# Program #1

|  |
| --- |
| (spark.readStream  .table("bronze")  .filter("topic = 'orders'")  .select(F.from\_json(F.col("value").cast("string"), schema).alias("v"))  .select("v.\*")  .withWatermark("order\_timestamp", "30 seconds")  .dropDuplicates(["order\_id", "order\_timestamp"])) |

1. In Spark Structured Streaming, **dropDuplicates** with a watermark only removes duplicates that arrive within the defined event-time threshold, for example, within 30 seconds of order\_timestamp.
2. However, any records arriving later than that threshold are considered “too late” and are not deduplicated by Spark’s in-memory state.
3. To ensure complete deduplication (including very late-arriving data), the foreachBatch sink can use an idempotent write pattern with Delta Lake’s MERGE operation.
4. The MERGE INTO statement compares each micro-batch of incoming data (microbatch) with the target Delta table (orders\_silver) based on unique keys (order\_id and order\_timestamp).
5. It only inserts rows that do not already exist in the target table, preventing duplicates even across micro-batches or late arrivals.
6. This combination of in-stream deduplication (for near-real-time performance) and MERGE-based deduplication (for completeness and correctness) provides an end-to-end reliable way to handle duplicates in streaming pipelines.

# Program #2

It adds a column showing the cumulative average score of each student from their first exam up to and including the current exam.

|  |
| --- |
| from pyspark.sql.window import Window  from pyspark.sql.functions import avg, col    window\_spec = Window.partitionBy("student\_id").orderBy("exam\_date")\  .rowsBetween(Window.unboundedPreceding, Window.currentRow)    df\_new = df\_student\_results.withColumn("avg\_score", avg("score").over(window\_spec)) |

**Overall explanation**

The PySpark code uses a **Window function** to calculate a cumulative or running average score for each student.

1. **Window.partitionBy("student\_id")**:

This divides the data into partitions (groups) based on the **student\_id**. The average calculation will be performed independently within each student's set of results.

1. **.orderBy("exam\_date")**:

This sorts the rows *within* each student's partition by the **exam\_date** (oldest to newest). This is crucial for a running calculation.

1. **.rowsBetween(Window.unboundedPreceding, Window.currentRow)**:

This defines the *frame* for the window.

* **Window.unboundedPreceding** means the frame starts at the very first row in the current student's partition (the first exam).
* **Window.currentRow** means the frame ends at the current row being processed (the current exam).
* This combination ensures that for any given row, the calculation includes all preceding rows *and* the current row, effectively defining a **cumulative** set of data.

1. **avg("score").over(window\_spec)**:

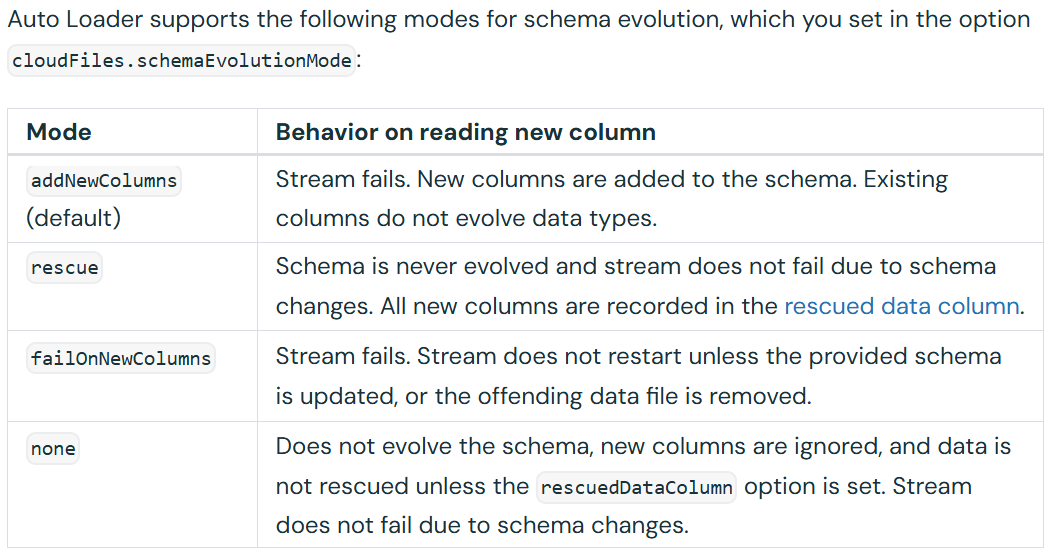
The **avg("score")** function is applied over the defined window\_spec. Because the window is partitioned by student\_id and is cumulative over exam\_date, the result in the new **avg\_score** column is the **cumulative average score** for that specific student up to that specific exam date.

# Program #3

A data engineer is configuring the following **Databricks Auto Loader stream** to ingest JSON data from an S3 bucket:



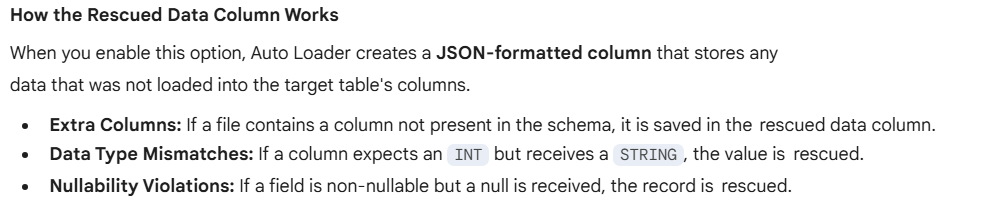
How does Auto Loader schema evolution work?



**Important Note on “rescue”**

* If you choose the **rescue** mode, Auto Loader will automatically create a hidden column named **\_rescued\_data** (by default) that stores any data that was dropped or didn't fit the schema.
* This is excellent for ensuring zero data loss during ingestion.
* You can rename the column or include it in cases where you provide a schema by setting the option **rescuedDataColumn**







Renaming the “rescue” data column name



# Program #4



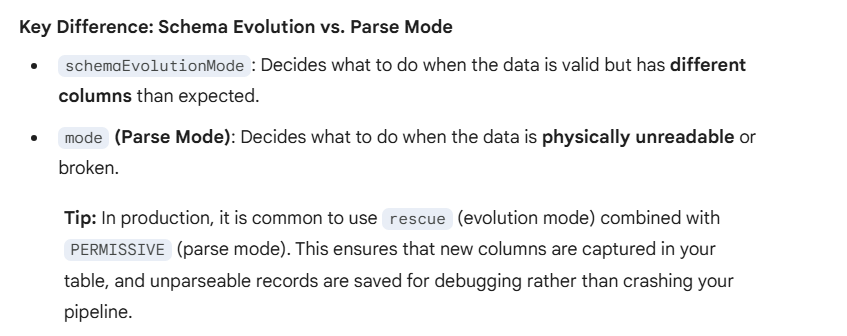
The JSON and CSV parsers support 3 modes when parsing records: PERMISSIVE, DROPMALFORMED, and FAILFAST.

* When used together with rescuedDataColumn, data type mismatches do not cause records to be dropped in DROPMALFORMED mode or throw an error in FAILFAST mode.
* Only corrupt records are dropped or throw errors, such as incomplete or malformed JSON or CSV. If you use badRecordsPath when parsing JSON or CSV, data type mismatches are not considered as bad records when using the rescuedDataColumn.
* Only incomplete and malformed JSON or CSV records are stored in badRecordsPath

**How these modes handle "Bad" data:**

When Spark encounters a record that is not valid JSON (e.g., a missing brace or a string where a number should be), these modes dictate the outcome:

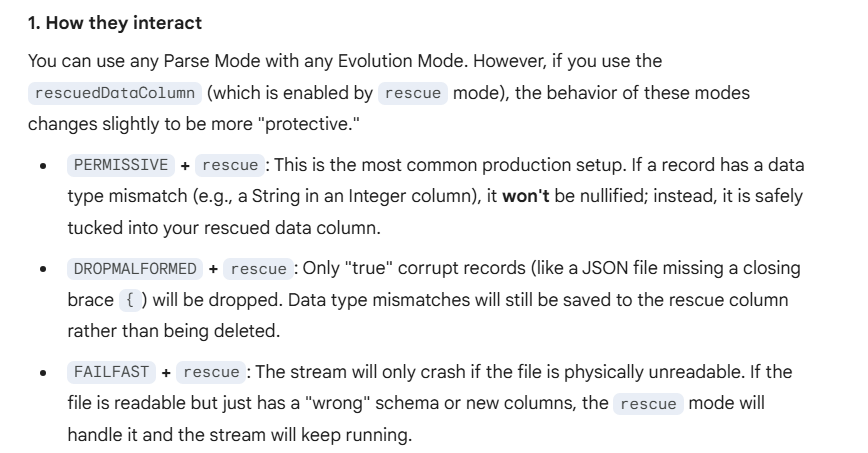
| **Mode** | **Result of a Corrupt Record** |
| --- | --- |
| **PERMISSIVE** | **(Default)** Sets all fields to null and puts the corrupt record in a column named \_corrupt\_record. |
| **DROPMALFORMED** | Simply ignores the bad record and continues processing the rest of the file. |
| **FAILFAST** | Throws an exception immediately and stops the stream so no bad data is processed. |

