# An exercise in Discovery, Streaming data in the analytical world. – Part 1

## Apache Kafka, Apache Flink, Apache Iceberg on S3, Apache Paimon on HDFS, Apache Hive with internal standalone metastore (DerbyDB), external on PostgreSQL & on HDFS.

**Overview**

A hair brain idea… with good intensions that ended with “some…!!!” scope creep and lots of learning along the way.

This is as informal as I did it, it’s a blog (or make that will be blogs, plural), a bit longer than the normal, because of all the things done and learned, reading time, well various, I read slow, some read fast, enjoy the time, promise from my view point it will be of value.

Well originally this started as a very simple idea, lets create some data, publish it onto some Kafka topics, sink that into a [MongoDB](http://mongodb.com/) Atlas database/collection and then utilize the new Mongo stream processing to extract some value via aggregations and push (emit) that back onto [Apache Kafka](https://kafka.apache.org/) topics… to be displayed in terminal windows via simple Python consumers… other words an end to end flow. Well, that was the original concept sold to MongoDB Creator Community.

First, I discovered realized that due to work that this will take significantly longer than the 1 month of free Confluent Cloud access/credits, so the plan pivoted to deploying Confluent Platform locally via docker-compose.

I wanted the data to be used to have association / relevance and not simple fake random data so a small [Golang](https://go.dev/) (picked the language just because) application was created that constructed the source data from provided seed data/options. Note to full time coders, I am aware of various improvements that could be made, I know it can be split into a basket creator and a separate payment creator and deployed as individual containers themselves… The app was not the intent of the project so allow me some peace ;)

The concept, simulate a day in the life of a store, do the all to well-known shopping basket and payments example,

1. create a basket (constructed at a store selected at random from set of stores defined in seed file), comprised from random number of items (selected from seed file), random quantity of each item, once constructed the basket is posted onto a *salesbaskets* topic and then
2. create a *salespayment* record, associated with the basket, posted onto a separate *salespayments* topic.

At this point we have 2 input streams, simulating a store of some kind, idea, move the data along, create some real time aggregations, sink it into a long-term persistent data store (MongoDB), with further enrichment, Dashboard/Chart the various values, and make it available for downstream consumers.  
  
Added some more scope creep… How about using Apache Iceberg tables on S3, writing the data out using Apache Parquet as OTF (Open Table Format), hosted on AWS S3 compatible object storage (provided locally using a MinIO container).

Also included, or make that required is a catalog store, provided by Apache Hive (with a Metastore standalone and backed by a PostgreSQL DB). Other words the current HOT topic, go to Lakehouse architecture.

But then, more scope creep, it never ends. I decided to have a look at [Apache Paimon](https://paimon.apache.org/docs/master/) as a table format, utilizing HDFS for storage, storing the data as either avro, parquet or orc, we now talking streaming lakehouse also known as streamhouse. Very new!!! - See [The majesty of Apache Flink and Paimon](https://medium.com/@ipolyzos_/the-majesty-of-apache-flink-and-paimon-d36e73571fc9) for a nice writeup.

**Note**: there is a Master README.md in the root directory and most of the sub directories have README.md’s also, they include further explanation and notes that might be helpful/insightful or well, probably confusing also.

This document is built on top of work by allot of work by other people. Their work lays the initial groundwork, introduction into the various technologies used.

This document by no means takes anything away from their amazing work, this was purely me taking it further, for my own benefit/education.

Where desired/required I expanded by making my payloads more complex to see what impact that has (i.e. the unnesting step of the salescompleted basket to flatten it for additional aggregations).

As it stands this will be a 6-part posting, but it’s by no means complete.

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

See [my GIT repo for the entire document and code/article.](https://github.com/georgelza/MongoCreator-GoProducer-avro.git)

**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, Streaming data in the analytical world. – Part 2

## Apache Kafka, Apache Flink, Apache Iceberg on S3, Apache Paimon on HDFS, Apache Hive with internal standalone metastore (DerbyDB), external on PostgreSQL & on HDFS.

### (See [Part 1](https://medium.com/@georgelza/an-exercise-in-discovery-streaming-data-in-the-analytical-world-part-1-e7c17d61b9d2))

**Sections**

A short’ish descriptive breakdown of the components/modules that make up the project/rabbit hole…

**Source data generator**

A small custom Golang app… Originally did this for another project and found that it can easily be modified to be specific for the data required, thinking to see how much of this can be replaced via [Shadowtraffic](https://shadowtraffic.io/), as a simpler light weight alternative.

TODO: Still want to containerize this to make it easier for this project, it is not the focus though, although it did teach me allot of producing data onto Kafka in varies serialization formats.

Primary data records/documents:

*Salesbaskets* document

{

"InvoiceNumber": "1341243123341232",

"SaleDateTime\_Ltz": "2023-12-23T16:53:39.911+02:00",

"SaleTimetamp\_Epoc": "1718117619911",

"Store" : {

"Id": "1033",

"Name": "Derry"

},

"Clerk": {

"Id": "231",

"Name": "Martin",

"Surname": "Smith"

},

"TerminalPoint": "14",

"BasketItems":[

{

"Id": "234123412",

"Name": "Minty Fresh",

"Brand": "Colgate",

"Category": "Healthcare",

"Price":12412.00,

"Quantity":3

},

{

"Id": "234123421",

"Name": "All Bran",

"Brand": "Kellog's",

"Category": "Cereal",

"Price":12.00,

"Quantity":3

},

{

"Id": "534123412",

"Name": "Sugar Free",

"Brand": "Coke",

"Category": "Cool drinks",

"Price":112.00,

"Quantity":2

},

{

"Id": "224123412",

"Name": "Auto Wash",

"Brand": "OMO",

"Category": "Cleaning",

"Price":22.00,

"Quantity":4

}

],

"Net": 442.23,

"VAT":10.00,

"Total":452.23

}

Salespaymentsdocument

{

"InvoiceNumber": "1341243123341232",

"PayDateTime\_Ltz": "2023-12-23-T16:55:39.000+02:00",

"PayTimetamp\_Epoc": "1718117619911",

"Paid": 452.23,

"FinTransactionID": "42dfgt245wsdg34231rfwfg234234"

}

These 2 documents get joined to create Salescompleted*:*

{  
 "invoiceNumber": "1341243123341232",  
 "saleDateTime\_Ltz": "2023-12-23T16:53:39.911+02:00",  
 "saleTimestamp\_Epoc": "1718117619911",  
 "store" : {  
 "id": "1033",  
 "name": "Derry"  
 },  
 "clerk": {  
 "id": "231",  
 "name": "Martin",  
 "surname": "Smith"  
 },  
 "terminalPoint": "14",  
 "basketItems":[  
 {  
 "id": "234123412",  
 "name": "Minty Frsh",  
 "brand": "Colgate",  
 "category": "Healthcare",  
 "price":12412.00,  
 "quantity":3  
 },  
 {  
 "id": "234123421",  
 "name": "All Bran",  
 "brand": "Kellog's",  
 "category": "Cereal",  
 "price":12.00,  
 "quantity":3  
 }  
 ],  
 "nett": 442.23,  
 "vAT":10.00,  
 "total":452.23,  
 "payDateTime\_Ltz": "2023-12-23-T16:55:39.000+02:00",  
 "payTimestamp\_Epoc": "1718117619911",  
 "paid": 452.23,  
 "finTransactionId": "42dfgt245wsdg34231rfwfg234234"  
}

**Serialization Formats**

Now first thing, the data is structured, either as a row with columns, or can be a xml message or a JSON structure, you get simply JSON, one level and then complex JSON which surprisingly as a massive impact. All of this is simple text at this point, what’s published onto the topic and read from the topic however is a byte stream, and this is where serialization comes in, the options being:

* [JSON](https://www.w3schools.com/js/js_json_intro.asp)
* [Protobuf](https://protobuf.dev/)
* [Apache Avro](https://avro.apache.org/)

Each have its Pro’s and Con’s as will be discussed later.

