Semantic Pub and Sub with Apache Kafka

Extending Topics from Keywords to Patterns

Shivam Gulati
Masters in Computer Science Student, NCSU
Raleigh, NC
sgulati2@ncsu.edu

Isha Bobra
Masters in Computer Science Student, NCSU
Raleigh, NC
ibobra@ncsu.edu

David Ball
Masters in Computer Science Student, NCSU
Raleigh, NC
dball@ncsu.edu

***Abstract* – Kafka [1] is used widely in multiple organizations to manage the flow of real time data. It is essential that we focus on how Kafka manages and stores this data so that it can be used in the most efficient way possible by its users. One of the essential component of the Kafka system is how it manages the real time streams and segregates them so that all related messages are grouped together. The Kafka system at present, groups the messages from the producers and calls each of these groups as "Topic". Consumers are allowed to subscribe to as many "Topics" as they like and receive messages from these topics in a timely fashion as produced by the producers. These messages are ordered based on the time stamps and are available to consume by the consumers as soon as the publishers publish them. With the current implementation of Kafka, consumers are allowed to subscribe to a pattern, such that they receive messages from all the topics which belong to that pattern without subscribing to the topics individually. The constraint we see here is that the consumer's subscription is limited to a regex pattern. What if the consumer wanted to receive all messages published under a "Topic" matching any city in North Carolina? Is that a reasonable regular expression? We don't think so and we also feel that such rules could be implemented by the broker itself, taking the logic out of the individual consumers. We aim to propose a system in which we create a hierarchy of topics and show that if a consumer subscribes to a topic, the consumer also receives messages from all the topics which are a below that topic in the hierarchy. This way only by subscribing to a high level topic, the consumer automatically gets subscribed to its low level topics following the superclass-subclass model.**

Keywords- Apache; Kafka; topics; publish; subscribe; semantics; mapping; location; cluster; architecture; experimentation; analysis

I. INTRODUCTION

A. Introduction to Kafka

Apache Kafka, developed by LinkedIn [2] is a widely used messaging system. It has several important components: Producers, Consumers, Topics, Partitions and Brokers. We need to understand each one of them in detail to get a clear picture of how this messaging system works.

It follows a publish-subscribe mechanism, making it easy and simple to manage, store and transfer the real time streaming data produced by various sources, called "Producers". Kafka stores these messages segregated in groups. Each group is known by its name, called a "Topic" and each of these topics are further divided into "Partitions". The messages stored in Kafka are used by clients called "Consumers". The messages are sorted based on the content and the publishers, and are arranged in a sorted manner based on the timestamps. The consumers can subscribe to any number of topics and are guaranteed access to all the messages published under those topics. The consumers maintain an offset which they set to point to the latest message consumed by them from the message queue. They can navigate in any way they like using this offset. It means that they can re-consume a message or skip a few messages depending on their requirements. The messages stays in the Kafka memory for a predefined amount of time, which can be configured by changing the configuration setting of the Kafka cluster. Kafka cluster consists of several servers, which are called "Brokers".

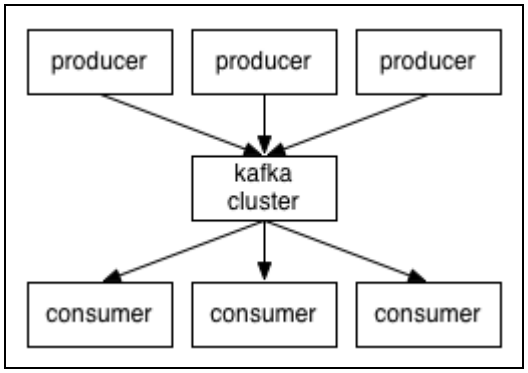

Figure 1: Architecture of Kafka

Figure 1 explains the high-level architecture of Kafka. Many producers push the data to the Kafka cluster and many consumers consume messages from the Kafka Cluster [3] [4]. To understand how the messages are collected and grouped, refer to figure 2.

