Streams and Tables: Two Sides of the Same Coin

Twelfth International Workshop on Real-Time Business Intelligence and Analytics
27 August, 2018, Rio de Janeiro

Matthias J. Sax\textsuperscript{1,2}, Guozhang Wang\textsuperscript{1}, Matthias Weidlich\textsuperscript{2}, Johann-Christoph Freytag\textsuperscript{2}

\textsuperscript{1}\textit{Confluent Inc., Palo Alto (CA)}
\textsuperscript{2}\textit{Humboldt-Universität zu Berlin}

\texttt{mjsax@informatik.hu-berlin.de}
\texttt{matthias.weidlich@hu-berlin.de}
\texttt{freytag@informatik.hu-berlin.de}

@MatthiasJSax
Count Clicks per Page

Distributed Data Source

Input Data Stream

BIRTE(3)

VLDB(4)

VLDB(7)

VLDB(4) BIRTE(3)

VLDB(7)
Arrival order non-deterministic

Even-time semantics implies out-of-order data
Ordering: Common Approaches

**Input Data Stream**

VLDB(7)  BIRTE(3)  VLDB(4)

**buffer and re-order**

VLDB(7)  VLDB(4)  BIRTE(3)

**punctuations/watermarks**

VLDB(7)  BIRTE(3)  VLDB(4)

time=3

SPS
Design Space

Buffering and Reordering
- Ref: CQL\(^1\), Trill\(^2\)

Punctuations/Watermarks
- Ref: Li et al.\(^3\), Krishnamurthy et al.\(^4\)
Problem Statement

How to design a model

• for the evaluation of expressive operators
• with low latency over potentially unordered data streams
• that can be implemented by mean of distributed online algorithms?
High-Level Proposal

• To reduce latency, we need to avoid any processing delays
  • Process data in arrival order
  • Emit current result immediately
  • Law et al.\textsuperscript{5}: \textit{cannot handle out-of-order data}

• To handle out-of-order data, we need to be able to update/refine previous results
  • Data streams must allow for \textit{update records}
  • Update/delete records by Babu and Widom\textsuperscript{6}: \textit{no operator semantics} defined
  • Borealis\textsuperscript{7} replays data stream after “updating/reordering”; \textit{very high cost}
Data Model

- Offset: physical order (arrival/processing order)
- Timestamp: logical order (event-time)
- Key: optional for grouping
- Value: payload
Stream Processing Operators

- Stateless, order agnostic
  - filter, projection, flatMap
  - No special handling necessary

- Stateful, order sensitive
  - aggregation, joins, windowing
  - Need to handle out-of-order data
Data Stream Aggregation

- Model output of (windowed) aggregations as table
  - State is not internal but first-class citizen
  - Update stateful operator continuously
  - Emit changelog stream to downstream operators

- Streams, Table, and Changelogs
  - Define operator semantics over changelogs and updating tables
  - Temporal operator semantics
Example: Count Clicks per Page

```
countTable = stream.groupBy(r->url).count()
```
Example: Count Clicks per Page

```scala
countTable = stream.groupBy(r->url).count()
```

<table>
<thead>
<tr>
<th>url</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIRTE</td>
<td>1</td>
</tr>
<tr>
<td>VLDB</td>
<td>1</td>
</tr>
</tbody>
</table>

VLDB

record stream

<VLDB,1>  <BIRTE,1>

changelog stream
Example: Count Clicks per Page

countTable = stream.groupBy(r->url).count()
Example: Processing a Changelog Stream

countTable2 = countTable.filter(url='VLDB').toTable()

<table>
<thead>
<tr>
<th>url</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIRTE</td>
<td>1</td>
</tr>
<tr>
<td>VLDB</td>
<td>2</td>
</tr>
</tbody>
</table>

changelog stream

<table>
<thead>
<tr>
<th>url</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>VLDB</td>
<td>2</td>
</tr>
</tbody>
</table>
Example: Windowed Count

```
countTable = stream.groupBy(r-&gt;url).windowedBy(5sec).count()
```

