Contents

1 Data Lakehouse

Data management system that combines the benefits of `Data Lakes` and `Data Warehouses`.

- Provides scalable storage and processing capabilities
- Can establish a single source of truth, eliminate redundant costs, and ensure data freshness
- Often use data design pattern that incrementally improves, enriches, and refines data as it moves through layers of staging and transformation → Medaillon architecture

1.1 Data Ingestion

- Logical layer provides a place for (batch/streaming) data to land in its raw format
- Files are converted into `Delta Tables` and schema enforcement capabilities from `Delta Lake` can be used to check for missing or unexpected data
- Use `Unity Catalog` (`UC`) to register tables according to data governance model and require data isolation boundaries
- `UC` allows to track lineage and apply unified governance to keep sensitive data private and secure

2 Delta Lake

`Delta Lake` is an open source storage layer that supports `ACID` transactions, scalable metadata handling, and unification of streaming and batch data processing.

2.1 Key Features

- `ACID` transactions:
  - `A`tomicity: Entire transaction (either failing or succeeding) completes
  - `C`onsistency: Data follows rules or will be rolled back
  - `I`solation: One transaction completed before the start of another transaction
  - `D`urability: Data is saved in a persistent state once completed
- Problems solved by `ACID`:
  - Streamlined data append
  - Simplified data modification
  - Data integrity through job failures
  - Support for real-time operations
  - Efficient historical data version management
- DML Operations:
  - Modify data using multiple frameworks, services, and languages
  - `CRUD`-Operations: `INSERT, UPDATE, DELETE, MERGE`
- Data Skipping Index: Employs file statistics to optimize query performance by skipping unnecessary data scans
- Time Travel:
  - `Delta Tables` keep a transaction log for each version (writes) of the table
  - Historical querying possible by Delta transaction log
  - Snapshot Isolation
  - Useful for audits, regulatory compliance, and data recovery
- Schema evolution and enforcement:

  - Evolution: Automatically adjusts the schema of `Delta Table` as data changes (adding new columns, rename columns, etc.)
  - Enforcement: Ensures that any data written to the `Delta Table` matches the table's defined schema
- Scalable metadata:
  - Transaction log providing transactional consistency per `ACID` transaction
  - Efficient managing of metadata for large scale datasets without its operations impacting query or processing performance
- Audit history:
  - Detailed logs of all changes made to the data (who, what, when)
  - Crucial for compliance and regulatory requirements
- Incrementally improve quality of data
- Data Objects: `Metastore` → `Catalog` → `Schema` (Database) → `Table, View, Function`
- Liquid Clustering:
  - Data layout to support efficient query access and reduce data management and tuning overhead
  - Flexible and adaptive to data pattern changes, scaling, and data skew
  - Most consistent data skipping
- Predictive I/O:
  - Automates `OPTIMIZE` and `VACUUM` operations
  - Uses ML to determine most efficient access pattern to read data

2.2 Change Data Capture (CDC)

- Technique to capture and process the changes made to data in a source database and deliver these changes in real time to a target system
- Enables to keep `Data Lake` or `Warehouse` in sync with operational databases
- Supports real-time analytics and data science
- Useful for data synchronization, replication, auditing, and analytics
- `Delta Lake` supports `CDC` through `Change Data Feed` (`CDF`), which allows `Delta Tables` to track low-level changes between versions of a `Delta Table`
- Challenges: Handling slowly changing dimensions (`SCDs`), which are dimensions that store both current and historical data over time in a `Data Warehouse`
- Syntax: `APPLY CHANGES INTO`

2.3 Change Data Feed (CDF)

- Gives ability to track changes to a `DLT`
- `UPDATE, MERGE, DELETE` operations will be put into a new `_change_data` folder, while `APPEND` operations already have their own entries in the table history
- Through this tracking, the combined operations can be read as a feed of changes from a table to use downstream
- Ability to only have the rows that have changed between versions makes downstream consumption of `UPDATE, MERGE, DELETE` operations extremely efficient
- `CDF` captures changes only from a `Delta Table` and is only forward-looking once enabled → it will capture changes once the table property is set up and not earlier

3 Workflows

3.1 Compute

3.1.1 Job (Automated) Cluster

- Created and terminated automatically for each job run reducing usage costs
- Best for automated jobs or data pipelines, and operational use cases
- Only job clusters should be used in production
- Single-user

3.1.2 All-purpose (Interactive) Cluster

- Can be shared by multiple users
- Best for interactive tasks, streaming workloads, data exploration, development, ad-hoc and ongoing analysis
- Should not be used in production as they are not cost-efficient
- Manually managed

