03: Dynamic Tables

CONFLUENT
Global Education

Module Overview

This module contains five lessons:

- Traditional SQL vs. Streaming SQL
- Stream-Table Duality
- Dynamic Table Creation
- Column Types
- State Store & Stateful Operations

03a: Traditional SQL vs. Streaming SQL

Description

Comparison between Traditional SQL and Streaming SQL. Advantages and disadvantages.

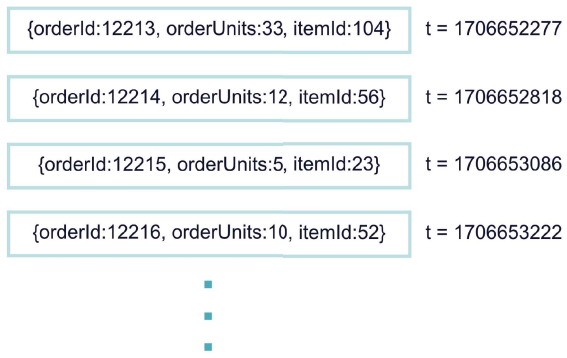
Traditional SQL vs. Streaming SQL

TRADITIONAL SQL	STREAMING SQL
Tables are bounded	Streams are an infinite sequence of records
A query has to access to the complete input data	A streaming query cannot access all data when it is started and has to "wait" for data to be streamed in
A batch query terminates after it produced a **fixed sized** result	A streaming query **continuously** updates its result based on the received records and never completes

Tables vs. Streams

Table (Relational Database)	Streaming Data

orderId (int)	orderUnits (int)	itemId (int)
12213	33	104
12214	12	56
12215	5	23
12216	10	52



SQL (Structured Query Language)

SQL is a strongly typed language
Needs an explicit table schema

Table (Relational Database) 👍	Streaming Data 👎

orderId (int)	orderUnits (int)	itemId (int)
12213	33	104
12214	12	56
12215	5	23
12216	10	52

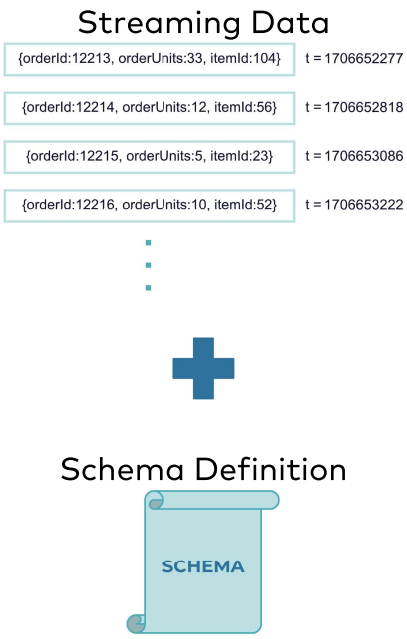
{orderId:12213, orderUnits:33, itemId:104} t = 1706652277

{orderId:12214, orderUnits:12, itemId:56} t = 1706652818

{orderId:12215, orderUnits:5, itemId:23} t = 1706653086

{orderId:12216, orderUnits:10, itemId:52} t = 1706653222

Streaming Data + Schema → Dynamic Table



Dynamic Table (Flink SQL)

orderId (int)	orderUnits (int)	itemId (int)
12213	33	104
12214	12	56
12215	5	23
12216	10	52

Same Result?

Running the same SQL query on both systems:

```
SELECT count(*) AS `count` FROM orders;
```

Table (Relational Database)	Dynamic Table (Flink SQL)

orderId (int)	orderUnits (int)	itemId (int)
12213	33	104
12214	12	56
12215	5	23
12216	10	52

orderId (int)	orderUnits (int)	itemId (int)
12213	33	104
12214	12	56
12215	5	23
12216	10	52

Will there be any difference?

Traditional SQL vs. Streaming SQL - Query Example

SQL Query:

```
SELECT count(*) AS `count` FROM orders;
```

orderId (int)	orderUnits (int)	itemId (int)
12213	33	104
12214	12	56
12215	5	23
12216	10	52

Result:

Traditional SQL

```
 count
--------
   4
```

Streaming SQL

```
op              count
----------------------
+I                  1
-U                  1
+U                  2
-U                  2
+U                  3
-U                  3
+U                  4
```

Traditional SQL vs. Streaming SQL - Query Example

After 2 minutes of running the query a new order is created:

```
SELECT count(*) AS `count` FROM
orders;
```

orderId (int)	orderUnits (int)	itemId (int)
12213	33	104
12214	12	56
12215	5	23
12216	10	52
12217	4	31

Result:

Traditional SQL

```
 count
--------
   4
```

Streaming SQL

```
op              count
----------------------
+I                  1
-U                  1
+U                  2
-U                  2
+U                  3
-U                  3
+U                  4
-U                  4
+U                  5
```

Lesson 03b

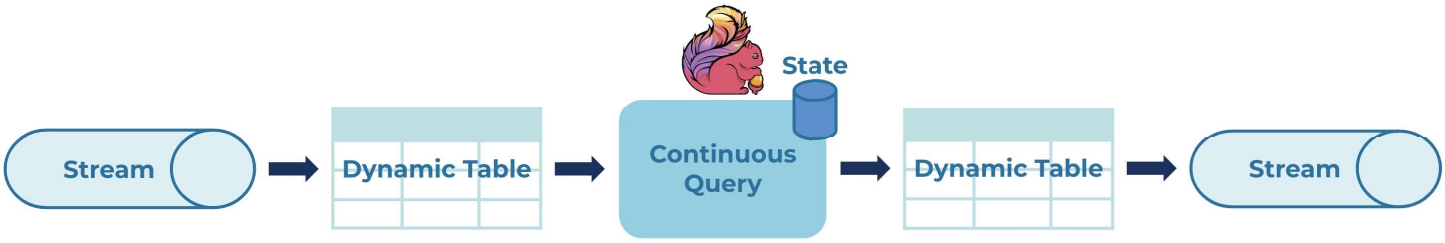
03b: Stream-Table Duality

Description

Understand the relationship between a stream and a table in Flink SQL.

Stream-Table Duality

- Streams can be turned into dynamic tables and dynamic tables into streams
- A stream is a record of changes in a dynamic table over time (a.k.a changelog)



So, is Flink SQL a database? **No, bring your own data and systems!**

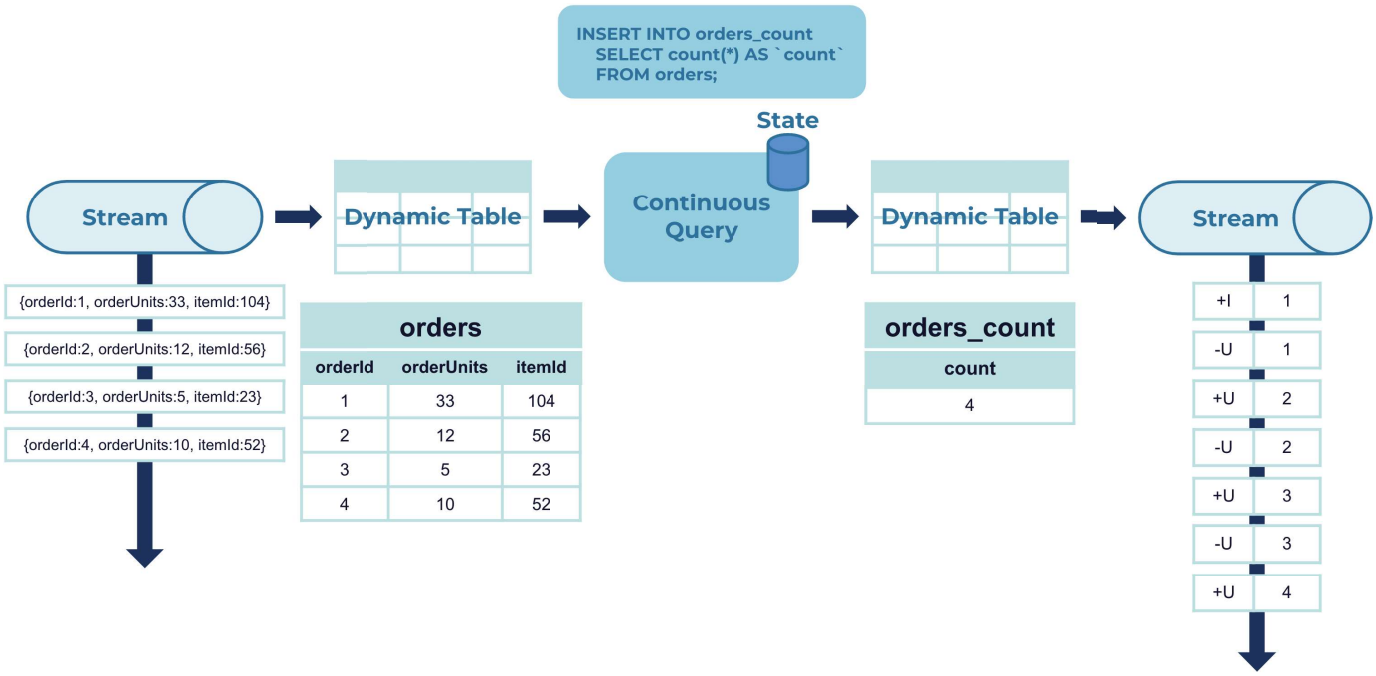
A stream is a record of changes in a dynamic table over time, known as a changelog. The changelog stream contains all changes, including "before" and "after" values, and can reconstruct the current state of the table at any point in time.

Querying dynamic tables yields a Continuous Query. A continuous query never terminates and produces dynamic results - another dynamic table. The query continuously updates its (dynamic) result table to reflect changes on its (dynamic) input tables. Essentially, a continuous query on a dynamic table is very similar to a query that defines a materialized view.

1. A stream is converted into a dynamic table.
2. A continuous query is evaluated on the dynamic table yielding a new dynamic table.
3. The resulting dynamic table is converted back into a stream.

Dynamic tables are foremost a logical concept. Dynamic tables are not necessarily (fully) materialized during query execution. The only state that is actually materialized by the Flink SQL runtime is whatever is strictly necessary to produce correct results for the specific query being executed.

Stream-Table Duality



A stream is a log of changes in a dynamic table over time, known as a changelog. The changelog stream contains all changes, including "before" and "after" values, and can reconstruct the current state of the table at any point in time.

Table Changes

A dynamic table can be continuously modified by:

- Inserting a new row
- Updating an existing row
- Deleting a row

Abbrev	Name	Description
+ I	Insertion	Adds a new row to the table
- U	Update Before	Retracts a previously emitted result
+ U	Update After	Updates a previously emitted result
- D	Deletion	Removes the last result

Changelog Modes - Table to Stream Conversion

- Table changes need to be encoded into the Stream (changelog)
- There are 3 changelog modes for Table to Stream conversion

Changelog Mode	Supported Table Changes
Append-Only	+I
Updating:	
• Retract	+I -U +U -D
• Upsert	+I +U -D

For Kafka, retract table changes (UPDATE BEFORE and UPDATE AFTER) are encoded into the Kafka message in the metadata header.

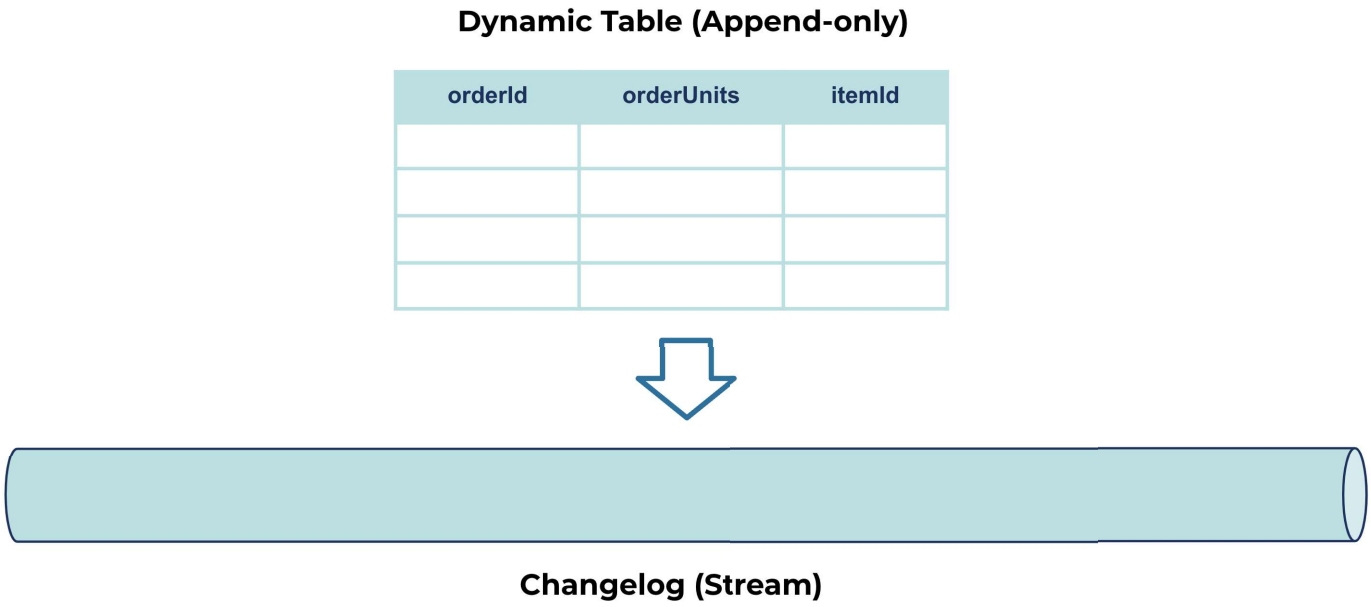
UPSERT BEFORE:

```
"key": "op",
"value": "\u0001"
```

