

Dynamic Change Data Capture with Flink CDC and Consistent Hashing



Hi!



Xiao Meng

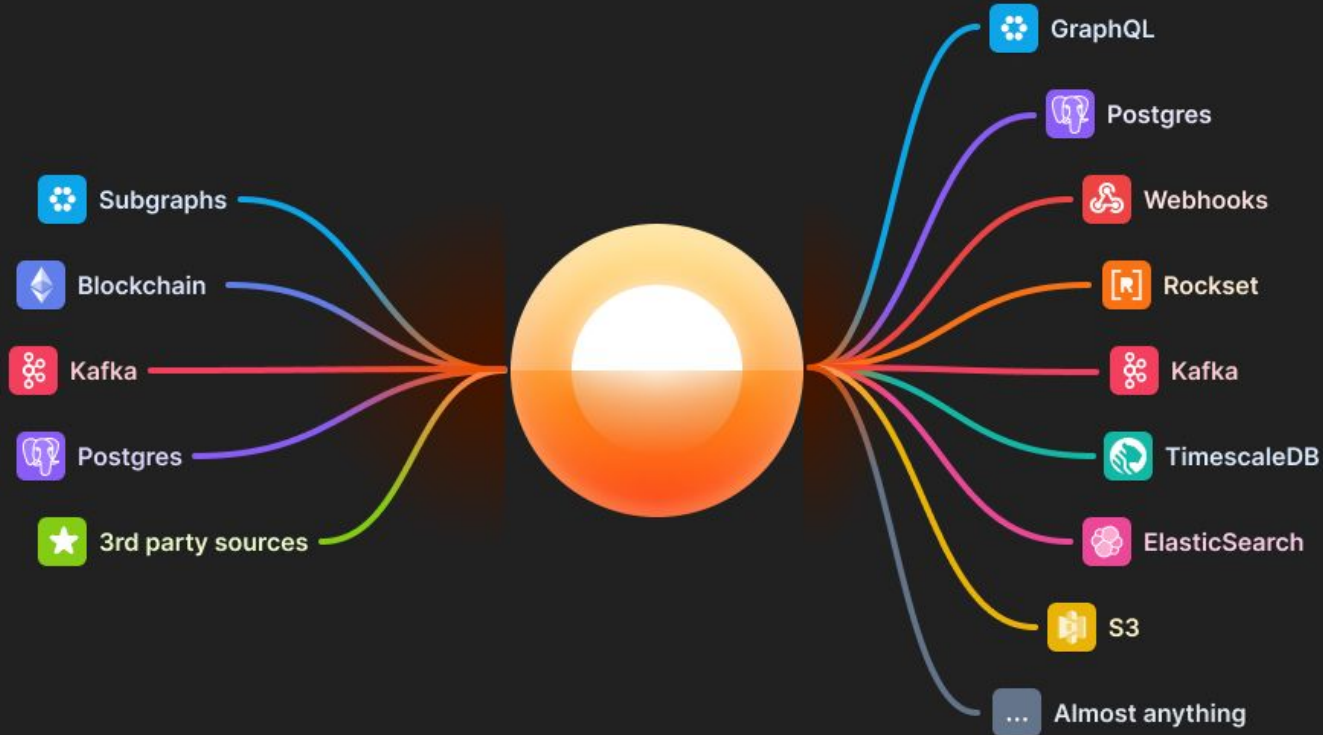
- Software Engineer @ Goldsky
- Expert Data Engineer @ Activision
- Data, Infrastructure, and SRE



Yaroslav Tkachenko

- Founding Engineer @ Goldsky
- Staff Data Engineer @ Shopify
- Software Architect @ Activision
- Kafka, Flink, and all things streaming!

Goldsky





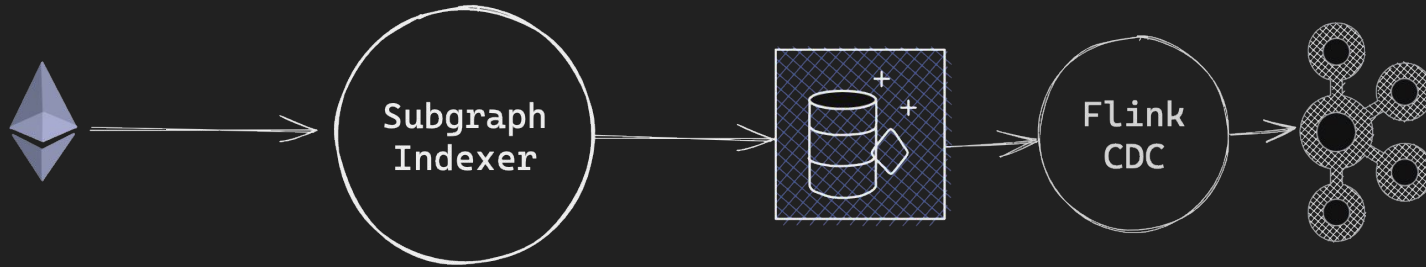
Use case

- `goldsky subgraph deploy poap-subgraph/1.0.0 --path .`
- `goldsky pipeline create def.json poap-subgraph-pipeline`

Indexing Blockchain Data



EVM event stream



GraphQL Manifest

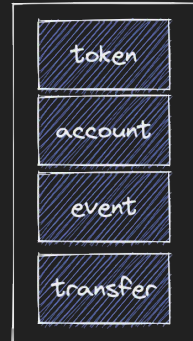
schema.graphql

```
type Token @Entity
type Account @Entity
type Event @Entity
type Transfer @Entity
```

schema



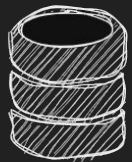
tables



Thousands of Schemas!



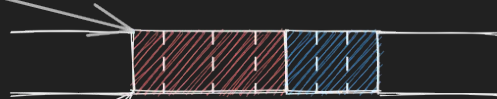
Postgres CDC 101



xid (transaction id): 10298

```
BEGIN;  
INSERT INTO data(data) VALUES('1');  
INSERT INTO data(data) VALUES('2');  
COMMIT;
```

Write Ahead Log



Logical Decoding

Log Sequence Number

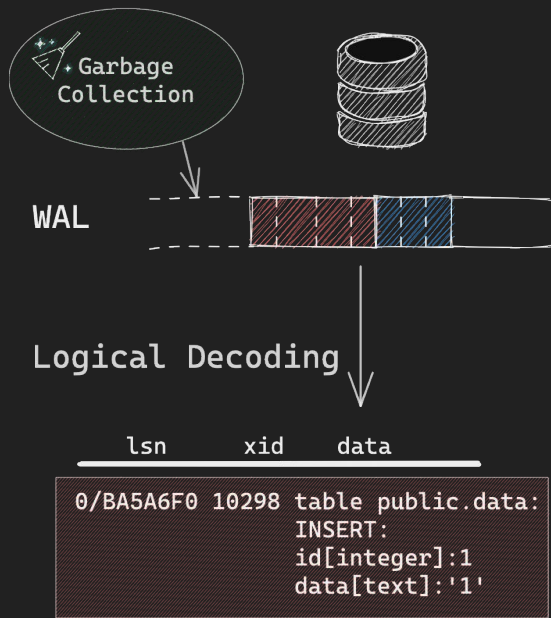
64-bit offset
0 / BA5A688

lsn	xid	data
0/BA5A688	10298	BEGIN 10298
0/BA5A6F0	10298	table public.data:INSERT: id[integer]:1 data[text]:'1'
0/BA5A7F8	10298	table public.data:INSERT: id[integer]:2 data[text]:'2'
0/BA5A8A8	10298	COMMIT 10298



Postgres CDC 101

Flink CDC Consumer



Debezium (Change Event Format)



```
{
  "before": null,
  "after": {
    "id": "1",
    "data": "2"
  },
  "source": {
    "schema": "public",
    "table": "data",
    "txId": 10298,
    "lsn": 0/BA5A688,
  },
  "op": "c",
}
```

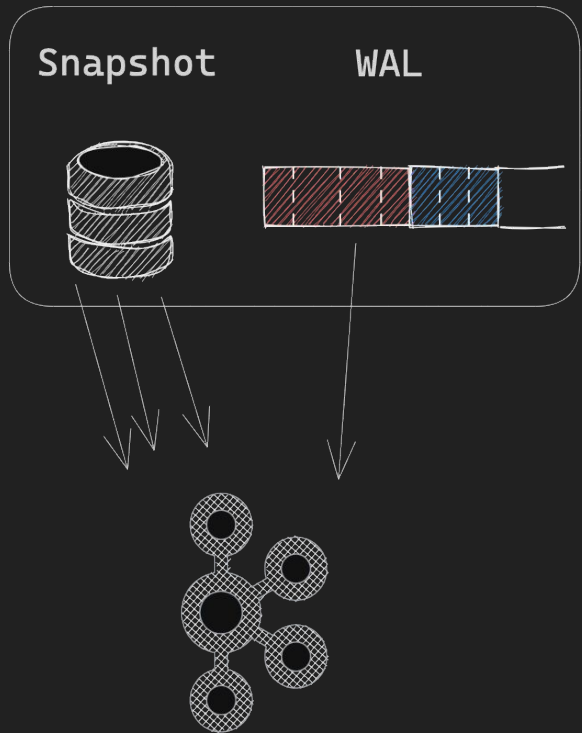
Replication Slot (Tracking "consumer offset")

Replication Slot

restart_lsn, confirmed_flush_lsn

Our Requirements

- Replication Slot Management
 - RDS Aurora: default 20
 - One slot to rule them all: hard to scale
 - One slot per table/job: operational burden
- Snapshot should be lock-free and scalable.
- Dynamic Schema
 - self-serve deployments generate new schema and tables all the time

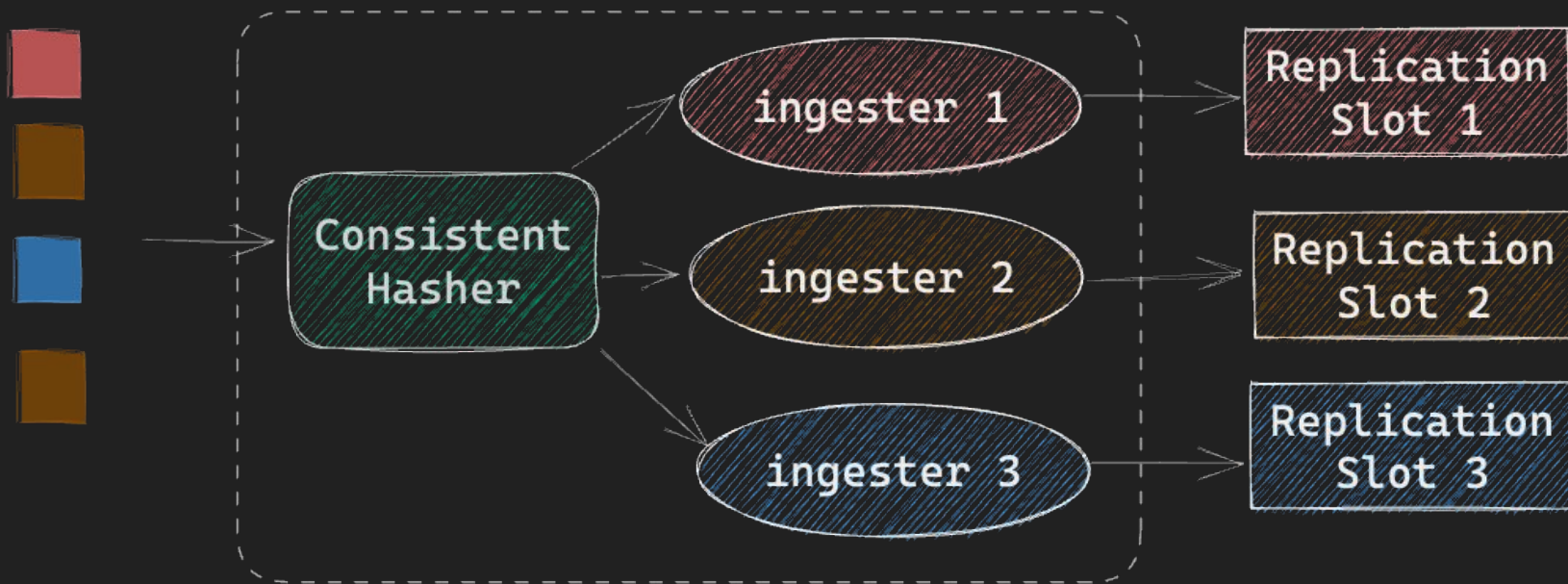


Consistent Hashing & Sharing Replication Slots



new tables

Ingestor Cluster





Our Requirements

-  Replication Slot Management
- **Lock-Free and Parallel Snapshot**
- **Scan Newly Added Tables**

The Journey: Flink-CDC & Goldsky Timeline



Postgres CDC Connector [PR](#)
Scan Newly Added Tables [PR](#)

Exploration

In Production 🐼
config-file driven
by our engineers

Self-Serve CDC
directly driven by
end-users!

2022

Mar

Oct

Nov

Dec

Jan

2023

Jun

Sep



Flink-CDC 2.2

Incremental Snapshot
Framework based on MySQL
(flink-cdc base)
Debezium 1.6.4

Release 2.3

Oracle & MongoDB CDC
Connector

Release 2.4

Postgres CDC PR Merged 🎉
Debezium 1.9.7

Roadmap 2.5

Scan Newly Added Tables [PR](#)

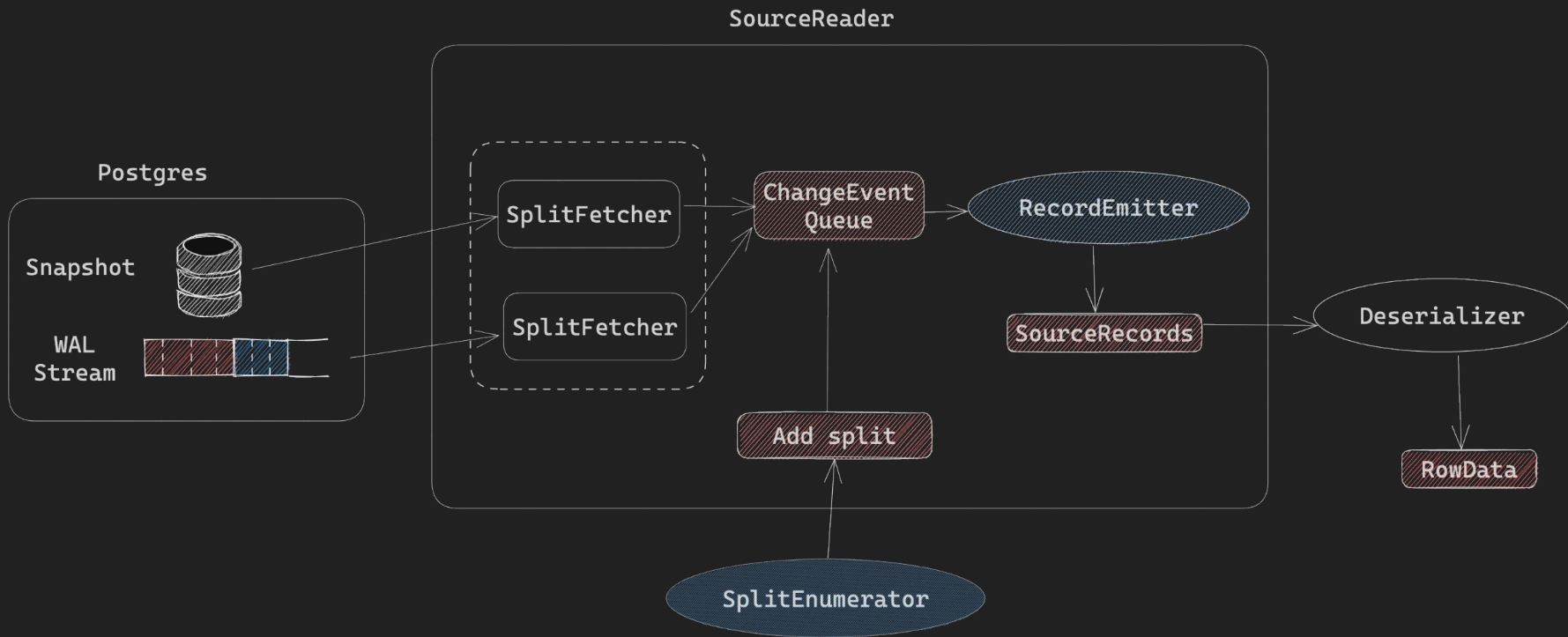


Flink-CDC: SQL/DataStream API

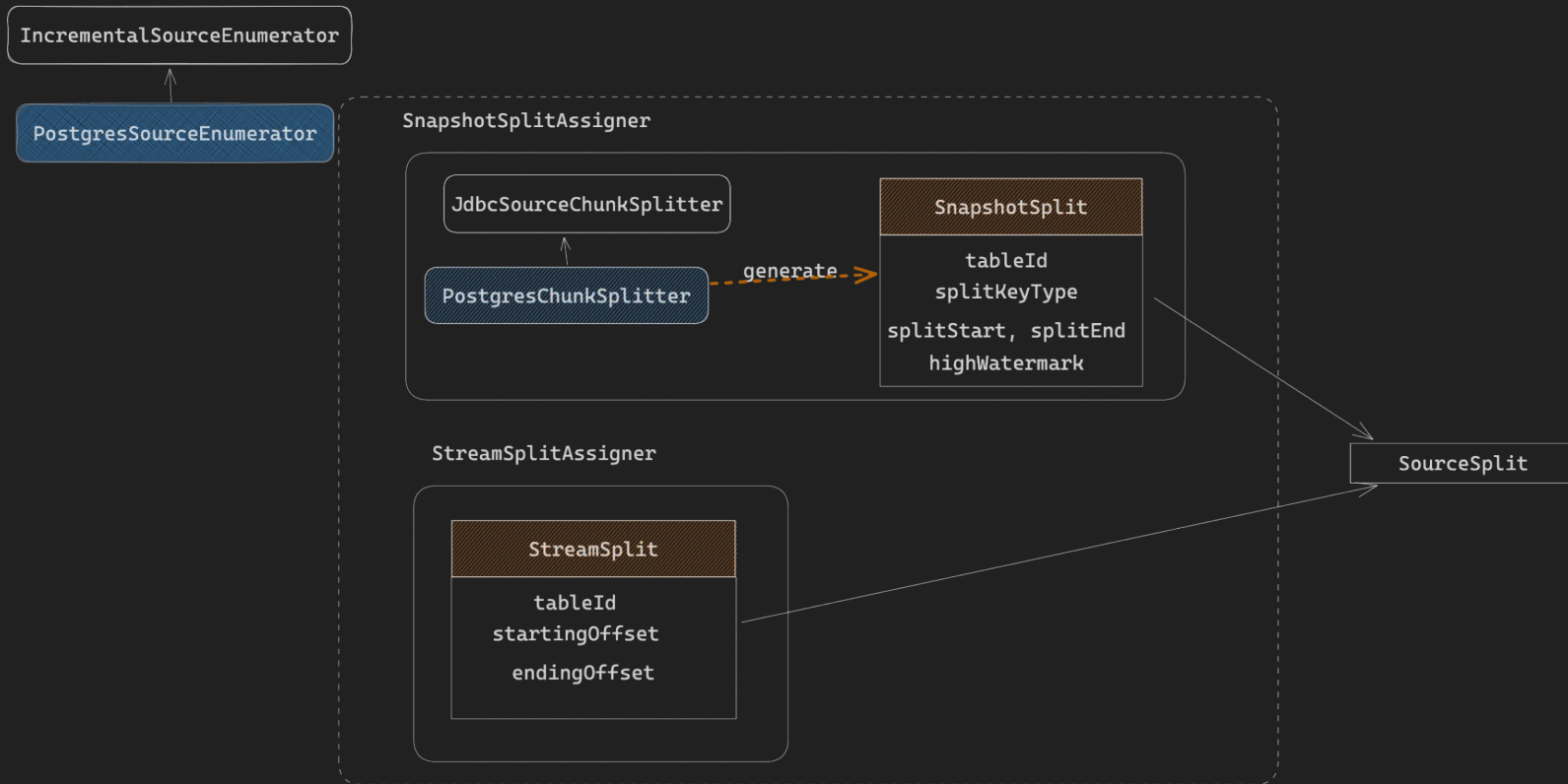
```
CREATE TABLE shipments (  
  shipment_id INT,  
  order_id INT,  
  origin STRING,  
  destination STRING,  
  is_arrived BOOLEAN  
) WITH (  
  'connector' = 'postgres-cdc',  
  'hostname' = 'localhost',  
  'port' = '5432',  
  'username' = 'postgres',  
  'password' = 'postgres',  
  'database-name' = 'postgres',  
  'schema-name' = 'public',  
  'table-name' = 'shipments',  
  'slot.name' = 'flink',  
  'scan.incremental.snapshot.enabled' = 'true'  
);
```

```
JdbcIncrementalSource<String> postgresIncrementalSource =  
  PostgresSourceBuilder.PostgresIncrementalSource.<String>builder()  
    .hostname("localhost")  
    .port(5432)  
    .database("postgres")  
    .schemaList("inventory")  
    .tableList("inventory.products")  
    .username("postgres")  
    .password("postgres")  
    .slotName("flink")  
    .deserializer(deserializer)  
    .includeSchemaChanges(true)  
    .splitSize(2)  
    .build();
```

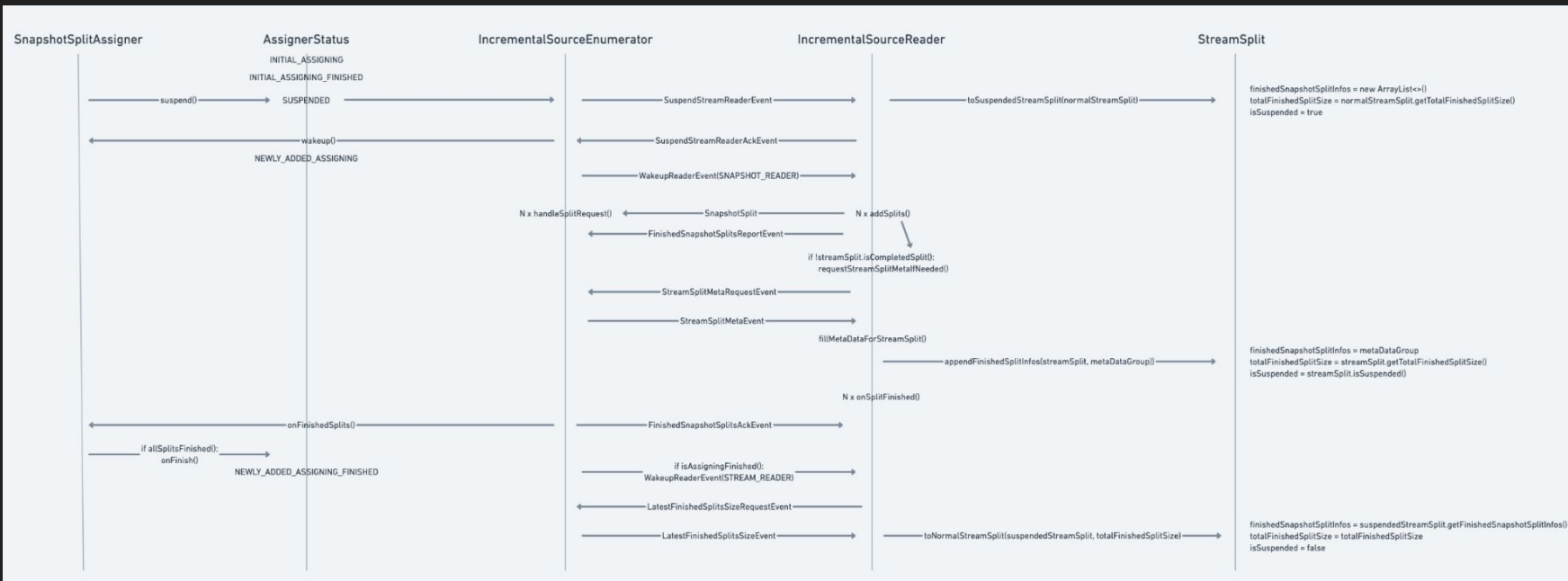
Foundation: FLIP-27 Source Interface & CDC-base



Implementation



Troubleshooting async code







Well, things didn't work!

We noticed that the connector tried to use the replication slot during snapshotting... But *why*?



DBLog: A Watermark Based Change-Data-Capture Framework

Andreas Andreakis
Netflix
aandreakis@netflix.com

Ioannis Papapanagiotou
Netflix
ipapanagiotou@netflix.com

ABSTRACT

It is a commonly observed pattern for applications to utilize multiple heterogeneous databases where each is used to serve a specific need such as storing the canonical form of data or providing advanced search capabilities. For applications it is hence desired to keep multiple databases in sync. We have observed a series of distinct patterns that have tried to solve this problem such as dual-writes and distributed transactions. However, these approaches have limitations with regard to feasibility, robustness, and maintenance. An alternative approach that has recently emerged is to utilize Change-Data-Capture (CDC) in order to capture changed rows from a database's transaction log and eventually deliver them downstream with low latency. In order to solve the data synchronization problem one also needs to replicate the full state of a database and transaction logs typically do not contain the full history of changes. At the same time, there are use cases that require high-availability of the transaction log events so that databases stay as closely in-sync as possible.