3.1.3 Other

- Fully managed services that are operationally simpler and more reliable
- `Serverless compute for notebooks`: On-demand, scalable compute used to execute `SQL` and `Python` in notebooks
- `Serverless compute for jobs`: On-demand, scalable compute used to run `Databricks` jobs without configuring and deploying infrastructure
- `Instance pools`: Compute with idle, ready-to-use instances, used to start and auto-scaling times
- `Serverless SQL warehouses`: On-demand elastic compute used to run `SQL` commands on data objects in the `SQL` editor or interactive notebooks
- `Classic SQL warehouses`: Used to run `SQL` commands on data objects in the `SQL` editor or interactive notebooks. `Photon` included, ad-hoc `SQL` and BI analytics

3.2 Cluster

- CPU should always be >80% utilization
- 8 → 16 cores per executor (depending on workload)
- `Shared cluster`:
  - Multiple users can work on a cluster in parallel
  - `Python, SQL, SCALA`
- `Single user cluster`:
  - Dedicated to a single user at creation time
  - Supports ML Runtimes, `Spark submit` jobs
  - Fine-grained access control
- `SQL Warehouse (Classic, Pro, and Serverless)`:
  - Multiple users can work on a shared cluster in parallel
  - `SQL` only

3.3 Jobs Features

- Job parameters: Passed into each task within a job, formatting and behavior of the parameter is dependent on the task type
- Job contexts: Special set of templated variables that provide introspective metadata on the job and task run
- Task values: Custom parameters that can be shared between tasks in a `Databricks` job

3.4 Late Jobs

- Allows customers to define a `soft timeout` after which they receive a warning that a job task run takes longer than expected and can also be timed out
- No alerts will be triggered by default, but the event will be logged

3.5 Run Job Task Type

- With jobs-as-a-task, a job can be broken into 2 (or more) jobs, chained together
- Jobs triggered by a `Run Job Task` use their own cluster configuration
- Separate larger jobs into multiple components, as well as creating generalized, modular, reusable jobs that can be parameterized by other jobs

3.6 Databricks Asset Bundle (DAB)

- Collection of `Databricks` artifacts (e.g., jobs, ML models, `DLT` pipelines, and clusters) and assets (e.g., `Python` files, notebooks, `SQL` queries, and dashboards)
- These bundles are represented in a configuration file and can be co-versioned
- Using CLI, these bundles can be materialized across multiple workspaces like `dev, staging` and `production` enabling to integrate these into `CI/CD` and automation processes

3.7 Best Practices

- Automated/`Job Clusters`:
  - Automated/`Job Clusters` for production workloads
  - Interactive/`All Purposes Clusters` are for development only
- Prefer Multi-Task jobs:
  - Each task could run on its own cluster
  - Tailored job clusters enable each task to run independently without resource sharing bottlenecks
- Cluster reuse: Enable running tasks in a `Databricks` Job on the same cluster for more efficient cluster utilization and decreased job latency
- Enable `Photon`: High-performance vectorized query engine
- Repair and re-run only the failed tasks to reduce time and resources
- Late jobs
- Serverless Workflows:
  - Fully managed service operationally simpler and more reliable
  - Faster clusters and auto-scaling capabilities
- Triggers: Use file arrival and `Delta Table` triggers
- Modular orchestration: Reuse jobs and break down huge jobs into multiple smaller / manageable tasks

- Run as `Service Principal`
- `CI/CD` with `DAB`

4 Delta Lake Table

Open-source `Data Lake` storage format providing features such as transactional capabilities, schema evolution, time travel and concurrency control to provide a robust, scalable, and efficient storage solution.

4.1 Anatomy

- Data files:
  - Store data in `Parquet` file format
  - Files are stored in a distributed cloud or on-premises file storage system such as `HDFS`, `AWS S3`, etc.
- Transaction (Delta) log:
  - Ordered record of every transaction performed on a `DLT`
  - Ensures `ACID` properties by recording all changes to a table in a series of `JSON` files
  - Each transaction is recorded as new `JSON` file in the `_delta_log` directory
- Metadata:
  - Includes information about the table's schema, partitioning, and configuration settings
  - Stored in the transaction log and can be retrieved using `SQL`, `Spark`, `Rust`, and `Python` APIs
  - Helps manage and optimize table by providing information for schema enforcement and evolution, partition strategies, and data skipping
- Schema evolution:
  - Defines the table's structure, including its columns, data types, etc.
  - Enforced on write, ensuring that all data written to the table adheres the defined structure
- Checkpoints:
  - Periodic snapshots of transaction log helping to speed up the recovery process
  - `Delta Lake` consolidates the state of the transaction log by default every 10 transactions
  - Stored as `Parquet` files
- `SQL`-Implementierung
- `Python`-Implementierung

