Yes, it is indeed true that financial institutions process, deliver and store huge amount of data whether it is in the form of stocks, digital currencies or electronic money. Data processing of large data becomes even more challenging when time constraints are involved. In financial markets time constraint is very important limitation on how the data should be processed. In case of stocks for example, even a small percentage of change in value during the transaction (which might happen in a fraction of milliseconds) might result in the difference of millions of dollars. So, in such types of scenarios, a technology which promises **real time streaming or processing** and also **real time retrieval** of data from storage is preferable.

As the data gets bigger and bigger, **scalability** become more important. Data in legacy systems can’t persist at a single place for so long. What happens in case of power failure or the storage is full? The definite answer is the data loss! So for such cases, nowadays we need distributed storage across multiple servers to ensure data integrity for financial systems.

We need a technology or framework which ensures zero loss of data. For this requirement to fulfill we need to **replicate** the data across multiple nodes depending on the load and also make sure that the data is **available** when required. So if for example, network has problems at one of the nodes, it can be retrieved by one of the other available nodes.

Last but not least, the framework should be able to retrieve data through web socket APIs, Rest APIs and web scraping. The frameworks should perform distributed processing and storage as depicted in the following diagram:

Input Data

Data node 1

Data node N

Data node 2

Stream

Processing

**Figure1: The suggested framework**

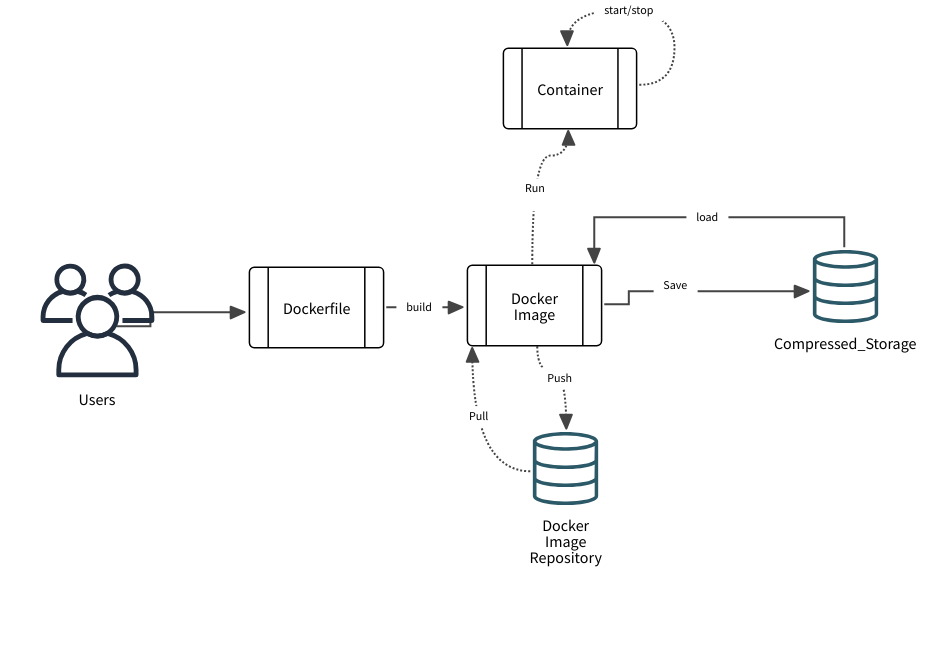
**Framework and Technologies:**

To fulfill above requirements, we need a framework which uses some essential technologies. There are lot of technologies available for stream processing such as Apache Kafka and Apache Flink. For distributed storage we might use Cassandra and Hadoop. But irrespective of which technology to use, I will prefer some essential technologies for such distributed applications which require portability, and scalability to be part of the framework i.e Docker and Kubernetes.

**Docker and Kubernetes**

Docker helps us to package the applications with all the toolsets and dependencies required in the form of containers. This feature helps us to deploy the applications on the distributed systems. Docker provides abstraction such that it can be deployed independent of operating system. In the above diagram, each of the node in the data processing stage can be containerized with the help of Docker. As the number of Docker containers increase, Kubernetes becomes more and more important because it helps us in the networking and scheduling of containers, and in the load balancing across multiple containers.

Following UML diagram explains how Docker containerization works. The diagram shows how typically a Docker image can be created by using a Docker file. We can run and stop the Docker containers. Dockerfile normally contains the dependencies and the toolsets required within the container. In that way we can distribute the stream processor(used to process financial data) into components. Each component can run on the container standalone with own toolsets and dependencies managed. I will go into details regarding this topic when I will document the practical task.



