Introduction to Streams and Tasks

🔴 1. Streaming on YouTube (Live Streaming)

This is what **most people think of** when they hear "streaming":

Aspect	Description
🔊 Type	**Media streaming**
📺 Example	YouTube Live, Netflix, Twitch
⏱️ Real-Time	Yes — audio/video is sent over the internet in real-time
💡 Purpose	Deliver content to users **continuously and live**
⬅️ Direction	Data flows **from server to viewer**

🧠 **In short**: You're watching or listening to a continuous stream of multimedia content, like a live concert or game stream.

🧊 2. Streaming in Snowflake (Data Stream)

This is a **technical, backend concept** used in databases and data engineering:

Aspect	Description
🔍 Type	**Change data tracking** (CDC)
📊 Example	Snowflake `STREAM` object
🕓 Real-Time	Not necessarily — usually **incremental but not continuous**
💡 Purpose	Keep track of **row-level changes** (inserts/updates/deletes) to a table, so you can process just the new data
⬅️ Direction	Data changes **from source table into downstream pipeline**

🧠 **In short**: You're tracking what changed in a table since the last time you checked — useful in ETL, auditing, and data sync.

🎯 Real-World Analogy

Imagine a **YouTube stream** as:

> Watching a live news broadcast.

And a **Snowflake stream** as:

> Getting a log of **what news stories were added, updated, or removed** from the newsroom database since yesterday — so you can publish just the new articles on your website.

🔁 Key Differences

Feature	YouTube Live Streaming	Snowflake Data Streaming
Domain	Media & content delivery	Data engineering & databases
Real-time?	Yes	Usually near-real-time / batch
What is streamed?	Video/audio	Table changes (insert/update/delete)
Viewer/Consumer	Human users	ETL tools, data pipelines

Snowflake supports continuous data pipelines with Streams and Tasks:

Streams

A *stream* object records the delta of change data capture (CDC) information for a table (such as a staging table), including inserts and other data manipulation language (DML) changes. A stream allows querying and consuming a set of changes to a table, at the row level, between two transactional points of time.

In a continuous data pipeline, table streams record when staging tables and any downstream tables are populated with data from business applications using continuous data loading and are ready for further processing using SQL statements.

Tasks

Definition: A *task* object runs a SQL statement, which can include calls to stored procedures. Tasks can run on a schedule or based on a trigger that you define, such as the arrival of data. You can use task graphs to chain tasks together, defining directed acyclic graphs (DAGs) to support more complex periodic processing.

Combining tasks with table streams is a convenient and powerful way to continuously process new or changed data. A task can transform new or changed rows that a stream surfaces using SYSTEM$STREAM_HAS_DATA. Each time a task runs, it can either consume the change data or skip the current run if no change data exists.

HOW is Streaming achieved?

An individual table stream tracks the changes made to rows in a *source table*. A table stream (also referred to as simply a “stream”) makes a “change table” available of what changed, at the row level, between two transactional points of time in a table. This allows querying and consuming a sequence of change records in a transactional fashion.

In Snowflake, a table stream is a way to track changes made to a table—insertions, updates, and deletions—at the row level. This feature is part of Snowflake's change data capture (CDC) capabilities.
Let’s break the explanation down:
🔄 What is a Table Stream?
A stream in Snowflake is an object that records row-level changes in a table over time. When you create a stream on a table, Snowflake starts tracking how that table changes—what rows were added, removed, or updated.
🕒 "Between Two Transactional Points of Time"
Snowflake streams track changes from the last time the stream was read up until the current moment. This gives you a change snapshot that represents what happened since the last consumption of the stream.
Imagine it as a difference report between two points in time.
📄 "Change Table"
A stream acts like a virtual table that you can query. It contains only the changed rows, along with metadata to indicate:
Whether a row was inserted, deleted, or updated
For updates: both the before and after versions of the row (if configured as an append-only = FALSE stream)
🔁 "Transactional Fashion"
Streams ensure that changes are tracked consistently and reliably, aligned with Snowflake’s transactional model. This means:
You won’t miss or double-count changes if you query the stream properly.
Once you consume changes (e.g., using SELECT * FROM my_stream), the stream advances its pointer to the latest snapshot.

🧠 **Why Use It?**
Streams are especially useful for:
ETL / ELT pipelines: Load only the changed data, rather than scanning the whole table.
Auditing: See what changed and when.
Replicating changes to other systems or tables.

