CDC Stream Processing with Apache Flink®
A peek under the hood of a changelog engine

Timo Walther, Principal Software Engineer

FOSDEM 2023, 2023-02-04
About me

Open source
• Long-term committer since 2014 (before ASF)
• Member of the project management committee (PMC)
• Top 5 contributor (commits), top 1 contributor (additions)
• Among core architects of Flink SQL

Career
• Early Software Engineer @ DataArtisans (acquired by Alibaba)
• SDK Team, SQL Team Lead @ Ververica
• Co-Founder @ Immerok (acquired by Confluent)
• Principal Software Engineer @ Confluent
What is Apache Flink?
Building Blocks for Stream Processing

Streams
- Pipeline
- Distribute
- Join
- Enrich
- Control
- Replay

Time
- Synchronize
- Progress
- Wait
- Timeout
- Fast-forward
- Replay

State
- Store
- Buffer
- Cache
- Model
- Grow
- Expire

Snapshots
- Backup
- Version
- Fork
- A/B test
- Time-travel
- Restore
What makes Apache Flink unique?

- Source 1
- Source 2
- Normalize
- Join
- Sink

- Subtask 1
- Subtask 2
- Filter
- Shard 1
- Shard 2
- Subtask 1
- Subtask 2
- Partition 1
- Partition 2

- Fast local state that scales with the operator
- Long-term durable storage
What is Apache Flink used for?

- Transactions
- Logs
- IoT
- Events
- Interactions

...
Apache Flink’s APIs
API Stack

DataStream API

Table / SQL API

Optimizer / Planner

Stateful Functions

Low-Level Stream Operator API

Dataflow Runtime
DataStream API

```java
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

env.setRuntimeMode(STREAMING);

DataStream<Integer> stream = env.fromElements(1, 2, 3);

stream.executeAndCollect().forEachRemaining(System.out::println);
```

### Properties

- **Exposes** the building blocks for stream processing
- Arbitrary operator **topologies** using map(), process(), connect(), ...
- Business logic is written in **user-defined functions**
- Arbitrary **user-defined record** types flow in-between
- Conceptually always an **append-only / insert-only** log!

Output

```
1
2
3
```
Table / SQL API

TableEnvironment env = TableEnvironment.create(EnvironmentSettings.inStreamingMode());

// Programmatic
Table table = env.fromValues(row(1), row(2), row(3));

// SQL
Table table = env.sqlQuery("SELECT * FROM (VALUES (1), (2), (3))");

table.execute().print();

Properties

• Abstracts the building blocks for stream processing
• Operator topology is determined by planner
• Business logic is declared in SQL and/or Table API
• Internal record types flow, Flink’s Row type is exposed in Table API
• Conceptually a table, but a changelog under the hood!
DataStream API ➔ Table / SQL API

Mix and match APIs!

```java
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);

// Stream -> Table
DataStream<?> inStream1 = ...;
Table appendOnlyTable = tableEnv.fromDataStream(inStream1);
DataStream<Row> inStream2 = ...
Table anyTable = tableEnv.fromChangelogStream(inStream2);

// Table -> Stream
DataStream<T> appendOnlyStream = tableEnv.toDataStream(insertOnlyTable, T.class);
DataStream<Row> changelogStream = tableEnv.toChangelogStream(anyTable);
```
Changelog Stream Processing
Data Processing is a Stream of Changes

- Business data is always a stream: **bounded** or **unbounded**
- Every record is a changelog entry: **insertion as the default**
- Batch processing is just a **special case** in the runtime
How do I Work with Streams in Flink SQL?

• You don't. You work with **dynamic tables**!
• A concept similar to **materialized views**

```
CREATE TABLE Transactions
  (name STRING, amount INT)
WITH (...)
```

```
<table>
<thead>
<tr>
<th>name</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>56</td>
</tr>
<tr>
<td>Bob</td>
<td>10</td>
</tr>
<tr>
<td>Alice</td>
<td>89</td>
</tr>
</tbody>
</table>
```

```
INSERT INTO Revenue
  SELECT name, SUM(amount)
  FROM Transactions
  GROUP BY name
```

```
CREATE TABLE Revenue
  (name STRING, total INT)
WITH (...)
```

```
<table>
<thead>
<tr>
<th>name</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>145</td>
</tr>
<tr>
<td>Bob</td>
<td>10</td>
</tr>
</tbody>
</table>
```