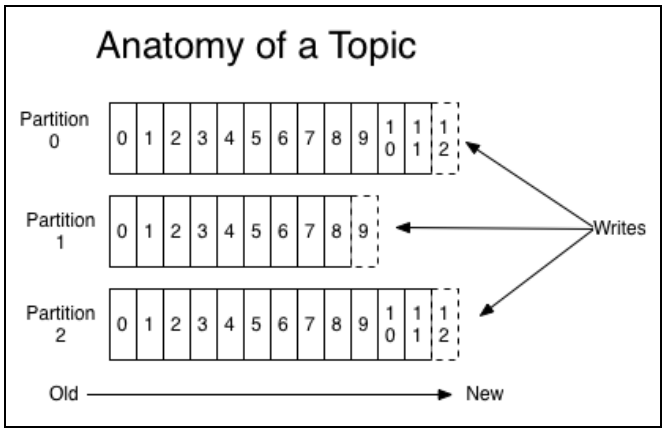

Figure 2

In figure 2, we can see that each topic is divided into several partitions. Messages are immutable and get appended to the end of the queue with an offset to maintain the ordering. These offsets can be used to identify the messages uniquely within a partition.

B. Distributed Architecture

Kafka [5] [6] follows a distributed architecture and each of these partitions are replicated over several servers. One of the server is chosen as the leader for each partition and the leader is responsible for managing the reads and writes on that partition. The rest of the servers on which the partition is replicated are treated as the followers and they just replicate the changes that the leader makes. If the leader is down, one of the followers will take its place.

C. Mechanism

The consumers are grouped together into various consumer groups [7]. Each of these groups are subscribed to several topics. When a message is published in a topic, one of the consumer in each of the consumer groups gets that message. The load is balanced internally within a consumer group. Figure 3 explains the publish subscribe mechanism clearly.

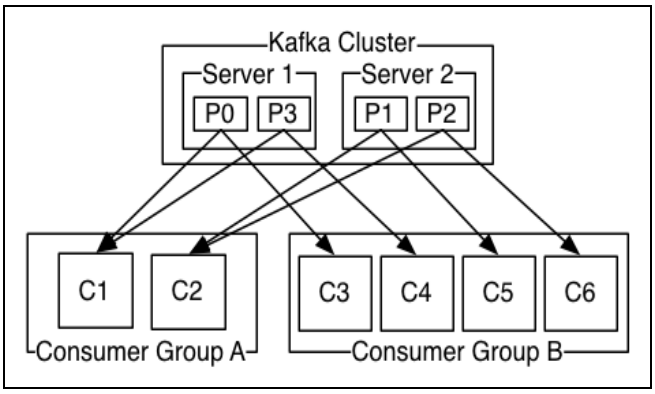

Figure 3: Publish subscribe mechanism of Kafka

II. RELATED WORK

The foundation for this project revolves around the semantic web, W3C Web Ontology Language (OWL) [8], and Resource Description Framework (RDF) work, in which there are many relationships between subjects and objects that can be derived from information outside of each particular RDF triple. One of the classic examples is when it can be established by some other triple that a subject is a subclass of some other object. In that case, it is now true that the new object, original property, original object is valid in addition to the original subject, original property, and original object triple. While our work will concentrate on the subclass/superclass relationship, we believe that it will be extendable to other relationships as well, such as the domain and range relationships.

Currently, Apache Kafka supports partitioning, where a topic is partitioned based on some key into segments. This allows a consumer to subscribe to only some subset of the messages published to a particular topic. Kafka also supports pattern matching subscriptions, where the consumer provides a regular expression and is subsequently subscribed to all topics matching that expression. We can see from Kafka's current implementation of partitioning and pattern matching subscriptions that the desire for splitting or combining topics exists within the community. Each of these approaches has drawbacks, requiring each consumer to implement the solution that meets their needs. In the case of partitioning, all consumers must be aware of the partitioning scheme being implemented by the broker and customize their code to subscribe to the correct partitions. With pattern matching subscriptions, again the consumer must customize their code to create a regular expression which will match all topics they wish to consume.

