🡪 Consider splitting into multiple articles…

1. Confluent Kafka => kSql => MongoDB Atlas =>Change Streams
2. Confluent Kafka => Apache Flink (FlinkSql) => MongoDB Atlas

2 Confluent Kafka => Apache Flink (FlinkSql) => Apache Iceberg

1. Apache Hive Metastore Standalone with Catalog on DerbyDB and
2. Apache Hive Metastore Standalone with Catalog on PostgreSql

3 Confluent Kafka => Apache Flink (FlinkSql) => Apache Paimon

1. Apache Hive Server with file store located on Hadoop DFS.
2. Apache Hive Server with Metastore Catalog on PostgreSql

**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 game, 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 salesbasket topic and then create a salespayment record, associated with the basket, posted onto a separate salespayment 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… Added a sink to Apache Iceberg using Apache Parquet OTF (Open Table Format), hosted on AWS S3 compatible storage (provided locally using Minio). Also included is a catalog store via Apache Hive (with the Metastore stand alone and backed by a PostgreSql DB). Other words the goto lakehouse type of architecture. But then, more scope creep, I decided to add a HDFS store and use [Apache Paimon](https://paimon.apache.org/docs/master/) as datastore, storing the data as either avro, parquet or orc, we now talking streaming lakehouse. See [The majesty of Apache Flink and Paimon](https://medium.com/@ipolyzos_/the-majesty-of-apache-flink-and-paimon-d36e73571fc9) for a nice writeup.

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

By [George Leonard](https://www.linkedin.com/in/george-leonard-945b502/) – 17 July 2024

**Sections**

Source data generator

Golang app… -> Still want to containerise this to make it easier for this project, it is not the focus.

Data sets:

Salesbasket

{

"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 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

},

{

"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

}

Salespayments

{

"InvoiceNumber": "1341243123341232",

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

"PayTimetamp\_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, 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

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 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 Stream processing.

Docker compose.yml

Everything was done using docker images and deployed via docker compose.

There are basically 5 sub projects:

* devlab-mongo
  + Simply publish to Kafka topic, use sink connector to MongoDB.
* devlab-hms-standalone
  + Publish to Kafka, use some kSql and Apache Flink Sql to do aggregations, pushing calculated values back onto Kafka Topics. We also now push the values onto Apache Iceberg, storage on AWS S3 via [MinIO](https://min.io/). This version utilized a stand-alone [Apache Hive Meta](https://hive.apache.org/) store with internal DerbyDb.
* devlab-hms-postgres
  + As per above, with the [DerbyDb](https://db.apache.org/derby/) swopped out for a PostgreSql DB.
* devlab-hdfs-paimon-hdfs-cat
  + Here we push output data onto Apache Paimon, storage via Hadoop DFS, Paimon catalog on the HDFS storage, Flink Catalog still on HMS as per above.
* devlab-hdfs-paimon-hms-cat
  + As per above, but we now relocating the Paimon catalog also onto HMS with a PostgreSql back end.

*Note: 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:

*Note: as you going to run multiple Flink 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.

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

Broker

…

kSql

kStream

Connect

Custom Connect container with additional source/sink connectors installed.

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

Little Note, all of Apache Flink 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 tables, and exit Flink, when you restart it, all your tables/jobs are gone, and you will need to recreate them.

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 whats 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 against.

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

Let’s complete our environment for a future use case, might want to source or sink records from the database.

Perfect to sink the unnested\_sales into

[MySql](https://www.mysql.com/)

Let’s complete our environment for a future use case, might want to source or sink records from the database.

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

AWS Compatible S3

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

Minio Console by Min.io

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

Option as a Apache Flink 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 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 supports 3 types of catalogs, namely:

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

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 high level” version after which I discuss the “Apache Flink” option:



Next up is a more detailed view of what we’re going to be doing using kSql inside the Apache Kafka cluster to create kStreams and kTables objects.



Aggregations via kSql and 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 to create a new kSql object calls salescompleted, as a join between salesbaskets and salespayment, based on the invoiceNumber column.

The salescompleted stream is then used to create kTable objects, which output (using tumbling windows):

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

See kSql directory for the sql utilized to create the various kStream and kTable 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 aggregations, the below creates a 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:



In this scenario we use Apache Flink to mirror some of what was done above using kSql, but this time using Apache Flink Sql, and some additional magic.

In the Flink case we do things in 2 steps (I just found it easier), first we create a Flink 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 Flink 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 Flink 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 SQL run on the Flink 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, ie.:

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

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.

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.

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 happily also be sink’d into a “old style” RDBMS, even thought our old-style RDBMS database engines themselves are extending their capabilities to include storage of JSON structured records.

Data serialization format

First there was [Protobuf](https://protobuf.dev/) and then [Avro](https://avro.apache.org/) came.

Avro Schema

A word of warning: Case Sensitivity between Apache Flink 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.

Examples:

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

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 going to output the group by into this 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 hour / per day
* 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 Collections
* Sink (push) to Kafka Topics
* Source (pull) from Mongo

Dashboards/Charts in MongoDB

Thinking was to use the output from the [MongoDB Change Streams](https://www.mongodb.com/resources/products/capabilities/change-streams) and sink the aggregated values back into collections and then dashboard this 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).



Apache Flink Catalog

See <https://www.decodable.co/blog/catalogs-in-flink-sql-a-primer> and <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.

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.

Another option, and this is more for the data warehouse/data lake/lake house/analytics world, push it directly from Apache Flink.

To explore this further, I ended doing 2 additional mini PoV’s.

1. Apache Flink pushing into Iceberg based store with storage provided on AWS S3 via a local MinIO container.



1. Apache Flink pushing into Paimon 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, parquet or orc.

NOTE: Little Catch here, you can create a default file format by specifying it in the catalog create (simplifies the CTAS statements), as we’re working on Apache Paimon, Apache Parquet is assumed as the “industry” default OTF.

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

Lesson’s

1. Case matters,
2. Schemas are important, and can be pain,
3. It’s fun doing different versions, i.e. JSON -> Protobuf-> 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 and Dockerfile’s 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.

Todo

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 table based on the Kafka topic Key. Partitioning as a implementation has been depreciated and replaced by “[distributed by](https://docs.confluent.io/cloud/current/flink/reference/statements/create-table.html)” concepts.
5. Containerize the Golang Application. => In process.
6. Well scope creep… again, was asked to see how to do [Apache Paimon](https://paimon.apache.org/) as a data store. => Done ;)

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 Slack Community

(PMC and Committer for Apache Flink, Product Manager at Confluent)

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

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

Apache Kafka, Apache Flink, 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. So now we are using what everyone says is the best serialization across streaming architecture and, we have rich support across various stacks and the data warehouse, lake house … monster.

Below is all 3, if anyone cares to compare.

Version 1: JSON payload

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

Version 2: JSON Protobuf

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

Version 3: JSON Avro

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

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

See the Makefile in the same directory, which will first pull the source images, after which the various images can be build. Below is a diagram depicting the ancestry of the various images used inside the project.



**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)

References

Apache Flink 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

* <https://medium.com/geekculture/open-table-formats-delta-iceberg-hudi-732f682ec0bb>
* <https://jack-vanlightly.com/blog/2024/8/7/table-format-comparisons-how-do-the-table-formats-represent-the-canonical-set-of-files>