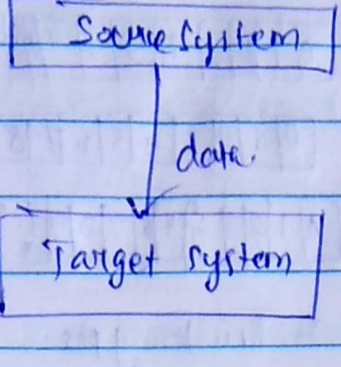
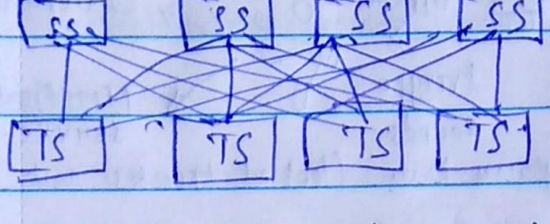
Apache Kafka.

lets take a scenario we need to exchange a data between source to target system.



simple at first

After a while things get complicated,



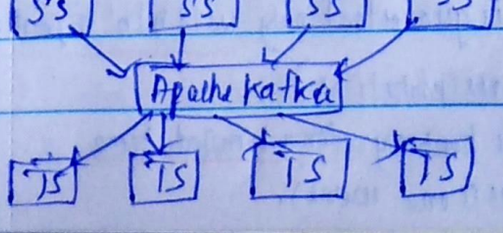
→ So if we have 4 SS & 6 TS we need to write 24 integrations.

→ Each integration comes with difficulties around.

* Protocol ⇒ how data is transported. (TCP, HTTP, REST, FTP..)
* Data format - how data is parsed (Binary, csv, JSON, Avro...)
* Data schema & evolution - how data is shaped & may change.

→ Each source system will have an increased load from the connection.

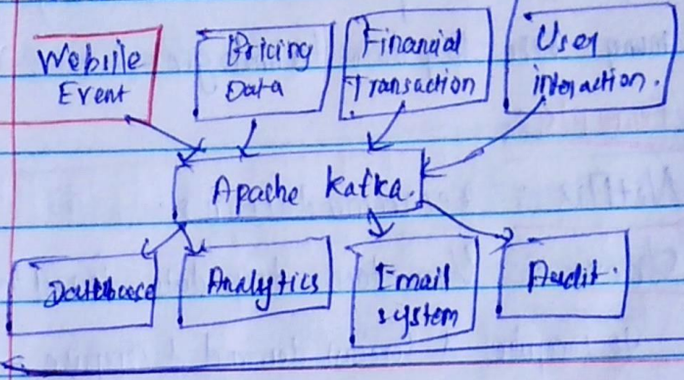
Decoupling of data streams & systems:



Example for data streams:

→ Website events.
→ Pricing data.
→ Financial transaction
→ User interaction.
→ etc.

Once we have data in kafka, we can put it in any system we like.



Why Apache Kafka:

→ Created by LinkedIn, Open sourced project mainly maintained by Confluent.
→ Distributed, resilient architecture, fault tolerant.
→ Horizontal scalability.
* can scale to 100 brokers.
* can scale to millions of message per second.
→ High performance (latency of less than 10ms) - real time.
→ Used by 2000+ firms, 35% of the Fortune 500.

Apache Kafka Usecases:

- Messaging System
- Activity Tracking
- Gather metrics from many different locations.
- Applications logs gathering
- Stream Processing (with the kafka Stream API or Spark for example).
- Decoupling of system dependencies.
- Integration with spark flink storm Hadoop and many other Big data technologies.

For Example:-

- Netflix: recommendations.
- Uber: User taxi, trip data in real time to compute & forecast demand & compute surge pricing in real time.
- LinkedIn: Prevent spam, collect user interactions to make better connection.
- Kafka is only used as a transportation mechanism!

Kafka Theory:

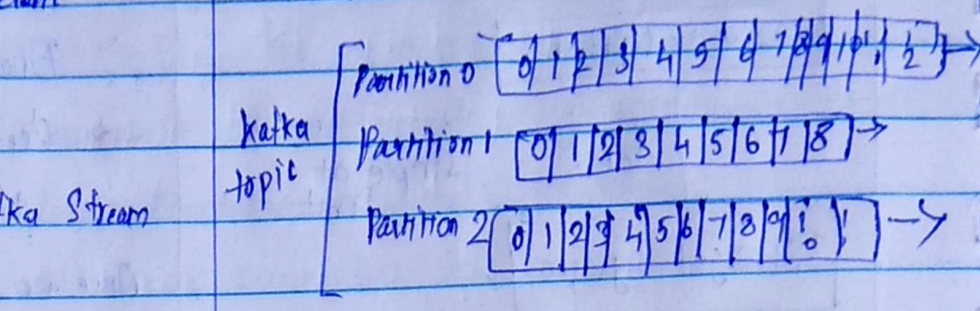
Topics, Partitions & offsets:

Topics: a particular stream of data

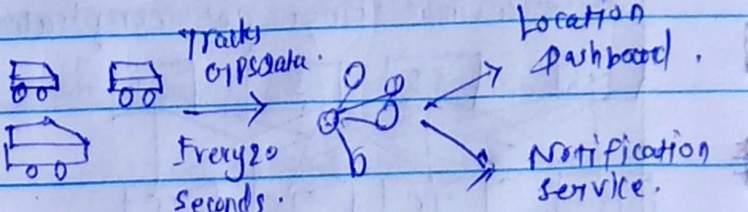
- Similar to a table in a database (without all the constraints)
- You can have as many topics as you want
- A topics is identified by its name.

