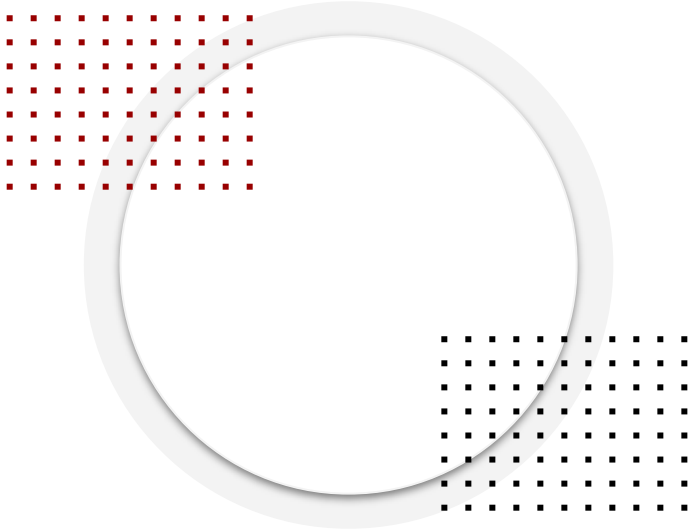
Table & SQL API in Apache Flink

Real-Time Data Engineering Training

About Instructor

- About Instructor ... *<text size should be 16 and style should be Trebuchet MS>*



NAME

Datacouch Instructor

DATACOUCH
CREATING INFINITE POSSIBILITIES

DSAI
Data Science and AI Academy

Piramal Finance

AGENDA

- Table API vs. SQL API vs. DataStream API
- Defining tables over Kafka, filesystem, CDC sources
- Stream-table duality: Dynamic tables & temporal joins

Introduction to Flink's APIs

- Apache Flink provides three main APIs for building real-time data processing applications.
- DataStream API: For low-level control and stateful stream processing.
- Table API: A language-integrated query API for relational stream and batch processing.
- SQL API: A standard SQL interface for querying and processing data.

The DataStream API

- The foundational API of Flink, offering the most control.
- **Best for:** Complex event processing, stateful applications, and fine-grained control over time and state.
- **Example (Java) :**

```
DataStream<String> text = ...;
DataStream<WordWithCount> windowCounts = text
    .flatMap(new FlatMapFunction<String, WordWithCount>() { ... })
    .keyBy("word")
    .timeWindow(Time.seconds(5), Time.seconds(1))
    .reduce(new ReduceFunction<WordWithCount>() { ... });
```

The Table API

- A declarative, fluent API that enables relational queries on streaming data.
- Can be seamlessly integrated with the DataStream API.
- Best for: Relational operations like selection, projection, aggregation, and joins.
- Example (Java):

```
Table table = tableEnv.fromDataStream(dataStream);
Table result = table.where("amount > 2")
                    .groupBy("id")
                    .select("id, SUM(amount)");
```

The SQL API

- The most abstract API, allowing you to use standard SQL to query streaming data.
- Reduces the amount of code you need to write.
- Best for: Data analysts and developers familiar with SQL who want to perform real-time data analysis.
- Example (SQL):

```sql
SELECT id, SUM(amount)
FROM Orders
WHERE amount > 2
GROUP BY id;
```

API Comparison

Feature	DataStream API	Table API	SQL API
Abstraction Level	Low	High	Highest
Control	High	Medium	Low
Ease of Use	Low	Medium	High
Expressiveness	High	Medium	**Limited to SQL**

When to Use Which API?

DataStream API

When you need to implement complex, stateful logic that isn't easily expressed in SQL.

Table API & SQL API

For most common data processing tasks like filtering, aggregation, and joining. These are often easier and more concise.

Interoperability

- Flink allows you to seamlessly switch between all three APIs.
- You can convert a DataStream to a Table and vice-versa.
- This allows you to use the right tool for the right job within the same application.

AGENDA

- Table API vs. SQL API vs. DataStream API
- Defining tables over Kafka, filesystem, CDC sources
- Stream-table duality: Dynamic tables & temporal joins

Connectors in Flink

Flink uses connectors to read from and write to external systems.

