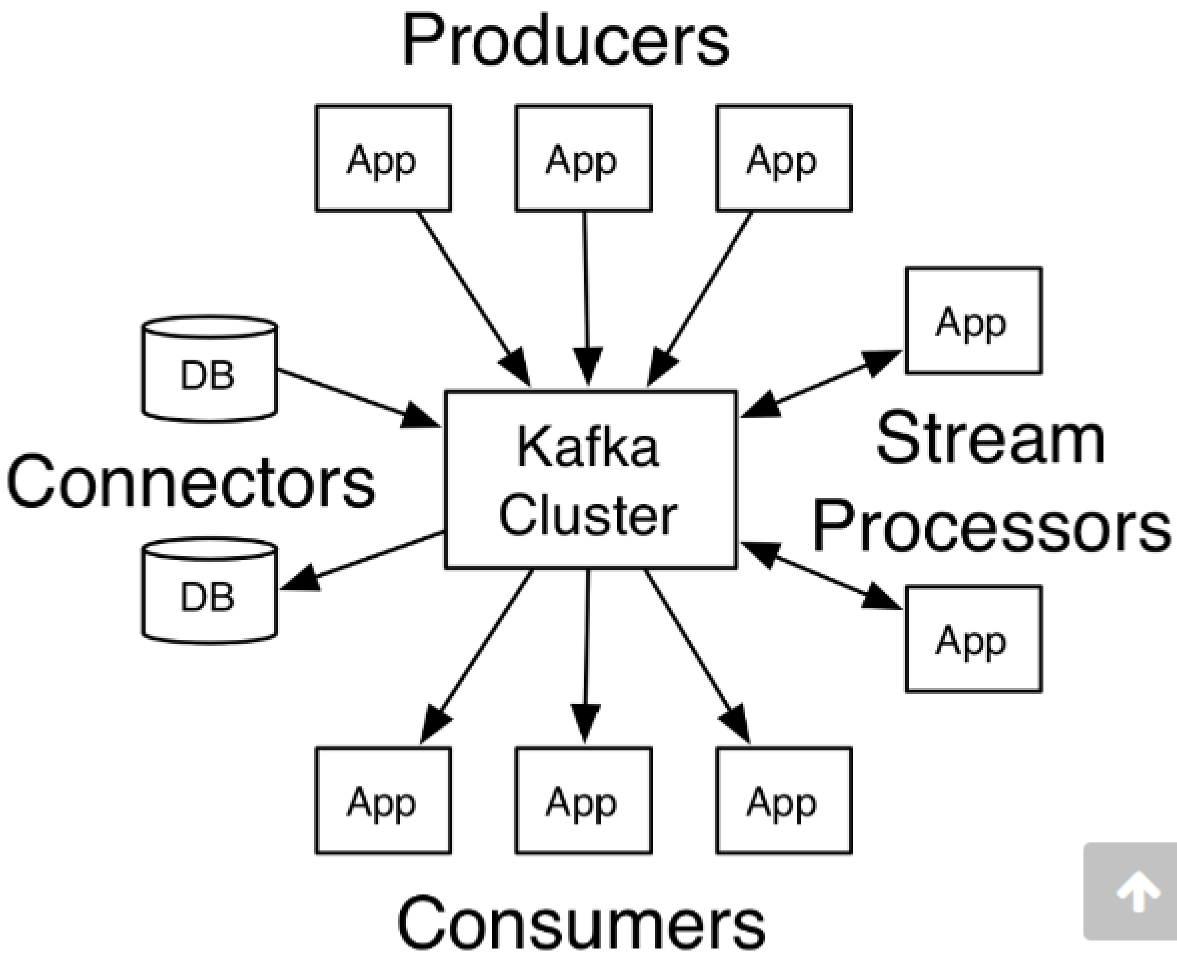
Apache Kafka ¶

Apache Kafka is a fast, scalable, durable, and fault-tolerant publish-subscribe messaging system. It’s written in Scala and Java and uses Apache Zookeeper for reliable distributed coordination.

It's mainly used for handling big volumes of real-time data streams generated by systems like IOT or web services, to provide real-time analytics rather than traditional big data analytics.

Kafka provides four core APIs: **Producer**, **Consumer**, **Streams**, and **Connector**.



