Real-time E-commerce Analytics

Team 8

Sai Sri Harsha Kanamanapalli

Srikara Krishna Kanduri

Srinivas Parasa

CSC 591 Data Intensive Computing

January 2, 2019

Abstract

This paper discusses building a data-intensive pipeline that processes e-commerce data, specifically customer order data, to provide aggregated product ratings. These aggregations can be also used to compare product-wise, company-wise ratings and sales. The process is as follows; Data is generated synthetically at a single (or multiple) sources that include customer order information in the form of an object. These data objects are streamed to multiple Apache Kafka brokers which buffer this data and send it to Apache Spark for processing. Multiple Spark clusters perform aggregations on the ratings attribute while putting this aggregated data back to Kafka. A typical application might not include this extra Kafka layer and why its done is explained in detail in the paper. Finally, the aggregated data is pushed into a single Mongo DB instance on which analytics are performed. We claim to have built a system that handles voluminous data and is more resilient than a typical data-intensive system, using an extra layer of Kafka instead of a distributed database.

1 Introduction

In the past few years, the number of internet users had grown enormously. As a result, there has been an increase in online shopping, e-commerce, which resulted in the amount of information generated increase exponentially. The biggest challenge then was to find a method to process and store this information but as the big data technologies evolved, the challenge is now to recognize purchase trends and provide relevant product suggestions to users. Because of the competitive markets, there is a need to process this data quickly and efficiently [1].

Consider a scenario where a user wants to purchase a best-rated laptop, he filters products by the laptop ratings but he doesnt know that 100s to 1000s of ratings are being given every minute. While velocity is certainly important in such scenarios, it is more important that the user is able to see the average rating of the product reliably (and correctly) even if a few seconds late. Traditional systems boast to achieve this using a distributed pipeline

and a distributed database. While this is an excellent solution to the e-commerce big-data problem, there is an unnecessary overhead of using a distributed database as there are only 100s to 1000 aggregated records for a million input records. This paper proposes an alternative to this by using a single-point-of-failure database which overcomes the overhead while having the same level of reliability and speed by storing the aggregated records in a reliably distributed buffer (such as Kafka). This paper describes our approach of this using Kafka-Spark-Kafka-Mongo pipeline architecture, its performance, its success (and failures) and the future work.

1.1 Apache Kafka

Kafka[**?**] is often used in real-time streaming data architectures to provide real-time analytics. Since Kafka is a fast, scalable, durable, and fault-tolerant publish-subscribe messaging system, Kafka has higher throughput, reliability, and replication characteristics, and can work with Flume/Flafka, Spark Streaming, Storm, HBase, Flink, and Spark for real-time ingesting, analysis and processing of streaming data. Kafka is a data stream used to feed Hadoop BigData lakes. Kafka brokers support massive message streams for low-latency follow-up analysis in Hadoop or Spark [2].

Kafka has operational simplicity. Kafka is to set up and use, and it is easy to figure out how Kafka works. However, the main reason Kafka is very popular is its excellent performance. It is stable, provides reliable durability, has a flexible publish-subscribe/queue that scales well with N-number of consumer groups, has robust replication, provides producers with tunable consistency guarantees, and it provides preserved ordering at the shard level (i.e. Kafka topic partition). In addition, Kafka works well with systems that have data streams to process and enables those systems to aggregate, transform, and load into other stores. But none of those characteristics would matter if Kafka was slow. The most important reason Kafka is popular is Kafkas exceptional performance [3].

Kafka is used most often for streaming data in real-time into other systems. Kafka is a middle layer to decouple your real-time data pipelines. Kafka core is not good for direct computations such as data aggregations. Kafka can be used to feed fast lane systems (real-time and operational data systems) like Storm, Flink, Spark Streaming, and your services and CEP systems. Kafka is also used to stream data for batch data analysis. Kafka feeds Hadoop. It streams data into your big data platform or into RDBMS, Cassandra, Spark, or even S3 for some future data analysis. These data stores often support data analysis, reporting, data science crunching, compliance auditing, and backups.

1.2 Apache Spark

Spark[4] is a general-purpose distributed data processing engine that is suitable for use in a wide range of circumstances. On top of the Spark core data processing engine, there are libraries for SQL, machine learning, graph computation, and stream processing, which can be used together in an application. Programming languages supported by Spark include: Java, Python, Scala, and R. Application developers and data scientists incorporate Spark into their applications to rapidly query, analyze, and transform data at scale[4].

Spark is capable of handling several petabytes of data at a time, distributed across a cluster of thousands of cooperating physical or virtual servers. It has an extensive set of developer libraries and APIs and supports languages such as Java, Python, R, and Scala; its flexibility makes it well-suited for a range of use cases. Spark is often used with distributed data stores such as MapR-XD, Hadoops HDFS, and Amazons S3, with popular NoSQL databases such as MapR-DB, Apache HBase, Apache Cassandra, and MongoDB, and with distributed messaging stores such as MapR-ES and Apache Kafka.

