronze source to obtain it.



Bronze Layer Guiding Principles

- The goal of this layer is data capture and provenance:
 - Capture exactly what is ingested, without parsing or change.
- Typically a Delta Lake table with these fields in each row:
 - Date received/ingested
 - Data source (filename, external system, etc)
 - Text field with raw unparsed JSON, CSV, or other data
 - Other metadata
- Should be append only (batch or streaming)
- Plan ahead if data must be deleted for regulatory purposes



Processing Deletes

- Retain all records when possible
- Soft-deletes if necessary
- Hard-deletes may be required by regulatory processes



Silver Layer



Why is the Silver Layer important?

- Easier to query than the non-curated Bronze “data lake”
 - Data is clean
 - Transactions have ACID guarantees
- Represents the “Enterprise Data Model”
- Captures the 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



Silver Layer Guiding Principles

- Uses Delta Lake tables (with SQL table names)
- 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



Gold Layer



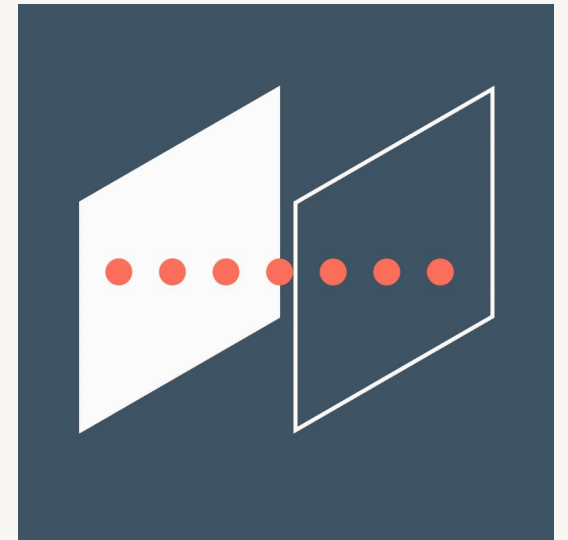
Why is the Gold Layer important?

- Powers ML applications, reporting, dashboards, ad hoc analytics
- Reduces costs associated with ad hoc queries on silver tables
- Allows fine grained permissions
- Reduces strain on production systems
- Shifts query updates to production workloads



Notebook

Streaming Design Patterns



Bronze Ingestion Patterns

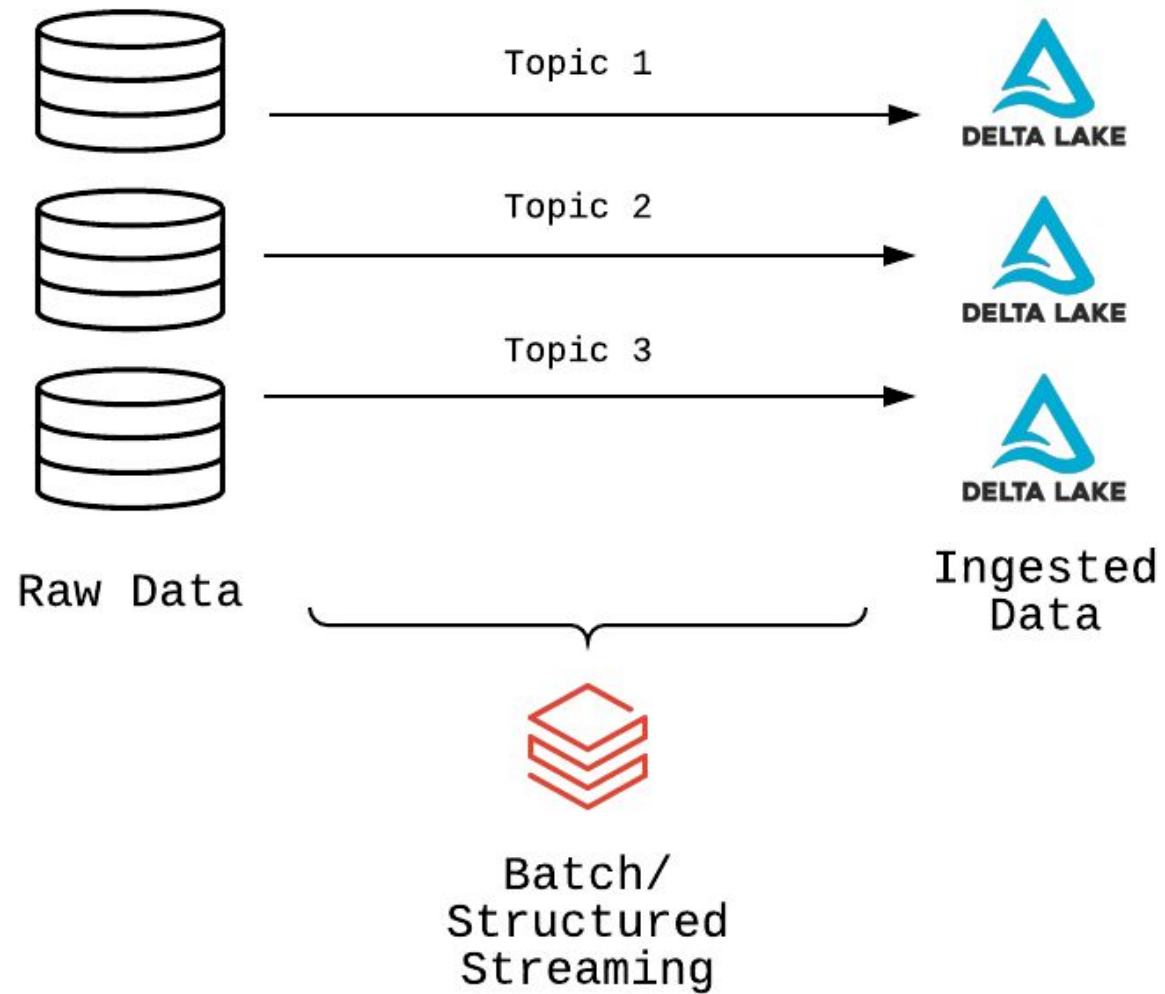
Bronze Ingestion Patterns
Auto Load to Multiplex Bronze
Streaming from Multiplex Bronze



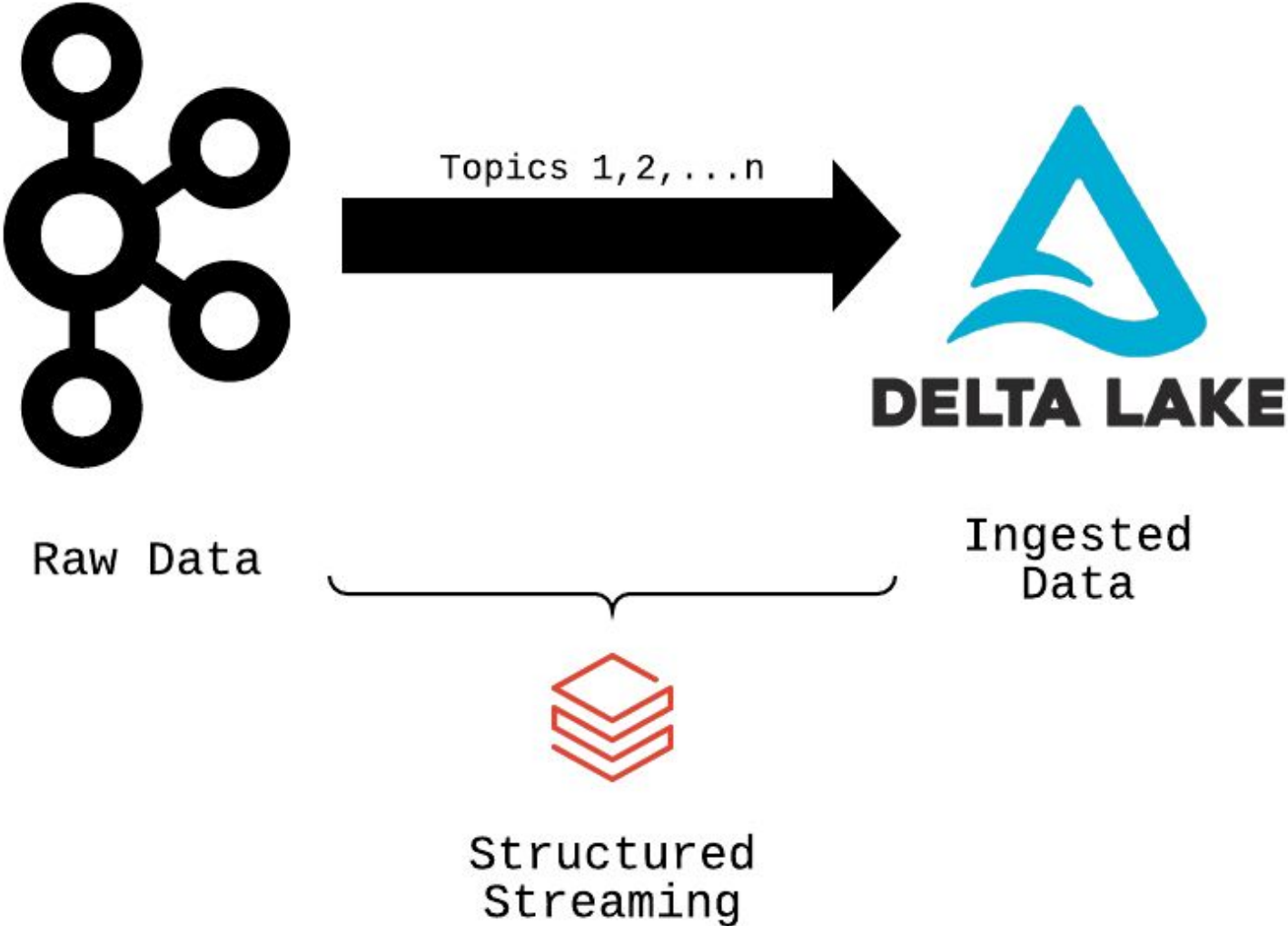
Bronze Ingestion Patterns



Singleplex Ingestion

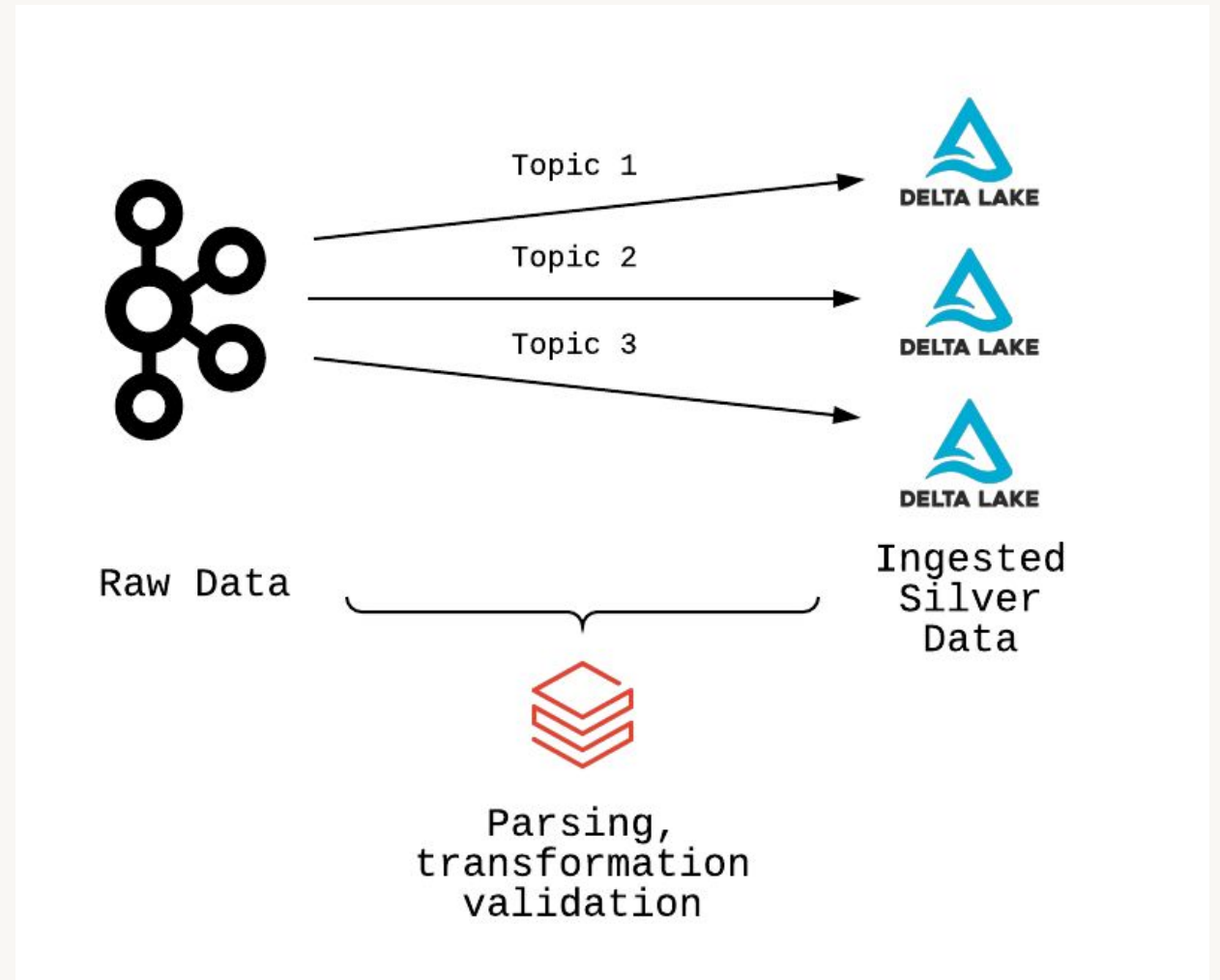


Multiplex Ingestion



Don't Use Kafka as Bronze

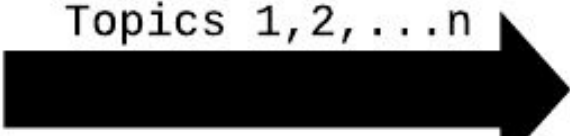
- Data retention limited by Kafka; expensive to keep full history
- All processing happens on ingest
- If stream gets too far behind, data is lost
- Cannot recover data (no history to replay)



