## An Exercise in Discovery. Apache Flink CDC & Apache Paimon on S3. Part  1

## Confluent Kafka, Apache Flink + Flink CDC from PostgreSql or MySql => Apache Paimon (Storage via S3) with a Apache Hive Metastore

(7 October 2024)

**Overview**

So, this is a continuation of the previous series. Totally unplanned but honestly not unexpected.

I had a request from a good friend to mix up the inbound data sourcing a bit…

For this series we’re going to modify things a bit. What we’re doing here is splitting that inbound data streams. Let’s face it, not all your data is going to come from one nicely shrink-wrapped source, it is all over the place/business… so ye we’re simulating that with this slight modification.

We still have our imaginary shop where we created our two payloads, SalesBaskets and SalesPayments, previously we posted both payloads onto two [Confluent](https://www.confluent.io/) Kafka topics, avro\_salesBaskets and avro\_salespayments.

This time round.

1. The salesbaskets payload will still go into the Kafka topic, avro\_salesbaskets,
2. The salespayments payload will now be inserted into a database called sales, into a table called salespayments. I show how to do this into either MySql or PostgreSql.

From the [Confluent](https://www.confluent.io/) Kafka Topic (avro\_salesbaskets) the salesbasket payload will be pulled into [Apache Flink](https://flink.apache.org/) via a Apache Flink table definition with a connector=”kafka” definition.

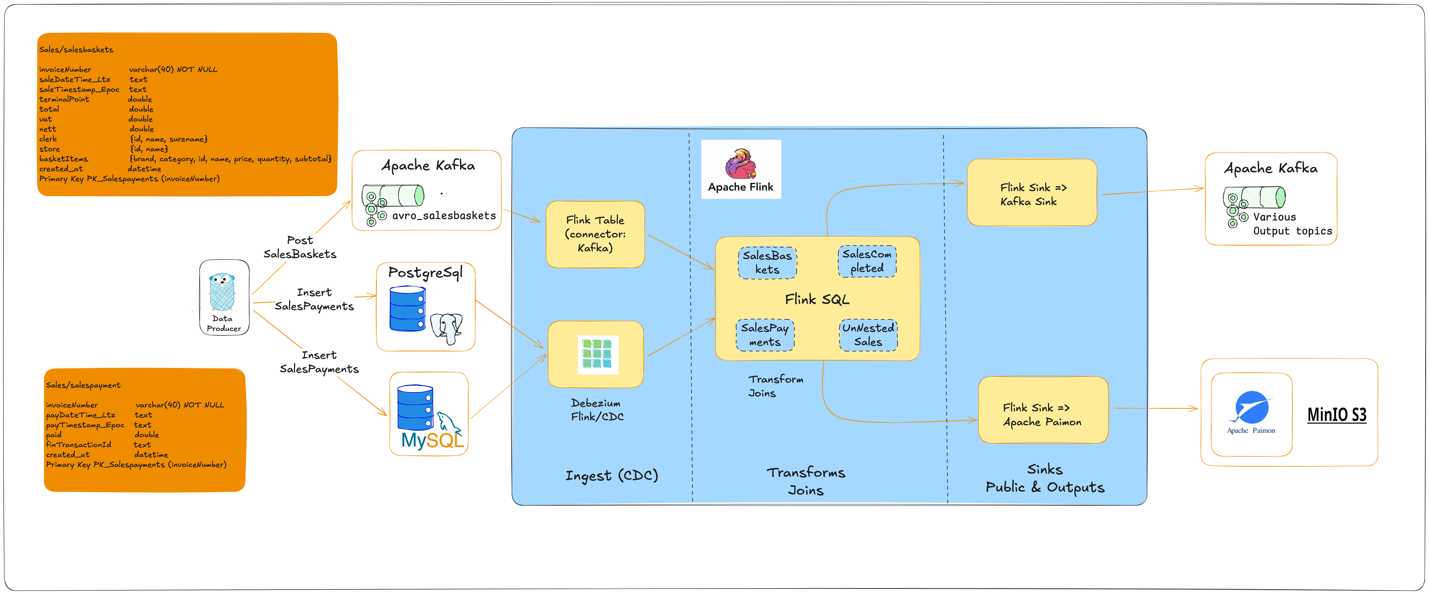
We will then source the salespayments payloads from either [MySql](https://www.mysql.com/) or [PostgreSql](https://www.postgresql.org/) database.

This data sourcing will be accomplished using CDC ([Change Data Capture](https://en.wikipedia.org/wiki/Change_data_capture)). Everyone is aware of this capability on Kafka platforms using their connector framework. Surprise however, [Apache Flink has a rich CDC](https://nightlies.apache.org/flink/flink-cdc-docs-release-3.1/docs/get-started/introduction/) functionality also, utilizing the build in [Debezium](https://debezium.io/) engine.

Currently the GIT repo is configured to use the MySql database, but the PostgreSql example and code is also included, the database used in both cases are called sales and the table: salespayments.

The following two diagrams depicts the above flow described.