5 Optimizing Delta Lake Tables

- `OPTIMIZE`:
  - Compact and sort data in the `Delta Table`
  - Merges smaller files into larger ones → reducing storage overhead and enhance query performance
- `ZORDER`:
  - Organizes data based on selected columns to improve query performance
  - Groups related data together physically, reducing the amount of data/files it takes to read or scan during queries
  - Data write optimization that requires considering query patterns and selected columns for optimal results
  - Identify columns that are commonly used in query predicates and have a high cardinality
  - `Z-Ordering` is more effective when the query predicates are selective and the data is skewed
- Partitioning:
  - Organize data files in a table or `Data Lake` based on specific column values
  - Optimizes query performance by reducing the amount of data that needs to be scanned during queries
  - Data of partitioned tables are physically stored in directories that correspond to the unique value of the partitioned column
- `VACUUM`:
  - Remove stale files that are no longer referenced by the table
  - Use `RETAIN` to specify number of hours to retain the history of a table
- `SQL`-Implementierung
- `Python`-Implementierung

5.1 Summary

- Performance impact: Both `OPTIMIZE` and `ZORDER` operations can be resource-intensive and time-consuming → may impact query performance on tables with heavy workloads
- Choose right column for `ZORDER`
- Data distribution: Partitions should not end up with larger files than others → skew
- Data evolution: `ZORDER` is not dynamic; it remains static after the data is written. If data distribution changes significantly over time, initial well-optimized `Z-Ordering` may become less effective
- Running `OPTIMIZE` twice on the same dataset, the second run will have no effect → `idempotent operation`

6 Delta Live Table

`Delta Live Table (DLT)` is a Data Engineering pipeline framework running on top of a `Delta Lake`. It is a materialized view for the `Lakehouse` as well as defined by a `SQL` query and created/kept up to date by a pipeline.

6.1 Key Features

- Allows users to write `SQL` queries by extending standard `SQL` syntax with unified declarative way of specifying data definition language (`DDL`) and data manipulation language (`DML`) operations
- Combines incremental ingestion, streamlined ETL, and automated data quality processes like `expectations`
- Rather than building out a processing pipeline piece by piece, the declarative framework allows to simply define tables and views with less syntax
- Manages compute resources, data quality monitoring, processing pipeline health, and optimizes task orchestration
- `ACID` transactions, schema enforcement, time travel, and unified batch and streaming processing
- Materialized view (`DLT`):
  - Stores results of a query and can be refreshed periodically or on demand to reflect latest state of the entire data
  - Only available in the pipeline, not persistent to the metastore
- Streaming table:
  - Processes streaming data incrementally and continuously updates the results of a query based on new data arriving
  - Only supports reading from `append`-only streaming sources and only reads input batch once
  - Can perform operations on the table outside the managed `DLT` pipeline
- Use `CREATE OR REFRESH STREAMING LIVE TABLE` to define a streaming table that processes each record exactly once
- Read a table with `STREAM()` to denote that we only want to process the incremental/new data and not the full table (`append`-only)
- Use `CREATE OR REFRESH LIVE TABLE` to define a materialized view that processes records as required
- General syntax for `DLT` query in `SQL`:

```
CREATE OR REFRESH [STREAMING] LIVE TABLE <table> AS
<select_statement>
```

6.1.1 Medaillon Architecture

Data design pattern that organizes data in a `Lakehouse` into bronze, silver and, gold layers.

- Bronze:
  - Contains raw data from multiple sources (raw ingestion and history)
  - Replaces traditional data lake
  - Provides efficient storage and querying of full, unprocessed history of data
- Silver:

  - Contains validated and conformed data (filtered, cleaned, augmented)
  - Reduces data storage complexity, latency, and redundancy
  - Optimizes ETL throughput and analytic query performance
  - Preserves grain of original data
  - Enforces production schema, data quality checks
- Gold:
  - Curated and enriched data for analytics and AI (business-level aggregates)
  - Powers ML applications, reporting, dashboards, ad-hoc analysis
  - Refined views of data, typically with aggregations
  - Reduces strain on production systems
  - Optimizes query performance for business-critical data

6.2 Data Quality and Validation Rules

- Define and enforce data quality rules on datasets using `expectations` → optional clauses that check the quality of each record in a query
- Specify actions, such as warning, dropping, failing, or quarantining the record
- `SQL`-Implementierung

6.3 Types of Exception Handling Actions and Syntax

- Warn (default):
  - Result: Invalid records are written to the target; the dataset reports the failure as a metric

```
# SQL
CONSTRAINT <expectation> EXPECT (<expectation_expr>);

# Python
@dlt.expect(<description>, <constraint>)
@dlt.expect_all(expectations)
```