Spark is designed for speed, operating both in memory and on disk. Spark can perform even better when supporting interactive queries of data stored in memory. In those situations, there are claims that Spark can be 100 times faster than Hadoops MapReduce. Much of Spark's power lies in its ability to combine very different techniques and processes together into a single, coherent whole. Outside Spark, the discrete tasks of selecting data, transforming that data in various ways, and analyzing the transformed results might easily require a series of separate processing frameworks, such as Apache Oozie. Spark, on the other hand, offers the ability to combine these together, crossing boundaries between batch, streaming, and interactive workflows in ways that make the user more productive.

Spark jobs perform multiple operations consecutively, in memory, and only spilling to disk when required by memory limitations. Spark simplifies the management of these disparate processes, offering an integrated whole a data pipeline that is easier to configure, easier to run, and easier to maintain [5].

1.3 MongoDB

As a definition, MongoDB is an open source database that uses a document-oriented data model and a non-structured query language. It is one of the most powerful NoSQL systems and databases around today [6].

Being a NoSQL tool means it does not use the usual rows and columns that we so much associate with the relational database management. It is an architecture that is built on collections and documents. The basic unit of data in this database consists of a set of key-value pairs. It allows documents to have different fields and structures. This database uses a document storage format called BSON which is a binary style of JSON style documents. The data model that MongoDB follows is a highly elastic one that lets you combine and store data of multivariate types without having to compromise on the powerful indexing options, data access, and validation rules. There is no downtime when you want to dynamically modify the schemas. So what it means that you can concentrate more on making your data work harder rather than spending more time on preparing the data for the database.

The Database: In simple words, it can be called the physical container for data. Each of the databases has its own set of files on the file system with multiple databases existing on a single MongoDB server.

The Collection: A group of database documents can be called as a collection. The RDBMS equivalent of the collection is a table. The entire collection exists within a single database. There are no schemas when it comes to collections. Inside the collection, the various documents can have varied fields but mostly the documents within a collection are meant for the same purpose or serving the same end goal.

The Document: A set of key-value pairs can be designated as a document. Documents

are associated with dynamic schemas. The benefit of having dynamic schemas is that document in a single collection does not have to have the same structure or fields. Also, the common fields in a collections document can have varied types of data.

This technology overcame one of the biggest pitfalls of the traditional database systems, that is, scalability. With the ever-evolving needs of businesses their database systems also needed to be upgraded. MongoDB has exceptional scalability. It makes it easy to fetch the data and provides continuous and automatic integration.

1.4 MongoDB Charts

MongoDB Charts is a tool to create visual representations of MongoDB data. Data visualization is a key component in providing a clear understanding of the data, highlighting correlations between variables and making it easy to discern patterns and trends within the dataset. MongoDB Charts makes communicating data a straightforward process by providing built-in tools to easily share and collaborate on visualizations.

One of the most powerful features of MongoDB Charts is its built-in aggregation functionality. Aggregation allows processing the collection data by a variety of metrics and perform calculations such as mean and standard deviation to provide further insight into the data [7].

2 Architecture

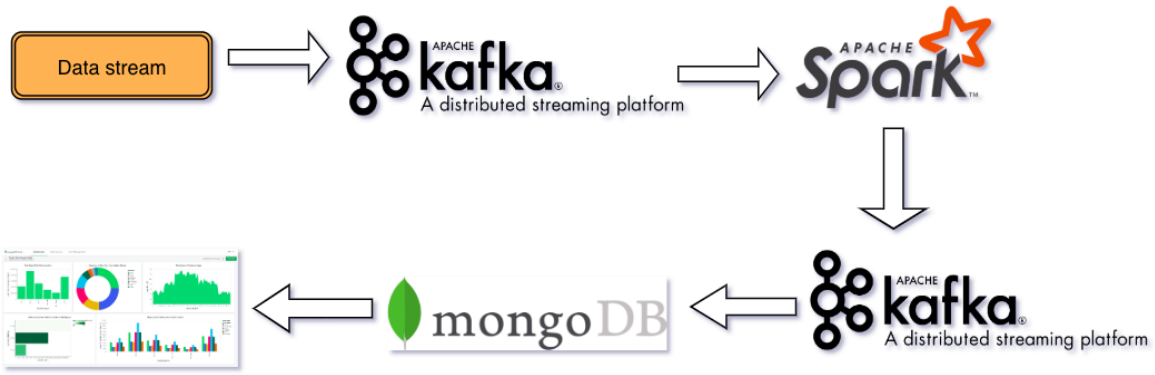

Figure 1: Architecture

2.1 Data

Firstly, lets discuss about the data source that is being used by project. This application can take any realtime ecommerce data containing the details of an order placed by a customer as input. However, for the purpose of this project we didn't choose a realtime streaming

source which provides the actual data. Instead, we have created a synthetic source which generates the data resembling a real time ecommerce data. Each record contains the details like order ID, product name, name of the customer, rating of the product, state of purchase, manufacturer of the product etc., While generating the input, we have carefully introduced bias in the various parameters in the input to help realize the trends in the data. Certain parameters were generated uniformly while certain others follow normal distribution and a few others for a random order with different priorities for each record.