**Project environment**

This became a project in itself on how to structure things, re-use at the right time in the right way, lots of potential improvements have been identified, for another day.

Below we’re going to have a “*quick*” discussion about what we need to deploy to run the project. Allot of this was done locally, to safe cost & some of it cannot be done locally (MongoDB Atlas Change Streams and Charts).

Doing this locally yourself also happens to give you more insight into how the stack fit together, how to extend the stack/capabilities by importing additional jar files.

**For now, the MongoDB Atlas bit is on hold, don’t have access to a MongoDB Atlas environment that exposes Change Stream processing. :( as this was the original primary intension of this project.**

Everything was done using docker images and deployed via docker compose files: docker-compose.yml.

There are basically 5 sub projects (at the moment) exploring the different ways to work with the data and how to persist the data into purpose build analytic storage environments:

The environment is stood up by executing “make” against the various Makefile steps via a make <option> command. It defaults to make help.

* Infrastructure
  + In here we start with some docker pull (“pullall”) commands to localize the base images we will use,
  + After this we can do the various build (“buildall”) command to extend the base images pulled above into our source images that will be used by each of the following environments.

NOTE: each of the below have a build command that will build the required images from the images build in the infrastructure section.

* + After the build for each of the below sections there is a run command that will start the docker compose environment,
  + After the run command was executed execute the “deploy” command to create the topics, schema entries and various kStream, kTable, Flink Table, [Iceberg](https://iceberg.apache.org/) and [Paimon](https://paimon.apache.org/) catalogs and tables.
  + There are various other commands to make life simpler like down, ps, stop etc.
  + I work on a M1 based MAC Book Pro, so my architecture is arm64, aka aarch64, as such see all Dockerfile’s under “./infrastructure\*” for the base images used and change it back to AMD64. Also scan through the various Dockerfile’s as some jar files utilised are aarch64 based, simply remove that bit.

* devlab-mongo
  + Publish documents to 2 Kafka topics, use a Kafka sink connector on each to push documents into MongoDB collections.
  + From her all processing is done on the MongoDB Atlas environment utilizing Mongo Change Streams and then displayed using MongoDB Charts. By far the simplest solution.
* devlab-hms-standalone
  + Publish documents to 2 Kafka topics, use some Kafka kSql and Apache Flink Sql to do transformations & aggregations, pushing calculated values back onto new Kafka Topics.
  + We now also push the data streams onto an [Apache Iceberg](https://iceberg.apache.org/) environment, storage provided via a AWS S3 via [MinIO](https://min.io/) container. This version further utilized a stand-alone [Apache Hive Meta](https://hive.apache.org/) store with internal DerbyDb as a catalog.
* devlab-hms-postgres
  + Same as per above, with the [DerbyDb](https://db.apache.org/derby/) swopped out for an external PostgreSql DB.
* devlab-hdfs-paimon-hdfs-cat
  + Here we push the output stream into an [Apache Paimon](https://paimon.apache.org/) stack, storage provided via Hadoop DFS cluster. The catalog for the [Paimon](https://paimon.apache.org/) objects in this example was hosted on the already available HDFS storage sub system. For [Apache Flink](https://flink.apache.org/) the catalog was left inside HMS with a PostgreSQL backend datastore.
* devlab-hdfs-paimon-hms-cat
  + As per above, but we now relocating the [Apache Paimon](https://paimon.apache.org/) catalog from the HDFS storage tier into a HMS with a PostgreSQL back end data store.
    - Stuck!!!…

Flink SQL> 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`,

> `saleTimestamp\_WM`

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

[ERROR] Could not execute SQL statement. Reason:

java.lang.IllegalArgumentException: Field name [invoiceNumber, saleDateTime\_Ltz, saleTimestamp\_Epoc, terminalPoint, basketItems, saleTimestamp\_WM] cannot contain upper case in the catalog.

***Lesson***: It seems when the catalog is in Apache Hive with a PostgreSQL back-end data store then the mix case field names are a problem… This might be a very invasive fix as the field names come from step 1…

For now, hitting PAUSE button… -> We will put this for now under ***Lesson:***

Each of the above have a Makefile that can be used to first build the environment, then start it via the run command and then deploy/create the topics and table objects via the deploy command.

***Lesson****: For the network name: don’t include a “\_”, hms (hive metastore), I discovered have some naming conventions… which don’t like the “\_” in the network name.*

To control the container names and hostnames you can follow the following structure:

***Lesson****: as you going to run multiple* [*Apache Flink*](https://flink.apache.org/) *Task Managers don’t include a container name, it will be uniquely created as:*

“flink-taskmanager\_#”

*with the # being a number.*

Include network name:

# Without a network explicitly defined, you hit this Hive/Thrift error

# java.net.URISyntaxException Illegal character in hostname

# https://github.com/TrivadisPF/platys-modern-data-platform/issues/231

networks:

default:

name: ${COMPOSE\_PROJECT\_NAME}

Beware “localhost” monster in a docker-compose/container environment, what’s the saying if all else fails “It’s always DNS”. When talking, referencing services in a container based environment, always think, when a instruction is given to a container, how does it know who to talk to, on your local machine you can point to different services simply by saying localhost:<port> but that same localhost in the container is the container itself, so always remember, refer to other services by their service name as defined in the docker-compose.yml file, it will safe you years of grieve.

Note: Take notice of the port change for the Kafka schema\_manager, this is to remove a port conflict with the Flink Jobmanager which also wanted 8081. It is easier to change the schema\_manager default port.

*.env* values

Each directory where a docker-compose.yml file is located will have a .envfile, with variables that’s then pushed into the compose file at execution time. Here is the .env file from “devlab*-*hms-postgres” sub project.

COMPOSE\_PROJECT\_NAME=devlab  
  
AWS\_ACCESS\_KEY\_ID=admin  
AWS\_SECRET\_ACCESS\_KEY=password  
AWS\_REGION=za-south-1  
AWS\_DEFAULT\_REGION=za-south-1  
  
DATABASE\_HOST=postgres  
DATABASE\_PORT=5432  
DATABASE\_DB=hms\_iceberg  
DATABASE\_USER=hive  
DATABASE\_PASSWORD=hive  
  
S3\_ENDPOINT\_URL=http://minio:9000  
S3\_PREFIX=db01  
S3\_BUCKET=iceberg

There are also a .pwd file in the very root of the project that will contain the MongoDB Atlas connection and Confluent Kafka cluster credentials.