There are connectors for a wide variety of systems, including:

- Apache Kafka
- File Systems (CSV, JSON, Avro)
- Databases (JDBC)
- Change Data Capture (CDC) sources

Defining a Table over Kafka

You can define a table over a Kafka topic using Flink's SQL DDL.

- Example (SQL):

```
CREATE TABLE KafkaTable (
  `user_id` BIGINT,
  `item_id` BIGINT,
  `behavior` STRING,
  `ts` TIMESTAMP(3) METADATA FROM 'timestamp'
) WITH (
  'connector' = 'kafka',
  'topic' = 'user_behavior',
  'properties.bootstrap.servers' = 'localhost:9092',
  'format' = 'json'
);
```

Defining a Table over a Filesystem

Similarly, you can define a table over files in a filesystem.

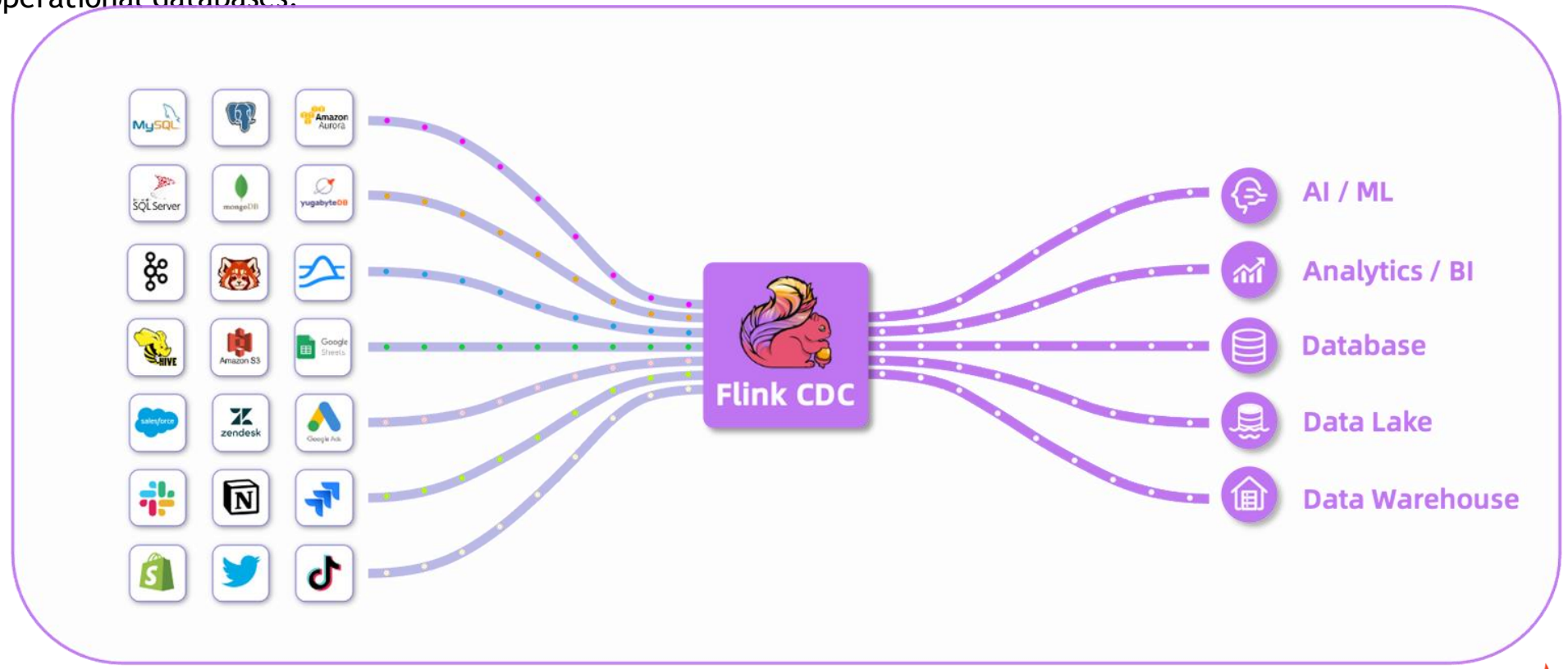
- Example (SQL):

```
CREATE TABLE MyTable (
  `user` BIGINT,
  `product` STRING,
  `amount` INT
) WITH (
  'connector' = 'filesystem',
  'path' = '/path/to/your/data',
  'format' = 'csv'
);
```

Change Data Capture (CDC)

CDC allows you to capture row-level changes in a database and stream them to Flink.