🧪 Example:

```
-- Create a stream on a table
CREATE OR REPLACE STREAM my_table_stream ON TABLE my_table;
```

-- Later, you can query changes
SELECT * FROM my_table_stream;
This would return a result set of changes (inserts/updates/deletes) since the last query to my_table_stream.

Is "Change Table" a Part of Table Stream?

Yes — the **"change table"** is not a separate table. It's a **virtual view** that the **stream exposes**.

- When you query a **stream**, Snowflake gives you a **change table**.
  That change table shows what **changed in the base table** since the last time the stream was consumed.
- It's **not a new physical table**, but it behaves like a queryable object.

🎬 Example: Netflix Watch History

Use Case:
Netflix tracks users watching videos. It wants to **incrementally update a data warehouse or data lake** with **only the new changes** to the `watch_history` table, like:

- New rows (new shows watched)
- Updates (user rewatched a video, updated status)
- Deletes (user deletes history)

📊 Step 1: Create the Base Table

```
CREATE OR REPLACE TABLE watch_history (
    user_id STRING,
    show_id STRING,
    watch_time TIMESTAMP,
    status STRING  -- e.g., 'started', 'finished', 'rewatched'
);
```

💽 Step 2: Insert Some Data

```
INSERT INTO watch_history VALUES
('U001', 'S001', CURRENT_TIMESTAMP, 'started'),
('U002', 'S003', CURRENT_TIMESTAMP, 'finished');
```

🔄 Step 3: Create a Stream

```
CREATE OR REPLACE STREAM watch_history_stream ON TABLE watch_history;
```

This **starts tracking changes**, but nothing has changed yet **since the stream was created**

✍️ Step 4: Modify the Base Table

```
-- Insert a new row
INSERT INTO watch_history VALUES ('U003', 'S002', CURRENT_TIMESTAMP, 'started');
```

```
-- Update a row
UPDATE watch_history
SET status = 'rewatched'
WHERE user_id = 'U001' AND show_id = 'S001';

-- Delete a row
DELETE FROM watch_history
WHERE user_id = 'U002' AND show_id = 'S003';
```

🔍 **Step 5: Query the Stream (Change Table)**

```
SELECT * FROM watch_history_stream;
```

This will return a **change table** like:

METADATA$ACTION	METADATA$ISUPDATE	USER_ID	SHOW_ID	WATCH_TIME	STATUS
INSERT	FALSE	U003	S002	2025-07-08 13:15:00	started
DELETE	TRUE	U001	S001	2025-07-08 13:10:00	started
INSERT	TRUE	U001	S001	2025-07-08 13:16:00	rewatched
DELETE	FALSE	U002	S003	2025-07-08 13:12:00	finished

📌 METADATA$ACTION

- **Type**: String (`'INSERT'` or `'DELETE'`)
- **Purpose**: Tells you **what kind of change** happened to the row.

🔹 **Values:**

Value	Meaning
`INSERT`	A row was **inserted** into the base table.
`DELETE`	A row was **deleted** from the base table.

💡 **For updates**, Snowflake records **two events**:

1. A `DELETE` of the old row
2. An `INSERT` of the updated row

📌 METADATA$ISUPDATE

- **Type**: Boolean (`TRUE` or `FALSE`)
- **Purpose**: Indicates whether the row is part of an **update operation**.

🔹 **Values:**

Value	Meaning
`TRUE`	This row is part of an **update**. You'll see a DELETE and INSERT with this flag.
`FALSE`	This is a **pure insert** or **pure delete**, not related to an update.

How They Work Together (Example)

Let's say this row in the base table is updated:

```
-- Original row
('U001', 'S001', 'started')

-- Update operation
UPDATE watch_history SET status = 'rewatched' WHERE user_id = 'U001' AND show_id = 'S001';
```

Then querying the stream gives:

METADATA$ACTION	METADATA$ISUPDATE	USER_ID	SHOW_ID	STATUS
DELETE	TRUE	U001	S001	started
INSERT	TRUE	U001	S001	rewatched

✅ **Summary**

Column	Description
`METADATA$ACTION`	`'INSERT'` or `'DELETE'`, describing what change occurred
`METADATA$ISUPDATE`	`TRUE` if part of an update (you'll see both DELETE + INSERT)

Conclusion: Either TRUE update or just an INSERT/DELETE wit

UPSERT Vs Standard UPDATE

A standard `UPDATE` is **not** considered an **UPSERT** in Snowflake.

