BDI-TT2 ANSWER KEY

1. Schedulers in Apache Spark

- In Spark, the DAG (Directed Acyclic Graph) scheduler and the task scheduler are two important components of the execution engine that work together to execute the user's application code.
- DAG Scheduler
  1. The DAG scheduler is responsible for generating a DAG of stages based on the user's code and optimizing it for efficient execution.
  2. The stages represent a set of tasks that can be executed in parallel.
  3. The DAG scheduler divides the computation into smaller stages based on the dependencies between RDDs (Resilient Distributed Datasets) and operators in the code.
  4. The DAG scheduler then submits the stages to the task scheduler for execution.
- Task Scheduler
  1. The task scheduler is responsible for assigning tasks to workers in the cluster.
  2. It receives the stages from the DAG scheduler and breaks them down into smaller tasks that can be executed in parallel.
  3. The task scheduler then assigns these tasks to workers based on the data locality of the tasks and the availability of resources in the cluster.
  4. It also monitors the progress of the tasks and handles any failures that may occur.
- Together, the DAG scheduler and task scheduler ensure that the user's code is executed efficiently and reliably on the Spark cluster.
- They work in tandem to maximize the parallelism of the computation and minimize the data movement across the network.
- This helps to achieve high performance and scalability when processing large amounts of data.

2. RDD features, advantages and limitations

- RDD is a core abstraction in Spark, which stands for Resilient Distributed Dataset.
- It enables partition of large data into smaller data that fits each machine, so that computation can be done parallelly on multiple machines.
- Moreover, RDDs automatically recover from node failures to ensure the storage resilience.
- It is an immutable fault-tolerant, distributed collection of objects that can be operated on in parallel.
- An RDD can contain any type of object and is created by loading an external dataset or distributing a collection from the driver program.
- RDDs support two types of operations:
  - Transformations are operations (such as map, filter, join, union, and so on) that are performed on an RDD and which yield a new RDD containing the result.
  - Actions are operations (such as reduce, count, first, and so on) that return a value after running a computation on an RDD.
- **Features**:
  - **Resilience**: RDDs track data lineage information to recover lost data, automatically on failure. It is also called fault tolerance.
  - **Distributed**: Data present in an RDD resides on multiple nodes. It is distributed across different nodes of a cluster.
  - **Lazy evaluation**: Data does not get loaded in an RDD even if you define it. Transformations are actually computed when you call action, such as count or collect, or save the output to a file system.
  - **Immutability**: Data stored in an RDD is in the read-only mode - you cannot edit the data which is present in the RDD. But you can create new RDDs by performing transformations on the existing RDDs.
  - **In-memory computation**: An RDD stores any immediate data that is generated in the memory (RAM) than on the disk so that it provides faster access.
  - **Partitioning**: Partitions can be done on any existing RDD to create logical parts that are mutable. You can achieve this by applying transformations to the existing partitions.

- **Advantages of RDD**
  - RDD aids in increasing the execution speed of Spark.
  - RDDs are the basic unit of parallelism and hence help in achieving the consistency of data.
  - RDDs help in performing and saving the actions separately
  - They are persistent as they can be used repeatedly.
- **Limitation of RDD**
  - There is no input optimization available in RDDs
  - One of the biggest limitations of RDDs is that the execution process does not start instantly.
  - No changes can be made in RDD once it is created.
  - RDD lacks enough storage memory.
  - The run-time type safety is absent in RDDs.

3. SPARKQL features, advantages and limitations

- Spark SQL is a Spark component that supports querying data either via SQL or via the HQL. It originated as the Apache Hive port to run on top of Spark (in place of MapReduce) & is now integrated with the Spark stack.
- Spark SQL has SchemaRDD, new data abstraction that provides support for structured & semi-structured data.
- **Features**:
  - **Easy to Integrate**: One can mix SQL queries with Spark programs easily. Structured data can be queried inside Spark programs using either SQL or a Dataframe API. Running SQL queries alongside analytic algorithms is easy because of this tight integration.
  - **Compatibility with Hive**: Hive queries can be executed in Spark SQL as they are.
  - **Unified Data Access**: Loading and querying data from various sources is possible.
  - **Standard Connectivity**: Spark SQL can connect to Java and Oracle using JDBC (Java Database Connectivity) and ODBC (Oracle Database Connectivity) APIs.
  - **Performance and Scalability**: To make queries agile, alongside computing hundreds of nodes using the Spark engine, Spark SQL incorporates a code generator, a cost-based optimizer, and columnar storage. This provides complete mid-query fault tolerance.
- **Advantages**:
  - It helps in easy data querying. The SQL queries are mixed with Spark programs for querying structured data as a distributed dataset (RDD). Also, the SQL queries are run with analytic algorithms using Spark SQL's integration property.
  - Loading and querying can be done for data from different sources. Hence, the data access is unified.
  - It offers standard connectivity as Spark SQL can be connected through JDBC or ODBC.
  - It can be used for faster processing of Hive tables.
  - It can run unmodified Hive queries on existing warehouses as it allows easy compatibility with existing Hive data & queries.
- **Disadvantages**:
  - Creating or reading tables containing union fields is not possible with Spark SQL.
  - It doesn't convey if is any error in situations where varchar is oversized & doesn't support Hive transactions.
  - It also does not support the Char type (fixed-length strings). Hence, reading or creating a table with such fields is not possible.

4. SQL vs NoSQL

SQL	**NoSQL**
Relational database management system (RDBMS)	Non-relational or distributed database system.
These databases have fixed or static or predefined schema	They have a dynamic schema
These databases are not suited for hierarchical data storage.	These databases are best suited for hierarchical data storage.
These databases are best suited for complex queries	These databases are not so good for complex queries
Vertically Scalable	Horizontally scalable
Follows ACID property	Follows CAP (consistency, availability, partition tolerance)
Ex: MySQL, PostgreSQL, Oracle, MS-SQL Server, etc	Ex: MongoDB, HBase, Neo4j, Cassandra, etc

5. MongoDB vs RDBMS

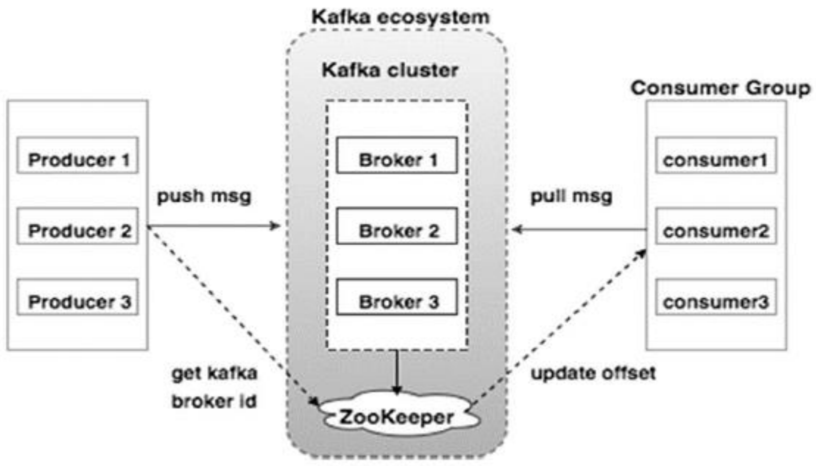
RDBMS	MongoDB
It is a relational database.	It is a non-relational and document-oriented database.
Not suitable for hierarchical data storage.	Suitable for hierarchical data storage.
It is vertically scalable i.e increasing RAM.	It is horizontally scalable i.e we can add more servers.
It has a predefined schema.	It has a dynamic schema.
It is quite vulnerable to SQL injection.	It is not affected by SQL injection.
It centers around ACID properties (Atomicity, Consistency, Isolation, and Durability).	It centers around the CAP theorem (Consistency, Availability, and Partition tolerance).
It is row-based.	It is document-based.
It is slower in comparison with MongoDB.	It is almost 100 times faster than RDBMS.
Supports complex joins.	No support for complex joins.
It is column-based.	It is field-based.
It does not provide JavaScript client for querying.	It provides a JavaScript client for querying.
It supports SQL query language only.	It supports JSON query language along with SQL.

6. Kafka components, cluster architecture, workflow

- Apache Kafka (Kafka) is an open source, distributed streaming platform that enables the development of real-time, event-driven applications.
- Today, billions of data sources continuously generate streams of data records, including streams of events. An event is a digital record of an action that happened and the time that it happened. Typically, an event is an action that drives another action as part of a process.
- A customer placing an order, choosing a seat on a flight, or submitting a registration form are all examples of events.
- A streaming platform enables developers to build applications that continuously consume and process these streams at extremely high speeds, with a high level of fidelity and accuracy based on the correct order of their occurrence.
- LinkedIn developed Kafka in 2011 as a high-throughput message broker for its own use, then open-sourced and donated Kafka to the Apache Software Foundation.
- **Components of Apache Kafka**
    - **Topics**: A stream of messages that are a part of a specific category or feed name is referred to as a Kafka topic. In Kafka, data is stored in the form of topics. Producers write their data to topics, and consumers read the data from these topics.
    - **Brokers**: A Kafka cluster comprises one or more servers that are known as brokers. In Kafka, a broker works as a container that can hold multiple topics with different partitions. A unique integer ID is used to identify brokers in the Kafka cluster. Connection with any one of the Kafka brokers in the cluster implies a connection with the whole cluster. If there is more than one broker in a cluster, the brokers need not contain the complete data associated with a particular topic.
    - **Consumers and Consumer Groups**: Consumers read data from the Kafka cluster. The data to be read by the consumers has to be pulled from the broker when the consumer is ready to receive the message. A consumer group in Kafka refers to a number of consumers that pull data from the same topic or same set of topics.
    - **Producers**: Producers in Kafka publish messages to one or more topics. They send data to the Kafka cluster. Whenever a Kafka producer publishes a message to Kafka, the broker receives the message and appends it to a particular partition. Producers are given a choice to publish messages to a partition of their choice.
    - **Partitions**: Topics in Kafka are divided into a configurable number of parts, which are known as partitions. Partitions allow several consumers to read data from a particular topic in parallel. Partitions are separated in order. The number of partitions is specified when configuring a topic, but this number can be changed later on. The partitions comprising a topic are distributed across servers in the Kafka cluster. Each server in the cluster handles the data and requests for its share of partitions. Messages are sent to the broker along with a key. The key can be used to determine which partition that particular message will go to. All messages which have the same key go to the same partition. If the key is not specified, then the partition will be decided in a round-robin fashion.

- **Partition Offset**: Messages or records in Kafka are assigned to a partition. To specify the position of the records within the partition, each record is provided with an offset. A record can be uniquely identified within its partition using the offset value associated with it. A partition offset carries meaning only within that particular partition. Older records will have lower offset values since records are added to the ends of partitions.
- **Replicas**: Replicas are like backups for partitions in Kafka. They are used to ensure that there is no data loss in the event of a failure or a planned shutdown. Partitions of a topic are published across multiple servers in a Kafka cluster. Copies of the partition are known as Replicas.
- **Leader and Follower**: Every partition in Kafka will have one server that plays the role of a leader for that particular partition. The leader is responsible for performing all the read and write tasks for the partition. Each partition can have zero or more followers. The duty of the follower is to replicate the data of the leader. In the event of a failure in the leader for a particular partition, one of the follower nodes can take on the role of the leader.

Cluster Architecture



- **Broker**
  - Kafka cluster typically consists of multiple brokers to maintain load balance. Kafka brokers are stateless, so they use ZooKeeper for maintaining their cluster state.
  - One Kafka broker instance can handle hundreds of thousands of reads and writes per second and each broker can handle TB of messages without performance impact.
  - Kafka broker leader election can be done by ZooKeeper.
- **ZooKeeper**
  - ZooKeeper is used for managing and coordinating Kafka broker.
  - ZooKeeper service is mainly used to notify producer and consumer about the presence of any new broker in the Kafka system or failure of the broker in the Kafka system.
  - As per the notification received by the Zookeeper regarding presence or failure of the broker then producer and consumer takes decision and starts coordinating their task with some other broker.
- **Producers**
  - Producers push data to brokers. When the new broker is started, all the producers search it and automatically sends a message to that new broker.
  - Kafka producer doesn't wait for acknowledgements from the broker and sends messages as fast as the broker can handle.
- **Consumers**
  - Since Kafka brokers are stateless, which means that the consumer has to maintain how many messages have been consumed by using partition offset.
  - If the consumer acknowledges a particular message offset, it implies that the consumer has consumed all prior messages.
  - The consumer issues an asynchronous pull request to the broker to have a buffer of bytes ready to consume.
  - The consumers can rewind or skip to any point in a partition simply by supplying an offset value. Consumer offset value is notified by ZooKeeper.

Workflow

- Following is the stepwise workflow of the Pub-Sub Messaging in Apache Kafka.
  1. Producers send message to a topic at regular intervals.
  2. Kafka broker stores all messages in the partitions configured for that particular topic. It ensures the messages are equally shared between partitions. If the producer sends two messages and there are two partitions, Kafka will store one message in the first partition & the second message in the second partition.
  3. Consumer subscribes to a specific topic.

4. Once the consumer subscribes to a topic, Kafka will provide the current offset of the topic to the consumer and also saves the offset in the Zookeeper ensemble.
5. Consumer will request the Kafka in a regular interval (like 100 Ms) for new messages.
6. Once Kafka receives the messages from producers, it forwards these messages to the consumers.
7. Consumer will receive the message and process it.
8. Once the messages are processed, consumer will send an acknowledgement to the Kafka broker.
9. Once Kafka receives an acknowledgement, it changes the offset to the new value and updates it in the Zookeeper. Since offsets are maintained in the Zookeeper, the consumer can read next message correctly even during server outrages.
10. This above flow will repeat until the consumer stops the request.
11. Consumer has the option to rewind/skip to the desired offset of a topic at any time and read all the subsequent messages.

- Kafka has three primary capabilities:
  1. It enables applications to publish or subscribe to data or event streams.
  2. It stores records accurately (i.e., in the order in which they occurred) in a fault-tolerant and durable way.
  3. It processes records in real-time (as they occur).
- Developers can leverage these Kafka capabilities through four APIs:

  1. **Producer API:** This enables an application to publish a stream to a Kafka topic. A topic is a named log that stores the records in the order they occurred relative to one another. After a record is written to a topic, it can't be altered or deleted; instead, it remains in the topic for a preconfigured amount of time—for example, for two days—or until storage space runs out.
  2. **Consumer API**: This enables an application to subscribe to one or more topics and to ingest and process the stream stored in the topic. It can work with records in the topic in real-time, or it can ingest and process past records.
  3. **Streams API**: This builds on the Producer and Consumer APIs and adds complex processing capabilities that enable an application to perform continuous, front-to-back stream processing—specifically, to consume records from one or more topics, to analyze or aggregate or transform them as required, and to publish resulting streams to the same topics or other topics. While the Producer and Consumer APIs can be used for simple stream processing, it's the Streams API that enables development of more sophisticated data- and event-streaming applications.
  4. **Connector API:** This lets developers build connectors, which are reusable producers or consumers that simplify and automate the integration of a data source into a Kafka cluster.

7. Storm components, cluster architecture, workflow

- 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. It is a streaming data framework that has the capability of highest ingestion rates. Though Storm is stateless, it manages distributed environment and cluster state via Apache ZooKeeper. It is simple and you can execute all kinds of manipulations on real-time data in parallel.
- **Components**:

The storm comprises of 3 abstractions namely Spout, Bolt and Topologies.

- **Spout** helps us to retrieve the data from the queueing systems. Spout implementations exist for most of the queueing systems.
- **Bolt** used to process the input streams of data retrieved by spout from queueing systems. Most of the logical operations are performed in bolt such as filters, functions, talking to databases, streaming joins, streaming aggregations and so on.
- **Topology** is a combination of both Spout and Bolt. Data transferred between spout and Bolt is called as Tuple. Tuple is a collection of values like messages and so on.

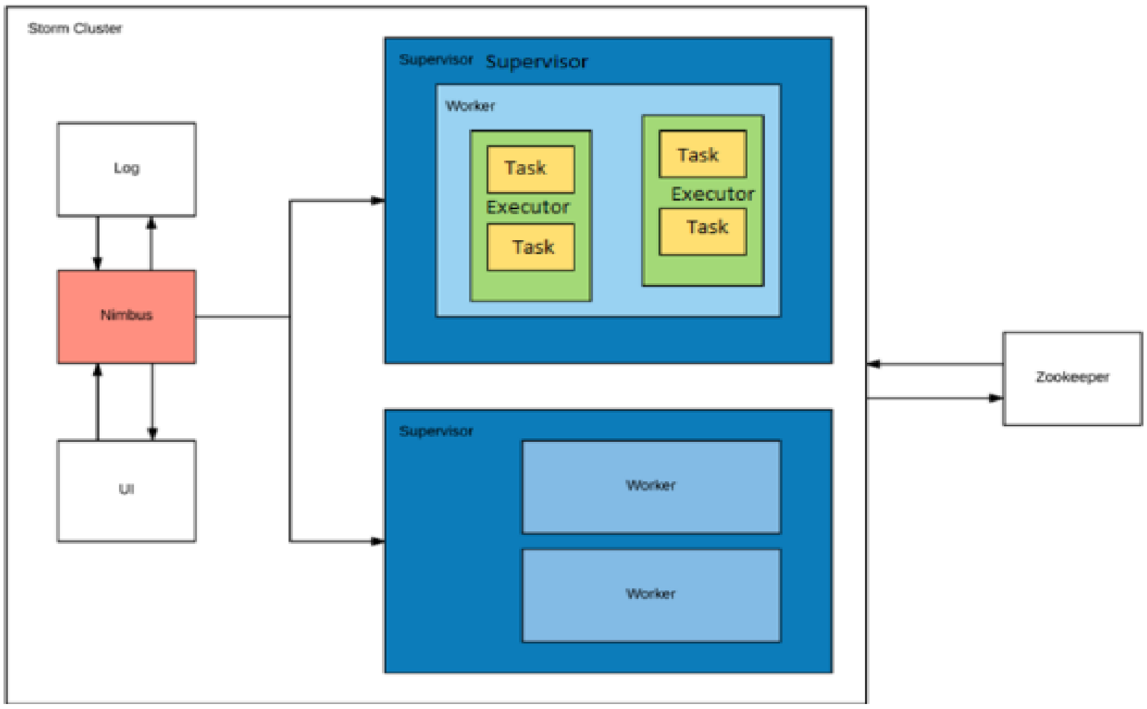
Nodes of Apache Storm

A Storm cluster has 3 sets of nodes namely Nimbus, Zookeeper and Supervisor nodes.

- **Nimbus node** is also called as a master node and acts as a job tracker. It is responsible for distributing the code among supervisors, assigning input data sets to machines for processing and monitoring for failures. Nimbus node relies on zookeeper service to monitor the message processing tasks as all the supervisor nodes update their tasks status in zookeeper node.

- **Zookeeper node** acts as an intermediate between nimbus and supervisor node. Communication and data exchange takes place with the help of zookeeper node. It helps to coordinate the entire cluster.
- **Supervisor node** is also called as worker node and acts as task tracker. It receives the assigned work to a machine by nimbus service. It manages work processes to complete tasks assigned by Nimbus. It will start and stop the workers according to the signals from Nimbus. All supervisor nodes are interconnected with each other with the help of Zero MQ (Messaging Queue). The tasks from one supervisor node can be transferred to another by using Zero MQ.

Cluster Architecture



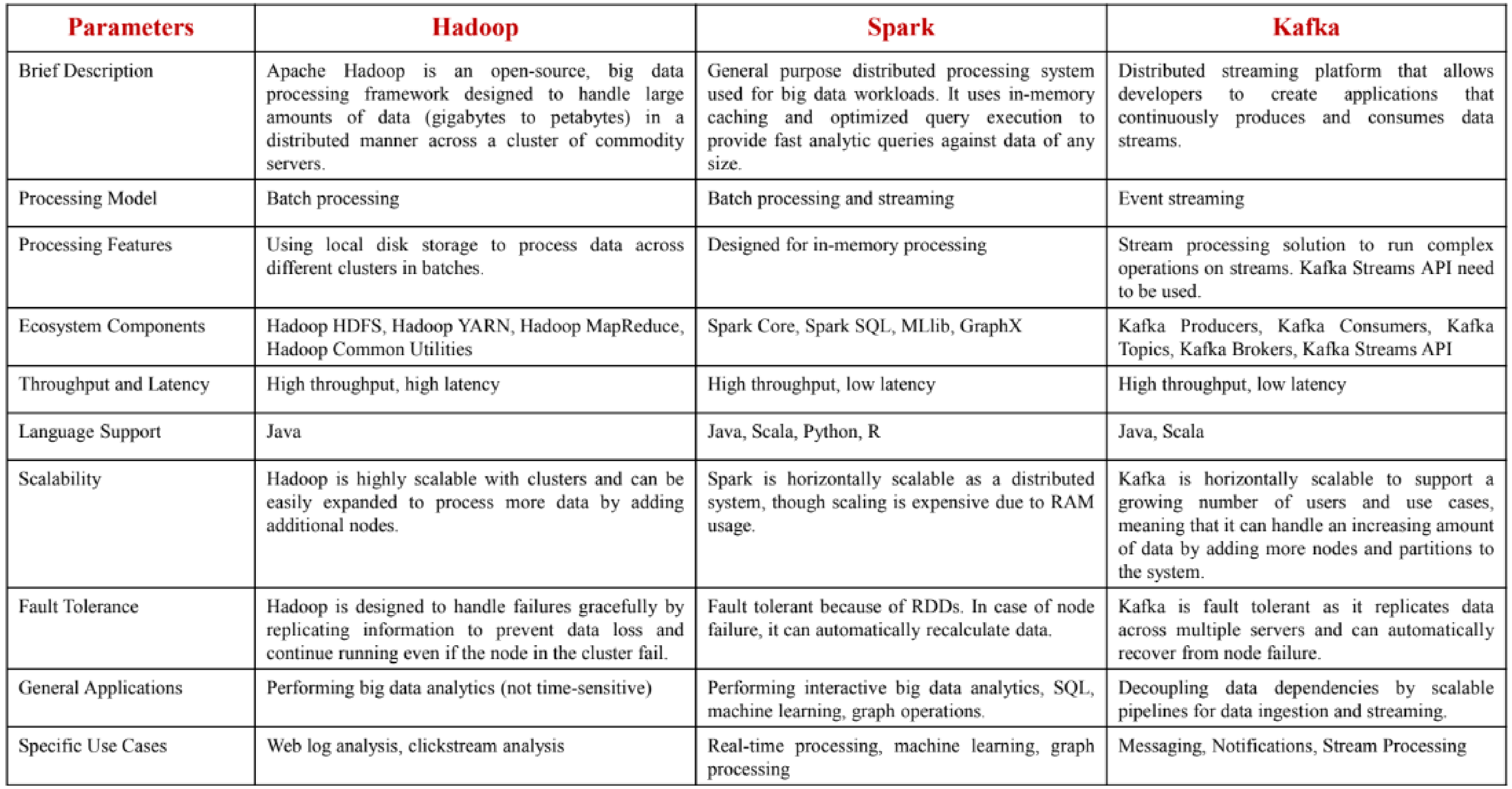
- **Nimbus**: Nimbus is a master node of Storm cluster. All other nodes in the cluster are called as worker nodes. Master node is responsible for distributing data among all the worker nodes, assign tasks to worker nodes and monitoring failures.
- **Supervisor**: The nodes that follow instructions given 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.
- **Worker process**: 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.
- **Executor**: 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.
- **Task**: A task performs actual data processing. So, it is either a spout or a bolt.
- **ZooKeeper framework:** Apache ZooKeeper is a service used by a cluster (group of nodes) to coordinate between themselves and maintaining shared data with robust synchronization techniques. Nimbus is stateless, so it depends on ZooKeeper to monitor the working node status. ZooKeeper helps the supervisor to interact with the nimbus. It is responsible to maintain the state of nimbus and supervisor.

Workflow

- Firstly, the nimbus will wait for the storm topology to be submitted to it.
- When the 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 execute.
- At a stipulated time-interval, all supervisors will send status (alive or dead) to the nimbus to inform that they are still alive.
- If a supervisor dies and doesn’t address the status to the nimbus, then the nimbus assigns the tasks to another supervisor.
- When the Nimbus itself dies, the supervisor will work on an already assigned task without any interruption or issue.

- When all tasks are completed, the supervisor will wait for a new task to process.
- In a meanwhile, the dead nimbus will be restarted automatically by service monitoring tools.
- The restarted nimbus will continue from where it stopped working. The dead supervisor can restart automatically. Hence itis guaranteed that the entire task will be processed at least once.

8. Hadoop Vs Spark Vs Kafka



Parameters	Hadoop	Spark	Kafka
Brief Description	Apache Hadoop is an open-source, big data processing framework designed to handle large amounts of data (gigabytes to petabytes) in a distributed manner across a cluster of commodity servers.	General purpose distributed processing system used for big data workloads. It uses in-memory caching and optimized query execution to provide fast analytic queries against data of any size.	Distributed streaming platform that allows developers to create applications that continuously produces and consumes data streams.
Processing Model	Batch processing	Batch processing and streaming	Event streaming
Processing Features	Using local disk storage to process data across different clusters in batches.	Designed for in-memory processing	Stream processing solution to run complex operations on streams. Kafka Streams API need to be used.
Ecosystem Components	Hadoop HDFS, Hadoop YARN, Hadoop MapReduce, Hadoop Common Utilities	Spark Core, Spark SQL, MLlib, GraphX	Kafka Producers, Kafka Consumers, Kafka Topics, Kafka Brokers, Kafka Streams API
Throughput and Latency	High throughput, high latency	High throughput, low latency	High throughput, low latency
Language Support	Java	Java, Scala, Python, R	Java, Scala
Scalability	Hadoop is highly scalable with clusters and can be easily expanded to process more data by adding additional nodes.	Spark is horizontally scalable as a distributed system, though scaling is expensive due to RAM usage.	Kafka is horizontally scalable to support a growing number of users and use cases, meaning that it can handle an increasing amount of data by adding more nodes and partitions to the system.
Fault Tolerance	Hadoop is designed to handle failures gracefully by replicating information to prevent data loss and continue running even if the node in the cluster fail.	Fault tolerant because of RDDs. In case of node failure, it can automatically recalculate data.	Kafka is fault tolerant as it replicates data across multiple servers and can automatically recover from node failure.
General Applications	Performing big data analytics (not time-sensitive)	Performing interactive big data analytics, SQL, machine learning, graph operations.	Decoupling data dependencies by scalable pipelines for data ingestion and streaming.
Specific Use Cases	Web log analysis, clickstream analysis	Real-time processing, machine learning, graph processing	Messaging, Notifications, Stream Processing