CSCI E-103

Data Engineering for Analytics to Solve Business Challenges

Data Transformations

Lecture 04

Anindita Mahapatra & Eric Gieseke

Harvard Extension, Fall 2025

Agenda

- Common Design Patterns
- Compliance: GDPR, CCPA and other variants
- Concepts (Partitions, Z-Order, Spark Joins)
- APIs & UDFs
- CRUD (Create Read Update Delete)
- CDC (Change Data Capture)
- SCD (Slowly changing Dimensions)

Review Past Lecture

2 streaming types - File based & …..	Event based
What is checkpointing ?	Recover from failure by remembering the last known offset processed
What is watermarking	How long to tolerate arrival of late data before closing the gates and dropping data
Streaming output modes	append/overwrite/update
What is Processing trigger?	The microbatch interval in the readStream
Lambda Vs Kappa	Slow & Fast Processing Vs 1 pipeline
Data Warehouse + Data Lake =	Lakehouse
Managed Vs External Tables	Location
Metadata brings better discoverability &	Governance
What is a materialized view	Pre-computed query with storage

Big Data Design Patterns

- Design Patterns
  - Reusable patterns or templates that help address common problems
  - Promote design principles
    - Abstraction, Divide and Conquer, Separation of Concerns, Keep it Simple
  - Vocabulary for data personas to understand & share design/architecture
  - Choose design patterns that match the problem

- In Big Data, architectural patterns and design principles have emerged to address challenges around scale, velocity, volume, and data processing needs.
  - These patterns differ from the traditional Gang of Four (GoF) design patterns and focus on managing and processing large datasets.
  - GoF software design patterns are Object Oriented
    - Creational, Behavioral, Structural

Applying Design Patterns in Big Data Landscape

- Modeling
  - Star/Snowflake dimensional modeling
  - Vault dimensional modeling
- Ingestion
  - Multiple sources: Various Connectors
  - Speed : Separate Pipelines for Batch Vs Streaming ? (Lambda Vs Kappa)
  - Separation of Compute from Storage
- Transformations
  - Handling schema changes (schema on read Vs schema on write)
  - ACID Transactions - Updates/Deletes/Merges at scale
  - Multihop pipeline - different SLAs for different use case needs
- Persist
  - Compressed Columnar Storage formats
  - Denormalized Wide tables to avoid joins
  - Multiple Destinations: to accommodate diverse consumers
- Analytics
  - In-Stream analytics

Architectural Design Patterns

1. Lambda Architecture

- **Purpose**: Lambda Architecture is designed to process massive quantities of data both in batch and real-time, ensuring robustness and fault-tolerance.
- **Components**:
  - **Batch Layer**: Stores all the data and computes batch views on it (often using Hadoop, Spark, etc.).
  - **Speed Layer**: Provides real-time updates by processing data as soon as it is ingested (using tools like Apache Storm, Kafka Streams, etc.).
  - **Serving Layer**: Merges the output from the batch and speed layers and serves query results.
- **Use Case**: When both real-time data insights and historical analysis are required, like in fraud detection and financial monitoring.

2. Kappa Architecture

- **Purpose**: Kappa Architecture simplifies the Lambda pattern by focusing only on streaming data, discarding the need for a separate batch layer.
- **Components**:
  - **Stream Processing**: It treats all data as streams, processing them in real-time (often using Kafka, Apache Flink, etc.).
  - **Storage**: Data streams are stored in a scalable system that can replay or reprocess data when needed.
- **Use Case**: Ideal for use cases where real-time processing is sufficient, and batch processing can be avoided, such as IoT data processing.

3. Event-Driven Architecture (EDA)

- **Purpose**: This is an architectural pattern where systems respond to events (state changes or data inputs) in real-time.
- **Components**:
  - **Event Producers**: Create and publish events (e.g., sensors, applications).
  - **Event Consumers**: Process or react to those events (e.g., microservices, analytics engines).
  - **Event Streaming**: Platforms like Apache Kafka or Pulsar enable asynchronous communication and scalable message passing.
- **Use Case**: Event-driven architectures are often used in microservices systems or applications where systems must react quickly to changes, like in e-commerce or real-time analytics.

Storage Design Patterns

4. CQRS (Command Query Responsibility Segregation)

- **Purpose**: Separates reading and writing operations into distinct models, enhancing performance and scalability.
- **Components**:
    - **Command Model**: Handles writes or updates to the data.
    - **Query Model**: Handles reads and fetches from the data.
- **Use Case**: Particularly useful when you need to scale writes independently of reads, often used in event-sourcing systems where performance is critical.