Topics are split in partitions.

- Each partition is ordered.
- Each message within a partition gets an incremental id called offset



Topic Example: truck_gps.



- Say you have fleet of trucks, each truck reports its GPS position to kafka.
- You can have a topic truck_gps that contains the position of all trucks.
- Each truck will send a message to kafka every 20 seconds, each message will contain the truck ID and the truck position (latitude & longitude).
- We choose to create that topic with 10 partitions.

Offset:

- Offset only have a meaning for a specific partitions.

  Eg: offset 3 in partition 0 does not represent the same data as offset 3 in partition 1.
- Order is guaranteed only within a partition (not across partitions).
- Data is kept only for a limited time (Default is one week).

⇒ Once the data is written to a partition, it can't be changed (immutability)

⇒ Data is assigned randomly to a partition unless a key is provided (more).

Brokers:

⇒ It a holder for Topics

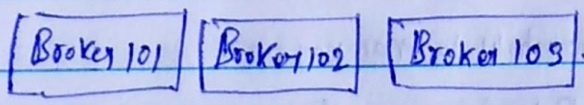
⇒ A kafka cluster is composed of multiple brokers (servers)

⇒ Each broker is identified with its ID (integer).

⇒ Each broker contains certain topic partitions.

⇒ After connecting to any broker (called a bootstrap broker) you will be connected to the entire cluster.

⇒ Big cluster have over 100 brokers.

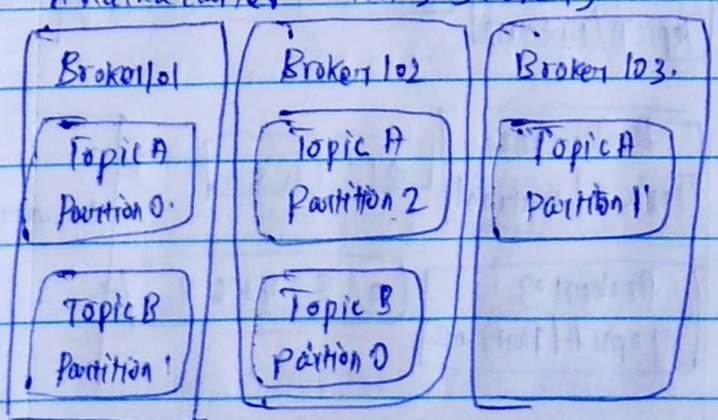


Brokers & Topics:

⇒ Example of Topic-A with 3 partitions

⇒ Example of Topic-B with 2 partitions.

A kafka cluster with 3 Brokers.



⇒ Data is distributed and Broker 103 does not have any Topic B data.

Topic replication factor

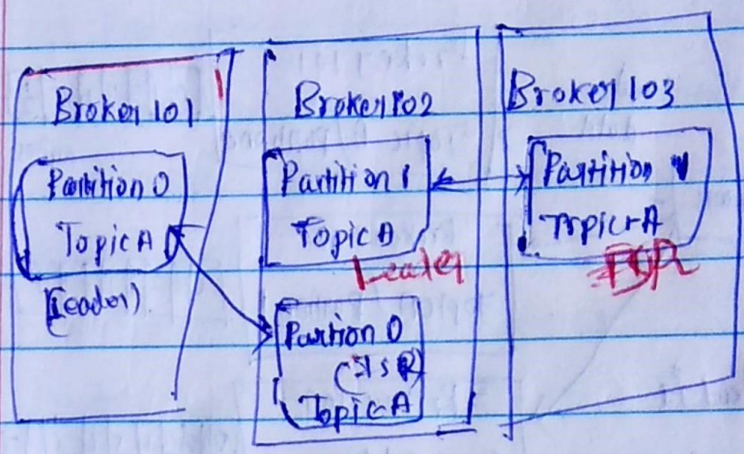
⇒ Kafka is a distributed system. In distributed system, we have replication.

⇒ If things goes down application still works.

⇒ Topics should have a replication factor > 1 (usually btw 2 & 3)

⇒ This way if a broker is down, another broker can serve the data.

⇒ Example: Topic A with 2 partition & replication factor of 2.



Example: we lost Broker 102.

Result: Broker 101 & 103 can still serve the data.