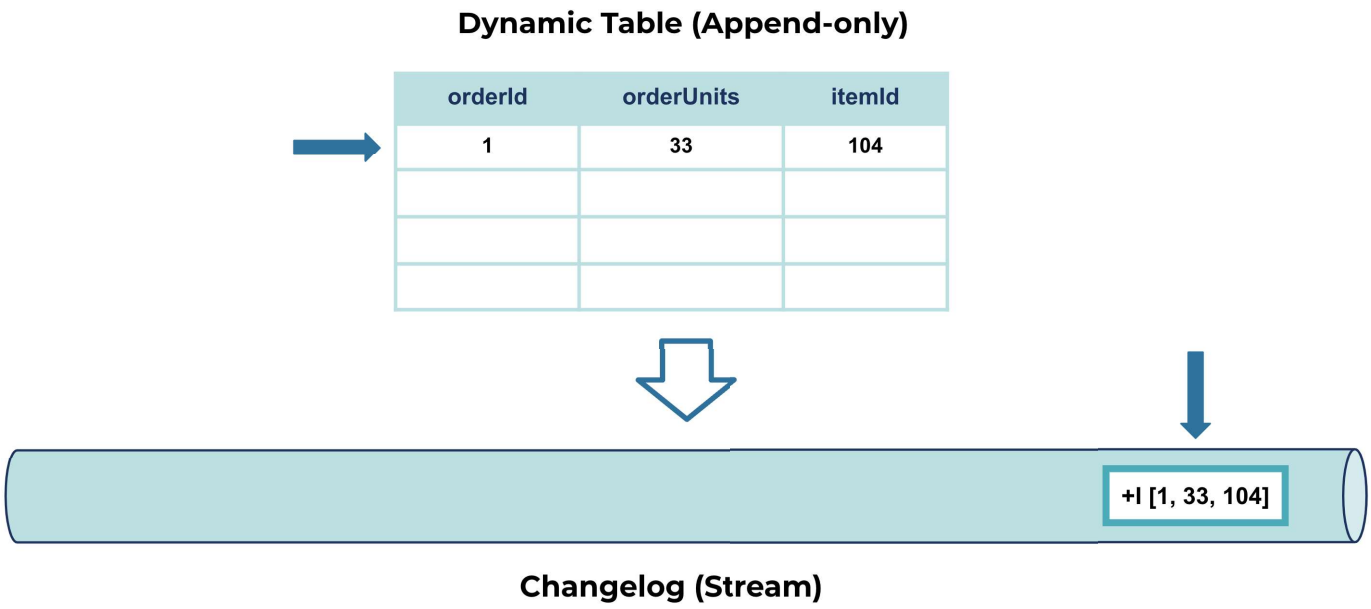
UPSERT AFTER:

```
"key": "op",
"value": "\u0002"
```

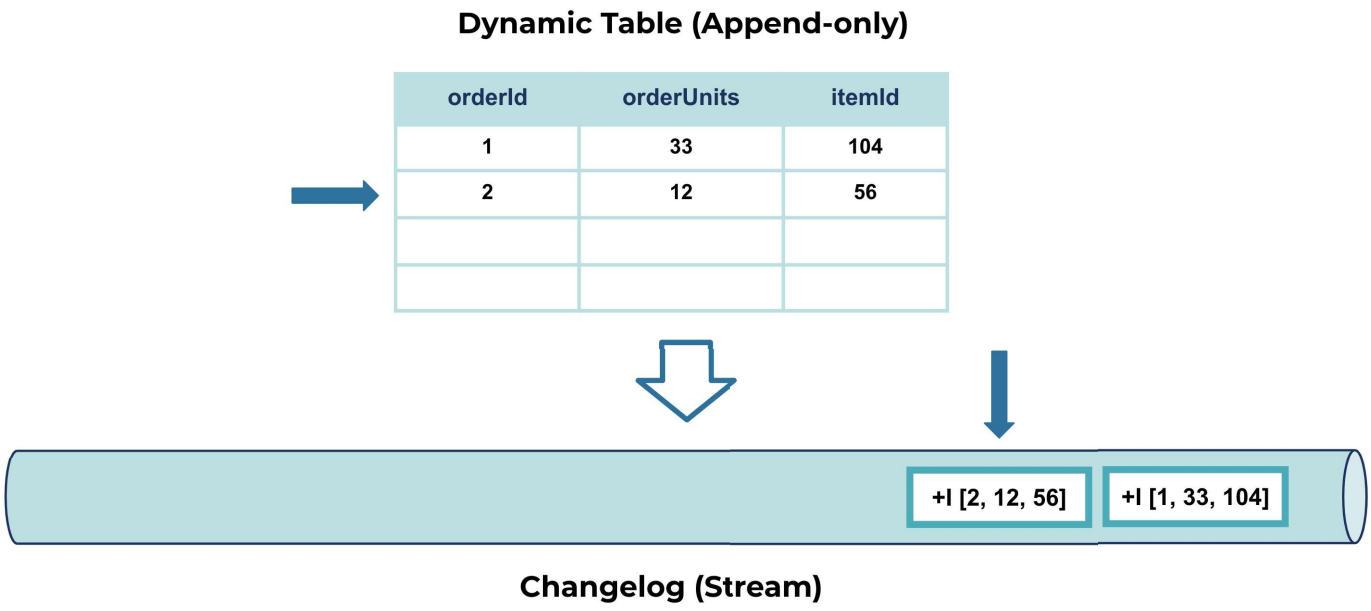
Append-Only Mode (I)



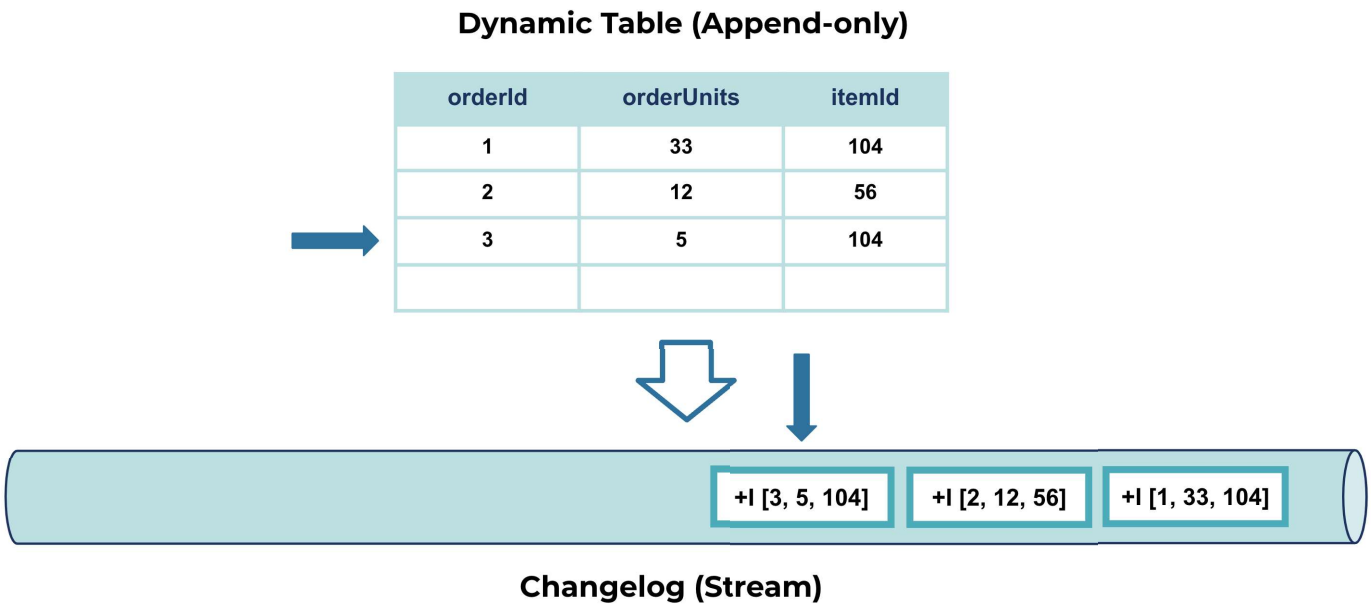
Append-Only Mode (II)



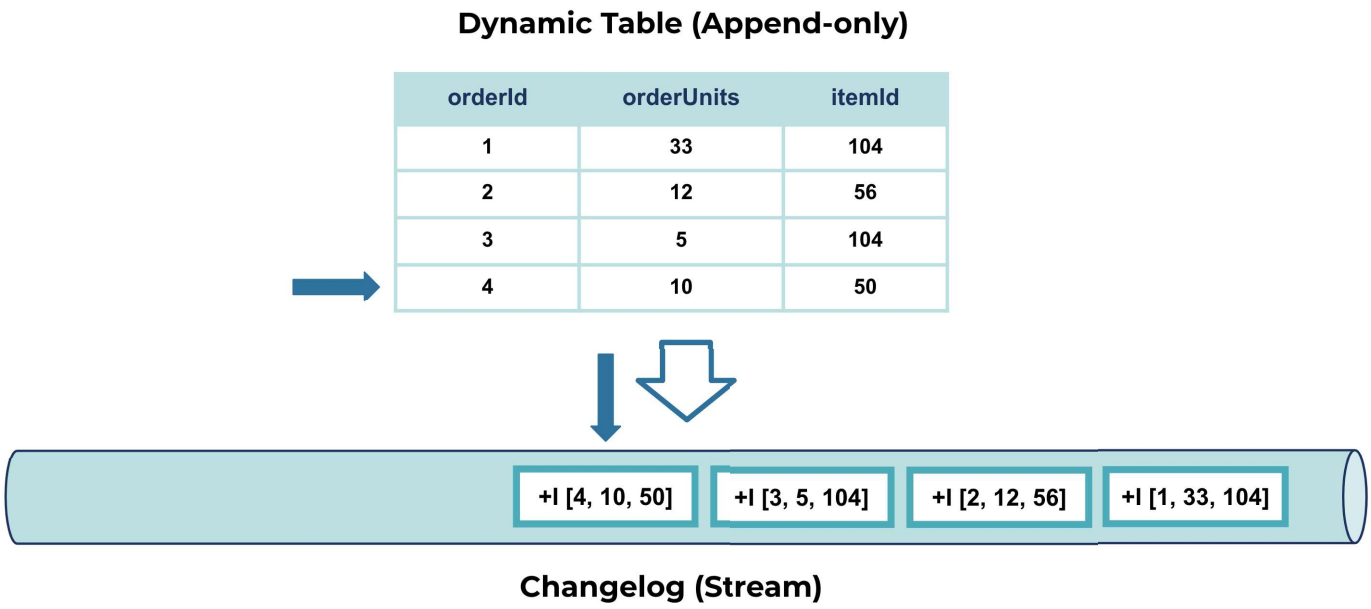
Append-Only Mode (III)



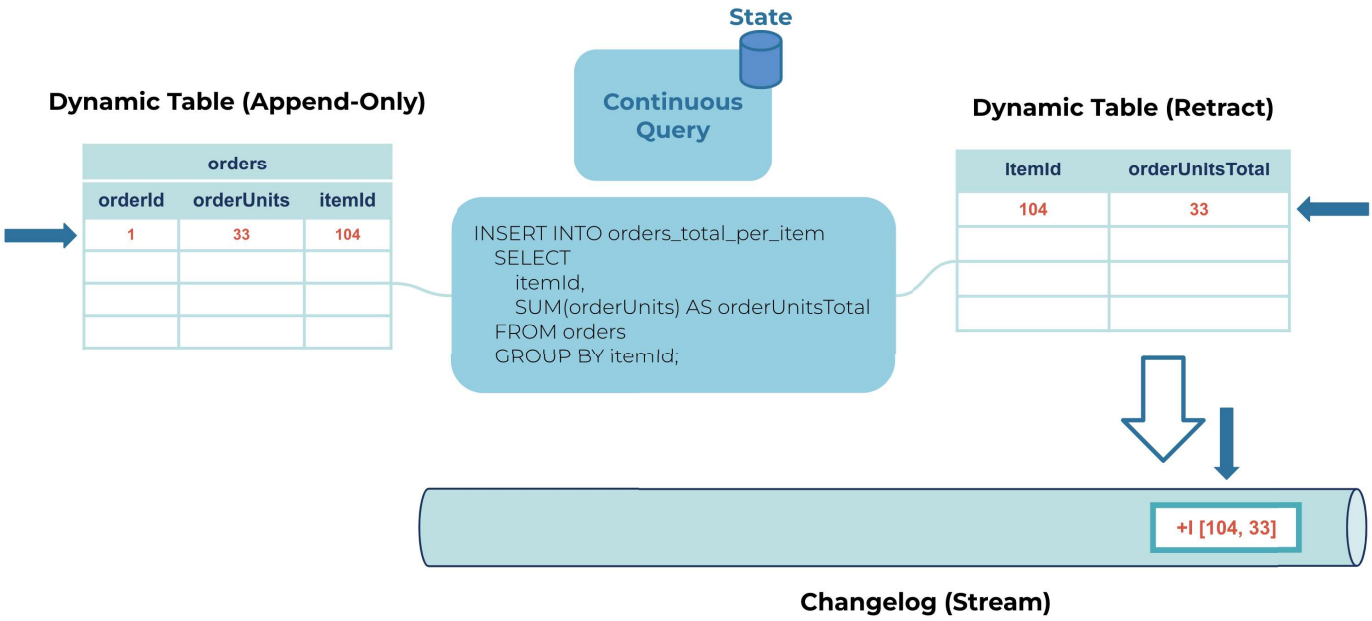
Append-Only Mode (IV)