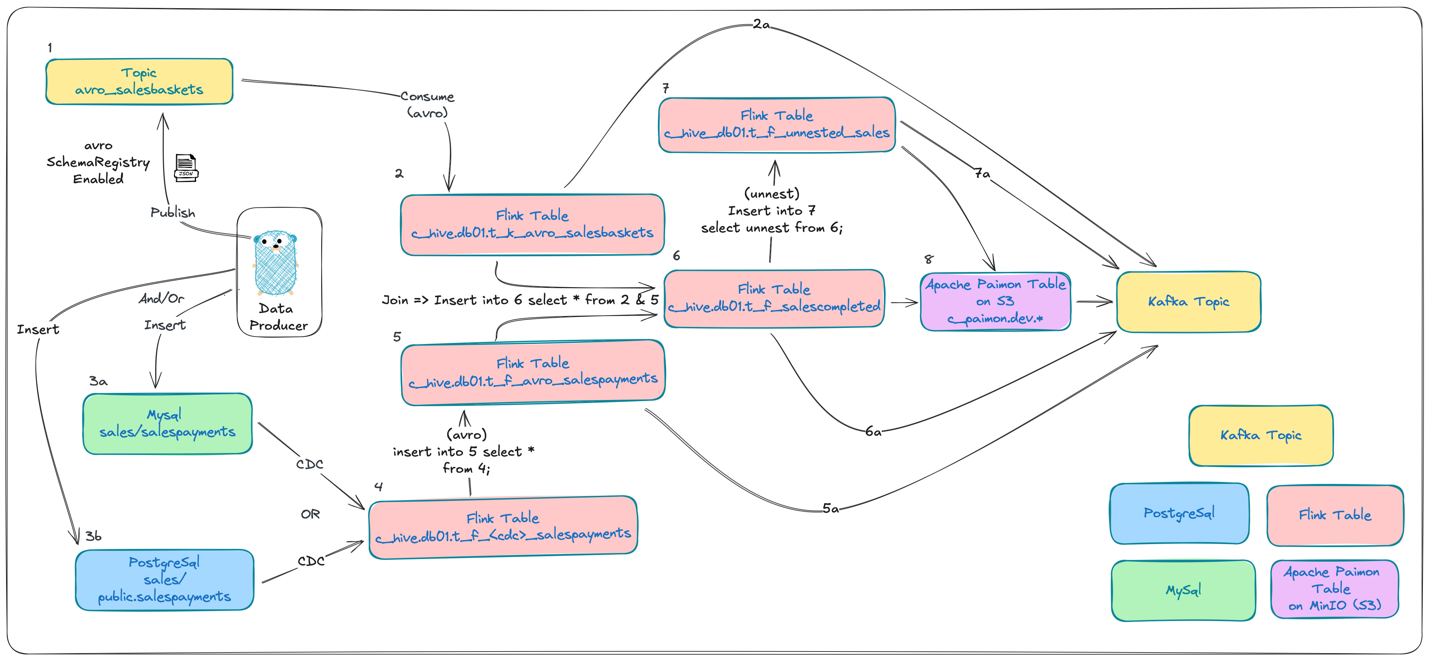
High Level



As previously mentioned, we have 2 payloads, salesbaskets contains the when, where, which clerk serviced us, the what and financial bits of the salesbaskets.

We then have a separate payload salespayments created by the pay point utilized. This payload references the original salesbasket reference number, a datetime of payment, a pay point reference number and the amount processed.

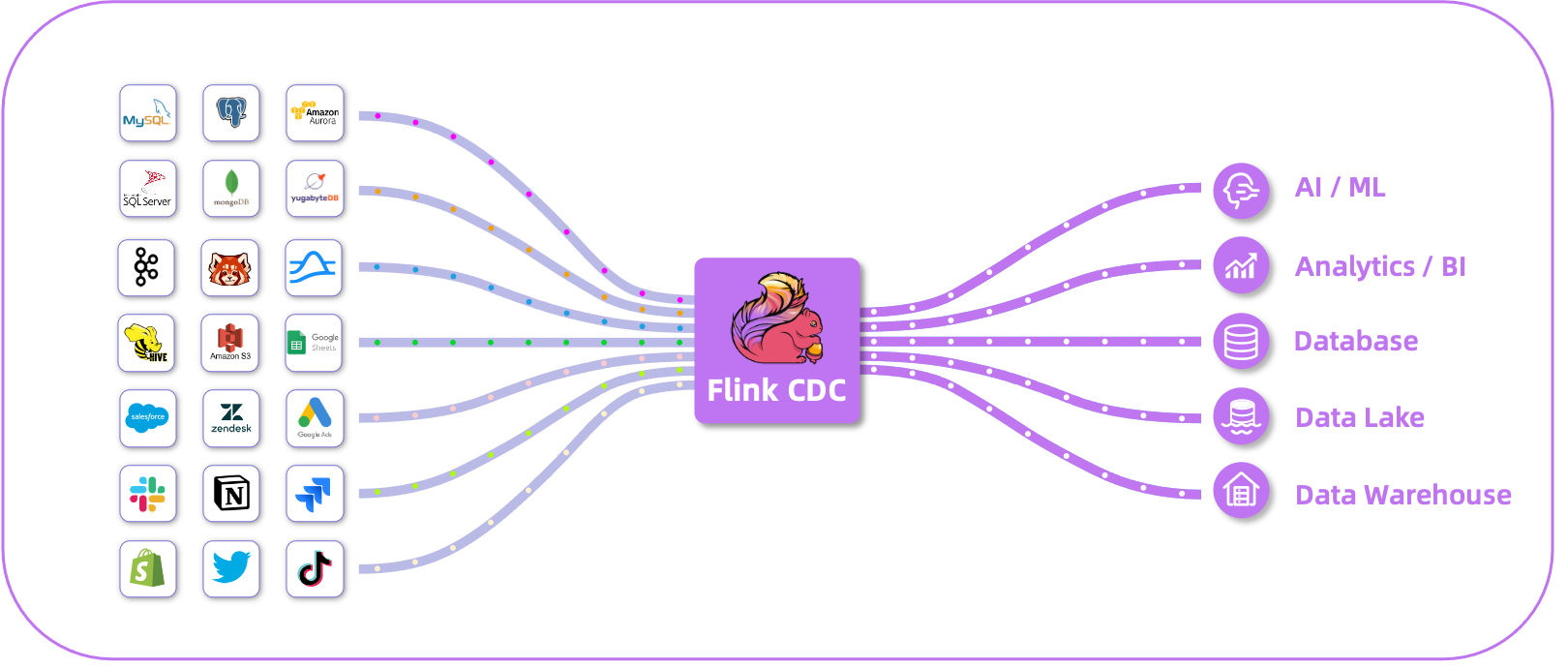
And a little more detail view of our data flow:



The two major new components that come into play in this blog series is the usage of CDC processing and [Apache Paimon](https://paimon.apache.org/) table format with storage on S3 (we’re using a MinIO container for the S3 object storage service, but any AWS S3 compatible object storage service can be used).

Regarding [Apache Paimon](https://paimon.apache.org/) and the S3 usage. You will find that most implementations of [Apache Paimon](https://paimon.apache.org/) will use Apache Hadoop DFS for the storage… but we all know in cloud S3 is big, it’s cheap, it’s pervasive, its available everywhere and easy to consume as a service..

As far as CDC is concerned, the [below diagram](https://nightlies.apache.org/flink/flink-cdc-docs-release-3.1/docs/connectors/flink-sources/overview/) at a very high level shows how rich [Apache Flink CDC](https://nightlies.apache.org/flink/flink-cdc-docs-release-3.1/docs/get-started/introduction/) Source/Target capability is in fact.

[](https://nightlies.apache.org/flink/flink-cdc-docs-release-3.1/docs/connectors/flink-sources/overview/)

**NOTE**: As discovered, the proper functioning of the CDC stack is very very reliant on the various versions of the components deployed, i.e. databases utilized, the associated JAR libraries deployed inside the $FLINK\_HOME/lib directory, and all of this is further driven by the version of Apache Flink deployed.

See [Supported Connectors](https://nightlies.apache.org/flink/flink-cdc-docs-release-3.1/docs/connectors/flink-sources/overview/) for a Compatibility matrix (this matrix directly refers to Apache Flink 1.18.1 as used in this blog, but make sure to reference the correct matrix depending on which version you deploy).

As mentioned, this blog follows a previous series which can be found [here](https://medium.com/@georgelza/an-exercise-in-discovery-streaming-data-in-the-analytical-world-part-1-e7c17d61b9d2), and the associated GIT repo is [here](https://github.com/georgelza/MongoCreator-GoProducer-avro.git). (the previous blog series discussed Apache Iceberg on S3 and a 2nd example of Apache Paimon on HDFS) for storage.

This rabbit hole of discovery however will stand on its own from a code viewpoint and the GIT repo is [here](https://github.com/georgelza/split-sources-in-streaming-world.git).

NOTE: I added a field (subtotal) to the basketitems array of document contained inside the Salesbaskets document, should you be comparing it to the original basket,

The Stack is again comprised out of:

* Confluent Kafka stack
* Apache Flink
* Apache Flink CDC
* Apache Hive Metastore as Catalog
* Apache Paimon as Open table Format
* Apache Avro as Open File Format
* MySql as datasource for Flink CDC
* PostgreSql as a second option as a datasource for Flink CDC
* PostgreSql as datastore for Apache Hive Metastore
* MongoDB as a possible Kafka Sink destination via defined Connector
* Docker
* Docker Compose
* Lots of Makefiles

If you keen to explore Docker, Docker Compose and Makefiles see: [An exercise in Discovery, Building Docker Images, using Makefiles & Docker Compose](https://medium.com/@georgelza/an-exercise-in-discovery-building-docker-images-using-makefiles-docker-compose-part-1-ed89f3546da7)

NOTE: In this project I further mixed it up a bit in the Dockerfiles. I introduced a repo\_name variable pulled in using the arguments primitive, allowing the entire build to be pulled into a personal hub.docker.com account more easily, see the various .env files for the repo\_owner variable.

As it stands this will be a 5-part posting, but it’s by no means complete.

Good luck, this is all fraught with rabbit holes, as always, so many and you can disappear so easily…



**About Me**

I’m a techie, a technologist, always curious, love data, have for as long as I can remember always worked with data in one form or the other, Database admin, Database product lead, data platforms architect, infrastructure architect hosting databases, backing it up, optimizing performance, accessing it. Data data data… it makes the world go round.

In recent years, pivoted into a more generic Technology Architect role, capable of full stack architecture.

[George Leonard](https://www.linkedin.com/in/george-leonard-945b502/)

[georgelza@gmail.com](mailto:georgelza@gmail.com)

------------------------------------------------------------------------------------------------------------------

An Exercise in Discovery. Apache Flink CDC & Apache Paimon on S3. Part 2

## Confluent Kafka, Apache Flink + Flink CDC from PostgreSql or MySql => Apache Paimon (Storage via S3) with a Apache Hive Metastore

(8 October 2024)

**Overview**

Let’s start building.

To build and deploy the stack, we’ll start with the Apache Flink images. As per previous I build an Apache Flink server and a Apache Flink Sql-Client images. As the Java libraries are the same, I’m only referencing the sql-client below.

See devlab/sql-client/Dockerfile for which jar files to include in image build to enable the CDC functionality required. We then build my Apache Flink server images with the same list JAR files. See: devlab/flink/Dockerfile.

Once our above two images are built, we can turn our attention to preparing our database/s. To accomplish this, see the sql/mysqlcdc and sql/postgrescdc directories for the 2 scripts (mysql-init.sql & postgresql-init.sql) executed as part of the container creates of mysqlcdc and postgrescdc services of our docker-compose.yaml.

NOTES: it’s important to include a primary key on your tables in both MySql, PostgreSQL and inside your Apache Flink table definitions.

MySqlDB

* For MySQL CDC into Apache Paimon using the [action jar](https://paimon.apache.org/docs/master/flink/cdc-ingestion/mysql-cdc/#synchronizing-databases) includes:
  + Full database synchronisation
  + Synchronises the schemas. When creating tables via SQL, the schema is static and you might lose the schema synchronisation function.
* For MySql to work you will need to make sure your database is in binary log mode:
  + See: [General FAQ](https://nightlies.apache.org/flink/flink-cdc-docs-master/docs/faq/faq/)
* Make sure to set: SET GLOBAL binlog\_expire\_logs\_seconds = 60000;
  + Ignore the comments to set the following 2 settings, they being depreciated:

show variables like 'expire\_logs\_days';

set global expire\_logs\_days=7;

* Permissions (for our flinkcdc user defined in the create table executed via mysql-init.sql script is defined when we created the user/role via the mysql-init.sql script at container create time) and postgresql-init.sql for PostgreSql.

I originally used the below:

GRANT SELECT, SHOW DATABASES, REPLICATION SLAVE, REPLICATION CLIENT ON \*.\* TO 'flinkcdc'@'sales';

Discovered I needed to change to (basically the flinkcdc user need access to the database where the data is “sales” and also the “postgres” database as it needs to get to the write ahead logs and other postgres internals):

GRANT SELECT, SHOW DATABASES, REPLICATION SLAVE, REPLICATION CLIENT ON \*.\* TO 'flinkcdc'@'%';

[Additional note re MySqlDB setup for Replication](https://dev.mysql.com/doc/refman/8.4/en/replication-howto-repuser.html)

* And a BIG one… case matters…, let’s just say that again, CASE MATTERS. Your MySql Table/column names case need to match the case of the [Apache Flink](https://flink.apache.org/) table being created. – Lesson learned, maybe don’t put the column names in quotes, allowing the database to either lower case or upper case, and then match that in the Create Table in ‘creFlinkFlows/creCdc.sql’.

PostgreSQL

* Re your replication role/user, see [PostgreSql DB setup for Replication.](https://www.alibabacloud.com/help/en/flink/developer-reference/postgresql-cdc-connector/):
* Make sure ‘wal\_level='logical' is configured, this can be confirmed via the below steps.
  + First make sure you are starting up your database with the desired postgresql.conf file as per the configs section, to do this enter the container:

“docker compose exec -it <name> bash”

Ie:

docker compose exec -it postgrescdc bash

* + Then execute plsql cli utility:

psql -h localhost -p 5432 -U <user> -d <database>

SHOW config\_file;

It should respond with /etc/postgresql/postgresql.conf, we specify this in our docker-compose.yml using the configs section of the postgrescdc service.

If not then you can use the command primitive syntax in the postgrescdc service section to override the postgresql.conf file used by default and instruct postgres binary on startup using our desired config file.

[command: -c config\_file=/etc/postgresql/postgresql.conf](https://stackoverflow.com/questions/30848670/how-to-customize-the-configuration-file-of-the-official-postgresql-docker-image)

See the above URL embedded, scroll down to “With Docker Compose” on how this works.

Once you are now using the correct postgresql.conf file you can confirm the setting wal\_level setting by issuing ‘show wal\_level;’ in plsql cli tool as we used above.

[How to use the PostgreSql Docker Official image:](https://www.docker.com/blog/how-to-use-the-postgres-docker-official-image/)

PostgreSql was a bit more complicated in the end than I expected.

Was surprised when google told me the solution to some errors… but I was not using the tech as far as I was concerned.

IE: The advise was to configure “decoding.plugin.name”. But I nowhere saw [Debezium](https://debezium.io/) or configured anything. Then SLACK came to the saving and was informed that Apache Flink CDC actually uses a built in [Debezium](https://debezium.io/) engine internally and that you can pass Debezium configuration values using debezium. as a prefix: See below.



The PostgreSQL permissions required is “complicated” to say the least.

See: devlab/sql/postgrescdc/postgresql-init.sql for the SQL used to create the flinkcdc user and the replicator role in the end to enable all of this to work.

Hopefully I will be able to figure out how to reduce the permissions signed to flinkcdc and replicator role. Not a fan of assigning “superuser” as an example.

Looking at the Apache Flink CDC site, ([How to create a Postgres CDC table](https://nightlies.apache.org/flink/flink-cdc-docs-release-3.1/docs/connectors/flink-sources/postgres-cdc/)) they show a source table.

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',

-- experimental feature: incremental snapshot (default off)

'scan.incremental.snapshot.enabled' = 'true'

);

My working example ended as:

CREATE TABLE c\_hive.db01.t\_f\_pgcdc\_salespayments (

invoicenumber STRING

,paydatetime\_ltz STRING

,paytimestamp\_epoc STRING

,paid DOUBLE

,fintransactionid STRING

,created\_at TIMESTAMP

,PRIMARY KEY(invoicenumber) NOT ENFORCED

) WITH (

'connector' = 'postgres-cdc',

'hostname' = 'postgrescdc',

-- NOTE: this is the port of the db on the container, not the external docker exported port via a port mapping.

'port' = '5432',

'username' = 'flinkcdc',

'password' = <password>,

'database-name' = 'sales',

'schema-name' = 'public',

'table-name' = 'salespayments',

'slot.name' = 'flinkcdc',

-- experimental feature: incremental snapshot (default off)

'scan.incremental.snapshot.enabled' = 'true',

'scan.startup.mode' = 'latest-offset',

'decoding.plugin.name' = 'pgoutput'

);

Primarily look at the bits in RED, the most important differences being the addition of decoding.plugin.name=pgoutput

And there we thought this was a block about CDC… Just a little side track on how to setup MySql and PostgreSQL databases.

Also useful during this trip down this rabbit hole:

* [PostgreSQL: When wal\_level to logical](https://www.dbi-services.com/blog/postgresql-when-wal_level-to-logical/)
* [Understanding Replication in PostgreSQL – How to Set Up PostgreSQL Streaming Replication](https://www.percona.com/blog/setting-up-streaming-replication-postgresql/#:~:text=PostgreSQL%20streaming%20replication%20is%20a,mirror%20the%20primary%20database%20accurately.)
* [PostgreSQL CDC Connector](https://www.alibabacloud.com/help/en/flink/developer-reference/postgresql-cdc-connector/)

At this point if I insert records into either the MySql: sales/salespayment table or PostgreSQL: sales/public.salespayments table I am able to see that inside flink-sql-client when I run:

select \* from c\_hive.db01.t\_f\_msqlcdc\_salespayments

or

select \* from c\_hive.db01.t\_f\_pgcdc\_salespayments;

We will stop here for today. The biggest part of this is actually now done.

We have two working CDC streams flowing into our [Apache Flink](https://flink.apache.org/) environment. See you in the next instalment.

**About Me**

I’m a techie, a technologist, always curious, love data, have for as long as I can remember always worked with data in one form or the other, Database admin, Database product lead, data platforms architect, infrastructure architect hosting databases, backing it up, optimizing performance, accessing it. Data data data… it makes the world go round.

In recent years, pivoted into a more generic Technology Architect role, capable of full stack architecture.

[George Leonard](https://www.linkedin.com/in/george-leonard-945b502/)

[georgelza@gmail.com](mailto:georgelza@gmail.com)

------------------------------------------------------------------------------------------------------------------

# An Exercise in Discovery. Apache Flink CDC & Apache Paimon on S3. Part 3

## Confluent Kafka, Apache Flink + Flink CDC from PostgreSql or MySql => Apache Paimon (Storage via S3) with a Apache Hive Metastore

(9 October 2024)

**Overview**

At this point we’re going to go back to what we did in the original article.

First, as we created salespayments table as a CDC source above we will now push this data stream back into a Kafka by creating a new [Apache Flink](https://flink.apache.org/) table configured to output to a Kafka topic. The data will be pushed into this table utilizing a “insert into table <> select <columns> from <the source cdc table>.

And here is where things got interesting….