- Drop:
  - Result: Invalid records are not written to the target; the dataset reports the failure as a metric

```
# SQL
CONSTRAINT <expectation> EXPECT (<expectation_expr>)
ON VIOLATION DROP ROW;

# Python
@dlt.expect_or_drop(<description>, <constraint>)
```

- Fail:
  - Result: Invalid records stop the update from completing. Fix issue before re-running the query

```
# SQL
CONSTRAINT expectation EXPECT (<expectation_expr>)
ON VIOLATION FAIL UPDATE;
```

```

# Python
@dlt.expect_or_fail(<description>, <constraint>)
```

7 Data Governance with Unity Catalog

7.1 Unity Catalog

- `UC` is a universal catalog for data and AI on a `Lakehouse`
- Helps to control, audit, trace, and discover data across `Databricks`
- Set up data access rules in one place and enforce them in all workspaces, following ANSI `SQL` standards
- Create and manage storage credentials, external locations, storage locations, and volumes using `SQL` or `UC` UI
- Access data from various cloud platforms and storage formats
- Apply fine-grained access control and data governance policies
- Unified governance for data and AI:
  - Assets within `UC` include `catalogs, databases (schemas), tables`, notebooks, workflows, queries, dashboards, filesystem volumes, ML models, etc.
  - Protect data and AI assets with `UC's` built-in governance and security with unified solution
- `SQL`-Implementierung

7.2 Create and Manage Catalogs, Schemas, Volumes and Tables using UC

7.2.1 Metastore

- Centralized metadata layer
- Provides ability to catalog and share data assets across the `Lakehouse`
- Convention: {`catalog`}.{`schema`}.{`table`} or {`catalog`}.{`schema`}.{`volume`} → organize data and asset hierarchically
- Hierarchy enables data applications to read and join across boundaries that traditionally required copying data between `Hive` tables

7.2.2 Catalog

- Object that groups data assets in the first level of structure
- Can include `schemas, tables` or `volumes`
- Can also specify a storage location that is used by default for its `schemas, tables` or `volumes`

7.2.3 Table

- Access tabular data stored in cloud object storage
- `Managed table`:
  - Backed by a managed storage location
  - Automatically deleted when the table is dropped
  - Only `DELTA` format

- `External table`:
  - Backed by an external location
  - Is not deleted when the table is dropped
  - Introduced by keyword `LOCATION` when created
  - Supports many file formats

7.2.4 Volume

- `UC` object representing a logical volume of storage in a cloud object storage location
- Provide capabilities for accessing, storing, governing, and organizing files (non-tabular, (semi-/un-) structured data)
- A `managed volume` is backed by a managed storage location and automatically deleted when the volume is dropped
- An `external volume` is backed by an external location but not deleted when the volume is dropped

7.2.5 View

- Read-only object that is the result of a query over one or more tables and views in a `UC` metastore
- Materialized views: Incrementally calculate and update results returned by the defining query
- Temporary views: Has limited scope and persistence and is not registered to a schema or catalog
- Dynamic views: Used to provide row- and column-level access control, in addition to data masking

7.3 Define Access Control Policies in UC

- Use `ANSI SQL` to grant permissions to existing `Data Lake`
- Fine-grained access control method controls who can access data based on multiple conditions or entitlements
- Allows to specify policies for each data item and attribute and apply them consistently across all workspaces and platforms
- Use `GRANT` and `REVOKE` statements to manage privileges on securable objects to principals
- Securable objects can be a `catalog, schema, table` or `view`
- Principal is a user, a service principal, or a group that can have privileges
- General Syntax:

```
GRANT <privilege> ON <securable_object> TO <principal>;
REVOKE <privilege> ON <securable_object> FROM <principal>;
```

  with `privilege`: e.g., `SELECT, ALL PRIVILEGES, APPLY TAG, CREATE SCHEMA, USE CATALOG`
- Inheritance model: lower-level objects inherit from higher-level objects
- `UC` object ownership:
  - Object owners get all privileges on their own objects by default and can give privileges to their own objects and any objects inside them
  - Schema owners do not automatically have all privileges over the tables in the schema, but can give themselves privileges over the tables in the schema

7.4 Data Lineage

- Describes the transformations and refinements of data from source to insight and includes the metadata and events associated with the data lifecycle
- Benefits:
  - Impact analysis: Users can see downstream consumers of a dataset
  - Data understanding: Users gain better context and thrustworthiness of data
  - Data provenance and governance: Users can access lineage data through `Catalog Explorer` or `REST API`
- Lineage include source and destination tables and columns, notebooks, workflows, and dashboards related to a query
- Audit logs include user, timestamp, action, and query details for each access
- Users need the `SELECT` privilege on a table to see its lineage