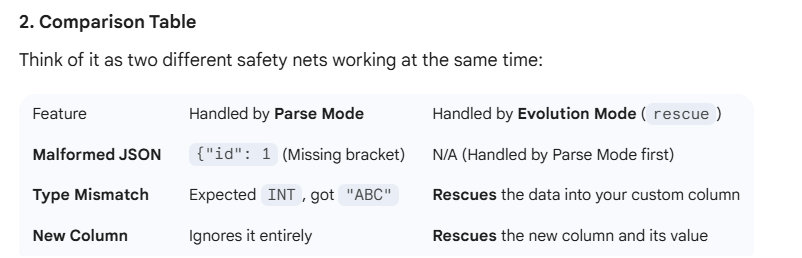


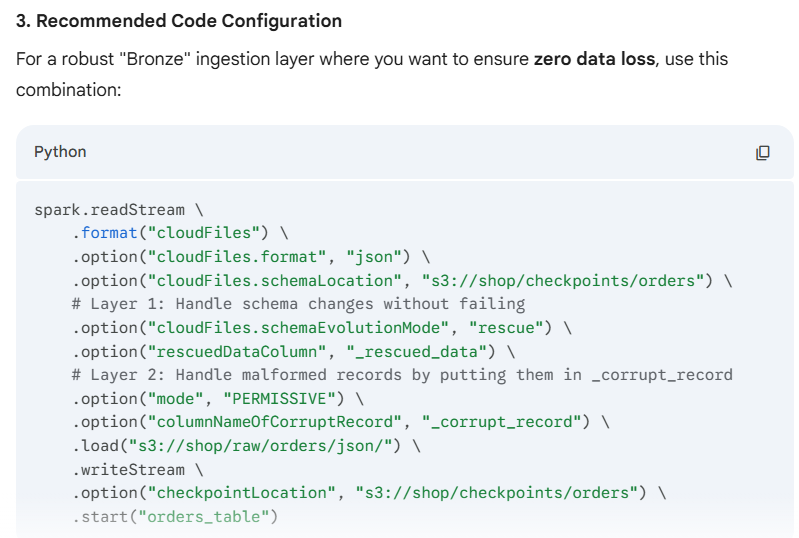
**Note:**

**Parse modes** are **NOT** restricted to the **rescue mode**. They work independently to handle different types of "bad" data.

To understand how they work together, it helps to distinguish between **Corrupt Records** (broken files) and **Unexpected Schema** (new columns).







# Program #5

DLT Expectations



Why is **LIVE** keyword used with SQL?

The difference comes down to how Python and SQL handle **namespace resolution** and the **dependency graph** within Databricks Delta Live Tables (DLT).

**The "Live" Keyword in SQL**

In DLT SQL, the live keyword is a virtual schema. It tells the DLT engine: *"Don't look for this table in the permanent Hive Metastore or Unity Catalog yet; look for it within the context of this specific pipeline."*

* **Dependency Tracking:** It allows DLT to understand that Table B depends on Table A.
* **Environment Agnostic:** By using live, you don't have to hardcode schema names (like dev\_db or prod\_db). DLT replaces live with the target schema defined in your pipeline settings at runtime.

**Why Python doesn't need "Live"**

In your Python code, you used **dlt.read("raw\_product\_source")**. Python handles this differently:

1. **Function Call vs. String:** **dlt.read()** is a specific library function designed to look within the DLT internal manifest. The string you pass to it is automatically treated as a reference to another object in the pipeline.
2. **Explicit Context:** Because you are using the dlt module, the "live" context is already implied. The library knows that any string passed to **read()** or **read\_stream()** should be resolved against the pipeline's datasets first.

**Comparison Table**

| **Feature** | **SQL Syntax** | **Python Syntax** |
| --- | --- | --- |
| **Reference Type** | Prefix required (live.table\_name) | Function call (dlt.read("table\_name")) |
| **Resolution** | Resolved by the SQL compiler at graph-build time. | Resolved by the dlt library at runtime. |
| **External Sources** | Uses standard identifiers (e.g., source\_db.table). | Often uses spark.read or dlt.read for managed sources. |

**Summary**

* In SQL, without the live. prefix, the engine would try to find **raw\_product\_source** in your default **database/catalog** immediately.
* Since that table might not even be built yet (or exists only within the pipeline), the code would fail.
* In Python, the **dlt** module acts as the "middleman" that handles this logic for you.

Reference data output of DLT pipeline

To reference data from **outside** the DLT pipeline—meaning tables that already exist in your Unity Catalog or Hive Metastore—you stop using the "internal" references and use the full path of the data source.

**1. Referencing External Tables in SQL**

In SQL, you simply omit the live. prefix and use the three-level namespace (catalog.schema.table).

CREATE OR REFRESH LIVE TABLE internal\_refined\_table

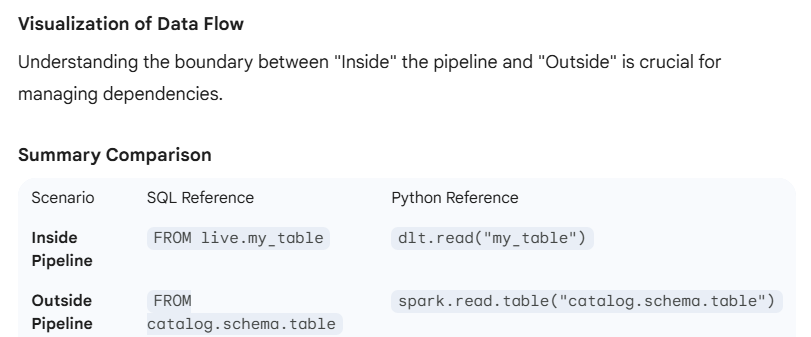
AS SELECT \* FROM production\_catalog.raw\_data\_schema.external\_source\_table;

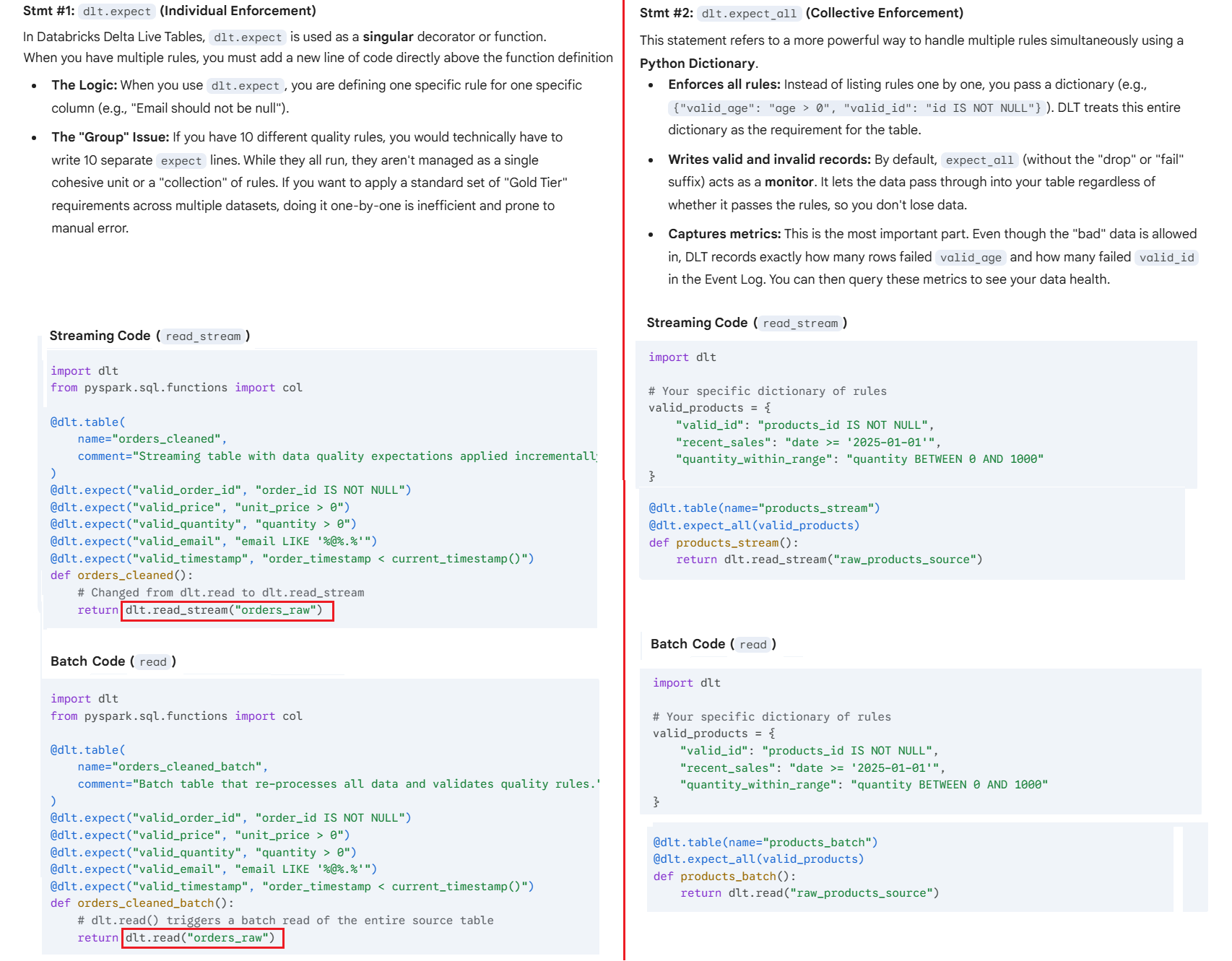
* **Why?** Without live., DLT treats the reference as a standard Spark SQL identifier and looks for it in the global Metastore.