Delta Lake Bronze

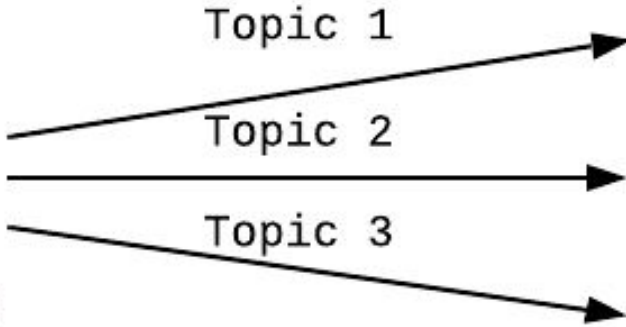


Raw Data



DELTA LAKE

Ingested
Bronze
Data



DELTA LAKE



DELTA LAKE



DELTA LAKE

Enriched
Silver
Data



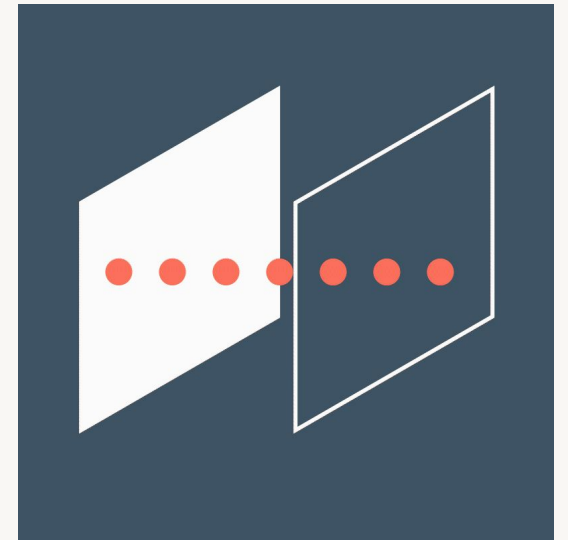
Ingest raw
data and
metadata



Parsing,
transformation
validation

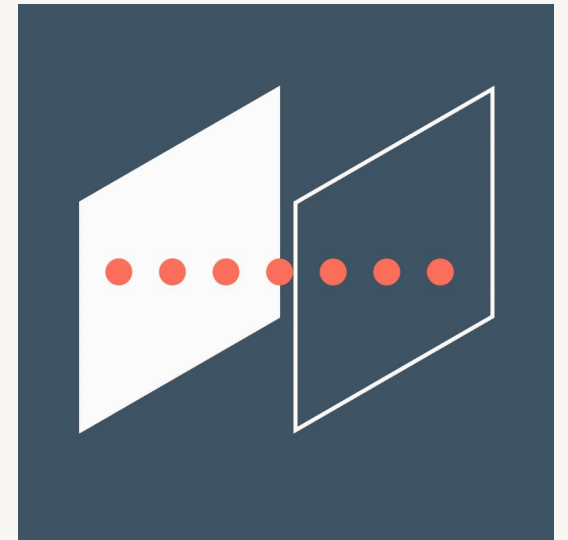
Notebook

Auto Load to Multiplex Bronze



Notebook

Streaming from Multiplex Bronze



Promoting to Silver

Streaming Deduplication
Quality Enforcement
Slowly Changing Dimensions
Streaming Joins and Statefulness



