Connecting streams and databases

Gian Merlino
Druid committer • Cofounder @ Imply
Overview

Why streams and why databases?

Things you may care about

How popular stream systems work

How we deal with streams in Druid
Stream processing

Streams
Stream processing
Stream processing

Streams

Actions
Stream processing

Streams

You
Stream processing
Stream processing

Streams → Stream processor → Derived Streams
Stream processing
Stream processing
Why **streams**?

- Real-time monitoring
- Real-time response
- Shorter feedback loops
- Better user experience
Why stream processing?

- Original streams are not exactly what we want
- Common things to want
  - Enhancement
  - Session reconstruction (and other joins)
  - Load into databases
Why databases?

- Lots of questions to ask
- Streaming through raw data, for every question, is slow
- Faster to load a derivative into a database, query it there
  - DB strengths: ad-hoc search, aggregation, key lookup
  - DB weaknesses: joining big distributed tables, joining external data
Stream operations
Basic operations

Filter

Map

Output

INSERT INTO mydata VALUES …
Grouping operations

GroupByKey
Grouping operations

- Tricky!
- Data points for a window may come in “late”
- Windows may be aligned (e.g. aggregates)
- Windows may be unaligned (e.g. sessions)
Requirements
Requirements

- Correctness
- Latency
- Cost
- (Thanks, Akidau et. al)
Correctness

- Want accurate reflection of reality
Correctness

- Message processing guarantees
  - None
  - At most once
  - At least once
  - Exactly once
Correctness

- Window emitting guarantees
  - Wait for “enough” data before emitting, and emit once
  - Emit periodic updates
Latency

Very low latency
subsecond

“Low” latency
seconds – minutes

High latency
hours – days
Data pipelines
Goals

- Low-latency results
- Strong correctness guarantees
- Ability to do backfills
Why backfills?

- Bugs in processing code
- Need to restate existing data
- Limitations of some current streaming software
Backfills: Lambda

- Hybrid batch/realtime a.k.a. “lambda architecture”
- Backfills automated or on-demand
- Pros: Can achieve goals with a wide variety of OSS
- Cons: Operations and development are complex
Backfills: Lambda
Backfills: Lambda

- Batch technologies
  - Hadoop Map/Reduce
  - Spark
- Streaming technologies
  - Samza
  - Storm
  - Spark Streaming
Backfills: Lambda

- Software exists to simplify development
  - Summingbird
  - Google Cloud Dataflow
  - Starfire (internal tool)
Backfills: Stream replay

- Stream replay a.k.a. “kappa architecture”
- Backfills on demand
- Simpler development and operations
- Workable if stream processing guarantees are strong enough
Stream and batch processing not too different on unbounded datasets

Batch processors must still deal with late data
Streaming systems
Kafka

http://kafka.apache.org/documentation.html
Kafka: API

- Producer: Send message to a (topic, partition)
- Consumer: Read messages from a (topic, partition)
- Very low latency
- Can do simple operations directly with the Kafka API
- More complex processing is easier with a “real” stream processor
Kafka: API

- Possible to integrate closely, and efficiently, with databases
- Not an accident
Kafka: Guarantees

- Producer: At-least-once, if configured appropriately
- Consumer
  - At-least-once straightforward with high-level consumer
  - Exactly-once (from Kafka data!) can be done with more work
Kafka: Guarantees

- Exactly-once strategies
- Naturally unique message IDs
  - Must assign outside of Kafka
  - De-duplicate messages while consuming
  - Must make sure to keep around enough de-duplication data
- Single-writer-per-partition
  - Duplicate messages will be adjacent; ignore them
Storm

http://storm.apache.org/tutorial.html
Storm

- Messages acked at spout when fully processed
- Spouts typically checkpoint after acks
- At-least-once if spouts are able to replay
- Exactly-once with idempotent operations
- No innate concept of state
Storm / Trident

- Does have concept of state
- Messages grouped into batches
- Each batch given a transaction id (txid)
- Txids globally ordered, meant to be stored in DB
- Skip DB update for stale txids
- Coordination overhead
Samza

http://samza.apache.org/learn/documentation/0.9/introduction/concepts.html
● Periodically flush output and checkpoint Kafka offsets
● At-least-once
● Exactly-once with idempotent operations
Druid
Druid