Concept of Leader for a Partition:-

⇒ At any time only one broker can be a leader for a given partition.

⇒ Only that leader can receive and serve data for a partition.

⇒ The other brokers will synchronize the data.

⇒ Therefore each partition has one leader and multiple ISR (in-sync replica).

Producers:-

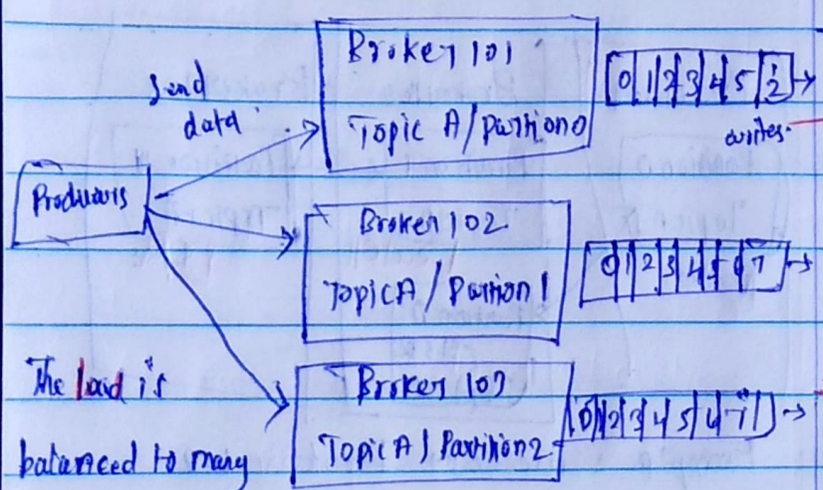
→ How do we get data in kafka? That is the job of producers.

→ Producer does the load balancing in round robin fashion.

→ Producers write data to topics (which is made of partitions)

→ Producers automatically know to which broker and partition to write to.

→ In case of Broker failures, Producers will automatically recover.



Producers can choose to receive acknowledgment of data writes.

acks=0 : Producers won't wait for acknowledgment (possible data loss).

acks=1 : Producers will wait for leader acknowledgement (limited data loss)

acks=all Leader + replicas acknowledgment (No data loss).

Messages key:

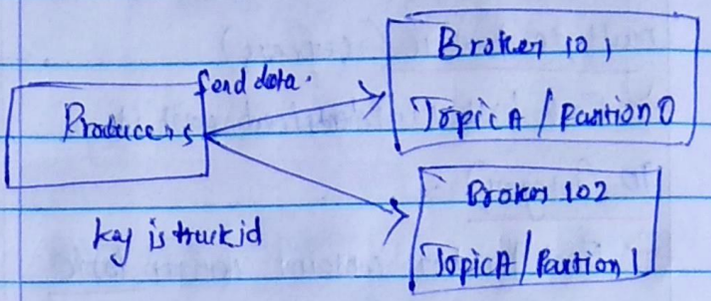
Producers can choose to send a key with message. (string, number etc...)

→ If key=null, data is sent round robin (Broker 101 then 102 then 103).

→ If key is sent, then all message for that key will go to same partition.

→ A key is basically sent if you need message ordering for a specific field.

Ex: truck_id



Consumers:-

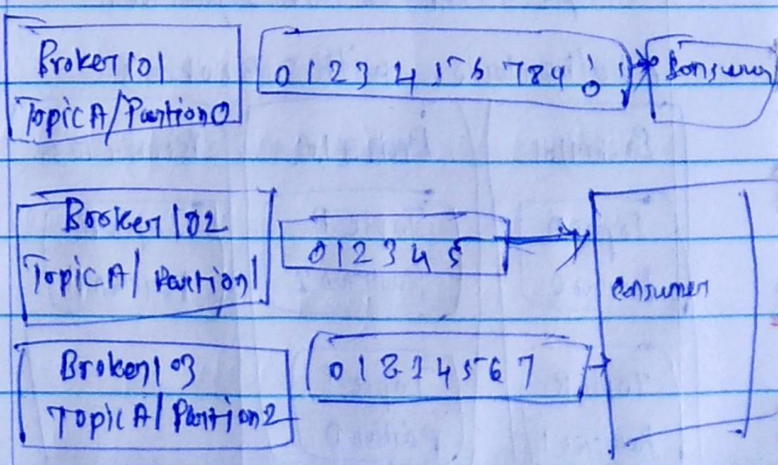
→ Consumers read data from a topic (identified by name)

→ Consumer know which broker to read from.

→ In case of broker failures, consumers know how to recover.

→ Data is read in order within each partition.

→ Across partition offset order not matters.



Consumer Groups:

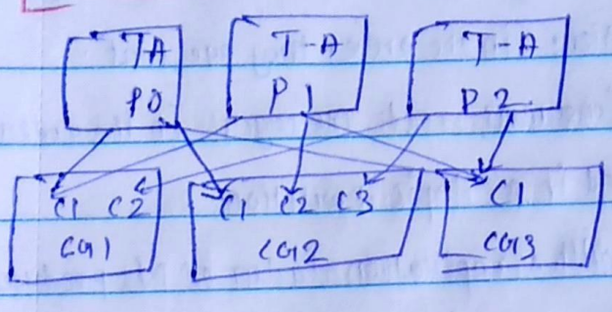
→ Consumers read data in consumer group.

→ Each consumer within a group read from exclusive partitions.

→ If you have more consumers than partitions, some consumers will be inactive.

CG - Consumer group.
C - consumer.
T - Topic
P - Partion.

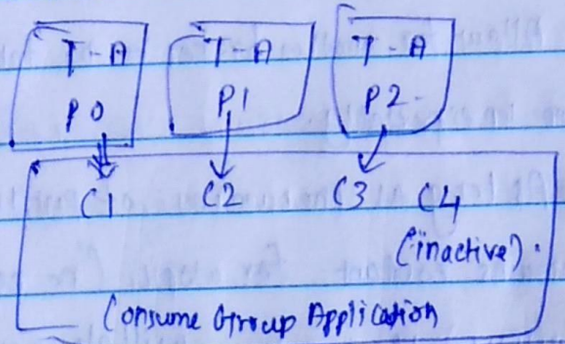
⇒ Shortform



Consumers will automatically use a Group Coordinator and a Consumer Coordinator to assign a consumer to a partition.

What if too many consumers?

If you have more consumers than partitions, some consumer will be inactive.



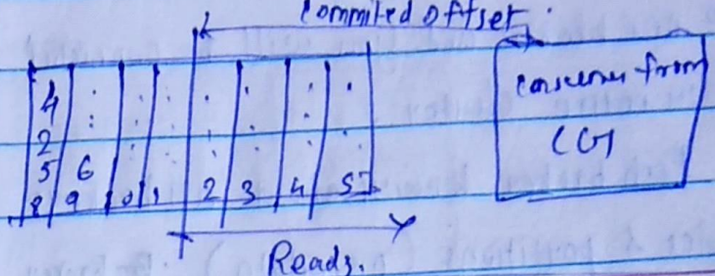
Consumer Offets:

→ Kafka stores the offset at which a consumer group has been reading

→ The offset commited live in a kafka topic named __consumer_offsets.

→ When a consumer in a group has processed data received from kafka it should be commiting the offset.

→ If a consumer dies, it will be able to read back from where it left off, thanks to the commited consumer offsets.



Delivery sementics for consumer:-

→ Commiting offset implies it.

→ Consumer choose when to commit offsets.

→ There are 3 delivery semantics.

At most once:

→ offsets are commited as soon as the message is received.

→ If the proces goes wrong, the message will be lost (it won't read again)

At least once:

→ commited after message is processed.

→ Process goes wrong message will be read again.

⇒ This can result in duplicate processing of message. Make sure your processing is idempotent. (processing again the message wont impact your system).

Exactly once:-

→ can be achieved for kafka => kafka workflows using kafka streams API.

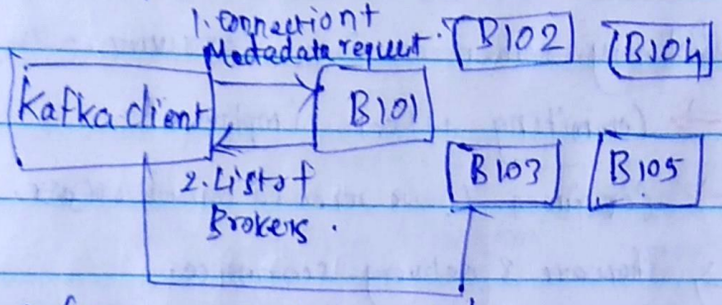
→ For kafka → External system workflows use an idempotent consumer.

Kafka Broker Discovery :-

Every kafka broker is also called a "bootstrap server".

That means that you only need to connect to one broker and you will be connected to the entire cluster.

Each broker knows about all brokers topics & partitions (metadata). B- Broker

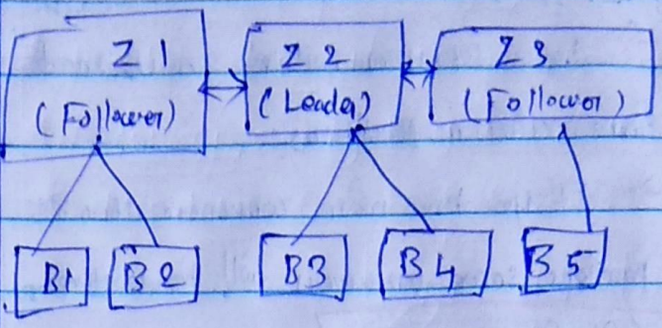


Zookeeper :-

* Manages brokers (keeps a list of them).
* Helps in performing leader election for partitions.
* Sends notification to kafka in case of changes (e.g: new topic, broker dies, broker comes up, delete topics, etc....).
* Kafka can't work without Zookeeper.
* By design, operates with an odd number of servers (3, 5, 7) (handle write).
* Has a leader, the rest of the servers are followers (handle reads)
* Zookeeper does not store consumer offsets with kafka > v0.10.