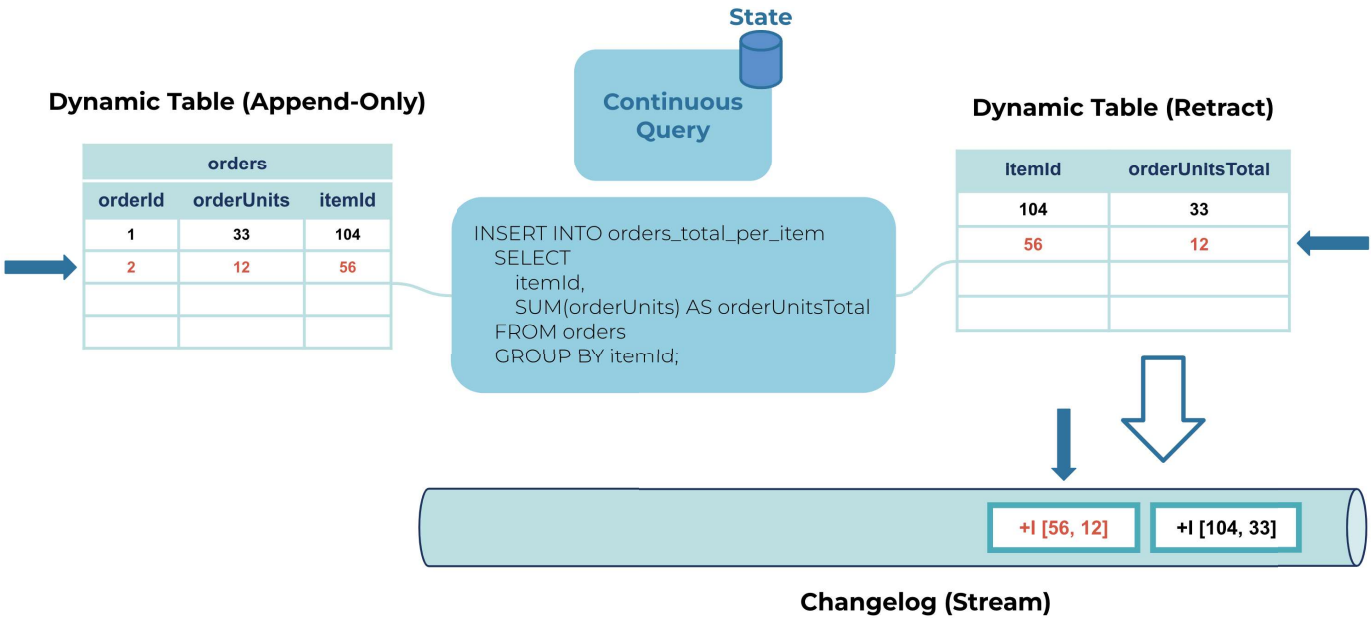
Append-Only Mode (V)



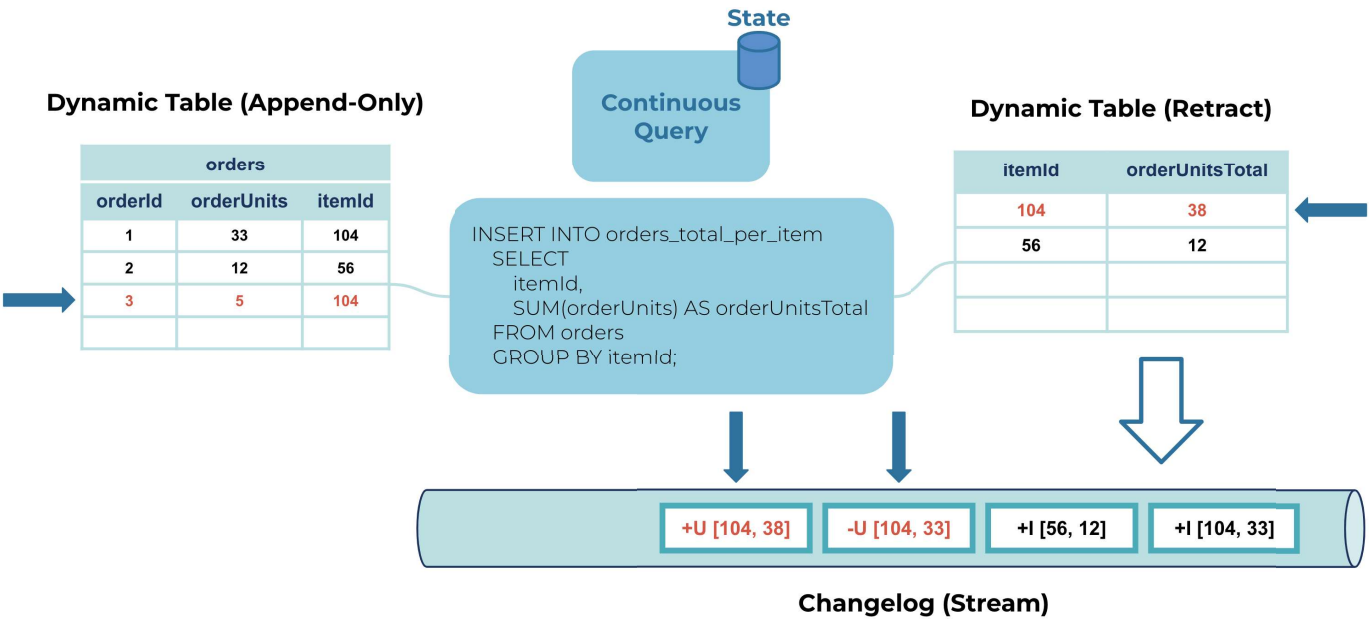
Retract Mode (I)



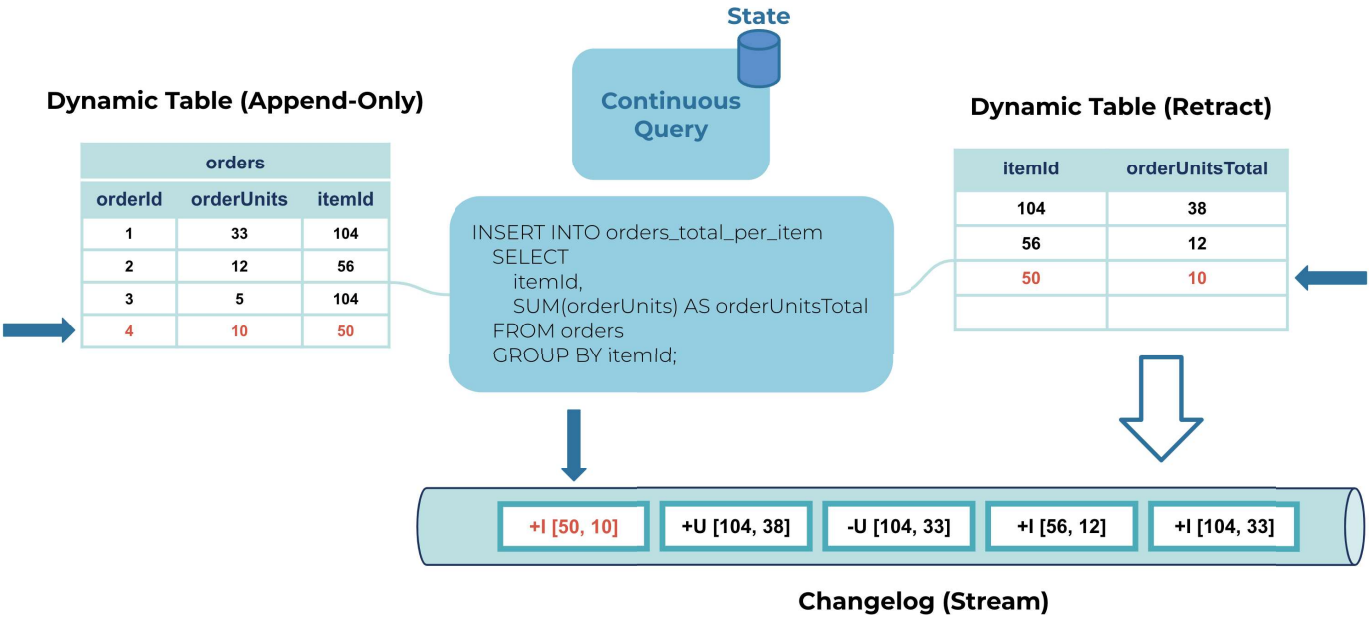
Retract Mode (II)



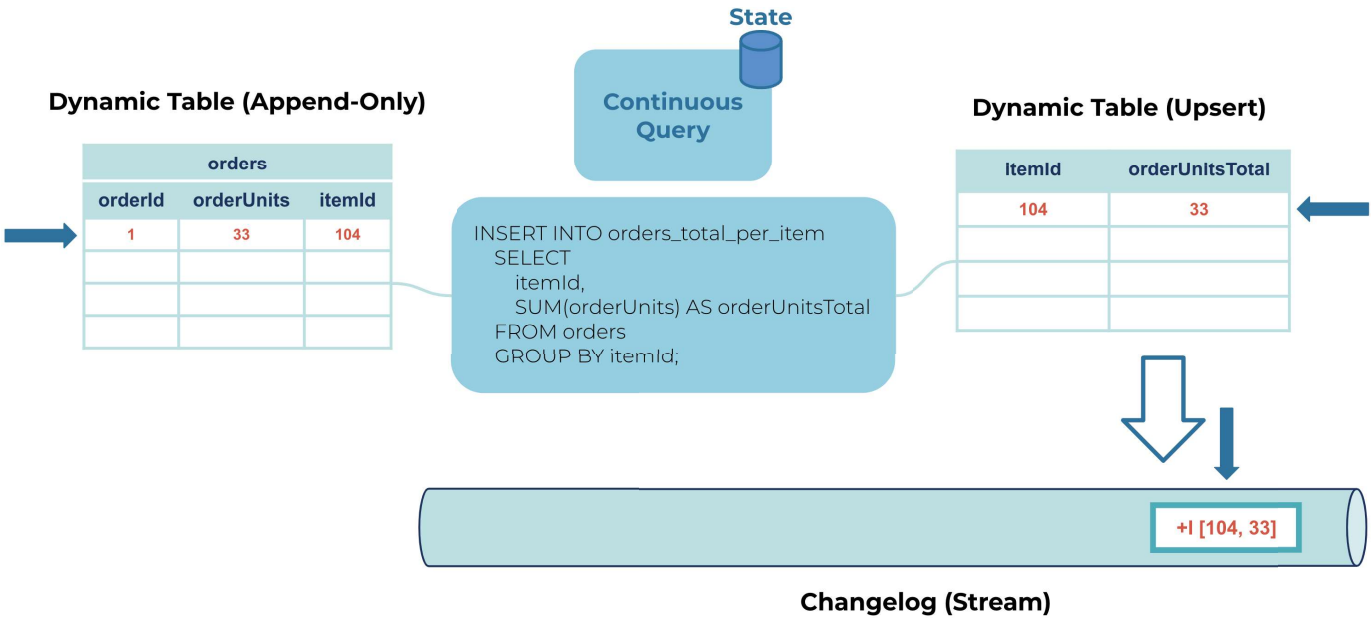
Retract Mode (III)