- Open source column store
- Designed for fast filtering and aggregations
- Unique optimizations for event data
- Data partitioning/sharding first done on time
- Data is partitioned into defined time buckets (hour/day/etc)
## Druid Segments

### Partition by time

<table>
<thead>
<tr>
<th>Timestamp</th>
<th>Page</th>
<th>Views</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015-01-01T00</td>
<td>p1</td>
<td>1</td>
</tr>
<tr>
<td>2015-01-01T01</td>
<td>p2</td>
<td>1</td>
</tr>
<tr>
<td>2015-01-02T05</td>
<td>p3</td>
<td>1</td>
</tr>
<tr>
<td>2015-01-02T07</td>
<td>p4</td>
<td>1</td>
</tr>
<tr>
<td>2015-01-03T05</td>
<td>p5</td>
<td>1</td>
</tr>
<tr>
<td>2015-01-03T07</td>
<td>p6</td>
<td>1</td>
</tr>
</tbody>
</table>

**Segment**

- **Segment_2015-01-01/2014-01-02**
<table>
<thead>
<tr>
<th>Timestamp</th>
<th>Page</th>
<th>Views</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015-01-01T00</td>
<td>p1</td>
<td>1</td>
</tr>
<tr>
<td>2015-01-01T01</td>
<td>p2</td>
<td>1</td>
</tr>
</tbody>
</table>

- **Segment_2015-01-02/2014-01-03**
<table>
<thead>
<tr>
<th>Timestamp</th>
<th>Page</th>
<th>Views</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015-01-02T05</td>
<td>p3</td>
<td>1</td>
</tr>
<tr>
<td>2015-01-02T07</td>
<td>p4</td>
<td>1</td>
</tr>
</tbody>
</table>

- **Segment_2015-01-03/2014-01-04**
<table>
<thead>
<tr>
<th>Timestamp</th>
<th>Page</th>
<th>Views</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015-01-03T05</td>
<td>p5</td>
<td>1</td>
</tr>
<tr>
<td>2015-01-03T07</td>
<td>p6</td>
<td>1</td>
</tr>
</tbody>
</table>
### Rollup on ingestion

<table>
<thead>
<tr>
<th>timestamp</th>
<th>publisher</th>
<th>advertiser</th>
<th>gender</th>
<th>country</th>
<th>click</th>
<th>revenue</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011-01-01T01:01:35Z</td>
<td>bieberfever.com</td>
<td>google.com</td>
<td>Male</td>
<td>USA</td>
<td>0</td>
<td>0.65</td>
</tr>
<tr>
<td>2011-01-01T01:03:63Z</td>
<td>bieberfever.com</td>
<td>google.com</td>
<td>Male</td>
<td>USA</td>
<td>0</td>
<td>0.62</td>
</tr>
<tr>
<td>2011-01-01T01:04:51Z</td>
<td>bieberfever.com</td>
<td>google.com</td>
<td>Male</td>
<td>USA</td>
<td>1</td>
<td>0.45</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2011-01-01T01:00:00Z</td>
<td>ultratrimfast.com</td>
<td>google.com</td>
<td>Female</td>
<td>UK</td>
<td>0</td>
<td>0.87</td>
</tr>
<tr>
<td>2011-01-01T02:00:00Z</td>
<td>ultratrimfast.com</td>
<td>google.com</td>
<td>Female</td>
<td>UK</td>
<td>0</td>
<td>0.99</td>
</tr>
<tr>
<td>2011-01-01T02:00:00Z</td>
<td>ultratrimfast.com</td>
<td>google.com</td>
<td>Female</td>
<td>UK</td>
<td>1</td>
<td>1.53</td>
</tr>
</tbody>
</table>
## Rollup on ingestion

<table>
<thead>
<tr>
<th>timestamp</th>
<th>publisher</th>
<th>advertiser</th>
<th>gender</th>
<th>country</th>
<th>impressions</th>
<th>clicks</th>
<th>revenue</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011-01-01T01:00:00Z</td>
<td>ultratrimfast.com</td>
<td>google.com</td>
<td>Male</td>
<td>USA</td>
<td>1800</td>
<td>25</td>
<td>15.70</td>
</tr>
<tr>
<td>2011-01-01T01:00:00Z</td>
<td>bieberfever.com</td>
<td>google.com</td>
<td>Male</td>
<td>USA</td>
<td>2912</td>
<td>42</td>
<td>29.18</td>
</tr>
<tr>
<td>2011-01-01T02:00:00Z</td>
<td>ultratrimfast.com</td>
<td>google.com</td>
<td>Male</td>
<td>UK</td>
<td>1953</td>
<td>17</td>
<td>17.31</td>
</tr>
<tr>
<td>2011-01-01T02:00:00Z</td>
<td>bieberfever.com</td>
<td>google.com</td>
<td>Male</td>
<td>UK</td>
<td>3194</td>
<td>170</td>
<td>34.01</td>
</tr>
</tbody>
</table>
Druid Segments

- Can be built from streams
- Immutable once built: no contention between reads and writes
- Simple parallelization: one thread scans one segment
- Streaming append + atomic batch replace
- Want to avoid having a unique key for messages
Druid: Batch ingestion

- Exactly-once, from Hadoop
- Uses atomic replacement
Druid: Stream ingestion

Events

{time: 1440000000000, user: alice, page: /foo, count: 2}
{time: 1440000000000, user: alice, page: /bar, count: 1}
{time: 1440000000000, user: bob, page: /bar, count: 1}

Row Buffer
in-memory
limited in size
grouped on dimensions
Druid: Stream ingestion

Events → [144e10, 144e10, 144e10] [alice, alice, bob] [/foo, /bar, /bar] [2, 1, 1] → Column Store
memory-mapped persisted async from row buffer
Druid: Stream ingestion

{time: 14500000000000, user: carol, page: /baz, count: 1}

Events

[144e10, 144e10, 144e10]
[alice, alice, bob]
[/foo, /bar, /bar]
[2, 1, 1]

Reads
use row buffer and all column stores
Druid: Stream ingestion

Final persist
all data now in column stores

```
[144e10, 144e10, 144e10]
[alice, alice, bob]
[/foo, /bar, /bar]
[2, 1, 1]
```

```
[145e10]
[carol]
[/baz]
[1]
```
Druid: Stream ingestion

Merge all data in a single segment queried along with all existing data target size 500MB–1GB

[144e10, 144e10, 144e10]
alice, alice, bob
[/foo, /bar, /bar]
[2, 1, 1]

[145e10]
carol
[/baz]
[1]
Druid: Stream push

Druid-aware embedded client (tranquility)

Any Stream

Task #0a

Task #0b

Task #1a

Task #1b
Druid: Kafka pull

Partition #0

Partition #1

Partition #2

Partition #3

Task #0

Task #1
Druid: Kafka pull (current)

Events ➔ Kafka Firehose

Kafka Firehose
- uses Kafka high-level consumer
- (+) commit offsets when persisted to disk
- (+) easy to scale ingestion up and down
- (-) not HA
- (-) can generate duplicates during rebalances

Kafka Firehose ➔ Task #N
Druid: Kafka pull (next-gen)

New Kafka Ingestion
- uses Kafka simple or new consumer
- (+) store offsets along with Druid segments
- (+) easy to scale ingestion up and down
- (+) HA—control over who consumes what
- (+) no duplicates during rebalances
Do try this at home
Software

- Druid - [druid.io](http://druid.io) - @druidio
- Kafka - [kafka.apache.org](http://kafka.apache.org) - @apachekafka
- Samza - [samza.apache.org](http://samza.apache.org) - @samzastream
- Storm - [storm.apache.org](http://storm.apache.org) - @stormprocessor
Take aways

- Databases and streams are best friends
- Consider latency and correctness in system design
- Know the guarantees provided by your tools
- Have a backfill strategy if you’re interested in historical data
Thanks!

@implydata
@druidio
@gianmerlino

imply.io
druid.io