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;
```



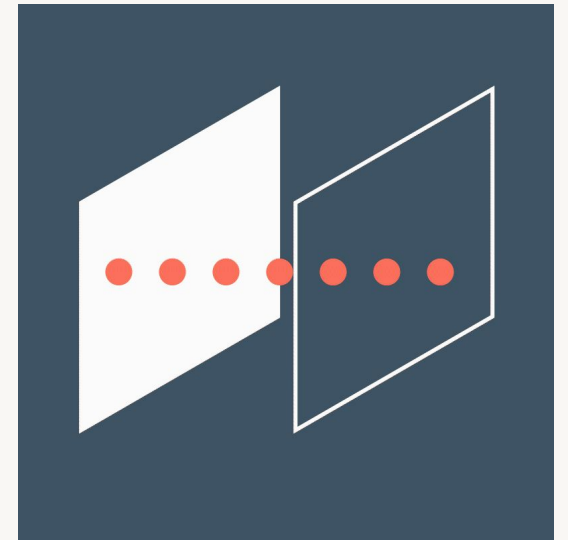
Alternative Quality Check Approaches

- 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



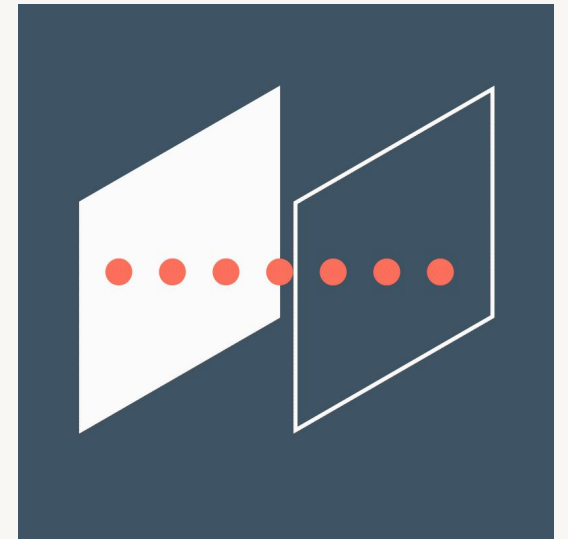
Notebook

Streaming Deduplication



Notebook

Quality Enforcement



Lab

Promoting to Silver



Slowly Changing Dimensions in the Lakehouse



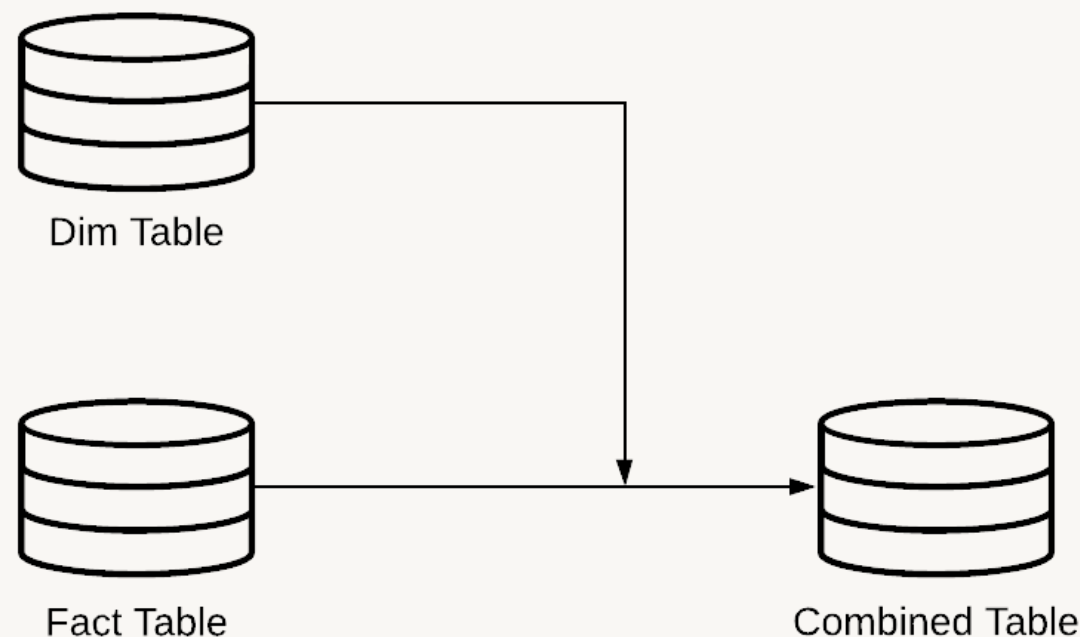
Fact Tables as Incremental Data

- 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

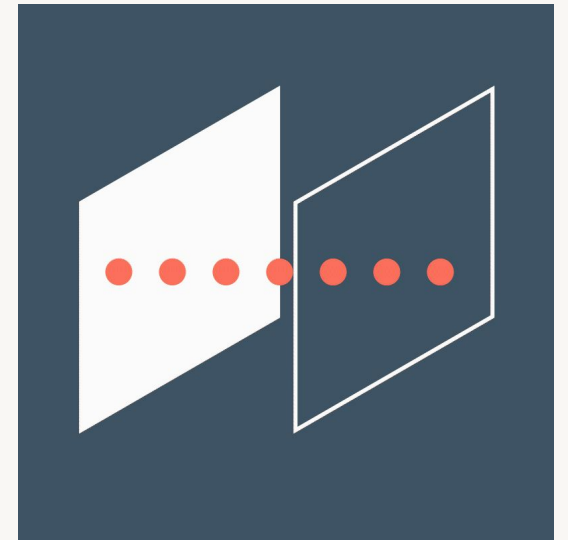
- Fact table usually append-only (Type 0)
- Can leverage event and processing times for append-only history

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

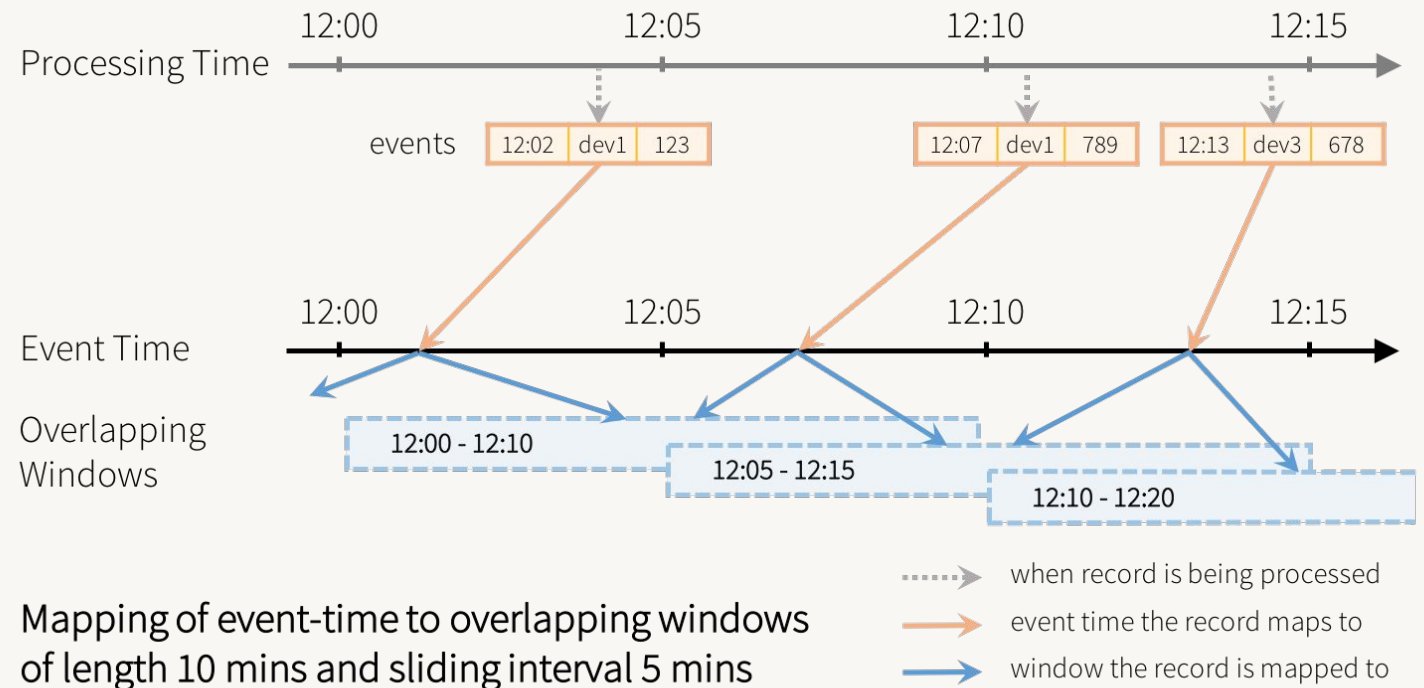


Streaming Joins and Statefulness



The Components of a Stateful Stream

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



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 are 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



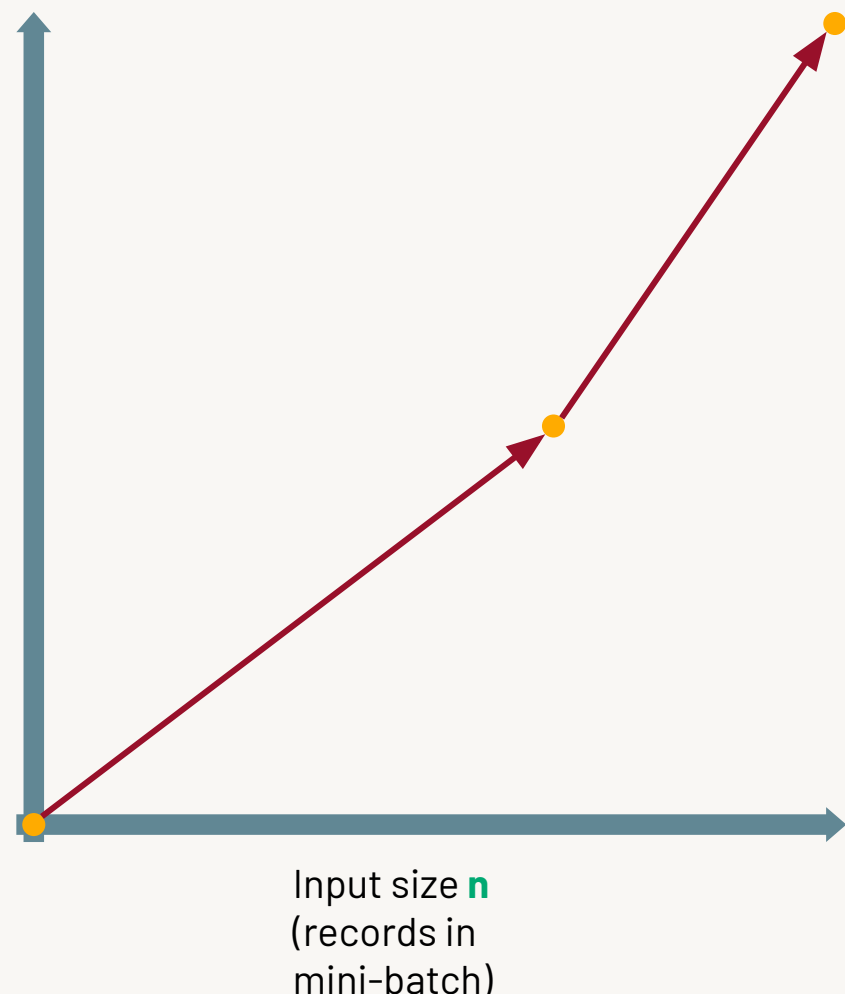
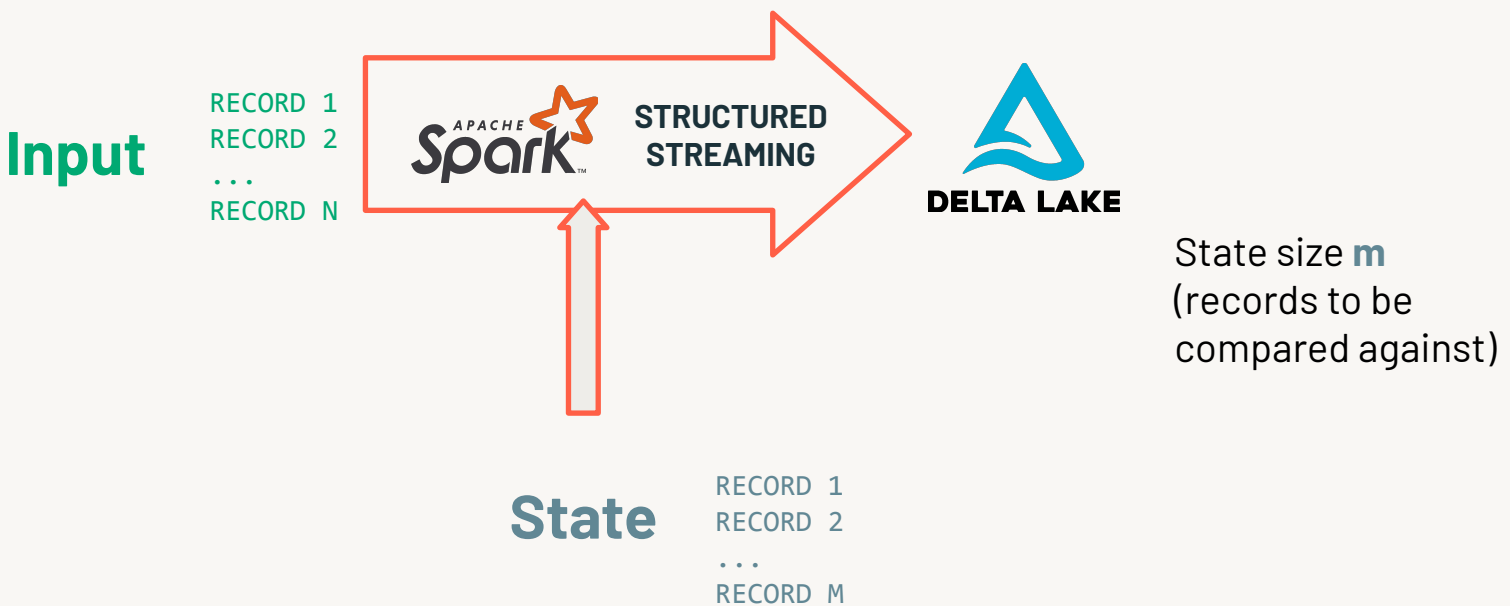
Managing Stream Parameters

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



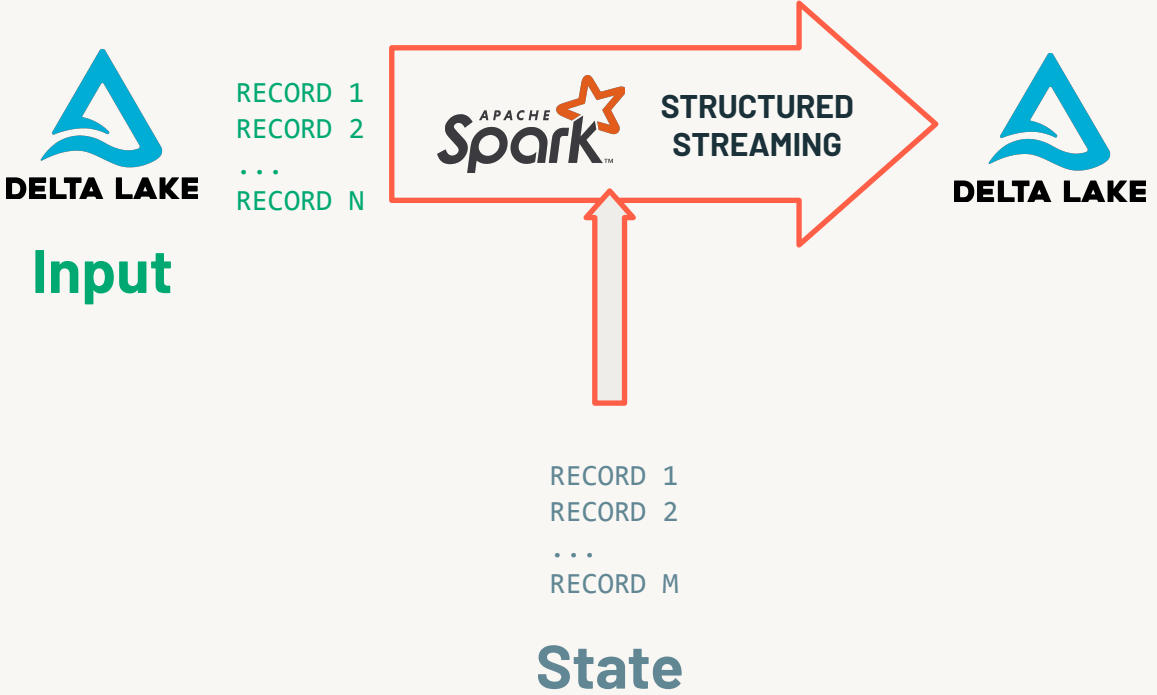
Reasoning about Stream Dimensions



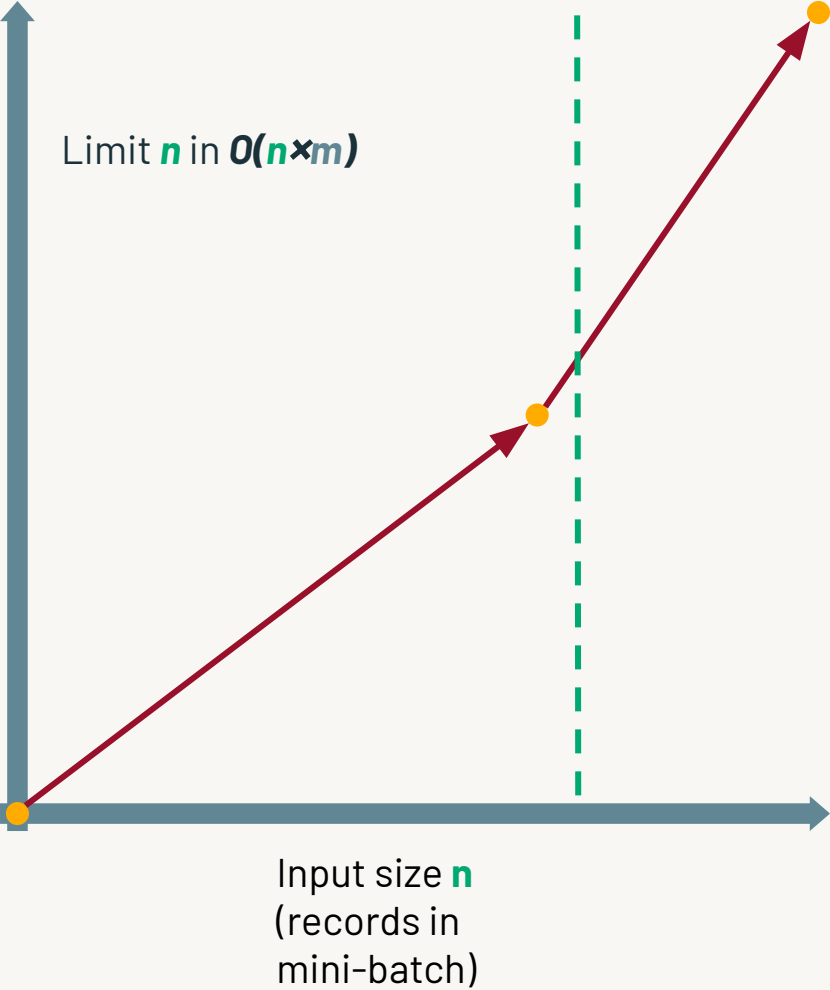
Input Parameters



Limiting the input dimension



State size m
(records to be compared against)



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

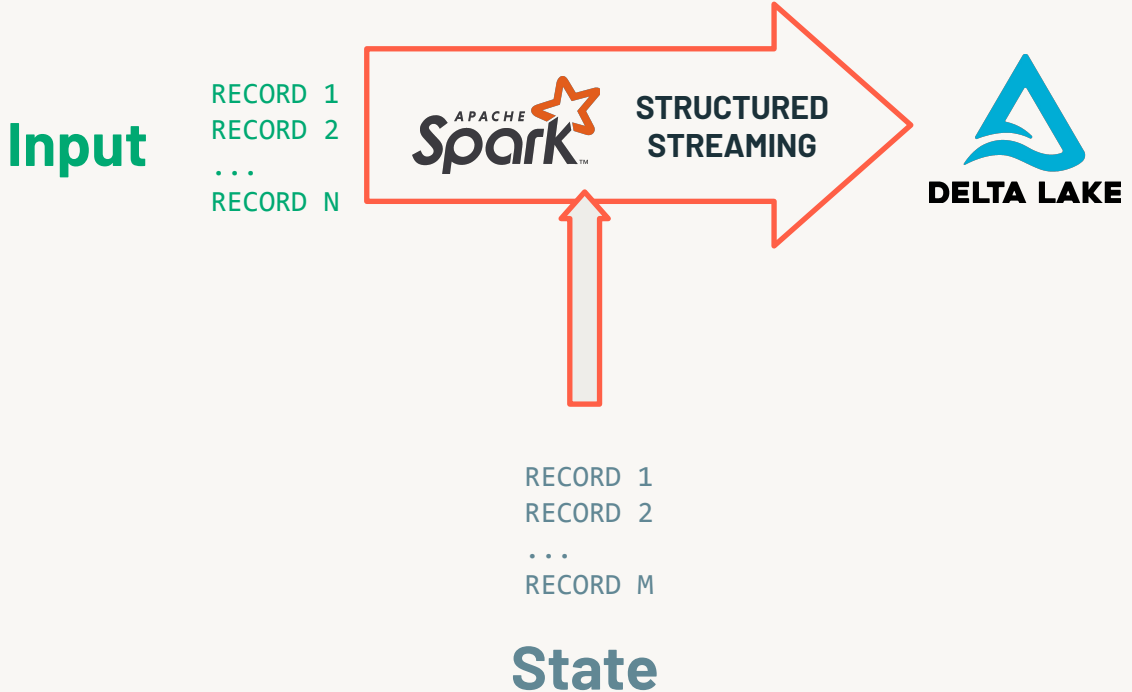
- **Rule of thumb 1:** Optimal shuffle partition size ~100–200 MB
- **Rule of thumb 2:** Set shuffle partitions equal to # of cores
- Use Spark UI to tune `maxFilesPerTrigger` until you get ~100–200 MB per partition
- Note: Size on disk is **not** a good proxy for size in memory