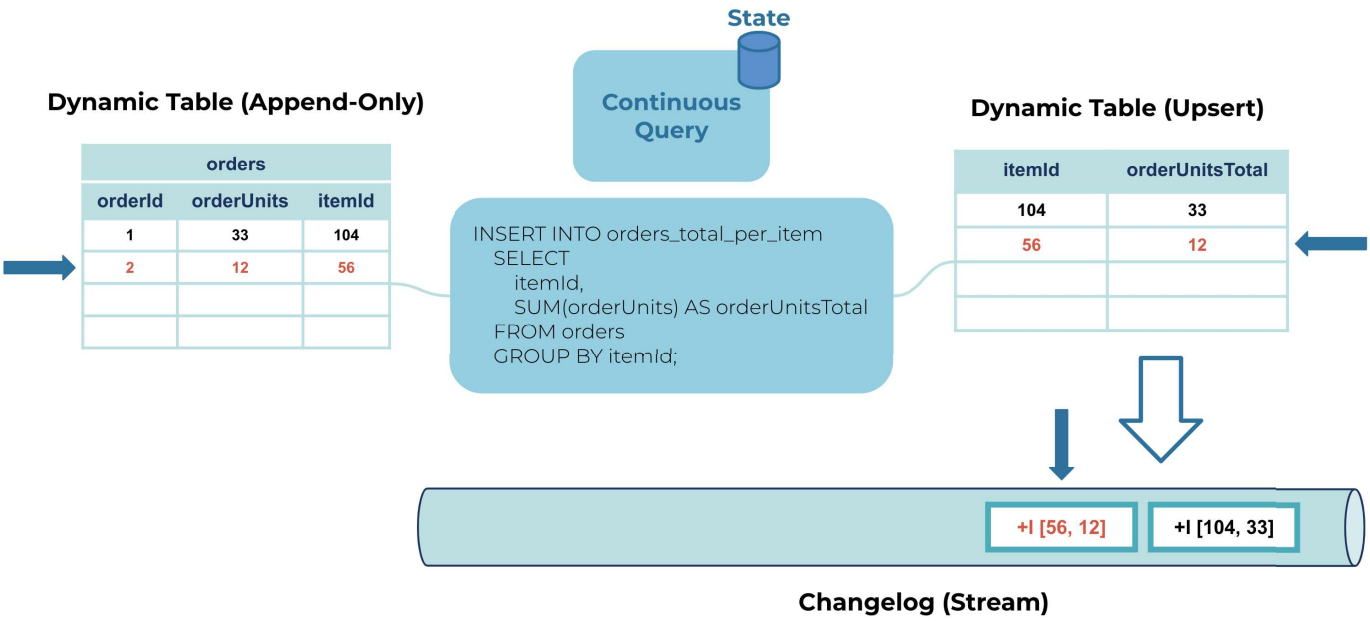
Retract Mode (IV)



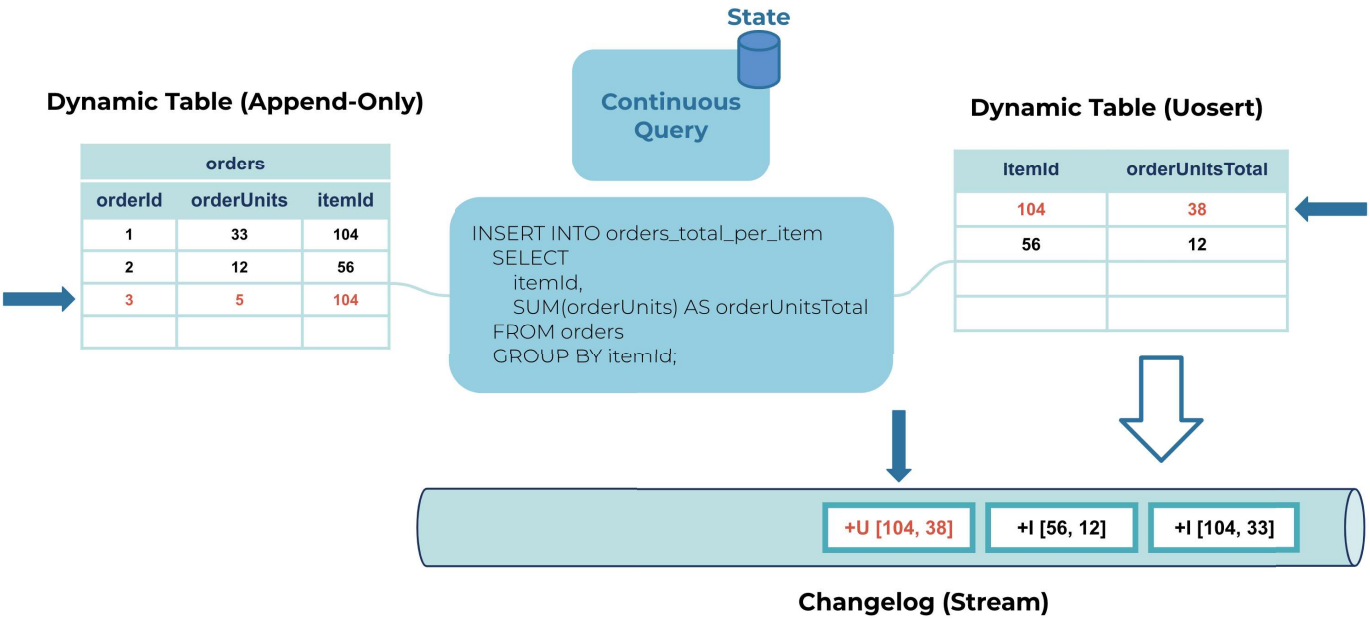
Upsert Mode (I)



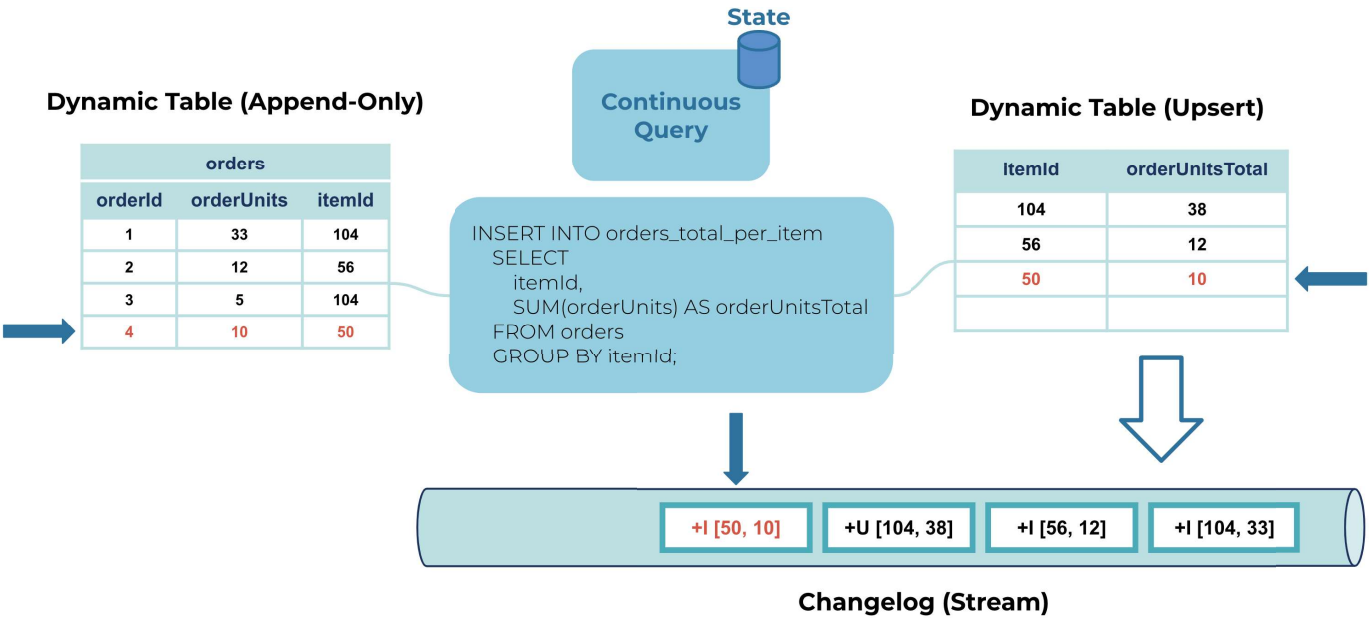
Upsert Mode (II)



Upsert Mode (III)



Upsert Mode (IV)



Retract vs. Upsert

RETRACT

- No PRIMARY KEY requirements
- Can work with NULL keys
- Most flexible changelog mode (works for almost every external system)

UPSERT

- Requires a PRIMARY KEY (unique and non null)
- Traffic + computation optimization (Save ~50% of traffic)
- In-place updates (idempotency)

Idempotency

Idempotent operations can be applied multiple times without changing the result beyond the initial application.

UPSERT:

- In upsert mode, when reading data back from Kafka, only the last record for each key is considered. This effectively makes previous duplicates with the same key irrelevant.
- In this context, writing the same key multiple times will have the same effect as writing it once, as only the latest write for a key is considered.

RETRACT:

- In retract mode, inserts (addition of new records) and retracts (removal of existing records) are paired to manage updates.
- Retract mode requires careful handling to ensure that inserts and retracts are balanced correctly. If the stream processing is disrupted, such as by failure and recovery, duplicates or missing retractions can lead to inconsistencies.

Changelog Modes - Analogy with Kafka Streams/ksqlDB

Kafka Streams / ksqlDB	Flink	Description
KStream / Stream	Dynamic Table (Append-Only)	Records are always appended
KTable / Table	Dynamic Table (Upsert)	Primary Key is required (Unique and Non NULL)
-	Dynamic Table (Retract)	Primary Key is NOT required (Key can be NULL)

Changelog Modes in Confluent Cloud

Automatic Dynamic Table creation for the Topics in Kafka:

Topic in Kafka		Dynamic Table in Flink
Uncompacted (delete)	→	Append-Only
Compacted	→	Upsert

Default Changelog Mode for manually created Dynamic Tables:

CREATE TABLE statement		Changelog Mode		Topic in Kafka
Without Primary Key	→	Append-Only	→	Uncompacted (delete)
With Primary Key	→	Upsert	→	Uncompacted (delete)

Lesson 03c

03c: Dynamic Table Creation

Description

Dive into the creation of Dynamic Tables and their parameters (PRIMARY KEY, DISTRIBUTED BY, ALTER, LIKE, ...)

Mapping between Kafka and Flink in Confluent Cloud

Kafka		Flink
Confluent Cloud Environment	→	Catalog
Kafka Cluster	→	Database
Kafka Topic	→	Dynamic Table

Tables are automatically created from Kafka topics.
Kafka topics are automatically created from manually created Tables.

To delete a Table, you have to delete the Kafka Topic (and the associated Schema).

Manual Table Creation (I)

Simplified Flink SQL syntax:

```
CREATE TABLE table_name (
    col1 type,
    col2 type,
    col3 type,
    ...
  );
```

Example:

```
CREATE TABLE orders (
    orderId BIGINT,
    orderUnits BIGINT,
    itemId BIGINT
);
```

Manual Table Creation (II)

More complete Flink SQL syntax:

```
CREATE TABLE  [[catalog_name.] db_name.]table_name (
    col1 type,
    col2 type,
    col3 type,
    … ,
    PRIMARY KEY (col1) NOT ENFORCED
  )
  DISTRIBUTED BY (partition_column_1, partition_column_2, ...) INTO n
BUCKETS
  WITH (param1=value1, param2=value2, ...);
```

- PRIMARY KEY - serves as a hint for optimizations (uniqueness and non-nullability)
- DISTRIBUTED BY - defines the partitioning columns for distributing data (Key of Kafka message)
- WITH - allows you to configure specific table parameters

SQL standard specifies that a constraint can either be `ENFORCED` or `NOT ENFORCED`. This controls if the constraint checks are performed on the incoming/outgoing data. Flink does not own the data therefore the only mode we want to support is the `NOT ENFORCED` mode. It is up to the user to ensure that the query enforces key integrity.

Flink will assume correctness of the primary key by assuming that the columns nullability is aligned with the columns in primary key.

Table Creation - NO PRIMARY KEY

Table creation:

```
CREATE TABLE orders (
    orderId BIGINT,
    orderUnits BIGINT,
    itemId BIGINT
);
```

Inserting data to the table:

```
INSERT INTO orders VALUES (1, 33, 104), (2, 12, 56), (3, 5, 23), (4, 10,
52);
```

View records in topic `orders`:

```
Key:"", Value:{"orderId": 1,"orderUnits": 33,"itemId": 104}
Key:"", Value:{"orderId": 2,"orderUnits": 12,"itemId": 56}
Key:"", Value:{"orderId": 3,"orderUnits": 5,"itemId": 23}
Key:"", Value:{"orderId": 4,"orderUnits": 10,"itemId": 52}
```

Table Creation - PRIMARY KEY (I)

Table creation:

```
CREATE TABLE orders_PK (
    orderId BIGINT,
    orderUnits BIGINT,
    itemId BIGINT,
    PRIMARY KEY (orderId) NOT
ENFORCED );
```

Inserting data to the table:

```
INSERT INTO orders_PK (orderId, orderUnits, itemId) SELECT * FROM orders;
```

View records in topic `orders_PK`:

```
Key:{"orderId":1}, Value:{"orderUnits": 33,"itemId": 104}
Key:{"orderId":2}, Value:{"orderUnits": 12,"itemId": 56}
Key:{"orderId":3}, Value:{"orderUnits": 5,"itemId": 23}
Key:{"orderId":4}, Value:{"orderUnits": 10,"itemId": 52}
```

A primary key constraint is a hint for Flink SQL to leverage for optimizations which specifies that a column or a set of columns in a table or a view are unique and they do not contain null.