This is a powerful way to build real-time applications that react to changes in your operational databases.



Defining a Table over a CDC Source

Flink has CDC connectors for databases like MySQL, PostgreSQL, and Debezium.

Example (SQL for MySQL):

```
    id INT PRIMARY KEY,
    name STRING,
    email STRING
) WITH (
    'connector' = 'mysql-cdc',
    'hostname' = 'localhost',
    'port' = '3306',
    'username' = 'user',
    'password' = 'password',
    'database-name' = 'my_db',
    'table-name' = 'users'
);
```

AGENDA

- Table API vs. SQL API vs. DataStream API
- Defining tables over Kafka, filesystem, CDC sources
- Stream-table duality: Dynamic tables & temporal joins

What is Stream-Table Duality?

This is the concept that a stream can be converted into a table, and a table can be converted back into a stream.

- A stream is an unbounded sequence of events.
- A table is a snapshot of the stream at a particular point in time.

Dynamic Tables

Dynamic tables are the core concept of Flink's Table API and SQL support for streaming data.

In contrast to the static tables that represent batch data, dynamic tables change over time.

But just like static batch tables, systems can execute queries over dynamic tables.

Querying dynamic tables yields a ***Continuous Query***.

The following figure visualizes the relationship of streams, dynamic tables, and continuous queries:

Continuous Queries

A continuous query is evaluated on a dynamic table and produces a new dynamic table as a result. In contrast to a batch query, a continuous query never terminates and updates its result table according to its input tables' updates.

At any point in time, a continuous query is semantically equivalent to the result of the same query executed in batch mode on a snapshot of the input tables.