It is simply an **update operation**, and **Snowflake will track it as such** in a stream by splitting it into:

- A `DELETE` of the old row
- An `INSERT` of the new version

🔄 So What *Is* an UPSERT in Snowflake?

In Snowflake, an **UPSERT** is usually implemented using the `MERGE` statement.

✅ UPSERT = "Update if match, Insert if not"

The `MERGE` syntax combines **insert** and **update** logic:

```
MERGE INTO target_table AS T
USING source_table AS S
ON T.id = S.id
WHEN MATCHED THEN
    UPDATE SET T.status = S.status
WHEN NOT MATCHED THEN
    INSERT (id, status) VALUES (S.id, S.status);
```

So in the `watch_history` example above, an **UPSERT** would look like this:

🔁 UPSERT via MERGE (on `user_id`, `show_id`)

```
MERGE INTO watch_history AS target
USING (SELECT 'U001' AS user_id, 'S001' AS show_id, CURRENT_TIMESTAMP
AS watch_time, 'rewatched' AS status) AS source
ON target.user_id = source.user_id AND target.show_id = source.show_id
WHEN MATCHED THEN
    UPDATE SET status = source.status, watch_time = source.watch_time
WHEN NOT MATCHED THEN
    INSERT (user_id, show_id, watch_time, status)
   VALUES (source.user_id, source.show_id, source.watch_time, source.status);
```

Summary: UPDATE vs UPSERT

Operation	Type	Description	Captured in Stream As
`UPDATE`	Standard	Modifies existing rows only	`DELETE` + `INSERT` (with `METADATA$ISUPDATE = TRUE`)
`MERGE` (UPSERT)	Conditional	Updates if match, inserts if not	Same as above — results in `INSERT` or `UPDATE` (as DELETE+INSERT)

When to Use UPSERT (MERGE)

- Loading incremental data (e.g. from Kafka, S3)
- Syncing dimension tables (e.g., user or product info)
- ETL pipelines that need idempotency (avoid duplicates)

Streams can be created to query change data on the following objects:
- Standard tables, including shared tables.
- Views, including secure views
- Directory tables
- Dynamic tables
- Apache Iceberg™ tables with Limitations.
- Event tables
- External tables

Offset Storage

When created, a stream logically takes an initial snapshot of every row in the source object (e.g. table, external table, or the underlying tables for a view) by initializing a point in time (called an *offset*) as the current transactional version of the object. The change tracking system utilized by the stream then records information about the DML changes after this snapshot was taken. Change records provide the state of a row before and after the change. Change information mirrors the column structure of the tracked source object and includes additional metadata columns that describe each change event.

Streams use the current table schema. However, since streams may read deleted data to track changes over time, any incompatible schema changes between the offset and the advance can cause query failures.

Note that a stream itself does ***not*** contain any table data. A stream only stores an offset for the source object and returns CDC records by leveraging the versioning history for the source object. When the first stream for a table is created, several hidden columns are added to the source table and begin storing change tracking metadata. These columns consume a small

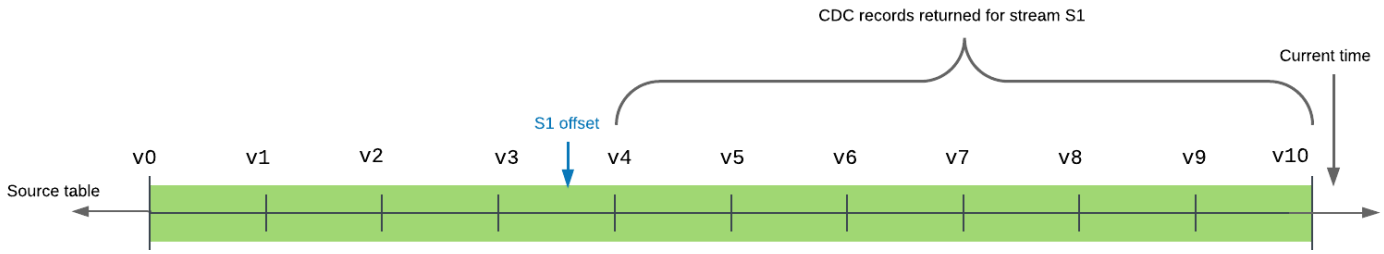
amount of storage. The CDC records returned when querying a stream rely on a combination of the *offset* stored in the stream and the *change tracking metadata* stored in the table. Note that for streams on views, change tracking must be enabled explicitly for the view and underlying tables to add the hidden columns to these tables.
It might be useful to think of a stream as a bookmark, which indicates a point in time in the pages of a book (i.e. the source object). A bookmark can be thrown away and other bookmarks inserted in different places in a book. So too, a stream can be dropped and other streams created at the same or different points of time (either by creating the streams consecutively at different times or by using Time Travel) to consume the change records for an object at the same or different offsets.
One example of a consumer of CDC records is a data pipeline, in which only the data in staging tables that has changed since the last extraction is transformed and copied into other tables.