A primary key uniquely identifies a row in a table. No columns in a primary key can be nullable.

The `PRIMARY KEY` constraint partitions the table implicitly by the key column. A Kafka message key is defined either by an implicit `DISTRIBUTED BY` clause from a `PRIMARY KEY` constraint or an explicit `DISTRIBUTED BY`.

Table Creation - PRIMARY KEY (II)

Table creation:

```
CREATE TABLE orders_PK_2 (
    orderId BIGINT,
    itemId BIGINT,
    orderUnits BIGINT,
    PRIMARY KEY (orderId, itemId) NOT
ENFORCED );
```

Inserting data to the table:

```
INSERT INTO orders_PK_2 (orderId, orderUnits, itemId) SELECT * FROM orders;
```

View records in topic `orders_PK_2`:

```
Key:{"orderId":1,"itemId":104}, Value:{"orderUnits": 33}
Key:{"orderId":2,"itemId":56}, Value:{"orderUnits": 12}
Key:{"orderId":3,"itemId":23}, Value:{"orderUnits": 5}
Key:{"orderId":4,"itemId":52}, Value:{"orderUnits": 10}
```

Table Creation - DISTRIBUTED BY

Table creation:

```
CREATE TABLE orders_DB (
    orderId BIGINT,
    orderUnits BIGINT,
    itemId BIGINT
) DISTRIBUTED BY (orderId) INTO 4
BUCKETS;
```

`DISTRIBUTED BY column/s INTO # BUCKETS`:

- Buckets the created table by the specified column/s
- Kafka partitions map 1:1 to SQL buckets
- Default: 6 buckets (partitions)

Inserting data to the table:

```
INSERT INTO orders_DB (orderId, orderUnits, itemId) SELECT * FROM orders;
```

View records in topic `orders_DB`:

```
Key:{"orderId":1}, Value:{"orderUnits": 33,"itemId": 104}
Key:{"orderId":2}, Value:{"orderUnits": 12,"itemId": 56}
Key:{"orderId":3}, Value:{"orderUnits": 5,"itemId": 23}
Key:{"orderId":4}, Value:{"orderUnits": 10,"itemId": 52}
```

The automatically created topic `orders_DB` will have 4 partitions.

DISTRIBUTED Clause

- Distributes records into 4 buckets/partitions using this partitioner `hash(orderId) % 4`

```
DISTRIBUTED BY orderId INTO 4 BUCKETS
```

- Distributes records into 6 buckets/partitions using this partitioner `hash(orderId) % 6`

```
DISTRIBUTED BY orderId
```

- Distributes records into 3 buckets/partitions using:
  - Append-Only table (No PK): `round-robin`
  - Retract table (No PK): `hash(entire VALUE) % 3`
  - Upsert table: `hash(PRIMARY KEY) % 3`

```
DISTRIBUTED INTO 3 BUCKETS
```

The following two lines do the exact same thing:

```
DISTRIBUTED BY orderId INTO 4 BUCKETS
```

```
DISTRIBUTED BY HASH(orderId) INTO 4 BUCKETS
```

The `HASH()` function can be omitted. Both lines will apply this partitioner:
`bucket = hash(orderId) % number_of_buckets`

The `hash` algorithm is murmur2.

PRIMARY KEY vs. DISTRIBUTED BY

`PRIMARY KEY` enforces a Non-Null Key

AVRO schema of the Key

```
{
  "fields": [
    {
      "name": "orderId",
      "type": "long"
    }
  ],
  "name": "record",
  "namespace":
"org.apache.flink.avro.generated",
  "type": "record"
}
```

`DISTRIBUTED BY` allows the Key to be Null

AVRO schema of the Key

```
{
  "fields": [
    {
      "default": null,
      "name": "orderId",
      "type": [
        "null",
        "long"
      ]
    }
  ],
  "name": "record",
  "namespace":
"org.apache.flink.avro.generated",
  "type": "record" }
```

Table Creation - WITH (I)

- Used to set the table properties

```
CREATE TABLE orders (
    orderId STRING,
    orderUnits BIGINT,
    itemId BIGINT
) WITH (
  'changelog.mode' = 'retract',
  'kafka.cleanup-policy' = 'compact',
  'value.format' = 'json-registry'
);
```

As of August 2024, these are the configurable properties available in Confluent Cloud:

- changelog.mode
- kafka.cleanup-policy
- kafka.max-message-size
- kafka.retention.size
- kafka.retention.time
- key.fields-prefix
- key.format
- scan.bounded.mode
- scan.bounded.timestamp-millis
- scan.startup.mode
- scan.startup.specific-offsets
- scan.startup.timestamp-millis
- value.fields-include
- value.format

Check for more information about each property in this link: https://docs.confluent.io/cloud/current/flink/reference/statements/create-table.html#with-options

Table Creation - WITH (II)

Table creation:

```
CREATE TABLE orders_PK_all (
    orderId BIGINT,
    itemId BIGINT,
    orderUnits BIGINT,
    PRIMARY KEY (orderId, itemId)
NOT ENFORCED
) WITH (
  'value.fields-include' = 'all'
);
```

`value.fields-include` = [`except-key`, `all`]

Default: `except-key`

Inserting data to the table: `INSERT INTO ...`

View records in topic `orders_PK_all`:

```
Key:{"orderId":1,"itemId":104}, Value:{"orderId": 1,"orderUnits":
33,"itemId": 104}
Key:{"orderId":2,"itemId":56},  Value:{"orderId": 2,"orderUnits":
12,"itemId": 56}
Key:{"orderId":3,"itemId":23},  Value:{"orderId": 3,"orderUnits":
5,"itemId": 23}
Key:{"orderId":4,"itemId":52},  Value:{"orderId": 4,"orderUnits":
10,"itemId": 52}
```

`value.fields-include` = [`all`, `except-key`]

The default is `except-key`.

If `all` is specified, all physical columns of the table schema are included in the value format, which means that key columns appear in the data type for both the key and value format.

Check All Table Properties - SHOW CREATE TABLE

Query:

```
SHOW CREATE TABLE orders_DB;
```

Result:

```
+-----------------------------------------------------+
|                  SHOW CREATE TABLE                  |
+-----------------------------------------------------+
| CREATE TABLE `default`.`kafka_cluster`.`orders_DB` ( |
|   `orderId` BIGINT,                                 |
|   `orderUnits` BIGINT,                              |
|   `itemId` BIGINT                                   |
| ) DISTRIBUTED BY HASH(`orderId`) INTO 4 BUCKETS     |
| WITH (                                              |
|   'changelog.mode' = 'append',                      |
|   'connector' = 'confluent',                        |
|   'kafka.cleanup-policy' = 'delete',                |
|   'kafka.max-message-size' = '2097164 bytes',       |
|   'kafka.retention.size' = '0 bytes',               |
|   'kafka.retention.time' = '7 d',                   |
|   'key.format' = 'avro-registry',                   |
|   'scan.bounded.mode' = 'unbounded',                |
|   'scan.startup.mode' = 'earliest-offset',          |
|   'value.format' = 'avro-registry'                  |
| )                                                   |
+-----------------------------------------------------+
```

Note that the table property values in the `WITH` block are the default values.

'scan.startup.mode' - The default is `earliest-offset`, which differs from the default in open-source Flink, which is `group-offsets`.

ALTER TABLE . . . SET

Modify the Table properties:

```
ALTER TABLE orders_DB SET (
    'changelog.mode' = 'retract',
    'scan.startup.mode' = 'latest-offset'
);
```

Query:

```
SHOW CREATE TABLE
orders_DB;
```

Result:

```
+-----------------------------------------------------+
|                 SHOW CREATE TABLE                   |
+-----------------------------------------------------+
| CREATE TABLE `default`.`kafka_cluster`.`orders_DB` ( |
|   `orderId` BIGINT,                                 |
|   `orderUnits` BIGINT,                              |
|   `itemId` BIGINT                                   |
| ) DISTRIBUTED BY HASH(`orderId`) INTO 4 BUCKETS     |
| WITH (                                              |
|   'changelog.mode' = 'retract',                     |
|   'connector' = 'confluent',                        |
|   'kafka.cleanup-policy' = 'delete',                |
|   'kafka.max-message-size' = '2097164 bytes',       |
|   'kafka.retention.size' = '0 bytes',               |
|   'kafka.retention.time' = '7 d',                   |
|   'key.format' = 'avro-registry',                   |
|   'scan.bounded.mode' = 'unbounded',                |
|   'scan.startup.mode' = 'latest-offset',            |
|   'value.format' = 'avro-registry'                  |
| )                                                   |
+-----------------------------------------------------+
```

LIKE

Create a table based on the definition of an existing table

Examples:

- Changing the value format to Protobuf

```
CREATE TABLE orders_protobuf
  WITH (
    'value.format' = 'proto-registry'
  )
  LIKE orders;
```

- Applying a custom watermark strategy

```
CREATE TABLE orders_new_watermark (
    WATERMARK FOR `$rowtime` AS `$rowtime` -
INTERVAL '5' SECOND
  )
  LIKE orders (
    EXCLUDING WATERMARKS
);
```


03d: Column Types

Description

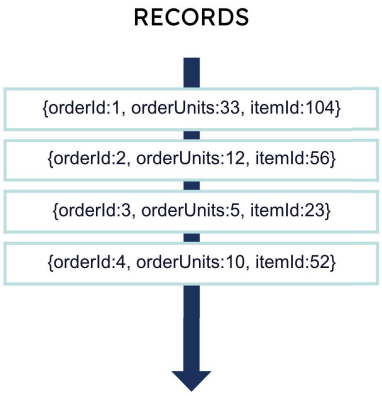
Definition of the three different types of columns available in Flink SQL.

Column Types in Dynamic Tables

- Physical columns (regular columns)
- Metadata columns
  - System columns (only Confluent Cloud)
- Computed columns

Physical Columns

- Are regular columns
- Define the physical record (name, type and order of fields)

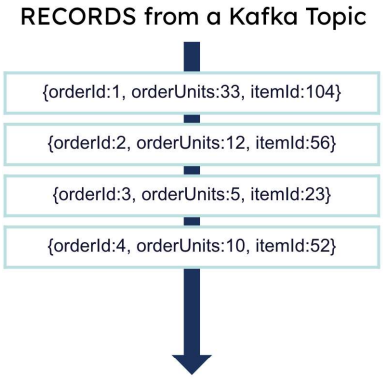



```
CREATE TABLE orders (
    orderId BIGINT,
    orderUnits BIGINT,
    itemId BIGINT
);
```

Physical columns

Metadata Columns

- Represent additional information of the actual data
- Are indicated by the `METADATA` keyword
- Metadata fields are **readable** or **readable/writable**
  - **Read-only** columns must be declared **VIRTUAL**




```
CREATE TABLE orders (
    orderId BIGINT,
    orderUnits BIGINT,
    itemId BIGINT,
    `partition` INT METADATA VIRTUAL
);
```

orderId	orderUnits	itemId	partition
1	33	104	3
2	12	56	0
3	5	23	2
4	10	52	3

Metadata columns are not part of the actual data being processed but are instead used for internal purposes, such as tracking event time, processing time, or watermark information in event streams.

You can declare a writable column VIRTUAL, but it won't be writable.

Metadata Columns - Kafka Specific

Kafka records provide the following METADATA columns:

`METADATA` column	Operation	Type
headers	Read/Write	MAP
leader-epoch	Read	INT
offset	Read	BIGINT
partition	Read	INT
timestamp	Read/Write	TIMESTAMP_LTZ(3)
timestamp-type	Read	STRING
topic	Read	STRING

Activity: Access to the `headers` of the records

Suppose you have a Kafka topic `sales` with data from an external system

Confluent Cloud will automatically create a Dynamic Table in Flink called `sales` from the topic with this structure:

```
+--------------+----------------+----------+--------+
| Column Name  |   Data Type    | Nullable | Extras |
+--------------+----------------+----------+--------+
| sale_id      | BIGINT         | NULL     |        |
| product_name | STRING         | NULL     |        |
| sale_amount  | DECIMAL(10, 2) | NULL     |        |
+--------------+----------------+----------+--------+
```

However, the `headers` of the records contain relevant information to process the table `sales`

- What would you do to access the `headers` to process the table `sales`?

Instructor note: For now, either let students work on their own for a few minutes or work in groups for a few minutes and then discuss as a class.

Example input record and corresponding output records:

Activity: Adding a new METADATA column

Using `ALTER TABLE … ADD` to the table `sales` to add a new METADATA column `headers`

```
ALTER TABLE `sales` ADD (
  `headers` MAP<STRING, BYTES> METADATA
);
```

If you describe now the tables `sales`:

```
+--------------+--------------------+----------+----------+
| Column Name  |     Data Type      | Nullable |  Extras  |
+--------------+--------------------+----------+----------+
| sale_id      | BIGINT             | NULL     |          |
| product_name | STRING             | NULL     |          |
| sale_amount  | DECIMAL(10, 2)     | NULL     |          |
| headers      | MAP<STRING, BYTES> | NULL     | METADATA |
+--------------+--------------------+----------+----------+
```


System Columns (Metadata Columns)

- Only in Confluent Cloud
- Currently, there is only one SYSTEM column `$rowtime`
  - It's included in all Dynamic Tables as METADATA VIRTUAL columns

`SYSTEM` column	Operation	Type
$rowtime	Read	TIMESTAMP_LTZ(3) NOT NULL

You can use the `$rowtime` system column to get the timestamp from a Kafka record, because `$rowtime` is exactly the Kafka record timestamp.

