**Kafka**

*Kafka is a distributed event streaming platform*. It's designed to handle high-throughput, real-time data feeds, making it particularly suitable for building real-time data pipelines and streaming applications.

Kafka allows for the publishing and subscribing to streams of records in a fault-tolerant, durable manner. It's commonly used for building data pipelines, log aggregation, real-time analytics, data ingestion, and event-driven architectures.

An event streaming platform captures events in order and these streams of events are stored durably for processing, manipulation, and responding to in real time or to be retrieved later. Event streaming ensures a continuous flow and interpretation of data so that the right information is at the right place, at the right time.

**Use cases**

As a messaging system

Activity tracking

To gather metrics data.

## Architecture Diagram

## 

**Terminology**

Kafka is a distributed system consisting of different kinds of servers and clients that communicate events via a high-performance TCP network protocol. These servers and clients are all designed to work together

**Brokers**

A broker refers to a server in the Kafka storage layer that stores event streams from one or more sources.

**Topics**

Kafka cluster organizes and durably stores streams of events in categories called topics. A topic is a log of events, similar to a folder in a filesystem, where events are the files in that folder.

Characteristics: -Topic is append only, Events in the topic are immutable, Topics in Kafka are always multi-producer and multi-subscriber

**Producers**

Producers are clients that write events to Kafka. The producer specifies the topics they will write to and the producer controls how events are assigned to partitions within a topic.

**Consumers**

Consumers are clients that read events from Kafka. They can subscribe to one or more topics and process the incoming messages.

**Partitions**

Topics are broken up into partitions, meaning a single topic log is broken into multiple logs located on different Kafka brokers. This way, the work of storing messages, writing new messages, and processing existing messages can be split among many nodes in the cluster.

**Offset**

An offset is a unique identifier associated with each record within a partition. It helps Kafka keep track of what has been consumed so far.

**Replication**

Replication is an important part of keeping your data highly-available and fault tolerant. Every topic can be replicated, even across geo-regions or datacenters. A common production setting is a replication factor of 3, meaning there will always be three copies of your data. This replication is performed at topic partition level.

**Schema Registry**

The Schema Registry provides a serving layer for your metadata. It provides a RESTful interface for storing and retrieving Avro schemas. It stores a versioned history of all schemas and allows evolution of schemas according to the configured compatibility setting.

**Kafka Connect**

Kafka Connect is a tool for scalably and reliably streaming data between Apache Kafka and other systems. It provides connectors for importing/exporting data from/to various sources like databases, message queues, search indexes, etc.

There are two different types of connectors:

* Source connectors that act as producers for Kafka
* Sink connectors that act as consumers for Kafka

**Kafka Streams**

Kafka Streams is a client library for building applications and micro services, where the input and output data are stored in Kafka clusters.

Streams help solve problems such as: handling out-of-order data, reprocessing input as code changes, performing stateful computations, etc.

Streams builds on the core Kafka primitives, specifically it uses:

* The producer and consumer APIs for input Kafka for stateful storage
* The same group mechanism for fault tolerance among the stream processor instances

**Kafka Clients**

These are libraries that allow applications to produce and consume messages to/from Kafka topics. There are clients available for various languages including Java, C/C++, Python,.NET, Node.js, Go, Rust, and others

**ZooKeeper**

ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. All Kafka brokers will use ZooKeeper to elect the leader among themselves at startup. ZooKeeper is also used for maintaining the cluster membership and configuration.

**Log**

In Kafka, a log is simply a sequence of records, where each record consists of a key, a value, and a timestamp. The term "log" here refers to the concept of a log file in computing, not the traditional database sense.

**Migration Points**

***Stateful Components:***

Brokers: Kafka brokers maintain the state of the data they store. When migrating, the state of the data needs to be preserved to avoid data loss.

ZooKeeper: ZooKeeper holds the state of the Kafka cluster, including leadership elections and configurations. Its state must be transferred during migration.

***Stateless Components:***

Producers and Consumers: These clients do not maintain state about the data they produce or consume. Their operation depends on the current state of the Kafka cluster and the data it contains.

Kafka Connect: While Kafka Connect itself is stateful in terms of connector configurations, the data being streamed does not inherently carry state beyond the current transaction.

**1. Ensure All Kafka Brokers Are Running the Latest Compatible Versions**

#bin/kafka-topics.sh --version

**2. Backup ZooKeeper Data and Kafka Logs**

->ZooKeeper data is stored in the /var/lib/zookeeper/data

#sudo tar czvf zookeeper-backup.tar.gz /var/lib/zookeeper/data

->Backing Up Kafka Logs

Kafka logs are stored in the directories specified by the log.dirs configuration option in the Kafka broker configuration file (server.properties). Typically, this is set to a directory like /var/lib/kafka/data

#sudo tar czvf kafka-logs-backup.tar.gz /var/lib/kafka/data

**3.Transferring it to a cloud storage**

Also keep in mind below points during migration   
***1. Network Configuration***

**Security Groups and Network Policies:** Ensure that the security groups and network policies in EKS allow for communication between your Kafka brokers, producers, and consumers. This includes allowing inbound and outbound traffic on the ports used by Kafka (default is 9092).

**DNS and Routing:** If you're changing the IP addresses or domain names of your Kafka brokers, ensure that DNS entries are updated accordingly and that routing is correctly configured to point to the new locations.

***2. Storage and Persistence***

**Persistent Volumes:** In EKS, use Persistent Volumes (PVs) and Persistent Volume Claims (PVCs) for Kafka and ZooKeeper data storage. This ensures that your data persists across node failures and cluster upgrades.

**Storage Class:** Choose an appropriate StorageClass based on your performance and cost requirements. Consider using SSD-backed storage classes for better performance

**Step 1: Install Strimzi Operator**

Strimzi operator simplifies the deployment and management of Kafka on Kubernetes.

**kubectl apply -f** [**https://strimzi.io/install/latest?namespace=kafka-operator**](https://strimzi.io/install/latest?namespace=kafka-operator)

Step 2: Deploy Kafka Using Strimzi

define a Kafka custom resource definition (CRD) to specify your Kafka cluster configuration. Create a YAML file named kafka-cluster.yaml with the desired configuration for your Kafka cluster

apiVersion: kafka.strimzi.io/v1beta2

kind: Kafka

metadata:

name: my-kafka-cluster

namespace: kafka-operator

spec:

kafka:

version: 2.8.0

replicas: 3

listeners:

plain: {}

config:

offsets.topic.replication.factor: 3

transaction.state.log.replication.factor: 3

transaction.state.log.min.isr: 2

zookeeper:

replicas: 3

image: strimzi/zookeeper:0.28.0

entityOperator:

topicOperator: {}

userOperator: {}

Step 3: Verify Kafka Deployment

kubectl get kafka my-kafka-cluster -n kafka-operator

kubectl get pods -n kafka-operator

Step 4: Migrate Data

* Export Data: Use Kafka's tools (kafka-topics.sh, kafka-console-producer.sh, etc.) to export your data.
* Import Data: Once your new Kafka cluster is ready, use similar tools to import your data.

**Scylla**

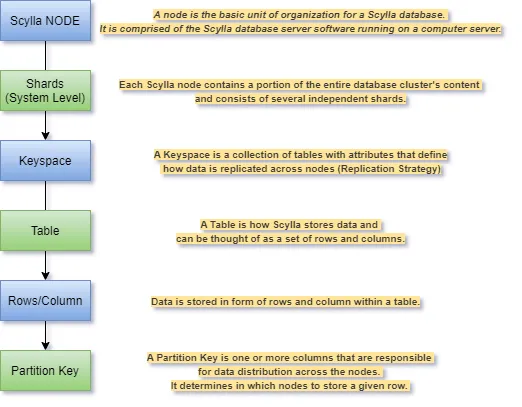
Scylla is a highly performant distributed NoSQL database. It's designed as a drop-in replacement for Apache Cassandra but promises better performance and lower latencies.

It is built around the C++ Seastar framework and the Apache Cassandra Query Language (CQL), making it compatible with existing Cassandra applications while offering improved performance.

**Scylla's Use Cases**:

Scylla is well-suited for applications that require real-time analytics, IoT data processing, time-series data storage, and caching layers. Its performance characteristics make it ideal for use cases where traditional databases would struggle to keep up with demand.

**Scylla components:**



**Node:** A Node is a unit of storage in Scylla. It is comprised of the Scylla database server software running on a computer server — a physical machine — and all its subsystems (CPUs, memory, storage, network interfaces and so on), or, in a virtualized environment, a subset of a server’s resources assigned to a container.

**Cluster:** A minimum Cluster typically consists of at least 3 nodes. Data is replicated across the cluster, depending on the Replication Factor.

**Table:** A Table is how Scylla stores data and can be thought of as a set of rows and columns.

**Keyspace:** A Keyspace is a collection of tables with attributes that define how data is replicated on nodes. It defines several options that apply to all the tables it contains, most prominently of which is the replication strategy used by the Keyspace. It is generally encouraged to use one Keyspace per application, and thus a Cluster may define only one Keyspace.

**CQL:** A query language for interacting with the Scylla (or Cassandra) database.

**CQL Shell:** A command-line interface for interacting with Scylla through the Cassandra Query Language (CQL)

**Replication:** The process of replicating data across Nodes in a Cluster.

**Consistency Level:** A configurable setting which dictates how many replicas in a Cluster must acknowledge read or write operations.

**Replication Factor:** The total number of replica Nodes across a given Cluster. A Replication Factor of 1 means that the data will only exist on a single Node in the Cluster and this setup will not have any fault tolerance. The Replication Factor is set for each Keyspace. All replicas share equal priority; there are no primary or master replicas.

**Scylla Loader (scylla-loader):** A tool provided by Scylla for loading data into the database. It supports various formats including CSV, JSON, and Avro, and can parallelize data ingestion to improve performance.

**Scylla Manager:** An open-source web-based management tool for Scylla clusters. It provides monitoring, alerting, and management capabilities, making it easier to operate and maintain Scylla clusters.

**Data Model:** Scylla supports a wide column store model similar to Cassandra, allowing for efficient storage and retrieval of large amounts of data. The data model includes features like compression, compaction, and versioning to optimize storage usage and durability.

**CAP Theorem:** The CAP Theorem is a concept that states that a distributed database system can only have 2 of the 3: Consistency, Availability, and Partition Tolerance.

**Important Points:**

1. Scylla has a ring-type architecture
2. It’s a distributed, highly available, high performance, low maintenance, highly scalable NoSQL database
3. In Scylla all nodes are created equal, there are no master/slave nodes
4. Data is automatically distributed and replicated on the cluster according to the replication strategy
5. Scylla supports multiple data centers
6. Scylla transparently partitions data by using the hash values of keys

**Migration Points**

**1-Prepare Scylla for Kubernetes**

->ScyllaDB has a Kubernetes Operator that simplifies deploying and managing Scylla clusters within Kubernetes environments like EKS.

-->Download the Scylla Operator: Clone the Scylla Operator repository from GitHub: git clone https://github.com/scylladb/scylla-operator.git.

-->Deploy the Scylla Operator: Navigate to the cloned directory and deploy the operator using Helm or Kubernetes manifests provided in the repository.

#cd scylla-operator

#helm install scylla-operator./charts/operator

**2-Create a ScyllaCluster Custom Resource**

The Scylla Operator uses custom resources to manage Scylla clusters. You'll need to define a ScyllaCluster resource that specifies your desired Scylla configuration.

-->Define a ScyllaCluster YAML file: Create a YAML file named scyllacluster.yaml with the following content, adjusting parameters as necessary for your use case.

apiVersion: cassandra.datastax.com/v1beta1

kind: ScyllaCluster

metadata:

name: my-scylla-cluster

spec:

replicas: 3

storage:

class: s3

options:

s3.endpoint: "s3.amazonaws.com"

s3.access\_key: "<YOUR\_S3\_ACCESS\_KEY>"

s3.secret\_key: "<YOUR\_S3\_SECRET\_KEY>"

s3.bucket: "<YOUR\_S3\_BUCKET\_NAME>"

s3.path: "/scylla-backups/"

seedProvider:

class: ec2

options:

region: "<REGION>"

availabilityZone: "<AVAILABILITY\_ZONE>"

network:

broadcastAddress: "192.168.0.100"

broadcastRpcAddress: "192.168.0.100"

listenAddress: "192.168.0.100"

rpcAddress: "192.168.0.100"

**3-Migrate Data**

Note-->Before migrating data, ensure your new Scylla cluster is fully operational and accessible.

**Export Data from Old Cluster:** Use Scylla's native backup tools or third-party solutions to export your data from the old EC2-based Scylla cluster.

**Import Data into New Cluster:** Import the exported data into your newly deployed Scylla cluster on EKS. The method of importing will depend on how you exported the data but typically involves ***using Scylla's nodetool import command***or similar.

**4-How to Migrate Data**

**A-Export Data from the Old EC2-Based Scylla Cluster**

Using Scylla's Native Backup Tools

Scylla provides native tools like **nodetool for backups**. To export data, you can use the **sstableloader** tool if you have already dumped your data into **SSTables**. **If not, you might first need to dump your data using nodetool cfexport.**

**🡪** *Dumping Data*: Use nodetool cfexport to dump data from each table in your database. For example:

# nodetool cfexport keyspace\_name.table\_name > table\_data\_export.json

NOTE-> Repeat this for each table you want to export.

🡪 *Loading Data into SSTables*: Once you have the JSON dumps, you can load them into SSTables using sstableloader. First, create a directory structure that matches your Scylla schema:

# mkdir -p /path/to/sstables/keyspace\_name/table\_name

Then, use sstableloader to load the data:

# sstableloader /path/to/sstables/keyspace\_name/table\_name

**B-Import Data into the New Scylla Cluster**

Using Scylla's nodetool import Command

If you've loaded your data into SSTables, you can directly import it into your new Scylla cluster using nodetool import. *Assuming you have the SSTables ready in a directory structure matching your Scylla schema, you can run:*

#nodetool --host <new\_cluster\_ip> --port <new\_cluster\_port> import /path/to/sstables

Replace <new\_cluster\_ip> and <new\_cluster\_port> with the IP address and port of one of your EKS Scylla nodes.

ClickHouse:

ClickHouse is an open-source columnar database management system (DBMS) for online analytical processing (OLAP). It's optimized for analytical queries and high-performance data ingestion.

ClickHouse is known for its speed and scalability, capable of processing large volumes of data with low latency. It's commonly used for real-time analytics, data warehousing, and business intelligence applications.