Table Versioning

A new table version is created whenever a transaction that includes one or more DML statements is committed to the table. This applies to the following table types:

- Standard tables
- Directory tables
- Dynamic tables
- External tables
- Apache Iceberg™ tables
- Underlying tables for a view

In the transaction history for a table, a stream offset is located between two table versions. Querying a stream returns the changes caused by transactions committed after the offset and at or before the current time.
The following example shows a source table with 10 committed versions in the timeline. The offset for stream `s1` is currently between table versions `v3` and `v4`. When the stream is queried (or consumed), the records returned include all transactions between table version `v4`, the version immediately after the stream offset in the table timeline, and `v10`, the most recent committed table version in the timeline, inclusive.



A stream provides the minimal set of changes from its current offset to the current version of the table.

Multiple queries can independently consume the same change data from a stream without changing the offset. A stream advances the offset ***only*** when it is used in a DML transaction. This includes a Create Table As Select (CTAS) transaction or a COPY INTO location transaction and this behavior applies to both explicit and *autocommit* transactions. (By default, when a DML statement is executed, an autocommit transaction is implicitly started and the transaction is committed at the completion of the statement. This behavior is controlled with the AUTOCOMMIT parameter.) Querying a stream alone does not advance its offset, even within an explicit transaction; the stream contents must be consumed in a DML statement.
Note: To advance the offset of a stream to the current table version without consuming the change data in a DML operation, complete either of the following actions

- Recreate the stream (using the CREATE OR REPLACE STREAM syntax).
- Insert the current change data into a temporary table. In the INSERT statement, query the stream but include a WHERE clause that filters out all of the change data (e.g. `WHERE 0 = 1`).

When a SQL statement queries a stream within an explicit transaction, the stream is queried at the stream advance point (i.e. the timestamp) when the transaction began rather than when the statement was run. This behavior pertains both to DML statements and CREATE TABLE … AS SELECT (CTAS) statements that populate a new table with rows from an existing stream.
A DML statement that selects from a stream consumes all of the change data in the stream as long as the transaction commits successfully. To ensure multiple statements access the same change records in the stream, surround them with an explicit transaction statement (BEGIN .. COMMIT). This locks the stream. DML updates to the source object in parallel transactions are tracked by the change tracking system but do not update the stream until the explicit transaction statement is committed and the existing change data is consumed.

Repeatable Read Isolation(FIRST REFER THE NEXT PAGE EXAMPLE)

Streams support repeatable read isolation. In repeatable read mode, multiple SQL statements within a transaction see the same set of records in a stream. This differs from the read committed mode supported for tables, in which statements see any changes made by previous statements executed within the same transaction, even though those changes are not yet committed.
The delta records returned by streams in a transaction is the range from the current position of the stream until the transaction start time. The stream position advances to the transaction start time if the transaction commits; otherwise it stays at the same position.

Consider the following example:

Time	Transaction 1	Transaction 2
1	Begin transaction.	
2	Query stream s1 on table t1. The stream returns the change data capture records between the current position to the Transaction 1 start time. If the stream is used in a DML statement the stream is then locked to avoid changes by concurrent transactions.	
3	Update rows in table t1.	
4	Query stream s1. Returns the same state of stream when it was used at **Time** 2.	
5	Commit transaction. If the stream was consumed in DML statements within the transaction, the stream position advances to the transaction start time.	
6		Begin transaction.
7		Query stream s1. Results include table changes committed by Transaction 1.