5. Polyglot Persistence

- **Purpose**: A system or application uses different types of databases (SQL, NoSQL, Graph, etc.) based on the specific use case or query type.
- **Components**:
    - **Multiple Data Stores**: Using relational databases for transactional data, document stores for unstructured data, and graph databases for relationship-centric data.
- **Use Case**: Applications that require a mix of transactional consistency and high scalability, like e-commerce platforms, social networks, and large-scale analytics.

6. Data Lake Pattern

- **Purpose**: A storage repository that holds vast amounts of raw data in its native format until it is needed for processing.
- **Components**:
    - **Raw Storage**: Stores unstructured and structured data.
    - **Processing Engines**: Tools like Apache Hadoop, Spark, or Presto are used to process and analyze data from the lake.
- **Use Case**: Ideal for large enterprises that need to store and analyze a variety of structured and unstructured data, often used in combination with Machine Learning (ML) and advanced analytics.

Processing Design Patterns

7. Micro-Batch Processing

- **Purpose**: A hybrid approach between batch processing and stream processing, where small "batches" of data are processed in rapid succession, often treated as streams.
- **Components**:
  - **Batch Interval**: The frequency of data collection and processing (e.g., every second, minute).
  - **Data Processing Tools**: Spark Streaming, Flink, etc., allow for micro-batch processing.
- **Use Case**: This is useful when near real-time insights are needed, but the infrastructure for true stream processing isn’t in place.

8. CAP Theorem-Based Patterns

- **Purpose**: Big Data systems must deal with the trade-offs defined by the **CAP Theorem** (Consistency, Availability, Partition Tolerance), leading to patterns based on which properties the system prioritizes.
- **Components**:
  - **CP Systems**: Prioritize Consistency and Partition Tolerance, ensuring that data is correct, but might not always be available (e.g., HBase, Cassandra in some modes).
  - **AP Systems**: Prioritize Availability and Partition Tolerance, ensuring data is always available but may not be perfectly consistent in distributed setups (e.g., DynamoDB).
- **Use Case**: Useful when designing distributed data stores, depending on your application’s tolerance for eventual consistency or strong availability requirements.

9. Zero Data Loss (Zero Clone) Pattern

- **Purpose**: Ensures data durability and consistency by guaranteeing that no data is lost in the process of replication or streaming.
- **Components**:
  - **Data Stream Reprocessing**: The ability to reprocess streams ensures no data is dropped or missed.
  - **Message Replication**: Distributed messaging systems like Apache Kafka and Pulsar use multi-node replication to ensure no data loss.
- **Use Case**: Critical in financial, healthcare, and compliance-related applications where even a small amount of data loss can have severe consequences.

Serving Data Product Design Patterns

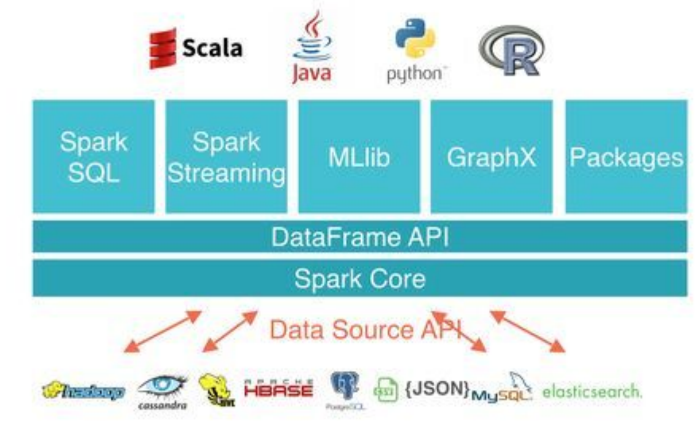
10. Data Mesh

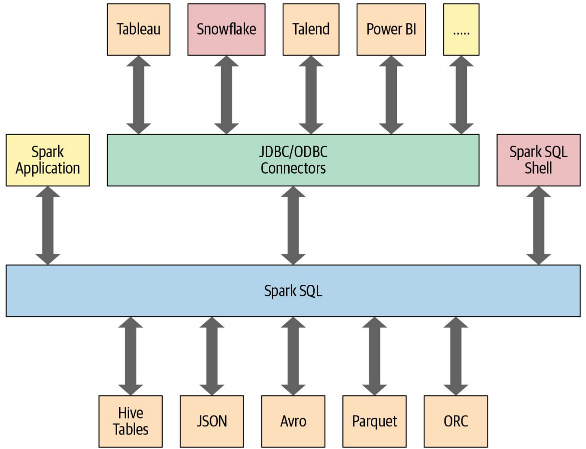
- **Purpose**: A decentralized approach to data architecture, enabling domain teams to manage their data as products while maintaining global standards for interoperability.
- **Components**:
  - **Domain-Oriented Data Ownership**: Each team manages its own data pipeline end-to-end.
  - **Self-Service Data Infrastructure**: Standardized tools for building and maintaining data products.
  - **Data Product Thinking**: Data is treated as a product, and the "producers" ensure it's high-quality and easily consumable.
- **Use Case**: Useful in large organizations where different teams generate and consume data independently, fostering cross-team collaboration.