`$rowtime` takes the internal timestamp of the Kafka message. `$rowtime` column is created by Confluent in order to avoid the `ALTER TABLE` for including the timestamp in the automatic created table, so it's readily available (for read only).

If you alter the table to include the `timestamp METADATA`, you have the possibility to change the timestamp for the resulting Kafka message.

System Columns - DESCRIBE EXTENDED

If you DESCRIBE a table, SYSTEM columns are not shown:

```
$ DESCRIBE orders;

+-------------+-----------+----------+--------+
| Column Name | Data Type | Nullable | Extras |
+-------------+-----------+----------+--------+
| orderId     | BIGINT    | NULL     |        |
| orderUnits  | BIGINT    | NULL     |        |
| itemId      | BIGINT    | NULL     |        |
+-------------+-----------+----------+--------+
```

To view the SYSTEM columns, use DESCRIBE EXTENDED:

```
$ DESCRIBE EXTENDED orders;

+-------------+---------------------------+----------+-------------------------------------------------------+---------+
| Column Name |         Data Type         | Nullable |                        Extras                         | Comment |
+-------------+---------------------------+----------+-------------------------------------------------------+---------+
| orderId     | BIGINT                    | NULL     |                                                       |         |
| orderUnits  | BIGINT                    | NULL     |                                                       |         |
| itemId      | BIGINT                    | NULL     |                                                       |         |
| $rowtime    | TIMESTAMP_LTZ(3) *ROWTIME* | NOT NULL | METADATA VIRTUAL, WATERMARK AS `SOURCE_WATERMARK`() | SYSTEM  |
+-------------+---------------------------+----------+-------------------------------------------------------+---------+
```

Computed Columns

- Generated columns using the syntax:

```
`column_name` AS `computed_column_expression`
```

Example:

```
CREATE TABLE MyTable (
  user_id BIGINT,
  price DOUBLE,
  quantity DOUBLE,
  cost AS price * quantity
);
```

Data types are not declared, automatically inferred from the expression

---

Computed columns are virtual columns that are generated using the syntax `column_name AS computed_column_expression`.

A computed column evaluates an expression that can reference other columns declared in the same table. Both physical columns and metadata columns can be accessed. The column itself is not physically stored within the table. The column's data type is derived automatically from the given expression and does not have to be declared manually.

Lesson 03e

03e: Stateless & Stateful Operators

Description

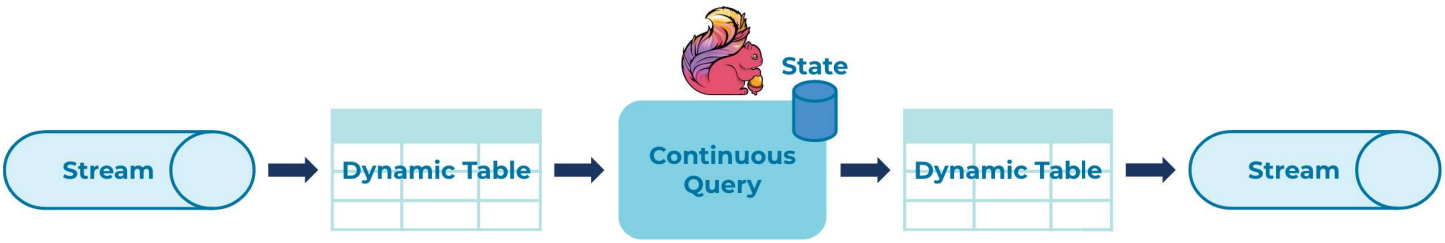
Differences between Stateless and Stateful Operators

Stateless vs. Stateful Operators

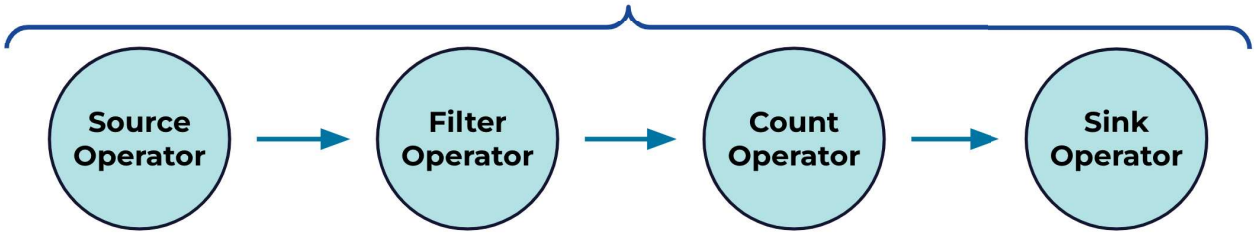
- **Stateless Operator**
  Records are independently processed without retaining any state

- **Stateful Operator**
  Maintains state information across records for processing

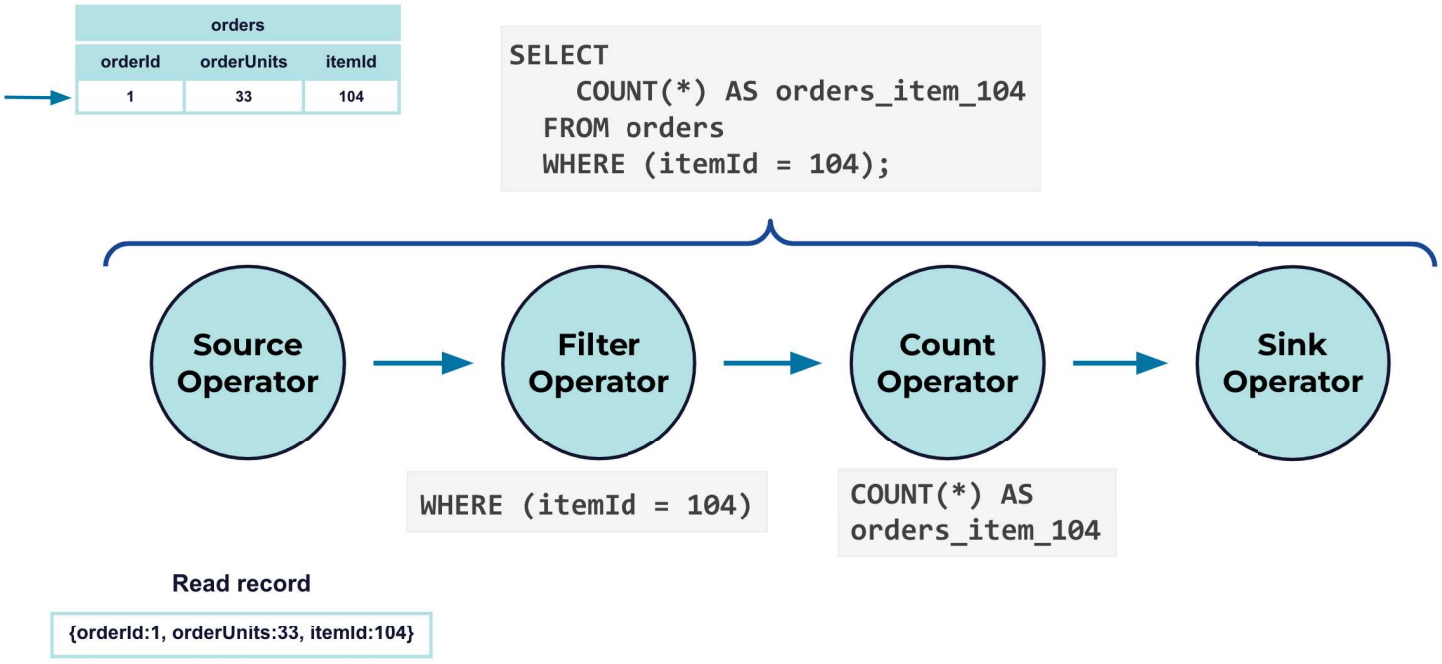
Stateless & Stateful Example (1)



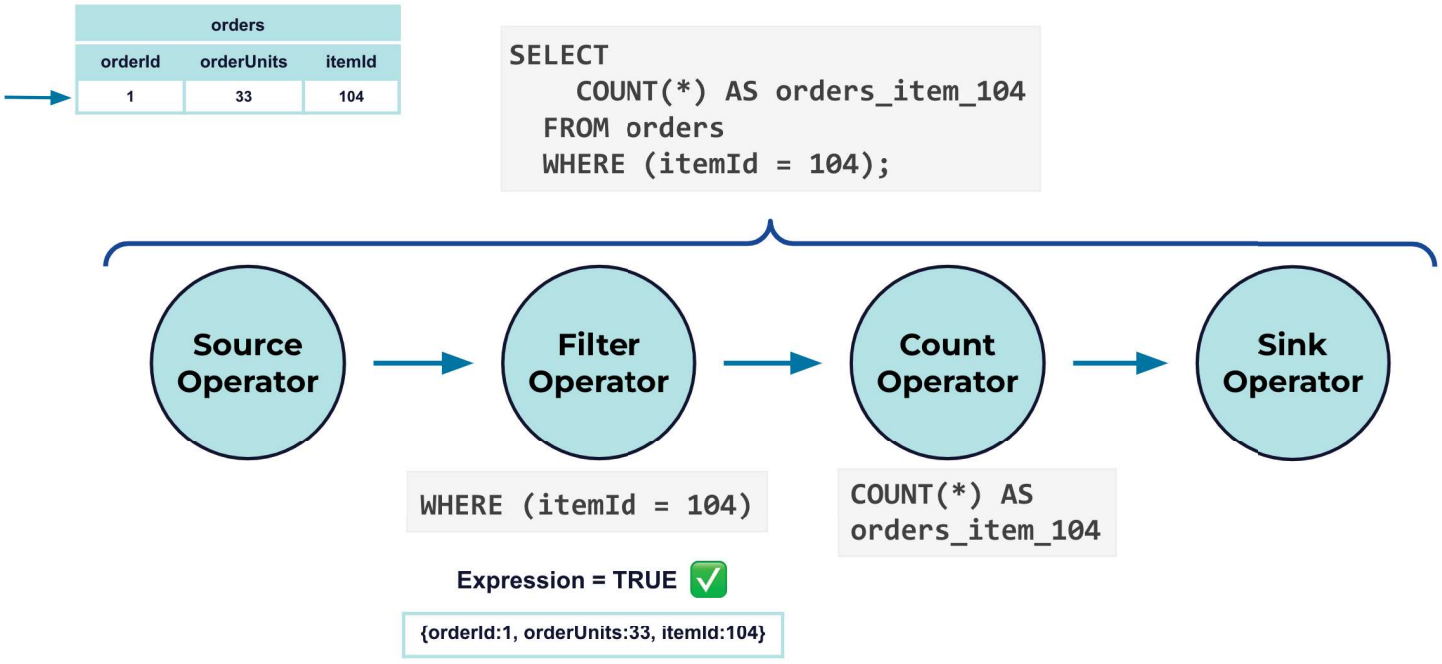

```
SELECT
    COUNT(*) AS orders_item_104
  FROM orders
  WHERE (itemId = 104);
```



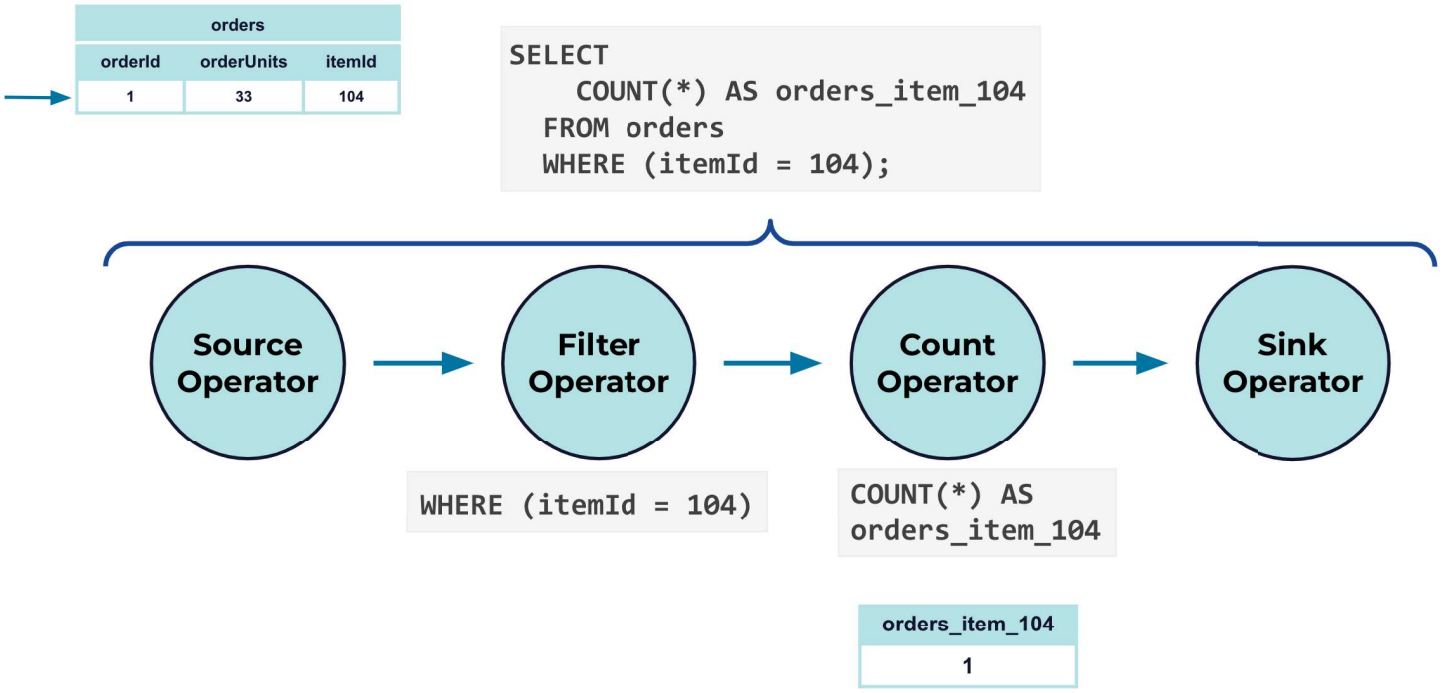
Stateless & Stateful Example (2)