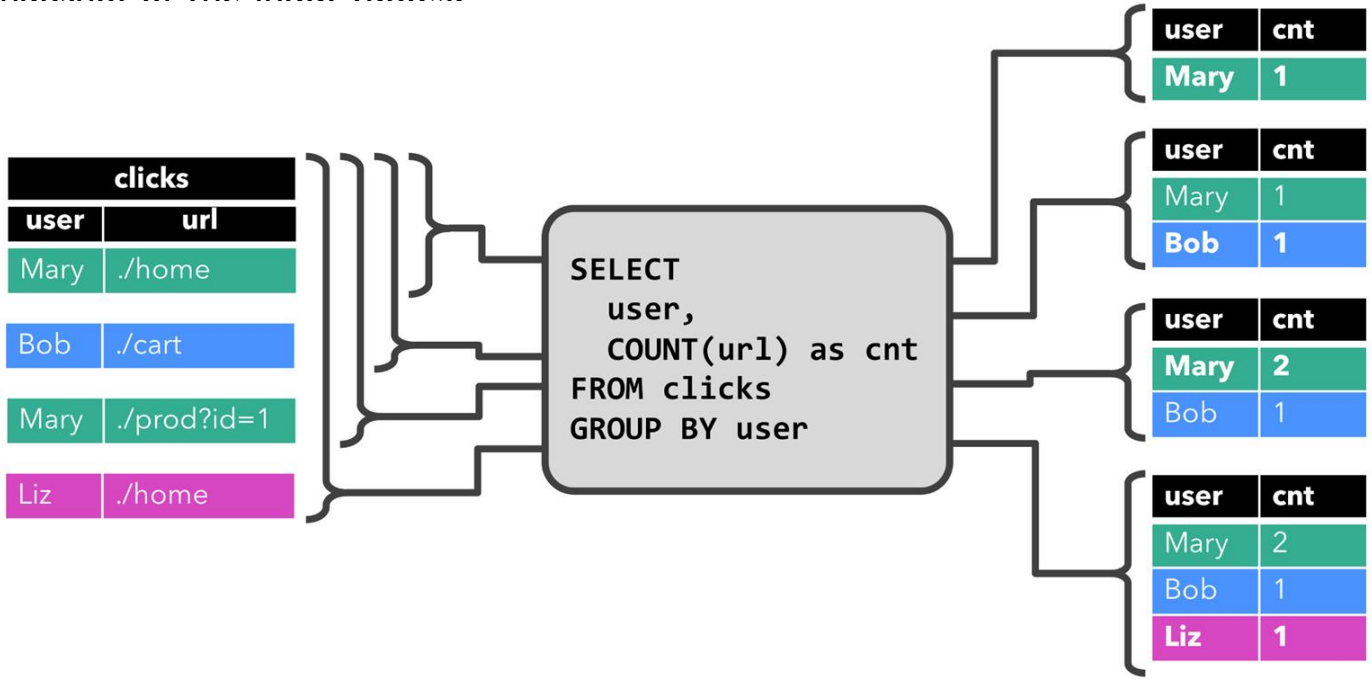


Table to Stream Conversion

A dynamic table can be continuously modified by INSERT, UPDATE, and DELETE changes just like a regular database table. It might be a table with a single row, which is constantly updated, an insert-only table without UPDATE and DELETE modifications, or anything in between.

When converting a dynamic table into a stream or writing it to an external system, these changes need to be encoded. Flink's Table API and SQL support three ways to encode the changes of a dynamic table:

- Append-only stream
- Retract stream
- Upsert stream

Table to Stream Conversion (continued)

Append-only stream: A dynamic table that is only modified by INSERT changes can be converted into a stream by emitting the inserted rows.

Retract stream: A retract stream is a stream with two types of messages, add messages and retract messages. A dynamic table is converted into a retract stream by encoding an INSERT change as add message, a DELETE change as a retract message, and an UPDATE change as a retract message for the updated (previous) row, and an additional message for the updating (new) row.

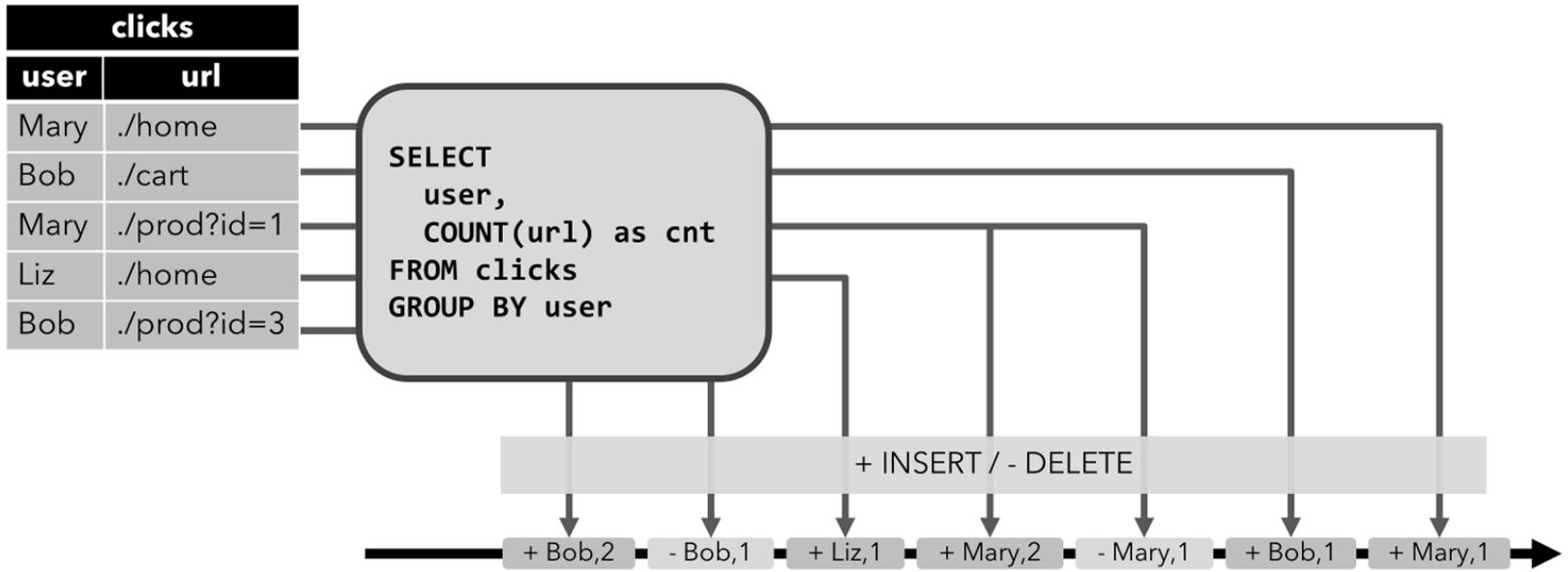
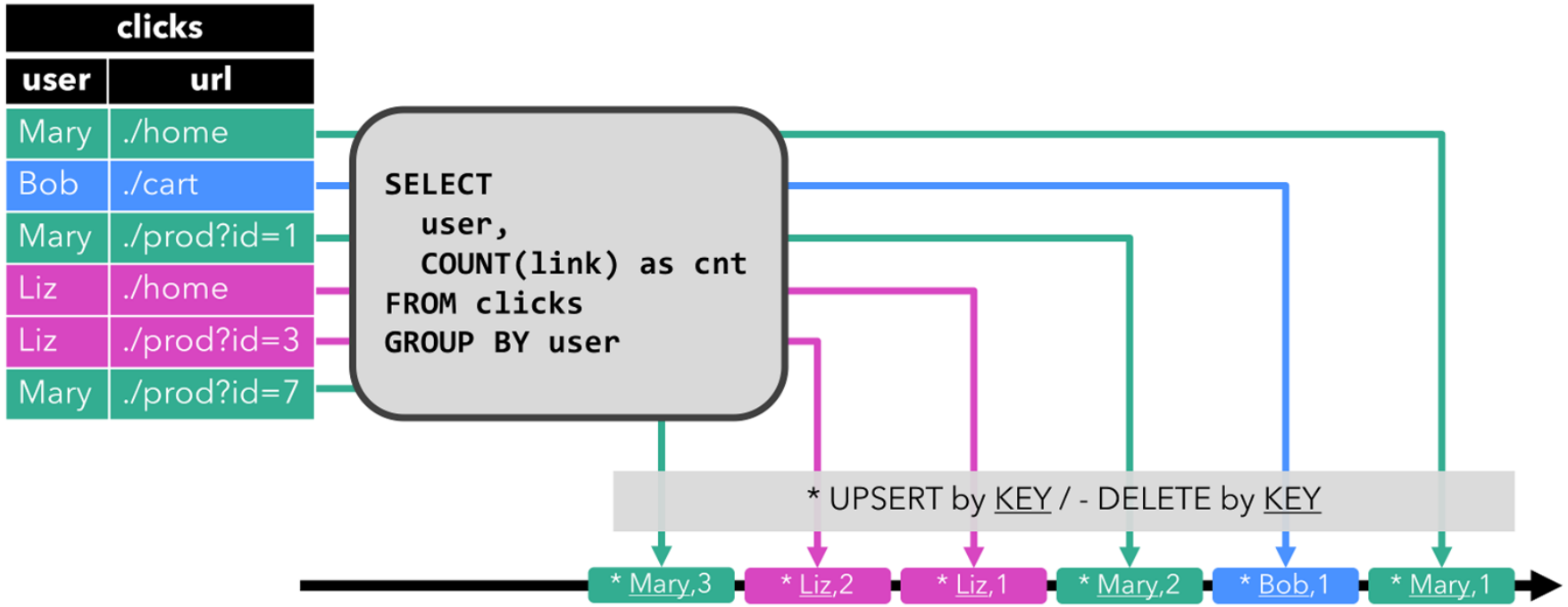


Table to Stream Conversion (continued)

Upsert stream: An upsert stream is a stream with two types of messages, upsert messages and delete messages. A dynamic table that is converted into an upsert stream requires a (possibly composite) unique key.