2.2 Buffering using Kafka

The first layer of our pipeline is a Kafka layer which is used to process the streaming data and buffer it for a certain period of time before it is finally consumed by the Spark layer to perform the aggregation. From the documentation of Kafka, it is generally used for two broad classes of applications: Building real-time streaming data pipelines that reliably get data between systems or applications and Building real-time streaming applications that transform or react to the streams of data [2].

In our application, we have used Kafka to build a real-time streaming data pipeline to reliably retrieve the data from streaming source and push it to a Spark layer which performs the aggregations. In our applications, we run Kafka brokers on multiple physical machines located at different physical locations. The Kafka's publish and subscribe methodology helps us in processing different streams of records from different input sources and store it in the buffers for a limited period of time. The Kafka cluster stores the streams of records in categories which are called Kafka topics. Our pipeline also contains a layer of KafkaProducer which fetches the records from the streaming source and publishes it to a Kafka topic. While storing the records in the topics, Kafka stores it in the form of key, value and timestamp. To publish the records from the input stream, we are using Kafka's producer API and our language of choice to implement this KafkaProducer program is Java with no particular preference as there was almost equal support for Java, Python and Scala [2].

2.3 Aggregation using Spark

The data from the Kafka layer is then pulled into Apache Spark using spark's streaming API. Apache Spark is a fast and general purpose cluster computing system. Spark supports a rich set of higher-level tools including Spark SQL for SQL and structured data processing, MLlib for machine learning, GraphX for graph processing, and Spark Streaming [4]. To achieve this, we are using the kafka-spark connector. In the spark layer, we have a master node and two slaves. The primary purpose of these nodes is to fetch the e-commerce data from the corresponding Kafka topic, perform aggregation on the data and spit it out to a different topic. As part of the aggregation, we are calculating the sales of various products in a particular location and from a particular company etc., Here, we try to store the data with respect to a particular location. This data is later used to perform analytics based on a particular location. We have currently fine-grained the location-wise analytics to the level of individual states in the US. Similarly, we were able to achieve the same thing for every company that sells at least one product using the e-commerce site.

2.4 Extra Kafka Layer

We have an extra Kafka layer after the aggregation and this was introduced to remove the burden on output database as this was the single point of failure at this moment. To remove the single point of failure we came up with two solutions: 1) Replicate MongoDB but that would be an overkill as the output database was just storing a few 100s of records for every million input records being processed. 2) Utilize the power of Kafka and publish the aggregated data to Kafka without pushing it to the database directly.

Out of the above two solutions, the second one seemed obvious with two main advantages: The first one, as discussed, was to avoid the loss of aggregated data when the Mongo instance was down. Second one was, it didn't require any infrastructure and extra effort of adding and maintaining the extra Mongo instances making the whole system more robust.

Coming to the implementation, we have introduced a KafkaConsumer layer which sits between the Spark layer and the output MongoDB instance. The Kafka consumer consumes the aggregated output records from the Spark cluster and pushes it to the MongoDB instance. In case the MongoDB goes down for some reason, the KafkaConsumer waits till the instance gets up and running. Once the instance restarts, the KafkaConsumer continues to pull the records from Kafka and push it to the MongoDB instance.

2.5 Aggregated data storage in MongoDB

The aggregated data is pulled from the Kafka brokers and is finally stored in the MongoDB instance which can later be used to visualize and perform various analytics. MongoDB's inbuilt support to display real time analytics has made us choose MongoDB for storing the aggregated data. Mongo Charts is a tool to create visual representation of the data stored in MongoDB. In this step, we have created various views that show the all time sales of a particular product from a particular company or a particular state.

3 Performance and Results

For the purpose of building this system, we used 8 VCL nodes which included 3 instances for Kafka, 3 for Spark, 1 node for generating and streaming input and 1 for the output. We measured the performance of our system using processing speeds at each level.

At the input generation, the speed is dependant on the system (VCL in this case) and any number of input streamers can be started but only one data streaming source was implemented for our system. Kafka pulls data from this data source and similar to the input node, Kafka's speed also depends on how fast the data is being generated from the input system. The most significant thing that affects the performance of our system is Spark because aggregations are slow.

