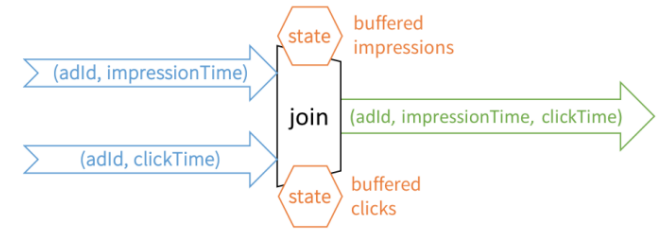
Introducing Stream-Stream Joins in Apache Spark 2.3

Now Available on Databricks Runtime 4.0



ream join use case: Ad-Monetization (joining ad clicks to im

Published: March 13, 2018 | Open Source | 7 min read | By Tathagata Das and Joseph Torres

Since we introduced Structured Streaming in Apache Spark 2.0, it has supported joins (inner join and some type of outer joins) between a streaming and a static DataFrame/Dataset. With the release of Apache Spark 2.3.0, now available in Databricks Runtime 4.0 as part of

Contents in this

Databricks Unified Analytics Platform, we now support stream-stream joins. In this post, we will explore a canonical case of how to use stream-stream joins, what challenges we resolved, and what type of workloads they enable. Let's start with the canonical use case for stream-stream joins - ad monetization.

The Case for Stream-Stream Joins: Ad Monetization

Imagine you have two streams - one stream of ad impressions (i.e., when an advertisement was displayed to a user) and another stream of ad clicks (i.e., when the displayed ad was clicked by the user). To monetize the ads, you have to match which ad impression led to a click. In other words, you need to join these streams based on a common key, the unique identifier of each ad that is present in events of both streams. At a high-level, the problem looks like as follows.

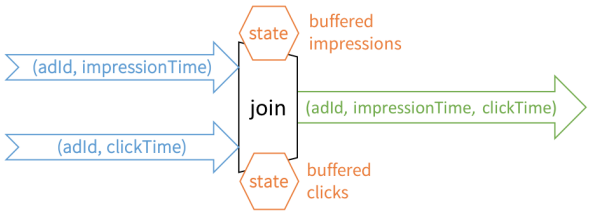
story

The Case for Stream-Stream Joins: Ad...

Managing the Streaming State for Stream-Stream...

Using Stream-Stream Outer Joins

Further Reading



Stream-stream join use case: Ad-Monetization (joining ad clicks to impressions)

While this is conceptually a simple idea, there are a few core technical challenges to overcome.

1. **Handling of late/delayed data with buffering:** An impression event and its corresponding click event may arrive out-of-order with arbitrary delays between them. Hence, a stream processing engine must account for such delays by appropriately buffering them until they are matched. Even though all joins (static or streaming) may use buffers, the real challenge is to avoid the buffer from growing without limits.
2. **Limiting buffer size:** The only way to limit the size of a streaming join buffer is by dropping delayed data beyond a certain threshold. This maximum-delay threshold should be configurable by the user depending on the balance between the business requirements and systems' resource limitations.

3. **Well defined semantics:** Maintain

consistent SQL join semantics between static joins and streaming joins, with or without the aforementioned thresholds.

We have solved all these challenges in our stream-stream joins. As a result, you can express your computation using the clear semantics of SQL joins, as well as control the delay to tolerate between the associated events. Let's see how.

First let's assume these streams are two different Kafka topics. You would define the streaming DataFrames as follows:

PYTHON

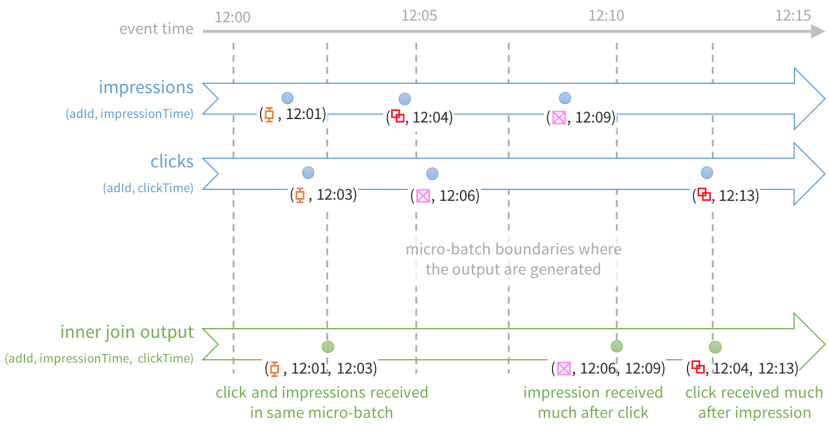
```
impressions = (     # schema - adId: String, im
  spark
    .readStream
    .format("kafka")
    .option("subscribe", "impressions")
    …
    .load()
)

clicks = (      # schema - adId: String, clickI
  spark
    .readStream
    .format("kafka")
    .option("subscribe", "clicks")
    …
    .load()
)
```

Then all you need to do inner equi-join them is as follows.