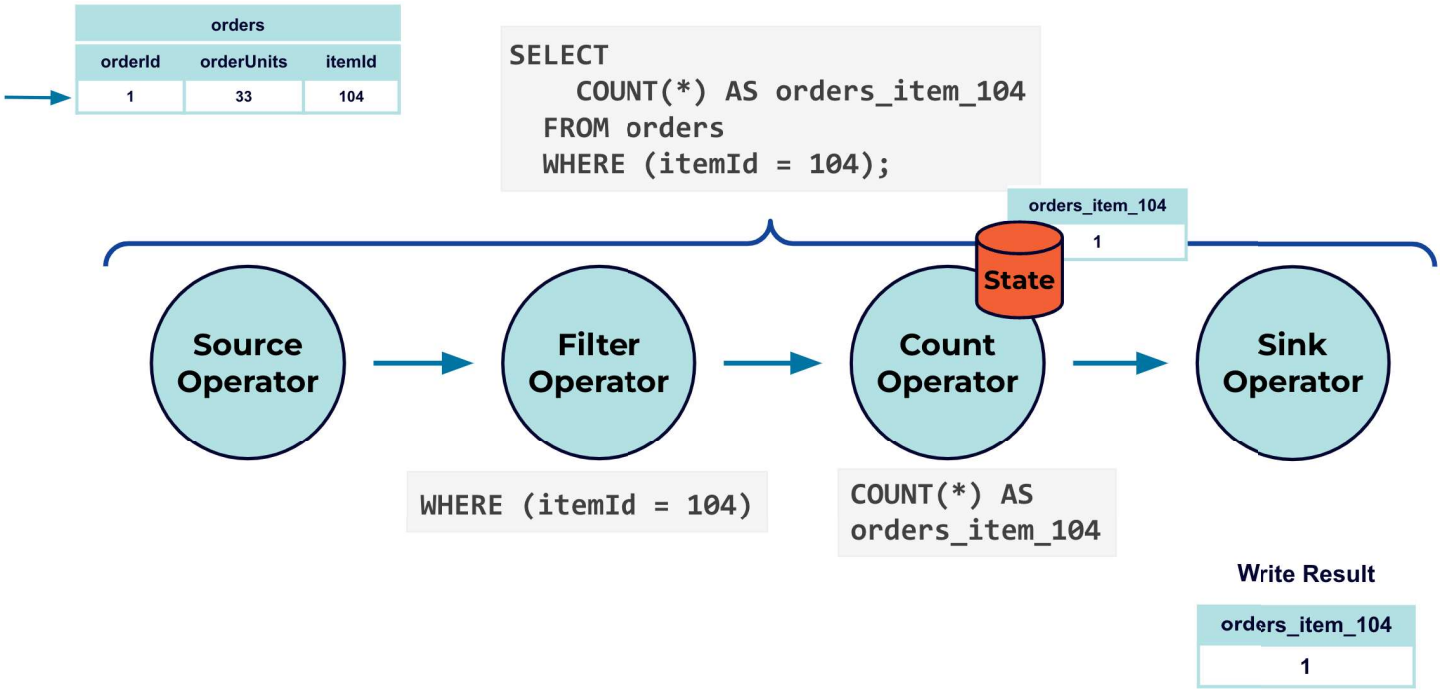
Stateless & Stateful Example (3)



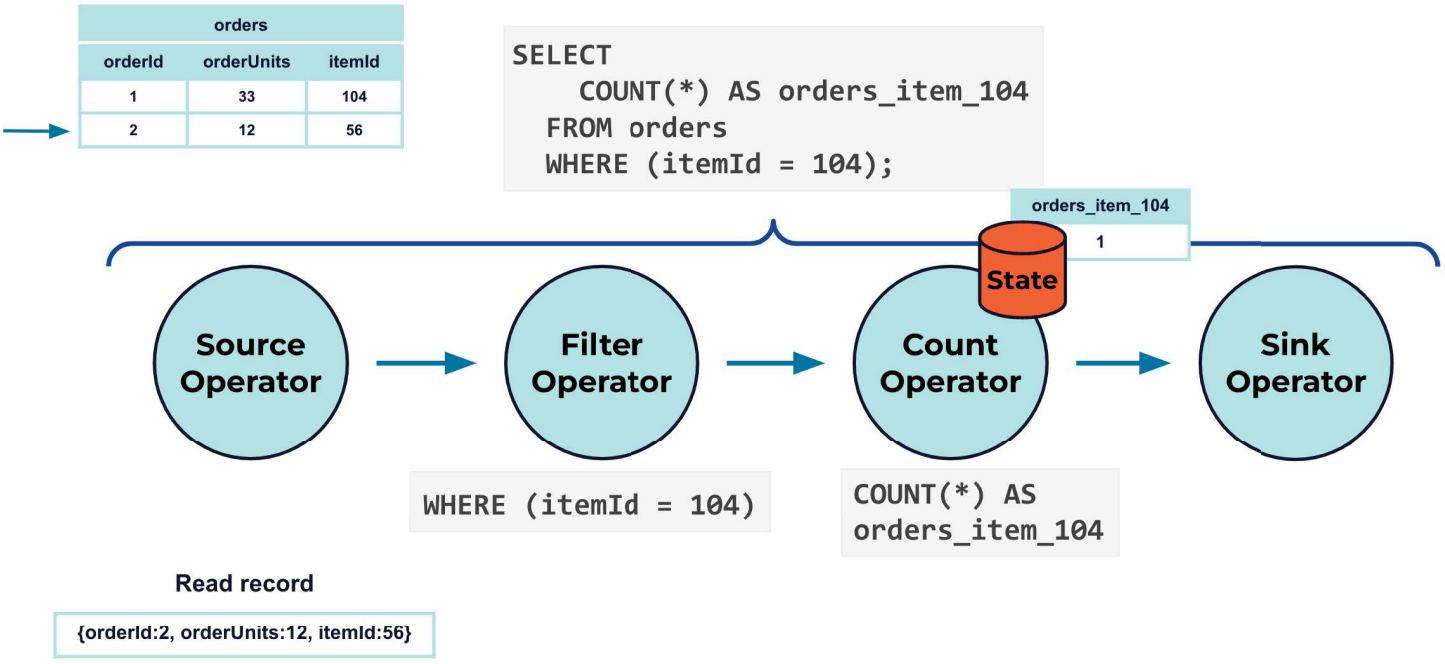
Stateless & Stateful Example (4)



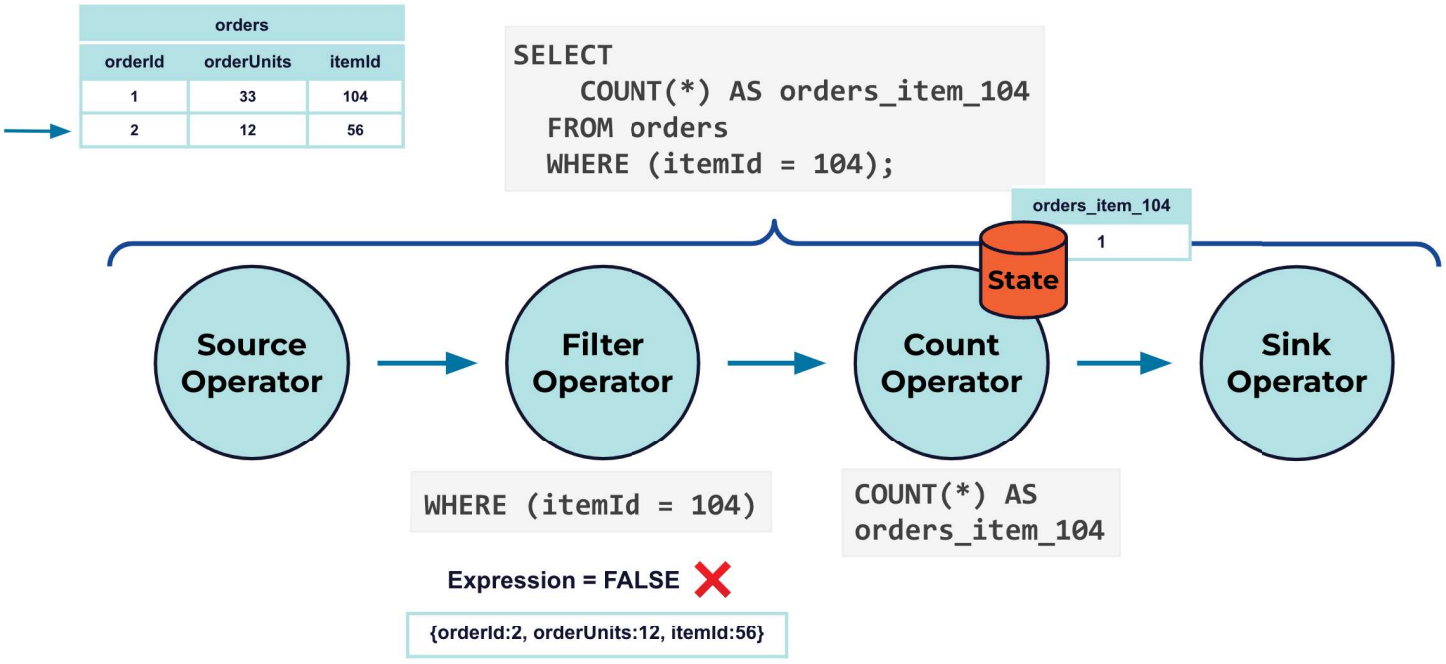
Stateless & Stateful Example (5)



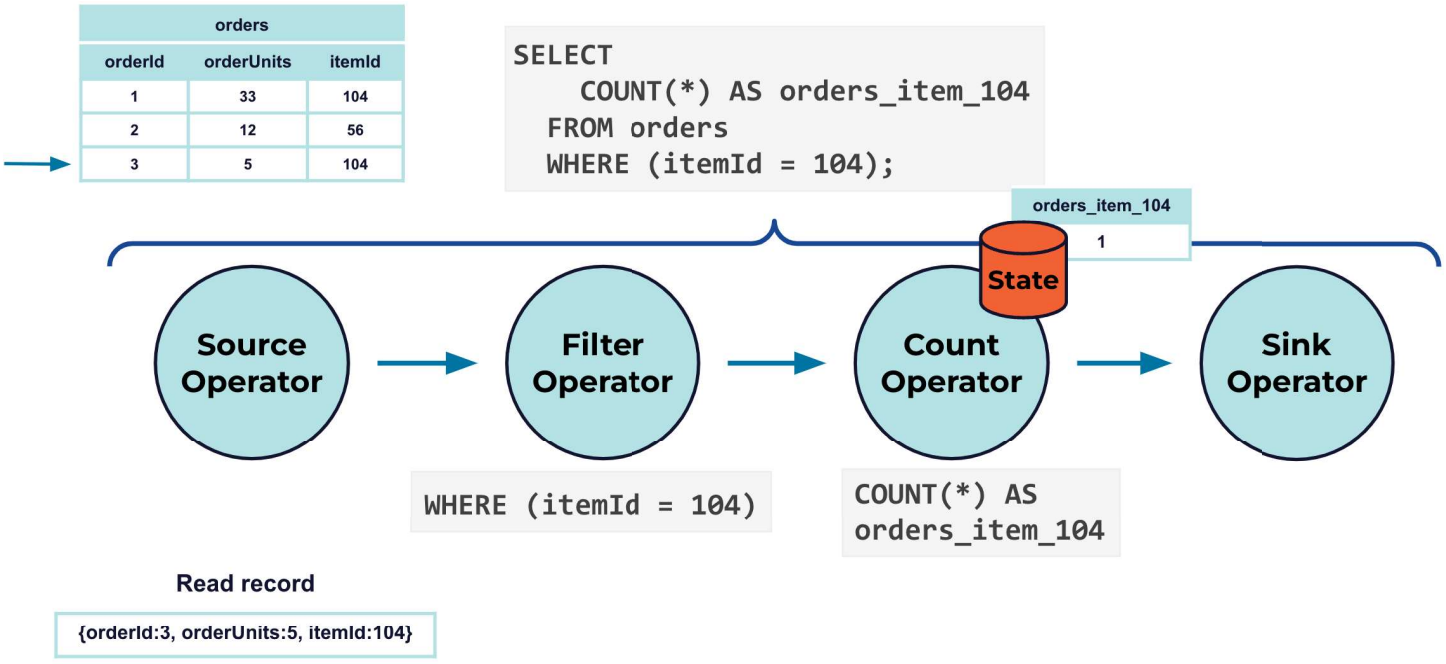
Stateless & Stateful Example (6)