**2. Referencing External Tables in Python**

In Python, you have two main ways to grab external data. You typically move away from **dlt.read()** for these:

****



****

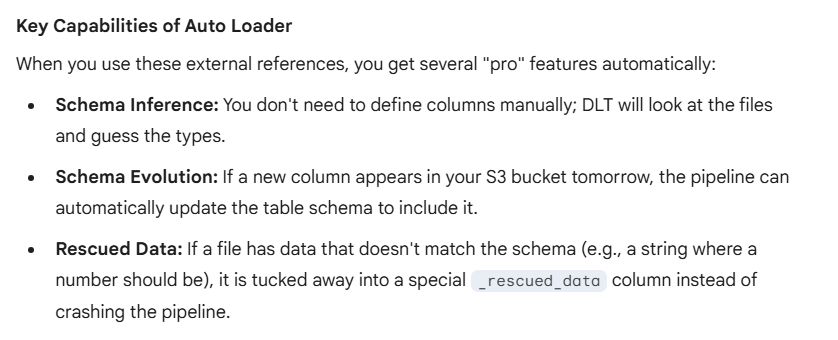
# Program #6

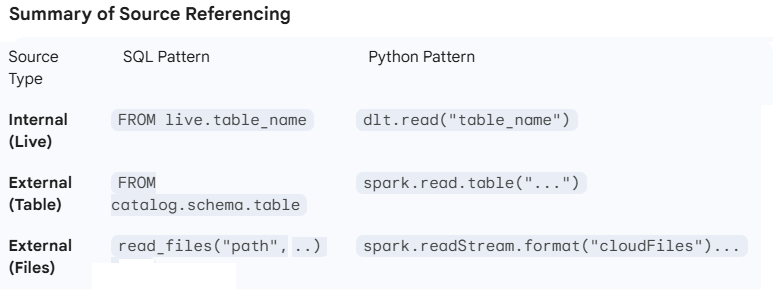
How to use **Auto Loader** (cloudFiles) to ingest raw files from cloud storage into your DLT pipeline

In Databricks, the preferred way to ingest files from cloud storage (S3, ADLS, GCS) into Delta Live Tables is using **Auto Loader**.

Auto Loader uses the cloudFiles format to automatically detect new files as they arrive, meaning you don't have to manage complex state or manual file lists.





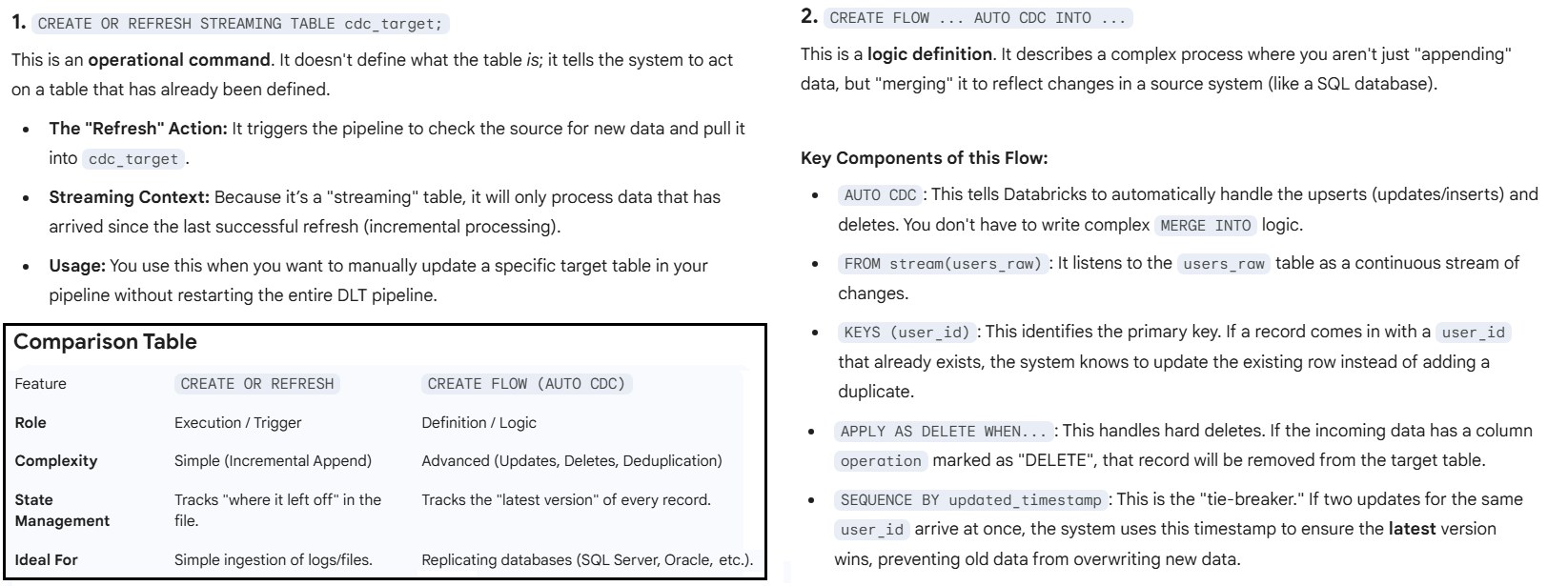


# Program #7

**Implementing Change Data Capture (CDC) using “Auto CDC” vs “Merge Statement”**

**Auto CDC**

* These two statements i.e. 1) CREATE OR REFRESH STREAMING TABLE users\_target & 2) CREATE FLOW ... AUTO CDC INTO ... represent two different ways to handle data movement in Databricks (specifically within Delta Live Tables).
* One is a high-level command to update a specific table, while the other is a sophisticated declaration for handling Change Data Capture (CDC).



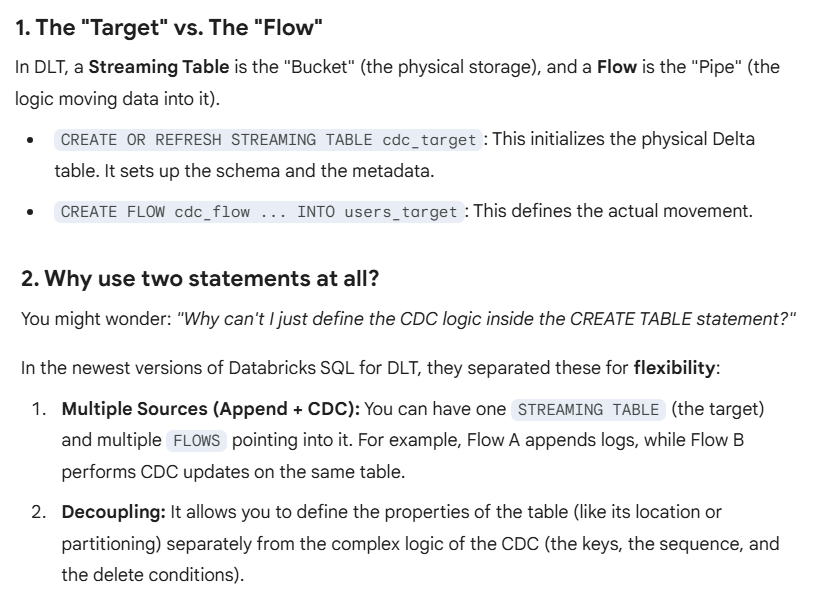
Main reason why **CREATE FLOW ... AUTO CDC** is used?

Because it shifts the burden of **data synchronization** from you to the engine.

Standard streaming tables are great at *appending* new logs, but they are terrible at *updating* existing records. Here is why AUTO CDC is the right tool for this job:

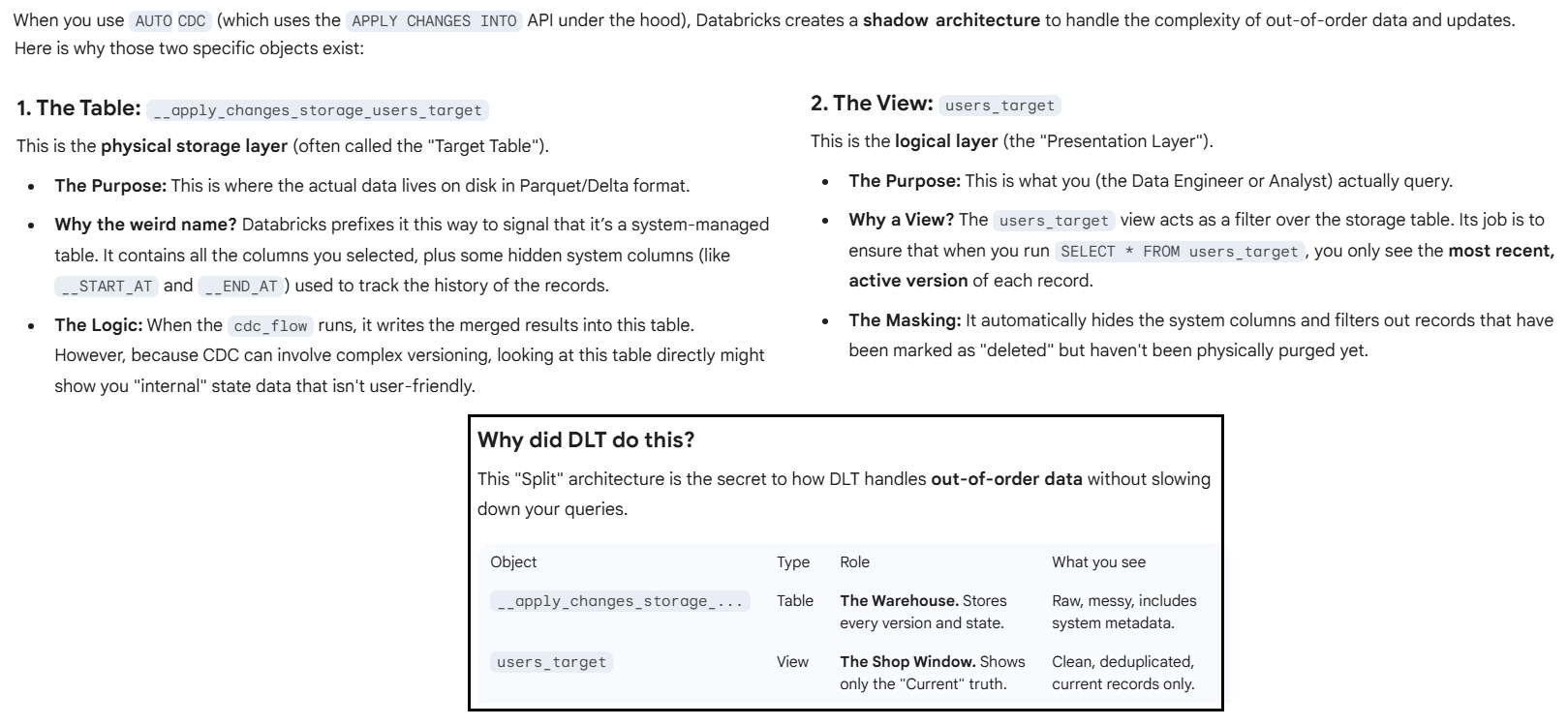


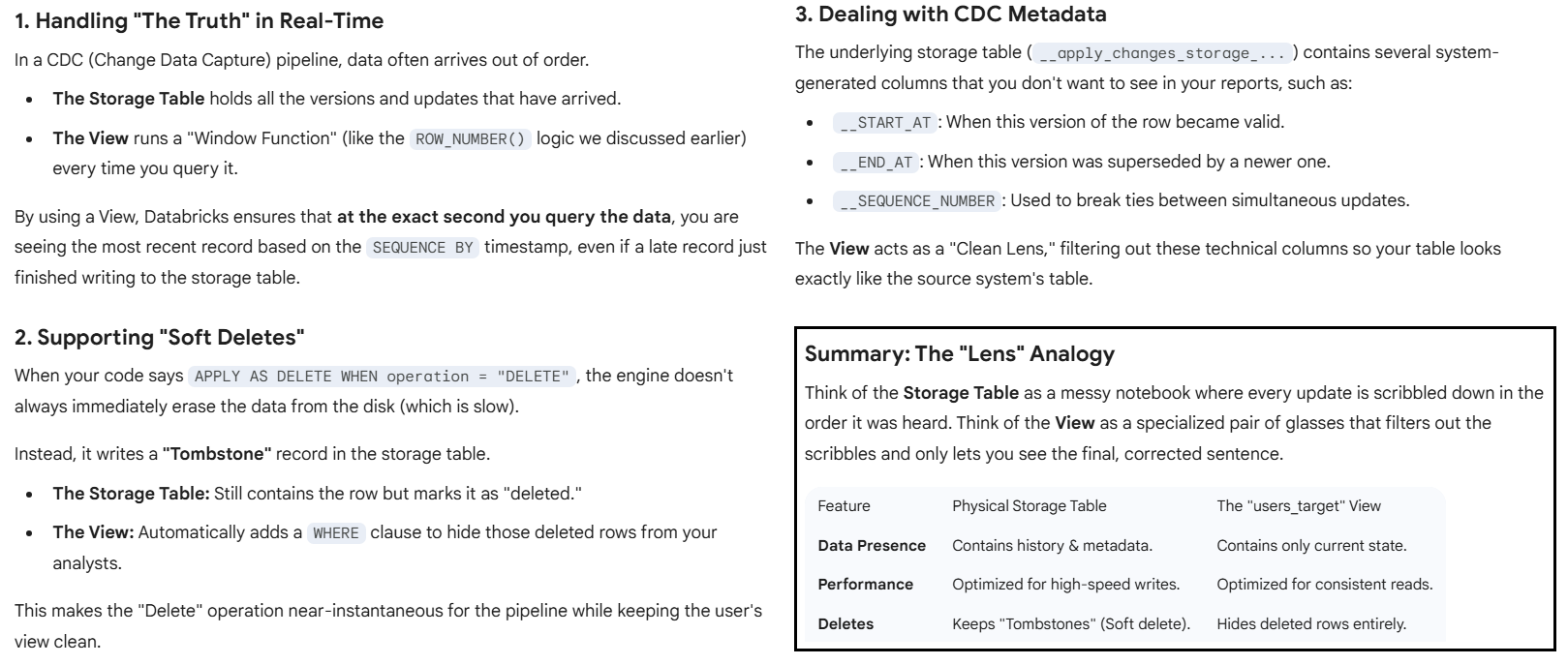
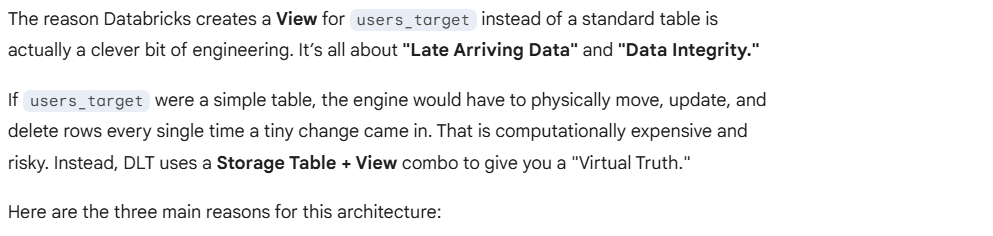
There is a deeper architectural reason why above 2 specific statements exist together. Let’s break down the relationship.



After running this code, the data engineer noticed that two objects were created the metastore in addition to the users\_target table:

* A view named **users\_target**.
* A table named **\_\_apply\_changes\_storage\_users\_target**.

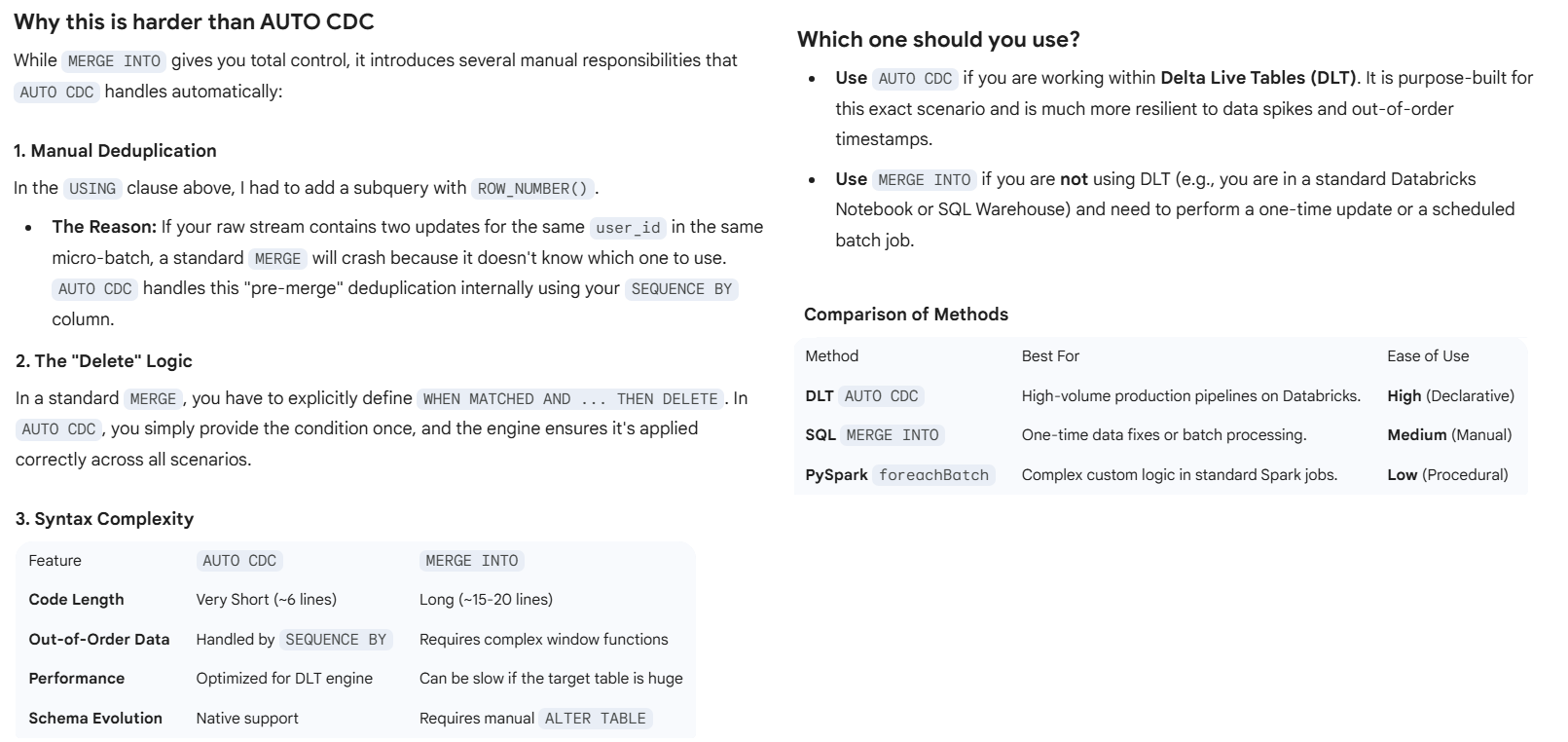




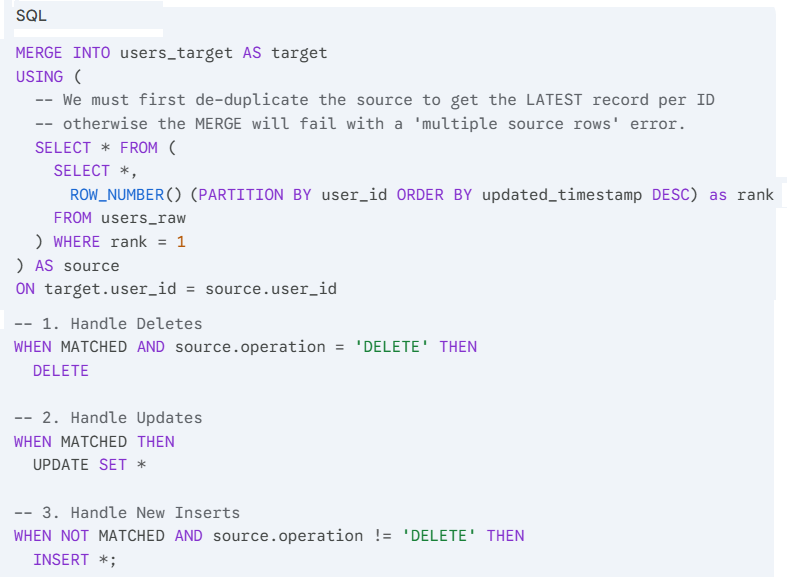


To replace the AUTO CDC flow with a MERGE INTO statement, you have to transition from a declarative style (where the system handles the logic) to a procedural style (where you define exactly how to join and update).