See the top of the runs\_avro.sh file for which file it executes to set these variables.

#!/bin/bash  
  
# CP Kafka Cluster - No more, this is now here just as an example how to point to CP.  
export Sasl\_username=KB5MJIsdfasdWQEOQYQ   
export Sasl\_password=ZLtU89ymZMCh5dsfgfdjghjdhJjyoGmnyFFcZPVOUC+DiMYdNaUxjJ+2kh  
  
# MongoDB ATLAS  
# https://medium.com/mongoaudit/how-to-enable-authentication-on-mongodb-b9e8a924efac  
export mongo\_username=adminuser # MongoDB Atlas  
export mongo\_password=g12576ehefdgbWFyrR

See [my GIT repo for the entire document and code/article.](https://github.com/georgelza/MongoCreator-GoProducer-avro.git)

**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, Streaming data in the analytical world. – Part 3

## Apache Kafka, Apache Flink, Apache Iceberg on S3, Apache Paimon on HDFS, Apache Hive with internal standalone metastore (DerbyDB), external on PostgreSQL & on HDFS.

### (See Part 2)

**Components/Technologies**

[Confluent Platform](https://www.confluent.io/) Kafka environment

* Broker
* Control Center
* kSqlDb Server
* kSqlDb Client
* Schema Registry
* Connect

Custom Connect container with additional source/sink connectors installed.

[Apache Flink](https://flink.apache.org/)

* Little Note, all [Apache Flink](https://flink.apache.org/) stacks are build using a single image: flink-kafka:1.18.1-scala\_2.12-java11, this image contains all off: jobmanager, taskmanager & sql\_client. For easy I use one image for the task manager and job manager (with libraries/JAR’s added) and a second similar (with libraries/ JAR’s added), the only difference being slight build difference and the final start command.
* [Apache Flink](https://flink.apache.org/) is package by various groups, i.e. Apache, [Ververita](https://www.ververica.com/), [Confluent](https://www.confluent.io/):
* It’s really helpful to just scan through: <https://nightlies.apache.org/flink/flink-docs-master/docs/deployment/resource-providers/standalone/docker/>
* Note: Persistence is not configured by default (in a docker compose lab is deployed when deploying the previous mentioned images). Well, what do I mean by this… surprise if you create [Flink](https://flink.apache.org/) objects, and exit [Flink](https://flink.apache.org/) SQL, when you restart it, all your tables/jobs are gone, and you will need to recreate them.
* This is where we dropped into the entire subject of catalogs.

See [Robin Moffatt](https://www.linkedin.com/in/robinmoffatt/)’s [Decodablecoe](https://github.com/decodableco/examples/tree/main/catalogs) GIT repo for examples on how to configure object persistence.

Kcat

* Sometimes it helps to be able to peek at what’s going on inside topics, by just echo what posted it to the terminal, for that I included kcat in the docker-compose file, already plumbed into the project network.
* BTW: kcat is the new replacement for kafkacat.

[MongoDB Atlas local](https://hub.docker.com/r/mongodb/atlas) & MongoDB Atlas cloud

* Local MongoDB to test Kafka MongoDB Sinks.

[PostgreSql](https://www.postgresql.org/)

* The PostgreSQL datastore is firstly used primarily for the [Apache Hive Metastore](https://hive.apache.org/), but, for a future use case, might want to source or sink records from the database, we can take this further by using a MySQL datastore as a source or sink target.

TODO: Perfect to sink the unnested*\_*sales data stream into.

[MinIO](https://min.io/)

* AWS Compatible S3 – Storage layer used for Apache Iceberg persistence.

[MC](https://min.io/docs/minio/linux/administration/minio-console.html)

* MinIO Console by Min.io

[HDFS](https://min.io/)

* Apache Hadoop DFS – Storage Layer for the [Apache Paimon](https://paimon.apache.org/) persistence.

This deploys a 5 datanode cluster, with historyserver, namenode, resourcemanager & nodemanager.

[Apache Hive Metastore](https://hive.apache.org/)

* Option as an [Apache Flink](https://flink.apache.org/) Catalog.
* Do read [Catalogs in Flink SQL—A Primer](https://www.decodable.co/blog/catalogs-in-flink-sql-a-primer?_gl=1*ik71wd*_gcl_au*MjQ2ODI4NDQ0LjE3MjEzMTIwMjI.*_ga*NTM2ODA4OTg4LjE3MjEzMTIwODM.*_ga_G4YHDQYS1G*MTcyMTU4MjgxNC42LjEuMTcyMTU4NjQxNy42MC4wLjA.) by Robin Moffat for a great overview and [Catalogs in Flink SQL—Hands On](https://www.decodable.co/blog/catalogs-in-flink-sql-hands-on).

[Apache Flink](https://flink.apache.org/) by default only maintains/remembers your persistent objects created in the current session. During your sql-client session it uses an in-memory catalog, other words when you exit sql-client, and come back, whatever you created is gone or anything you created in your session won’t be visible/available to other sessions/users.

To store the definitions (meta-data) persistently a catalog is required.

[Apache Flink](https://flink.apache.org/) supports 3 types of catalogs, namely:

* in memory (default),
* [Apache Hive](https://hive.apache.org/) (using [Hive Metastore](https://hive.apache.org/), which is backed by a RDBMS) and lastly,
* JDBC (In this case [Apache Flink](https://flink.apache.org/) communicates directly with the JDBC target, basically no [Hive Metastore](https://hive.apache.org/), the gotcha it only exposes what is in the target RDBMS, it does not allow you to define new objects).

See [my GIT repo for the entire document and code/article.](https://github.com/georgelza/MongoCreator-GoProducer-avro.git)

**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, Streaming data in the analytical world. – Part 4

## Apache Kafka, Apache Flink, Apache Iceberg on S3, Apache Paimon on HDFS, Apache Hive with internal standalone metastore (DerbyDB), external on PostgreSQL & on HDFS.

### (See Part 3)

**Computations/Aggregations:**

Well in the end this is what all this is about actually (everything above was scaffolding, but nevertheless important to know and understand). This is where we want to take our raw feeds (salesbaskets & salespayments) and do the magic, analysis/aggregations to derive value/insight into what’s happening on the floor, how is business doing, really.

What follows is an overview end to end of the plan.

Important to know, if you pay attention, you will realize the same output is created using different methods, that was intentional. See how the different options can be used to get to the same end point and what’s involved, how is one easier or not than the other and what comes with the easy or what comes with a little bit more effort.



So now that we have an Overview, below I will first attempt to demonstrate the “[kSql](https://www.confluent.io/blog/ksql-streaming-sql-for-apache-kafka/) high level” version after which I discuss the “[Apache Flink](https://flink.apache.org/)” option:



Next up is a more detailed view of what we’re going to be doing using [KSQL](https://www.confluent.io/blog/ksql-streaming-sql-for-apache-kafka/) inside the Apache Kafka cluster to create [kStreams](https://docs.confluent.io/platform/current/streams/concepts.html#:~:text=A%20KTable%20is%20an%20abstraction,will%20be%20considered%20an%20INSERT).) and [kTables](https://docs.confluent.io/platform/current/streams/concepts.html#:~:text=A%20KTable%20is%20an%20abstraction,will%20be%20considered%20an%20INSERT).) objects.