7.5 Accessing System Tables

- System tables are an analytical store hosted by `Databricks` containing account's operational data
- Audit logs include records for all audit events (user actions, cluster events, job runs, and notebook executions)
- Table and column lineage includes records for each read or write event on a `UC` table or path, as well as source and destination columns involved in the event
- Clusters include full history of cluster configurations over time
- Node types include currently available node types with basic hardware information

8 Performance Tuning in Delta Lake

8.1 Avoiding Small File Problem

- Too many small files greatly increase overhead for reads
- Too few large files reduce parallelism on reads
- `Databricks` automatically tunes the size of `DLT` and compacts small files on write with `auto-optimize`

8.2 Table Statistics

- Collects statistics on all columns in a table
- Helps `Adaptive Query Execution`:
  - `Spark` automatically breaks down larger partitions into smaller, similar sized partitions
  - Choose proper `join` type
  - Select correct build in a `hash-join`
  - Calibrating the `join` order in a multi-way `join`
  - Syntax: ANALYZE TABLE <table> COMPUTE STATISTICS FOR ALL COLUMNS

8.3 Predictive Optimization

- Refers to using predictive analytics techniques to automatically optimize and enhance performance of systems, processes, or workflow

- Involves data-driven insights to proactively identify and implement optimizations, improving efficiency, cost-effectiveness, and overall system performance
- Maintenance operations like `OPTIMIZE` to improve query performance by optimizing file size and `VACUUM` to reduce storage costs by deleting unused data

9 Performance Optimization with Spark

9.1 Broadcast Variables

- Feature allowing to send read-only data to all the executors in a cluster to be cached in memory and used for local operations → useful when working with a large data set used in multiple tasks
- Will not be sent over the network for each tasks
- Cache the data on each executor node rather than sending it with every task → reduces network traffic and improves performance
- Once the data is distributed, it is cached on each executor node in a serialized form
- When a task needs to access the data, it deserializes it and uses it in the computation
- Data remains cached until the broadcast variable is destroyed
- Broadcast variables are read-only and cannot be modified once created

9.1.1 Limitations and Best Practice

- Are not automatically garbage collected → destroy them using `destroy` method of `Broadcast` class when done
- Not checkpointed → If an executor node fails and restarts, it needs to fetch the data again from another node or from the driver
- Use them sparingly and carefully → too many variables can consume a lot of memory

9.2 Minimize Data Shuffling

- Data shuffling is the process of transferring data across different partitions or nodes
- Can be expensive and time-consuming as it involves network, disk, and (de-)serialization of data
- Shuffling occurs when performing a
  - `join` on two or more `DataFrames`
  - global aggregations
  - `repartition` or `coalesce` operations
- Reduce shuffling by using
  - a `broadcast join` or `coalesce join` instead of a `sort-merge join` or shuffled `hash join`
  - partial or approximate aggregation instead of exact aggregations
  - optimal partitioning → dividing data into smaller logical units processed in parallel by different executors or cores
  - `broadcast join`: Send a copy of a small table to each executor node in the cluster so that it can be cached in memory and used for local join operations with the larger table
- Reduce the number of partitions to avoid creating too many files or tasks

- Increase number of partitions to avoid creating too few large files or tasks that can cause data skew or memory issues
- Reduce network IO by using fewer, larger workers

9.3 Avoiding Data Skew

- Occurs when the data being processed is not evenly distributed (inconsistent file size) across partitions, resulting in some tasks taking much longer than others and wasting cluster resources
- Can be caused by operations that require shuffling or repartitioning the data (`join`, `groupBy` or `orderBy`)
- How to handle skewed data:
  - Isolate the skewed data from the rest of the data and process it separately → avoid shuffling the skewed data and reduce load on the cluster
  - Broadcast `hash join`: broadcast one of the `DataFrames` to each executor and build hash table in memory
  - Key salting: modifies the join key column by adding a random suffix (`salt`) to each value → create more partitions and distribute the data more evenly across them

9.4 Caching and Persistence

- Allow `Spark` to store some or all of the data in memory or on disk → can be reused without computing
- Store some intermediate results in memory or other more durable storage, e.g., disk space → avoid recomputing