Within Transaction 1, all queries to stream `s1` see the same set of records.
DML changes to table `t1` are recorded to the stream only when the transaction is committed.
In Transaction 2, queries to the stream see the changes recorded to the table in Transaction 1.
Note that if Transaction 2 had begun ***before*** Transaction 1 was committed, queries to the stream would have returned a snapshot of the stream from the position of the stream to the beginning time of Transaction 2 and would not see any changes committed by Transaction 1.

Let's now walk through a **realistic FinTech example**, using a well-known company like **Stripe**.

🏦 FinTech Company: Stripe

Stripe processes **millions of payments and refunds per day**, and must:

- Keep a secure, auditable record of transactions
- Incrementally update downstream systems (analytics, fraud detection, billing, etc.)
- Avoid double-counting or skipping changes

They use Snowflake streams to build a **reliable CDC (Change Data Capture)** pipeline for their `payments` table.

🎯 Business Case:

Stripe wants to **track recent payment activity**, and move it to a downstream fraud detection system in **batches**, using a stream on the `payments` table.

They also need to **ensure that a stream queried multiple times within a transaction gives a consistent snapshot**, so mid-transaction updates don't confuse processing logic.

🧱 Step-by-Step Example (Applying Repeatable Read with Streams)

✅ Setup: Table and Stream

```
-- Base payments table
CREATE OR REPLACE TABLE payments (
  payment_id STRING,
  user_id STRING,
  amount NUMBER,
  status STRING, -- 'processed', 'failed', etc.
  payment_time TIMESTAMP
);

-- Create a stream to track changes to payments
CREATE OR REPLACE STREAM payments_stream ON TABLE payments;
```

📽️ Use Case Walkthrough: Multi-Step Transaction in Stripe's Pipeline

Let's simulate a fraud detection pipeline that reads new payments **within a transaction**. We'll use **repeatable read** to ensure consistency.

🕑 Time 1: Transaction Begins

```
BEGIN; -- Transaction 1
```

Stripe's ETL pipeline opens a transaction to read and process a batch of payment changes.

🕑 Time 2: Query the Stream

```
SELECT * FROM payments_stream;
```

This snapshot includes **only changes** made to `payments` **up to this transaction's start time**.

🕑 Time 3: New Changes to the Table (Outside This Transaction)

```
-- Outside Transaction 1 (maybe another Stripe service writes this)
INSERT INTO payments VALUES ('p999', 'u001', 100.0, 'processed', CURRENT_TIMESTAMP);
```

This change does **not appear** in Transaction 1's stream snapshot. It will appear only in future transactions.

🕑 Time 4: Query the Stream Again Within the Same Transaction

```
SELECT * FROM payments_stream;
```

⚠️ Returns **same data as Time 2**, even though new changes were made in Time 3.

This is **repeatable read** — all queries to the stream inside the same transaction are **isolated from concurrent changes**.

🕑 Time 5: Commit Transaction

```
-- Processed payments are now sent to fraud engine
-- Advance the stream position
COMMIT;
```

The stream pointer now moves forward to the **start of this transaction**. So, next time the stream is queried, it will start **from here**.

🕑 Time 6: New Transaction Begins

```
BEGIN; -- Transaction 2
```

This transaction will now see new changes made in Time 3.

🕑 Time 7: Query Stream in New Transaction

SELECT * FROM payments_stream;

Now the stream includes:

('p999', 'u001', 100.0, 'processed', '2025-07-08 10:33:00')

Because the `INSERT` made earlier is now committed and visible to the stream.

🧠 Why This Is Critical in FinTech

In companies like Stripe, Revolut, or PayPal:

Need	Why Streams with Repeatable Read Helps
✅ Prevent double-processing of payments	Each batch is **isolated and consistent** within a transaction
✅ Accurate fraud detection pipelines	Fraud models work with **stable, clean input**
✅ Auditing and replay safety	You can **replay from exact stream positions**, even across failures
✅ No data race conditions	Even if new data is inserted mid-batch, your stream read stays stable

🔁 Summary

Concept	Snowflake Stream Feature
Isolation level	**Repeatable Read** for Streams
Stream position update	Only advances **after COMMIT**
Use in FinTech	Ensures **clean, consistent snapshots** of financial data
Real-world impact	Avoids duplicate billing, false fraud flags, or missed data in pipelines

Stream Columns

A stream stores an offset for the source object and not any actual table columns or data. When queried, a stream accesses and returns the historic data in the same shape as the source object (i.e. the same column names and ordering) with the following additional columns:

METADATA$ACTION

Indicates the DML operation (INSERT, DELETE) recorded.