**Aggregations via KSQL & kTable**

What I do here is use KSQL to first create a stream object for the salesbaskets and salespayment source from the Kafka topics.

We then use [KSQL](https://www.confluent.io/blog/ksql-streaming-sql-for-apache-kafka/) to create a new [KSQL](https://www.confluent.io/blog/ksql-streaming-sql-for-apache-kafka/) object calls salescompleted, as a join between salesbaskets and salespayment, based on the invoiceNumber column.

The salescompleted stream is then used to create [kTable](https://docs.confluent.io/platform/current/streams/concepts.html#:~:text=A%20KTable%20is%20an%20abstraction,will%20be%20considered%20an%20INSERT).) objects, which output (using tumbling windows):

* Sales per store per terminal per 5 min & hour
* Sales per store per 5 min & hour

See crekSqlFlows directory for the SQL utilized to create the various [kStream](https://docs.confluent.io/platform/current/streams/concepts.html#:~:text=A%20KStream%20is%20an%20abstraction,in%20the%20unbounded%20data%20set.) and [kTable](https://docs.confluent.io/platform/current/streams/concepts.html#:~:text=A%20KTable%20is%20an%20abstraction,will%20be%20considered%20an%20INSERT).) objects.

Example:  
  
Create a stream object from source salesbaskets Kafka topic, same format/serialization as source. This becomes an input table for us.

CREATE STREAM avro\_salesbaskets (

InvoiceNumber VARCHAR,

SaleDateTime\_Ltz VARCHAR,

SaleTimestamp\_Epoc VARCHAR,

TerminalPoint VARCHAR,

Nett DOUBLE,

Vat DOUBLE,

Total DOUBLE,

Store STRUCT<

Id VARCHAR,

Name VARCHAR>,

Clerk STRUCT<

Id VARCHAR,

Name VARCHAR,

Surname VARCHAR>,

BasketItems ARRAY< STRUCT<

id VARCHAR,

Name VARCHAR,

Brand VARCHAR,

Category VARCHAR,

Price DOUBLE,

Quantity integer >>)

WITH (KAFKA\_TOPIC='avro\_salesbaskets',

VALUE\_FORMAT='Avro',

PARTITIONS=1);

Create a stream object from source salespayments. This is our second input table.

CREATE STREAM avro\_salespayments (

InvoiceNumber VARCHAR,

FinTransactionId VARCHAR,

PayDateTime\_Ltz VARCHAR,

PayTimestamp\_Epoc VARCHAR,

Paid DOUBLE )

WITH (

KAFKA\_TOPIC='avro\_salespayments',

VALUE\_FORMAT='Avro',

PARTITIONS=1);

Now let’s create our salescompleted stream, this will hold our joined output constructed from the previous 2 streams.

CREATE STREAM avro\_salescompleted WITH (

KAFKA\_TOPIC='avro\_salescompleted',

VALUE\_FORMAT='Avro',

PARTITIONS=1)

as

select

b.InvoiceNumber,

as\_value(p.InvoiceNumber) as InvNumber,

b.SaleDateTime\_Ltz,

b.SaleTimestamp\_Epoc,

b.TerminalPoint,

b.Nett,

b.Vat,

b.Total,

b.store,

b.clerk,

b.BasketItems,

p.PayDateTime\_Ltz,

p.PayTimestamp\_Epoc,

p.Paid,

p.FinTransactionId

from

avro\_salespayments p INNER JOIN

avro\_salesbaskets b

WITHIN 7 DAYS

on b.InvoiceNumber = p.InvoiceNumber

emit changes;

With the above created we can now do an aggregation, the below creates an output kTable, with a tumbling window over 5 minutes.

CREATE TABLE avro\_sales\_per\_store\_per\_5min WITH (

KAFKA\_TOPIC='avro\_sales\_per\_store\_per\_5min',

VALUE\_FORMAT='AVRO',

PARTITIONS=1)

as

SELECT

store->id as store\_id,

as\_value(store->id) as storeid,

from\_unixtime(WINDOWSTART) as Window\_Start,

from\_unixtime(WINDOWEND) as Window\_End,

count(1) as sales\_per\_store

FROM avro\_salescompleted

WINDOW TUMBLING (SIZE 5 MINUTE)

GROUP BY store->id

EMIT FINAL;

Aggregations via [Apache Flink](https://flink.apache.org/):



In this scenario we use [Apache Flink](https://flink.apache.org/) to mirror some of what was done above using kSql, but this time using [Apache Flink](https://flink.apache.org/) Sql, and some additional magic.

In the [Apache Flink](https://flink.apache.org/) case we do things in 2 steps (I just found it easier), first we create a [Apache Flink](https://flink.apache.org/) table (I like to think of this as a virtual table as nothing is actually persisted in the table, further, because the table itself actually only points to the salesbaskets and salespayments Kafka topic’s as sources.

When interacting with the [Apache Flink](https://flink.apache.org/) table it engages a Kafka consumer via the configured source topic.

When the virtual table is defined a connector parameter is configured which is either “upsert-kafka” or “kafka”. The “kafka” connector works perfectly for sourcing data or inserting/appending data/records to the back’ing Kafka topic. Pretty much how Kafka works as an immutable log.

The upsert-kafka, is however useful when consuming data from a topic, updating the [Apache Flink](https://flink.apache.org/) table/record. As such upsert-kafka requires a primary key to be defined to enable it to find the record being manipulated.

See: [for more on the subject.](https://docs.ververica.com/vvc/connectors-and-formats/built-in-connectors/upsert-kafka)

Once the 2 source virtual tables are created, we create our 3rd table, this time it’s an output, called salescompleted. An Insert/join statement is then executed that join the 2 input tables: salesbasket and salespayments on the invoiceNumber column. By executing this insert statement data is published onto the salescompleted topic hosted on the Kafka cluster. Now the fun begins.

The above [Apache Flink](https://flink.apache.org/) SQL run on the [Apache Flink](https://flink.apache.org/) cluster. If executed as per above, you will see a shorted version of you command as the description. To make it more descriptive see the usage of the SET command, i.e.:

SET 'pipeline.name' = 'Sales Basket[Source/Target] - Kafka[Topic/Table ]' ;

Using the above will assign the value in the quotes as the description in Running Jobs view.

From here we do aggregations, first up was sales per store per terminal per 5 min. Again, we create an output table followed by the required insert statement. That was the simple / easy one… Next up we want to compute:

* sales per store per product (name key) per time window (hour or minutes or xxx),
* sales per store per brand per time window
* sales per store per category per time window
* sales per store per terminal per time window

What I have not mentioned above, if you look at the basketItems array of objects, you will realize it’s a nested data set (complex structure). If we want to execute aggregations on the objects in the array, we will first need to unnest the array into a flat structure (unnested\_sales).

This is done by creating a table that is flat, for which each record will be inserted into the output table for each object from the basketItems arrays, associated with the original salesbasket invoiceNumber.