11. Agentic architecture

- **Purpose**: Autonomous decision making in complex scenarios involving real time needs.
- **Components**:
  - **Tools & Agents** : Handles specialized tasks
  - **Models**: Predictive & generative functions.
- **Use Case**: Complex multi-step processes that benefit from dynamic decision-making and collaboration between agents..

Connector Pattern

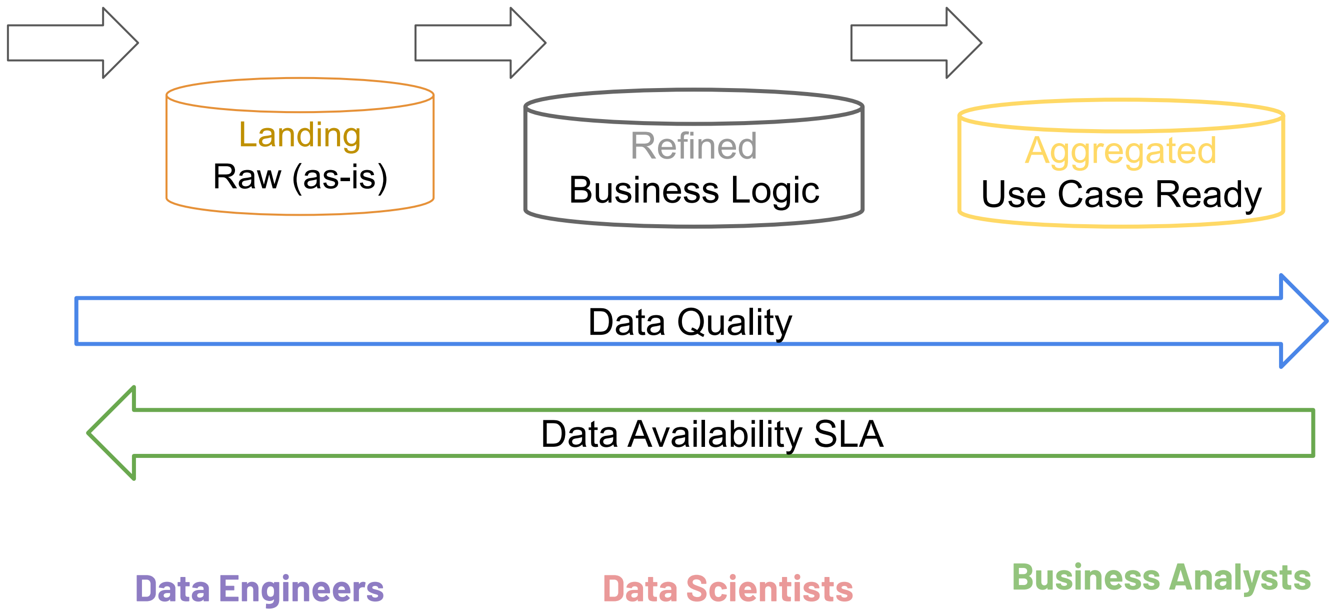




- **Single API**
- Multiple Sources/Sinks
- Multiple File Formats
- Multiple Languages
- Multiple Interfaces