The main components of a Kafka system are:

- **topics**: categories that Kafka uses to maintains feeds of messages - represents a stream of records. Each topic has one or more partitions that are physical separations of ordered and immutable sequence of records within a topic.
- **producers**: processes that publish messages to a Kafka topic
- **consumers**: processes that subscribe to topics and process the feed of published messages
- **broker**: server that is part of the cluster that runs Kafka which mantain the published data

In its earlier stages, Kafka was used by LinkedIn only for online and offline real-time event consumption, traditional messaging use cases, gathering system health metrics, activity tracking, and feeding the data streams into their Hadoop grid. However, today, Kafka is a critical part of LinkedIn’s central data pipeline, handling over 1.4 trillion messages a day, as a whole ecosystem has been built around it.

Apache Kafka guarantees:

- Messages sent by a producer to a particular topic partition will be appended in the order they are sent.

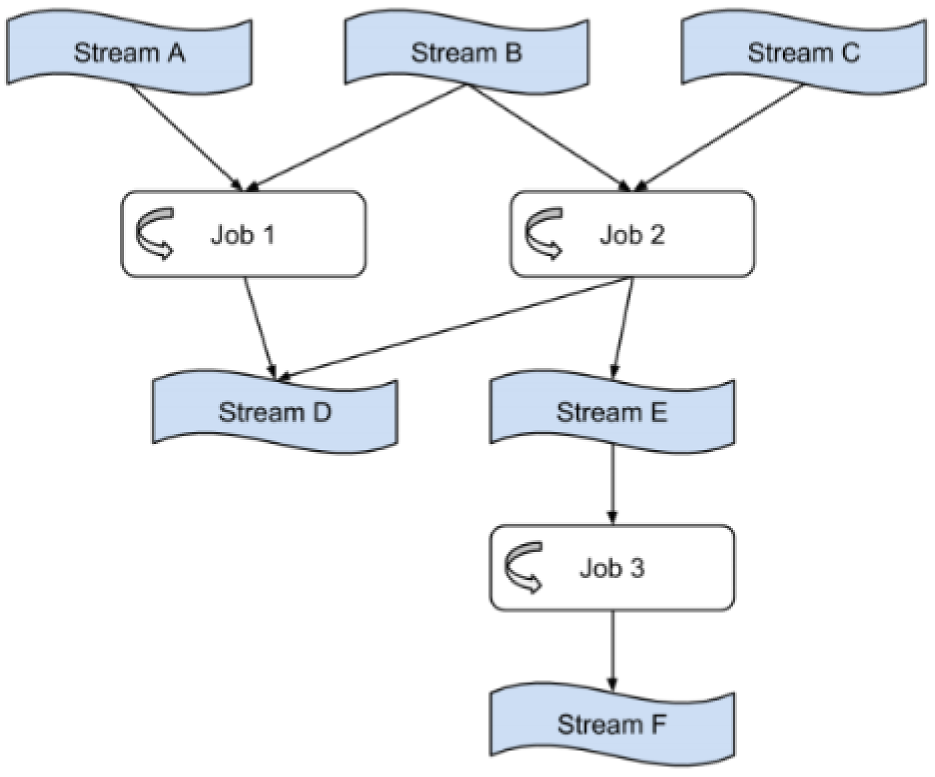
- A consumer instance sees messages in the order they are stored in the log.
- For a topic with replication factor *N*, Kafka tolerates up to *N-1* server failures without losing any messages committed to the log.

Apache Samza

Samza is a stream processing framework with the following features:

- **Simple API**: it provides a very simple callback-based "process message" API comparable to MapReduce.
- **Managed state**: Samza manages snapshotting and restoration of a stream processor's state.
- **Fault tolerance**: Whenever a machine fails, Samza works with YARN to transparently migrate your tasks to another machine.
- **Durability**: Samza uses Kafka to guarantee that messages are processed in the order they were written to a partition, and that no messages are ever lost.
- **Scalability**: Samza is partitioned and distributed at every level. Kafka provides ordered, partitioned, replayable, fault-tolerant streams. YARN provides a distributed environment for Samza containers to run in.
- **Pluggable**: Samza provides a pluggable API that lets you run Samza with other messaging systems and execution environments.
- **Processor isolation**: Samza works with Apache YARN

Samza is made of: **standard components**, which are **streams**, composed of immutable messages of a similar type or category, and **jobs**, code that performs a logical transformation on a set of input streams to append output messages to set of output streams; **parallel components**, which are **partitions**, a part of a stream which is a totally ordered sequence of messages, and **tasks**, a part of a job which can be parallelized just like a partition.



Apache Storm

Apache Storm is a distributed framework that is used for **real-time processing of data streams**. It is an open source project of the Apache Software Foundation. It is tipically used for real time stream processing and continuous computation, either by processing each message as it comes or by creating small batches over a little time, as well as for **Distributed RPC**, used to parallelize an intense function such as a query to compute it in real-time, and **Real-Time Analytics**, analysing and extracting insights from several real-time data streams.

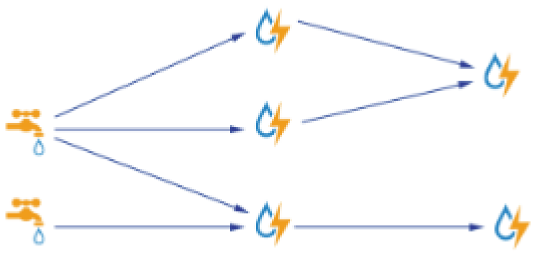
Storm is:

- distributed - can run on clusters of commodity servers horizontally scalable - linearly scalable with respect to the number of nodes, more nodes mean more power fast - can process up to 1 million tuples per second per node
- fault tolerant - worker processes are restarted in case of failure, either on the same node or on another one reliable - guarantees that each message/tuple will be processes *at least once*
- easy to operate - easy to deploy and manage, requires little maintenance applications that run over Storm can be written in any programming language that can read and write to standard input and output streams (even though Storm itself runs on Java VM)

Where Hadoop is based on batch processing and runs jobs until completion, Storm is more focused on real-time processing and continuously running topologies. Just like Hadoop, Storm is scalable, guarantees no data loss, and is Open Source.

Storm processes **streams of tuples**, an unbounded sequence of tuples, each having a name and composed of homogeneous tuples (all tuples have the same structure). Each application can process multiple heterogeneous streams. A **spout** is a component “generating/handling” the input data stream. It reads or listen data from external sources and publish (emit) them into streams.Each spout can emit multiple streams, with different schemas, and can be either “unreliable” (fire-and-forget) or “reliable” (can replay failed tuples). A **bolt** is the component that is used to apply a function over each tuple of a stream. It consumes one or more streams, emitted by spouts or other bolts, and potentially produce new multiple streams, with different schemas. Bolts can be used to:

- filter or transform the content of the input streams and emit new data streams that will be processed by other bolts
- process the data streams and store/persist the result of the computation in some form of “storage” (files, dbs, etc.) The overall network of spouts and bolts is called **topology**.



Part of defining a topology is specifying for each bolt which streams it should receive as input. A stream grouping defines how that stream should be partitioned among the bolt's tasks. Amon the main Storm Groupings:

- **Shuffle grouping**: Tuples are randomly distributed across the bolt's tasks in a way such that each bolt is guaranteed to get an equal number of tuples - ideal for uniform distribution of the load across the tasks.

- **Fields grouping**: The stream is partitioned by the fields specified in the grouping. For example, if the stream is grouped by the "user-id" field, tuples with the same "user-id" will always go to the same task, but tuples with different "user-id"'s may go to different tasks - it does not guarantee that each task will get tuples to process - useful when it is needed to send all the tuples of one or more streams with the same field value to the same task, such as counting tweet number, word frequencies or joining two streams based on common fields
- **Partial Key grouping**: The stream is partitioned by the fields specified in the grouping, like the Fields grouping, but are load balanced between two downstream bolts.
- **All grouping**: The stream is replicated across all the bolt's tasks. Use this grouping with care - a common use case is for sending signals to bolts, such as when filtering on a parameter
- **Global grouping**: The entire stream goes to a single one of the bolt's tasks. Specifically, it goes to the task with the lowest id - useful for a reduce phase to obtain a single final result, which can also be achieved by setting the number of tasks of the bolt to 1 (but this limits the parallelism)

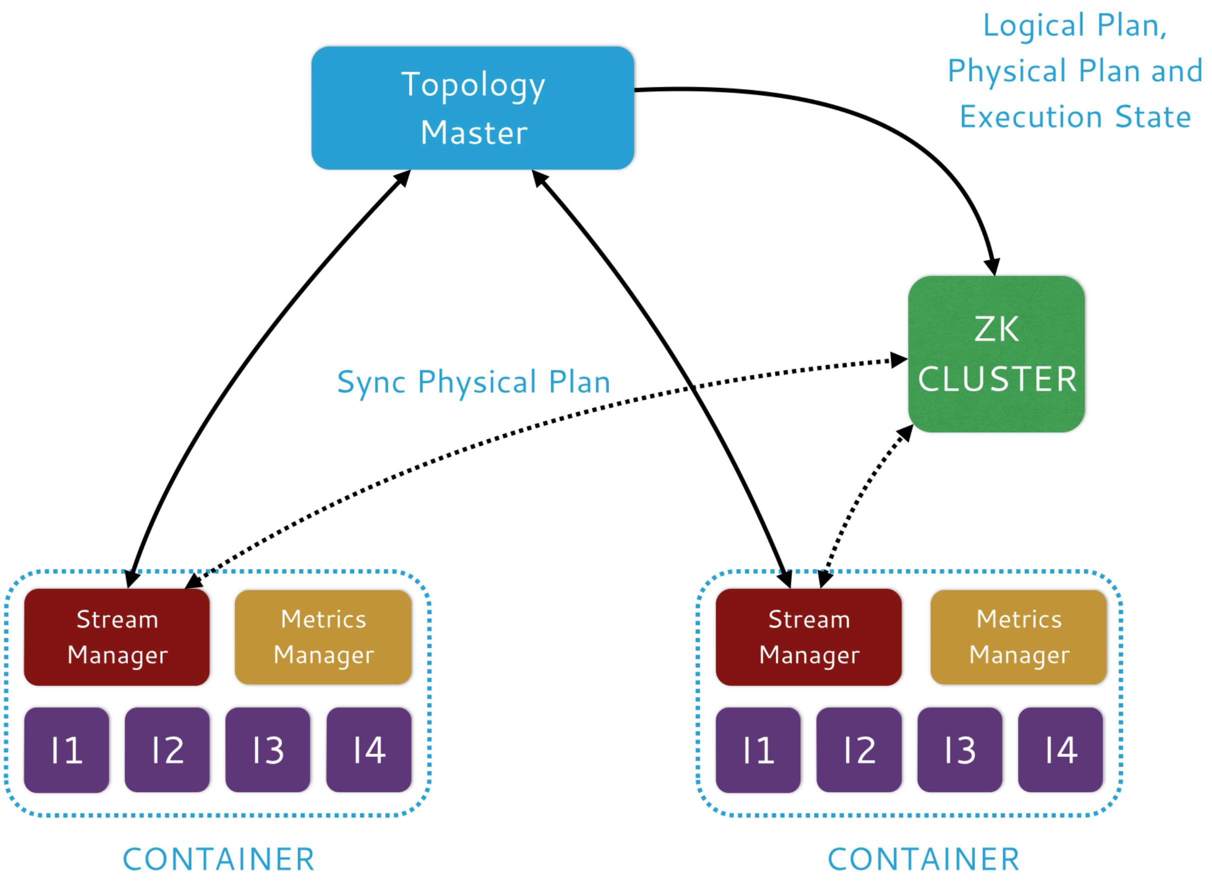
Twitter Heron

Heron is a general-purpose stream processing engine designed for speedy performance, low latency, isolation, reliability, and ease of use for developers and administrators alike. Heron was open sourced by Twitter in May 2016.

A Heron cluster is a mechanism for managing the lifecycle of stream-processing entities called **topologies**. Topologies can be written in Java or Python. Heron's goal is to solve some of the shortcomings with Storm:

- **resource isolation** - topologies, as well as containers within topologies, are process-based isolated
- **resource efficiency** - Heron uses cluster resources on demand
- **throughput** - higher throughput with lower latency

Moreover, Heron is back-compatible with Storm, but it is not open-source.


Topology Master
Logical Plan, Physical Plan and Execution State
ZK CLUSTER
Sync Physical Plan
Stream Manager
Metrics Manager
I1
I2
I3
I4
CONTAINER
Stream Manager
Metrics Manager
I1
I2
I3
I4
CONTAINER