**Apache Kafka**

Described as a “distributed commit log” or more recently as a “distrib‐ uting streaming platform.” A filesystem or database commit log is designed to provide a durable record of all transactions so that they can be replayed to consis‐ tently build the state of a system. Similarly, data within Kafka is stored durably, in order, and can be read deterministically

Keys are used when messages are to be written to partitions in a more controlled manner. The simplest such scheme is to generate a consistent hash of the key, and then select the partition number for that message by taking the result of the hash modulo, the total number of partitions in the topic. This assures that messages with the same key are always written to the same partition

messages are written into Kafka in batches.

A batch is just a collection of messages, all of which are being produced to the same topic and partition. An indi‐ vidual roundtrip across the network for each message would result in excessive over‐ head, and collecting messages together into a batch reduces this. Of course, this is a tradeoff between **latency and throughput**: the larger the batches, the more messages that can be handled per unit of time, but the longer it takes an individual message to propagate. Batches are also typically compressed, providing more efficient data trans‐ fer and storage at the cost of some processing power.

Schemas; it is recommended that addi‐ tional structure, or schema, be imposed on the message content so that it can be easily understood

Many Kafka developers favor the use of Apache Avro, which is a serialization framework originally developed for Hadoop. Avro pro‐ vides a compact serialization format; schemas that are separate from the message pay‐ loads and that do not require code to be generated when they change

Messages in Kafka are categorized into topics

A partition is a sin‐ gle log. Messages are written to it in an append-only fashion, and are read in order from beginning to end.

Partitions are also the way that Kafka provides redundancy and scalability. Each partition can be hosted on a different server, which means that a single topic can be scaled horizontally across multiple servers to provide performance far beyond the ability of a single server

Most often, a **stream** is considered to be a single topic of data, regardless of the number of partitions. This represents a single stream of data moving from the producers to the consumers

Major Kafka Clients:

Produsers and Consumers; and advanced client APIs—Kafka Connect API for data inte‐ gration and Kafka Streams for stream processing. T

In some cases, the pro‐ ducer will direct messages to specific partitions. This is typically done using the mes‐ sage key and a partitioner that will generate a hash of the key and map it to a specific partition. This assures that all messages produced with a given key will get written to the same partition. **The producer could also use a custom partitioner that follows other business rules for mapping messages to partitions.**

A key feature of Apache Kafka is that of **retention**, which is the durable storage of messages for some period of time. Kafka brokers are configured with a default reten‐ tion setting for topics, either retaining messages for some period of time (e.g., 7 days) or until the topic reaches a certain size in bytes (e.g., 1 GB).

The offset is another bit of metadata—an integer value that continually increases—that Kafka adds to each message as it is produced. **Each message in a given partition has a unique offset.** By storing the offset of the last consumed message for each partition, either in Zookeeper or in Kafka itself, a consumer can stop and restart without losing its place

Consumers work as part of a consumer group, which is one or more consumers that work together to consume a topic. The group assures that each partition is only con‐ sumed by one member.

The mapping of a consumer to a partition is often called **ownership** of the partition by the consumer

A single Kafka server is called a **broker**. The broker receives messages from producers, assigns offsets to them, and commits the messages to storage on disk. It also services consumers, responding to fetch requests for partitions and responding with the mes‐ sages that have been committed to disk.

Individual topics can also be config‐ ured with their own retention settings so that messages are stored for only as long as they are useful

Topics can also be configured as log compacted, which means that Kafka will retain only the last mes‐ sage produced with a specific key. This can be useful for changelog-type data, where only the last update is interesting.

Kafka is able to seamlessly handle **multiple producers**, whether those clients are using many topics or the same topic. This makes the system ideal for aggregating data from many frontend systems and making it consistent

**Multiple Kafka consumers** can choose to operate as part of a group and share a stream, assuring that the entire group processes a given message only once.

Disk Based retention: Messages are committed to disk, and will be stored with configurable retention rules

**uUsage:**

1. Activity Tracking
2. Messaging
3. Logging
4. Commit log

Apache Zookeeper, which is used by Kafka for storing metadata for the brokers

A Zookeeper cluster is called an ensemble. Due to the algorithm used, it is recom‐ mended that ensembles contain an odd number of servers (e.g., 3, 5, etc.) as a major‐ ity of ensemble members (a quorum) must be working in order for Zookeeper to respond to requests.

Many users will have the partition count for a topic be equal to, or a multiple of, the number of brokers in the cluster

**Limiting the size of the partition on the disk to less than 6 GB per day of retention often gives satisfactory results**

The performance of producer clients will be most directly influenced by the through‐ put of the broker disk that is used for storing log segments

The normal mode of operation for a Kafka consumer is reading from the end of the partitions, where the consumer is caught up and lagging behind the producers very little, if at all. In this situation, the messages the consumer is reading are optimally stored in the system’s page cache, resulting in faster reads than if the broker has to reread the messages from disk. **Therefore, having more memory available to the sys‐ tem for page cache will improve the performance of consumer clients**

Kafka utilizes Zookeeper for storing metadata information about the brokers, topics, and partitions

Whether you use Kafka as a queue, message bus, or data storage platform, you will always use Kafka by writing a producer that writes data to Kafka, a consumer that reads data from Kafka, or an application that serves both roles

Latency should be low but latencies up to **500ms** can be tolerated, and throughput should be very high—we expect to process up to a million messages a second

Once we send the **ProducerRecord**, the first thing the producer will do is serialize the key and value objects to ByteArrays so they can be sent over the network.

When the broker receives the messages, it sends back a response. If the messages were successfully written to Kafka, it will return a RecordMetadata object with the opic, partition, and the offset of the record within the partition.

3 ways to send message:

* Fire-and-forget ,
* Synchronous send We send a message, the send() method returns a Future object, and we use get() to wait on the future and see if the send() was successful or not.
* Asynchronous send We call the send() method with a callback function

Serializers:

Producer configuration includes mandatory serializers. We’ve seen how to use the default String serializer. Kafka also includes serializers for integers and ByteArrays,

**Avro serializer**

Avro data is described in a language-independent schema. The schema is usually described in JSON and the serialization is usually to binary files

One of the most interesting features of Avro, and what makes it a good fit for use in a messaging system like Kafka, is that when the application that is writing messages switches to a new schema, the applications reading the data can continue processing messages without requiring any change or update

The mapping of keys to partitions is consistent only as long as the number of parti‐ tions in a topic does not change

Kafka Consumer:

we need to allow multiple consumers to read from the same topic, splitting the data between them

The main way we scale data consumption from a Kafka topic is by adding more con‐ sumers to a consumer group. It is common for Kafka consumers to do high-latency operations such as write to a database or a time-consuming computation on the data. In these cases, a single consumer can’t possibly keep up with the rate data flows into a topic, and adding more consumers that share the load by having each consumer own just a subset of the partitions and messages is our main method of scaling. **This is a good reason to create topics with a large number of partitions—it allows adding more consumers when the load increases**

Partition ReBalance:

At the heart of the consumer API is a simple loop for polling the server for more data. Once the consumer subscribes to topics, the poll loop handles all details of coordina‐ tion, partition rebalances, heartbeats, and data fetching, leaving the developer with a clean API that simply returns available data from the assigned partitions

**one of Kafka’s unique characteristics is that it does not track acknowledgments from consumers the way many JMS queues do. Instead, it allows consumers to use Kafka to track their posi‐ tion (offset) in each partition**

**Asynschronous commit:???**

One drawback of manual commit is that the application is blocked until the broker responds to the commit request. This will limit the throughput of the application. Throughput can be improved by committing less frequently, but then we are increas‐ ing the **number of potential duplicates that a rebalance will create**

**Kafka Serialization Vs Deserialiation example???**

Replica:

1. Leader 2. Follower Replica

Reliability Guarantees:

Kafka provides order guarantee of messages in a partition. If message B was writ‐ ten after message A, using the same producer in the same partition, then Kafka guarantees that the offset of message B will be higher than message A, and that consumers will read message B after message A.

**Building Data PipeLines:**

**TO DO: KAFKA CONNECT. Develop an example using Kafka Connect**

* **Timeliness:**

Kafka, being a streaming data platform with scalable and reliable storage, can be used to support anything from near-real-time pipelines to hourly batches. Producers can write to Kafka as frequently and infrequently as needed and consumers can also read and deliver the latest events as they arrive. Or consumers can work in batches: run every hour, connect to Kafka, and read the events that accumulated during the previous hour

**Reliability:**

**How does Kafka gaurantess exactly once delivery**

We no longer need to implement a complex **back-pressure mechanism** because if producer throughput exceeds that of the consumer, data will accumulate in Kafka until the consumer can catch up. Kafka’s ability to scale by adding consumers or producers independently allows us to scale either side of the pipeline dynamically and independently to match the changing requirements.

There are two ways to build data pipeline: ETL and ELT.

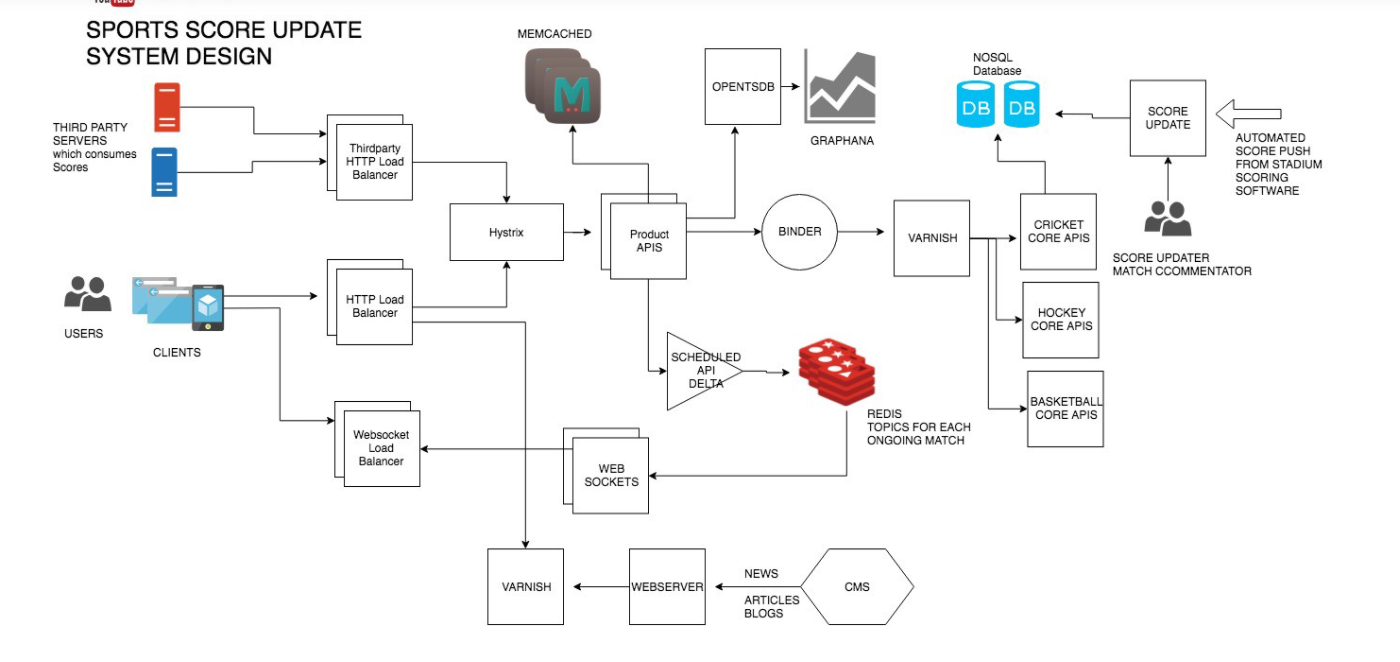
Security: Kafka allows encrypting data on the wire, as it is piped from sources to Kafka and from Kafka to sinks. It also supports authentication (via SASL) and authorization

TO DO:

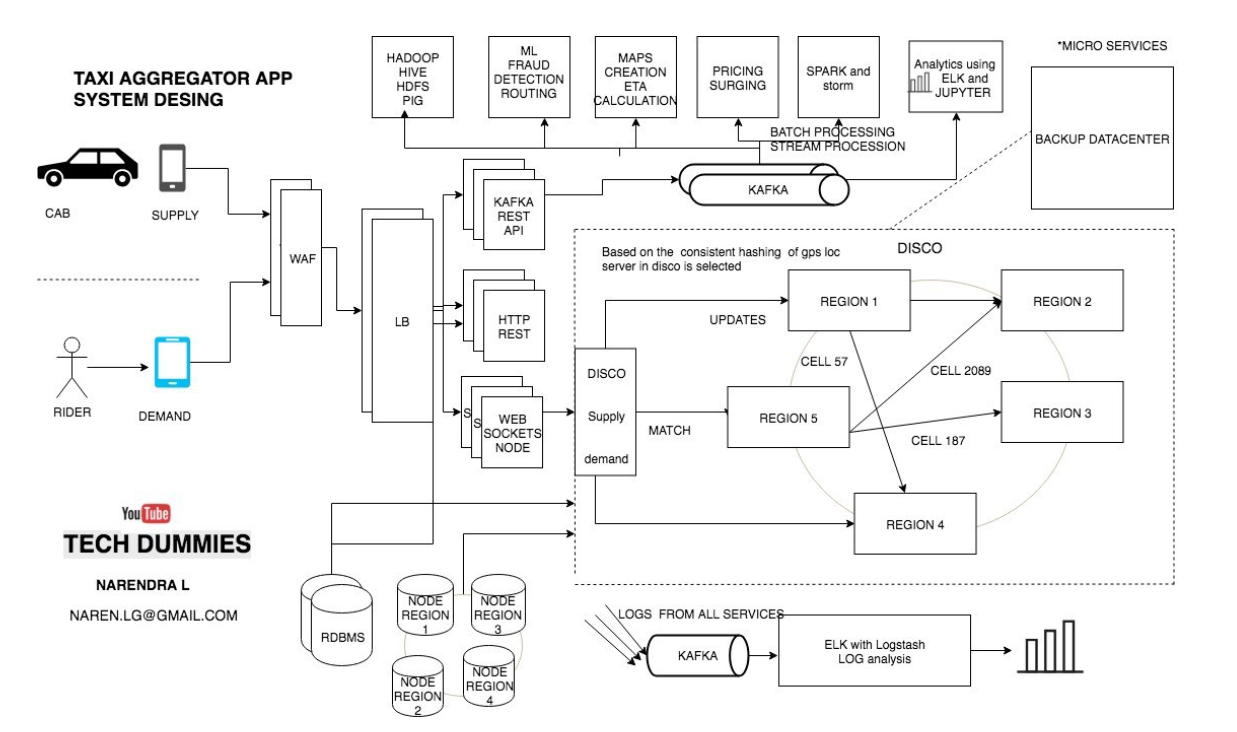
<https://github.com/IBMProjectEventStore/db2eventstore-kafka>

**IBM Db2 to MongoDB; as example of Kafka connect**

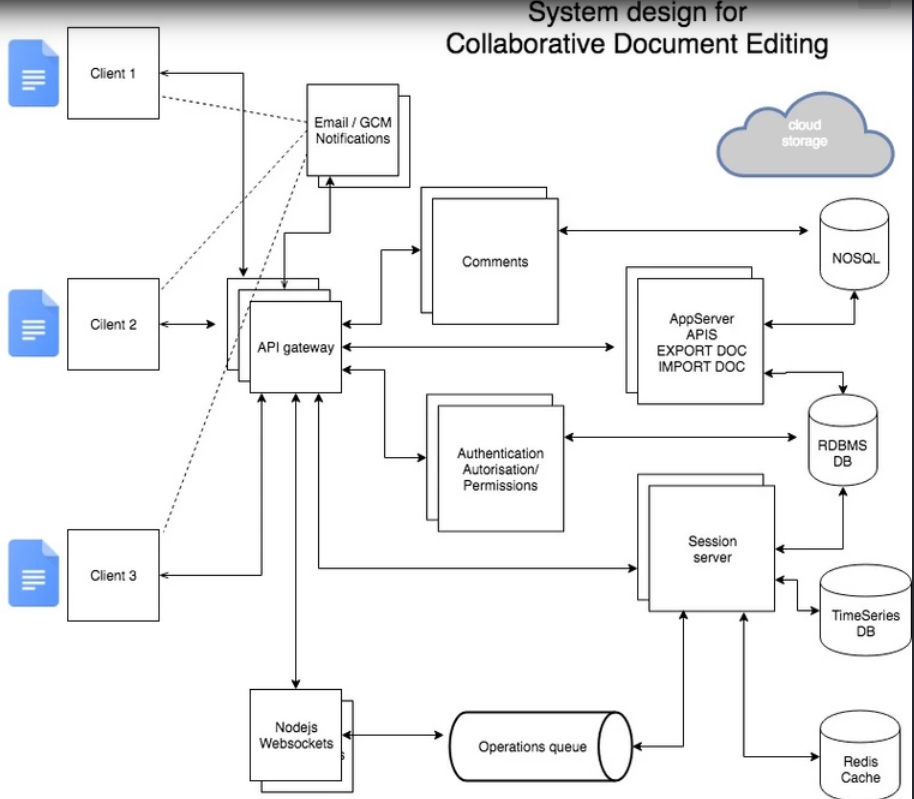
**CricBuzz:**

**CricBuzz:**

**Uber:**



**Google Docs:**



**>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>**