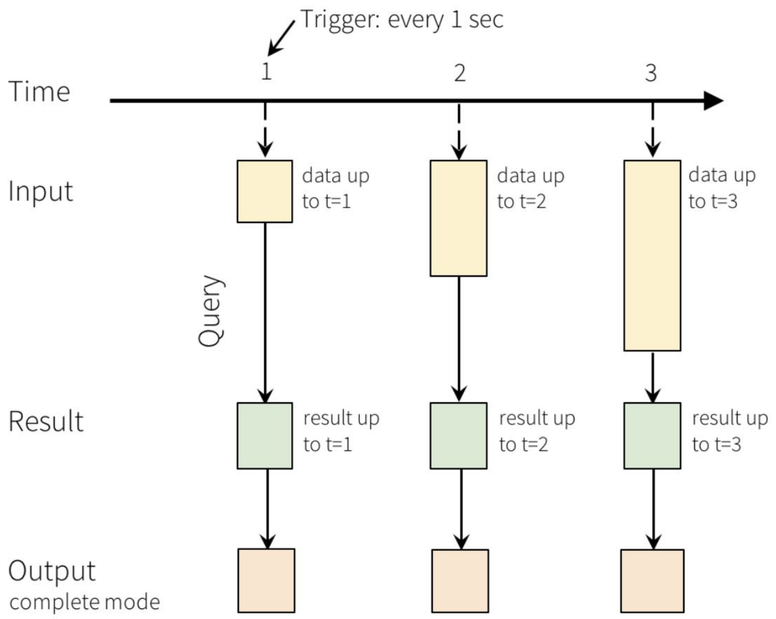
Also known as the “Bridge Pattern”, *decouple an abstraction from its implementation so that the two can vary independently*

Multi-Hop Access Pattern



- More robust to failures
- Modular structure helps in debugging
- Addresses different consumption needs
- Multiple personas work alongside on different parts of the pipeline

Near Real Time Pattern (Micro batching)



Requirements

- Resilient to Failures
- Manage state
- Exactly once semantics
- In-stream Analytics
- Handle late arriving data
- Single Pipeline

- Checkpointing
- Windowing
- Watermarking

Minimize Data Movement

- Time Travel
  - Maintaining data versions with timestamps
  - Users can access 'as of'
- Zero Copy Clone
  - Duplicate an object while neither creating a physical **copy** nor adding any additional storage costs.
  - The metadata repository still retains the record for all versions of the data set.
  - Delta Clone
    - Shallow Clone: metadata is copied
    - Deep Clone: both metadata and data are copied
  - Delta Share
    - No copy

Reduces:

- Data Storage Costs
- Data Access Costs
- Data Transfer Costs

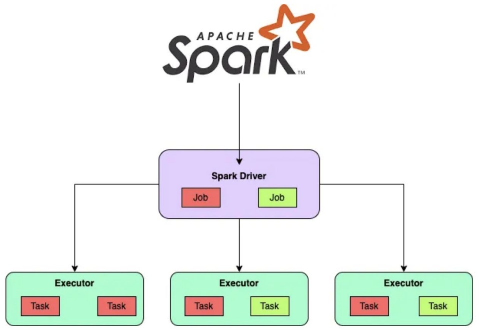
Avoid fragile disparate systems stitched together

Need to keep data in open format for multiple tools of the ecosystem to leverage

Compliance

- **GDPR** : General Data Protection Regulation
- **CCPA**: California Consumer Privacy Act
- What is it?
  - EU mandates local privacy laws for information of all its citizens even in other continents
  - Customer can demand his/her personal details to be erased from the system
  - It was introduced to standardize data protection law across the single market and give people in a growing digital economy greater control over how their personal information is used.
  - Organisations have to **ensure** that personal data is gathered legally and under strict conditions, but those who collect and manage it are obliged to protect it from misuse and exploitation, as well as to respect the rights of data owners - or face penalties for not doing so.
- Challenge
  - Org manage hundreds of terabytes worth of personal information
  - GDPR and CCPA compliance is of paramount importance to every
    - the right of erasure, or the right to be forgotten
    - the right of portability
- Requirement
  - Ability to locate and remove personal information
  - Fine grained updates/deletes as supported by Delta
  - "Pseudonymization," or reversible tokenization of personal information elements (identifiers) to keys (pseudonyms) that cannot be externally identified. destroy the linkage between the pseudonyms and identifiers
  - Maintenance of strict access and use policies