III. DETAILS

A. Initial Work

Our aim is to allow a consumer to subscribe to a superclass, also getting messages published to all subclasses, rather than needing to subscribe to each of those subclass

topics individually. The initial architecture that we came up with was as shown in figure 4.

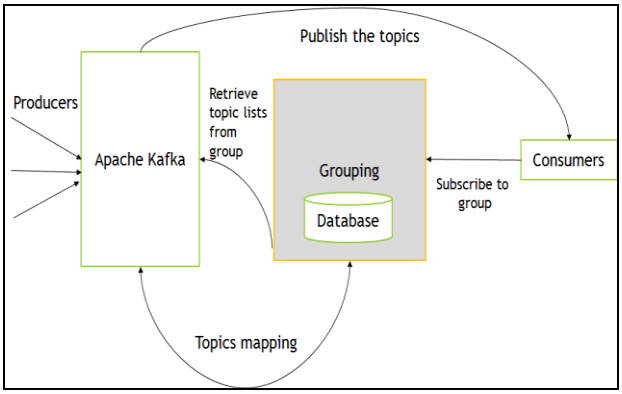

Figure 4: Initial structure

We thought of creating a system that would be placed in between the consumers and the producers which would do the mapping between the subclasses and super classes. The solid block shown in the diagram is our system. The flow is as follows:

- The consumer tells us which topics it needs to subscribe to.
- We maintain a database mapping the subclasses to their super classes and we create the subscription for all the subclass topics for the consumers.
- The consumers then receive messages for all those topics from the Kafka system.

The advantage of this system at first thought was that the mapping calculations were done in isolation from the actual Kafka system. But this turned out to be a disadvantage, since we had an overhead of mapping calculations, the "real time" aspect of the messaging system was lost. This kind of architecture would have created a lag in the publish-subscribe model and would not be able to be sustained in this fast-paced, real-time, streaming data analysis world.

B. Improved architecture

We came up with a better solution which would respect the real-time aspect. This solution references a properties file, which we created to maintain the mappings from subclasses to their super classes, from the Publisher class in Kafka when it is first loaded. Then each time a publisher sends a new message, the mapping is checked to see if there is a superclass for that message's topic. If a match is found, a new message is created with the original message body and the superclass's topic. By utilizing the key value pairs in the property file, the mappings are kept in memory and lookups are extremely fast.

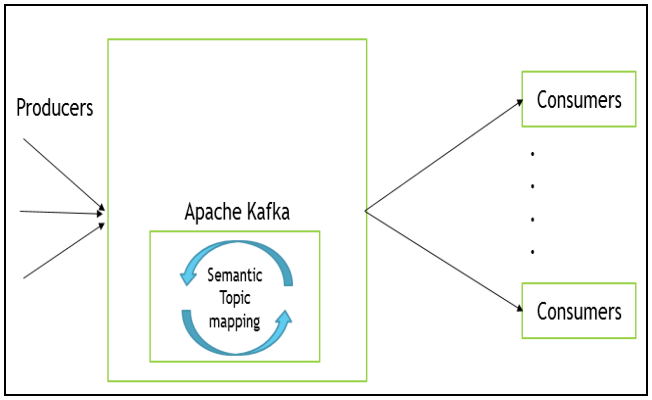

Figure 5: Improved structure

We see several advantages of this solution in addition to maintaining the real-time aspect of the Kafka messaging system which we will detail below:

- With mappings being stored in a properties file, not only are lookups extremely fast using the key-value pairs, but the mapping file can be updated at any time without rebuilding the entire code base.
- The entire solution is handled by the Kafka broker, which means that producers and consumers can enter and leave the system without any modifications to the code.
- Neither producers nor consumers need to do anything on their end for the system to function. No need to know all the zip codes and cities when all they are interested in is receiving all of the data for a particular state.
- An entire level of subclass could be added, counties or townships for example without changing anything other than the mappings.