METADATA$ISUPDATE

Indicates whether the operation was part of an UPDATE statement. Updates to rows in the source object are represented as a pair of DELETE and INSERT records in the stream with a metadata column METADATA$ISUPDATE values set to TRUE.

Note that streams record the differences between two offsets. If a row is added and then updated in the current offset, the delta change is a new row. The METADATA$ISUPDATE row records a FALSE value.

METADATA$ROW_ID

Specifies a unique, immutable row ID for tracking changes over time. If CHANGE_TRACKING is disabled and later re-enabled on the stream's source object, the row ID could change.

Snowflake provides the following guarantees with respect to METADATA$ROW_ID:

1. The METADATA$ROW_ID depends on the stream's source object.
   For instance, a stream `stream1` on table `table1` and stream `stream2` on table `table1` produce the same METADATA$ROW_IDs for the same rows, but a stream `stream_view` on view `view1` is not guaranteed to produce the same METADATA$ROW_IDs as `stream1`, even if `view` is defined using the statement `CREATE VIEW view AS SELECT * FROM table1`.
2. A stream on a source object and a stream on the source object's clone produce the same METADATA$ROW_IDs for the rows that exist at the time of the cloning.
3. A stream on a source object and a stream on the source object's replica produce the same METADATA$ROW_IDs for the rows that were replicated.

Types of Streams

The following stream types are available based on the metadata recorded by each:

Standard

Supported for streams on standard tables, dynamic tables, Snowflake-managed Apache Iceberg™ tables, directory tables, or views. **A standard (i.e. delta) stream tracks all DML changes to the source object, including inserts, updates, and deletes (including table truncates)**. This stream type performs a join on inserted and deleted rows in the change set to provide the row level delta. As a net effect, for example, a row that is inserted and then deleted between two transactional points of time in a table is removed in the delta (i.e. is not returned when the stream is queried).

Note: Standard streams cannot retrieve change data for geospatial data. We recommend creating append-only streams on objects that contain geospatial data.

Append-only

Supported for streams on standard tables, dynamic tables, Snowflake-managed Apache Iceberg™ tables, or views. An append-only stream exclusively tracks row inserts. **Update, delete, and truncate operations are not captured by append-only streams**. For instance, if 10 rows are initially inserted into a table, and then 5 of those rows are deleted before advancing the offset for an append-only stream, the stream would only record the 10 inserted rows.

An append-only stream specifically returns the appended rows, making it notably more performant than a standard stream for extract, load, and transform (ELT), and similar scenarios reliant solely on row inserts. For example, a source table can be truncated immediately after the rows in an append-only stream are consumed, and the record deletions do not contribute to the overhead the next time the stream is queried or consumed.

Creating an append-only stream in a target account using a secondary object as the source is not supported.