PYTHON

```
impressions.join(clicks, "adId")   # adId is c
```

As with all Structured Streaming queries, this code is the exactly the same as you would have written if the DataFrames `impressions` and `clicks` were defined on static data. When this query is executed, the Structured Streaming engine will buffer clicks and impressions as the streaming state as needed. For a particular advertisement, the joined output will be generated as soon as both related events are received (that is, as soon as the second event is received). As data arrives, the joined output will be generated incrementally and written to the query sink (e.g. another Kafka topic).



Stream-stream joins: sample timeline of join between an ad impressions stream and a click stream

Finally, the cumulative result of the join will be no different had the join query been applied on two static datasets (that is, same semantics as SQL joins). In fact, it would be the same even if one was presented as a stream and the other as a static dataset. However, in this query, we have not given any indication on how long the engine should buffer an event to find a match. Therefore, the engine may buffer an event forever and accumulate an unbounded amount of streaming state. Let's see how we can provide additional information in the query to limit the state.

Managing the Streaming State for Stream-Stream Joins

To limit the streaming state maintained by

stream-stream joins, you need to know the following information about your use case:

1. What is the time range between the generation of the two events at their respective sources? In the context of our use case, let's assume that a click can occur within 0 seconds to 1 hour after the corresponding impression.
2. What is the maximum duration an event can be delayed in transit between the source and the processing engine? For example, ad clicks from a browser may get delayed due to intermittent connectivity and arrive much later and out-of-order than expected. Let's say, that impressions and clicks can be delayed by at most 2 and 3 hours, respectively.

With these time constraints for each event, the processing engine can automatically calculate how long events need to be buffered for generating correct results. For example, it will evaluate the following.

1. Impressions need to be buffered for at most 4 hours (in event-time) as a 3-hour-late click may match with an impression made 4 hours ago (i.e., 3-hour-late + upto 1 hour delay between

the impression and click).

2. Conversely, clicks need to be buffered for at most 2 hours (in event-time) as a 2-hour-late impression may match with click received 2 hours ago.

Accordingly, the engine can drop old impressions and clicks from streams when it determines that any of the buffered event is not expected to get any matches in the future.

At high-level this animation illustrates how the watermark is updated with event time and how state is cleanup.

These time constraints can be encoded in the query as watermarks and time range join conditions.

- **Watermarks:** Watermarking in Structured

Streaming is a way to limit state in all stateful streaming operations by specifying how much late data to consider. Specifically, a watermark is a moving threshold in event-time that trails behind the maximum event-time seen by the query in the processed data. The trailing gap (aka watermark delay) defines how long should the engine wait for late data to arrive and is specified in the query using `withWatermark`. Read about it in more detail in our previous blog post on streaming aggregations. For our stream-stream inner joins, you can optionally specify the watermark delay but you must specify to limit all state on both streams.

- **Time range condition:** It's a join condition that limits the time range of other events that each event can join against. This can be specified one of the two ways:

  - Time range join condition (e.g. ... JOIN ON leftTime BETWEEN rightTime AND rightTime + INTERVAL 1 HOUR),
  - Join on event-time windows (e.g. ... JOIN ON leftTimeWindow = rightTimeWindow).

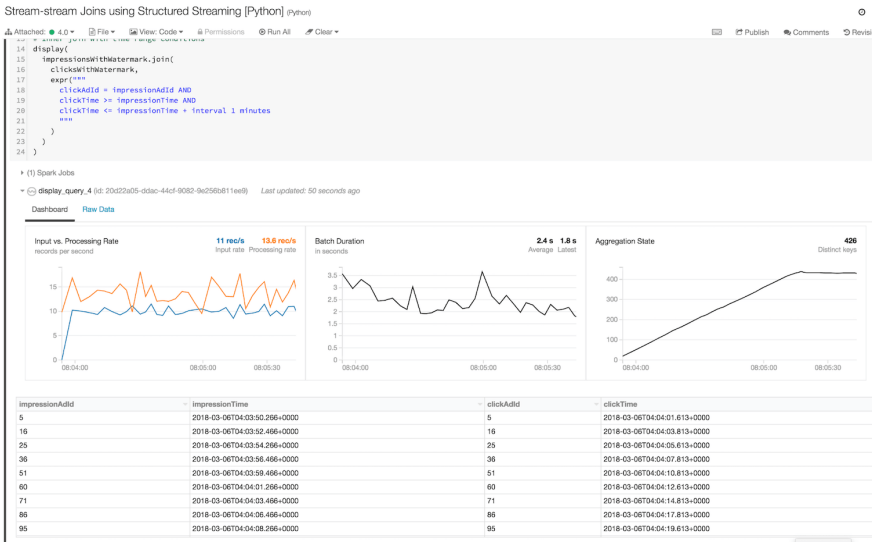
Together, our inner join for ad monetization will look like this.