Z - Zookeeper Server.

B - Kafka Broker



Kafka Guarantees :-

→ Messages are appended to a topic partition in the order they are sent.

→ Consumers read messages in the order stored in a topic partition.

→ With a replication factor of N, producers and consumers can tolerate up to N-1 broker being down.

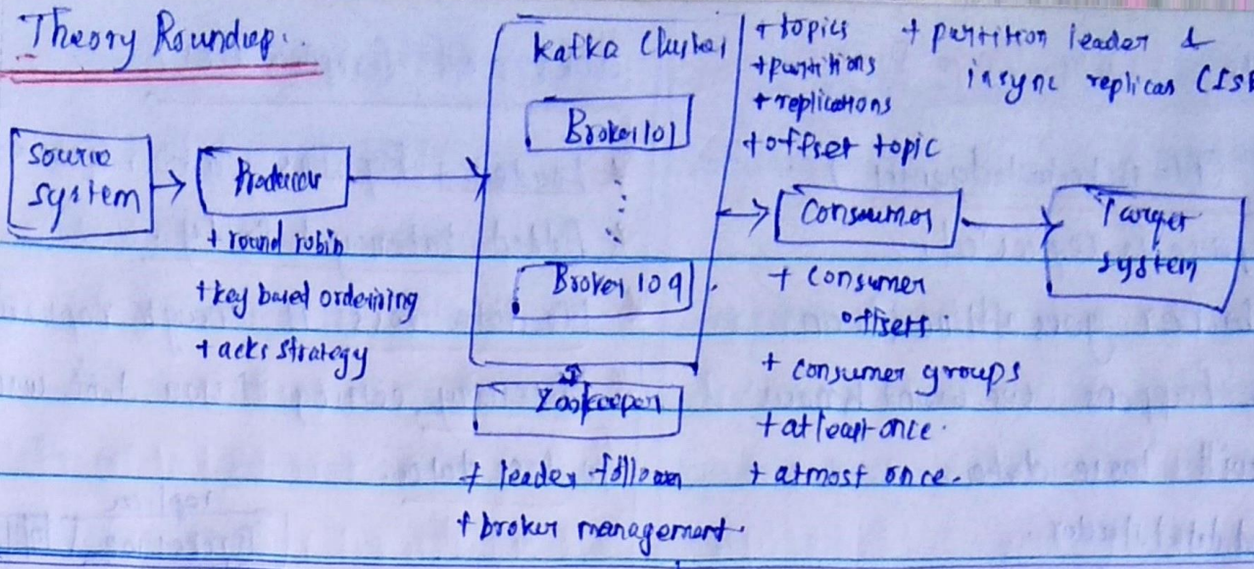
→ This is why a replication factor of 3 is good idea.

→ Allows for one broker to be taken down for maintenance.

→ Allow for another broker to be taken down unexpectedly.

* As long as the number of partitions remains constant for a topic, (no new partitions) the same key will always go to the same partition.

Theory Roundup:



Command for starting zookeeper:

Zookeeper-server-start config/
zookeeper.properties.

↑

For starting Zookeeper.

Starting Kafka:

kafka-server-start config/server.
properties.

Creating a topic:

kafka-topics --zookeeper 0.0.0.0:2181 --topic first_topic
--create --partition 3 --replication-factor 1

Please refer the word document for more commands.

Providing a key in producer.
always goes to the same partition

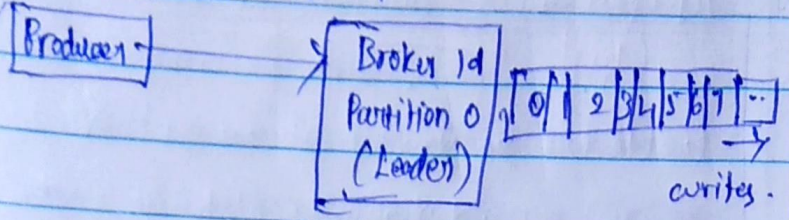
Consumer in consumer group
automatically rebalanced to different
partitions in topic.

Producers Acks Deep Dive:

acks = 0 (No Acknowledgement):

* No response is requested.
* If the broker goes offline or an exception happens we won't know and we will loose data.

Send data to leader.

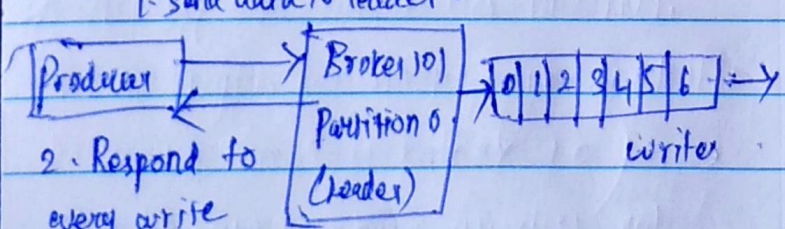


* Useful for data when it's okay to potentially lose messages.
  - Metric collection
  - Log collection.
* Good in performance

ack=1 (Leader acks)

* Leader response is requested, but replication is not gaurantee (happens in the background)
* If an ack is not received, the producer may retry.

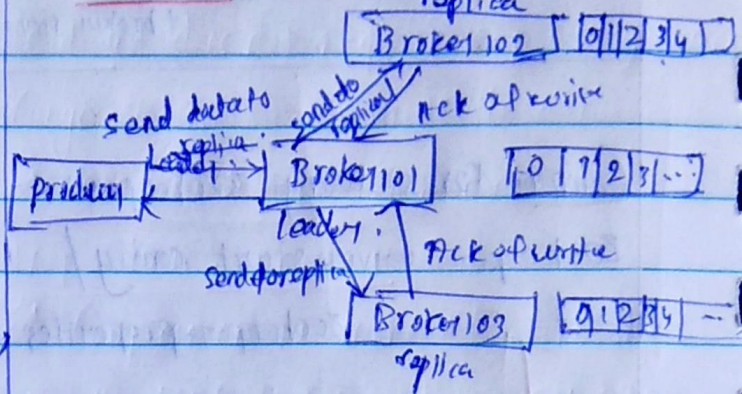
1. Send data to leader.



* If the broker goes offline but replicas haven't replicated the data yet. we have a data loss.

acks = all (replica acks):

* Leader + Replicas ack requested.
* Added latency & Safety.
* No data loss if enough replicas.
* Necessary setting if you don't want to loss data.



* Acks = all must be used in conjunction with min.insync.replicas
* Min.insync.replica can be set at the broker (or) topic level (override).
* min.insync.replica = 2 implies that at least 2 brokers that are ISR (including leader) must respond that they have the data
* That means if you use replication.factor = 3 min.insync = 2, acks = all you can only tolerate one broker going down otherwise producer will receive an exception on send.

Producer Retries:

* In case of transient failures, developers are expected to handle exceptions, otherwise the data will be lost.

* Example of transient failure:
  → Not Enough Replicas Exception.
* There is a retries setting
  → default to 0 for kafka <= 2.0
  → default to 2147483647 for kafka >= 2.1
* The retry.backoff.ms setting is by default 100 ms.

Producer Timeouts:

* if retries > 0, for example, retries = 2147483647;
* the producer won't try the request for ever, it's bounded by a timeout
* For this, you can set an intuitive Producer Timeout (kip-91 - kafka 2.1).
* delivery.timeout.ms = 120000 ms = 2min
* Records will be failed if they can't be acknowledged in delivery.timeout.ms.

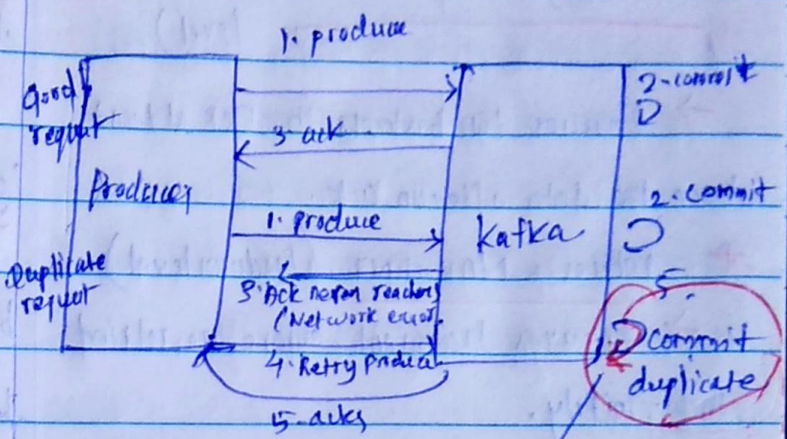
Producer Retries: Warning:

* In case of retries, there is a chance that message will be sent be sent out of order (if a batch has failed to be sent)
* For this, you can set the setting while controls how many produce request can be made in parallel.
  max.in.flight.request.per.connection
* Default: 5
* set it to 1 if you need to ensure ordering.

* In kafka >= 1.0.0 there's better solution with idempotent producers.

Idempotent Producers:

Problem: The producer can introduce duplicate messages in kafka due to network errors.



where as in idempontent producer, won't introduce duplicates on network error.