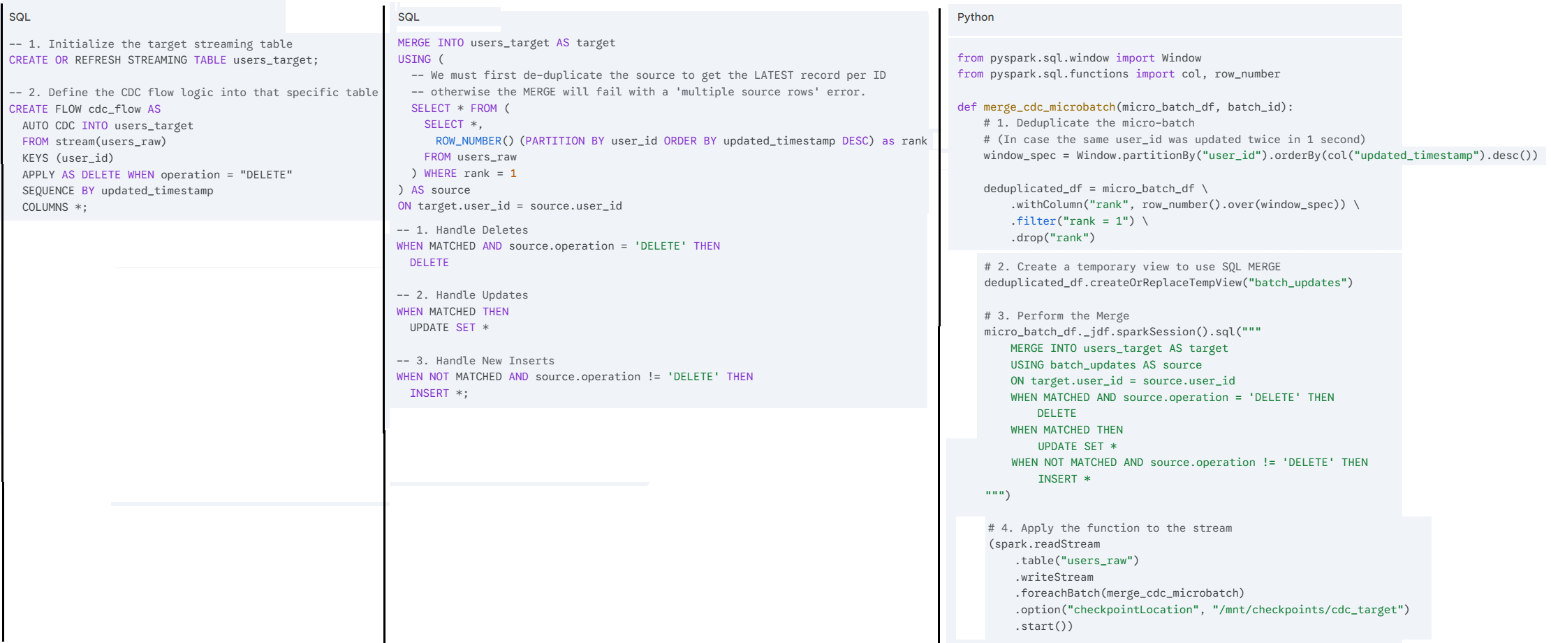
In a streaming context, MERGE INTO is typically used within a **foreachBatch** function in Python/Scala, but if you are using standard Databricks SQL, the equivalent logic looks like this:



**Merge Statement**





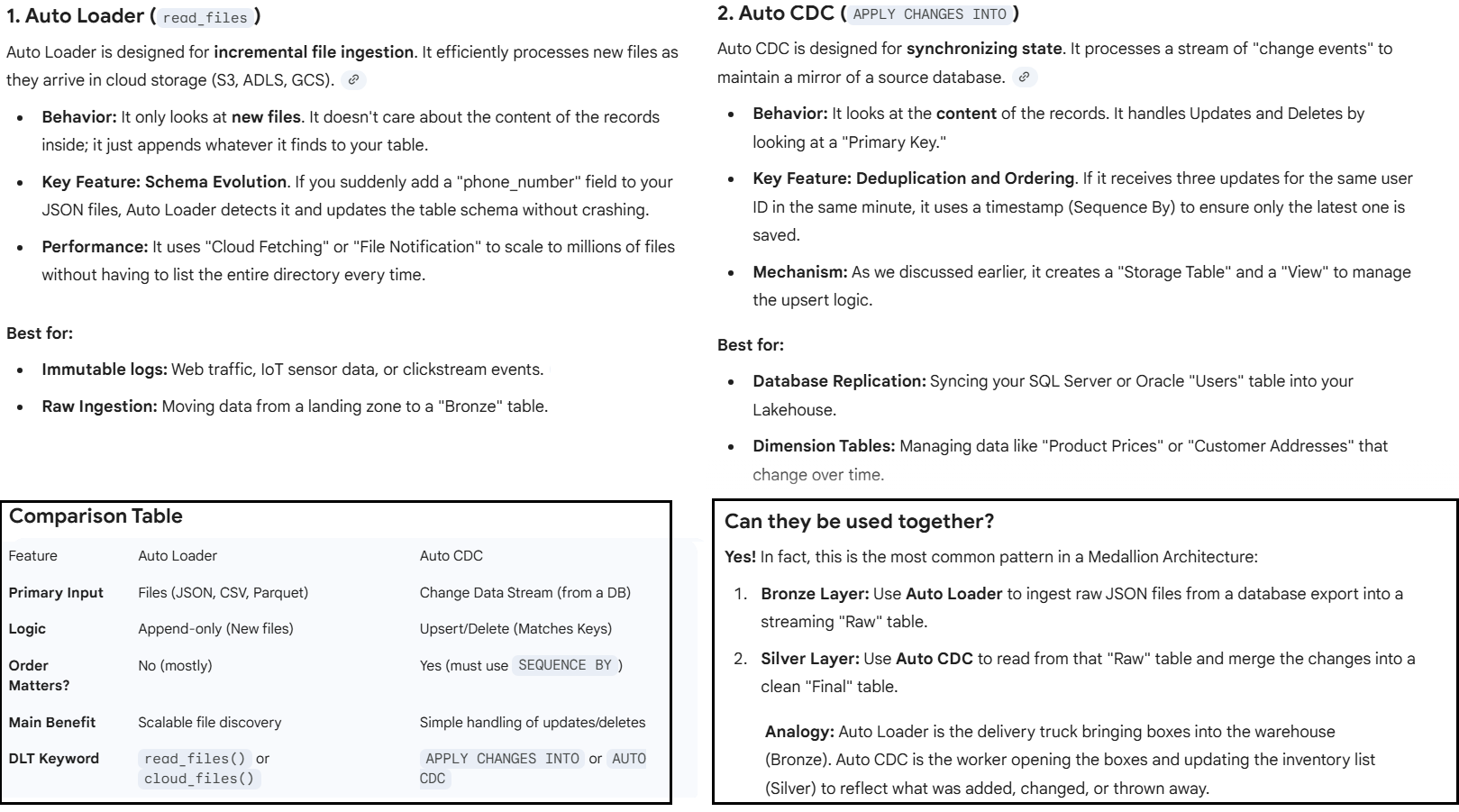


# Program #8

**Difference between Auto Loader and Auto CDC**

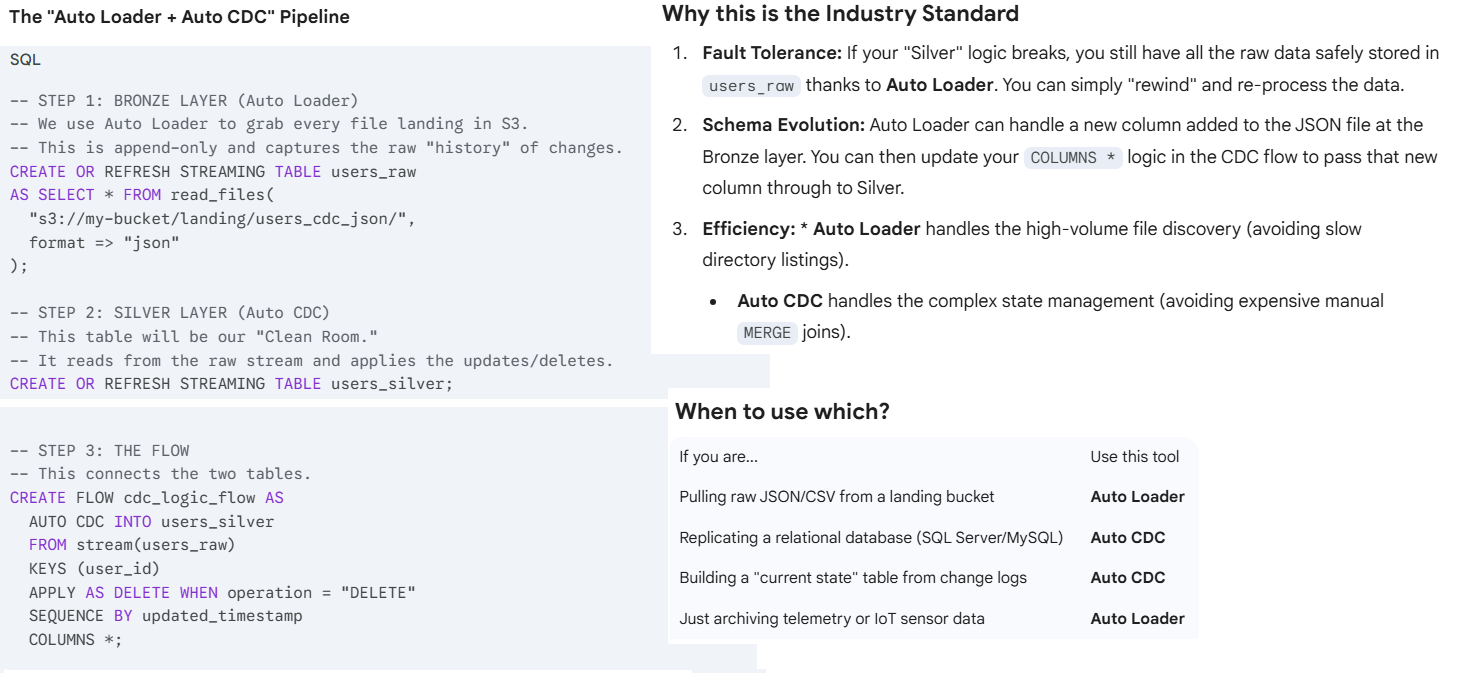
While both **Auto Loader** and **Auto CDC** are designed to automate data ingestion in Databricks, they solve two fundamentally different problems.