 - Reason is that file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



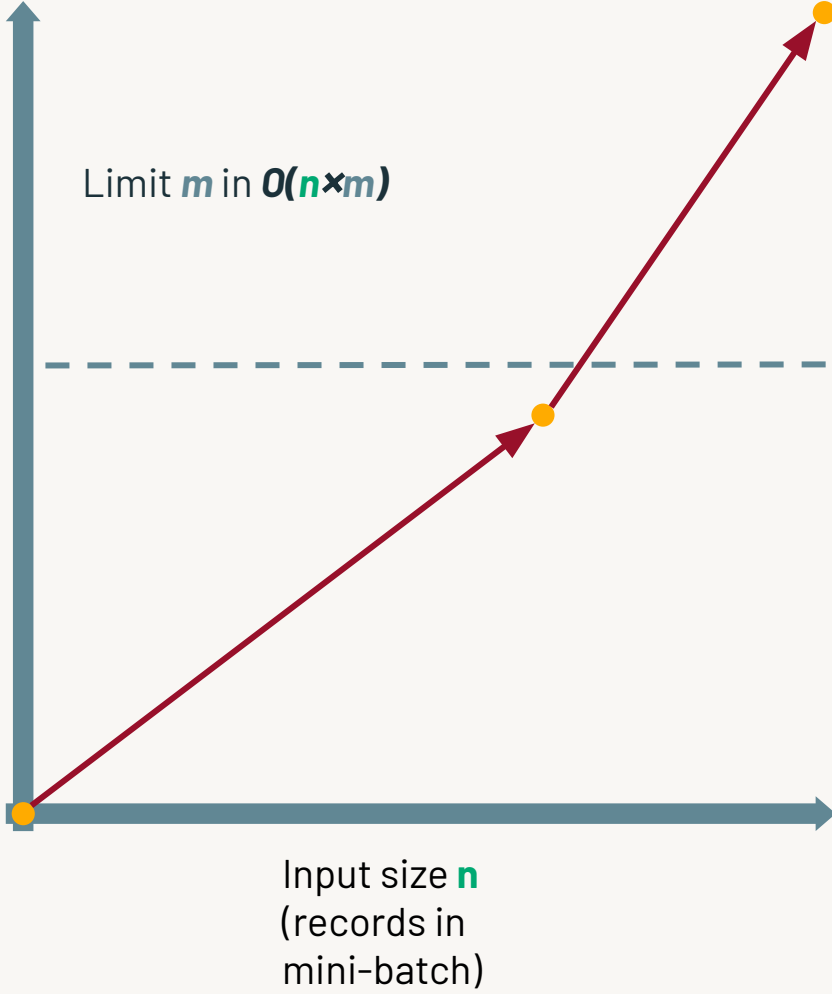
State Parameters



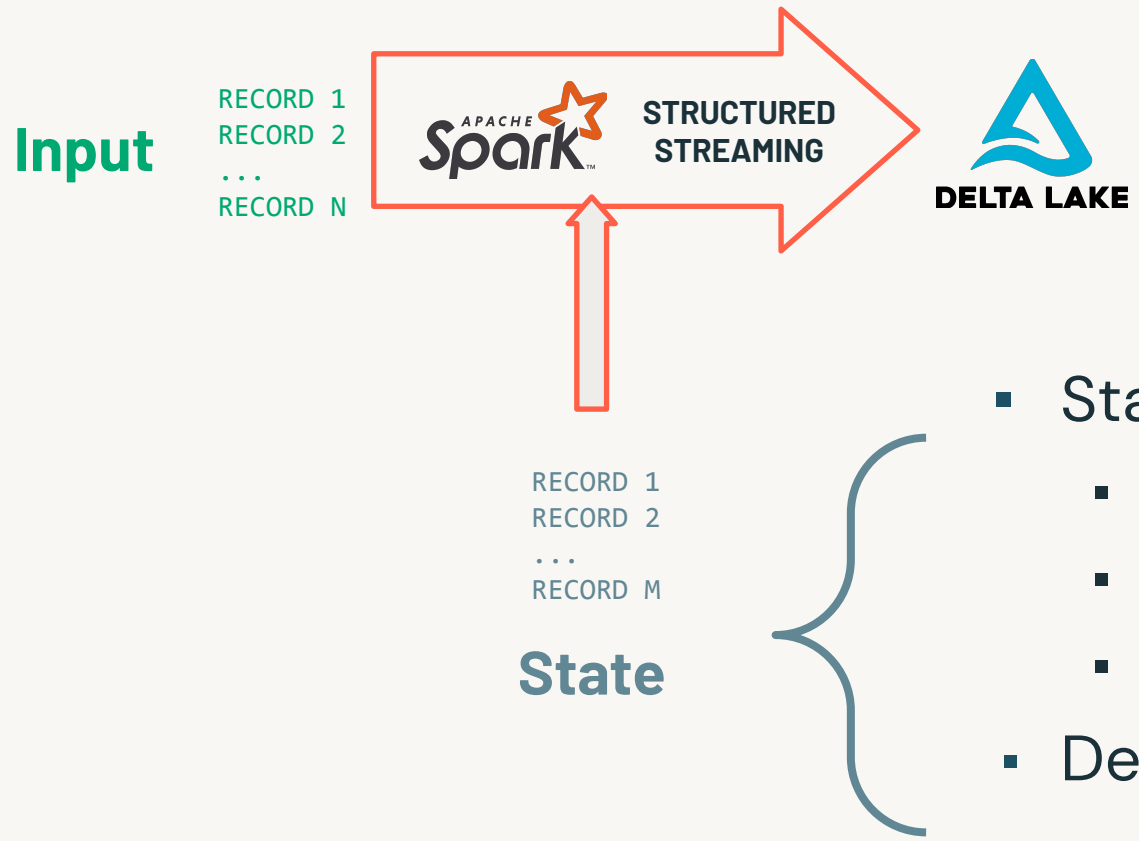
Limiting the state dimension



State size m
(records to be compared against)



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

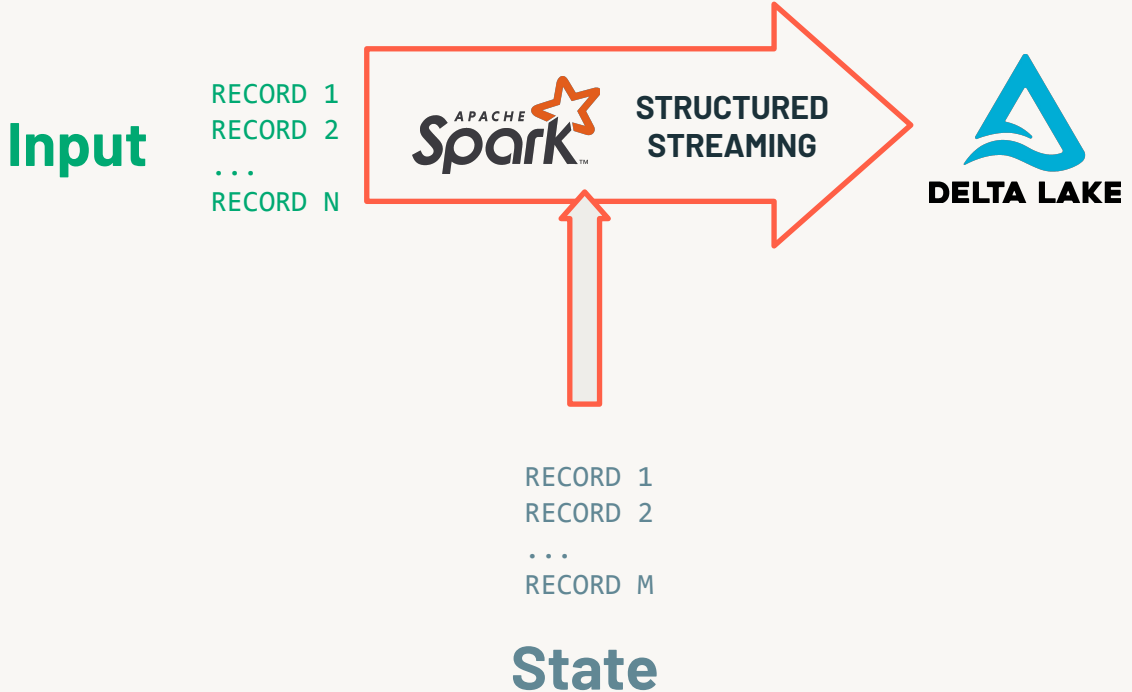
```
itemSalesSDF = (  
  salesSDF  
    .join(  
      spark.table("items")  
        .filter("category='Food'), # Predicate  
      on=["item_id"]  
    )  
)
```



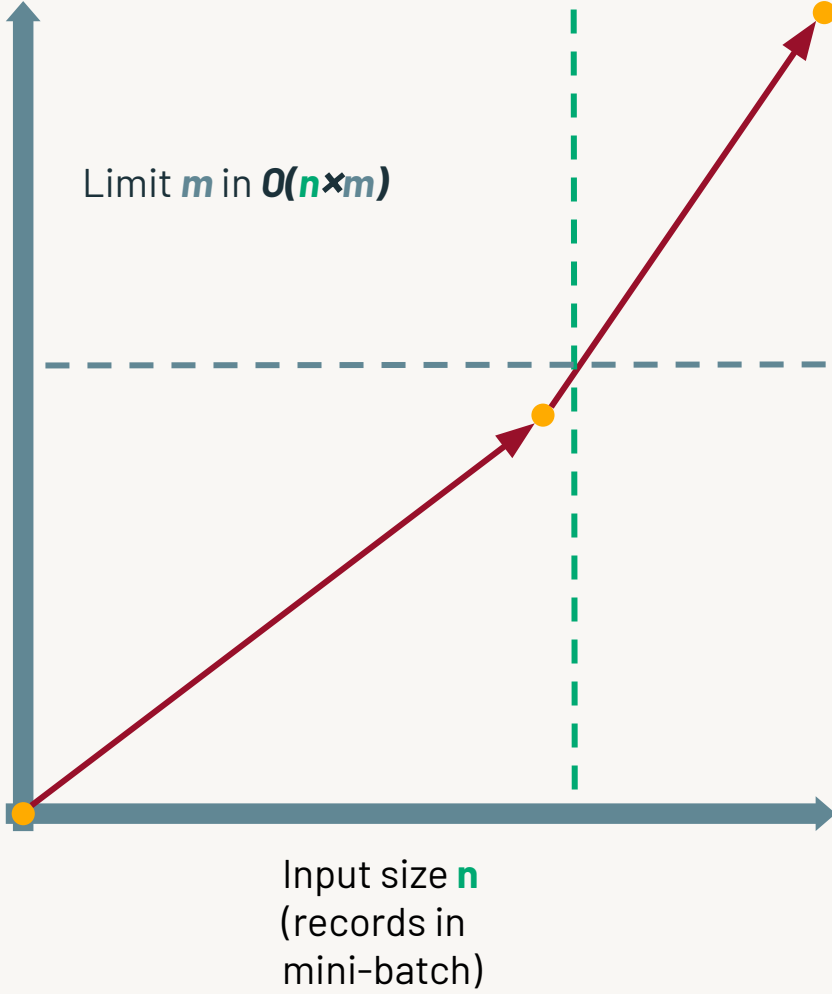
Output Parameters



Limiting the output dimension



State size m
(records to be compared against)



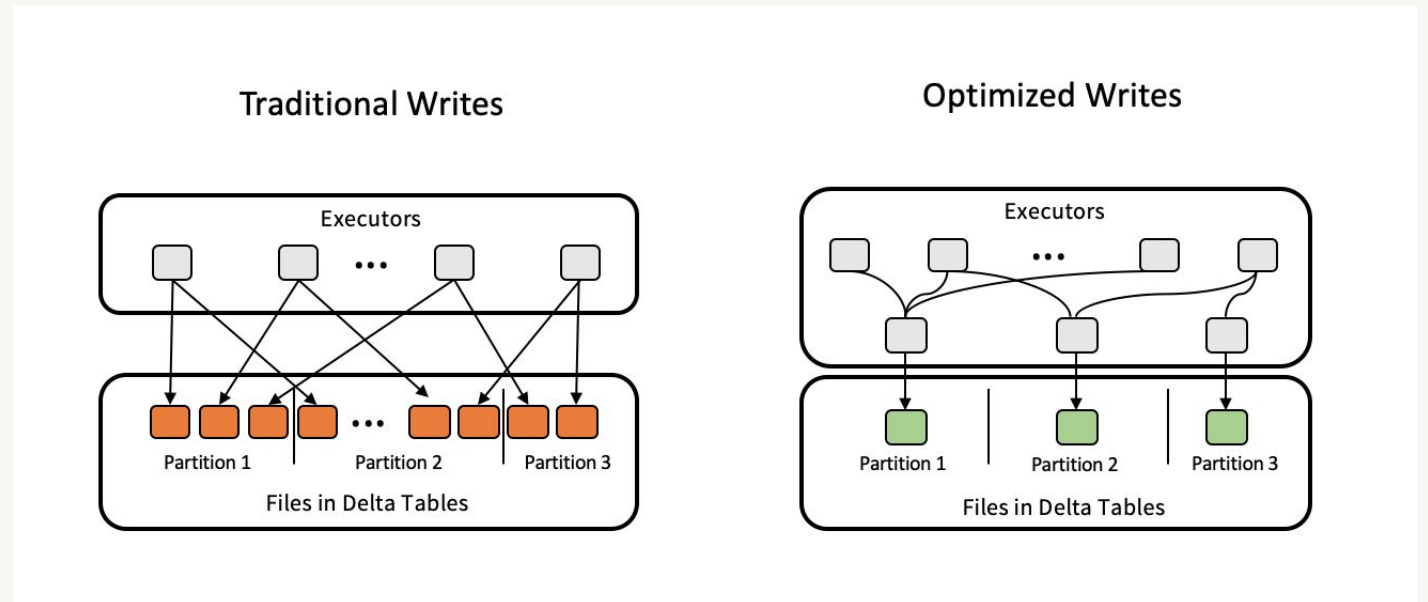
Why are output parameters important?

- 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



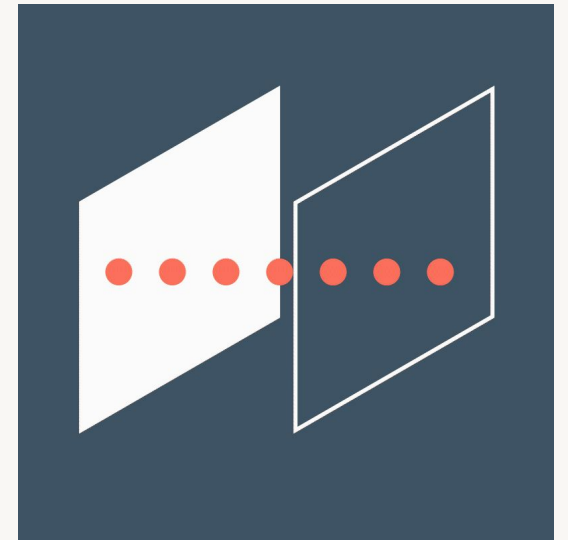
Delta Lake Output Optimizations

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



Notebook

Stream Static Joins



Gold Query Layer

Making Data Available for Analytics
Stored Views
Materialized Gold Tables



Lakehouse and the Query Layer



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
- Star-schemas and data marts for BI queries
- 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 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



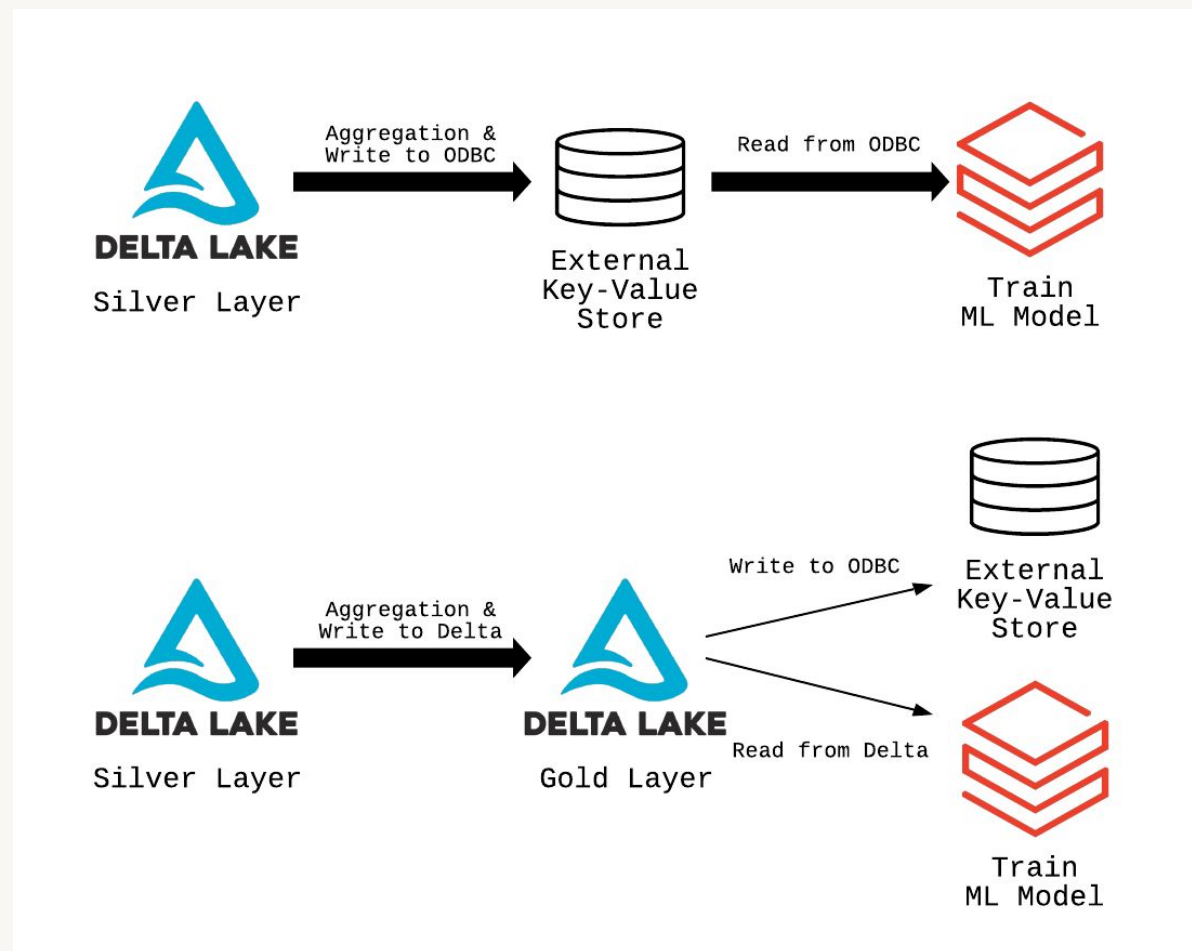
Databricks SQL Endpoints

- Clusters optimized for SQL queries
- Serverless option for quick cluster startup and autoscaling
- Photon-enabled for vectorized execution
- Enhanced throughput for exchanging data with external SQL systems
- Optimized connectors for popular BI tools



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

- Create views with column aliases