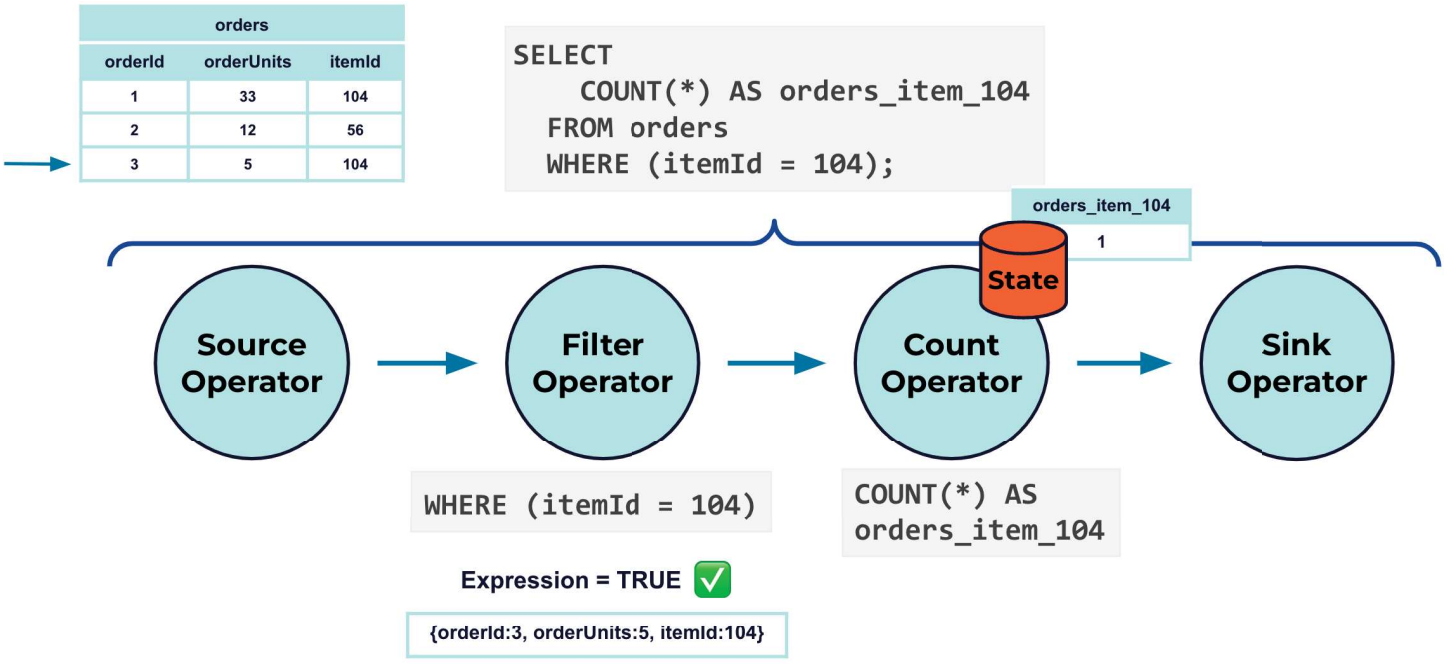
Stateless & Stateful Example (7)



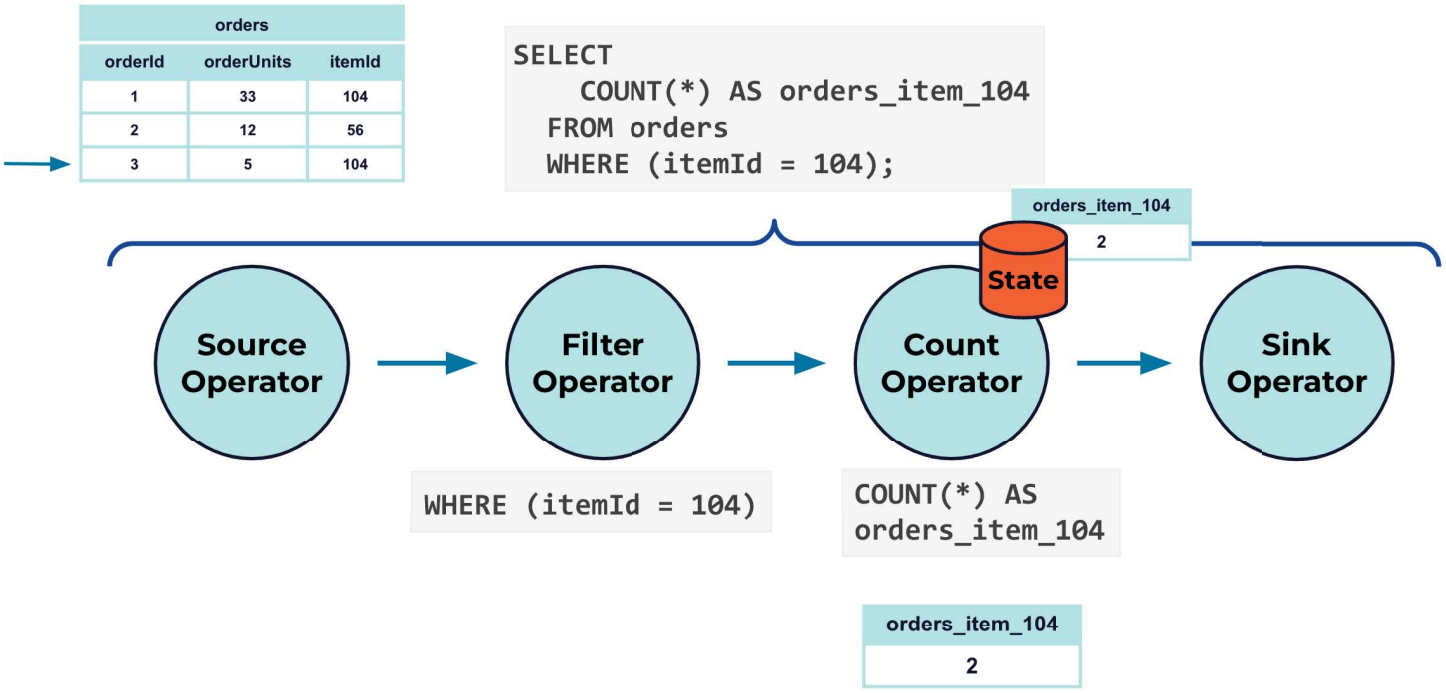
Stateless & Stateful Example (8)



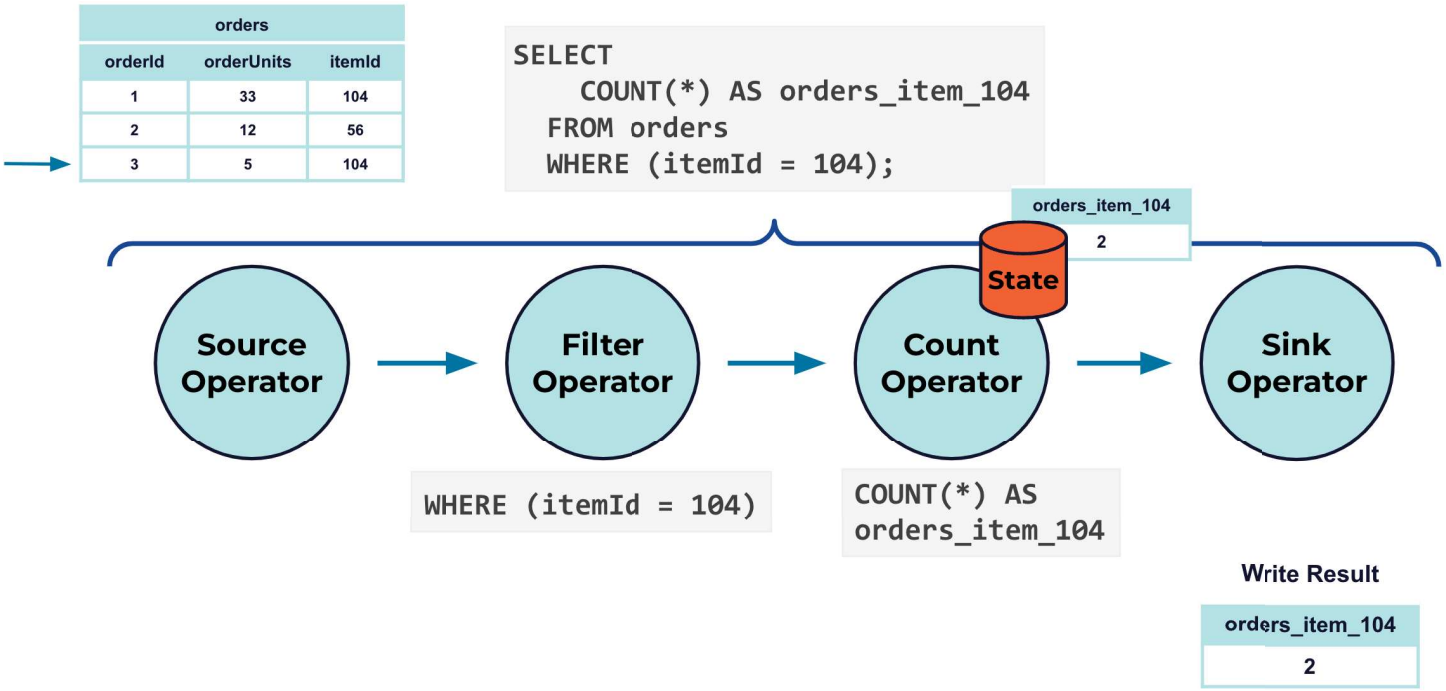
Stateless & Stateful Example (9)



Stateless & Stateful Example (10)

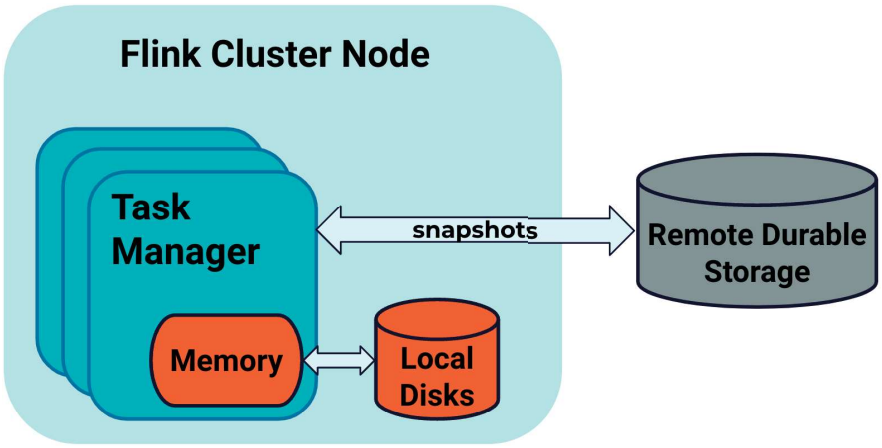


Stateless & Stateful Example (11)



State Store

- Local
- Fast
- Fault-Tolerant



To best understand state and state backends in Flink, it's important to distinguish between in-flight state and state snapshots. In-flight state, also known as working state, is the state a Flink job is working on. It is always stored locally in memory (with the possibility to spill to disk) and can be lost when jobs fail without impacting job recoverability. State snapshots, i.e., checkpoints and savepoints, are stored in a remote durable storage, and are used to restore the local state in the case of job failures. The appropriate state backend for a production deployment depends on scalability, throughput, and latency requirements.

In order to achieve high performance, Flink keeps its state local to each processing node. And in order to provide fault tolerance, Flink periodically checkpoints this state by copying it to a remote, durable object store, like S3.

Flink is able to provide fault-tolerant, exactly-once semantics through a combination of state snapshots and stream replay. These snapshots capture the entire state of the distributed pipeline, recording offsets into the input queues as well as the state throughout the job graph that has resulted from having ingested the data up to that point. When a failure occurs, the sources are rewound, the state is restored, and processing is resumed. As depicted above, these state snapshots are captured asynchronously, without impeding the ongoing processing.

Lab Module 03: Working with Dynamic Tables

Please work on **Lab Module 03: Working with Dynamic Tables**.

Refer to the Exercise Guide.