* It detect the duplicate, Don't commit twice.
* In kafka >= 0.11 we can define Idempotent producer.
* It provides stable & safe pipeline
* It comes with:
  * retries = Integer.MAX_VALUE (2^31-1)
  * max.in.flight.requests = 1
  * max.in.flight.requests = 5
* These settings are applied automatically after your producer has started if you don't set them ~~automatic~~ manually.

  properties.set("enable.idempotence", "true");

Safe Producer summary:-

kafka (<0.11)

* acks = all (producer level).
  → Ensures data is properly replicated before an ack is received.
* min.insync.replicas = 2 (broker/topic level)
  → Ensures two brokers in ISR at least have the data after an ack.
* retries = MAX_INT (Producer level).
  → Ensures transient errors are retried indefinitely.
* max.in.flight.requests.per.connection = 1 (producer level).
  → Ensure only one request is tried at any time, preventing message reordering in case of retries.

kafka >= 0.11

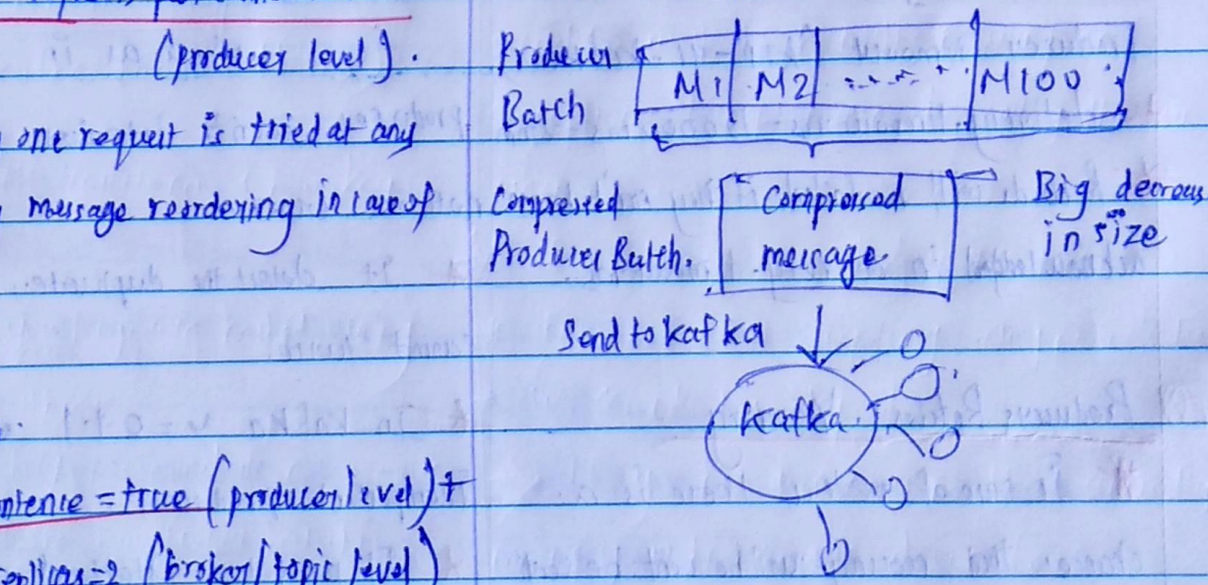
→ enable.idempotence = true (producer level) + min.insync.replicas = 2 (broker/topic level)

* implies (acks = all, retries = MAX_INT, max.in.flight.requests.per.connection = 1 if kafka 0.11 (or 5 if kafka >= 1.0))
* while keeping ordering guarantees and improving performance.

→ Running a safe producer might impact throughput & latency always test for your use case.

Message Compression:-

* Producer usually send data that is text based for example with JSON data
* In this case, it is important to apply compression to the producer.
* Compression is enabled at the producer level and does not require any configuration change in the Brokers or in the consumers.
* compression.type can be none (default) 'gzip', 'lz4', 'snappy'
* Compression is more effective the bigger the batch of message being sent to kafka.



Advantage:-

→ Much smaller producer request size (compression ratio upto 4x !)
→ Faster to transfer data over the n/w less latency
→ Better throughput
→ Better disk utilisation in kafka (stored messages on disk are smaller).

Disadvantage:

→ Producer must commit some CPU cycles to compression.

→ Consumer must commit some CPU cycle to decompression.

Overall:

→ Consider testing snappy (or) lz4 for optimal speed / compression ratio

Recommendation:

→ Test all algorithm and decide

→ Use it in production & especially if you have high throughput

→ Consider tweaking linger.ms & batch.size to have bigger batches, therefore more compression & higher throughput

Producer Batching

→ By default, kafka tries to send record as soon as possible

→ It will have up to 5 request in flight, meaning up to 5 messages individually sent at the same time.

→ After this, if more messages have to be sent while others are in flight kafka is smart & will start batching them while they wait to send them all at once.

→ This smart batching allow kafka to increase throughput while maintaining very low latency.

→ Batches have higher compression ratio so better efficiency.