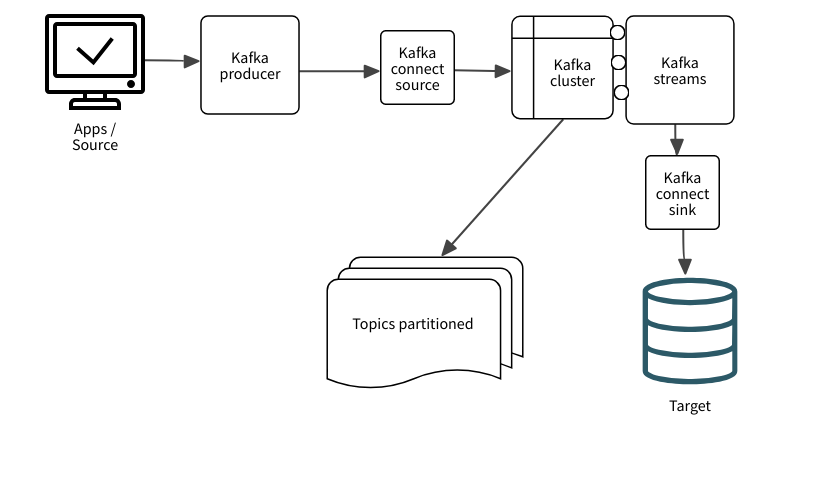
**Figure2: Docker framework**

**Apache Kafka stream processor**

Apache Kafka is one of the most popular event based stream processing framework which supports distributed processing and hence fulfills our requirement of scalability. This framework is employed by most of the companies which process huge customer data and it promises zero loss of data and also fulfills our real time constraint.

Let me introduce briefly how Kafka works. It has a component called ‘producer’ which publishes the messages with some topic. The ‘consumers’ are also called ‘sinks’ because they receive the messages. Kafka servers are also called brokers. The topics are partitioned in such a way that the load is balanced equally among the brokers. For example if a topic has 100 partitions and there are 10 brokers available in the Kafka cluster, then each broker gets 10 partitions of that topic.

The basic flow chart looks like this:

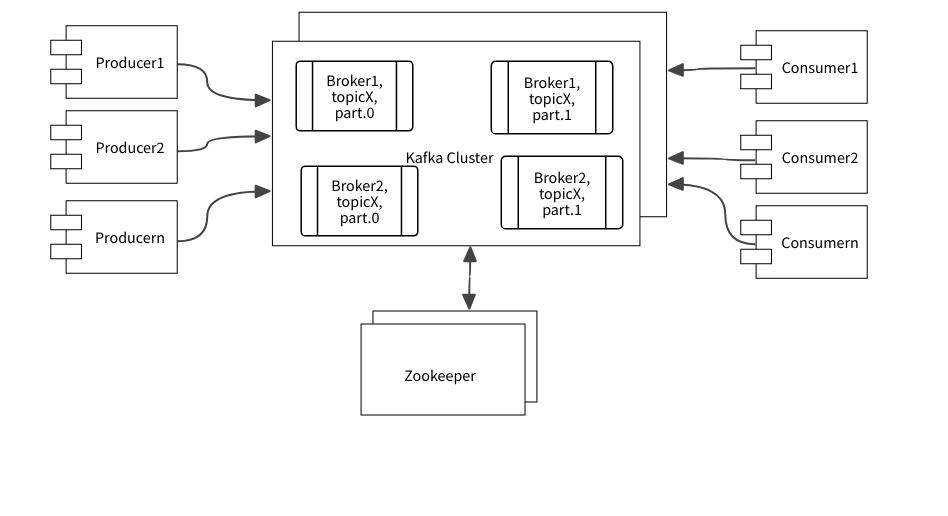


**Figure3: Kafka Framework**

Kafka Connect is a framework within Kafka which allows to reliably connect Kafka with external components, whether they be producers or targets/databases. Kafka stream is a streaming client library which interacts with Kafka topics and brokers to produce real time applications. Kafka streams has a processing pipeline while every step of pipeline manages parallelism. Kafka stream can also consume the messages and it can also act as producer to publish the results on any topic.

If the client want to connect to the Kafka cluster and it doesn’t exactly know which brokers are there in the cluster, it can connect to any of the broker in the cluster, because every broker has the metadata about the whole cluster. Through this metadata, it can inform the client that which other brokers are there on the cluster. That is the exact same reason that the Kafka brokers are also bootstrap servers. The servers persist the data for a minimum number of allowable days. The following diagram shows how ‘zookeeper’ interacts with kafka cluster. ‘zookeeper’ is mainly used for configuration management and distributed synchronization.