Spark Job



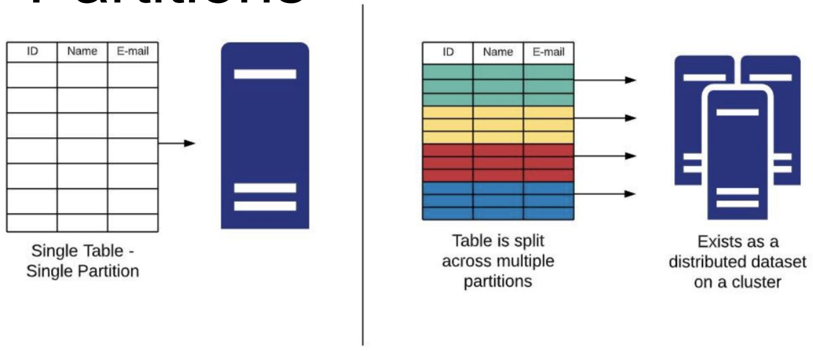
Jobs -> Stages -> Tasks

- **Jobs** are submitted to Spark through the driver program and are divided into smaller units called **stages** for execution.
- **Stage** is a logical unit of work, represents a set of tasks that can be executed together, typically resulting from a narrow transformation (e.g., `map` or `filter`) or a shuffle operation (e.g., `groupByKey` or `reduceByKey`).
  - Stages are determined by the Spark engine during the query optimization phase, based on the dependencies between RDDs (Resilient Distributed Datasets) or DataFrames.
- **Task** smallest unit of work in Spark, represents a single operation that can be executed on a partitioned subset of data.
  - Tasks are executed by the worker nodes in parallel, leveraging the distributed computing capabilities of Spark.
  - Each task operates on a portion of the data, applying the required transformations and producing intermediate or final results.

Table Partitions Vs Spark Partitions

	Table Partitioning (DB/Table level)	**Spark (Processing level)**
Concept/Scope	dividing a large table into more manageable pieces based on specific column values, such as date, region, or other business logic. Reduces data scan	Refers to how data is distributed across different nodes in a cluster during distributed processing.
Purpose	Data organization, query performance Physical data storage (files, directories)	Parallel processing, res optimization, fault tolerance. Logical division for processing
Control/ Management	Defined by the user at the time of table creation or data load (e.g., `CREATE TABLE ... PARTITIONED BY (...)`). Managed by user explicitly	Spark decides the number of partitions dynamically based on the input data and the configuration (e.g., the size of input files, number of tasks/executors). Managed by Spark, can be influenced by users
Opt	Enables **partition pruning**, which allows the query engine to avoid scanning unnecessary partitions based on the query predicates	Relates to **data shuffling** during transformations like `join()`, `groupBy()`, or `reduceByKey()`. Data from different partitions may need to be reorganized (shuffled) across nodes for accurate results, leading to network overhead.

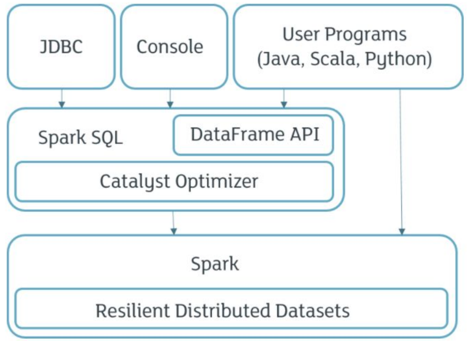
Partitions

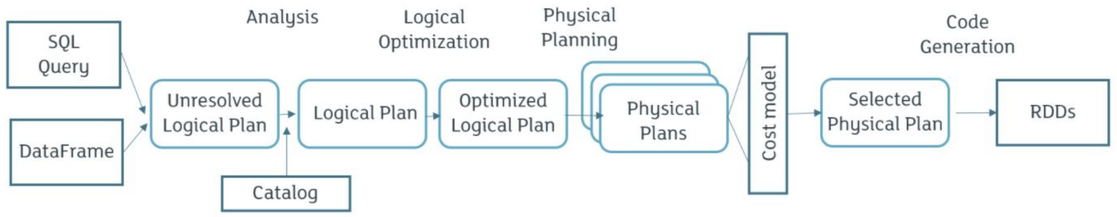

ID
Name
E-mail
Single Table - Single Partition
ID
Name
E-mail
Table is split across multiple partitions
Exists as a distributed dataset on a cluster

Data Partition?
To avoid full table scans when querying

Which columns to use? Best Practice

- Typically the most frequent columns in the where clause
- Do not use a column whose cardinality is very high
- Expect data in that partition to be at least 1 GB


JDBC
Console
User Programs (Java, Scala, Python)
Spark SQL
DataFrame API
Catalyst Optimizer
Spark
Resilient Distributed Datasets



- Distributed datasets exist as RDDs across multiple machines in the form of partitions.
- DataFrame is an abstraction of underlying RDDs sitting across multiple machines.
- APIs are a convenient way to manipulate data via DataFrames
- Partitions can run on worker node in parallel and are not splittable across nodes
- The number of partitions used in Spark is configurable, by default it should be set to number of available cores