This is our MySQL CDC sourced table:

CREATE TABLE c\_hive.db01.t\_f\_msqlcdc\_salespayments (

`invoiceNumber` STRING

,`payDateTime\_Ltz` STRING

,`payTimestamp\_Epoc` STRING

,`paid` DOUBLE

,`finTransactionId` STRING

,`created\_at` TIMESTAMP

,PRIMARY KEY(`invoiceNumber`) NOT ENFORCED

) WITH (

'connector' = 'mysql-cdc'

,'hostname' = 'mysqlcdc'

,'port' = '3306'

,'username' = 'flinkcdc'

,'password' = 'flinkpw'

,'database-name' = 'sales'

,'table-name' = 'salespayments'

,'scan.startup.mode' = 'latest-offset'

);

We then create the target Apache Flink table as:

CREATE OR REPLACE TABLE c\_hive.db01.t\_f\_avro\_salespayments (

`invoiceNumber` STRING

,`payDateTime\_Ltz` STRING

,`payTimestamp\_Epoc` STRING

,`paid` DOUBLE

,`finTransactionId` STRING

,`created\_at` TIMESTAMP(3)

,`payTimestamp\_WM` AS TO\_TIMESTAMP(FROM\_UNIXTIME(CAST(`payTimestamp\_Epoc` AS BIGINT) / 1000)),

WATERMARK FOR `payTimestamp\_WM` AS `payTimestamp\_WM`

,PRIMARY KEY (`invoiceNumber`) NOT ENFORCED

) WITH (

'connector' = 'upsert-kafka'

,'topic' = 'avro\_salespayments'

,'properties.bootstrap.servers' = 'broker:29092'

,'properties.group.id' = 'testGroup'

,'value.format' = 'avro-confluent'

,'value.avro-confluent.schema-registry.url' = 'http://schema-registry:9081'

,'key.format' = 'avro-confluent'

,'key.avro-confluent.schema-registry.url' = 'http://schema-registry:9081'

,'value.fields-include' = 'ALL'

);

And now populate the [Apache Flink](https://flink.apache.org/) table: salespayments with the data from the CDC stream using the below statement.

Couple of things that we had to changed.

* The connector changed to upsert-kafka
* We needed to add a primary key not enforced.
* We needed to add the two key.\* variables
* We needed to remove “scan.startup.mode”
* And oh… we needed to check our target topic schema, in this version we added the created\_at column, so ye… had to add that to our avro schema salespayments.avsc file.

NOTE, in our code you will notice that the PostgreSQL table columns have been created all as lower case… as said previously, CASE matters, which had the ripple effect of impacting the select <columns> from PostgreSQL CDC source vs the MySQL CDC source. This is a exploration on discovery… Showing lessons learn (well those I remembered to write down and which won’t make the blog to confusing), things to think about and why. I could very easily just have gone and lower cased everything and not have demonstrated the “small” impact CASE has.

And here I choose to show how decisions or lack of knowledge/agreement/standard early on can have impact later on.

* MySqlDB

INSERT INTO c\_hive.db01.t\_f\_avro\_salespayments (

`invoiceNumber`

,`payDateTime\_Ltz`

,`payTimestamp\_Epoc`

,`paid`

,`finTransactionId`

,`created\_at`

) SELECT

invoiceNumber

,payDateTime\_Ltz

,payTimestamp\_Epoc

,paid

,finTransactionId

,created\_at

FROM

c\_hive.db01.t\_f\_msqlcdc\_salespayments;

* PostgreSql

INSERT INTO c\_hive.db01.t\_f\_avro\_salespayments (

`invoiceNumber`

,`payDateTime\_Ltz`

,`payTimestamp\_Epoc`

,`paid`

,`finTransactionId`

,`created\_at`

) SELECT

invoicenumber

,paydatetime\_ltz

,paytimestamp\_epoc

,paid

,fintransactionid

,created\_at

FROM

c\_hive.db01.t\_f\_pgcdc\_salespayments;



And at this point I’m going to post this image… Play on words… Jar Jar Binx…

As the character use to (in Star Wars movie series)… a simple turn of the head and the world comes tumbling down, due to those ears. For us those ears are jar’s files and their versions… a small change and ye, things are tumbling…

After some digging I was eventually able to execute the Insert statement, and got my data flow appearing in my Kafka topic: avro\_salespayments.

Next up, we’re going to join the salesbasket stream and the salespayments stream to create salescompleted.

Good… And I think that will be it for this issue, see you next time.

**About Me**

I’m a techie, a technologist, always curious, love data, have for as long as I can remember always worked with data in one form or the other, Database admin, Database product lead, data platforms architect, infrastructure architect hosting databases, backing it up, optimizing performance, accessing it. Data data data… it makes the world go round.

In recent years, pivoted into a more generic Technology Architect role, capable of full stack architecture.

[George Leonard](https://www.linkedin.com/in/george-leonard-945b502/)

[georgelza@gmail.com](mailto:georgelza@gmail.com)

------------------------------------------------------------------------------------------------------------------

# An Exercise in Discovery. Apache Flink CDC & Apache Paimon on S3. Part 4

## Confluent Kafka, Apache Flink + Flink CDC from PostgreSql or MySql => Apache Paimon (Storage via S3) with a Apache Hive Metastore

(10 October 2024)

**Overview**

As promised previously, next up we’re going to show how to do the join our two tables into salescompleted.

You will notice this table, when we created it we already configured it to be pushing data back to Kafka. We will be using our trusty insert into <column list> select <column list> from <source table> as previous.

We start by creating our output table salescompleted for join and then inserted records using the below Flink SQL.