A dynamic table with a unique key is transformed into a stream by encoding INSERT and UPDATE changes as upsert messages and DELETE changes as delete messages.

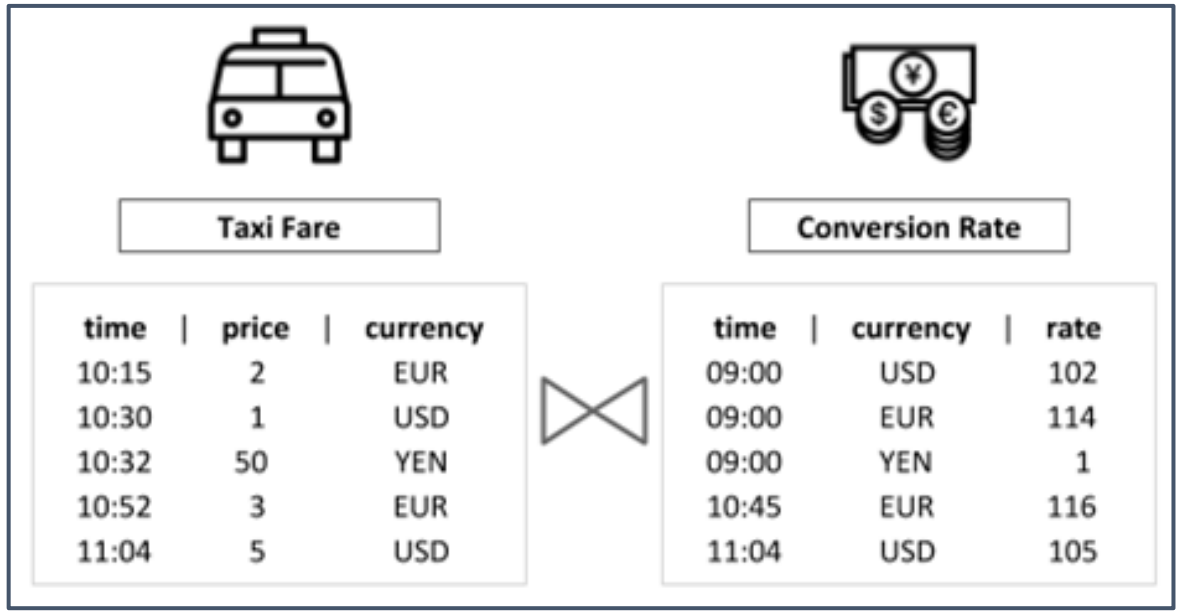


The Challenge of Temporal Data

How do we enrich streaming data against a reference dataset that is constantly changing? We need to perform a "point-in-time" lookup for every single event.

A Classic Example: **Currency Conversion**

- Imagine a real-time stream of taxi fares from around the world.
- We want to convert each fare into a single currency, like USD, for analysis.
- The problem is that currency exchange rates fluctuate throughout the day.
- To get an accurate result, each taxi fare event must be joined with the specific exchange rate that was valid at the exact moment the ride happened.