IV. EXPERIMENT

The system configuration we used was as follows:

- 64-bit operating system
- Intel Core i7 x64-based processor
- Processor running @ 2.40GHz
- 8 GB RAM

Our experiments involved running the data streams on the above mentioned system. We thought of doing a performance analysis on how the system would respond in the basic state as Apache Kafka works and then how it would perform in our scenario where we would have semantic data stream on locations. We took the US location database [9] and created the mappings for the location semantics. We chose a set of 10 ZIP codes, one starting with each digit from 0 through 9 chosen randomly, from the database to use as test cases. Our experimentation and analysis was done in two categories:

- Time analysis [10] on variable sized US location datasets with a stream of 1000 messages posted to each of the test zip codes
- Scalability test on the entire US location dataset, altering the number of messages posted to each of the test zip codes

Our first data recording involved a stream of 1000 messages which were published to each of the ten test zip codes. There were ten consumers for this test split as 3-3-3-1, where three consumers were subscribed to three different test zip codes, three to different cities which are super classes of the test zip codes, three to states which were super classes of those cities, and to consumer was subscribed to the USA topic. We recorded the time it took for the consumers to receive all 1000 messages in each of these scenarios:

- An empty properties [11] file. The zip code consumers would be the only ones to get messages in this scenario.
- A properties file of 210 rows, with ten zip code to city mappings for each digit 0-9 of possible zip codes, ten corresponding mappings from those cities to their respective states, and 10 mappings from those states to the USA topic.
- A properties file of ~10000 rows, with randomly selected zip to city and city to state mappings, along with mappings from each state to the USA topic and specifically including all mappings relevant to our test zip codes.
- A properties file with mappings of every zip code in the USA to their respective cities and states, having approximately 72,000 rows.

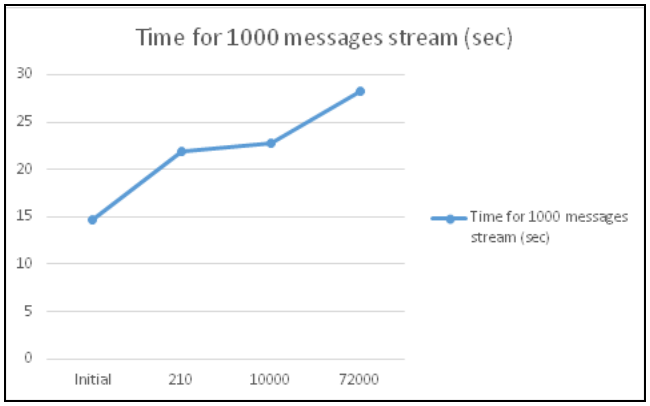

Figure 6

Location mapping size	Time for 1000 messages stream (sec)
Initial	14.7
210	21.9
10000	22.8
~72000	28.2

Table 1: Time duration for 1000 messages stream

In figure 6, you will see the average duration it took for each consumer to receive all 1000 messages. The initial duration when we did not have any USA location dataset semantics was an average of 14.7 seconds when the consumers were subscribed to ZIP codes. The consumers subscribed to cities, states, or USA did not get any messages since there were no semantic mappings for this scenario. Next, we mapped approximately 200 records from the entire USA location dataset, which had some randomly selected zip codes, cities, states, and the USA. The same ten consumers were subscribing to the set of data. This took an average of 21.9 seconds for the entire 1000 messages to be received by each consumer. Next, we used the same set of test case zip codes with a larger location dataset having 10000 rows, and duration increased to 22.8 seconds, a mere 0.9 seconds from the previous case. Now, we wanted to run it on the entire dataset and see how the timing would be affected when the dataset had approximately 72,000 rows, representing the mappings of all US zip codes, to check and match. The consumers were still subscribed to the same set of topics, and to our amazement, it took just an average of 28.2 seconds for the entire 1000 messages to be consumed. So, from the existing system which doesn't allow the semantic mapping to the entire location dataset with semantic mappings from ZIP to city, city to state and state to USA, it just took roughly 2x time the initial duration of sending 1000 messages!