{  
 "invoiceNumber": "1341243123341232",  
 "saleDateTime\_Ltz": "2023-12-23T16:53:39.911+02:00",  
 "saleTimestamp\_Epoc": "1718117619911",  
 "storeid" : "1033",  
 "product":"Minty Frsh",  
 "brand": "Colgate",  
 "salesvalue":12412.00,  
 "category": "Healthcare"  
}

This table can then be used as a source for the required select statements with required group by clauses based on time of sales. To improvement performance, we include a filter to run against recent data only.

Consider the difference in output that emit change vs emit final has.

* A “emit changes” outputs data, new value for the aggregations as it arrives, in this case into the salescompleted table followed by the unnested\_sales table.
* A “emit final” outputs data at the end of the window tumble period.

Because we have a unnested structure we now have a record that can also happily, easily be sink’d into a “old style” RDBMS (rows and columns), even though our old-style RDBMS database engines themselves are extending their capabilities to include storage of JSON structured records as a field/column type.

**Data serialization format’s**

First there was [Protobuf](https://protobuf.dev/) and then [Avro](https://avro.apache.org/) came, let’s not forget the JSON and then serialized/schema’s JSON known as JSOND.

What is it and why…

* Well, what’s transferred across the wire by Kakfa, and stored inside the topic/partitions/data files is a byte representation of what you published…
* But working with RAW bytes are no fun, so we Serialize the text into a format (above…), transport it across Kafka and then Deserialize the other end.
* Associated with the Serialization/de-serialization commonly known as Serdes is the schema registry. A full set of articles, but at the simplest, the old adage of Garbage in Garbage out applies, schema registry is your contract, making sure what’s publish abide by what was agreed on, which results in trust, quality, value out.

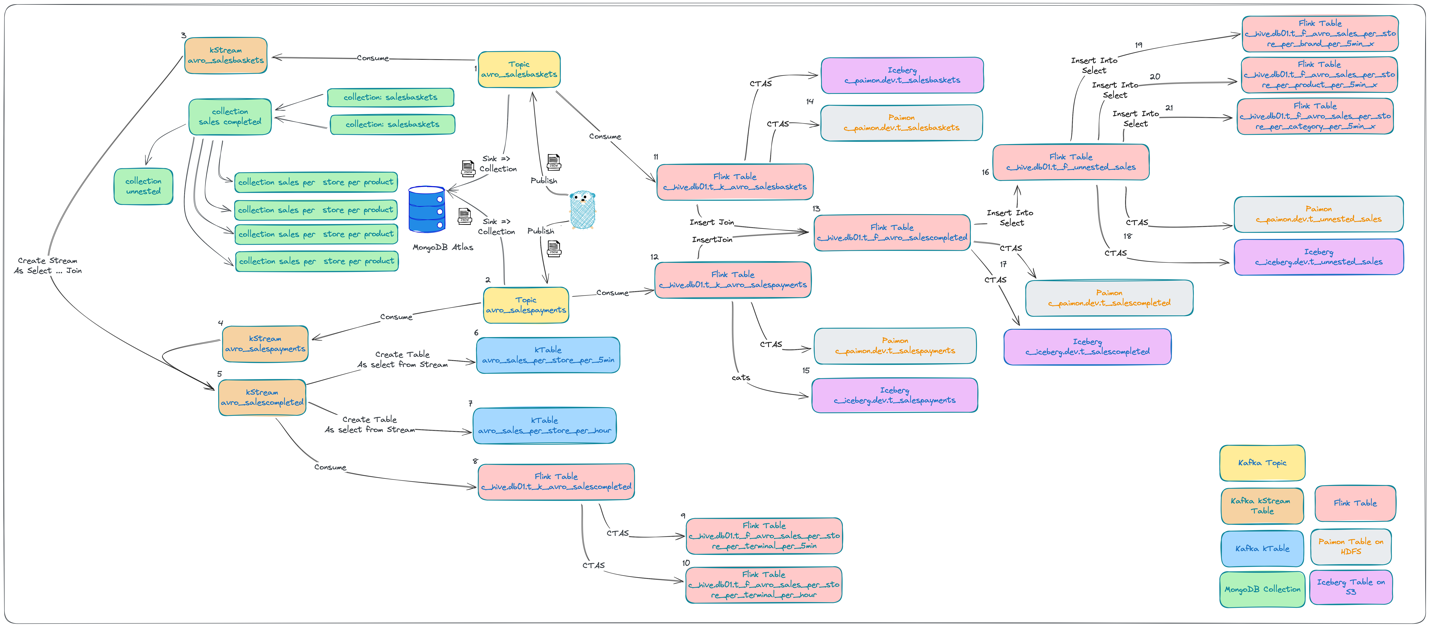
**Avro Schema**

**Lesson:** A word of warning: Case Sensitivity between [Apache Flink](https://flink.apache.org/) SQL’s: Create table <>, Select <> From <> and Kafka schema registry entries matters.

Create your pipeline, processing using small pieces of work, at least initially, once they work you can always consolidate them. The old saying of, how do you an elephant, one byte at a time applies.

**Data Lineage**

Below is a Diagram depicting how the original 2 topics are “grown” moved around, joined and pushed to the various storage options via the different sections of this project. There will be a more section specific diagram with each.



**Examples:**

The below builds a table avro\_salescompleted, backed/sourced from the Kafka topic/kSql created table via a Stream join.

CREATE TABLE avro\_salescompleted (

INVNUMBER 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>>,

FINTRANSACTIONID STRING,

PAYDATETIME\_LTZ STRING,

PAYTIMESTAMP\_EPOC STRING,

PAID DOUBLE,

SALESTIMESTAMP\_WM AS TO\_TIMESTAMP(FROM\_UNIXTIME(CAST(SALETIMESTAMP\_EPOC AS BIGINT) / 1000)),

WATERMARK FOR SALESTIMESTAMP\_WM AS SALESTIMESTAMP\_WM

) WITH (

'connector' = 'kafka',

'topic' = 'avro\_salescompleted',

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

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

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

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

'value.avro-confluent.schema-registry.url' = 'http://schema-registry:8081',

'value.fields-include' = 'ALL'

);

We now going to use the above as a source, where we going to output the group by from this into the below table, backed by topic which we will sink to MongoDB via connector.

CREATE TABLE avro\_sales\_per\_store\_per\_terminal\_per\_5min (

store\_id STRING,

terminalpoint STRING,

window\_start TIMESTAMP(3),

window\_end TIMESTAMP(3),

salesperterminal BIGINT,

totalperterminal DOUBLE,

PRIMARY KEY (store\_id, terminalpoint, window\_start, window\_end) NOT ENFORCED

) WITH (

'connector'= 'upsert-kafka',

'topic' = 'avro\_sales\_per\_store\_per\_terminal\_per\_5min',

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

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

'key.avro-confluent.url' = 'http://schema-registry:8081',

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

'value.avro-confluent.url'= 'http://schema-registry:8081',

'value.fields-include' = 'ALL'

);

Insert into avro\_sales\_per\_store\_per\_terminal\_per\_5min

SELECT

`STORE`.`ID` as STORE\_ID,

TERMINALPOINT,

window\_start,

window\_end,

COUNT(\*) as salesperterminal,

SUM(TOTAL) as totalperterminal

FROM TABLE(

TUMBLE(TABLE avro\_salescompleted, DESCRIPTOR(SALESTIMESTAMP\_WM), INTERVAL '5' MINUTES))

GROUP BY `STORE`.`ID`, TERMINALPOINT, window\_start, window\_end;

Aggregations via [MongoDB Change Stream](https://www.mongodb.com/resources/products/capabilities/change-streams) Processing

* Sales per store per terminal per hour / per day
* Sales per store per product per hour / per day
* Sales per store per brand per hour / per day
* Sales per store per category per hour / per day

Each of the above include a count and a monetary value.