Spark Partitions

SQL Reference

Example

```
%sql

USE salesdb;

CREATE TABLE customer(id INT, name STRING) PARTITIONED BY (state STRING, city STRING);

INSERT INTO customer PARTITION (state = 'CA', city = 'Fremont') VALUES (100, 'John');

INSERT INTO customer PARTITION (state = 'CA', city = 'San Jose') VALUES (200, 'Marry');

INSERT INTO customer PARTITION (state = 'AZ', city = 'Peoria') VALUES (300, 'Daniel');

SHOW PARTITIONS customer;

SHOW PARTITIONS customer PARTITION (state = 'CA', city = 'Fremont');

SHOW PARTITIONS customer PARTITION (city =  'San Jose');
```

Optimize, Restore, Vacuum

[Delta Cheat Sheet]

- Optimize
  - Bin-packing optimization (`spark.databricks.delta.optimize.maxFileSize Default=1G`)
  - Z-Ordering

```
OPTIMIZE events

OPTIMIZE events WHERE date >= '2017-01-01'

OPTIMIZE events WHERE date >= current_timestamp() - INTERVAL 1 day ZORDER BY (eventType)
```

- Restore

```
RESTORE TABLE db.target_table TO VERSION AS OF <version>

RESTORE TABLE delta.`/data/target/` TO TIMESTAMP AS OF <timestamp>
```

- Vacuum

```
VACUUM eventsTable   -- vacuum files not required by versions older than the default retention period
```

- Clone (shallow & deep)

```
CREATE OR REPLACE TABLE my_test SHALLOW CLONE my_prod_table;
CREATE OR REPLACE TABLE db.target_table CLONE db.source_table
```

Spark Joins

- Broadcast Hash Join / Nested Loop

  SELECT **/*+ BROADCAST(a) */** id
  FROM a JOIN b ON a.key = b.key

  Requires one side to be small. No shuffle, no sort, very fast.

- Shuffle Hash Join

  SELECT **/*+ SHUFFLE_HASH(a, b) */** id
  FROM a JOIN b ON a.key = b.key

  Needs to shuffle data but no sort.
  Can handle large tables, but will OOM if data is skewed.
  One side is smaller (3x or more) and a partition of it can fit in memory
  (enable by `spark.sql.join.preferSortMergeJoin = false`)

- Sort-Merge Join

  SELECT **/*+ MERGE(a, b) */** id
  FROM a JOIN b ON a.key = b.key

  Robust. Can handle any data size. Needs to shuffle and sort data, slower in most cases when the table size is small.

- Shuffle Nested Loop Join (Cartesian)

  SELECT **/*+ SHUFFLE_REPLICATE_NL(a, b) */** id
  FROM a JOIN b

  Does not require join keys as it is a cartesian product of the tables. Avoid doing this if you can

Vectorized UDFs aka Pandas UDF (more efficient)

```
from pyspark.sql.functions import pandas_udf, PandasUDFType

@pandas_udf('double', PandasUDFType.SCALAR)
def pandas_plus_one(v):
    return v + 1
```

```
from pyspark.sql.functions import col, rand

df = spark.range(0, 10 * 1000 * 1000)
display(df.withColumn('id_transformed', pandas_plus_one("id")))
```

UDFs allow for multiple column inputs.
Complex outputs can be designated with the use of a defined schema encapsulate in a `StructType()`

Not Spark specific
UDF - work on a row and produce a row
UDAF - User Defined Aggregate Functions - work on several rows and produce a row (ex. sum, count)
UDTF - User Defined transformation Fucions - many to many (ex. Explode)

On a efficiency scale:
Most efficient is built-in APIs, so use wherever possible
Next best is pandas UDF
Next is regular python UDF