9.5 Partitioning and Repartitioning

- Partitioning: Split data into multiple chunks that can be processed in parallel by different nodes in a cluster
- Repartitioning: Change the number or distribution of partitions in an existing dataset
- Partitioning drawbacks:
  - Increases metadata cost of managing data, as each partition adds an entry to the `Delta Lake` transaction log
  - It may introduce data skew or imbalance if some partitions are much larger or smaller than others
- Best practices:
  - Select column with high cardinality, low skew, and high selectivity
  - Avoid partitioning by a column with low cardinality, high skew, and low selectivity (column is rarely used for filtering or joining)
  - Use partitioning scheme that matches query patterns, e.g., if most of the queries filter by a single column, use a single-column partitioning scheme
  - Monitor and optimize partitioning scheme over time:
    * `ANALYZE TABLE` to collect statistics on partitions,
    * `VACUUM` to remove obsolete files from partitions,
    * `OPTIMIZE` to coalesce small files into larger ones
    * `ZORDER` to reorder data within partitions based on a column
  - Try to keep each partition less than 1TB and greater than 1GB

10 Spark Structured Streaming

10.1 Process

- When streaming, `Spark`
  - represents streaming computation as a series of transformations on an unbounded table
  - translates logical plan into a series of micro-batch jobs or continuous tasks running on the cluster
- Physical plan is then optimized → predicate pushdown, project pruning, and join reordering
- Depending on `source` type and `options`, `Spark` will append new data from the `source` and append them to an internal buffer → acts as an input table for streaming query
- Depending on `trigger` type and `options`, `Spark` then periodically processes new data from the buffer and updates an internal state store → keeps track of intermediate results such as aggregates, windows, and joins
- Depending on `sink` type and `options`, `Spark` periodically writes new data from the state store to the destination

10.2 Ingesting Streaming Data

- Use `spark.readStream` to create streaming `DataFrame` and specify `source type, options`, and `schema`
- Use `spark.writeStream` to write the output of the streaming `DataFrame` to the destination and specify `sink type, options`, and `trigger`
- `Source` types:
  - `file, socket, rate, memory, delta`
  - Syntax: `spark.readStream.format("<source>").load()`
- `Sink` types:
  - `console, file, memory, delta, foreach, foreachBatch` → write transformed output to external storage systems
  - Syntax: `spark.writeStream.format("<source>").start()`
- Output modes:
  - `append`: writes only new records to the destination, does not modify existing rows, e.g., no aggregations
  - `update`: writes only updated records to the destination, adds new rows based on values in the stream, e.g., aggregations with windows
  - `complete`: write all records to the destination, e.g., aggregations without windows
  - Syntax: `.outputMode("<mode>")`
- `Trigger` types:
  - Processing time `triggers`:
    * Executes a micro-batch at a regular interval based on the processing time
    * Syntax: `.trigger(processingTime="30 seconds").start()`
  - One-time `trigger`:
    * Executes a single micro-batch and then terminates the query
    * Syntax: `.trigger(once=True)`
  - Default `trigger`:
    * Executes a micro-batch as soon as the previous one finishes

        * Maximizes throughput of a streaming application by processing data as fast as possible
- Schema inference when using from_json or from_avro to perform schema validation and evolution
- Offset management (checkpoints):
    - Ensure that the query can resume from where it left off in case of failures or restarts
    - Save intermediate state of a query to a durable storage system that should be accessible from all nodes in the cluster (e.g., HDFS, S3, Azure Blob Storage)
    - Allow streaming query to be modified and still resume from where it left off → recovery semantics
    - Allow streaming query to report its progress asynchronously to an external system → asynchronous progress tracking
    - Syntax: `.option("checkpointLocation", "<path>/checkpoint")`
- Watermark:
    - Handles late data in streams and allows to specify threshold of how late the data can still be considered for processing
    - Trade-off between latency and completeness
    - Works well for streaming apps that have a bounded delay in the data source (e.g., sensors)
    - Does not work well for applications that have an unbounded delay (e.g., historical data, user data)
    - Syntax: `df.withWatermark("<column>", "<expression>")`

10.3 Reading from real-time Sources (Kafka)

- spark-sql-kafka-0-10 library provides source and sink for Kafka
- Source allows to read data from Kafka topics or partitions as a stream of records → each consists of a key, value, offset, partition, timestamp, and optional headers
- Sink allows to write data to Kafka topics as a stream of records → each consists of a key, value, and optional headers
- When creating a DataFrame from Kafka using readStream, specify options such as bootstrap servers, topic name/pattern, and the starting/ending offset:
    - Bootstrap server: Address of the Kafka brokers that the source connects to
    - Topic name/pattern: Determines which topics or partitions to subscribe to
    - Start/end offsets: Determine range of records to read from each partition, e.g., earliest, latest or none
- When writing output of the stream destination using writeStream, specify output mode, format, trigger, and interval → determines how output table is updated when new data arrives
- Format determines destination system, e.g., console, filesystem (PARQUET, CSV, JSON), database (jdbc), or Kafka
- Phyton implementation