Apache Kafka System Architecture:



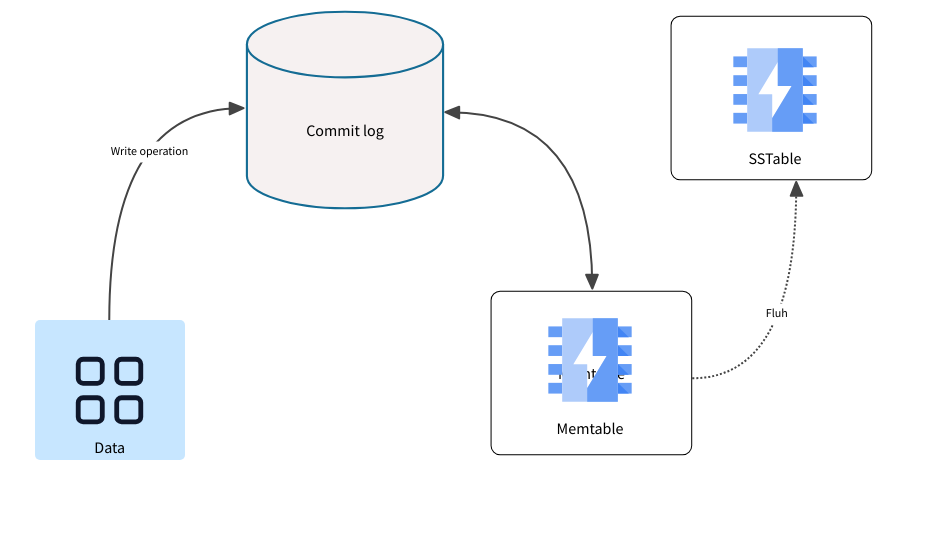
**Figure4: Kafka System Architecture**

**Apache Cassandra**

I will suggest Cassandra to fulfill our second requirement of distributed or scalable storage. Although there are also other options such as Google Cloud Storage and Amazon S3, I will focus on Cassandra because it is fully open source and it has no single point of failure. Cassandra can store the data across multiple servers promising data integrity. Its architecture allows to make the data available even in case of failure at one of the nodes. Thanks to the configurable replication scheme which makes it fault tolerant. Cassandra can output the data with high throughput and consistency.

Not only the data integrity is promised but also due to its flexible architecture it can be searched easily from within the tables using the query language Cassandra Query Language(CQL). According to the Brewer’s theorem, in distributed system we can’t guarantee all of the three properties namely consistency, partition tolerance and availability. We can only guarantee maximum of two properties. Cassandra allows to configure the trade-off between consistency and availability. We can ask Cassandra for example to give priority to consistency of the data even if the data reads might take longer, sacrificing availability.

Cassandra uses several techniques to ensure the data availability. One feature of Cassandra in that regard is that it writes to the logs first and maintains data structures called memtables before actually writing to the disk. Periodically however the data from memtables is deleted and written back to the disk in permanent data structure called SStables. This process is show below:



**Figure5: Kafka Write cycle**

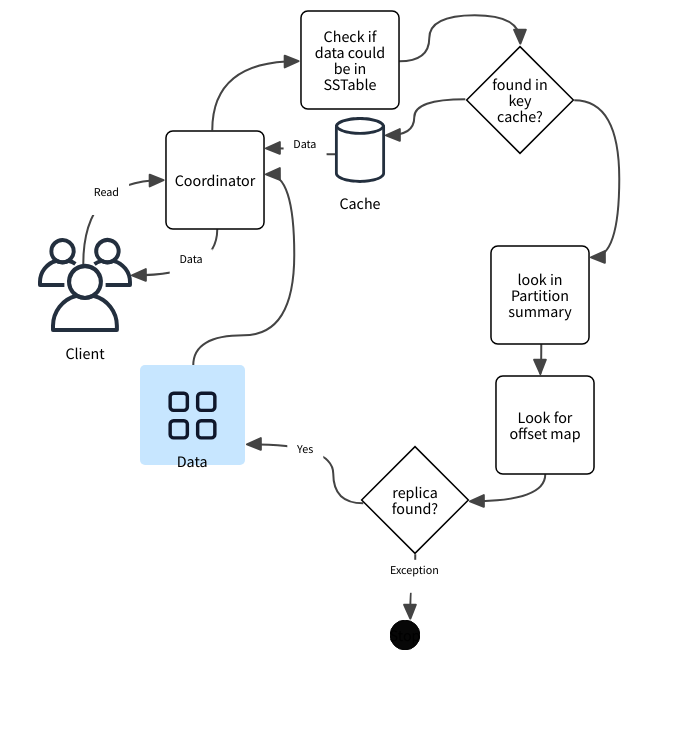
Another technique which Cassandra uses is called hinted trade-off. This means that if a node is not available at the time of write request, then the data is temporarily written to the next available node instead of waiting for the availability of the dead node in the cluster. However, once the dead node awakes, the data is written back on it. This helps also in the availability of data. If we prefer the availability over consistency, then the data loss in Cassandra is quite rare.