To address the above challenges, we developed a novel CDC framework for databases, namely DBLog. DBLog utilizes a watermark based approach that allows us to interleave transaction log events with rows that we directly select from tables to capture the full state. Our solution allows log events to continue progress without stalling while processing selects. Selects can be triggered at any time on all tables, a specific table, or for specific primary keys of a table. DBLog executes selects in chunks and tracks progress, allowing them to pause and resume. The watermark approach does not use locks and has minimum impact on the source. DBLog is currently used in production by tens of microservices at Netflix.

KEYWORDS

databases, replication, change-data-capture

1 INTRODUCTION

Netflix uses hundreds of microservices performing trillions of operations per day in the data layer. Since there is no single database design that fits all the needs, each of the microservices can utilize multiple heterogeneous databases. For example, a service can use MySQL, PostgreSQL, Aurora or Cassandra for the operational data and Elasticsearch for its indexing capabilities. To be able to keep multiple databases in sync we developed a data enrichment and synchronization platform namely Delba [1]. One of the key requirements is to have low propagation delays from the source to the derived stores and that the flow of events is highly available. A key requirement to achieve that is having Change-Data-Capture (CDC) that allows capturing changed rows from a database in near real-time and eventually propagating those rows to downstream consumers [1]. CDC is becoming increasingly popular for use cases that require keeping multiple heterogeneous databases in

sync [8, 12, 16] and addresses challenges that exist with traditional techniques like dual-writes and distributed transactions [15].

In database systems, the transaction log typically has limited retention and it is not guaranteed to contain the full history of changes. Therefore, the full state of a database needs to be captured as well. While operating data synchronization in production at Netflix, we identified some requirements in regards to the full state capture. We wanted to (a) trigger the full state capture at any point in time. That is because the full state may not only be needed initially and may be needed at any time afterwards. For instance if the database is restored from a backup or for repairs if there is data loss or corruption downstream. There are also cases where only a subset of data needs to be repaired, for example if a specific set of rows has been identified to be corrupt downstream. (b) pause or resume at any time so that full state capture does not need to start from the beginning for large tables after restarting the process. (c) capture transaction log events and the full state side by side without stalling one or the other. There are use cases that require high availability of transaction log events so that the replication lag to the source is kept to a minimum. (d) prevent time-travel, by preserving the order of history when transmitting events to a derived datastore. This way an earlier version of a row (like the residential address of a member account) is not delivered after a later version. Hence, a solution had to combine transaction log events and the full state in a way that preserves the history of changes. (e) offer this as a platform. Hence it was crucial to minimize the impact on the source database. Otherwise this can hinder adoption of the platform, especially for use cases that have high traffic. In that regard we want to avoid primitives such as table locks which can block application write traffic. (f) function across a variety of Relational Database Management Systems (RDBMS), such as MySQL, PostgreSQL, Aurora [19] etc, that we use in production. In order to achieve that we wanted to avoid using vendor specific features.

Based on these requirements we developed DBLog. DBLog runs as a process and utilizes a watermark based approach that allows interleaving of transaction log events with rows that we directly select from tables in order to capture the full state of a database. Our solution allows log events to continue progress without stalling while executing selects. Selects can be triggered at any time on all tables, a specific table, or for specific primary keys of a table. DBLog processes selects in chunks and tracks progress in a state store (currently Zookeeper) allowing them to pause and resume from the last completed chunk. The watermark approach does not use table locks and has therefore minimum impact on the source database. DBLog delivers captured events into an output by using the same format regardless if the origin is the transaction log or a table selection. The output can be a stream like Kafka [21] which is a common choice if there is more than one consumer of the events. However DBLog can also write directly to datastores or

Algorithm:

1. stop live event processing
2. read next chunk
3. resume live event processing
4. reconcile the events using watermarks



Postgres IS challenges

Snapshot task can create a live replication one!

- PostgresScanFetchTask →
- SnapshotSplitReadTask (read chunk)
- ? PostgresStreamFetchTask (execute backfill)
- SnapshotSplitReadTask (read chunk)
- ? PostgresStreamFetchTask (execute backfill)
- SnapshotSplitReadTask (read chunk)
- ...



Postgres IS challenges

Backfills are hard to scale.

- Replication slot can't be re-used between snapshot and live modes (it's already active).
- When parallelism > 1, multiple snapshot tasks will try to use the same slot concurrently.
- It's **SLOW**! Unbearably. Backfills require seeking a specific LSN in the WAL and it's very slow when it happens after reading every chunk.