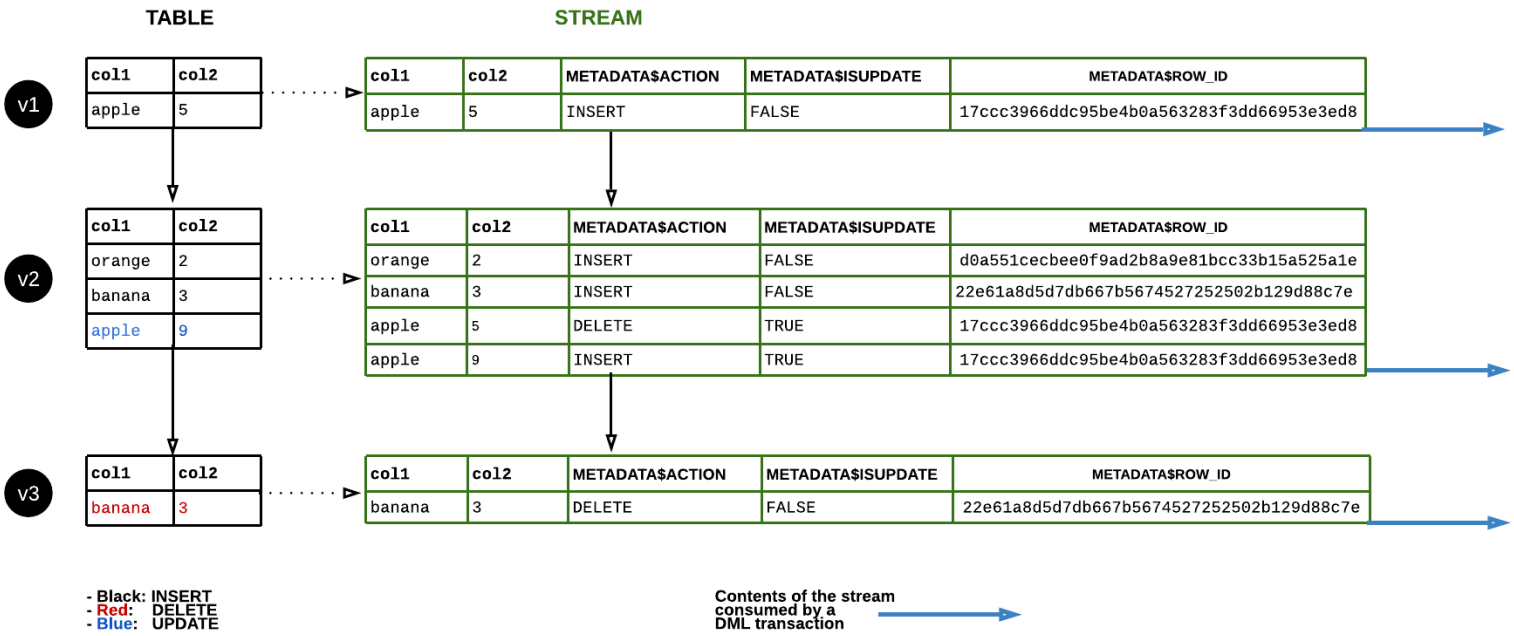
Insert-only

Supported for streams on externally managed Apache Iceberg™ or external tables. An insert-only stream tracks row inserts only; they do not record delete operations that remove rows from an inserted set (i.e. no-ops). For example, in-between any two offsets, if `File1` is removed from the cloud storage location referenced by the external table, and `File2` is added, the stream returns records for the rows in `File2` only, regardless of whether `File1` was added before or within the requested change interval. Unlike when tracking CDC data for standard tables, access to the historical records for files in cloud storage is not governed by or guaranteed to Snowflake.

Overwritten or appended files are essentially handled as new files: The old version of the file is removed from cloud storage, but the insert-only stream does not record the delete operation. The new version of the file is added to cloud storage, and the insert-only stream records the rows as inserts. The stream does not record the diff of the old and new file versions. Note that appends may not trigger an automatic refresh of the external table metadata, such as when using Azure AppendBlobs.

Data Flow

The following diagram shows how the contents of a standard stream change as rows in the source table are updated. Whenever a DML statement consumes the stream contents, the stream position advances to track the next set of DML changes to the table (i.e. the changes in a table version):



Snowflake TASKS: FinTech Use Case

What is a Snowflake Task?

> A **Snowflake Task** is a scheduled or event-driven job that **automates SQL-based operations** — such as ETL, data processing, alerts, or streaming consumption — **without the need for external orchestration tools**.

It can:

- Run **on a schedule**
- Be triggered **by another task**
- Or run **in response to table changes** (via streams)

Official Snowflake documentation definition: *task* object runs a SQL statement, which can include calls to stored procedures. Tasks can run on a schedule or based on a trigger that you define, such as the arrival of data. You can use task graphs to chain tasks together, defining directed acyclic graphs (DAGs) to support more complex periodic processing.

💼 Why Are Tasks Critical in FinTech?

FinTech companies like **Stripe**, **Revolut**, or **Square** deal with:

- High-volume **transactional data**
- Near-real-time **fraud detection**
- Regulatory & **compliance reporting**
- Complex **data pipelines** needing precise orchestration

✅ Snowflake `TASKS` provide:

- **Deterministic, atomic data processing**
- **Built-in orchestration** of pipelines without Airflow or cron
- **Fine-grained control** over CDC (Change Data Capture) logic

🧱 Core Components of a Task

Element	Purpose
`SCHEDULE`	Run periodically (e.g., every 5 min)
`AFTER`	Chain tasks in dependency trees
`WHEN`	Run only if a condition is met (SQL boolean)
`WAREHOUSE`	Compute resource to run task
`AS`	The SQL logic the task will execute