Think of **Auto Loader** as a "Mover" (moving new files) and **Auto CDC** as a "Sync" (keeping a target table identical to a source).



Here is how you combine both tools into a single, automated pipeline.

In this architecture, **Auto Loader** acts as the gatekeeper (Bronze) and **Auto CDC** acts as the brain (Silver).



--------------------------------------------------------------------------------------------------------------------  
**Rought Notes**

**Q.1:**

In the below statement what is Idempotent, explain with example

"Z-ordering is not idempotent but aims to be an incremental operation"

**Answer:**

In computing and mathematics, **idempotency** describes an operation that can be applied multiple times without changing the result beyond the initial application.

If an operation is idempotent, doing it once has the same effect as doing it five, ten, or a hundred times.

**Understanding the Terms**

In the context of your statement regarding **Z-ordering** (a technique used in databases like Delta Lake to co-locate related information):

* **Idempotent:** If you run the process on the same data twice, the second run should do nothing because the data is already "perfect."
* **Incremental:** The process is smart enough to only look at *new* or *changed* data rather than rewriting everything from scratch every time.

**The Example: Light Switches vs. Elevators**

To make this crystal clear, let's look at two everyday actions:

| **Action** | **Type** | **Explanation** |
| --- | --- | --- |
| **Pressing an Elevator Button** | **Idempotent** | If you press the "Level 5" button once, the elevator goes to 5. If you frantically press it 20 times, the result is the same: you end up on Level 5. |
| **Toggling a Light Switch** | **NOT Idempotent** | If you flip the switch once, the light turns on. If you flip it again, the state changes (it turns off). Each action changes the outcome. |

**Why Z-ordering is "Not Idempotent"**

When the statement says Z-ordering is **not idempotent**, it means that if you run the Z-order command on the same table twice, the system might actually perform work both times.

Because Z-ordering involves shuffling data into specific "curves" or patterns to optimize layouts, the underlying engine often treats each request as a fresh instruction to reorganize files. Even if the data is already sorted, the operation might re-cluster or create new file versions.

**Why it aims to be "Incremental"**

Since Z-ordering is a "heavy" (expensive) operation, you don't want to reorganize 10 years of data just because you added 5 minutes of new data today.

* **The Goal:** The system tries to identify only the new data files that haven't been Z-ordered yet and organizes those, leaving the previously optimized data alone.

**Q.2:**

Explain the below statement with the help of an example.

"Z-ordering aims to produce evenly-balanced data files with respect to the number of tuples, but not necessarily data size on disk. The two measures are most often correlated, but there can be situations when that is not the case, leading to skew in optimize task times.

For example, if you ZORDER BY date and your most recent records are all much wider (for example longer arrays or string values) than the ones in the past, it is expected that the OPTIMIZE job's task durations will be skewed, as well as the resulting file sizes. This is, however, only a problem for the OPTIMIZE command itself; it should not have any negative impact on subsequent queries. "

**Answer:**

This statement highlights a common "trap" in big data optimization: the difference between **count** (how many rows) and **volume** (how much space those rows take).

**The Core Concept: Rows vs. Weight**

Z-ordering (specifically the OPTIMIZE command in systems like Delta Lake) tries to be "fair" by giving every new file roughly the same number of **tuples** (rows). However, it doesn't always check how "fat" those rows are.

* **Tuples:** The number of individual records.
* **Data Size:** The actual bytes (MB/GB) those records occupy on the disk.

**The Example: The "Shipping Box" Analogy**

Imagine you have a packing robot (the OPTIMIZE job) programmed to put exactly **100 items** into every box.

1. **Monday's Data:** 100 feathers. The box is light and the robot packs it in 2 seconds.
2. **Tuesday's Data:** 100 bowling balls. The box is incredibly heavy, and the robot struggles for 2 minutes to lift and move it.

In this scenario, the robot met its goal (100 items per box), but the **task time** was "skewed"—Tuesday's task took much longer than Monday's.

**Why the "Skew" Happens**

In the technical example provided:

* **Old Records:** Might contain simple data (e.g., UserID, Date, Total). These are "thin" rows.
* **New Records:** Might include a new feature, like a Comments field with long paragraphs or a JSON\_Log array. These are "wide" rows.

When you run OPTIMIZE, the system might decide that every file should have 1 million rows.

* **Task A** processes 1 million "thin" rows $\rightarrow$ Fast, creates a 50MB file.
* **Task B** processes 1 million "wide" rows $\rightarrow$ Slow, creates a 500MB file.

This imbalance is called **Task Skew**.

**Why it doesn't hurt Queries**

The statement ends by saying this isn't a problem for your final users. Here is why:

1. **Pruning:** Z-ordering's main job is to help the engine skip files. If a query looks for Date = '2023-01-01', it will still find exactly which file that date is in, regardless of whether that file is 50MB or 500MB.
2. **Balanced Rows:** Since the row counts are even, the engine still processes a predictable number of records once it opens the file.

**Summary Table**

| **Metric** | **Status during Z-order** | **Resulting Impact** |
| --- | --- | --- |
| **Number of Tuples** | Balanced | Consistent row-processing for queries. |
| **File Size (Disk)** | Skewed | Some OPTIMIZE tasks take much longer than others. |
| **Query Performance** | Unaffected | Data skipping still works perfectly. |

**Q.3:**

Explain the below enclosed statement with the help of an example

"Z-ordering is not idempotent but aims to be an incremental operation.

The time it takes for Z-ordering is not guaranteed to reduce over multiple runs.

However, if no new data was added to a partition that was just Z-ordered, another Z-ordering of that partition will not have any effect."

**Answer:**

This statement clarifies that while Z-ordering isn't "perfectly efficient" by design (it doesn't automatically get faster just because you've done it before), it is "smart" enough to avoid redundant work.

To understand this, we need to look at how Z-ordering handles **files** rather than just raw rows.

**1. "Not Idempotent"**

**The Concept:** As established, an idempotent operation results in the same state every time. Z-ordering is technically **not** idempotent because if you force it to run again on the same data, it may rewrite files, generate new file IDs, and update the transaction log, even if the "shape" of the data doesn't change much.

**The Example:** Imagine you organize a bookshelf by genre. If you "Z-order" it again tomorrow, the engine doesn't just say "this is already done." It might pull all the books off the shelf and put them back in the exact same spots. You ended up with the same result, but **work was performed.**

**2. "Aims to be Incremental"**

**The Concept:** Instead of re-sorting your entire 5-year history every day, Z-ordering looks for **newly arrived files** that are still in a "random" state and only organizes those.

**The Example:**

* **Day 1:** You have 1,000 new records. You Z-order them.
* **Day 2:** You add 100 *new* records.
* **The Action:** The engine doesn't re-sort the 1,000 records from Day 1. It only focuses on the 100 new records, blending them into the optimized structure.

**3. "Time is not guaranteed to reduce"**

**The Concept:** You might think the second time you run Z-order it will be faster. That is a myth.

**The Example:**

If your data volume doubles, your Z-order time will likely double (or more). Because Z-ordering involves complex sorting (mapping multidimensional data to a 1D space), the computational "math" required doesn't get easier just because the data was partially sorted before.

**4. "No new data... will not have any effect"**

**The Concept:** This is the "safety valve." If the system detects that the **state of the partition** hasn't changed (no new files, no deletes, no updates), it will skip the operation entirely.

**The Example:**

* **10:00 AM:** You run OPTIMIZE table ZORDER BY (city). The system spends 5 minutes shuffling files.
* **10:05 AM:** You accidentally run the exact same command again.
* **The Result:** The system checks the transaction log, sees that no data has changed since 10:00 AM, and finishes in **0 seconds** with a message saying "Nothing to optimize."

**Summary of the Logic**

| **Scenario** | **Does it run?** | **Why?** |
| --- | --- | --- |
| **First Run** | Yes | Data is unorganized. |
| **New Data Added** | Yes | It runs **incrementally** on the new data. |
| **No Data Added** | **No** | It detects the state is unchanged and skips. |
| **Forced Re-run** | Yes | It is **not idempotent**; it will perform the work again if forced. |

**Shallow Clone vs Deep Clone**

**Q.1:**

With Shallow Clone, you create a copy of a table by just copying the Delta transaction logs. That means that there is no data moving during Shallow Cloning

Running the VACUUM command on the source table may purge data files referenced in the

transaction log of the clone.  Explain with example where running VACUUM command my purge data file?

**Answer:**

To understand why VACUUM can break a Shallow Clone, you have to think of the relationship between the two tables as a **pointer** and a **target**.

**The Concept: Shallow Clone vs. Deep Clone**

* **Deep Clone:** Copies the logs **and** the actual data files. It is an independent "twin."
* **Shallow Clone:** Copies **only** the logs (the "receipts"). The clone doesn't have its own data; it simply looks at the source table's files to answer queries.

**The Example: The "Ghost File" Scenario**

Imagine you have a source table called Sales\_Main and you create a Sales\_Clone using a Shallow Clone.

**Step 1: The Initial State**

The Sales\_Main table has a file called part-001.parquet. The Shallow Clone's transaction log points directly to this file on the disk.