In Cassandra we have top-level namespace called Keyspace. Keyspace not only maintains the data in tables but also has metadata containing settings related to replication and compaction. While creating a database one must first create a Keyspace with certain replication settings. It is recommended to use the replication factor of 3 at least if stronger consistency is required. That means if a node becomes dead, we have still two nodes to confirm the consistency. The rule of thumb is that the replication factor must not exceed the number of nodes.

Each node can act as a coordinator. If a node receives a read request initially, it becomes the coordinator. Its duty is to ask the other replica nodes using fast hashing algorithms for data. It plays critical role in the data availability in Cassandra.

The read operation in Cassandra is more complicated than the write operation. The Cassandra coordinator responds to the read request by searching the replicas using the hashes. If the replicas are found in the search which satisfy the read request, the coordinator proceeds to the next steps otherwise it throws an exception. However, if the replicas are found the coordinator sends the digest request to the replicas to know if all of them have consistent requested data. If the data is not consistent, the coordinator sends the read request for all the replicas and the data which is received last is returned to the client. If the data is already consistent, the coordinator then sends request only to the fastest responding replica. The question is how it is this determined which replica is responding with how much speed? Well the speed with which the nodes respond is calculated dynamically by Cassandra.

The coordinator first looks the data in key cache which maintains the keys or indexes of the partitions which are usually queried. If the key is found there, the data is sent back otherwise the coordinator tries to find it in the partition summaries or in the memory offset maps to locate the data. The following diagram shows how it might look like for a specific consistency level and use case.



**Figure6: Cassandra read cycle**

Cassandra is different from SQL databases. To write data into tables in Cassandra, one must specify the primary key. A primary key has two parts: Set of clustering columns which are optional and an essential partition key. The first element in the primary key is the partition key and the rest are the clustering columns. Why do we need both the partition key and clustering columns? It is to make the search faster. With only primary key, it is not possible to search the unique entry in the partition in a robust manner. The partition key creates a new partition for every different primary key. This helps us in scalability in the sense that the data is distributed on different partitions identified by the partition key. If one wants a specific entry within the partition, then clustering columns can be used. Selecting correct partition key is important. It must be chosen based on the retrieval policy. Let’s understand that with the help of an example:

**Node1, PartitionA**

Bid

Volume

Price

Bitcoin

**Node2,partitionB**

Bid

Price

Volume

In the above diagram, we have three partitions created because the partition key is the first one which is the name or symbol of the crypto currency. All the three partitions can be allocated on three different nodes on different servers. This fulfills our requirement of distribution and scalability. Moreover, it is far easier to search in terms of symbol or currency. In the clustering column we can add any of the other parameters namely price, volume or bid. What happens if we use the partition key as volume for example? In that case, not only the scalability will be effected but also the searching within the table will be slower.

**Figure7: Cassandra example use case**

**Node3, ParitionC**

Volume

Volume

Bid

Price

Ethereum

Bid

Bid

Price

Price

Volume

Bid

Price

Volume

Dogecoin

**Figure8: Cassandra example use case**

Node1, PartitionA

Volume2

Volume1

Bitcoin

Price

Bid

Dogecoin

Price

Bid

As shown in the above diagram scalability is badly affected if we select volume as a partition key. The search within the table will also be lot slower. Hence for scalability and availability of data I will prefer Cassandra over many other databases.

Node2, PartitionC

Node3, PartitionD

Node3, PartitionE

Node3, PartitionF

Node2, PartitionB

Volume4

Volume3

Dogecoin

Price

Bid

Ethereum

Price

Bid

Volume5

Ethereum

Price

Bid

Volume6

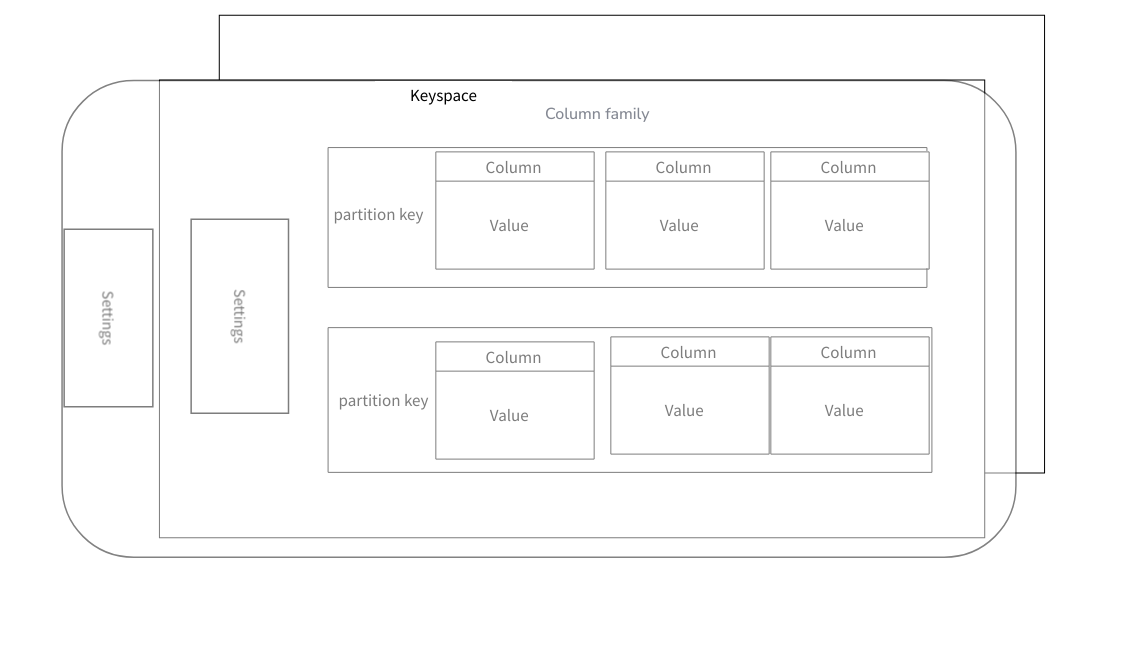
Ethereum

Price

Bid

**Architecture diagram**

As shown below, each Casandra cluster consists of keysspaces which consists of column families. The ‘settings’ help us for example to configure the replication factors.



**Figure7: Architecture Design Cassandra**

**The Practical implementation**

In my project I have following files:

* *server3.py* – This is the python script which receives the streams of data over websockets. The script connects with the Kafka server and publishes on the topics depending on the symbol of the currency. Since there are three symbols, three topics are published. It also stores the data in Cassendra using Keyspace: ‘binance’ and creates three tables each for every symbol of currency
* runscripts.py – This contains the script to consider the test output from unit test and run all the containers once the result is successful. Otherwise, error message is printed on the console
* test\_server.py – This contains a simple unit test for my application
* Dockerfile - Using this file Docker image is built. I have containerized my application. Here the necessary dependencies are also provided so that the container can work in a standalone way. My application server3.py is copied to the docker image here.
* Docker-compose.yml – This has been used to run all the three containers, namely: Cassandra, Kafka and the one which runs my application. It also defines the internal network through which all the docker containers are connected. In this file local volume is also mounted to the container to help the development process

To run the application, please install python3 and run the script runscripts.py. The script will pull the images for Cassandra and Kafka and use the local Dockerfile for my application. At the end all the containers will run and communicate with each other. ‘cqlsh’ can be used to see the values stored in the respective tables. All the three tables which are generated are called: *btcusdt\_node ethusdt\_node bnbusdt\_node.* **The table outputs have also been provided in the files.**

The ‘Kafka’ topics can also be consumed by listening on the respective topics. For that use the command ‘docker exec –it kafka /bin/bash’ and navigate to the directory /opt/kafka/bin and execute the command:

./kafka-console-consumer.sh --bootstrap-server 'kafka':9092 --topic BTCUSDT\_agg\_trades --from-beginning. **The output in kafka for one topic has also been provided with the code.**

**Remarks:** Before starting the application, please ensure that Cassandra has been initialized correctly. I have observed that once Cassandra has already stored huge tables and maintained gigabytes of storage including logs and cache it take a long time for initialization. During this time, if the application starts, the connection might not be established just in the beginning. The application tries to re-connect with the interval of 5s in case of failure.

The detailed illustrations regarding my implementation can be seen below:

