**Apache Kafka**

* Getting large amount of data from one place to another; data moving
* High throughput **distributed messaging system**
* **ETL “Extract, Transform and Load”**
* Design **for high throughput**

**Messaging (traditional)**

* moving data from application to datastore
* Not fault tolerant (app)
* Broker’s job is to send message and can’t keep them around for **very long**

***Middleware Challenges:***

* ***Multi-write pattern*** *where ur application relies on code written somewhere who handles data load*
* ***Message broker pattern*** *slow consumers or unavailable store; difficult to scale out*

THIS IS NOT THE BETTER WAY!

SO KAFKA WAS CONSIDERED TO BE AS A VIABLE SOLUTION

**Apache kafka’s architecture**

* **can scale out**
* **producers and consumers** are application use to implement producing and consuming APIs
* Producers that send messages to the cluster
* Consumers retrieve messages
* **Topics** a collection or grouping of messages; and these were all stored and maintain on a **Broker**
* Broker is a software process “executable or daemon service” that runs on a machine; and has access resources to the machine like file system
* **cluster** a grouping of **multiple** Kafka **brokers**

**Kafka Components:**

* **Zookeeper** used for config info; maintaining list of topics
* **Broker** daemon program which store messages
* **Topics** where messages are organized

**Clients:**

* **Producers**
* **Consumers**

**Concepts:**

* **Partitions** splits the msg distrib on a single topic throughout multiple brokers
* **Replication Factor** number of copies of data over multiple brokers
* **Consumer group** grp of consumer that shares the same grp id and topic
* **Consumer offset** consumer monitoring; maintain current position of a consumer; marker; pointer to the last record that kafka has sent to a consumer
* **Consumer lag** metrics; higher the consumer lag the slower ur consumer is getting ur msgs

***Principles of Distributed System***

* collection of resources that have instructions to achieve a goal or function
* Consists of multiple worker or nodes
* Each node **communicates** with each other
* **Controller** is basically like a worker; checks **availability** and **health**
* **Quorum** existing relationship when two or more server instances in db mirroring session are connected to each other

**ADDITIONAL:**

**A communication tool in a DS is called consensus or gossip protocol and without it, it cannot operate**

**Zookeeper** is a centralized service for metadata; config info; health status; group membership; **ensemble** is like a cluster in zk

* Path: zookeeper/conf/zoo.cfg
* **dataDir**=**/var/lib/zookeeper** where the snapshots are stored
* **clientPort=2181**which clients connect
* **admin.serverPort-8081** port which zk app is running
* **server.1=zookeeper\_1:2888:3888** run zk in replicated mode

**running zk: ./zkServer.sh start-foreground**

**Kafka**

* **server.properties** kafka config
* path: kafka/config/server.properties

important properties:

Text, letter

Description automatically generated

run kafka: **./kafka-server-start.sh /usr/local/kafka/config/server.properties**

**How to Start of Kafka Cluster:**

**Useful Kafka Commands**

**Graphical user interface, text, application, email

Description automatically generated**

**UNDERSTANDING TOPICS, PARTITIONS AND BROKERS**

**cd /usr/local/bin/kafka/site-docs** contains documentation

**cd /usr/local/bin/kafka/libs** contains all dependencies in order to run

**cd /usr/local/bin/kafka/config** all config files

**cd /usr/local/bin/kafka/bin** contains all the programs to get Kafka up

**Topics**

* named feed or category of messages (prod/cons)
* logical entity
* Physically represented as log

*Logical:*

*Ordered sequence (by time); immutable facts as events*

*Each message has timestamp; referenceable identifier; payload (binary)*

**Message Offset** understand how consumers can do read messages at their own pace and process them independently

* **placeholder** last read message position; maintained by Kafka consumer

**Kafka is immune because it can retain messages that is configurable and known as the message retention policy**

**DEMO:**

**> INSTALL KAFKA ON UR MACHINE**

**> START THE MAIN COMPONENTS OF THE CLUSTER:**

**//**running the shell program

//test if zookeeper is running

//starting up Kafka

**./bin/kafka-topics.sh** will more detailed about options in topics

//create topic; passed zookeeper because there are multiple per instances

//to know more

//check producer

//creating; u can keep ur terminal open; type any message there; **producer**

//pulling the msgs from the broker(display output what u write in producer; **consumer**

**APACHE KAFKA AS A DISTIRIBUTED COMMIT LOG**

**Transaction or Commit logs**

* source of truth
* Physically stored and maintained
* Point of recovery
* Basis for replication and distribution

**Kafka Partitions**

* Each topic has one or more partition
* Scale; become fault tolerant

Create a topic: Single partition

Each partition must fit on the **same machine**

**Scalability of Kafka is based on partitions**

**Partitioning Trade-offs**

* **Making sure that** all of the brokers are working
* Ensure proper ZK capacity
* Each messages are totally ordered
* The more partitions the longer the **leader fail over-time**

**Achieving reliability and apache Kafka replication**

**No redundancy between nodes**

**DONT FORGET THE REPLICATION FACTOR**

**Rep. Factor**

* reliable work distribution
* Fault-tolerance
* Guarantees up to n-1 broker failure tolerance; 2 or 3 minimum
* Configured on a per-topic basis

//multiple replica sets

//viewing topic state

**DEMO:**

**//both producer and consumer are up to check some information**

**PRODUCING MESSAGES WITH KAKFKA PRODUCERS**

**> INSTALL INTELLIJ**

**> CREATE PROJECT**

**> MAVEN > NAME IT KAFKA SOMETHING**

**//under client, u see producer and consumer namespace and other stuff**

**Basic of Creating an Apache Kakfa Producer**

* u need to have object first
* Configuration items are basically **key value pairs**
* **bootstrap.servers** when we use Kafka shell program, we need to supply a **list of brokers** for the **producer** to **connect** to
* **key and value serializer** message content is encoded as binary; compression and optimization

**//creating Kafka producer; starting sending messages with its default settings**

**ProducerConfig** when object is created, its properties were used

**Creating and Preparing Apache Kafka Records**

* **ProducerRecord** a critical class that represents what all be published by **Kakfa Producer;** basic and straightforward and has two values —**topic and value**
* **Topic** where records are destined
* **Value** contents the message need to be serialized using the specific serializer
* Can only s**end ProducerRecords** that **match** the **key and value** serializers types

ProducerRecords: Optional Properties

**Partition** within the topic to send PR

**Timestamp** unix ts applied to the record

**LogAppendTime** broker-set timestamps used when msg is appended to **commit log**

**Key** *(if present )*determine how and which within the topic will Kafka producer will be sending message; **avoid** leaving it blank or null

**Process of Sending messages**

**//wrap the call with: try cHAHUTOatch block**

**> when calling send method, the producer will reach out to the cluster using bootstrap.servers list to discover membership**

**> producer can now dispatch the producer record onto an in-memory queue-like data structure called “RecordAccumulator”** that gives it ability to micro-batch to sent at high volumes and high frequencies

***Micro-batching***

* *scale, efficiency is everything*
* *Sending (prod)*
* *Writing (blocker)*
* *Reading (cons)*

**Messaging Buffering and Micro-batching**

* **batch.size** a value represents the **maximum number** of bytes that can be buffered each r**ecord** batch
* **buffer.memory** config setting that establishes a ceiling or threshold for **how much memory** can be used
* [**linger.ms**](http://linger.ms)thatrepresents the number of ms an unfurl buffer should **wait**
* **RecordMetadata** contains info about the records that were successfully and unsuccessfully received

**Delivery Guarantees**

**Ordering Guarantees**

**DEMO: CREATING AND RUNNING AN APACHE KAFKA PRODUCER APP IN JAVA**

**CONSUMING MESSAGES WITH KAFKA CONSUMERS AND CONSUMER GROUPS**

* identical with producers
* U need to have **key and value** **deserializer** because u are reading messages here this time

//subscribing to topics; **subscribe** represent list of topics; when adding new topic to subscription it will not have new BUT It will OVERRIDE the existing topic

//unsubscribing to topics; not just in topic BUT the ENTIRE EXISTING TOPICS

**Comparing subscribe and assign APIs**

* not overloading the assigning
* common to them is they both taking lists

**The Pool Loop**

*With consumer context:*

* *primary function of Kafka* ***consumer***
* *Continuously polling the brokers for data*
* Opens network resources
* Single-threaded operation

**Kafka-producer-perf-test shell program** creating lots of messages; verify the producer’s performance

**Object** serves as the source of truth for any and all details; important role

**Fetcher** serves as the responsible object for most of the communication between consumer and cluster

**Timeout** represents the number of ms the time client spending

**Offset** that enables consumers to operate independently

* establish what it has and hasn’t; **last committed offset** represent the **last record** that the **consumer** that has to process
* Consumer may have multiple offset
* Enable **auto-commit** basically giving Kafka the responsibility to manage when current position offset are upgraded to full

**Read != committed**

**Offset commit is configurable**

Kafka **stored** committed **offsets** in a special topic called **\_consumer\_offsets**

**commitSync** when u want precise control over when to consider a record truly processed

* Synchronous
* Retries until succeeds or unrecoverable error **retry.**[backoff.ms](http://backoff.ms) **(default: 100 ms)**

**commitAsync -** async; non-blocking but non-deterministic

* no retries
* But have **callback** option; u can determine the status of commit

*Where offset management occurs when poll method timeout*

*Consistency control “when a message is ready to commit”*

**Scaling-out with Consumers Grp**

*add more with its respective practice*

* Collection of individual processes
* [**group.id**](http://group.id)setting
* Sharing the msg consumption and processing load

**Consumer Group Rebalancing**

* needs to know what offset need to starts from
* Every balances available Consumers to partitions; **group coordinator**

**Consumer Configuration**

* consumer performance and efficiency; **fetch.min.bytes -** batch size setting to the producer
* [**max.fetch.wait.ms**](http://max.fetch.wait.ms)establishes the amount of time to wait
* **max.partition.fetch.bytes** ensure poll isn’t retrieving data more than your process
* **max.poll.records** establish the maximum allowed per poll cycle

Graphical user interface

Description automatically generated with medium confidence

**EXPLORING THE KAFKA ECOSYSTEM AND ITS FUTURE**