▾ Active Batches (1)

Batch Time	Records	Scheduling Delay (?)	Processing Time (?)	Output Ops: Succeeded/Total	Status
2018/12/13 18:11:20	9000 records	0 ms	-	1/2 (1 running)	processing

▾ Completed Batches (last 207 out of 207)

Batch Time	Records	Scheduling Delay (?)	Processing Time (?)	Total Delay (?)	Output Ops: Succeeded/Total
2018/12/13 18:11:10	9000 records	0 ms	0.6 s	0.6 s	2/2
2018/12/13 18:11:00	6000 records	1 ms	0.4 s	0.4 s	2/2
2018/12/13 18:10:50	9000 records	0 ms	0.5 s	0.5 s	2/2
2018/12/13 18:10:40	6000 records	0 ms	0.4 s	0.4 s	2/2
2018/12/13 18:10:30	9000 records	1 ms	0.8 s	0.8 s	2/2
2018/12/13 18:10:20	9000 records	1 ms	0.5 s	0.5 s	2/2
2018/12/13 18:10:10	6000 records	0 ms	0.4 s	0.4 s	2/2

Figure 2: Spark Processing

Our Spark clusters processes around 18K records/second as seen in Fig.2. We believe this number must be higher if there were multiple sources but the focus of this project is Volume and not Velocity so not much effort was put into generating huge data in parallel. Also, the VCL systems used in our system have a high latency.

Figure 3: Analytics using MongoDB Charts

Finally, we achieved a processing speed of 1 Million records per couple of minutes and Fig.3 shows a snapshot of MongoDB Charts which is used for analytics. This data update in real-time based on the changing aggregated data in the output MongoDB instance. Between

100 to 1000 records are produced for every Million records processed by Spark. This number can be reduced further, to maybe less than a 100 for each product, if a daemon program is run on the output database periodically that aggregates the records further. As the focus is on Volume, our output database stores as many records as possible to be fit in the (VCL) system memory (around 30GB).

4 Conclusion

The system we built is capable of handling voluminous data and is fault-tolerant at both buffering (Kafka) and aggregation (Spark) levels. A drawback of this single instance database is that after a crash, the database may not come back online at all as its a single point of failure. The tradeoff here is that the database is assumed to always comeback online after few minutes versus the overhead in maintaining a distributed database for 1000s of records per 1M input data objects. Also, this number can be further reduced by running a daemon program periodically on the output MongoDB instance, to less than 100 records per product ID. Another drawback is that the data filters at our Spark cluster dont allow improperly structured data. This is not a very big concern for the system because each order data document we generate has the same fields/keys and we assume that this is also the case for e-commerce websites. The third drawback is that our Spark doesnt deal with missing data fields. This can be overcome by implementing Machine Learning at Spark that deals with missing data. Another advantage of using Machine Learning at Spark is that for complex interactive queries than just aggregating on ratings, Spark can perform iterative analysis for better results. One addition we would like to add to the system is a mechanism that automatically scales and load-balances the clusters, that is, a way to kill one or more Kafka/Spark nodes when the data is less and create new Kafka/Spark nodes when data is huge and needed to be processed in parallel. This can be done using a tool such as Kubernetes. Also, the whole system was built on VCL instances which might have impacted the performance and can be improved by running on reliable cloud services such as AWS or GCP. Finally, we would like to think that we have built an end-to-end data-intensive computing system that solves some of the biggest e-commerce challenges.

Acknowledgment

We would like to thank our professor Dr.Vincent Freeh, North Carolina State University, who provided valuable insight and expertise that greatly assisted the research for this project.

References

[1] *Impact of Big data on ecommerce*, 2017. `https://acadpubl.eu/jsi/2017-116-13-22/articles/21/28.pdf`. .

[2] *Kafka: a Distributed Messaging System for Log Processing*, 2015. `http://people.csail.mit.edu/matei/courses/2015/6.S897/readings/kafka.pdf`.

[3] *What is Kafka?* https://dzone.com/articles/what-is-kafka.

[4] *Spark: Cluster Computing with Working Sets.* http://static.usenix.org/legacy/events/hotcloud10/tech/full_papers/Zaharia.pdf.

[5] *What is Spark?* https://mapr.com/blog/spark-101-what-it-what-it-does-and-why-it-matters

[6] *What is MongoDB?* https://intellipaat.com/blog/what-is-mongodb/.

[7] *MongoDB* https://www.mongodb.com/what-is-mongodb.

[8] *MongoDB Charts* https://docs.mongodb.com/charts/master/.

[9] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker and Ion Stoica, *Spark: Cluster Computing with Working Sets.*NSDI12, April 2012, San Jose, CA, USA.

[10] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker and Ion Stoica, *Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing.*