**Step 2: An Update Happens**

You update a row in Sales\_Main. In Delta Lake, data isn't edited in place. Instead:

1. A new file, part-002.parquet, is created with the new data.
2. part-001.parquet is marked as **"tombstoned"** (deleted in the logs, but still physically on the disk).

**Step 3: The Shallow Clone's Perspective**

If you query Sales\_Clone, it is still looking for the version of the data from "Step 1." It still needs part-001.parquet to function.

**Step 4: The VACUUM Command**

You run VACUUM Sales\_Main RETAIN 0 HOURS.

The system looks at Sales\_Main, sees that part-001.parquet is no longer needed by the *Main* table, and **permanently deletes the file from the disk.**

**The Result:**

When you try to query Sales\_Clone, it throws a **"File Not Found" error**. The clone is now broken because the source table "cleaned up" the files the clone was secretly relying on.

**Key Takeaway Table**

| **Feature** | **Source Table (Sales\_Main)** | **Shallow Clone (Sales\_Clone)** |
| --- | --- | --- |
| **Transaction Log** | Owns its own log. | Has a copy of the log. |
| **Data Files** | Physically owns the files. | **References** the source's files. |
| **After VACUUM** | Clean and optimized. | **Broken** (missing data). |

**Important Rule**

Never run VACUUM with a short retention period on a source table if you have active Shallow Clones that need to see older versions of that data.

**How Deep Clone addresses this issue**

To help you prevent data loss and choose the right tool for your project, here is how you can implement these commands and a safer strategy for managing clones.

**1. How to run a Deep Clone**

If you want a copy that is **completely independent** and safe from the source table's VACUUM commands, use a Deep Clone. This physically moves the data.

-- This creates a full, independent copy

CREATE TABLE sales\_backup

DEEP CLONE sales\_main;

**2. How to run a Shallow Clone**

Use this only for quick testing or short-lived development branches where you know the source table won't be cleaned up immediately.

-- This creates a pointer-based copy (very fast)

CREATE TABLE sales\_dev\_test

SHALLOW CLONE sales\_main;

**Comparison of Clone Types**

| **Feature** | **Shallow Clone** | **Deep Clone** |
| --- | --- | --- |
| **Speed** | Near-instant | Slower (copies data) |
| **Storage Cost** | Minimal (logs only) | Full cost (duplicate data) |
| **Data Safety** | **Risky** (depends on source) | **Safe** (fully independent) |
| **Use Case** | Short-term testing | Backups / Disaster Recovery |

**3. Setting a "Safety Net" for Vacuum**

If you must use Shallow Clones, you should increase the retentionPeriod of your VACUUM command. By default, Delta Lake prevents you from vacuuming anything younger than 168 hours (7 days) to protect your clones and time-travel capabilities.

**To vacuum with a specific safety window:**

SQL

-- Deletes files older than 14 days, keeping your clones safe for 2 weeks

VACUUM sales\_main RETAIN 336 HOURS;

**Warning:** If you try to set RETAIN 0 HOURS, you usually have to manually disable a safety check (spark.databricks.delta.retentionDurationCheck.enabled = false). **Do not do this** if you have Shallow Clones active!

**How to fix a broken Shallow Clone?**

Unfortunately, once VACUUM has deleted the physical files from the source, a Shallow Clone **cannot be repaired**. You would need to:

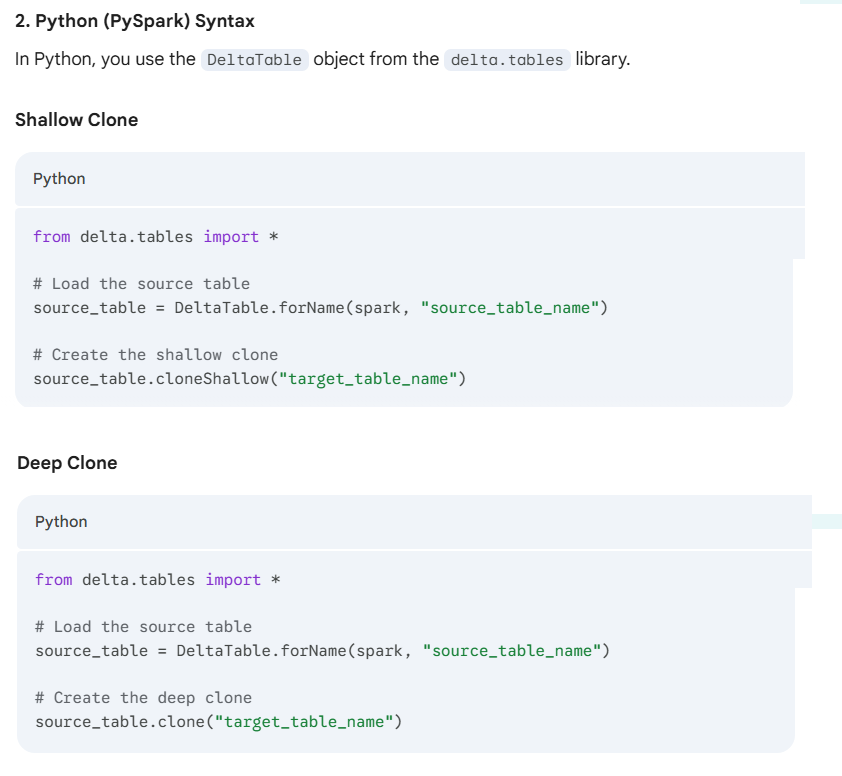
1. Drop the broken clone table.
2. Create a new clone from the current state of the source table.

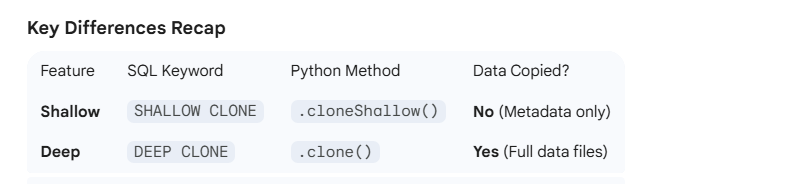
**Here is the syntax for creating both Shallow and Deep Clones in SQL and Python.**

1. **SQL Syntax**

****

1. Python Syntax





**Pro-Tip for Python**

If you are cloning to a specific file path rather than a table name in a catalog, you can pass the path directly:

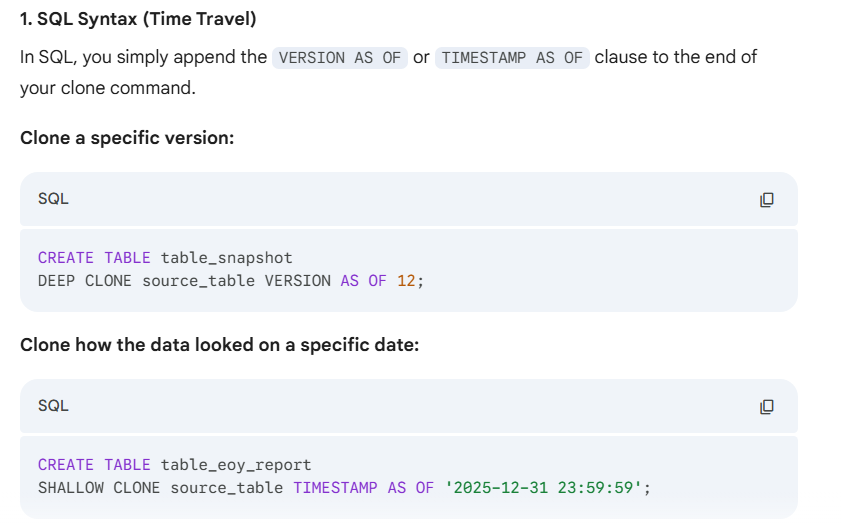
Python

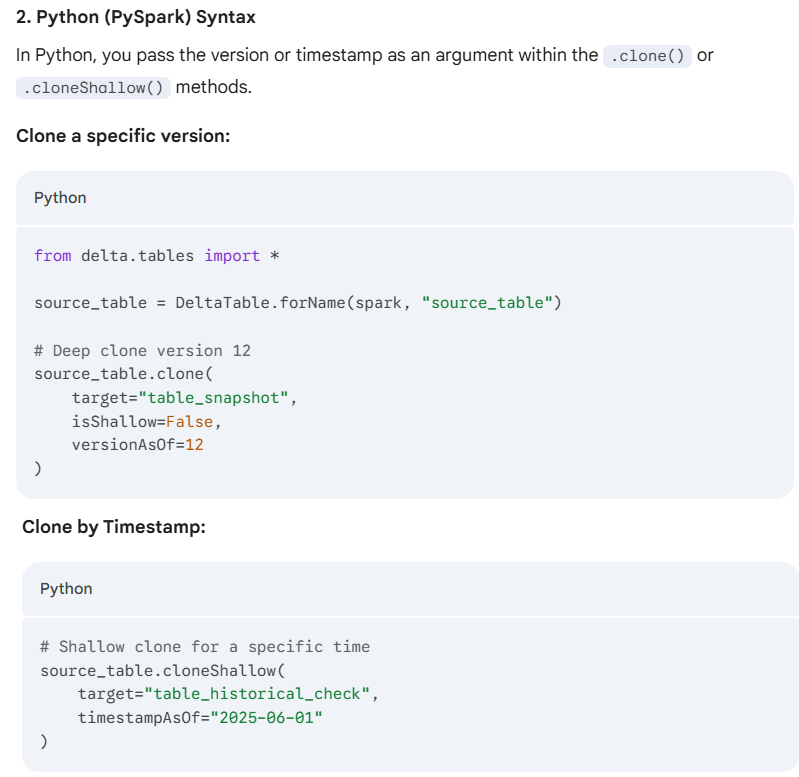
source\_table.clone("/mnt/delta/independent\_backup") # Deep Clone to path

**How to use Time Travel with cloning (e.g., cloning a table as it looked exactly 24 hours ago)**

Cloning a table from a specific point in history is one of the most powerful features of Delta Lake. It allows you to "freeze" a version of your data for auditing or debugging without stopping your main pipeline.

