**Overview**

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

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, with further enrichment, Dashboard the various values, and make it available for downstream consumers.

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

**Sections**

Source data generator

Golang app…

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 Dashboards). 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.

docker-compose.yml

To include the project name as part of container and host names.

Create a .env file in the same directory as the docker-compose.yaml and insert the following: COMPOSE\_PROJECT\_NAME: <project name>

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

hostname: ${COMPOSE\_PROJECT\_NAME}-control-center

container\_name: ${COMPOSE\_PROJECT\_NAME}-control-center

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}\_default

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.

[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:1.16.0-scala\_2.12-java11, this image contains all of: jobmanager, taskmanager and sql\_client.

Apache Flink is package by various groups, i.e. Ververita, Confluent:

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 persistence.

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

[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.

[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.

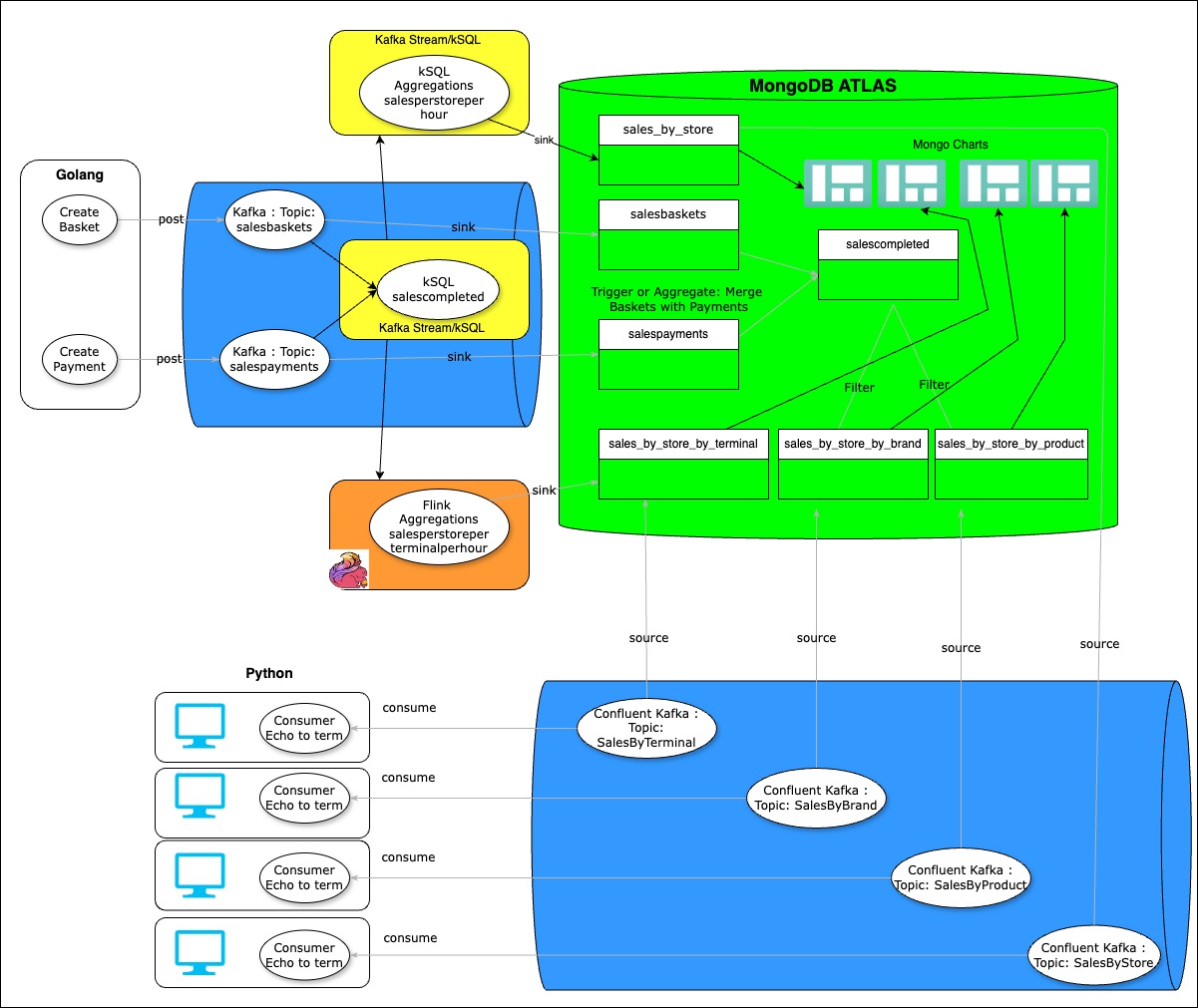
Kcat

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.

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.



Aggregations via kSql and kTable

What I do here is use ksql to first create a stream object for the salesbaskets, followed by a salespayment topic.

We then use kSql to create a new kSql object calles salescompleted, as a join between salesbaskets and salespayment, which is joined on the invoiceNumber column.

The salescompleted stream is then used to create kTable objects, which output:

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

See kSql directory for the sql utilized.

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 Flink Sql.

In the Flink case we do it in 2 steps, first we create a Flink table, or as I think of it, a virtual table. Why do I call it a virtual table, because the create itself points to the salesbaskets topic or salespayments topic. When interacting with the Flink table it engages a Kafka consumer from 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 on source/read or append/output to the back’ing topic. Pretty much how Kafka works as an immutable log.

Once the 2 source tables are created we create a 3rd output table called salescompleted. An Insert/join statement is then executed that join the 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 begin.

From here we do aggregations, first up was sales per store per terminal per 5 min. Again we create a 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 hour,

sales per store per brand per hour

sales per store per category per hour