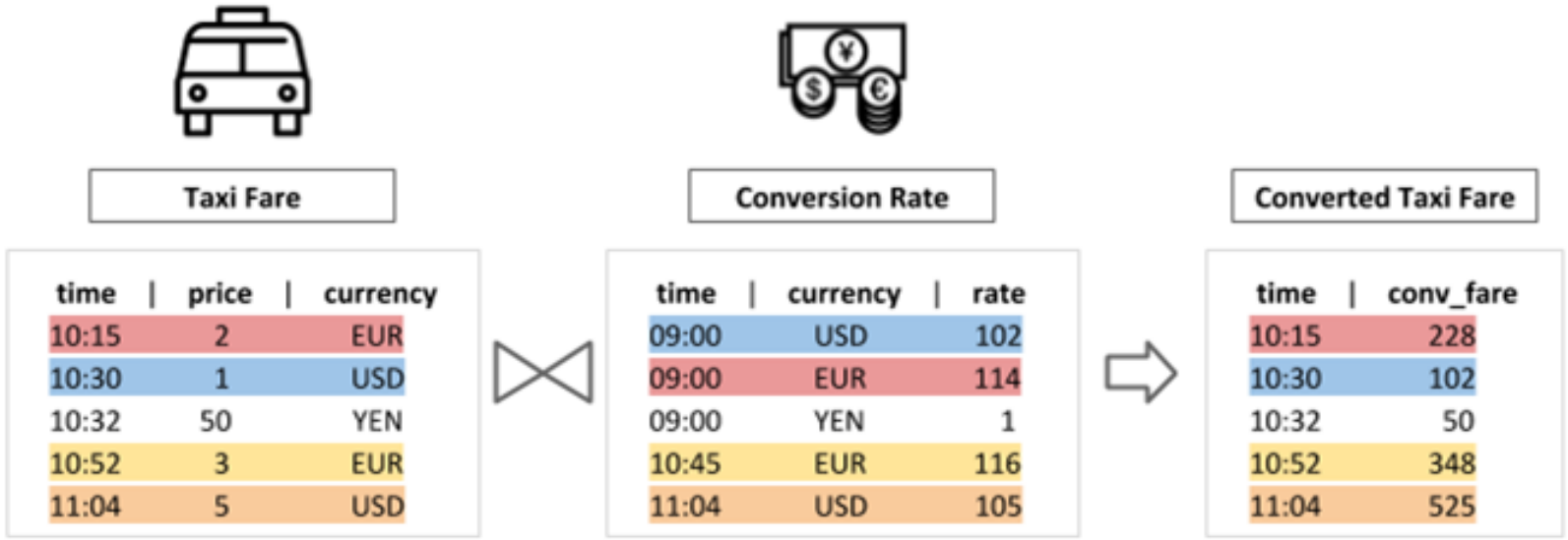
Flink's Solution: Temporal Tables

Flink solves this with Temporal Tables, which are special, versioned views on a stream of changes.

How do they work? A regular stream (like currency rate updates) is turned into a temporal table by defining two key things:

- A Primary Key to identify the entity (e.g., r_currency).
- A Versioning Field (a timestamp) to track changes over time (e.g., r_proctime).

Creating a Temporal Table Function: You can create this view from a regular table. This function effectively says, "Give me the state of this table at a specific point in time"



Temporal Joins in Streaming SQL

The real power comes from using a Temporal Join to combine a regular stream with a temporal table.

- For each record from the main stream (e.g., TaxiFares), Flink automatically looks up the correct version of the record from the temporal table (Rates) using the main stream's timestamp.

Why is this better than a regular windowed join?

- A windowed join would eventually drop old rate information from its state.
- A temporal join keeps all historical versions of the rates available, ensuring that even very old TaxiFare events can be correctly matched with the historical rate.

Conversion Rate

time	currency	rate
09:00	USD	102
09:00	EUR	114
09:00	YEN	1
10:45	EUR	116
11:04	USD	105

Rates('11:00')

time	currency	rate
09:00	USD	102
09:00	YEN	1
10:45	EUR	116

Temporal Join in Action & Key Benefits

A temporal join allows you to join a stream with a temporal table.

This allows you to enrich your streaming data with historical context.

Example: Joining a stream of orders with a table of currency exchange rates that changes over time.

With the Rates function registered, the complex point-in-time lookup becomes a simple and elegant SQL query.

Regular Join	Temporal Table Join

`SELECT SUM(tf.price * rh.rate) AS conv_fare`
`FROM taxiFare AS tf,`
`ratesHistory AS rh`
`WHERE tf.currency = rh.currency`
`AND r.time = (SELECT MAX(rh2.time)`
`FROM ratesHistory AS rh2`
`WHERE rh2.currency = tf.currency`
`AND rh2.time <= tf.time);`
	`SELECT tf.time,`
	`tf.price * rh.rate AS conv_fare`
	`FROM taxiFare AS tf`
	`LATERAL TABLE(Rates(tf.time)) AS rh`
	`WHERE tf.currency = rh.currency;`


THANK YOU