We then decided to analyze how scalable the entire US location semantic mapping would be when we publish a larger or smaller number of messages [12], ranging from 1 message to 10,000 messages (figure 7). We published different number of messages in each scenario to each of the ten test topics and the consumer were subscribed to the same topic as in the first test. First, we published just 1 message (Table 1, Row 1), and as expected it took hardly any time, an average of 0.36 seconds per consumer, irrespective of the depth of the mapping, i.e. if it was a zip code, city, state, or USA. Next, we published a sequence of 1000 messages (Table 1, Row 2) and in this scenario with all the US location dataset mappings, as highlighted earlier, it took a total of 28.2 seconds. We increased the message sequence by ten-fold to 10,000 (Table 1, Row 3) messages and the total average time of consumption in this case was 136.02 seconds. Each increase in duration was significantly less than the increase in the number of messages consumed.

Number of messages	Time for message consumption (sec)
1	0.36
1000	28.2
10000	136.02

Table 2: Time duration for consumption of different sized message sequence

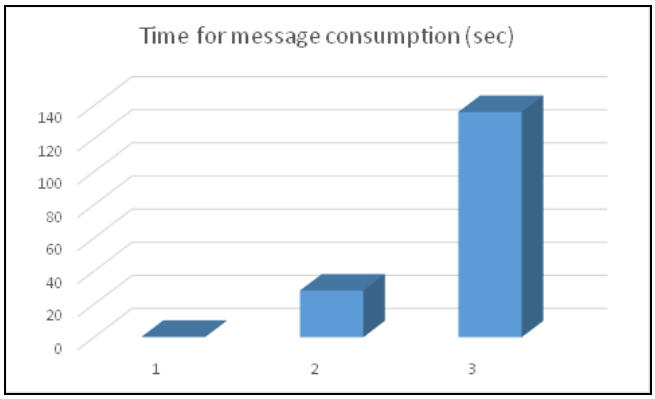

Figure 7

Another deduction we had from our experiment is that all the consumers, irrespective of the fact if they had subscribed to a zip (least level of dataset hierarchy) or city or state or USA (highest level of dataset hierarchy), the approximate time it takes for the consumers to receive the messages on a topic is approximately the same (figure 8). In our case, the duration for all the ten topics was between 28.222 seconds and 28.237 seconds. This is a good indication that the multiple level hierarchy won't adversely impact the performance.

Topic	Duration for 1000 messages(sec)
ZIP1	28.222
ZIP2	28.228
ZIP3	28.239
CITY1	28.222
CITY2	28.228
CITY3	28.238
STATE1	28.22
STATE2	28.227
STATE3	28.237
USA	28.234

Table 3: Time for 10 consumers to receive 1000 messages

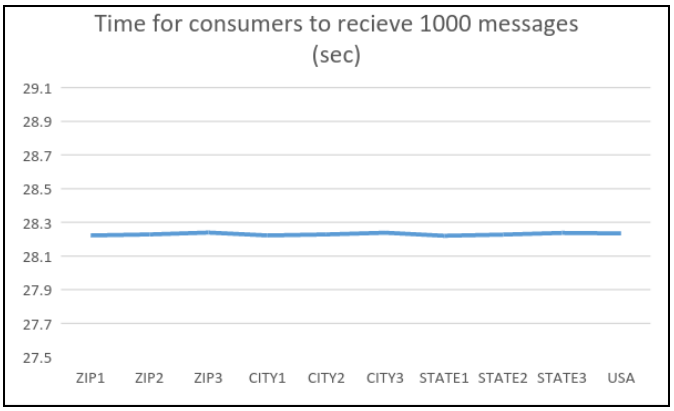

Figure 8

V. Conclusion

In conclusion, we were quite pleased with the results of our experiments. Although we had no empirical evidence to compare our results with, we felt that our results with the semantic mappings in place, publishing and consuming a total of four of message to four different topic in just about twice the amount of time taken to publish and consume on a single topic, were very encouraging. We were also pleased with the scalability of our solution, with the time taken to publish and consume ten times as many messages taking less than five times as long.

