

Storing State Forever: Why It Can Be Good For Your Analytics

Yaroslav Tkachenko



 Hi, I'm Yaroslav

Staff Data Engineer @ Shopify (Data Platform: Stream Processing)

Software Architect @ Activision (Data Platform)

Engineering Lead @ Mobify (Platform)

Software Engineer → Director of Engineering @ Bench Accounting
(Web Apps, Platform)



sap1ens



sap1ens.com

*A Story About **150+** Task Managers,
13 TB of State and a Streaming Join
of **9** Kafka Topics...*

**Shopify creates the best commerce tools for
anyone, anywhere, to start and grow a business.**

1.7 Million+

NUMBER OF MERCHANTS

~175 Countries

WITH MERCHANTS

~\$356 Billion

TOTAL SALES ON SHOPIFY

7,000+

NUMBER OF EMPLOYEES

Overview dashboard

Oct 1, 2017 – Oct 7, 2017 compared to Sep 24 – Sep 30, 2017

Total sales

\$30,303.64

[View report](#)

↑ 33%

Point of Sale	\$23,997.93	↑ 37%
Online Store	\$4,661.30	↑ 23%
Facebook	\$1,644.41	↑ 6%

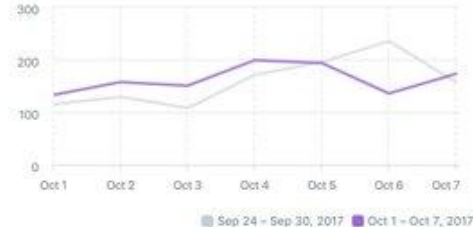
SALES OVER TIME



Average order value

\$159.14

↓ 4%



Sales by POS location

[View report](#)

Kingston	\$13,325.98	↑ 10%
Ottawa	\$10,671.95	↑ 97%

Top products by units sold

[View report](#)

T-shirt	8	↑ 33%
Coat	6	↑ 50%
Tote bag	6	↑ 500%
Necklace	5	↑ 150%
Coasters	5	↓ 17%

Total orders

194

[View report](#)

↑ 36%

ORDERS OVER TIME



Sales by staff

[View report](#)

John Smith	\$18,494.83	↑ 27%
Rebecca Brown	\$2,189.94	↑ 34%
Daniel Woods	\$1,819.30	-
Carys Mills	\$1,493.86	↑ 12%
Ben Courtice	\$2,691.81	↑ 28%

[Learn more about the overview dashboard.](#)

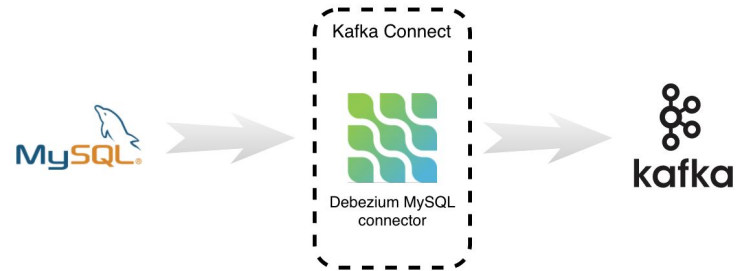
The Sales Model

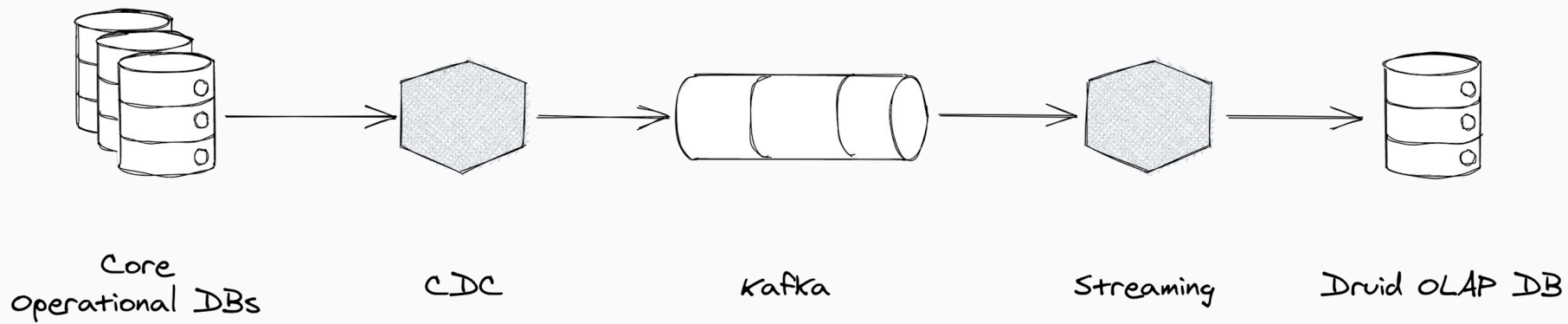
- One of the most important user-facing analytical models at Shopify
- Powers many dashboards, reports and visualizations
- Implemented with Lambda architecture and custom rollups, which means:
 - Data can be delayed: some inputs are powered by batch and some by streaming
 - Batch run is needed to correct some inconsistencies
 - As a result, it can take up to 5 days for correct data to be visible
 - Query time can vary a lot, rollups are used for the largest merchants

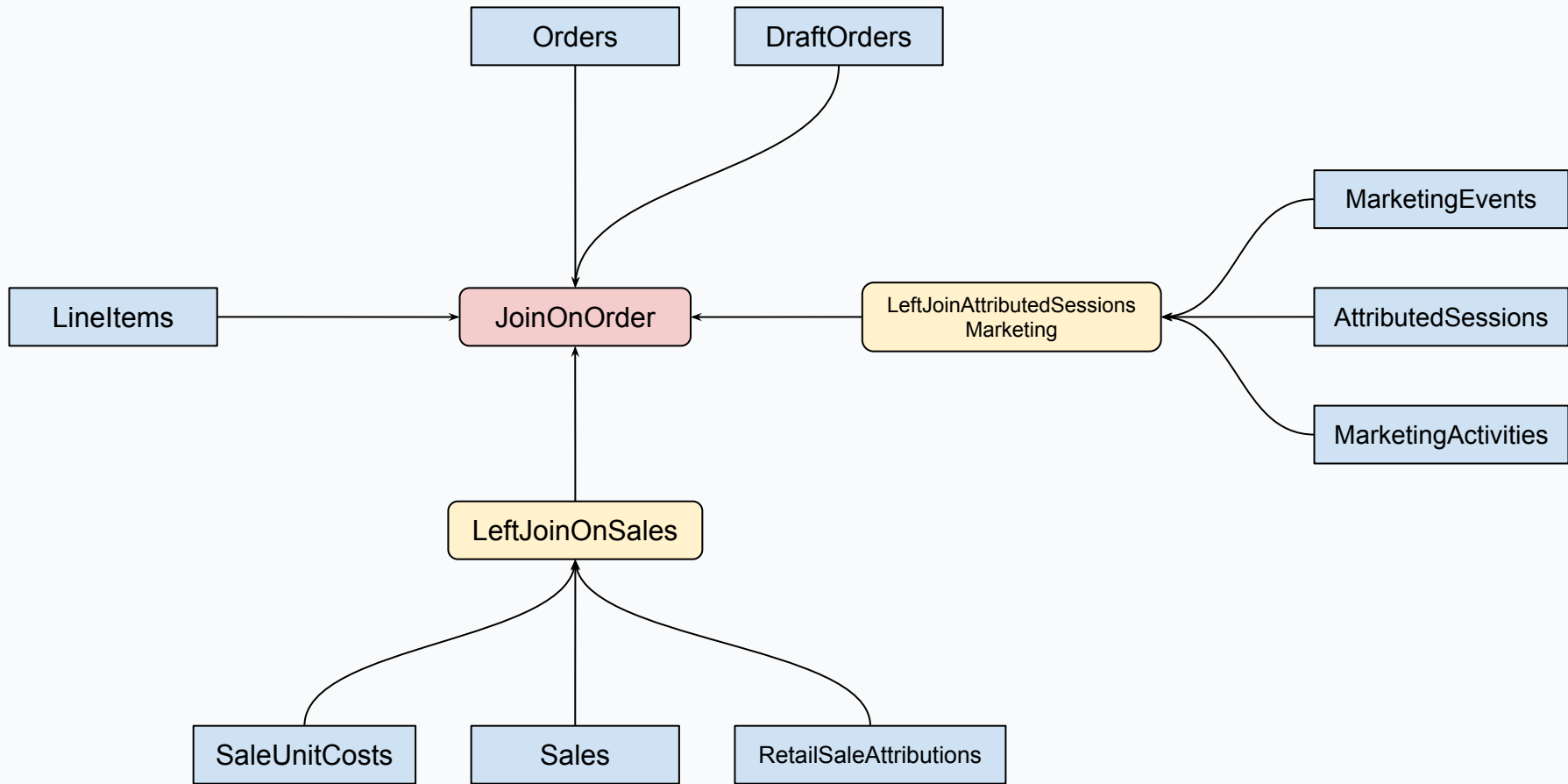
```
SELECT ...  
FROM sales  
LEFT JOIN orders ON orders.id = sales.order_id  
LEFT JOIN locations ON locations.id = orders.location_id  
LEFT JOIN customers ON customers.id = orders.customer_id  
LEFT JOIN addresses AS billing_addresses ON billing_addresses.id = orders.billing_address_id  
LEFT JOIN addresses AS shipping_addresses ON shipping_addresses.id = orders.shipping_address_id  
LEFT JOIN line_items ON line_items.id = sales.line_item_id  
LEFT JOIN attributed_sessions ON attributed_sessions.order_id = sales.order_id  
LEFT JOIN draft_orders ON draft_orders.active_order_id = sales.order_id  
LEFT JOIN marketing_events ON marketing_events.id = attributed_sessions.marketing_event_id  
LEFT JOIN marketing_activities ON marketing_activities.marketing_event_id = marketing_events.id  
LEFT JOIN sale_unit_costs ON sale_unit_costs.sale_id = sales.id  
LEFT JOIN retail_sale_attributions ON retail_sale_attributions.sale_id = sales.id  
LEFT JOIN users AS retail_users ON retail_sale_attributions.user_id = retail_users.id
```

The Sales Model in a nutshell

Change Data Capture







The Streaming Sales Model Requirements

- Streaming joins
- Low latency
- Arbitrarily late-arriving updates for any side of a join

The Streaming Sales Model Requirements

- Streaming joins
- Low latency
- **Arbitrarily late-arriving updates for any side of a join**



[Arbitrarily] Late-Arriving Updates

- Order edits
- Order imports
- Order deletions
- Refunds
- Session attribution

Order with
ID 123
Created
at 3:15pm



Join &
Emit!



Sale with
Order ID 123
Created
at 3:16pm

1 hour

Order with
ID 123
Updated
at 4:30pm



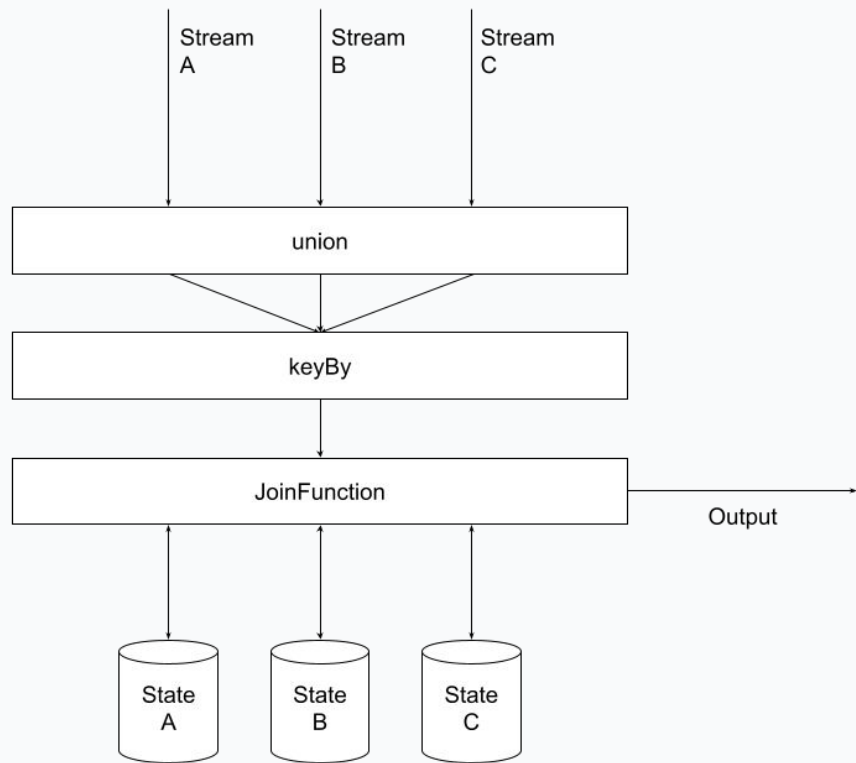
???

1 hour

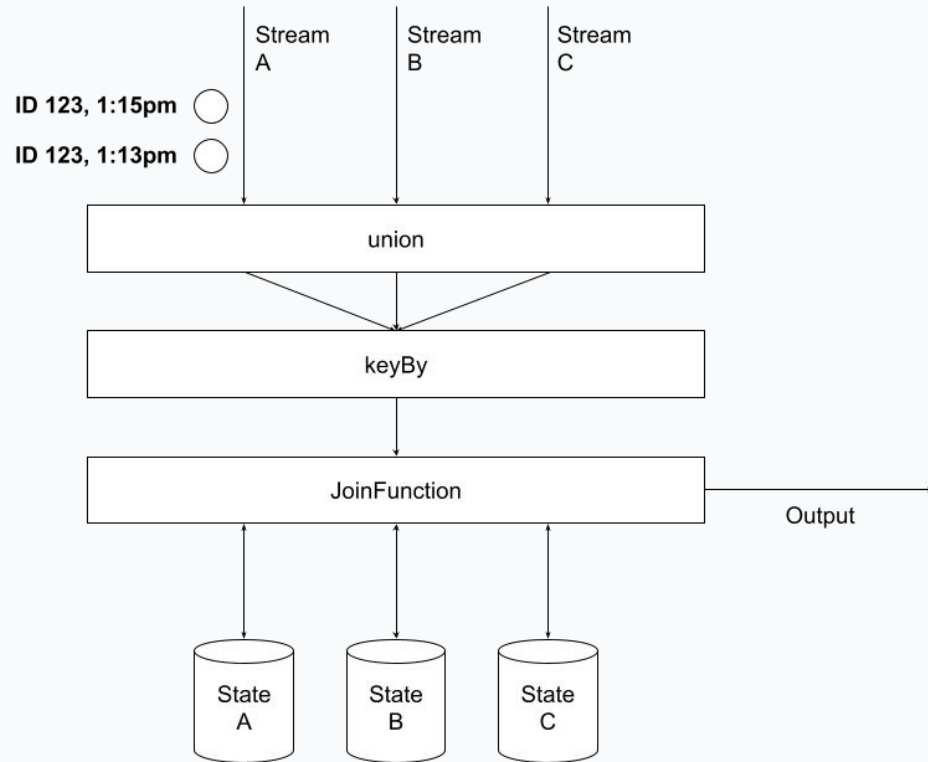
Join with fixed windows

```
1 stream.join(otherStream)
2     .where(<KeySelector>)
3     .equalTo(<KeySelector>)
4     .window(<WindowAssigner>)
5     .apply(<JoinFunction>)
```

Standard joins wouldn't work



Non-Temporal Join



Non-Temporal Join

```
1 val result = salesStream
2     .union(saleUnitCostsStream, retailStream)
3     .keyBy(_.sale_id)
4     .process(new JoinFunction())
```

Union to combine 2+ streams

```
1 class JoinFunction extends KeyedProcessFunction[...] with LatestRecordUpdater {
2   @transient private var latestStateSale: ValueState[SaleRecord] = _
3   // 2 more variables + 'open' for initialization
4
5   override def processElement(
6     value: Input, ctx: KeyedProcessFunction[...]#Context, out: Collector[Output]
7   ): Unit = {
8     val Input(currentSale, currentSaleUnitCost, currentRetailSaleAttribution) = value
9
10    val actualSale =
11      getAndUpdateRecord(latestStateSale, currentSale)
12
13    val actualSaleUnitCost =
14      getAndUpdateRecord(latestStateSaleUnitCost, currentSaleUnitCost)
15
16    val actualRetail =
17      getAndUpdateRecord(latestStateRetail, currentRetail)
18
19    actualSale foreach { sale =>
20      out.collect(output(sale, actualSaleUnitCost, actualRetail))
21    }
22  }
23 }
```

```
1 def getAndUpdateRecord[T <: Updatable](  
2   latestState: ValueState[T], currentRecord: Option[T]  
3 ): Option[T] = {  
4   val stateRecord = latestState.value()  
5   val records = List(currentRecord, Option(stateRecord)).flatten  
6  
7   if (records.isEmpty) {  
8     None  
9   } else {  
10    val actualLatestRecord = records.maxBy(extractTimestamp)  
11    if (actualLatestRecord != stateRecord) {  
12      latestState.update(actualLatestRecord)  
13    }  
14    Some(actualLatestRecord)  
15  }  
16 }  
17  
18 private def extractTimestamp[T <: Updatable](record: T): Long = {  
19   record.updated_at.getOrElse(MissingTimestamp)  
20 }
```

Custom Non-Temporal Join

- Union operator to combine 2+ streams (as long as they have the same id to key by)
- *KeyedProcessFunction* to store state for all sides of the join
- Special timestamp field (*updated_at*) to make sure the latest version is always emitted
- Can use *StateTtlConfig* or Timers to garbage collect state, but...

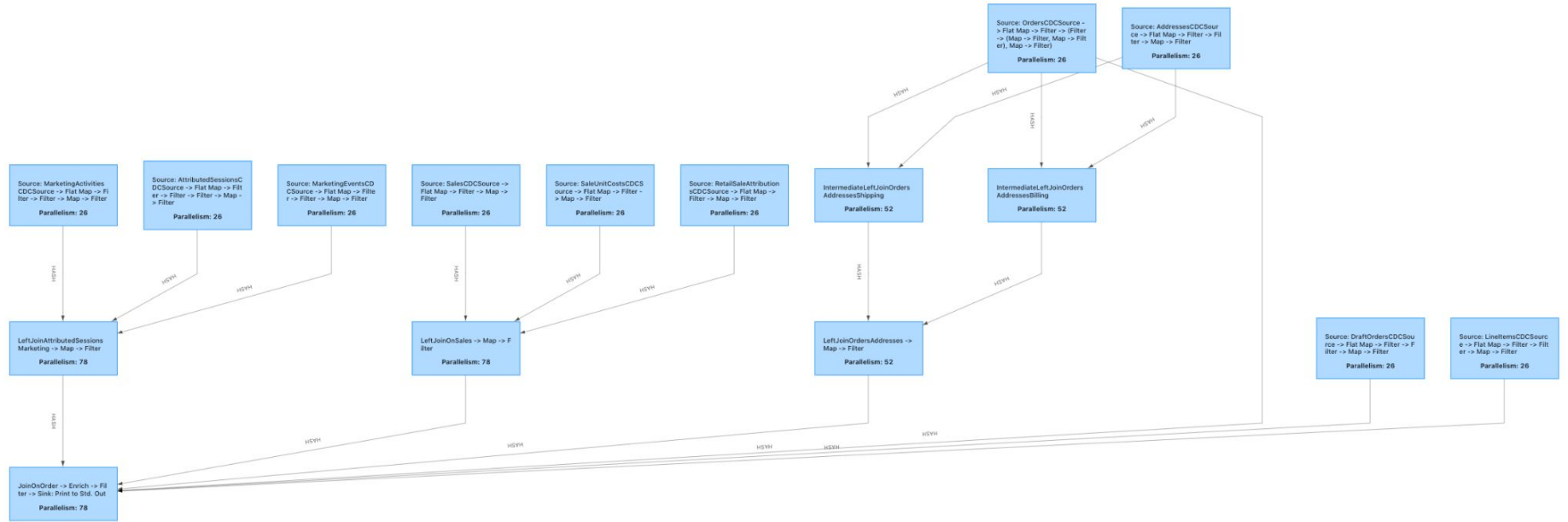
*But what if... We keep the state
indefinitely? Is stateful Flink
pipeline that different from Kafka?
Or even MySQL?*

	Typical Data Store	Stateful Flink App
Scalability	✓	✓
Fault-tolerance	✓	✓
APIs	✓	✓

It turns out that it's not that different...

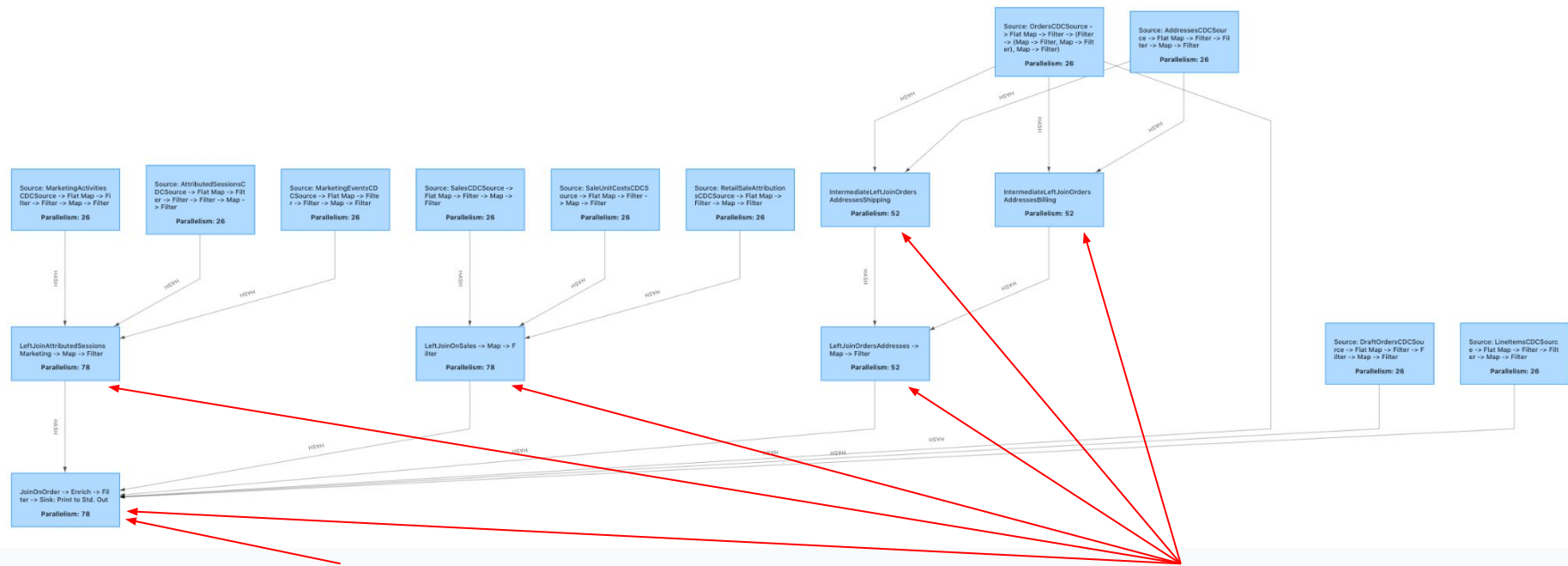
The Experiment

- Implement the Sales model topology using *union + JoinFunction*
- Ingest ALL historical Shopify data (tens of billions CDC messages)
- Store ALL of them (no state GC)
- Strategy: find a bottleneck, fix it, rinse & repeat
- Setup:
 - 156 Task Managers: 4 CPU cores, 16 GB RAM, 4 task slots each
 - Maximum operator parallelism: 624
 - Running in Kubernetes (GKE)
 - Using Flink 1.12.0
 - Using RocksDB
 - Using GCS for checkpoint and savepoint persistent locations



Pipeline steps





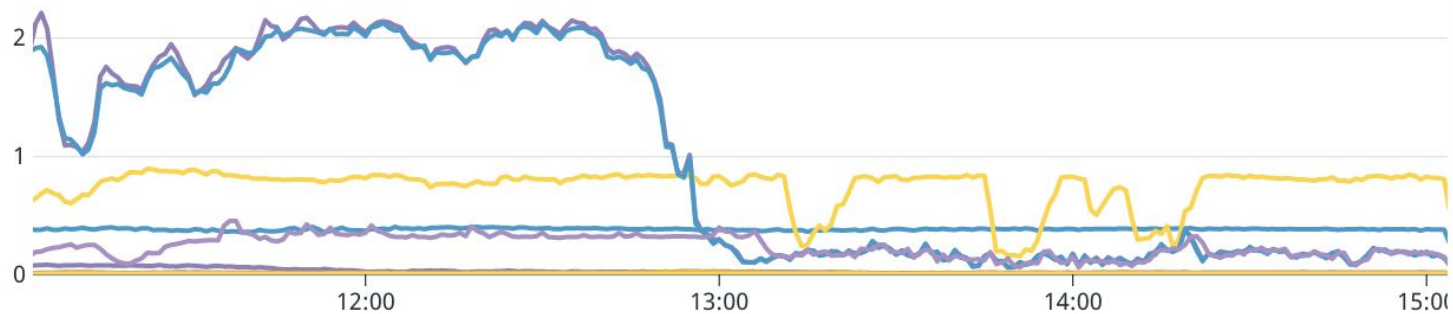
Final Join

Joins

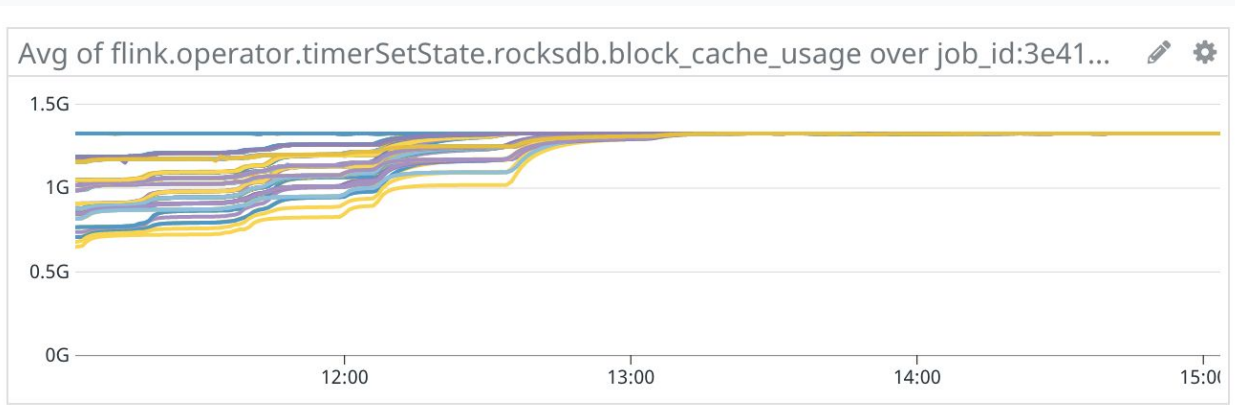
Pipeline steps



Bytes Out By Task



First problem



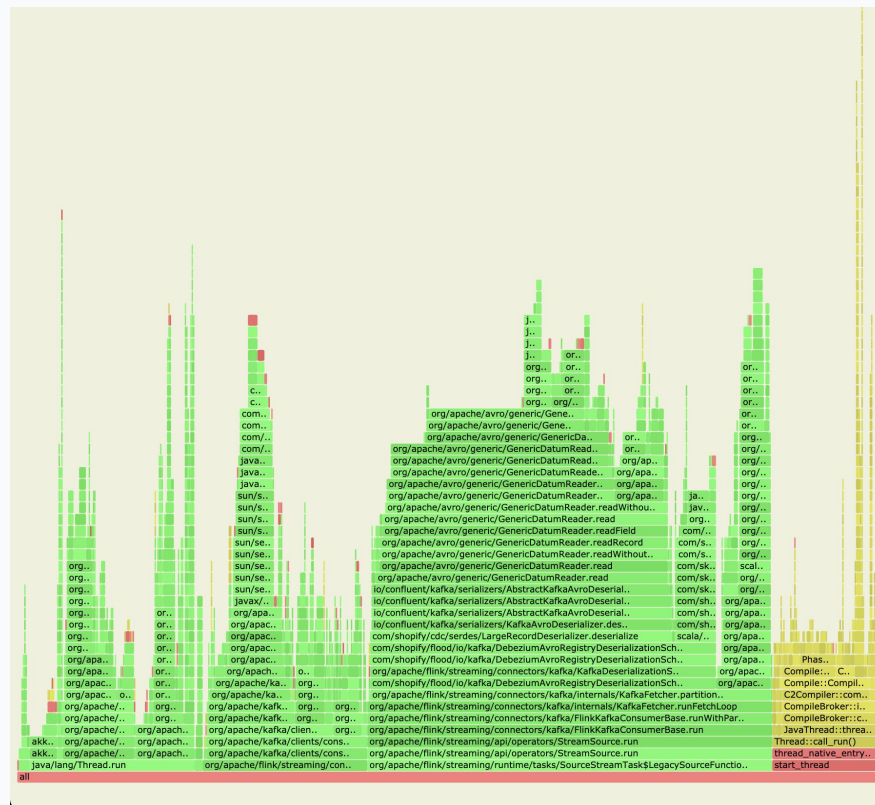
Reaching max RocksDB block cache

Solution

- **Switching to GCP Local SSDs**
- Flink configuration tuning:
 - state.backend.fs.memory-threshold: 1m
 - state.backend.rocksdb.thread.num: 4
 - state.backend.rocksdb.checkpoint.transfer.thread.num: 4
 - state.backend.rocksdb.block.blocksize: 16KB
 - state.backend.rocksdb.block.cache-size: 64MB
 - state.backend.rocksdb.predefined-options: FLASH_SSD_OPTIMIZED



env.getConfig.disableGenericType()



After switching to case class serialization

Results

- It took **30+ hours** to backfill (years of data)
 - Time depends on your Kafka setup a lot (# of partitions, locality)
- Savepoint in the steady state: **13 TB**
- **No performance degradation** during or after backfill
- **100% correct** when compared with the existing system
- Time to take a savepoint: **~30 minutes**
- Time to recover from a savepoint: **<30 minutes**

What If?

Global Window

- Similar semantics
- Less control over state (no StateTtlConfig or Timers)
- Bad previous experience related to Apache Beam

External State Lookups

- Much slower
- Can get complicated very quickly

*So... Will you actually keep all this
state around?*

Next Steps

- Scaling state is not the biggest problem, but savepoint recovery time is 😐
- Introducing state optimizations:
 - Apparently some sources are immutable (e.g. append-only sales ledger), so:
 - Accumulate all sides of the join
 - Emit the result and clear the state
 - Product tradeoffs to reduce state (e.g. only support deletes in the last month)
- Our dream:
 - State with TTL
 - When receiving late-arriving record without accumulated state: backfill state for the current key and re-emit the join 🥳

Questions?

Twitter: @sap1ens