CRUD

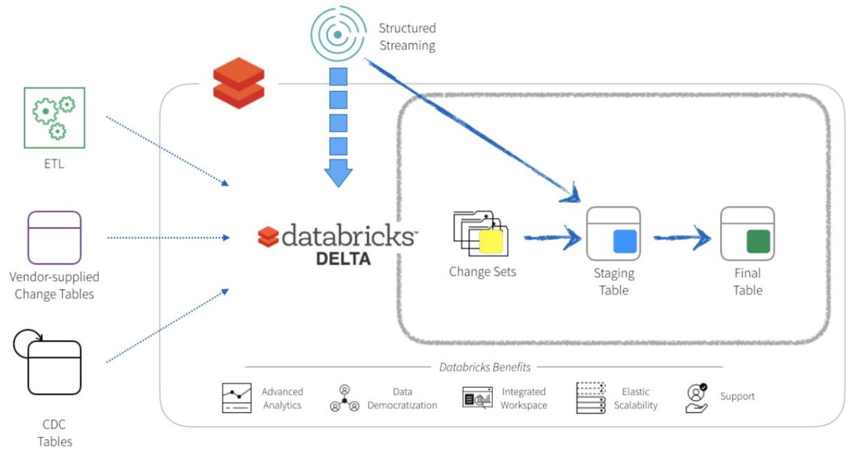
- Create
  - Append Vs Overwrite
- Read/Retrieve
  - Get
  - By partition, By Key, By Z-order field
- Update data & metadata
  - Replace partitions
  - Fine grained update
- Delete
  - Replace partitions
  - Fine grained delete
  - Block deletes and updates in a Delta table: delta.appendOnly=true
- Upsert
  - Update + Insert
  - Merge

```
INSERT INTO events SELECT * FROM newEvents
INSERT OVERWRITE TABLE events SELECT * FROM newEvents
```

```
df.write \
  .format("delta") \
  .mode("overwrite") \
  .option("replaceWhere", "date >= '2017-01-01' ) \
  .save("/mnt/delta/events")
```

```
df.option("mergeSchema", "true")
df.write.option("overwriteSchema", "true")
```

CDC - Change Data Capture



Simplify CDC in Delta [Part1]

Most Common Operation: append

Fine Grained Updates/deletes is difficult

Upserts/Merge

- Insert new data
- Update Old data

SCD (Slow Changing dimension)

- Maintaining history

CDC extracts from Source

SCD updates Target

Change Data Feed

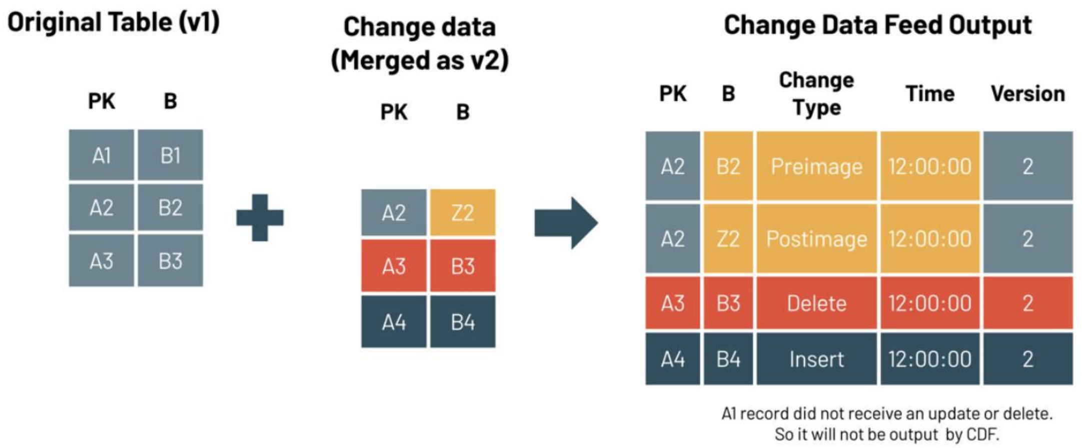
Simplify CDC in Delta [Part2]

[CDF]

- Represents row-level changes between versions of a Delta table. (_change_data folder)
- Includes the row data along with metadata indicating whether the specified row was inserted, deleted, or updated.
- `delta.enableChangeDataFeed = true`
- `SELECT * FROM table_changes('tableName', 0, 10)`

-
  ```
  spark.readStream.format("delta") \
  ```
  - `.option("readChangeFeed", "true") \`
  - `.option("startingVersion", 0) \`
  - `.table("myDeltaTable")`