```
CREATE VIEW sales_marketing AS
SELECT
  id cust_id,
  date deal_date,
  total sale_amount
FROM sales;
```



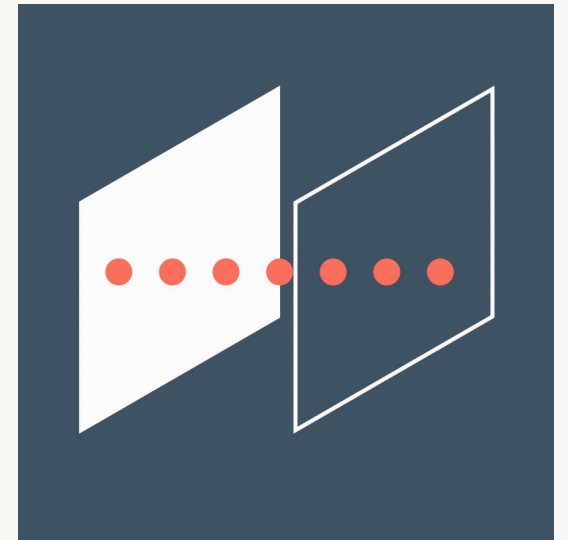
Recommendations

- 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



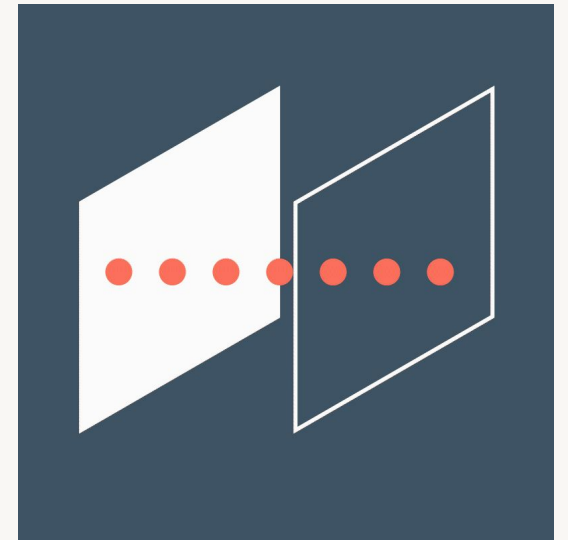
Notebook

Stored Views



Notebook

Materialized Gold Tables





Storing Data Securely

PII & Regulatory Compliance
Storing PII Securely
Granting Privileged Access to PII

PII & Regulatory Compliance



Regulatory Compliance

- EU = GDPR (General Data Protection Regulation)
- US = CCPA (California Consumer Privacy Act)
- Simplified Compliance Requirements
 - Inform customers what personal information is collected
 - Delete, update, or export personal information as requested
 - Process request in a timely fashion (30 days)



How Lakehouse Simplifies Compliance

- Reduce copies of your PII
- Find personal information quickly
- Reliably change, delete, or export data
- Use transaction logs for auditing



Manage Access to PII

- Control access to storage locations with cloud permissions
- Limit human access to raw data
- Pseudonymize records on ingestion
- Use table ACLs to manage user permissions
- Configure dynamic views for data redaction
- Remove identifying details from demographic views



Pseudonymization



Pseudonymization

- Switches original data point with pseudonym for later re-identification
- Only authorized users will have access to keys/hash/table for re-identification
- Protects datasets on record level for machine learning
- A pseudonym is still considered to be personal data according to the GDPR



Hashing

- Apply SHA or other hash to all PII
- Add random string "salt" to values before hashing
- Databricks secrets can be leveraged for obfuscating salt value
- Leads to some increase in data size
- Some operations will be less efficient



Tokenization

- Converts all PII to keys
- Values are stored in a secure lookup table
- Slow to write, but fast to read
- De-identified data stored in fewer bytes



Tokenization

Highest level of access control

Source

category	ip	email	ssn
cat1	a	b	c
cat2	e	f	g

kind	ip	email	ssn
k1	e	h	k
k2	e	f	i

Anonymization layer

key	value
ip	a
email	b
ssn	c
ip	e
email	f
ssn	g



kind	value	token
ip	a	1001
email	b	1002
ssn	c	1003
ip	e	1004
email	f	1005
ssn	g	1006

Safer exploration

category	ip	email	ssn
cat1	1001	1002	1003
cat2	1004	1005	1006

kind	ip	email	ssn
k1	1004	1007	1009
k2	1004	1005	1008



GDPR with vault

Data Erasure Requests
("right to be forgotten")



Highest level of access control



Thousands of tables

category	ip	email	ssn
cat1	1001	1002	1003
cat2	1004	1005	1006

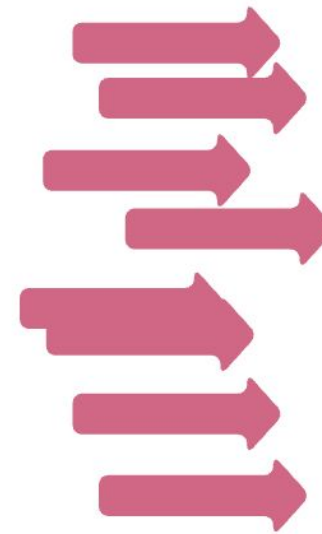
kind	ip	email	ssn
k1	1004	1007	1009
k2	1004	1005	1008

versus

GDPR without vault

Vault-based systems might be difficult to build, but without it coping with GDPR Data Erasure Requests might be, in fact, more difficult

Data Erasure Requests
("right to be forgotten")



Thousands of tables

category	ip	email	ssn
cat1	a78e	cd018e	23ffae
cat2	bce56f	baa661	ba43b4b

kind	ip	email	ssn
k1	010ab3	1007	ffaa887
k2	23aaef	b34eefd	baa678

Anonymization



Anonymization

- Protects entire tables, databases or entire data catalogues mostly for Business Intelligence
- Personal data is irreversibly altered in such a way that a data subject can no longer be identified directly or indirectly
- Usually a combination of more than one technique used in real-world scenarios



Data Suppression

- Exclude columns with PII from views
- Remove rows where demographic groups are too small
- Use dynamic access controls to provide conditional access to full data



Generalization

- Categorical generalization
- Binning
- Truncating IP addresses
- Rounding



Categorical Generalization

- Removes precision from data
- Move from specific categories to more general ones
- Retain level of specificity that still provides value



Binning

- Identify meaningful divisions in data and group on boundaries
- Allows access to demographic groups without being able to identify individual PII
- Can use domain expertise to identify groups of interest



Truncating IP addresses

- Rounding IP address to /24 CIDR
- Replace last byte with 0
- Generalizes IP geolocation to city or neighbourhood level