<table>
<thead>
<tr>
<th>window ID</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;BIRTE,0&gt;</td>
<td>1</td>
</tr>
<tr>
<td>&lt;VLDB,0&gt;</td>
<td>1</td>
</tr>
<tr>
<td>&lt;VLDB,5&gt;</td>
<td>1</td>
</tr>
</tbody>
</table>

windowID = <groupingKey,windowStartTimestamp>

windowStartTimestamp = recordTimestamp / windowSize
countTable = stream.groupBy(r->url).windowedBy(5sec).count()

<table>
<thead>
<tr>
<th>window ID</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;BIRTE,0&gt;</td>
<td>2</td>
</tr>
<tr>
<td>&lt;VLDB,0&gt;</td>
<td>1</td>
</tr>
<tr>
<td>&lt;VLDB,5&gt;</td>
<td>1</td>
</tr>
</tbody>
</table>

windowID = <groupingKey,windowStartTimestamp>

windowStartTimestamp = recordTimestamp / windowSize
Duality of Streams and Tables

Table → capture updates → Table Changelog Stream

Table Changelog Stream → materialize updates ← Table
Design Space

Buffering and Reordering
- Ref: CQL\(^1\), Trill\(^2\)

Punctuations/Watermarks
- Ref: Li et al.\(^3\), Krishnamurthy et al.\(^4\)

Dual Streaming Model
- continuous updates / changelogs
- decouple latency from correctness
- trade-off latency and cost
- trade-off cost and completeness (retention time)
Stream-Table Transformations

See the paper for details...
Implementation

- Implemented in Apache Kafka (v0.10)
- Kafka Streams / Streams API

```java
StreamsBuilder builder = new StreamsBuilder();

KStream<String, String> textLines = builder.stream("TextLinesTopic");
KTable<String, Long> wordCounts = textLines
    .flatMapValues(textLine -> Arrays.asList(textLine.toLowerCase().split("\W+"))
    .groupBy((key, word) -> word)
    .windowedBy(TimeWindows.of(5_000L))
    .count();
wordCounts.toStream().to("WordsWithCountsTopic");

KafkaStreams streams = new KafkaStreams(builder.build(), props);
streams.start();
```
Implementation

- Implemented in Apache Kafka (v0.10)
- Kafka Streams / Streams API
- Leveraged in Confluent’s KSQL

```
CREATE TABLE click_count_per_url AS
    SELECT url, count(*)
    FROM click_stream
    WINDOW TUMBLING (SIZE 1 MINUTE)
    WHERE url LIKE '%confluent%' OR url LIKE '%hu-berlin%'
    GROUP BY url;
```
Implementation

• Implemented in Apache Kafka (v0.10)
  • Kafka Streams / Streams API
  • Leveraged in Confluent’s KSQL
  • Widely adopted in industry

LINE

LINE uses Apache Kafka as a central datahub for our services to communicate to one another. Hundreds of billions of messages are produced daily and are used to execute various business logic, threat detection, search indexing and data analysis. LINE leverages Kafka Streams to reliably transform and filter topics enabling sub topics consumers can efficiently consume, meanwhile retaining easy maintainability thanks to its sophisticated yet minimal code base.

Pinterest

Pinterest uses Apache Kafka and the Kafka Streams at large scale to power the real-time, predictive budgeting system of their advertising infrastructure. With Kafka Streams, spend predictions are more accurate than ever.

Rabobank

Rabobank is one of the 3 largest banks in the Netherlands. Its digital nervous system, the Business Event Bus, is powered by Apache Kafka. It is used by an increasing amount of financial processes and services, one of which is Rabo Alerts. This service alerts customers in real-time upon financial events and is built using Kafka Streams.

The New York Times

The New York Times uses Apache Kafka and the Kafka Streams to store and distribute, in real-time, published content to the various applications and systems that make it available to the readers.

Zalando

As the leading online fashion retailer in Europe, Zalando uses Kafka as an ESB (Enterprise Service Bus), which helps us in transitioning from a monolithic to a microservices architecture. Using Kafka for processing events enables our technical team to do near-real time business intelligence.