✓	✗
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



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

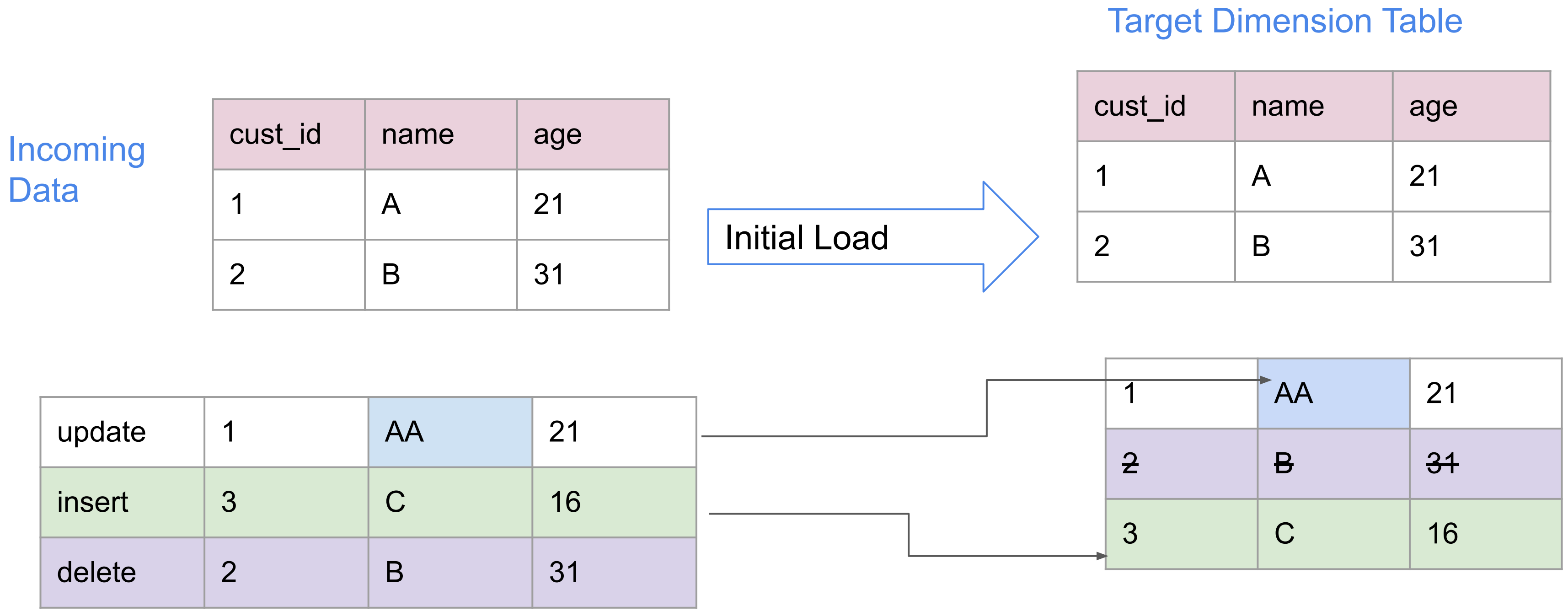
A1 record did not receive an update or delete.
So it will not be output by CDF.

SCD Types - handle changes to dimension tables

- Notebook
- OLAP : Facts + Dimensions
- Slowly Changing Dimensions
  - Change infrequently and without any pre-determined schedule
- Types
  - Type 0 : Fixed Dimension - No change allowed
    - Too rigid
  - **Type 1 - Overwriting the old value**
    - **Pro:** No additional memory/storage is required
    - **Con**: We cannot trace back to the history of modifications
  - **Type 2 - Creating a new additional record**
    - **Pro:** You can trace back to the history
    - **Con**: The memory/storage is getting consumed since keeping old records
  - Type 3 - Adding a new column
    - **Pro** of SCD 1
    - **Con:** Not sustainable over time ad some systems may have max limit on #colums
  - Type 4 - Using historical table
  - Type 6 - Combine approaches of types 1,2,3 (1+2+3=6)

SCD Type 1

- No need to store historical data in the Dimension table
- Overwrites old data
- Initial data -> load everything to target (Full Load)
- Changes to data - >
    - New data -> Insert
    - Existing data -> Update



Incoming Data

cust_id	name	age
1	A	21
2	B	31

Initial Load

Target Dimension Table

cust_id	name	age
1	A	21
2	B	31

update	1	AA	21
insert	3	C	16
delete	2	B	31

1	AA	21
~~2~~	~~B~~	~~31~~
3	C	16

SCD Type 2

- Stores historical data in the Dimension table => new records created for changes
- Changes to data - >
  - New data -> Insert
  - Existing data -> Add new record with surrogate key
  - 'effective date' and 'current indicator' columns are used
  - Only one row of that key with current indicator to true

Target Dimension Table

Incoming Data

cust_id	name	age
1	A	21
2	B	31

Initial Load **Date**: 09/29/2021

cust_id	name	age	s_date	e_date	current
1	A	21	09/29/2021	12/31/9999	T
2	B	31	09/29/2021	12/31/9999	T

Update	1	AA	21
Insert	3	C	16
Delete	2	B	31

Next Load **Date**: 10/2/2021

1	A	21	09/29/2021	10/1/2021	F
2	B	31	09/29/2021	10/1/2021	F
1	AA	21	10/2/2021	12/31/9999	T
3	C	16	10/2/2021	12/31/9999	T

Announcements

- Assignment 2 is out
- If you’ve not turned in Assignment 1, please do so