What to do with the results:

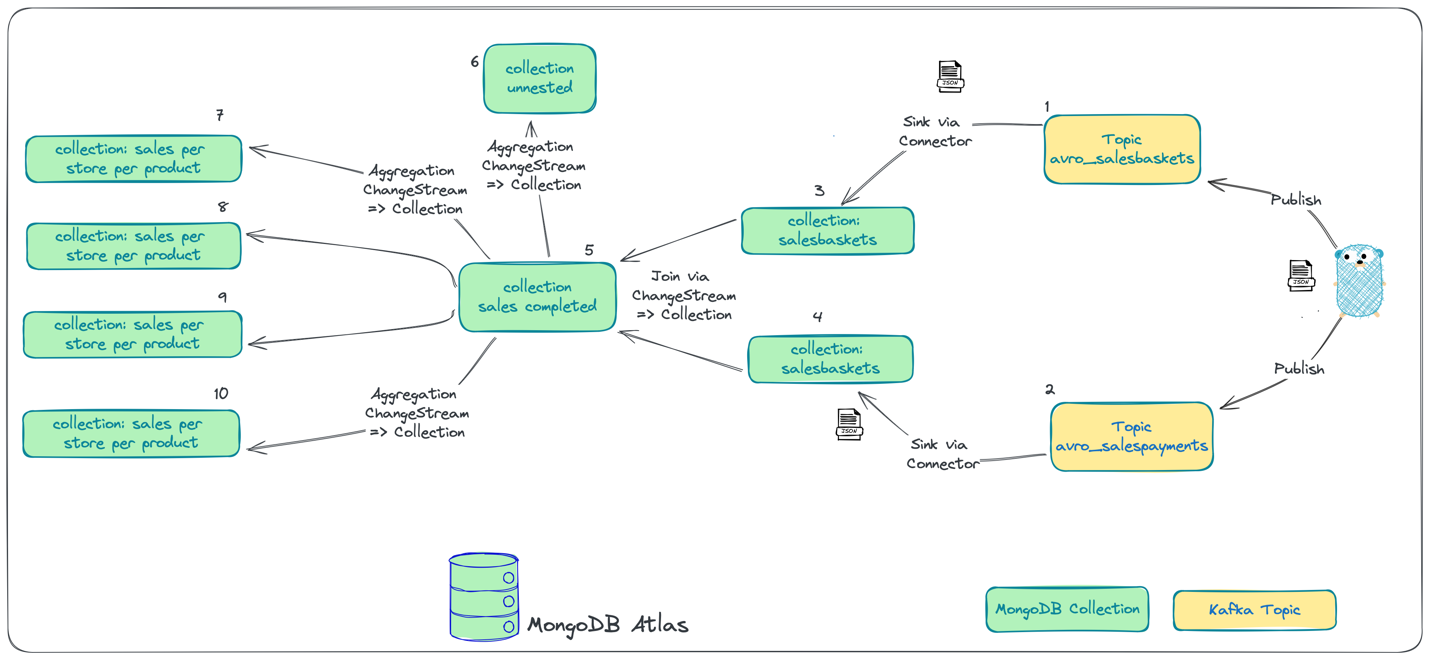
* Sink (push) to MongoDB Atlas Collections
* Sink (push) to Kafka Topics
* Sink (push) to [Apache Iceberg](https://iceberg.apache.org/) on [S3](https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html)
* Sink (push) to [Apache Paimon](https://paimon.apache.org/) on HDFS

**Dashboards/Charts in MongoDB**

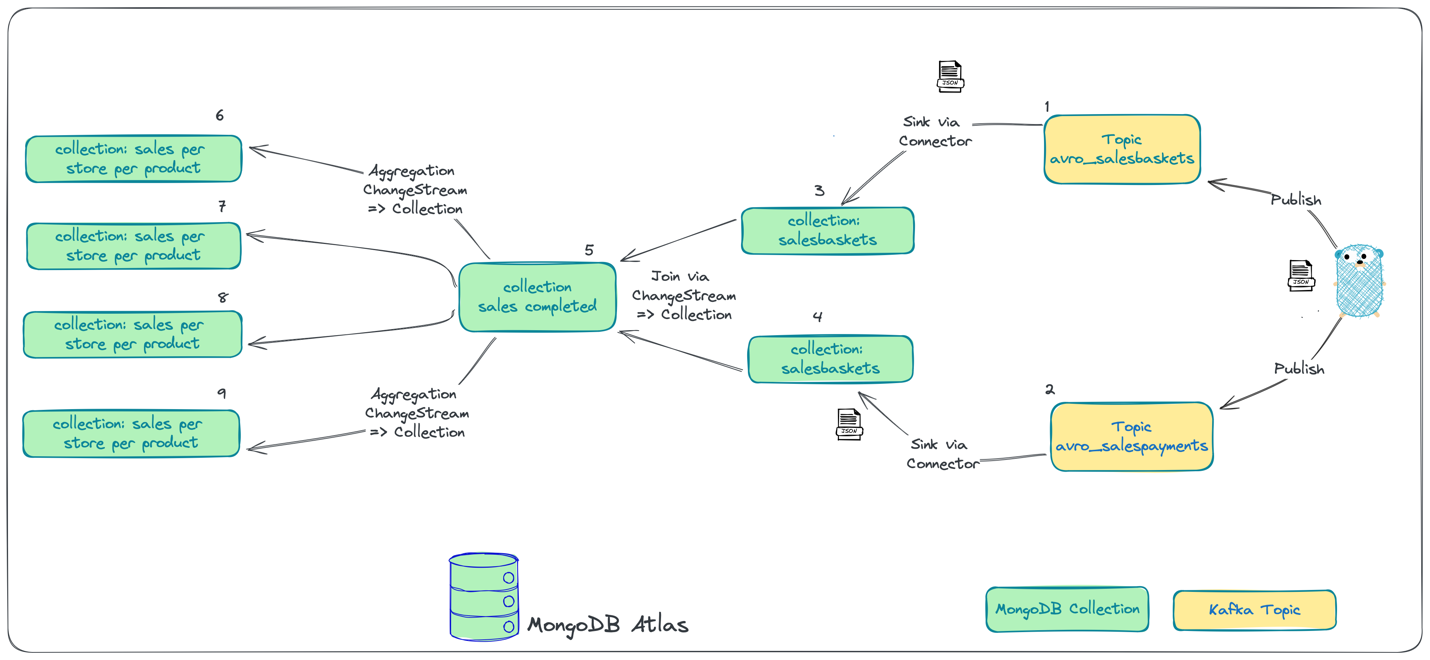
The thinking is to output the aggregated values from the [MongoDB Change Streams](https://www.mongodb.com/resources/products/capabilities/change-streams) and sink the values back into collections and then build dashboards using [MongoDB Charts](https://www.mongodb.com/products/platform/atlas-charts?utm_source=google&utm_campaign=search_gs_pl_evergreen_charts_product_prosp-brand_gic-null_ww-multi_ps-all_desktop_eng_lead&utm_term=mongodb%20chart&utm_medium=cpc_paid_search&utm_ad=e&utm_ad_campaign_id=11752580143&adgroup=109258985690&cq_cmp=11752580143&gad_source=1&gbraid=0AAAAADQ14013RIZMetFEUonZso5BgfPY0&gclid=Cj0KCQjw-uK0BhC0ARIsANQtgGMXFe_hhYL7GNePybQ_q4m9zrxYwI7M20FvZ4NFP663Aj20M7TVWE0aAuioEALw_wcB), all nicely inside the MongoDB Atlas platform.