Trivago

Trivago is a global hotel search platform. We are focused on reshaping the way travelers search for and compare hotels, while enabling hotel advertisers to grow their businesses by providing access to a broad audience of travelers via our websites and apps. As of 2017, we offer access to approximately 1.8 million hotels and other accommodations in over 190 countries. We use Kafka, Kafka Connect, and Kafka Streams to enable our developers to access data freely in the company. Kafka Streams powers parts of our analytics pipeline and delivers endless options to explore and operate on the data sources we have at hand.
Summary

• Suggest the Dual-Streaming-Model
  • Handles out-of-order data *within* the processing model
  • Optimized for low latency
• Streams and Tables are Dual
• Allows to trade-off processing cost, latency, completeness
• Adopted in industry via Kafka Streams and KSQL
Thank You

We are hiring!
References


Data Streams Types

Data Stream

Record Stream
⇒ $r.k = \bot$

Table Changelog Stream
⇒ $r.k \neq \bot$
# Evolving Table

## Record Stream

<table>
<thead>
<tr>
<th>window ID</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;BIRTE,0&gt;</td>
<td>1</td>
</tr>
<tr>
<td>&lt;VLDB,0&gt;</td>
<td>1</td>
</tr>
</tbody>
</table>

## Tables

### Table v3

- BIRTE(3)
- VLDB(7)

### Table v4

- BIRTE(1)
- VLDB(4)

### Table v7

- <BIRTE,0> 1
- <VLDB,0> 1
- <VLDB,5> 1
## Evolving Table

<table>
<thead>
<tr>
<th>window ID</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;BIRTE,0&gt;</td>
<td>1</td>
</tr>
<tr>
<td>&lt;VLDB,0&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;VLDB,5&gt;</td>
<td></td>
</tr>
</tbody>
</table>

The record stream is: BIRTE(1)  VLDB(7)  VLDB(4)  BIRTE(3)

### Table v1

<table>
<thead>
<tr>
<th>window ID</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;BIRTE,0&gt;</td>
<td>2</td>
</tr>
</tbody>
</table>

### Table v3

<table>
<thead>
<tr>
<th>window ID</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;BIRTE,0&gt;</td>
<td>2</td>
</tr>
</tbody>
</table>

### Table v4

<table>
<thead>
<tr>
<th>window ID</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;VLDB,0&gt;</td>
<td>1</td>
</tr>
</tbody>
</table>

### Table v7

<table>
<thead>
<tr>
<th>window ID</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;VLDB,5&gt;</td>
<td>1</td>
</tr>
</tbody>
</table>
Table-Table Join

left table $A$

right table $B$

result table $R$

$A_1$, $A_5$, $A_6$

$B_2$, $B_3$, $B_6$

$R_2$, $R_3$, $R_5$, $R_6$

1 2 3 4 5 6 time
Stream-Stream Join

- Sliding Window Join, i.e., band join
- Window size specifies additional timestamp based join predicate

```
SELECT * FROM stream1, stream2
WHERE
    stream1.key = stream2.key
AND
    stream1.ts - windowSize <= stream2.ts
AND
    stream2.ts <= stream1.ts + windowSize
```
Stream-Table Join

- Temporal table “lookup” join
- For each stream record, lookup for a matching table record
- Join condition: `streamRecord.key == tableRecord.key`
- The join is temporal in the sense, that the “correct” table `version` must be used
  - i.e., youngest table version that is before the stream records timestamp
Stream-Table Join

<table>
<thead>
<tr>
<th>table over time</th>
<th>$T_3$</th>
<th>$T_5$</th>
<th>$T_6$</th>
<th>$T_8$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k$</td>
<td>$v$</td>
<td>$k$</td>
<td>$v$</td>
<td>$k$</td>
</tr>
<tr>
<td>B</td>
<td>12.1</td>
<td>A</td>
<td>7.2</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B</td>
<td>12.1</td>
<td>B</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>B</td>
<td></td>
</tr>
</tbody>
</table>

| record stream | $\langle 2, B, 3.5 \rangle$ | $\langle 5, A, 4.2 \rangle$ | $\langle 6, C, 6.4 \rangle$ | $\langle 7, B, 1.2 \rangle$ |

| result         | $\langle 5, A, 4.2 \bowtie 7.2 \rangle$ | $\langle 7, B, 1.2 \bowtie 14.7 \rangle$ |

---

1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | time