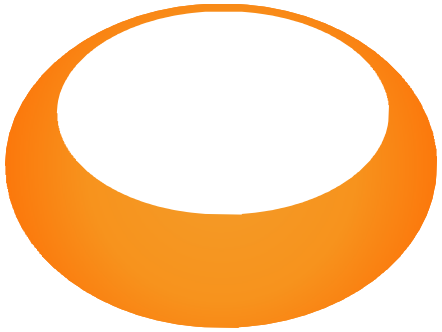


KAFKA AND STORM

Disclaimer

The instructions in this deck have been carefully checked for accuracy and are presumed to be reliable. The writers assume and reserve the right to modify and revise this document without notice. No part of this document in any form or by any means be reproduced, stored in a retrieval system or transmitted without prior written permission of the copyright owner. Information contained in this deck may have information obtained from open source and should be used as a reference guide. It is our goal to supply accurate and reliable documentation. If you discover a discrepancy in this document, please e-mail your comments to venkatraji71@gmail.com

Apache Storm

Apache Storm is a free and open source distributed real time computation system.

Storm makes it easy to reliably process unbounded streams of data, doing for real time processing what hadoop did for batch processing

1. Storm was originally created by **Nathan Marz** and team at **BackType**.
2. Later, Storm was acquired and open-sourced by **Twitter**.
3. In a short time, Apache Storm became a standard for distributed real-time processing system that allows you to process large amount of data, similar to Hadoop.
4. Apache Storm is written in Java and Clojure.
5. It is continuing to be a leader in real-time analytics

1. Apache Storm is a distributed real-time big data-processing system. Storm is designed to process vast amount of data in a fault-tolerant and horizontal scalable method.
2. Though Storm is stateless, it manages distributed environment and cluster state via Apache Zoo Keeper.
3. Storm is easy to setup, operate and it guarantees that every message will be processed through the topology at least once.

Storm and Hadoop Comparison

Apache Storm	Apache Hadoop
Real-time stream processing	Batch processing
Stateless	Stateful
Master/Slave architecture with Zoo Keeper based coordination. The master node is called as **nimbus** and slaves are **supervisors**	Master-slave architecture with/without Zoo Keeper based coordination. Master node is **Resource Manager** and slave node is **Node Manager**.
A Storm streaming process can access tens of thousands messages per second on cluster.	Hadoop Distributed File System (HDFS) uses Map Reduce framework to process vast amount of data that takes minutes or hours.
Storm topology runs until shutdown by the user or an unexpected unrecoverable failure.	Map Reduce jobs are executed in a sequential order and completed eventually
If nimbus / supervisor dies, restarting makes it continue from where it stopped, hence nothing gets affected.	If the Resource Manager dies, all the running jobs are lost.

Use Cases of Apache Storm

Apache Storm is very famous for real-time big data stream processing. For this reason, most of the companies are using Storm as an integral part of their system.

1. **Twitter** − Twitter is using Apache Storm for its range of "Publisher Analytics products". "Publisher Analytics Products" process each and every tweets and clicks in the Twitter Platform. Apache Storm is deeply integrated with Twitter infrastructure.

2. **NaviSite** − NaviSite is using Storm for Event log monitoring/auditing system. Every logs generated in the system will go through the Storm. Storm will check the message against the configured set of regular expression and if there is a match, then that particular message will be saved to the database.

3. **Wego** − Wego is a travel metasearch engine located in Singapore. Travel related data comes from many sources all over the world with different timing. Storm helps Wego to search real-time data, resolves concurrency issues and find the best match for the end-user.

https://storm.apache.org/Powered-By.html

Benefits of Apache Storm

Storm is open source, robust, and user friendly. It could be utilized in small companies as well as large corporations.

Storm is fault tolerant, flexible, reliable, and supports any programming language.

Allows real-time stream processing.

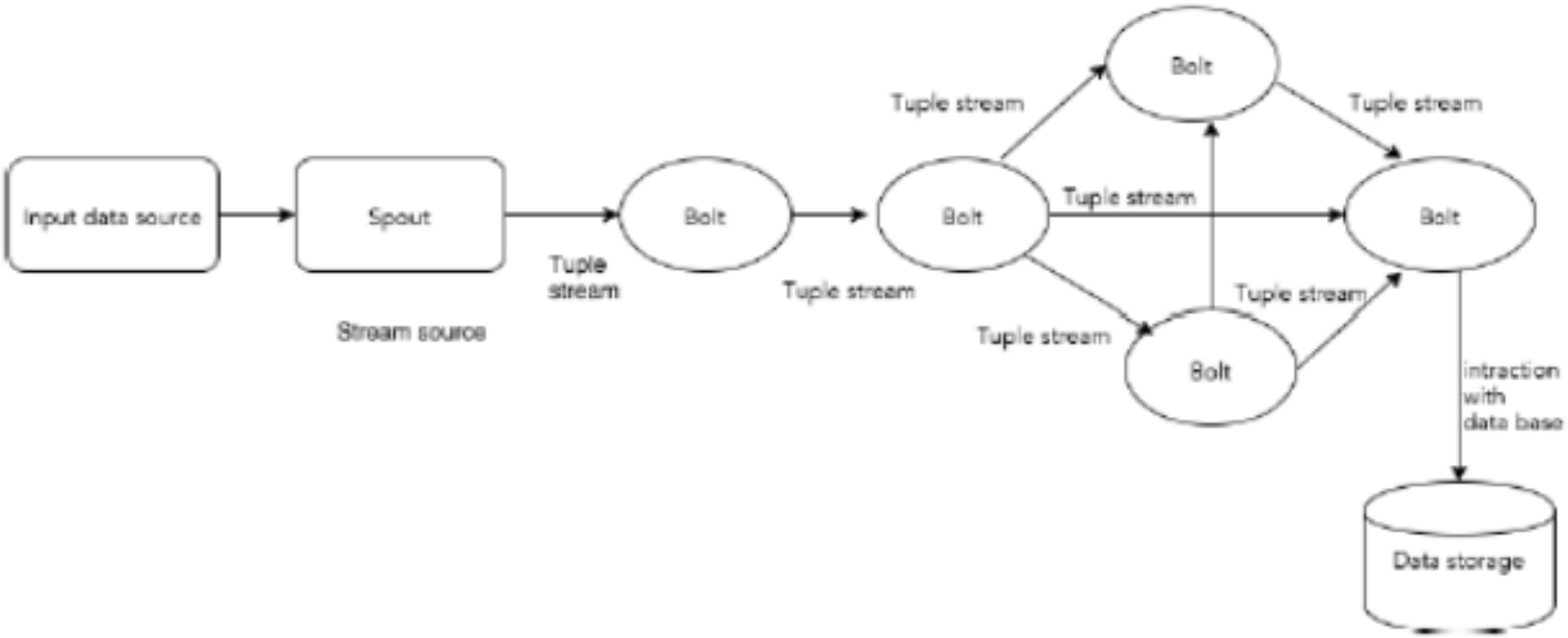
Storm is unbelievably fast because it has enormous power of processing the data.

Storm can keep up the performance even under increasing load by adding resources linearly. It is highly scalable.

Storm performs data refresh and end-to-end delivery response in seconds or minutes depends upon the problem. It has very low latency.

Storm provided guaranteed data processing even if any of the connected nodes in the cluster dies

Storm Flow


Input data source
Spout
Stream source
Tuple stream
Bolt
Tuple stream
Bolt
Tuple stream
Bolt
Tuple stream
Tuple stream
Bolt
Tuple stream
Tuple stream
Bolt
intraction with data base
Data storage

Apache Storm Flow

1. Apache Storm has two type of nodes, **Nimbus** (master node) and **Supervisor** (worker node). Nimbus is the central component of Apache Storm.
2. The main job of Nimbus is to run the Storm topology. Nimbus analyzes the topology and gathers the task to be executed. Then, it will distributes the task to an available supervisor.
3. A supervisor will have one or more worker process. Supervisor will delegate the tasks to worker processes.
4. Worker process will spawn as many executors as needed and run the task. Apache Storm uses an internal distributed messaging system for the communication between nimbus and supervisors.

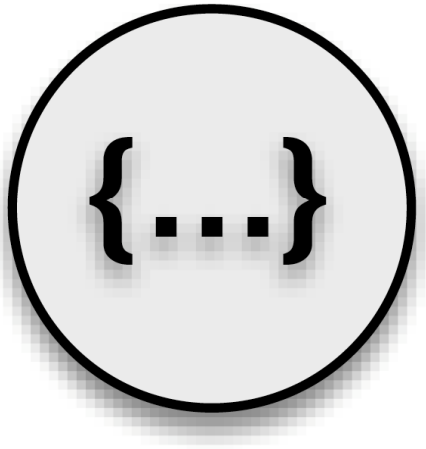
Apache Storm Flow

1. Nimbus is a master node of Storm cluster. All other nodes in the cluster are called as **worker nodes**. Master node is responsible for distributing data among all the worker nodes, assign tasks to worker nodes and monitoring failures.
2. The nodes that follow instructions given by the nimbus are called as Supervisors. A **supervisor** has multiple worker processes and it governs worker processes to complete the tasks assigned by the nimbus.
3. A worker process will execute tasks related to a specific topology. A worker process will not run a task by itself, instead it creates **executors** and asks them to perform a particular task. A worker process will have multiple executors.
4. An executor is nothing but a single thread spawn by a worker process. An executor runs one or more tasks but only for a specific spout or bolt.
5. A task performs actual data processing. So, it is either a spout or a bolt.

Storm Components

- Tuple
- Stream
- Spout
- Bolt
- Topology

Tuple

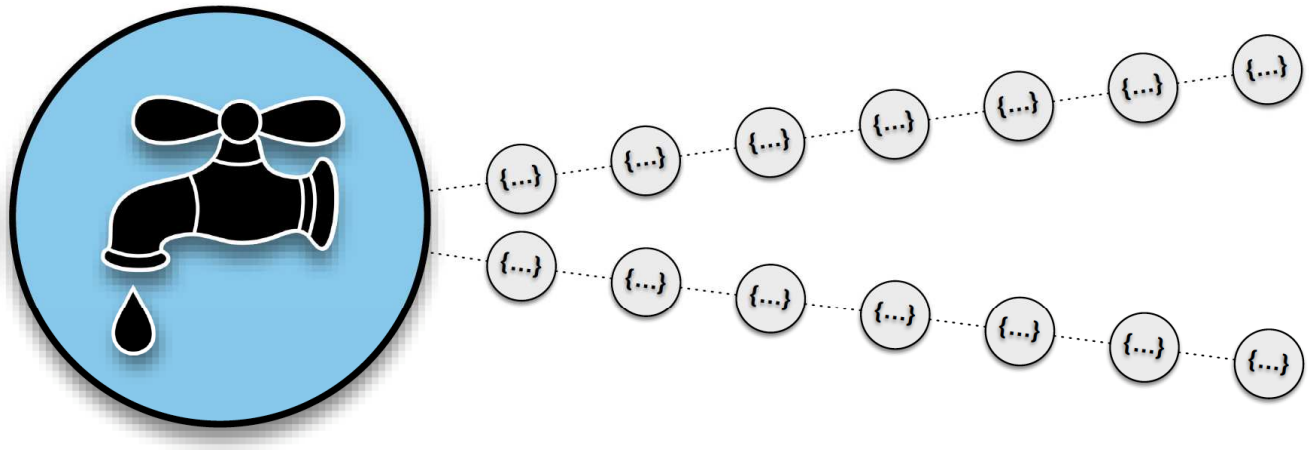


- Core Unit of Data
- Immutable Set of Key/Value Pairs

Streams

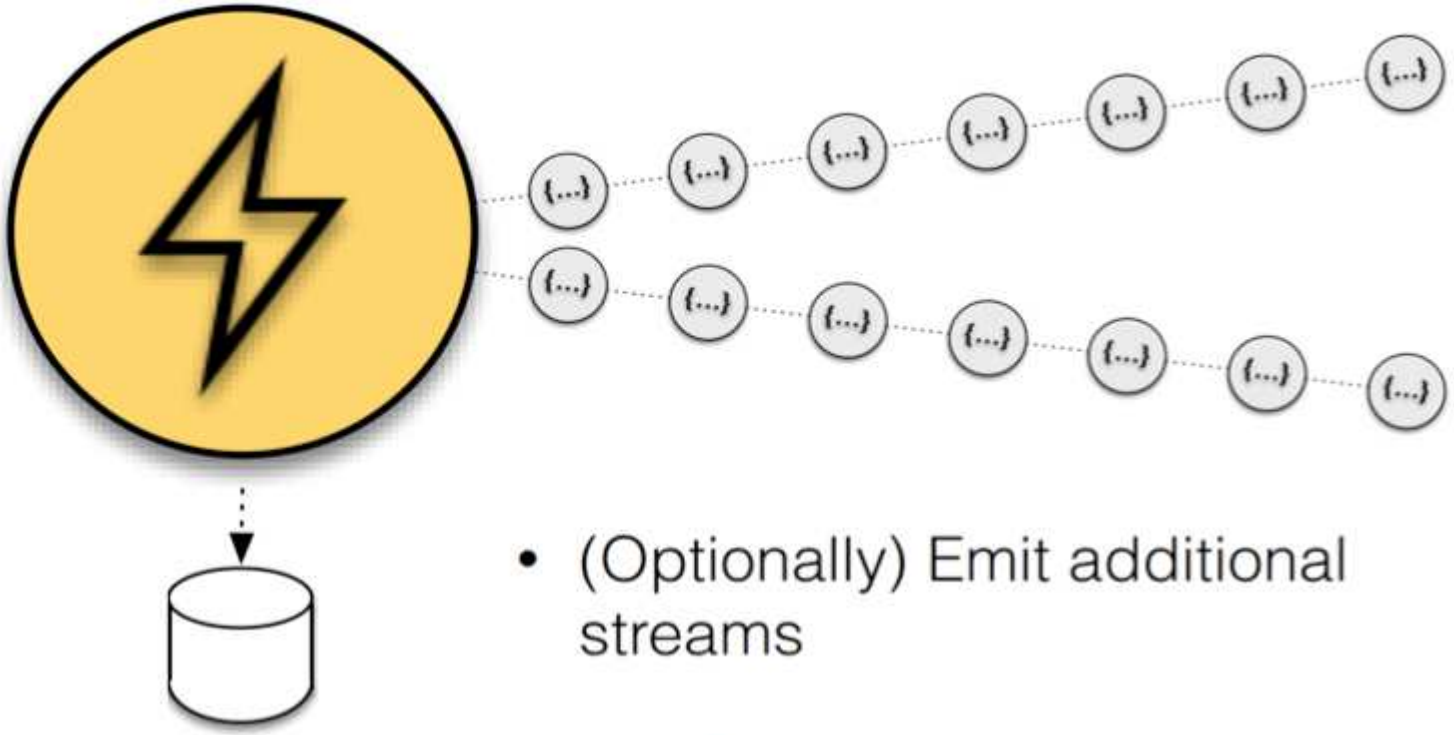
Unbounded Sequence of Tuples

Spouts



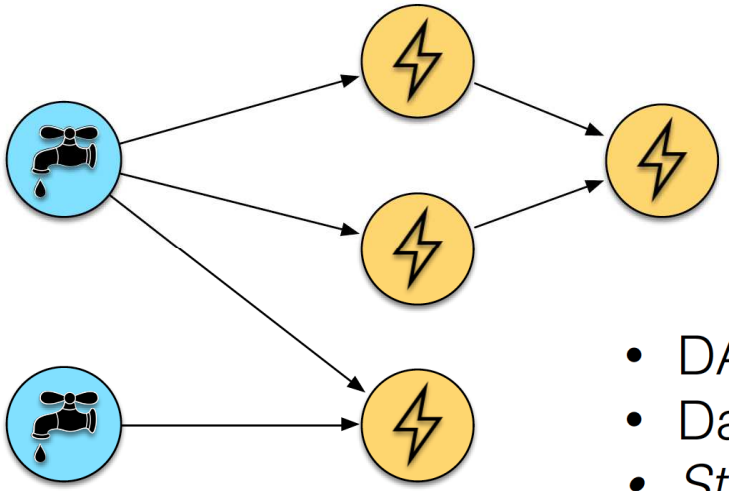
- Source of Streams
- Wraps a streaming data source and emits Tuples

Bolts



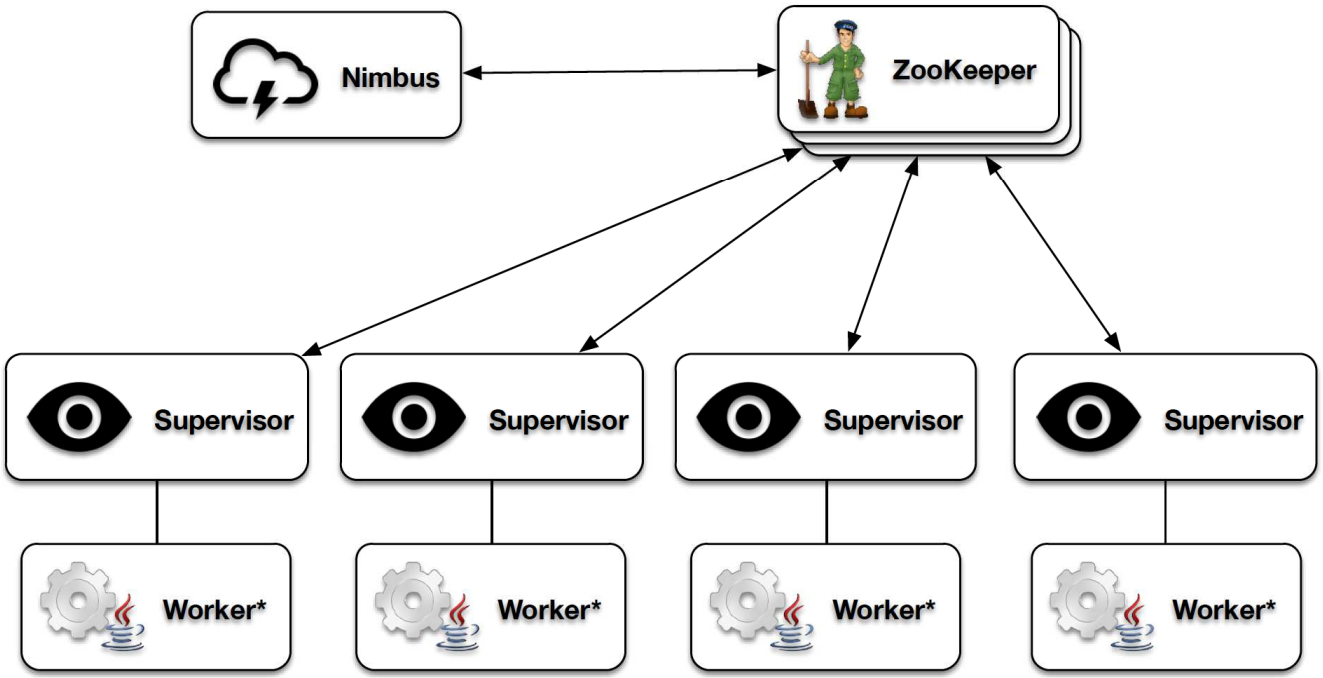
- (Optionally) Emit additional streams

Topologies



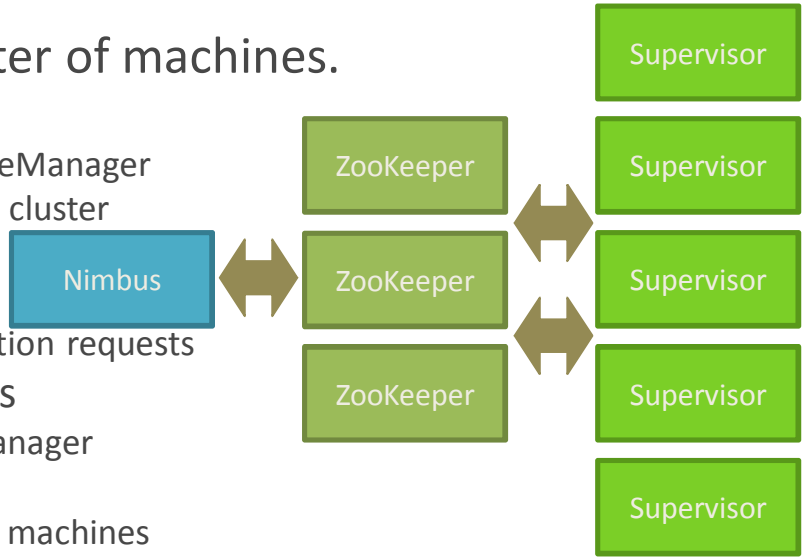
- DAG of Spouts and Bolts
- Data Flow Representation
- *Streaming Computation*

Storm Cluster View



Storm Architecture

- Storm is implemented as a cluster of machines.
  - Nimbus – master node daemon
    - Similar function to YARN ResourceManager
    - Distributes program code around cluster
    - Assigns tasks
    - Handles failures
    - Responds to topology administration requests
  - Supervisor – slave node daemons
    - Similar function to YARN NodeManager
    - Runs bolts and spouts as tasks
    - Commonly runs on Hadoop slave machines
  - ZooKeeper
    - Cluster coordination
    - Stores cluster metrics



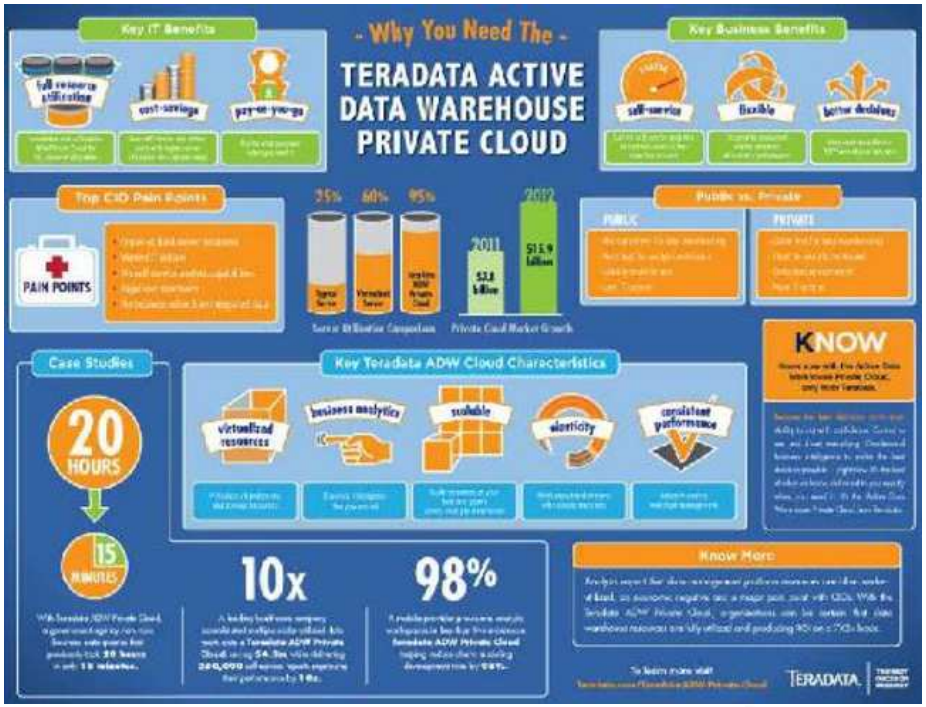
Apache Storm Flow

1. Initially, the nimbus will wait for the “Storm Topology” to be submitted to it.
2. Once a topology is submitted, it will process the topology and gather all the tasks that are to be carried out and the order in which the task is to be executed.
3. Then, the nimbus will evenly distribute the tasks to all the available supervisors.
4. At a particular time interval, all supervisors will send heartbeats to the nimbus to inform that they are still alive.
5. When a supervisor dies and doesn’t send a heartbeat to the nimbus, then the nimbus assigns the tasks to another supervisor.
6. When the nimbus itself dies, supervisors will work on the already assigned task without any issue.
7. Once all the tasks are completed, the supervisor will wait for a new task to come in.
8. In the meantime, the dead nimbus will be restarted automatically by service monitoring tools.
9. The restarted nimbus will continue from where it stopped. Similarly, the dead supervisor can also be restarted automatically. Since both the nimbus and the supervisor can be restarted automatically and both will continue as before, Storm is guaranteed to process all the task at least once.
10. Once all the topologies are processed, the nimbus waits for a new topology to arrive and similarly the supervisor waits for new tasks.

Apache Storm Setup and the first Example

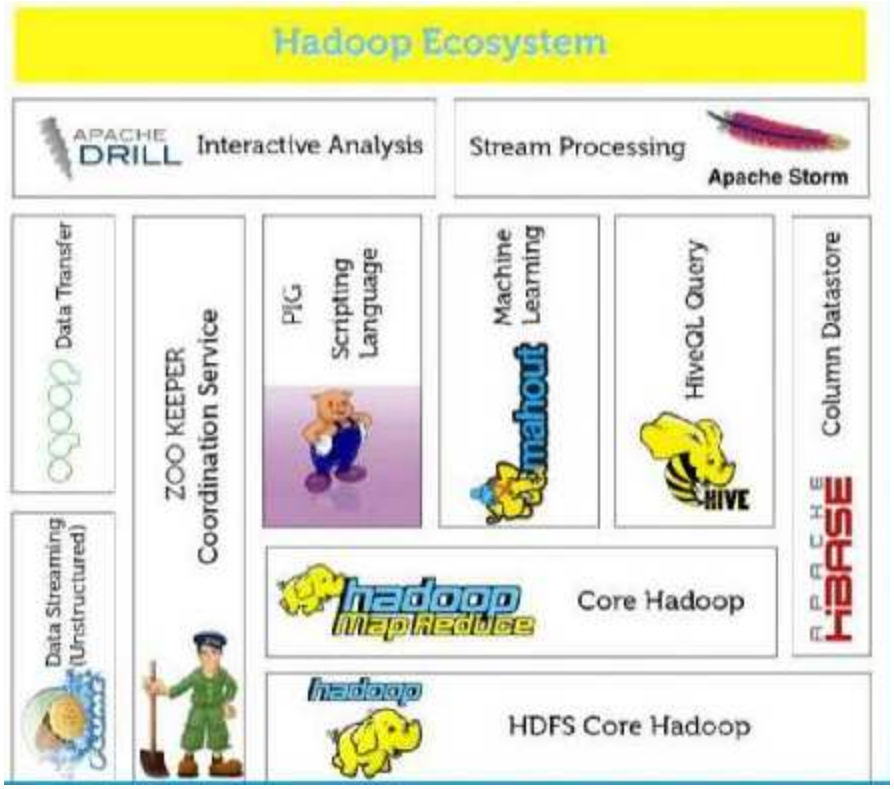
First Generation of Distributed Systems

1. Proprietary
2. Custom Hardware
3. Centralized Data
4. Hardware based Fault Recovery
5. E.g. Teradata, Neteeza



Second Generation of Distributed Systems

1. Open source
2. Commodity Hardware
3. Distributed data
4. Software based Fault recovery
5. E.g. Hadoop



Why do we need a Third Generation

1. Lot has changed from 2003
2. Both Hardware and Software has undergone changes
3. Big Data and Data analytics is now a necessity
4. Batch Processing was the primary way to processing. Limited capability of Real Time Processing in Second Generation
5. Systems not taking advantage of the new Hardware
6. Not designed to change depending on the use case.

Kafka History [Linkedin]

- ✓ Kafka got its start as an internal infrastructure system that was built at LinkedIn. Their observation was really simple: there were lots of databases and other systems built to *store* data, but what was missing in the architecture was something that would help to handle the continuous *flow* of data.

- ✓ Prior to building Kafka, they experimented with all kinds of off the shelf options; from messaging systems to log aggregation and ETL tools, but none of them gave what was wanted.

What is Kafka

Apache Kafka is a distributed streaming platform.

A streaming platform has three key capabilities:

1. Publish and subscribe to streams of records, similar to a message queue or enterprise messaging system.
2. Store streams of records in a fault-tolerant durable way.
3. Process streams of records as they occur.

Kafka is generally used for two broad classes of applications:

1. Building real-time streaming data pipelines that reliably get data between systems or applications
2. Building real-time streaming applications that transform or react to the streams of data

Messaging v/s Kafka

1. Kafka works as a modern distributed system that runs as a cluster and can scale to handle all the applications in even the most massive of companies. Rather than running dozens of individual messaging brokers, hand wired to different apps, this lets you have a central platform that can scale elastically to handle all the streams of data in a company

2. Kafka is a true storage system built to store data for as long as you might like. This has huge advantages in using it as a connecting layer as it provides real delivery guarantees—its data is replicated, persistent, and can be kept around as long as you like.

Messaging v/s Kafka

3. Finally, the world of stream processing raises the level of abstraction quite significantly. Messaging systems mostly just hand out messages. The stream processing capabilities in Kafka let you compute derived streams and datasets dynamically off of your streams with far less code. These differences make Kafka enough of its own thing that it doesn't really make sense to think of it as "yet another queue."

Hadoop v/s Kafka

- ✓ Hadoop lets you store and periodically process file data at a very large scale. Kafka lets you store and continuously process streams of data, also at a large scale.
- ✓ Whereas Hadoop and big data targeted analytics applications, often in the data warehousing space, the low latency nature of Kafka makes it applicable for the kind of core applications that directly power a business.
- ✓ This makes sense: events in a business are happening all the time and the ability to react to them as they occur makes it much easier to build services that directly power the operation of the business, feed back into customer experiences, and so on.

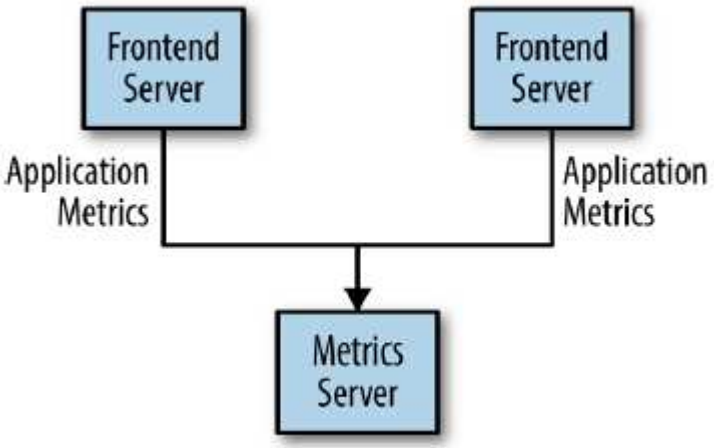
Use Cases of Kafka

https://kafka.apache.org/powered-by

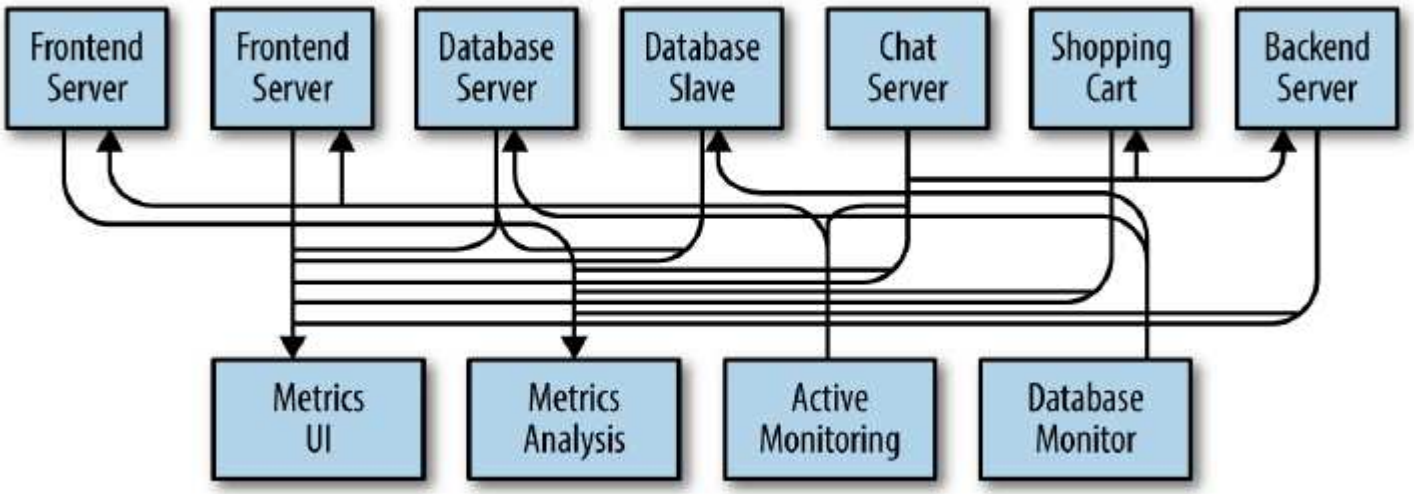
ETL Tools v/s Kafka

- ✓ After all, ETL and Data Integration tools move data around, and Kafka moves data around.
- ✓ Rather than a tool for scraping data out of one system and inserting it into another, Kafka is a platform oriented around real-time streams of events. This means that not only can it connect off-the-shelf applications and data systems, it can power custom applications built to trigger off of these same data streams.
- ✓ This architecture centered around streams of events is a really important thing.

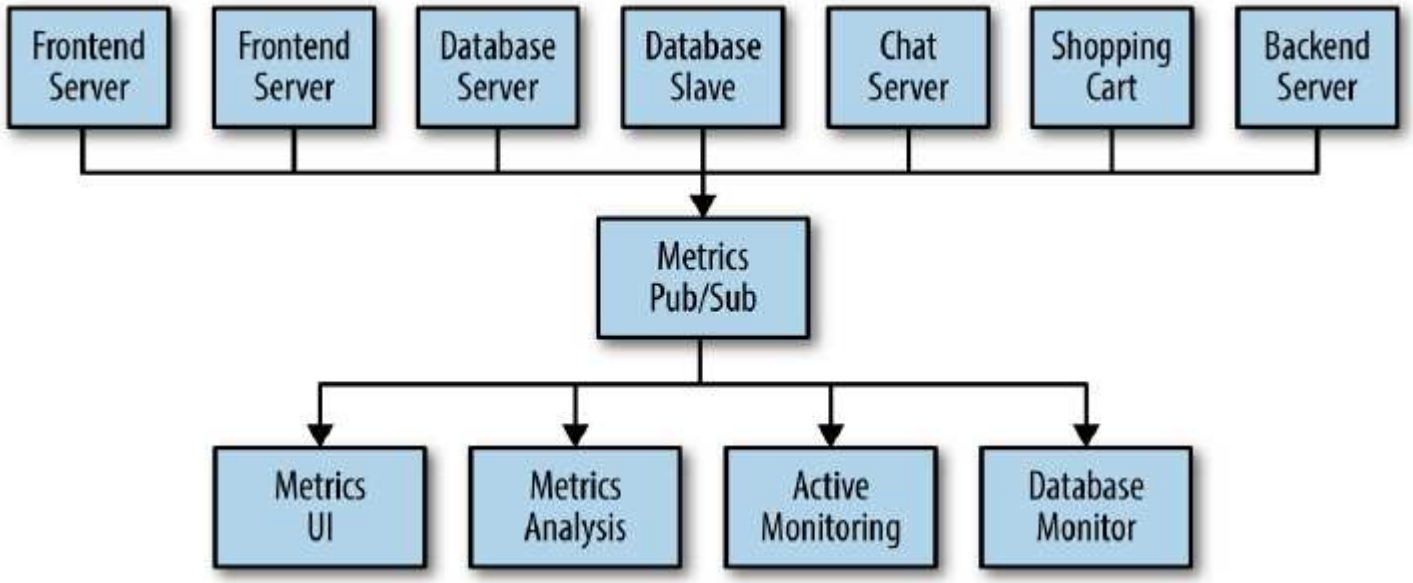
How it starts



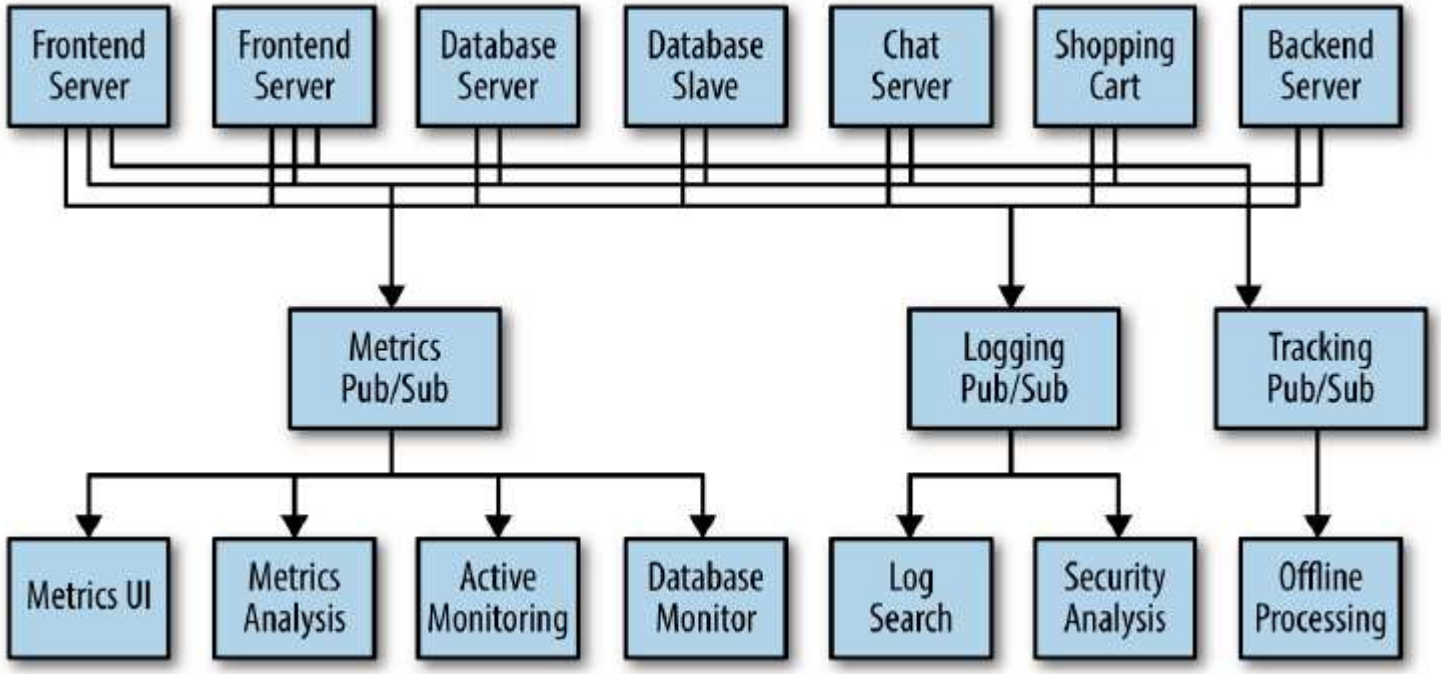
Spaghetti of connections



First Pub-Sub System



Multiple Pub-Sub Systems



Enter Kafka

- ✓ Apache Kafka is often described as a “distributed commit log” or more recently as a “distributing 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 consistently build the state of a system.
- ✓ Similarly, data within Kafka is stored durably, in order, and can be read deterministically. In addition, the data can be distributed within the system to provide additional protections against failures, as well as significant opportunities for scaling performance.

Message

- ✓ The unit of data within Kafka is called a *message.*
- ✓ A message is simply an array of bytes as far as Kafka is concerned, so the data contained within it does not have a specific format or meaning to Kafka.
- ✓ A message can have an optional bit of metadata, which is referred to as a *key* and is also a byte array.
- ✓ Keys are used when messages are to be written to partitions in a more controlled manner.

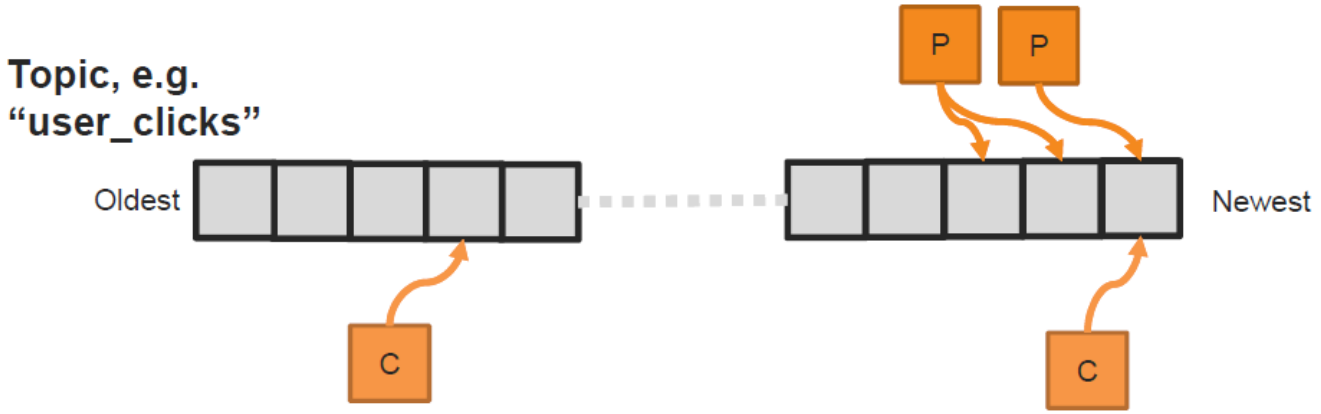
Topics and Partitions

- ✓ Messages in Kafka are categorized into *topics*. The closest analogies for a topic are a database table or a folder in a filesystem. Topics are additionally broken down into a number of *partitions*.

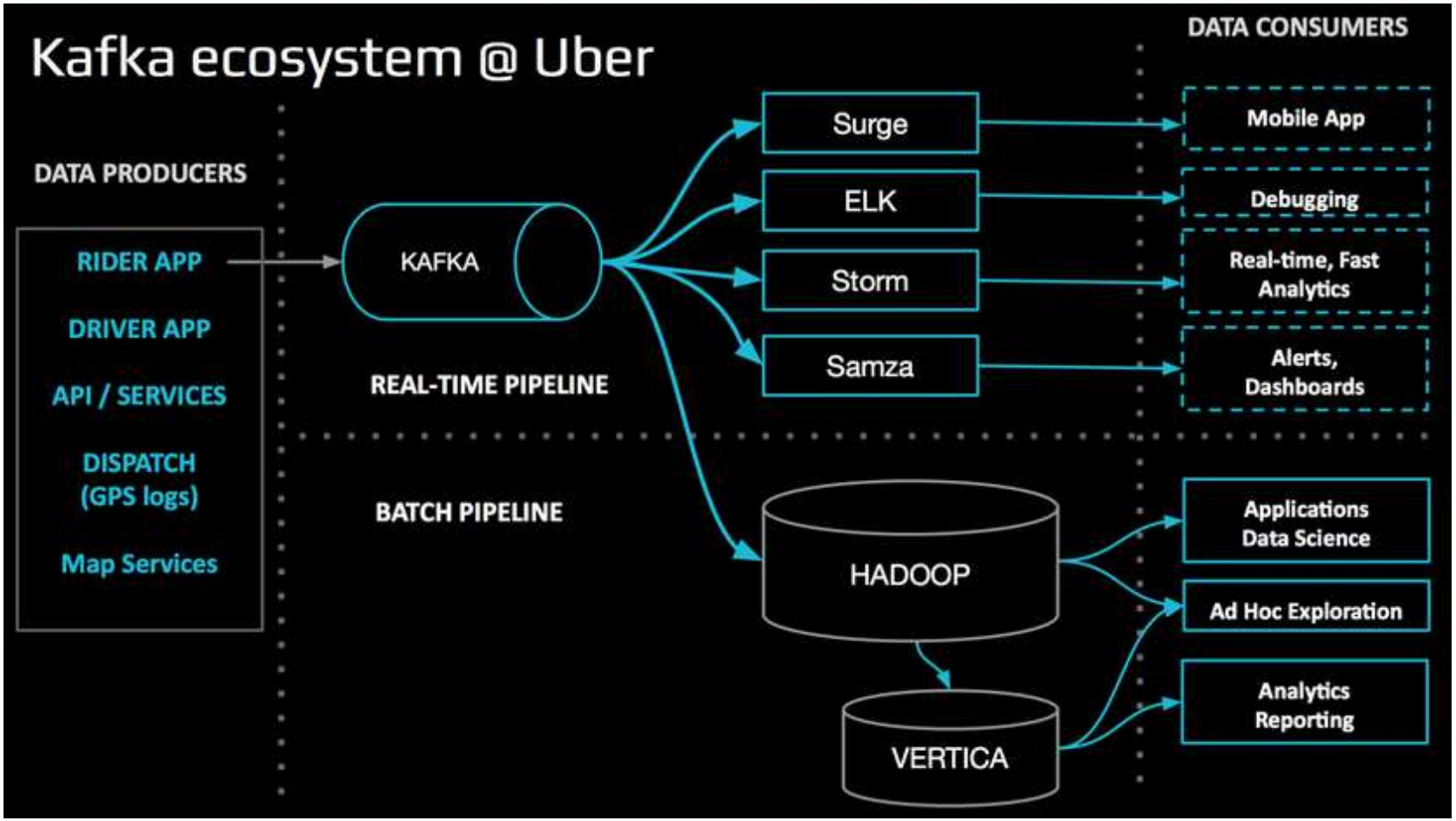
- ✓ Going back to the “commit log” description, a partition is a single log. Messages are written to it in an append-only fashion, and are read in order from beginning to end. Note that as a topic typically has multiple partitions, there is no guarantee of message time-ordering across the entire topic, just within a single partition.

What is Apache Kafka?

Apache Kafka is a publish-subscribe messaging rethought as a distributed commit log.



Uber Use Case



Exercise 1

Understanding Kafka Broker Configurations and working with the Console Producer - Consumer Example

Schema

- ✓ While messages are opaque byte arrays to Kafka itself, it is recommended that additional structure, or schema, be imposed on the message content so that it can be easily understood.
- ✓ Avro provides a compact serialization format; schemas that are separate from the message payloads and that do not require code to be generated when they change; and strong data typing and schema evolution, with both backward and forward compatibility.
- ✓ A consistent data format is important in Kafka, as it allows writing and reading messages to be decoupled

Kafka Producers

- ✓ *Producers* create new messages
- ✓ In general, a message will be produced to a specific topic.
- ✓ By default, the producer does not care what partition a specific message is written to and will balance messages over all partitions of a topic evenly.
- ✓ In some cases, the producer will direct messages to specific partitions. This is typically done using the message 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

Kafka Consumers

- ✓ *Consumers* read messages.

- ➢ The consumer subscribes to one or more topics and reads the messages in the order in which they were produced. The consumer keeps track of which messages it has already consumed by keeping track of the offset of messages.

- ➢ 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.

Consumer Groups

- ✓ Consumers work as part of a *consumer group*, which is one or more consumers that work together to consume a topic.
- ✓ The mapping of a consumer to a partition is often called *ownership* of the partition by the consumer
- ✓ In this way, consumers can horizontally scale to consume topics with a large number of messages. Additionally, if a single consumer fails, the remaining members of the group will rebalance the partitions being consumed to take over for the missing member.

The new Consumer API

- ✓ Prior to Apache Kafka 0.9.0.0, consumers, in addition to the brokers, utilized Zookeeper to directly store information about the composition of the consumer group, what topics it was consuming, and to periodically commit offsets for each partition being consumed (to enable failover between consumers in the group).
- ✓ With version 0.9.0.0, a new consumer interface was introduced which allows this to be managed directly with the Kafka brokers.

Kafka Docs

Kafka APIs:-
https://kafka.apache.org/documentation/#api

Kafka Docs:-
https://kafka.apache.org/11/javadoc/index.html?org/apache/kafka/clients/producer/KafkaProducer.html

Exercise 2

Looking at multiple broker setup and
checking how data is distributed.
Also looking at the Fault Tolerance of Kafka

Exercise 3

Creating a custom Producer via the new Kafka API

Properties in the Custom Kafka Producer

BOOTSTRAP_SERVERS_CONFIG: sets a list of host:port pairs used for establishing the initial connections to the Kakfa cluster in the host1:port1 format.

The Kafka server expects messages in byte[] key, byte[] value format. Rather than converting every key and value, Kafka's client-side library permits us to use friendlier types like String and int for sending messages.

1. KEY_SERIALIZER_CLASS_CONFIG:
2. VALUE_SERIALIZER_CLASS_CONFIG

Exercise 4

Creating a custom Consumer and understanding Customer Records via the new Kafka API

Properties in the Custom Kafka Consumer

1. BOOTSTRAP_SERVERS_CONFIG (bootstrap.servers)
2. KEY_DESERIALIZER_CLASS_CONFIG (key.deserializer)
3. VALUE_DESERIALIZER_CLASS_CONFIG (value.deserializer)
4. GROUP_ID_CONFIG (bootstrap.servers): The GROUP_ID_CONFIG is a group name in string format.

Broker Configurations

1. Zookeeper.connect
2. delete.topic.enable
3. auto.create.topics.enable
4. default.replication.factor and num.partitions
5. log.retention.ms and log.retention.bytes

Details in Page 16 of Kafka_Part1 Document


Thank
You!