The very big obvious bit here is, it’s very simple, it all local inside the [MongoDB](https://www.mongodb.com/) eco system, it’s all build on the very well know [MongoDB](https://www.mongodb.com/) aggregation framework.



Edit: (15 Aug) Removed “unnested\_sales” collection as it’s not required in MongoDB Atlas for computations/aggregations on nested objects/arrays in a document. Above is the before, below is the after edit.

****

**Apache Flink Catalog**

See [catalogs-in-flink-sql-a-primer](https://www.decodable.co/blog/catalogs-in-flink-sql-a-primer) & [catalogs-in-flink-sql-hands-on](https://www.decodable.co/blog/catalogs-in-flink-sql-hands-on) by Robbin Moffatt. You will notice in his Decodable GIT repo he also shows how to do a catalog using Hive and how to use PostgreSQL and JDBC driver.

**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, Streaming data in the analytical world. – Part 5

## Apache Kafka, Apache Flink, Apache Iceberg on S3, Apache Paimon on HDFS, Apache Hive with internal standalone metastore (DerbyDB), external on PostgreSQL & on HDFS.

### (See Part 4)

**Data Persistence**

Well, once we’ve calculated everything, we now want to store the data.

One way is to publish all data back onto the Kafka cluster into topics and then utilize Kafka Connect framework, with this we can push any value/topic from Kafka into a data store like [MongoDB Atlas](https://www.mongodb.com/).

Another option, and this is more for the data warehouse/data lake/lake house/analytics world, push it directly from [Apache Flink](https://flink.apache.org/).

To explore this further, I ended doing XXX additional mini PoV’s.

1. [Apache Flink](https://flink.apache.org/) pushing into Apache Iceberg based store with storage provided on AWS S3 via a local [MinIO](http://min.io/) container.





1. [Apache Flink](https://flink.apache.org/) pushing into [Apache Paimon](https://paimon.apache.org/) based store with storage provided on Apache Hadoop DFS via local Hadoop cluster deployed via containers.





For both the file format can be selected as either [avro](http://avro.apache.org/), [parquet](https://parquet.apache.org/) or [orc](https://orc.apache.org/).

**Lesson**: Little Catch here, you can create a default file format by specifying it in the catalog create (simplifies the create table as select: CTAS statements) or you can specify it at table create time.

As we’re working on [Apache Paimon](https://paimon.apache.org/), [Apache Parquet](https://parquet.apache.org/) is generally accepted.

as the “industry” default Open Table Format (OTF).

But but but... Parquet does not seem to handle complex JSON objects… don’t know if this document anywhere, just figured it out by illumination/trial and error. Avro and Orc seem to work… All 3 formats did fine with a flat table. – TO BE CONFIRMED!

# An exercise in Discovery, Streaming data in the analytical world. – Part 6

## Apache Kafka, Apache Flink, Apache Iceberg on S3, Apache Paimon on HDFS, Apache Hive with internal standalone metastore (DerbyDB), external on PostgreSQL & on HDFS.

### (See Part 5)

So now we are using what everyone says is the best serialisation across streaming architecture and, we have rich support across various stacks and into the data warehouse.

The last bit of scope creep took us down the part of how to store the data, where we looked at the current favourite “[Apache Iceberg](https://iceberg.apache.org/)” (lake house … monster) and the new kid on the block “[Apache Paimon](https://paimon.apache.org/)” which places us smack back in the middle of what’s starting to be referred as streamhouse.

Lesson’s / Take Aways

1. Case matters, ignore at your own peril.
2. Schemas are important (very) and can be helpful, and can be pain, accept it, it’s worth it.
3. It’s fun doing different versions, i.e. JSON -> [Protobuf](https://protobuf.dev/)-> Avro and you learn allot along the way, pro’s and cons of each, differences in implementation, the capability tore serialize from one format to another. If I just went from the start and just did Avro then this all would have been lost, so be willing to do thigs multiple times.
4. The difference between message structure as created by producer (that you work with and will store) and the byte stream created and moved around via the Kafka serializer across the topics.
5. It helps to skill up on [docker compose](https://docs.docker.com/compose/) and [Dockerfile’s](https://docs.docker.com/reference/dockerfile/) and how to build images, how to add Libraries, how to source JAR’s from [Maven](https://mvnrepository.com/) [repository](https://repo.maven.apache.org/maven2/org/apache/flink/). The following document helped, [Flink SQL and the Joy of JAR’s.](https://www.decodable.co/blog/flink-sql-and-the-joy-of-jars#putting-the-jar-in-the-right-place)
6. If you going to be building and rebuilding and deploying and redeploying your lab over and over and over, consider staging the various tar.gz and jar files locally, it just speeds up your cadence, my builds for now use a mix of locally staged files in stage directories and direct [curl](https://curl.se/) and/or [wget](https://www.gnu.org/software/wget/) steps.

TODO’s

1. Create Python app that consume the Kafka topics and echo the output to the terminal.
2. Create a Python app that query the MongoDB Atlas salescompleted collection, aggregating stats per store (based on a store filter supplied) and reports the value for the day, total sales, total number of baskets, min, max avg and medium basket count and value per salesterminal, best-selling product, best-selling brand, terminal with most baskets, terminal with higher avg baskets per hour… This is all to demonstrate the Aggregation framework available in MongoDB.
3. … let me stop, otherwise this might never end ;), it did not end…
4. Introduce partitioning on the source [Apache Flink](https://flink.apache.org/) table based on the Kafka topic Key. Partitioning as a implementation has been depreciated and replaced by the concept referred to as: “[distributed by](https://docs.confluent.io/cloud/current/flink/reference/statements/create-table.html)”.
5. Containerize the Golang Application (In process) or refactor the data creation process via [ShadowTraffic](https://shadowtraffic.io/).
6. Well scope creep… again, was asked to see how to do [Apache Paimon](https://paimon.apache.org/) as a destination data store, aligning with the Streamhouse concept => Done ;)
7. Explore [Apache Flink](https://flink.apache.org/) CDC as an alternate inbound (directly from data stores) and outbound data source/exporter, publishing data onto Kafka topics, sinking into [MongoDB Atlas](http://mongodb.com/) database, [Apache Iceberg](https://iceberg.apache.org/) and [Apache Paimon](https://paimon.apache.org/) tables, etc.
8. Consider additional output options; [Apache Pinot](https://pinot.apache.org/), [Starrocks](https://www.starrocks.io/), [Apache Druid](https://druid.apache.org/) (by [Imply](https://imply.io/)).
9. Consider pushing the aggregate business metrics via a Confluent Connector into a [Time Series DB (TSDB)](https://en.wikipedia.org/wiki/Time_series_database) like [MongoDB Time Series](https://www.mongodb.com/docs/v5.3/core/timeseries-collections/) or [Prometheus](https://www.confluent.io/hub/confluentinc/kafka-connect-prometheus-metrics), (there are more), Allowing for real time business insight via [Grafana](https://grafana.com/) Dashboards.