This extension to the current Kafka system could open doors for many applications which require a lot of subscriptions in order to use the existing Kafka model and would widen the scope of such applications. Applications where the consumer is just aware of the superclass topic (which might not exist in the system) and has no knowledge of the subclass topics are also handled by our proposed model. In such cases, the superclass topic is created on the fly when the consumer requests a subscription for it.

We think that we have taken an important first step toward introducing semantics into the Apache Kafka publish-subscribe architecture. While there is still significant room for improvements, it is our hope that this work will be a foundation upon which much work can be done in the future. In the last section we discuss some of our ideas for what this future work might include.

VI. Future Work

Even though we demonstrated only subclass/superclass relationships between the topics, this model could be extended to different types of relationships as well. Experiments would need to be conducted in order to achieve further extensions to prove that there is no lag involved and the real-time aspect is maintained. Since we have only

demonstrated the functionality using a superclass-subclass relation, future work would include extending this to a set of topics which have a tree like or graph like structure depicting the relationship between them. In our proposal we have considered that the mappings which show the relation between the topics are known to us beforehand. More work can be done to produce these mappings on the go. Many applications might not be able to create a complete mapping beforehand, and hence proposing a way to create the mappings on the fly could prove beneficial. Certain applications may require more mappings that would fit in main memory, so an embedded database model could be explored.

ACKNOWLEDGMENT

We would like to express our appreciation to Dr. Kemafor Anyanwu Ogan who has been helpful on how to go about the project process, scenarios, architectural changes and data collection. We thank our fellow classmates for presenting various ideas throughout the course which helped us gain better insights into Apache tools and topics, and the same led us to produce better results for this research project and paper.

REFERENCES

[1] Apache Kafka documentation
http://kafka.apache.org/documentation.htm

[2] Apache Kafka's code repository
https://github.com/apache/kafka

[3] Putting Apache Kafka To Use: A Practical Guide to Building a Stream Data Platform by Jay Kreps
http://www.confluent.io/blog/stream-data-platform-1/

[4] Message Distribution and Topic Partitioning in Kafka
https://www.jakubkorab.net/2015/12/message-distribution-and-topic-partitioning-in-kafka.html

[5] Putting Apache Kafka To Use: A Practical Guide to Building a Stream Data Platform by Jay Kreps
http://www.confluent.io/blog/stream-data-platform-2/

[6] Kafka: a Distributed Messaging System for Log Processing by Jay Kreps, Neha Narkhede and Jun Rao
http://research.microsoft.com/en-us/um/people/srikanth/netdb11/netdb11papers/netdb11-final12.pdf

[7] Apache Kafka: Next Generation Distributed Messaging System Khin Me Me Thein
http://ijsetr.com/uploads/654213IJSETR3636-621.pdf

[8] Web Ontology Language: OWL
http://my-msc-dissetation.googlecode.com/svn/trunk/Thesis/update1/owl%20overview.pdf

[9] Complete List of United State Zip Codes
https://www.aggdata.com/node/86

[10] Benchmarking Apache Kafka: 2 Million Writes Per Second by Jay Kreps
https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines

[11] Our GitHub Repository with updated architecture code
https://github.com/d-ball/kafka/tree/Isha

[12] Repository for our expiermentation publisher and consumer code
https://github.com/d-ball/KafkaDemo

[13] IEEE Big Data conference template
http://cci.drexel.edu/bigdata/bigdata2016/PaperSubmission.html