CREATE OR REPLACE TABLE c\_hive.db01.t\_f\_avro\_salescompleted (

`invoiceNumber` STRING

,`saleDateTime\_Ltz` STRING

,`saleTimestamp\_Epoc` STRING

,`terminalPoint` STRING

,`nett` DOUBLE

,`vat` DOUBLE

,`total` DOUBLE

,`store` row<`id` STRING, `name` STRING>

,`clerk` row<`id` STRING, `name` STRING, `surname` STRING>

,`basketItems` array<row<`id` STRING, `name` STRING, `brand` STRING, `category` STRING, `price` DOUBLE, `quantity` INT, `subtotal` DOUBLE>>

,`payDateTime\_Ltz` STRING

,`payTimestamp\_Epoc` STRING

,`paid` DOUBLE

,`finTransactionId` STRING

,`payTimestamp\_WM` AS TO\_TIMESTAMP(FROM\_UNIXTIME(CAST(`payTimestamp\_Epoc` AS BIGINT) / 1000))

,`saleTimestamp\_WM` AS TO\_TIMESTAMP(FROM\_UNIXTIME(CAST(`saleTimestamp\_Epoc` AS BIGINT) / 1000))

,WATERMARK FOR `saleTimestamp\_WM` AS `saleTimestamp\_WM`

,PRIMARY KEY (`invoiceNumber`) NOT ENFORCED

) WITH (

'connector' = 'upsert-kafka'

,'topic' = 'avro\_salescompleted'

,'properties.bootstrap.servers' = 'broker:29092'

,'value.format' = 'avro-confluent'

,'value.avro-confluent.schema-registry.url'= 'http://schema-registry:9081'

,'key.format' = 'raw'

,'properties.group.id' = 'mysqlcdcsourced'

,'value.fields-include' = 'ALL'

);

INSERT INTO c\_hive.db01.t\_f\_avro\_salescompleted

SELECT

b.invoiceNumber

,b.saleDateTime\_Ltz

,b.saleTimestamp\_Epoc

,b.terminalPoint

,b.nett

,b.vat

,b.total

,b.store

,b.clerk

,b.basketItems

,a.payDateTime\_Ltz

,a.payTimestamp\_Epoc

,a.paid

,a.finTransactionId

FROM

c\_hive.db01.t\_f\_avro\_salespayments a,

c\_hive.db01.t\_k\_avro\_salesbaskets b

WHERE a.invoiceNumber = b.invoiceNumber

AND a.created\_at > b.saleTimestamp\_WM

AND b.saleTimestamp\_WM > (b.saleTimestamp\_WM - INTERVAL '1' HOUR);

For various aggregations we want to run we need to unnest the items in the sales basket first. As always, 2 steps, first we create our target table and then we use a insert statement.

CREATE OR REPLACE TABLE c\_hive.db01.t\_unnested\_sales (

`invoicenumber` STRING

,`store\_id` STRING

,`category` STRING

,`brand` STRING

,`product` STRING

,`subtotal` DOUBLE

,`saleDateTime\_Ltz` STRING

,`saleTimestamp\_Epoc` STRING

,`saleTimestamp\_WM` AS TO\_TIMESTAMP(FROM\_UNIXTIME(CAST(`saleTimestamp\_Epoc` AS BIGINT) / 1000))

,WATERMARK FOR `saleTimestamp\_WM` AS `saleTimestamp\_WM`

,PRIMARY KEY (`invoicenumber`, `store\_id`, `category`, `brand`, `product`) NOT ENFORCED

) WITH (

'connector' = 'upsert-kafka'

,'topic' = 'avro\_unnested\_sales'

,'properties.bootstrap.servers' = 'broker:29092'

,'value.format' = 'avro-confluent'

,'value.avro-confluent.schema-registry.url'= 'http://schema-registry:9081'

,'key.format' = 'avro-confluent'

,'key.avro-confluent.schema-registry.url' = 'http://schema-registry:9081'

,'properties.group.id' = 'mysqlcdcsourced'

,'value.fields-include' = 'ALL'

);

Now for those with a keen eye, you would have noticed for t\_f\_avro\_salescompleted above we used key.format of raw, but now here we’re using avro-confluent, why?

That’s because t\_f\_avro\_salescompleted primary key is based on a single column. When the primary key is a composite index based multiple columns we aren’t able to use raw, but rather need to use avro-confluent as per above.

INSERT INTO c\_hive.db01.t\_unnested\_sales

SELECT

`invoiceNumber` AS `invoicenumber`

,`store`.`id` AS `store\_id`

,bi.`category` AS `category`

,bi.`brand` AS `brand`

,bi.`name` AS `product`

,bi.`subtotal` AS `subtotal`

,`saleDateTime\_Ltz` AS saleDateTime\_Ltz

,`saleTimestamp\_Epoc` AS saleTimestamp\_Epoc

FROM c\_hive.db01.t\_f\_avro\_salescompleted -- assuming avro\_salescompleted is a table function

CROSS JOIN UNNEST(`basketItems`) AS bi;

At this point we have all the information we want, in Apache Flink and being pushed back to our Confluent Kafka cluster into topics.

Short section, but we actually have allow of our scaffolding done now.

**About Me**

I’m a techie, a technologist, always curious, love data, have for as long as I can remember always worked with data in one form or the other, Database admin, Database product lead, data platforms architect, infrastructure architect hosting databases, backing it up, optimizing performance, accessing it. Data data data… it makes the world go round.

In recent years, pivoted into a more generic Technology Architect role, capable of full stack architecture.

[George Leonard](https://www.linkedin.com/in/george-leonard-945b502/)

[georgelza@gmail.com](mailto:georgelza@gmail.com)

------------------------------------------------------------------------------------------------------------------

# An Exercise in Discovery. Apache Flink CDC & Apache Paimon on S3. Part 5

## Confluent Kafka, Apache Flink + Flink CDC from PostgreSql or MySql => Apache Paimon (Storage via S3) with a Apache Hive Metastore

(11 October 2024)

**Overview**

In this part we now turn to persisting the data, remember we said Apache Paimon…

Let’s face it, everything we have done up to now is of no value if we can’t do this… As there are enough written about [Apache Iceberg](https://iceberg.apache.org/) as an Open Table Format I decided to change it up again and rather push the data to [Apache Paimon](https://paimon.apache.org/).

[Apache Paimon](https://paimon.apache.org/) originates out of the [Apache Flink](https://flink.apache.org/) environment originally as known as the [Apache Flink tablestore](https://nightlies.apache.org/flink/flink-table-store-docs-master/docs/development/overview/), before being split out as its own [Apache](https://www.apache.org/) project.

*Anecdotal, it’s the datastore used by* [*Alibaba*](http://alibabacloud.com/) *behind their online store, driving sales/recommendations, so ye it can handle any load you can imagine.*

[Apache Paimon](https://paimon.apache.org/) can persist data in either:

* [Apache Hadoop (HDFS)](https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html)
* [AWS S3](https://aws.amazon.com/pm/serv-s3/?trk=c8974be7-bc21-436d-8108-722e8ab912e1&sc_channel=ps&ef_id=CjwKCAjw6c63BhAiEiwAF0EH1LA0JwlNpmLmXHogsLFp7K5zew60UdpC2QZz-lpoFv2A6JPowvtLFxoCBg0QAvD_BwE:G:s&s_kwcid=AL!4422!3!645125274431!e!!g!!aws%20s3!19574556914!145779857032&gbraid=0AAAAADjHtp_ui_GZQySpr62ifWS_Wj-9n&gclid=CjwKCAjw6c63BhAiEiwAF0EH1LA0JwlNpmLmXHogsLFp7K5zew60UdpC2QZz-lpoFv2A6JPowvtLFxoCBg0QAvD_BwE)

using either of the following Open File formats:

* [Apache Avro](https://avro.apache.org/),
* [Apache Parquet](https://parquet.apache.org/)
* [Apache ORC](https://orc.apache.org/)

In our case we’re going persist the data onto our [MinIO](https://min.io/) S3 backend store. This is primarily as [AWS S3](https://aws.amazon.com/pm/serv-s3/?trk=c8974be7-bc21-436d-8108-722e8ab912e1&sc_channel=ps&ef_id=CjwKCAjw6c63BhAiEiwAF0EH1LA0JwlNpmLmXHogsLFp7K5zew60UdpC2QZz-lpoFv2A6JPowvtLFxoCBg0QAvD_BwE:G:s&s_kwcid=AL!4422!3!645125274431!e!!g!!aws%20s3!19574556914!145779857032&gbraid=0AAAAADjHtp_ui_GZQySpr62ifWS_Wj-9n&gclid=CjwKCAjw6c63BhAiEiwAF0EH1LA0JwlNpmLmXHogsLFp7K5zew60UdpC2QZz-lpoFv2A6JPowvtLFxoCBg0QAvD_BwE) is becoming the standard for high volume, low cost storage in the Cloud, notwithstanding that most Enterprise storage platforms are now also providing [S3 Object](https://aws.amazon.com/s3/) storage functionality, enabling the same design for on Premise.

And off we go… ☺

Below I show two methods, the first is doing what’s referred to as CTAS (Create Table as Select)

CREATE TABLE c\_paimon.dev.t\_salesbaskets WITH (

'file.format' = 'avro'

) AS SELECT

`invoiceNumber`

,`saleDateTime\_Ltz`

,`saleTimestamp\_Epoc`

,`terminalPoint`

,`nett`

,`vat`

,`total`

,`store`

,`clerk`

,`basketItems`

FROM c\_hive.db01.t\_k\_avro\_salesbaskets;

CREATE TABLE c\_paimon.dev.t\_salespayments WITH (

'file.format' = 'avro'

) AS SELECT

`invoiceNumber`

,`payDateTime\_Ltz`

,`payTimestamp\_Epoc`

,`paid`

,`finTransactionId`

FROM c\_hive.db01.t\_f\_avro\_salespayments;

And as a second option we first create our output, target table, i.e.

CREATE OR REPLACE TABLE c\_paimon.dev.t\_unnested\_sales (

`invoicenumber` STRING

,`store\_id` STRING

,`category` STRING

,`brand` STRING

,`product` STRING

,`subtotal` DOUBLE

,`saleDateTime\_Ltz` STRING

,`saleTimestamp\_Epoc` STRING

,PRIMARY KEY (`invoicenumber`, `store\_id`, `category`, `brand`, `product`) NOT ENFORCED

) WITH (

'file.format' = 'avro'

);

And then we push the data from the c\_hive.db01. object into the above created table utilising a Insert into <target table> select <column list> from <source table> query.

INSERT INTO c\_paimon.dev.t\_unnested\_sales

SELECT

`invoicenumber`

,`store\_id`

,`category`

,`product`

,`brand`

,`subtotal`

,`saleDateTime\_Ltz`

,`saleTimestamp\_Epoc`

FROM c\_hive.db01.t\_unnested\_sales;

And just like that, we have data coming from 2 sources, throught Apache Flink, and out to Apache Kafka and into our Apache Paimon tables into Apache Avro files.

**About Me**

I’m a techie, a technologist, always curious, love data, have for as long as I can remember always worked with data in one form or the other, Database admin, Database product lead, data platforms architect, infrastructure architect hosting databases, backing it up, optimizing performance, accessing it. Data data data… it makes the world go round.

In recent years, pivoted into a more generic Technology Architect role, capable of full stack architecture.

[George Leonard](https://www.linkedin.com/in/george-leonard-945b502/)

[georgelza@gmail.com](mailto:georgelza@gmail.com)

------------------------------------------------------------------------------------------------------------------

# An Exercise in Discovery. Apache Flink CDC & Apache Paimon on S3. Part 6

## Confluent Kafka, Apache Flink + Flink CDC from PostgreSql or MySql => Apache Paimon (Storage via S3) with a Apache Hive Metastore

(12 October 2024)

**Overview**

This is going to be a very short section. … It was not planned…

Ok, so I was done… then I figured I know I have problems with the standalone [Apache Hive](https://hive.apache.org/) Metastore which I was using as the catalog service, and well, the next bit was pretty close to being done.

Eventually figured out that the Apache Hive Metastore 3.0 standalone container that I was using was not compatible with the rest of the stack…

I just so was working on building an Apache Hive Metastore 3.1.3 stack…. The base image builds was mostly done, so let’s complete it and rather use that.

What’s included is a split tier [Apache Hiveserver2 and Metastore](https://hive.apache.org/) server based on [Apache Hadoop 3.3.5](https://hadoop.apache.org/) and [Apache Hive 3.1.3](https://hive.apache.org/).

To build the stack the following steps are followed, (see the Makefile in the infrastructure directory). All directories also have a local README.md file with some notes and instructions.