**Credit’s**

Some People I do feel I need to mention, that went out of their way to help, with advice, guidance and at times just a sound board.

Without these guys and their willingness to entertain allot of questions and sometimes simply dumb ideas and helping me slowly onto the right path all of this would simply not have been possible.

[Dave Troiano,](https://www.linkedin.com/in/dave-troiano-49a8932/)

(Developer support on Confluent Forum @dtroiano),

<https://www.linkedin.com/in/dave-troiano-49a8932/>

[Barry Evans,](https://confluentcommunity.slack.com/team/U04UNKMRL4U)

Someone that I consider a friend, just stepped in, started helping me and as he happily calls it his community service. Helping others figure problems out that they have, whatever the nature, and another always curious mind himself.

<https://confluentcommunity.slack.com/team/U04UNKMRL4U>

[Martijn Visser,](https://apache-flink.slack.com/team/U03GADV9USX)

[Apache Flink](https://flink.apache.org/) Slack Community

(PMC and Committer for [Apache Flink](https://flink.apache.org/), Product Manager at Confluent)

<https://apache-flink.slack.com/team/U03GADV9USX>

[Ben Gamble,](https://confluentcommunity.slack.com/team/U03R0RG6CHZ)

Apache Kafka, [Apache Flink](https://flink.apache.org/), streaming and stuff (as he calls it)

A good friend, that’s always great to chat to… and we seldom stick to original topic.

<https://confluentcommunity.slack.com/team/U03R0RG6CHZ>

[Robin Moffat](https://rmoff.net/)

Many many blog posts and examples available from his GIT repo link in posts. A true resource you can’t be without in our industry.

**My Repo’s**

Originally, I started with a JSON structure, serialized version, with no schema registry. Performance was north of 10 000 txn/second. Then I realized for kStream and kSQL I really require a schema… Introduce the little devil which whispered in my ear let’s see what this thing called the schema registry is all about. Well performance dropped to 500 txn/second, unimpressed, but let’s move ahead.

So, I’ve long since heard Protobuf’s are the dope, it’s fast & payload is small as the schema is not transmitted with the payload. So, let’s refactor app into Protobuf structured, using a Confluent Kafka Protobuf serialization client, interfacing now with the schema registry.

Great, we’re back at 8000tx/second. Issue now… Support/adoption in the data warehouse/lake house… world, by default included libraries inside Apache Flink for one is not… First work around, on Kafka cluster create Protobuf serialized stream, and then create Avro serialized stream sourcing from Pb serialized, it works… but there must be a better way.

Ok, I’ve heard about this thing called Avro, lets refactor again, surprise, was allot more complicated to get working than originally expected… but eventually got it working, and we’re back at 8000+txn/second.

Below is all 3, if anyone cares to compare.

* Version 1: JSON payload – We started here

<https://github.com/georgelza/MongoCreator-GoProducer-json>

* Version 2: JSON Protobuf – 2nd progression

<https://github.com/georgelza/MongoCreator-GoProducer-pb>

* Version 3: JSON Avro – Final version build around Avro as a source Serialization.

<https://github.com/georgelza/MongoCreator-GoProducer-avro>

In the main root branch of the repo is an infrastructure directory where all the source images are build using docker, which is then utilised inside the various versions of the project.

See the [Makefile](https://opensource.com/article/18/8/what-how-makefile" \t "_blank) in the same directory, which will first pull the source images, after which the various images can be build. Remember to go change the base OS architecture if you not on a MBP and also look at the JAR files pulled, some of them are arm64/aarch64 specific.

Below is a diagram depicting the ancestry of the various images used inside the project.



And that’s it for now… Thank you for sticking with me through this exploration. All it did was create a small little to do list that is growing as I am typing this, of things/subjects I’d like to explore more and blog . 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)

Some more References:

[Apache Flink](https://flink.apache.org/) originally by [Ververica](https://docs.ververica.com/)

* [Get Started - Installation](https://docs.ververica.com/vvp/getting-started/installation?_gl=1*a1ub31*_gcl_au*MjI3NTE0OTU0LjE3MjA2OTY4NDY).
* [flink-sql-joins-part-1](https://www.ververica.com/blog/flink-sql-joins-part-1)
* [flink-sql-joins-part-2](https://www.ververica.com/blog/flink-sql-joins-part-2-0)

Open Table Format’s

* [Open Table Formats — Delta, Iceberg & Hudi](https://medium.com/geekculture/open-table-formats-delta-iceberg-hudi-732f682ec0bb)
* [Understanding Apache Iceberg architecture | Starburst Academy](https://www.youtube.com/watch?v=xfAYLAFCLvM) - YouTube
* [Apache Paimon: A fresh face joins the fray](https://medium.com/@balachandar-paulraj/apache-paimon-a-fresh-face-joins-the-fray-310a89fc0da4)
* [Table Format Comparisons - How Do The Table Formats Represent The Canonical Set Of Files?](https://jack-vanlightly.com/blog/2024/8/7/table-format-comparisons-how-do-the-table-formats-represent-the-canonical-set-of-files)

Apache Parquet

* I Found the following YouTube video that’s easy to follow that explains the Apache Parquet file format. [Parquet File Format - Explained to a 5year old](https://www.youtube.com/watch?v=5NA57Pfpdr4)
* [An introduction to Apache Parquet](https://www.youtube.com/watch?v=KLFadWdomyI)

You have data stored in parquet… but don’t know the schema.

* [Extract schema from parquet file](https://pypi.org/project/parquet-tools/)
* [Querying Parquet Metadata using DuckDB](https://duckdb.org/docs/data/parquet/metadata.html)

Apache Avro File: Format and/or Serialization…

* [What is Apache Avro file?](https://www.youtube.com/watch?v=xfI4mXjW9oE)

How do we query our data in iceberg… I’m not a big PostgreSQL fanatic… but you can’t ignore the article… We are already using PostgreSQL as the datastore behind the Hive Metastore, so using it even further for this just makes sense.

* [Putting DuckDB in Postgres to Query Iceberg](https://blog.paradedb.com/pages/iceberg_lakehouse)
* [Querying Parquet with Precision Using DuckDB](https://duckdb.org/2021/06/25/querying-parquet.html)
* [PostgreSQL For search and Analytics](https://docs.paradedb.com/analytics/quickstart)
  + [ParadeDB](https://www.paradedb.com/) is not a fork of Postgres, but regular Postgres with custom extensions installed. ParadeDB itself ships with Postgres 16