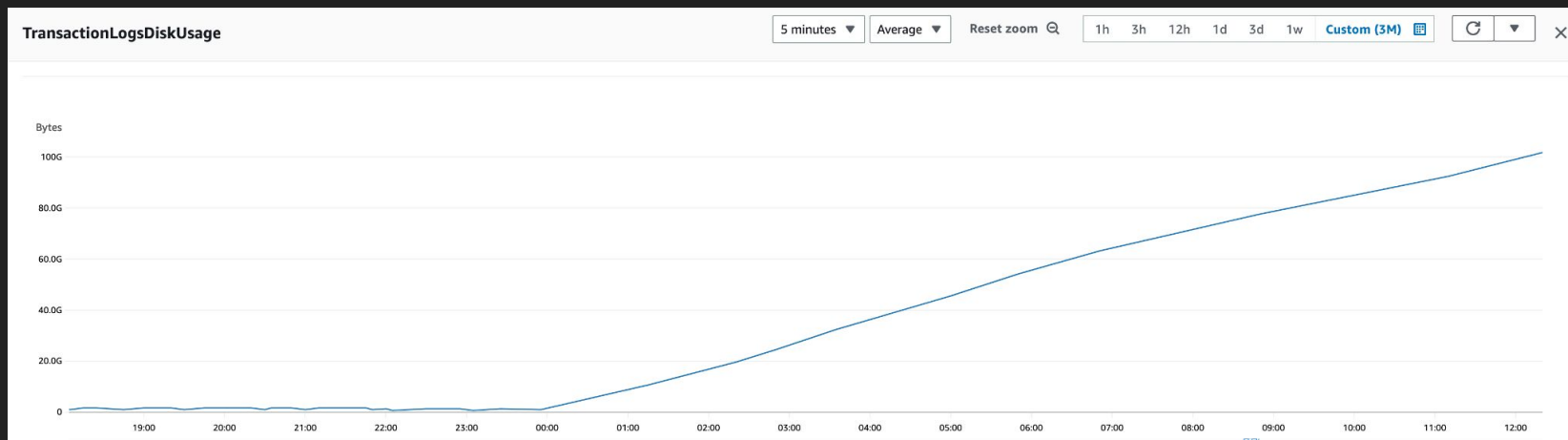


Postgres IS challenges

Backfills are hard to scale.

- You could create a backfill slot per task, but:
 - It's still very slow.
 - You could end up with too many slots! And think about their lifecycle.
 - Can be brittle for large snapshots.

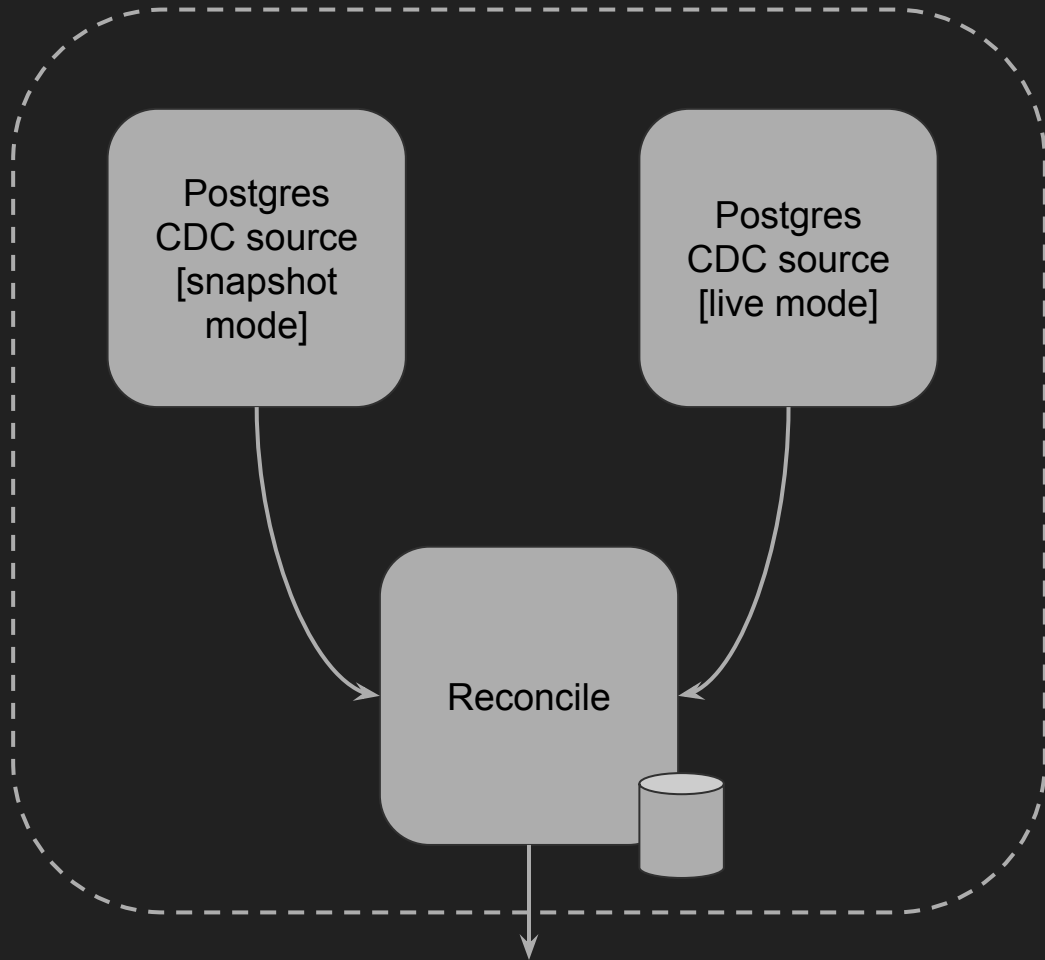
Postgres IS challenges





Flink to the rescue!