PYTHON

```
from pyspark.sql.functions import expr

# Define watermarks
impressionsWithWatermark = impressions \
  .selectExpr("adId AS impressionAdId", "impre
  .withWatermark("impressionTime", "10 seconds

clicksWithWatermark = clicks \
  .selectExpr("adId AS clickAdId", "clickTime'
  .withWatermark("clickTime", "20 seconds")


# Inner join with time range conditions
impressionsWithWatermark.join(
  clicksWithWatermark,
  expr("""
   clickAdId = impressionAdId AND
    clickTime >= impressionTime AND
    clickTime
```

With this, the engine will automatically calculate state limits mentioned earlier and drop old events accordingly. And as with all things stateful in Structured Streaming, checkpointing ensures that you get exactly-once fault-tolerance guarantees.

Here is a screenshot of the query running in the Databricks notebook linked with this post. Note the third graph, the number of the records in query state, flattens out after a while indicating the state cleanup by watermark is cleaning up old data.



Using Stream-Stream Outer Joins

The earlier inner join will output only those ads for which both events have been received. In other words, ads that received no click would not be reported at all. Instead you may want *all* ad impressions to be reported, with or without the associated click data, to enable additional analysis later

(e.g. click through rates). This brings us to stream-stream outer joins. All you need to do is specify the join type.

PYTHON

```
from pyspark.sql.functions import expr

# Left outer join with time range conditions
impressionsWithWatermark.join(
  clicksWithWatermark,
  expr("""
    clickAdId = impressionAdId AND
    clickTime >= impressionTime AND
    clickTime
```

As expected of outer joins, this query will start generating output for every impression, with or without (i.e., using NULLS) the click data. However, outer joins have a few additional points to note.

- Unlike inner joins, the watermarks and event-time constraints are not optional for outer joins. This is because for generating the NULL results, the engine must know when an event is not going to match with anything else in future. Hence, the watermarks and event-time constraints must be specified for enabling state expiration and generating

correct outer join results.

- Consequently, the outer NULL results will be generated with a delay as the engine has to wait for a while to ensure that there neither were nor would be any matches. This delay is the maximum buffering time (wrt to event-time) calculated by the engine for each event as discussed in the earlier section (i.e., 4 hours for impressions and 2 hours for clicks).

Further Reading

For full details on supported types of joins and other query limits take a look at the Structured Streaming programming guide. For more information on other stateful operations in Structured Streaming, take a look at the following:

- **Notebook:** Try stream-stream join notebook on Databricks
- **Blog post:** Arbitrary Stateful Processing in Apache Spark's Structured Streaming
- **Spark Summit Europe 2017 talk:** Deep Dive into Stateful Stream Processing in Structured Streaming

What's next?

Explore more from the authors

OPEN SOURCE

MARCH 22, 2024 / 10 MIN READ

GGML GGUF File Format Vulnerabilities

OPEN SOURCE

JUNE 5, 2024 / 3 MIN READ

BigQuery adds first-party support for Delta Lake

Introducing Low-latency Continuous Processing Mode in Structured Streaming in Apache Spark 2.3

Get Your Free Copy of Delta Lake: The Definitive Guide (Early Release)

Faster MERGE Performance With Low-Shuffle MERGE and Photon

et, 15th Floor
CA 94105
21

s

Why Databricks

Discover

For Executives

For Startups

Lakehouse Architecture

Mosaic Research

Customers

Customer Stories

Partners

Cloud Providers

Technology Partners

Data Partners

Built on Databricks

Consulting & System Integrators

C&SI Partner Program

Partner Solutions

Product

Databricks Platform

Platform Overview

Sharing

Governance

Artificial Intelligence

Business Intelligence

Database

Data Management

Data Warehousing

Data Engineering

Data Science

Application Development

Pricing

Pricing Overview

Pricing Calculator

Open Source

Solutions

Databricks For Industries

Communications

Financial Services

Healthcare and Life Sciences

Manufacturing

Media and Entertainment

Public Sector

Retail

View All

Cross Industry Solutions

Cybersecurity

Marketing

Data Migration

Professional Services

Solution Accelerators

Resources

Documentation

Customer Support

Community

Learning

Training

Certification

Free Edition

University Alliance

Databricks Academy Login

Events

Data + AI Summit

Data + AI World Tour

Data Intelligence Days

Event Calendar

About

Company

Who We Are

Our Team

Databricks Ventures

Contact Us

Careers

Open Jobs

Working at Databricks

Press

Awards and Recognition

Newsroom

Security and Trust

Integrations and Data

Marketplace

IDE Integrations

Partner Connect

Blog and Podcasts

Databricks Blog

Databricks Mosaic Research Blog

Data Brew Podcast

Champions of Data & AI Podcast