So, is Flink SQL a database?  No, bring your own data and systems!
Stream-Table Duality - Basics

- A stream is the changelog of a **dynamic table**
- Sources, operators, and sinks work on **changelogs under the hood**

<table>
<thead>
<tr>
<th>Short name</th>
<th>Long name</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>+I</td>
<td>Insertion</td>
<td>Default for scans + output of bounded results.</td>
</tr>
<tr>
<td>-U</td>
<td>Update Before</td>
<td>Retracts a previously emitted result.</td>
</tr>
<tr>
<td>+U</td>
<td>Update After</td>
<td>Updates a previously emitted result. Requires a primary key if -U is omitted for idempotent updates.</td>
</tr>
<tr>
<td>-D</td>
<td>Delete</td>
<td>Removes the last result.</td>
</tr>
</tbody>
</table>

- Each component declares the **kind of changes it consumes/produces**

  only +I  Appending/Insert-only
contains -…  Updating
contains -U  Retracting
never -U but +U  Upserting
Stream-Table Duality - Example

An applied changelog becomes a real (materialized) table.

```
CREATE TABLE Transactions
(name STRING, amount INT)
WITH (…)
```

<table>
<thead>
<tr>
<th>name</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>56</td>
</tr>
<tr>
<td>Bob</td>
<td>10</td>
</tr>
<tr>
<td>Alice</td>
<td>89</td>
</tr>
</tbody>
</table>

```
CREATE TABLE Revenue
(name STRING, total INT)
WITH (…)
```

<table>
<thead>
<tr>
<th>name</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>145</td>
</tr>
<tr>
<td>Bob</td>
<td>10</td>
</tr>
</tbody>
</table>

INSERT INTO Revenue
SELECT name, SUM(amount)
FROM Transactions
GROUP BY name

```
+I[Alice, 89]  +I[Bob, 10]  +I[Alice, 56]
```

```
+U[Alice, 145] -U[Alice, 56] +I[Bob, 10] +I[Alice, 56]
```

**Materialization**

**Changelog**
Stream-Table Duality - Example

An applied changelog becomes a real (materialized) table.

CREATE TABLE Transactions
(name STRING, amount INT)
WITH (…)

<table>
<thead>
<tr>
<th>name</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>56</td>
</tr>
<tr>
<td>Bob</td>
<td>10</td>
</tr>
<tr>
<td>Alice</td>
<td>89</td>
</tr>
</tbody>
</table>

INSERT INTO Revenue
SELECT name, SUM(amount)
FROM Transactions
GROUP BY name

CREATE TABLE Revenue
(PRIMARY KEY(name) ..)
WITH (…)

<table>
<thead>
<tr>
<th>name</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>145</td>
</tr>
<tr>
<td>Bob</td>
<td>10</td>
</tr>
</tbody>
</table>


materialization

Save ~50% of traffic if downstream system supports upserting!
Stream-Table Duality - Propagation

- Source declares set of emitted changes i.e. changelog mode

  CREATE TABLE ... (for sources)

  ... WITH ('connector'='filesystem') +I

  ... WITH ('connector'='kafka') +I

  ... WITH ('connector'='kafka-upsert') +I -D

  ... WITH ('connector'='jdbc') +I

  ... WITH ('connector'='kafka', 'format' = 'debezium-json') +I -U +U -D

- Optimizer tracks changelog mode and primary key through pipeline
- Sink declares changes it can digest
Retract vs. Upsert

**Retract**
- No primary key requirements
- Works for almost every external system
- Supports duplicate rows
- In distributed system often unavoidable

→ **most flexible changelog mode**
→ **default mode**

**Upsert**
- Traffic + computation **optimization**
- In-place updates (**idempotency**)
Changelog Insights – Append-only

CREATE TABLE Transaction (tid BIGINT, amount INT);
CREATE TABLE Payment (tid BIGINT, method STRING);
CREATE TABLE Result (tid BIGINT, ...); // accepts all changes
INSERT INTO Result SELECT * FROM Transactions T JOIN Payments P ON T.tid = P.tid;

Sink(table=[Result], changelogMode=[NONE])
+- Join(leftInputSpec=[NoUniqueKey], rightInputSpec=[NoUniqueKey], changelogMode=[I])
   :  Exchange(changelogMode=[I])
   :  +- TableSourceScan(table=[[Transaction]], changelogMode=[I])
+- Exchange(changelogMode=[I])
   +- TableSourceScan(table=[[Payment]], changelogMode=[I])
Changelog Insights – Updating

CREATE TABLE Transaction (tid BIGINT, amount INT);
CREATE TABLE Payment (tid BIGINT, method STRING);
CREATE TABLE Result (tid BIGINT, ...);
INSERT INTO Result SELECT * FROM Transactions T LEFT JOIN Payments P ON T.tid = P.tid;

Sink(table=[Result], changelogMode=[NONE])
+- Join(leftInputSpec=[NoUniqueKey], rightInputSpec=[NoUniqueKey], changelogMode=[I,UB,UA,D])
  :- Exchange(changelogMode=[I])
  :  +- TableSourceScan(table=[[Transaction]], changelogMode=[I])
  +- Exchange(changelogMode=[I])
     +- TableSourceScan(table=[[Payment]], changelogMode=[I])
CREATE TABLE Transaction (tid BIGINT, ..., PRIMARY KEY(tid) NOT ENFORCED);
CREATE TABLE Payment (tid BIGINT, ..., PRIMARY KEY(tid) NOT ENFORCED);
CREATE TABLE Result (tid BIGINT, ..., PRIMARY KEY(tid) NOT ENFORCED);
INSERT INTO Result SELECT * FROM Transactions T LEFT JOIN Payments P ON T.tid = P.tid;

Sink(table=[Result], changelogMode=[NONE])
+- Join(leftInputSpec=[UniqueKey], rightInputSpec=[UniqueKey], changelogMode=[I,UA,D])
  :- Exchange(changelogMode=[I])
  :  +- TableSourceScan(table=[[Transaction]], changelogMode=[I])
  +- Exchange(changelogMode=[I])
    +- TableSourceScan(table=[[Payment]], changelogMode=[I])
Mode Transitions

- **Append-only**
  - through operation
  - if operator/sink requires it
    - ChangelogNormalize
- **Updating**
  - if sink requires it
    - UpsertMaterialize
- **Retracting**
# Mode Transitions – Joins

<table>
<thead>
<tr>
<th>Append-only</th>
<th>regular join</th>
<th>Append-only</th>
<th>Append-only</th>
</tr>
</thead>
<tbody>
<tr>
<td>Append-only</td>
<td>regular join</td>
<td>Updating</td>
<td>Updating</td>
</tr>
<tr>
<td>Append-only</td>
<td>regular outer join</td>
<td>Append-only</td>
<td>Updating</td>
</tr>
<tr>
<td>Append-only</td>
<td>temporal join</td>
<td>Updating</td>
<td>Append-only</td>
</tr>
</tbody>
</table>
CREATE TABLE CurrencyRates (  
    WATERMARK FOR update_time AS …, PRIMARY KEY(currency) NOT ENFORCED,…);  

SELECT  
    order_id,  
    price,  
    currency,  
    conversion_rate,  
    order_time  
FROM Orders  
LEFT JOIN CurrencyRates FOR SYSTEM_TIME AS OF Orders.order_time  
ON Orders.currency = CurrencyRates.currency;
Demo

https://github.com/twalthr/flink-api-examples
Summary

TLDR

• Flink’s SQL engine is a powerful changelog processor
• Flexible tool for integrating systems with different semantics

There is more...

• Large coverage of the SQL standard
  → OVER for streaming aggregation and dedup
  → MATCH_RECOGNIZE for pattern matching
  → TUMBLE/HOP/SESSION for windowing
  → ...

• CDC connector ecosystem
  → 3.5k Github stars
  https://flink-packages.org/packages/cdc-connectors

• Table Store
  → unified storage engine for dynamic tables
  https://nightlies.apache.org/flink/flink-table-store-docs-master/docs/concepts/overview/
Thank you!

Feel free to follow:

@twalthr