```
1 import spark._
2 val df = Seq("10.130.176.215", "10.5.56.45",
3             "10.208.126.183", "10.10.14.84",
4             "10.106.62.87", "10.190.48.173",
5             "10.141.100.19", "10.153.248.211",
6             "10.121.140.220", "10.37.240.84").toDF
7   .withColumn("ip_truncated",
8               concat(substring_index('value, ".", 3), lit(".0/24")))
9   display(df)
```

	value ▲	ip_truncated ▲
1	10.130.176.215	10.130.176.0/24
2	10.5.56.45	10.5.56.0/24
3	10.208.126.183	10.208.126.0/24
4	10.10.14.84	10.10.14.0/24
5	10.106.62.87	10.106.62.0/24
6	10.190.48.173	10.190.48.0/24
7	10.141.100.19	10.141.100.0/24

Showing all 10 rows.

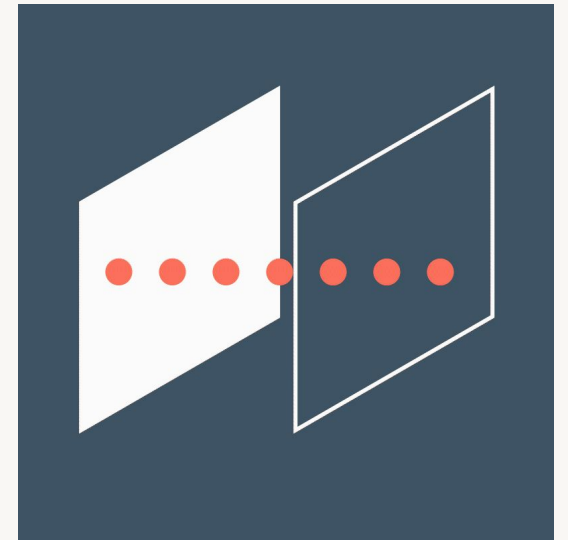
Rounding

- Apply generalized rounding rules to all number data, based on required precision for analytics
- Provides gross estimates without specific values
- Example:
 - Integers are rounded to multiples of 5
 - Values less than 2.5 are rounded to 0 or omitted from reports
 - Consider suppressing outliers



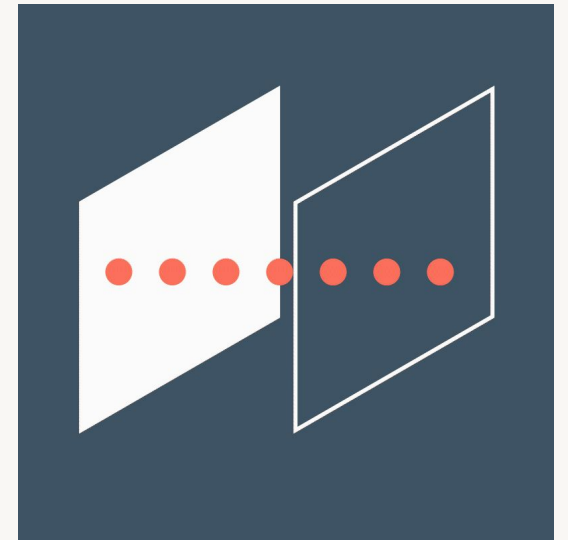
Notebook

Creating a Pseudonymized PII Lookup Table



Notebook

Storing PII Securely





Managing ACLs for the Enterprise Lakehouse



Challenges with Analytics in the Data Lake

- Difficult to provide access on need-to-know basis
- Unclear governance & ownership of data assets
- Data proliferation resulting in possible compliance issues
- Difficult to ensure integrity of data and reporting



The Goal

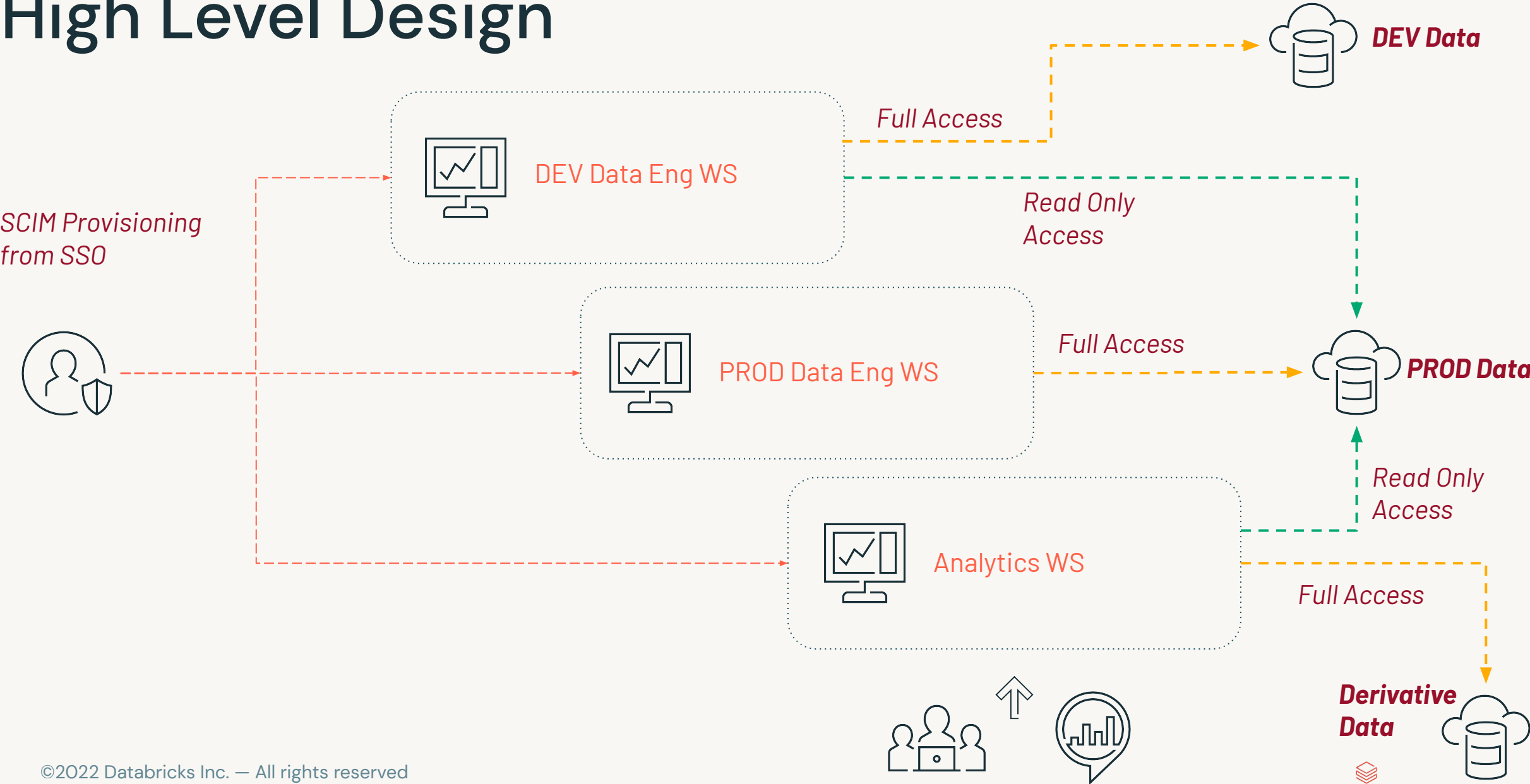
Provide access to valuable data to users across the company in a secure manner.

Ensure users are only able to access the data they're entitled to.

Detect whether any data has been altered or manipulated.



High Level Design



Grant Access to Production Datasets

Assumptions

- End-users need read-only access
- Datasets organized by database