🏦 FinTech Use Case: Stripe Payment Pipeline

🎯 Objective

Stripe wants to:

1. Track **new payment events** using a `STREAM`
2. Process them into a **fraud-check table**
3. Chain this into another task that flags **high-risk payments**
4. Run this **every 5 minutes**, reliably and automatically

📂 Step-by-Step Stripe's Task Pipeline

🔹 1. Base Table: `payments`

```
CREATE OR REPLACE TABLE payments (
  payment_id STRING,
  user_id STRING,
  amount NUMBER,
  status STRING, -- 'processed', 'failed'
  payment_time TIMESTAMP
);
```

🔹 2. Create a Stream on `payments`

```
CREATE OR REPLACE STREAM payments_stream ON TABLE payments;
```

🔹 3. Create a `fraud_events` table

```
CREATE OR REPLACE TABLE fraud_events (
  payment_id STRING,
  user_id STRING,
  amount NUMBER,
  risk_score NUMBER,
  flagged_at TIMESTAMP
);
```

🔹 4. Create a Task to Process Streamed Payments

```
CREATE OR REPLACE TASK process_new_payments
WAREHOUSE = analytics_wh
SCHEDULE = '5 MINUTE'
WHEN SYSTEM$STREAM_HAS_DATA('payments_stream')
AS
INSERT INTO fraud_events
SELECT
  p.payment_id,
  p.user_id,
  p.amount,
  CASE
    WHEN p.amount > 5000 THEN 90  -- high value, high risk
    WHEN p.status = 'failed' THEN 70
    ELSE 20
  END AS risk_score,
  CURRENT_TIMESTAMP
FROM payments_stream p;
--This task only runs if new payment data exists in the stream, ensuring efficient use of compute.
```

🔹 5. Create a Downstream Task to Flag High-Risk Payments

```
CREATE OR REPLACE TABLE flagged_payments (
  payment_id STRING,
  user_id STRING,
  reason STRING,
  flagged_at TIMESTAMP
);

CREATE OR REPLACE TASK flag_high_risk_payments
WAREHOUSE = analytics_wh
AFTER process_new_payments
AS
INSERT INTO flagged_payments
SELECT
  payment_id,
  user_id,
  'High Risk Score > 80' AS reason,
  CURRENT_TIMESTAMP
FROM fraud_events
WHERE risk_score > 80;
```

⛓️ Task Chain Overview

[process_new_payments] ⟶ [flag_high_risk_payments]
every 5 min

This builds an internal ETL **pipeline** without needing Airflow or dbt scheduler.

📈 Real-World Benefits in FinTech

Capability	FinTech Impact
✅ Stream-based triggers	Real-time fraud detection & alerts
✅ Task chaining (`AFTER`)	Build multi-stage pipelines (e.g., raw → filtered → alerts)
✅ `SYSTEM$STREAM_HAS_DATA`	Avoid wasteful compute when no new data
✅ SQL-based orchestration	Minimal tools — just Snowflake
✅ Auditable logic	Full query history for compliance audits (SOX, PCI-DSS)

🔐 Best Practices for FinTech Use

Practice	Why Important
🔒 Use dedicated warehouses	Prevent noisy neighbors affecting payment tasks
🛠️ Use `SHOW TASK HISTORY`	Monitor task runs, failures, execution time
🔁 Use `ALTER TASK ... RESUME`	Enable tasks explicitly after creation for control
🧪 Test with `SCHEDULE = 'USING CRON * * * * * UTC'`	Test tasks in non-prod at controlled intervals
📉 Minimize costs with `WHEN SYSTEM$STREAM_HAS_DATA(...)`	Run tasks **only when needed**

📜 Monitoring Example

```
SHOW TASKS LIKE 'process_new_payments';

SELECT * FROM TABLE(INFORMATION_SCHEMA.TASK_HISTORY())
WHERE NAME = 'PROCESS_NEW_PAYMENTS'
ORDER BY SCHEDULED_TIME DESC;
```

🧠 TL;DR

Concept	Description
Snowflake Task	A scheduled or triggered SQL job
FinTech Usage	Payment pipelines, fraud flags, compliance ETL
Best Feature	Built-in orchestration with stream awareness
Example Flow	`payments → fraud_events → flagged_payments`
Bonus	All native to SQL, no external orchestration needed

HAPPY LEARNING
REGARDS
SARANSH JAIN