11 Processing Streaming Data

- `Delta Lake` as `sink`:
  - Uses transaction log to keep track of all changes made to a table (ordered list of atomic, deterministic, and serializable commits that have occurred in the table)
  - Ensures that only one writer can commit at a time by using `optimistic concurrency control (OCC)`
- Idempotent stream: the same data can be written to a `Delta Table` multiple times without changing the final result → useful for exactly-once processing data

12 SQL Coding

12.1 Delta Table

12.1.1 Creating a Delta Table or View

```sql
CREATE TABLE <schema>.<table> (
   id LONG,
   country STRING,
   capital STRING
) USING DELTA;

# Load data - INSERT INTO
INSERT INTO <schema>.<table> VALUES
   (1, "UK", "London"),
   (2, "Canada", "Toronto");


# Alternatively
INSERT INTO <schema>.<table_name>
SELECT * FROM <file_format>. `<source_table>.<file_format>`;

# CTAS - Combines creation and insertion into single operation
CREATE TABLE <schema>.<new_table>
USING DELTA
AS SELECT * FROM <schema>.<table>;

# Alternative
CREATE TABLE <table> USING DELTA
AS SELECT *
FROM read_files(
   "<file_path>",
   format => "<format>",
   sep => "<sep>"
)

# Temp View
CREATE OR REPLACE TEMP VIEW <view> AS
SELECT *
FROM <format>.`<file_path>`;
```

12.1.2 Reading a Delta Table

```sql
SELECT * FROM <file_format>. `<file_path>`
LIMIT 10;

# Alternativ
SELECT * FROM <schema>.<table>;
```

```

# Time Travel
SELECT DISTINCT <column> from <schema>.<table>
VERSION AS OF 1;

# Update data
UPDATE default.countries
SET {country = "U.K"}
WHERE id = 1;

# COPY INTO
COPY INTO <table>
FROM "<file_path>"
FILEFORMAT <file_format>
COPY_OPTIONS("..." = "...")

# Delete
DELETE FROM <schema>.<table>
WHERE id=1;
```

12.1.3 Optimizing Delta Tables

```
# Compaction
OPTIMIZE "file_path"

# Z-Ordering
OPTIMIZE "<file_path>"
ZORDER BY (<column>)


# Partitioning
```

12.1.4 Tagging, Commenting, and Capturing Metadata

```
# Add a comment on a table
COMMENT ON TABLE <catalog>.<schema>.<table>
IS <comment>;

# Add a tag to a table
ALTER TABLE <catalog>.<schema>.<table>
SET TAGS (
    "<tag1>"="<description1>",
    "<tag2>"="<description2>"
);

# Remove a tag from a table
```

```
ALTER TABLE <catalog>.<schema>.<table>
UNSET TAGS ("<tag1>", "<tag2>");

# Add comments to table columns
ALTER TABLE <catalog>.<schema>.<table>
ALTER COLUMN <column> COMMENT "<comment>";

# Add tags to column
ALTER TABLE <catalog>.<schema>.<table>
ALTER COLUMN <column> SET TAGS (
   "<tag>"="<description>"
)

# Remove tag from column
ALTER TABLE <catalog>.<schema>.<table>
ALTER COLUMN <col> UNSET TAGS ("<tag>");
```

12.2 Delta Live Table

12.2.1 Data Quality and Validation Rules

```
# Create Live Table for customer data
CREATE OR REFRESH LIVE TABLE customers (
CONSTRAINT
    valid_customer_key
    EXPECT (c_custkey IS NOT NULL)
    ON VIOLATION DROP ROW,
CONSTRAINT
    <expectation_name>
    EXPECT (<EXPECTATION_EXPRESSION>)
    ON VALIDATION <ACTION>
) AS SELECT * FROM <catalog>.<schema>.customers;

# Deduplicate records
CREATE TEMPORARY LIVE TABLE duplicate_customers_test (
    CONSTRAINT unique_customer_key
    EXCPECT (cnt=1)
    ON VIOLATION DROP ROW
) AS
SELECT
    c_custkey, count(*) AS cnt
FROM
    live.customers
GROUP ALL;
```

12.3 Unity Catalog

12.3.1 Create Catalog, Volume, View

```
# Create a catalog
CREATE CATALOG <catalog>;

# Create a schema
USE CATALOG <catalog>;
CREATE SCHEMA <schema>;

# Create a volume
CREATE VOLUME <catalog>.<schema>.<volume>
LOCATION "<path>";

# Create a table
USE CATALOG <catalog>;
USE SCHEMA <schema>;
CREATE TABLE IF NOT EXISTS <table> (
   <col_1> <TYPE>,
   <col_2> <TYPE>
);
INSERT INTO <table>
VALUES (<value>, <value>);

# Create a view
CREATE OR REPLACE VIEW <table> (<col_1>, <col_2>)
COMMENT "<comment>"
AS SELECT <col_1>, ...
FROM <table>;
```