```
GRANT USAGE, SELECT, READ_METADATA ON DATABASE hr TO
```

```
`HR`;
```

Alternative, grant access on specific tables:

```
GRANT USAGE ON DATABASE hr TO `HR`;
```

```
GRANT SELECT, READ_METADATA ON TABLE employees TO `HR`;
```

```
GRANT SELECT, READ_METADATA ON TABLE addresses TO `HR`;
```



Enable Secure Data Sharing

Assumptions

- Teams/depts need a private area to collaborate in
- Datasets must not be shared outside team/dept
- New tables are not automatically shared to other members

```
GRANT USAGE, CREATE ON DATABASE project_data TO `Data Analysts`;
```

Alternatively, members automatically see all new tables:

```
GRANT USAGE, SELECT, READ_METADATA, CREATE ON DATABASE project_data TO `Data Analysts`;
```



Enable Private User Sandboxes

Assumptions

- Users need a private area to store derivative datasets
- Datasets must not be shared with other users

```
GRANT USAGE, CREATE ON DATABASE user1 TO  
`user1@databricks.com` ;
```



Delegate Administration of Access Policies

Assumptions

- Need to delegate management of ACLs to data stewards/owners
- Monitor using SQL Analytics query logs and/or workspace audit logs

```
ALTER DATABASE hr OWNER TO `HR Admins`;
```



Delegate Database Creation to Trusted Users

Assumptions

- Delegate database creation to trusted users
- Database creators manage ACLs
- Monitor grants using SQL Analytics query logs and/or workspace audit logs

```
GRANT CREATE ON CATALOG TO `Data Admins` ;
```



Give All Users Read-Only Access To All Datasets

Assumptions

- Simple security model where all users have same level of access
- All new datasets are automatically accessible