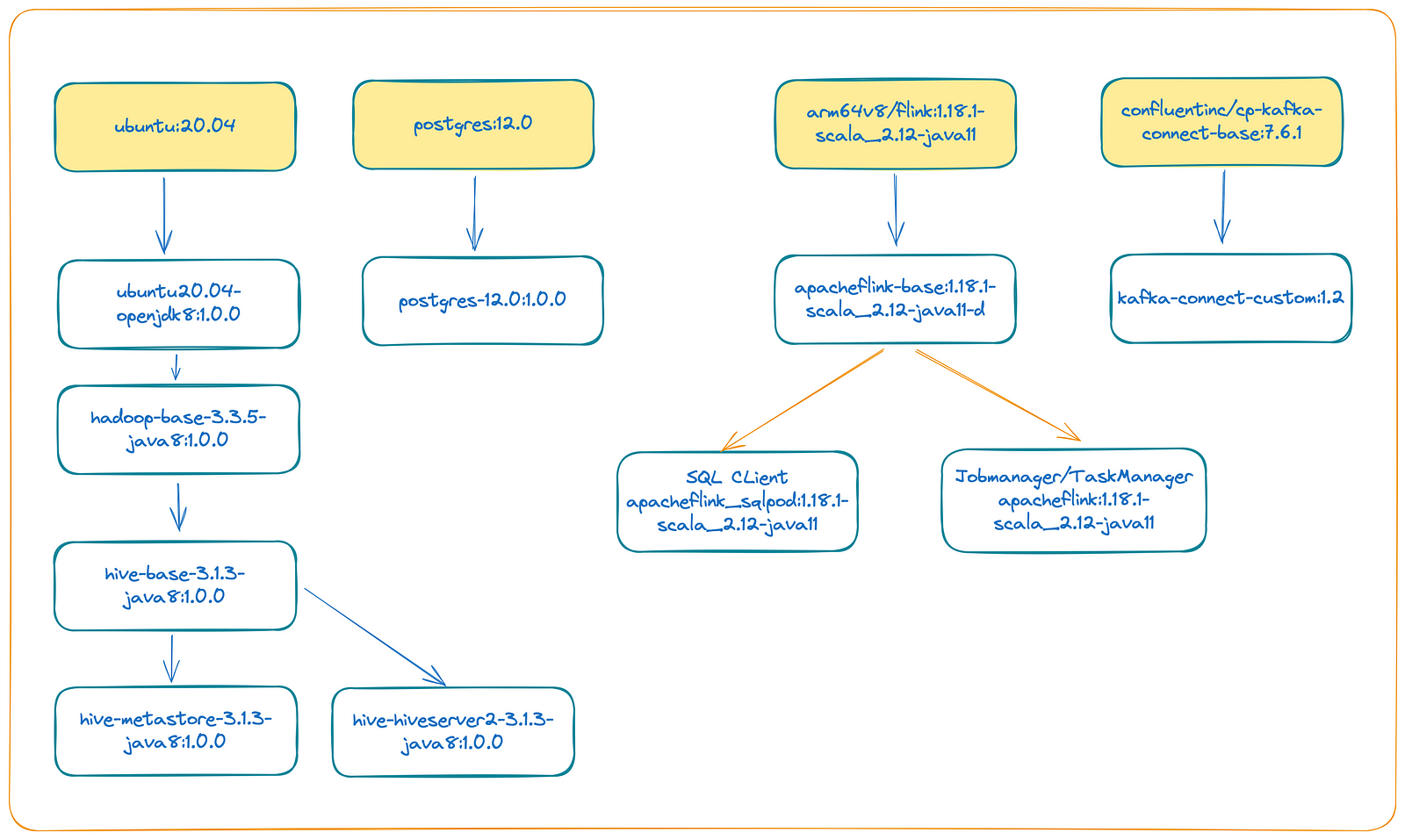
1. We first pull all our source images
2. We then execute buildall which will:

* First we build our base [Ubuntu 20.04](https://releases.ubuntu.com/focal/) image,
* We then extend this by building a new image with [OpenJDK 8](https://openjdk.org/projects/jdk8/) Installed.
* Next up was installation of [Apache Hadoop 3.3.5](https://hadoop.apache.org/)
* And lastly we install the [Apache Hive 3.1.3](https://hive.apache.org/) stack, itself configured to persists its data into a *PostgreSql 12.0*datastore.

**NOTE**, the variables required for Hive Metastore service to work are all inside the *hive.env* file located in our *devlab/conf* directory. Take note they need to match what’s in the larger project *.env* file.

**Docker Image Ancestry**

The below is an Ancestry diagram of sort, showing how I layered the images, how one deployment uses an image created previously. This all sits inside the infrastructure directory of the GIT repo.



**Some more References:**

[Apache Flink](https://flink.apache.org/) originally by [Ververica](https://docs.ververica.com/" \t "_blank) + [Flink CDC](https://nightlies.apache.org/flink/flink-cdc-docs-release-3.1/docs/get-started/introduction/" \t "_blank)

* [Welcome to Flink CDC](https://nightlies.apache.org/flink/flink-cdc-docs-release-3.1/docs/get-started/introduction/)
* [Connectors: Flink CDC sources](https://nightlies.apache.org/flink/flink-cdc-docs-release-3.1/docs/connectors/flink-sources/overview/)
* [Core Concepts: Data Pipeline](https://nightlies.apache.org/flink/flink-cdc-docs-release-3.1/docs/core-concept/data-pipeline/)
* Flink Forward: [How-to guide: Build Streaming ETL for MySQL and Postgres based on Flink CDC](https://www.ververica.com/blog/how-to-guide-build-streaming-etl-for-mysql-and-postgres-based-on-flink-cdc) (Good place to start).

Apache Flink CDC project on Github

* [CDC Connectors for Apache Flink](https://github.com/apache/flink-cdc/tree/release-2.3.0)

Do have a look at the above link, something very very Funky, you can define Sources, Targets and Pipelines (with transformation) via YAML definitions… See the data Pipeline link above.

And that’s it for this Blog series. As always, there are bits I still want to explore/switch out:

Thanks for sticking around, Hope the article was of value. Till next time.



**About Me**

I’m a techie, a technologist, always curious, love data, have for as long as I can remember always worked with data in one form or the other, Database admin, Database product lead, data platforms architect, infrastructure architect hosting databases, backing it up, optimizing performance, accessing it. Data data data… it makes the world go round.

In recent years, pivoted into a more generic Technology Architect role, capable of full stack architecture.

[George Leonard](https://www.linkedin.com/in/george-leonard-945b502/)

[georgelza@gmail.com](mailto:georgelza@gmail.com)