12.3.2 Row Filtering

```
# Row filtering - create function
CREATE FUNCTION <function> (<column_name> <column_type>, ...)
RETURN {<filter_clause_whose_input_must_be_a_boolean>};

# Apply filtering
ALTER TABLE <table> SET ROW FILTER <function_nyme> ON
(<column_name>, ...);

# Modify row filters
CREATE OR REPLACE FUNCTION
<function> (<column_name> <column_type>);

# Delete row filter
ALTER TABLE <table> DROP ROW FILTER;
DROP FUNCTION <function>;
```

Perform `ALTER TABLE <table> DROP ROW FILTER` before dropping the function or the table will be in an inaccessible state.

12.3.3 Column Masking

Apply masking functions (`UDFs`) to table columns, such as replacing, hashing, or redacting original values in order to control visibility of sensitive data.

```
# Create function
CREATE FUNCTION <function> (<column_name>, <column_type>, ...)
RETURN {<expression_with_same_type_as_first_column};

# Apply column mask
ALTER TABLE <table> ALTER COLUMN <column>
SET MASK <function>;

# Or apply when creating the table
CREATE TABLE <table> (
   <column> <column_type> MASK <function>
);

# Remove column mask from column
ALTER TABLE <table> ALTER COLUMN <column_where_mask_is_applied>
DROP MASK;
DROP FUNCTION <function>;
```

13 Python Coding

13.1 Delta Table

13.1.1 Creating a Delta Table

```
delta_table = (DeltaTable.create(spark)
    .tableName("<schema>.<table>")
    .addColumn("id", dataType=LongType(), nullable=False)
    .addColumn("country", dataType=StringType(), nullable=False)
    .addColumn("capital", dataType=StringType(), nullable=False)
    .execute()
)

# Load data - INSERT INTO
data = [
    (1, "UK", "London"),
    (2, "Canada", "Toronto")
]
schema = ["id", "country", "capital"]
df = spark.createDataFrame(data, schema=schema)
(df.write
    .format("delta")
    .insertInto("<schema>.<table>")
)

# Alternatively
# Read data into a DataFrame and write it to Delta Table
df = (spark.read
    .format("<file_format>")
    .option("header", "true")
    .load("file_path")
)
(df.write
    .format("delta")
    .mode("overwrite")
    .saveAsTable("<schema>.<table>")
)

# Append data
data = [(3, "United␣States", "Washington␣D.C.")]
schema = ["id", "country", "capital"]
df = spark.createDataFrame(data, schema=schema)
(df.write
    .format("delta")
    .mode("append")
    .saveAsTable("<schema>.<table>")
)
```

13.1.2 Reading a Delta Table

```
df = spark.read.format("<file_format>").load("<file_path>")

# Alternativ
from delta.tables import DeltaTable

delta_table = DeltaTable.forName(spark, "<schema>.<table>")
delta_table.toDF().show()

# Time travel
(spark.read
    .option("versionAsOf", "1")
    .load("<table>.<file_format>")
    .select("<column>").distinct()
)

# Anzahl Rows zu bestimmtem Zeitpunkt
(spark.read
    .option("timestampAsOf", "YYYY-MM-DD")
    .load("<table>.<file_format>")
    .count()
)

# Restore Table
delta_table = DeltaTable.forPath(spark, "<file_path>")
delta_table.restoreToVersion(3)

# Update table
delta_table_df.update(
    condition = "id=1",
    set = {"country": "'U.K.'"}
)

# Delete from table
delta_table.delete(F.col("id") == 1)
```

13.1.3 Optimizing Delta Tables

```python
# Compaction
delta_table = DeltaTable.forPath(spark, "<file_path>")
delta_table.optimize().executeCompaction()

# Z-Ordering
delta_table = DeltaTable.forPath(spark, "<file_path>")
delta_table.optimize().executeZOrderBy("<column>")

# Partitioning
df = (spark.read
    .format("<file_format>")
    .option(...)
    .load("<file_path>")
)

(df.write
    .format("delta")
    .mode("overwrite")
    .partitionBy("<column>")
    .save("<file_path>")
)
```

13.2 Delta Live Table

13.3 Streaming

```python
# Create streaming DataFrame
df = (spark.readStream
    .format(...)
    .option(...)
    .load()
)

# Write output
query = (df.writeStream
    .outputMode(...)
    .format(...)
    .start()
)
```