What I have not mentioned above, if you look at the basketItems array of objects, you will realise it’s a nested data set, which require that we first unnest the basketItems. This is done by creating a table that is flat, a record will be inserted into this table for each object from th 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. To improvement performance, we include a filter to run against recent data.

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 sinked 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 and then Avro came.

Avro Schema

A word of warning: Case Sensitivity between Flink Sql’s: Create table <>, Select <> From <> and Kafka schema registry entries matters.

Small pieces of work.

Aggregations via Mongo Stream Processing

* Sales per store per hour
* Sales per store 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.

Results:

Sink (push) to Collections

Sink (push) to Kafka Topics

Source (pull) from Mongo

Dashboards/Graphs in Mongo

Thinking was to use the output from the Streams engine, and sink the aggregated states per window back into a collection and then dashboard this using Mongo graphs.

Lesson’s

1. Case matters,
2. Schema’s 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.

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 ;)

Credit’s

Some People I do feel I need to mention, that went out of their way to help, with advise, 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,

(Developer support on Confluent Forum @dtroiano),

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

Barry Evans,

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,

Apache Flink Slack Community

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

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

Ben Gamble,

Apache Kafka, Apache Flink, streaming and stuff (as he calls it)

A good friend, thats always great to chat to... and we seldom stick to original topic.

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

Repo’s

Originally, I started with a JSON structure, serialized version, with no schema registry. Performance was north of 10 000 txn/second.

Then a little devil whispered in my ear lets add schema registry as all good practices/papers advised. Well performance dropped to 500 txn/second.

So I’ve long since heard Protobuf’s are the dope, it’s fast, performing so let’s refactor app into Protobuf structured, using a Confluent Kafka serialization client, plugging into the and then the associated schema registry changes. Great, we’re back at 8000tx/second. Issue now… Support, by default included libraries inside Apache Flink for one is not… Work around, on Kafka cluster create an Avro serialized stream 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 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.

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>

**About Me**

I’m a techie, a technologist, 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.

Recent years, pivoted to a more generic Technology Architect capable of full stack architecture.

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

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