What if we re-implement some parts of the incremental snapshotting algorithm with Flink?





Flink to the rescue!

```
1 val rawLiveStream: DataStream[SourceRecord] = source(SnapshotModeNever, 1)
2 val rawSnapshotStream: DataStream[SourceRecord] = source(SnapshotModeInitialOnly, env.getParallelism)
3
4 val serializedLiveStream: DataStream[RawKafkaPayload] = rawLiveStream
5   .flatMap(transformAndSerialize)
6   .setParallelism(1)
7
8 val serializedSnapshotStream: DataStream[RawKafkaPayload] = rawSnapshotStream
9   .flatMap(transformAndSerialize)
10
11 val combinedStream: DataStream[RawKafkaPayload] = serializedSnapshotStream
12   .union(serializedLiveStream)
13   .keyBy(_.key)
14   .flatMap(reconcileFn)
```



Flink to the rescue!

```
1 override def flatMap(payload: RawKafkaPayload, out: Collector[RawKafkaPayload]) = {
2   val operation = DebeziumUtils.extractOpFromHeaders(payload.headers)
3   val isSnapshotData = operation == Envelope.Operation.READ
4
5   if (!isSnapshotData) {
6     seenInLiveStream.update(true)
7     out.collect(payload)
8   } else {
9     if (seenInLiveStream.value() == null) {
10      out.collect(payload)
11    }
12  }
13 }
```

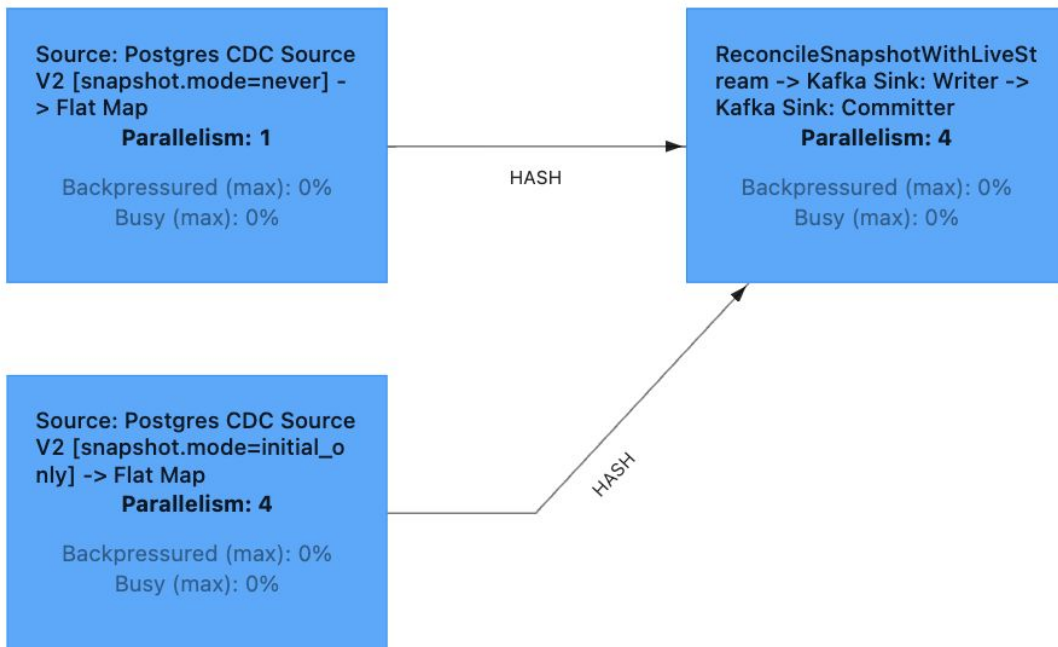


Flink to the rescue!

Reconciliation example:

- K1:  INSERT (L),  UPDATE (L)
- K2:  UPDATE(L),  READ (**S**), ...,  UPDATE(L)
- K3:  READ (**S**),  UPDATE (L)

Flink to the rescue!





Results

- **Fully self-serve, customer-driven**
 - No need to know about Kafka, Kafka Connect, Debezium, CDC, data pipelines, Postgres replication slots, etc...
- **450+** tables being ingested
- **< 3sec** p99 end-to-end latency



Key Takeaways

- Maintaining replication slots can be tricky!
 - Ensure you have the right monitoring & alerting.
- Use consistent hashing for managing a pool of resources.
- Flink has many low-level building blocks that can be used to customize any workflow.
 - You can build your own flavour of incremental snapshotting!



Links

- [DBLog](#): A Watermark Based Change-Data-Capture Framework
- Flink CDC
 - [Demo](#): Streaming ETL for MySQL and Postgres with Flink CDC
 - Postgres Incremental Snapshotting [PR](#)
 - Postgres [Connector](#)



Bonus: dealing with sparse updates

- db1_customerA <= many writes
- db1_customerB <= few writes
- db2_customerC <= no writes

To “unlock” the third database:

```
> SELECT * FROM pg_logical_emit_message(true, 'heartbeat', 'hello');
```



Questions?

@sap1ens

@xiaomeng