```
GRANT USAGE, SELECT, READ_METADATA ON CATALOG TO `users` ;
```



Dynamic Views on Databricks

- Need to redact fields based on user's identity
- Do not give access to underlying table, only view
- Uses existing group membership to filter rows or columns



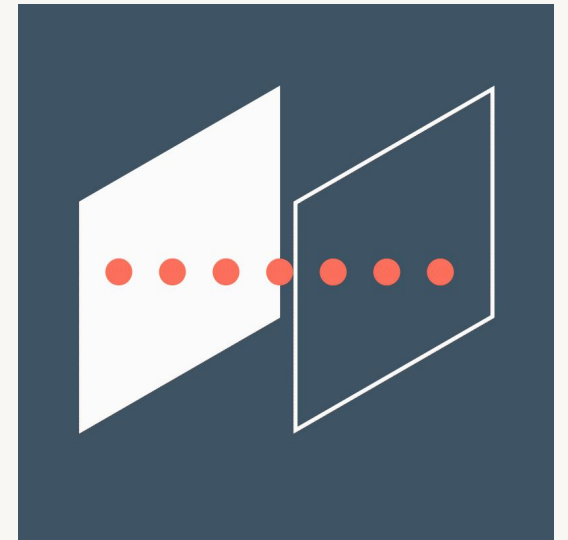
Rollout Plan

- SCIM provisioning with groups
- Identify & classify initial set of datasets
- Register in new workspace and apply ACLs
- For Fine-Grained
 - Define access policies (e.g. “Sales only see their own accounts...”, “Support technicians can only see accounts within their region”, etc.)
 - Identify missing attributes needed for dynamic views
- Lifecycle policy for derivative datasets
- Define process to “promote” new or derivative datasets
 - Need to classify data & define access policies
 - Legal & compliance review



Notebook

Deidentified PII Access





Propagating Updates and Deletes

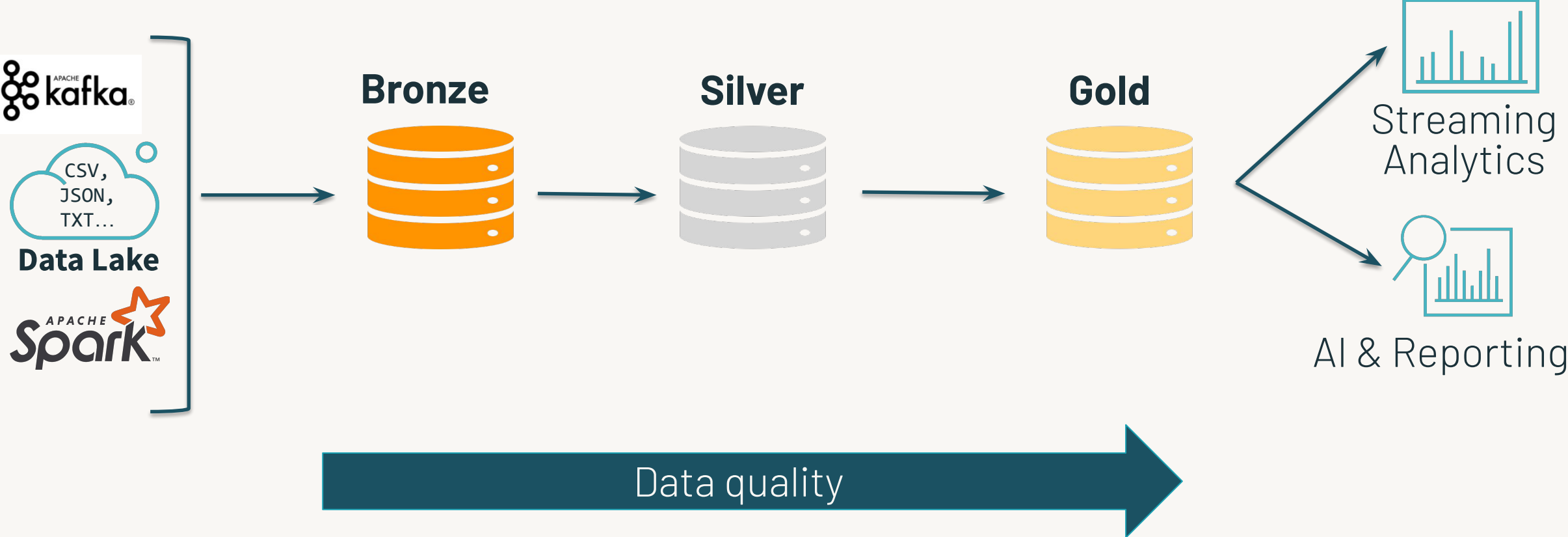
Processing Records from Change Data Feed
Deleting Data in the Lakehouse



Propagating Changes with Delta Change Data Feed



Multi-Hop in the Lakehouse



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

ignoreChanges

Allows deletion of full partitions

No new data files are written with full partition removal

ignoreDeletes

Allows stream to be executed against Delta table with upstream changes

Must implement logic to avoid processing duplicate records

Subsumes ignoreDeletes

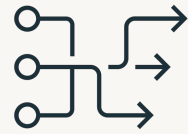


What Delta Change Data Feed Does for You



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



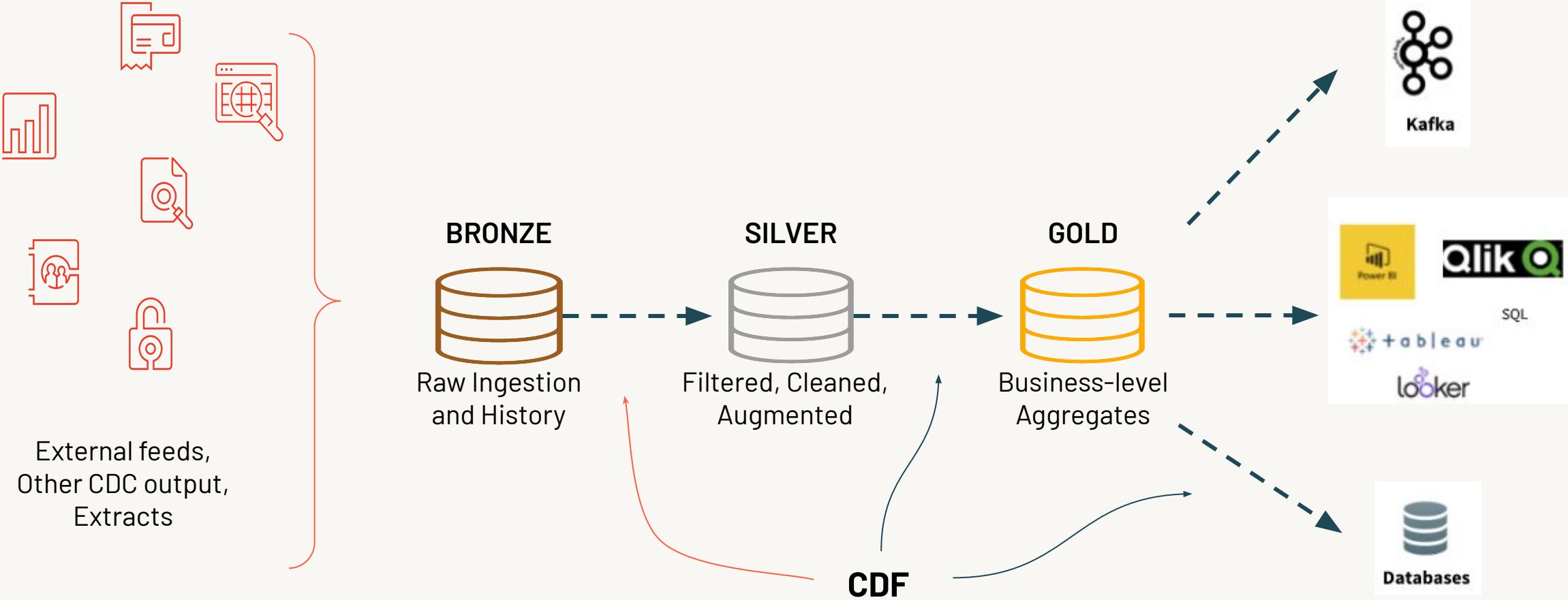
Meet regulatory needs

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

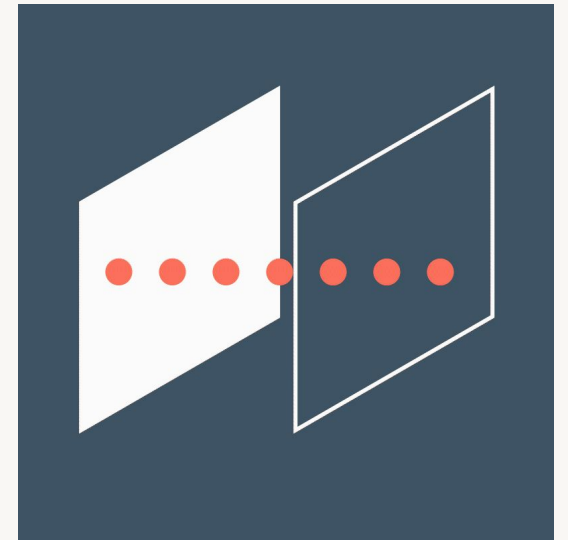
PK	B	Change Type	Time	Version
A2	B2	Preimage	12:00:00	2
A2	Z2	Postimage	12:00:00	2
A3	B3	Delete	12:00:00	2
A4	B4	Insert	12:00:00	2

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



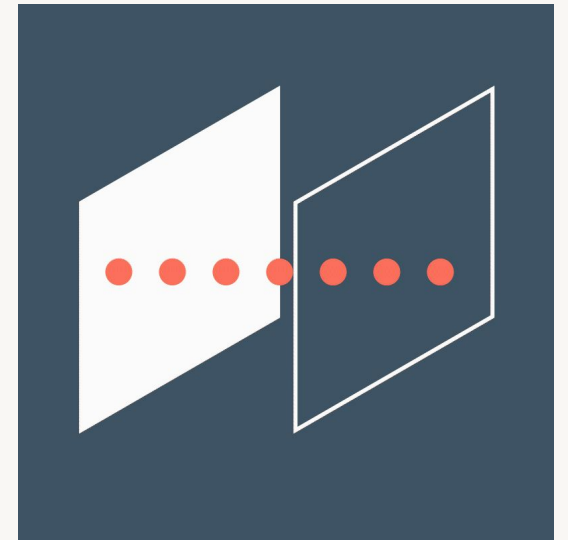
Notebook

Processing Records from Change Data Feed



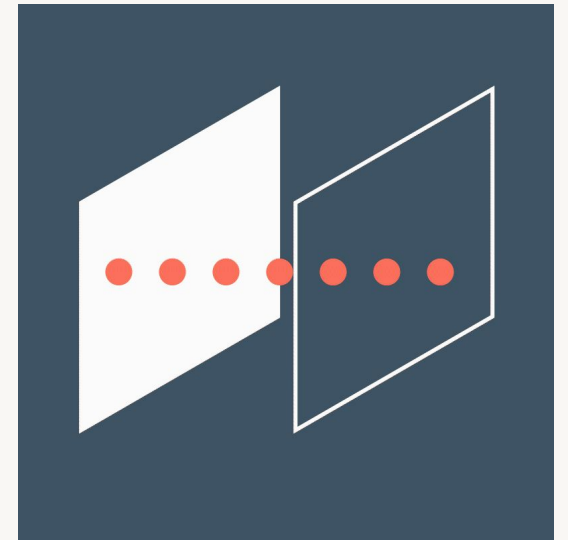
Notebook

Propagating Deletes with Change Data Feed



Notebook

Deleting at Partition Boundaries



Orchestration and Scheduling

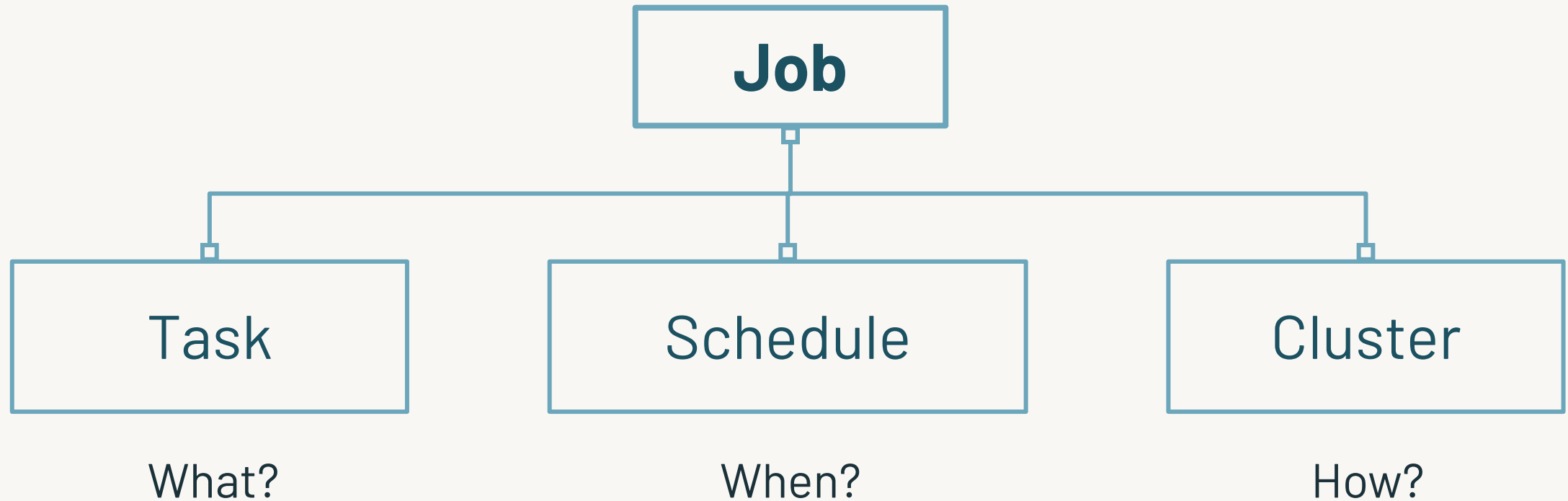
Multi-Task Jobs
Promoting Code with Repos
CLI and REST API
Deploying Workloads



Orchestration and Scheduling with Multi-Task Jobs



What is a Job?

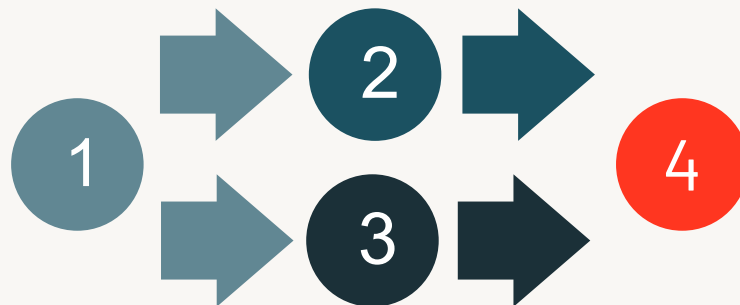


Orchestration with Multi-Task Jobs

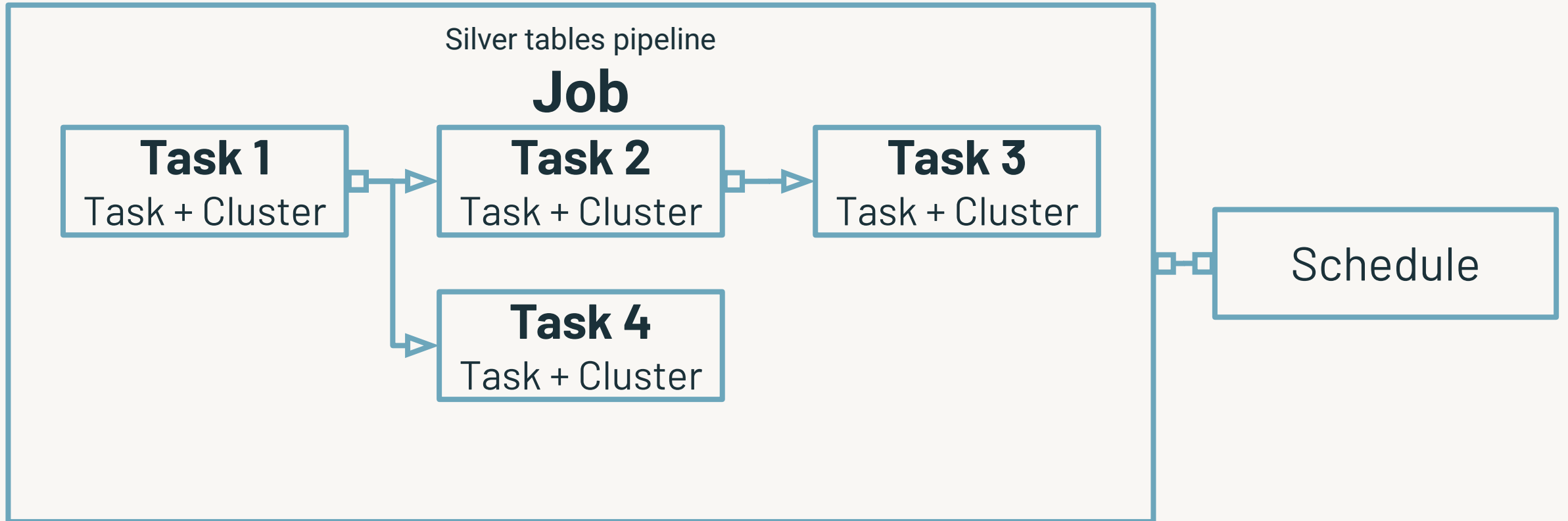
Serial



Parallel

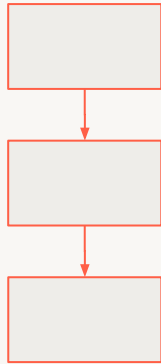


Jobs revisited



Common Jobs Patterns

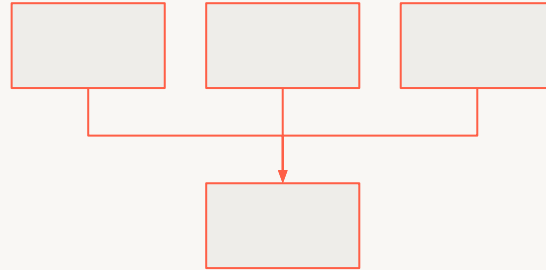
Sequence



Sequence

- Data transformation/processing/cleaning
- Bronze/Silver/Gold

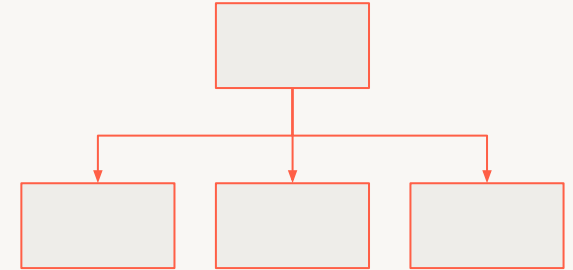
Funnel



Funnel

- Multiple data sources
- Data collection

Fan-out



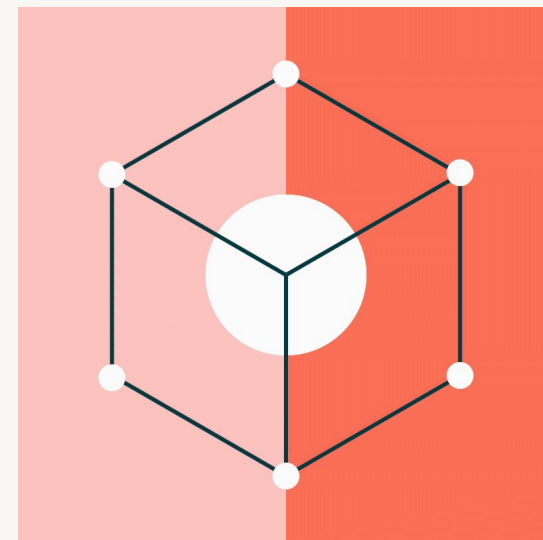
Fan-out, star pattern

- Single data source
- Data ingestion and distribution

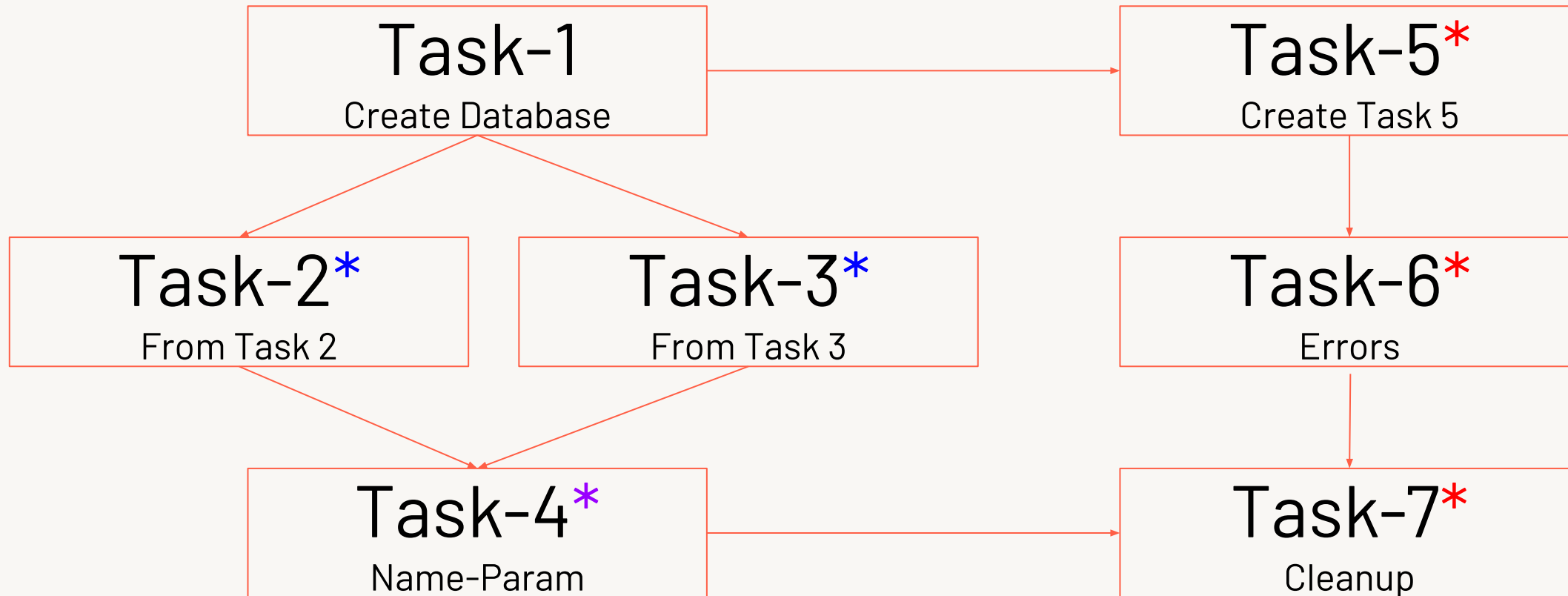


Demo

Creating a Multi-Task Job



Jobs UI Lab

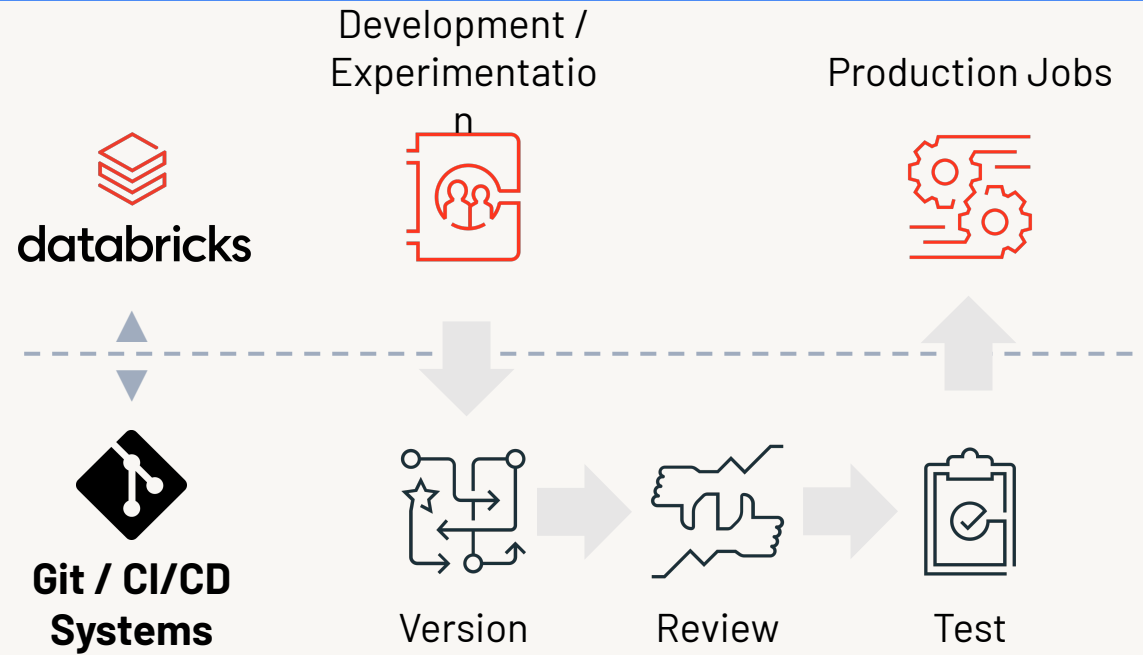


- * Make sure to use a **different** cluster for Tasks #2 & #3
- * Add the parameter "**name**" with some value (e.g. your name)
- * Make sure to use the **same** cluster for Tasks #5, #6 & #7
- * After running once with the error, update Task-6 to pass

Promoting Code with Databricks Repos



CI/CD Integration



Supported Git Providers



Enterprise Readiness

Admin Console | Workspace Settings

Enable Repos Git URL Allow List: **Disabled** [Enable](#)

[What this means >](#)

Repos Git URL Allow List: **Empty list**

[Save](#)

[What this means >](#)



Lab

Import a Git Repo



<https://github.com/databricks-academy/cli-demo>



Lab

Orchestration with the Databricks CLI



Lab

Using the Databricks REST API



Lab

Deploying Batch and Streaming Workloads