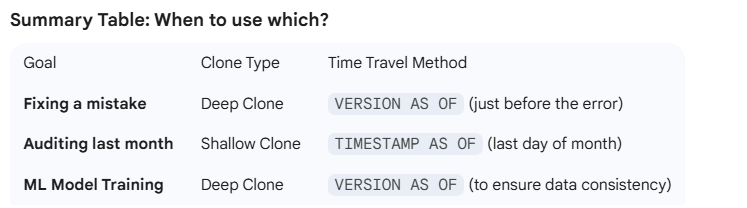
You can clone based on a **Version Number** or a **Timestamp**.





**Important Considerations for Time Travel Clones**

* **Retention Period:** You can only clone as far back as your logRetentionDuration (default 30 days) and before the last VACUUM. If you have already vacuumed the data from 2024, you cannot clone it.
* **Shallow Clone Risk:** If you shallow clone a version from 5 days ago, and then run VACUUM on the source table today, your clone will break immediately because those 5-day-old files will be purged.
* **Metadata Only:** When you clone a specific version, you are also cloning the **schema** (columns/types) exactly as they existed at that moment.



# Program #9

A data engineer has been tasked with writing streaming ingestion script to ensure only .png files are processed.

|  |
| --- |
| df = spark.readStream.format("cloudFiles") \  .option("cloudFiles.format", "binaryFile") \  .option("pathGlobFilter", "\*.png") \  .load("s3://shop/raw/invoices/") |

# Program #10

A data engineer wants to ingest input JSON data into a target Delta table. They want the data ingestion to happen incrementally in near real-time.

|  |
| --- |
| spark.readStream             .format("cloudFiles")             .option ("cloudFiles.format", "json")             .load(source\_path)  .writeStream             .option("checkpointLocation", checkpointPath)             .start("target\_table") |

# Program #11

A data engineer has the following streaming query with a blank. For handling late-arriving data, they want to maintain the streaming state information for 30 minutes.

**Note:** The **pyspark.sql.DataFrame.withWatermark** function allows you to only track state information for a window of time in which we expect records could be delayed.

|  |
| --- |
| spark.readStream  .table("orders\_cleaned")  .withWatermark("order\_timestamp", "30 minutes")  .groupBy(  "order\_timestamp",  "author")  .agg(  count("order\_id").alias("orders\_count"),  avg("quantity").alias("avg\_quantity"))  .writeStream  .option("checkpointLocation", "dbfs:/path/checkpoint")  .table("orders\_stats") |

# Program #12

Given the following query on the Delta table ‘**customers**’ on which **Change Data Feed** is enabled. Newly updated records will be appended to the target table.

The query uses **spark.readStream** to read the table's changes captured by CDF as a streaming source. These leverages checkpointing to track the progress of the stream processing and continue the stream from where it left off in the last execution.

The query then appends the data to the target table at each execution since it’s using the default writing mode, which is ‘append’.

|  |
| --- |
| spark.readStream  .option("**readChangeFeed**", "true")  .option("**startingVersion**", 0)  .table("customers")  .filter(col("\_change\_type").isin(["update\_postimage"]))  .writeStream  .option("checkpointLocation", "dbfs:/checkpoints")  .trigger (**availableNow=True**)  .table("customers\_updates") |

# Program #13

The following streaming query calculates business-level aggregates for each **non-overlapping** five-minute interval. Incremental state information is maintained for 10 minutes for late-arriving data.

**Pyspark.sql.functions.window** function bucketizes rows into one or more-time windows given a timestamp specifying column. In this query, we are performing aggregations per **order\_timestamp** for each non-overlapping five-minute interval.

**pyspark.sql.DataFrame.withWatermark** function allows you to only track state information for a window of time in which we expect records could be delayed. Here we define a watermark of 10 minutes.

|  |
| --- |
| spark.readStream  .table("orders\_cleaned")  .withWatermark("order\_timestamp", "10 minutes")  .groupBy(  window("order\_timestamp", "5 minutes").alias("time"),  "author")  .agg(  count("order\_id").alias("orders\_count"),  avg("quantity").alias("avg\_quantity"))  .writeStream  .option("checkpointLocation", "dbfs:/path/checkpoint")  .table("orders\_stats") |

# Program #14

A junior data engineer is testing the following code block to get the newest entry for each item added in the ‘sales’ table since the last table update.

Non-time-based window operations are not supported on streaming DataFrames. They need to be implemented inside a **foreachBatch** logic instead.

|  |
| --- |
| from pyspark.sql import functions as F  from pyspark.sql.window import Window    window = Window.partitionBy("item\_id").orderBy(F.col("item\_time").desc())    ranked\_df = (spark.readStream  .table("sales")  .withColumn("rank", F.rank().over(window))  .filter("rank == 1")  .drop("rank")  )   display(ranked\_df) |

**Note:**

* If you try to call such a window operation on a streaming DataFrames, this will generate an error indicating that “Non-time-based window operations are not supported on streaming DataFrames”.
* Instead, these window operations need to be implemented inside a **foreachBatch** logic.

**Corrected Program:**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| The error occurs because **Spark Structured Streaming** restricts standard Window functions (like rank() or row\_number()) when used directly on a stream. This is because, without a defined time boundary (watermark), Spark would have to keep all historical data in state indefinitely to calculate the rank.  To fix this, you should use **foreachBatch**, which allows you to treat each micro-batch as a static DataFrame, where standard window functions are permitted.  **The Corrected Program**  Here is how you can implement the logic using foreachBatch:  from pyspark.sql import functions as F  from pyspark.sql.window import Window  **# 1. Define the function to process each micro-batch**  def process\_batch(batch\_df, batch\_id):  # Within foreachBatch, batch\_df is a standard DataFrame  window\_spec = Window.partitionBy("item\_id").orderBy(F.col("item\_time").desc())    ranked\_df = (batch\_df  .withColumn("rank", F.rank().over(window\_spec))  .filter("rank == 1")  .drop("rank")  )    # Use display() inside for Databricks or write to a sink  ranked\_df.show()  **# 2. Set up the Read Stream**  streaming\_query = (spark.readStream  .table("sales")  .writeStream  .foreachBatch(process\_batch)  .start()  )  **Key Concepts to Remember**   | **Feature** | **Streaming Window** | **Static Window (via foreachBatch)** | | --- | --- | --- | | **Use Case** | Time-series analysis (e.g., averages over 10 mins). | Deduplication or ranking within a single micro-batch. | | **Requirement** | Requires a Watermark and a time column. | No watermark needed; treats batch as a snapshot. | | **State** | Maintains state across batches. | Clears state after the batch is processed. |   **Alternative: Deduplication**  If your goal is simply to get the most recent record per item\_id and you have a watermark defined, you can sometimes use dropDuplicates(["item\_id"]). However, if you specifically need the "latest" based on a timestamp and want to stay within the streaming API without foreachBatch, you must provide a watermark:  **Python**  # Only works if you want to deduplicate globally and have a watermark  ranked\_df = (spark.readStream  .table("sales")  .withWatermark("item\_time", "1 hour")  .dropDuplicates(["item\_id"]))  **Note:** In most "latest record" scenarios for streaming, **foreachBatch** is the most reliable way to ensure you are only looking at the data currently arriving in that specific trigger.  Would you like me to help you configure a specific output sink (like Delta or Parquet) for this foreachBatch logic? |  |

# Program #15

A data engineering team is building a LDP pipeline to clean and validate hotel reservations data. Some completed reservations have null check-in or check-out dates, which violates business rules.

To handle this, they implemented the following code:

|  |
| --- |
| rules = {  "valid\_check\_in": "(check\_in IS NOT NULL)",  "valid\_check\_out": "(check\_out IS NOT NULL)",  }  quarantine\_rules = "NOT({0})".format(" AND ".join(rules.values()))    @dlt.**table**(partition\_cols=["is\_quarantined"])  @dlt.**expect\_all**(rules)  def silver\_reservations():  return (  spark.readStream.table("bronze\_reservations")  .withColumn("is\_quarantined", expr(quarantine\_rules))  ) |

The function “**silver\_reservations** “streams all rows into the **silver\_reservations** table, flags those with missing check-in or check-out values as quarantined and partitions the table by the **is\_quarantined** flag.

# Program #16

Given the following query on the Delta table customers on which **Change Data Feed** is enabled. The **entire history of updated records will overwrite the target table** at each execution.

|  |
| --- |
| spark.read  .option("readChangeFeed", "true")  .option("startingVersion", 0)  .table("customers")  .filter(col("\_change\_type").isin(["update\_postimage"]))  .write  .mode(“overwrite”)  .table("customers\_updates") |

Reading table’s changes, captured by CDF, using **spark.read** means that you are reading them as a static source. So, each time you run the query, all table’s changes (starting from the specified **startingVersion**) will be read.

The query in the question then writes the data in mode “overwrite” to the target table, which completely overwrites the table at each execution.

# Program #17

In the following **streaming ingestion code** using Databricks **Auto Loader** - If a new column appears in the incoming JSON files that is not present in the existing schema, THEN

The stream fails and will not restart unless the schema is manually updated or the problematic data file is removed

|  |
| --- |
| spark.**readStream** \  .format("cloudFiles") \  .schema(expected\_schema) \  .option("cloudFiles.format", "json") \  .option("cloudFiles.schemaEvolutionMode", "failOnNewColumns") \  .load("s3://vendor/raw/sales/json/") \  **.writeStream** \  .option("checkpointLocation", "s3://vendor/checkpoints/sales") \  .start("sales\_table") |

* With the **failOnNewColumns** mode, the stream detects any new columns and fails immediately to enforce strict schema consistency.
* It will not automatically restart until the schema has been manually updated to include the new columns or the data files causing the schema mismatch are removed.
* This prevents silent schema drift and ensures deliberate schema management