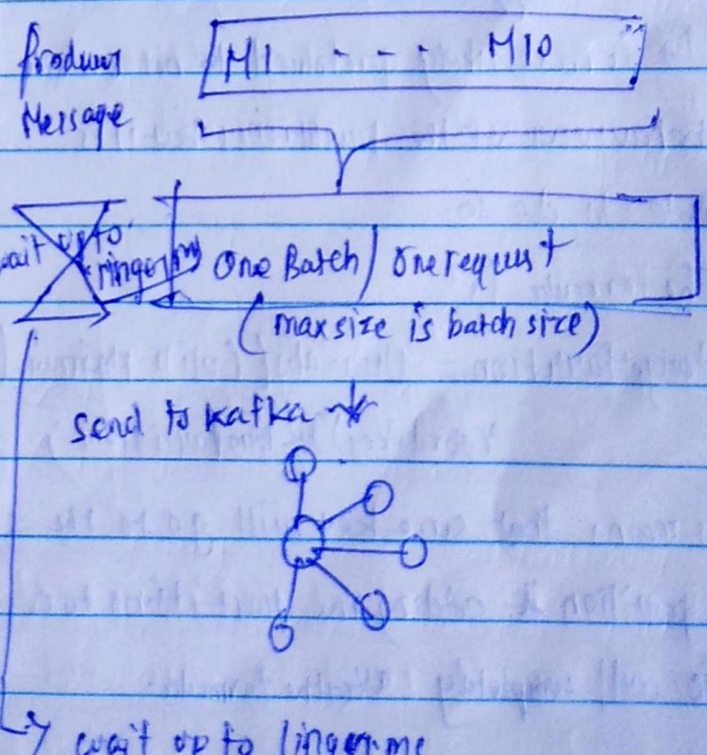
Linger.ms & Batch.size.

→ Linger.ms: Number of millisecond a producer is willing to wait before sending a batch out (default 0).

→ By introducing some lag (for example linger.ms=5) we increase the chance of messages being sent together in a batch.

→ So at the expense of introducing a small delay, we can increase Throughput compression & efficiency of our producer.

→ If a batch is full before the end of the linger.ms period, it will be sent to kafka right away.



Batch size: (batch.size)

→ Maximum number of bytes that will be included in a batch. The default is 16KB.

→ Increasing a batch size to something like 32KB (or 64KB can help increasing the compression, throughput efficiency of requests.

→ Any message that is bigger than the batch size will not be batched.

→ A batch is allocated per partition, so make sure that you don't set it to a number that's too high otherwise you'll run waste memory.

→ (You can monitor the average batch size metric using Kafka Producer Metrics).

Producer Default Partitioner & how keys are hashed.

→ By default, your keys are hashed using the murmur2 algorithm.

→ It is most likely preferred to not override the behaviour of the partitioner but it is possible to do so.

→ The formula is

targetPartition = Utils.abs(Utils.murmur(record.key())) % numPartitions;

→ This means that same key will go to the same partition & adding new partitions to a topic will completely alter the formula.

Max.block.ms & Buffer.memory.

→ If the producer produces faster than the broker can take, the record will be buffered in memory.

→ buffer.memory = 33554432 (32mb) the size of the send buffer.

→ That buffer will fill up over time & fill back down when the throughput to the broker increase.

→ If that buffer is full (all 32mb), then the .send() method will start to block (won't return right away).

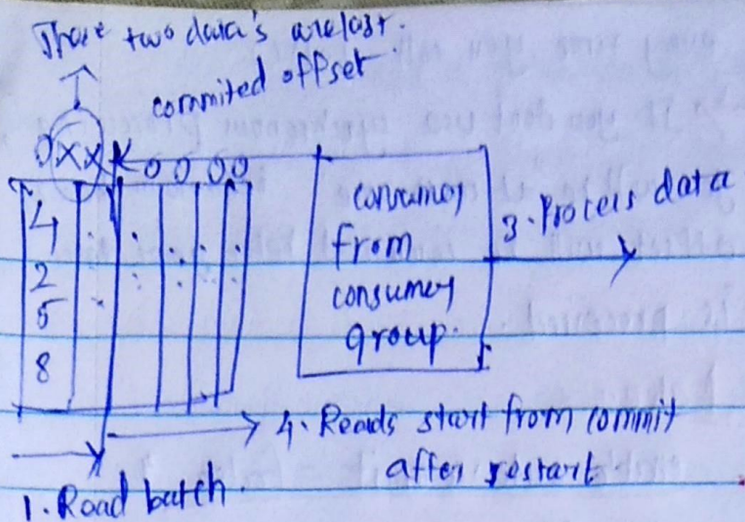
→ max.block.ms = 60000: the time the .send() will block until throwing an exception. Exception are basically thrown when.

* The producer has filled up its buffer.
* The broker is not accepting any new data.
* 60 seconds has elapsed

→ If you hit an exception that usually means your brokers are down (or) overloaded as they can't respond to requests.

Delivery Semantics:-

At most once: offsets are commited as soon as the message batch is received. If the processing goes wrong, the message will be lost (it won't be read again).



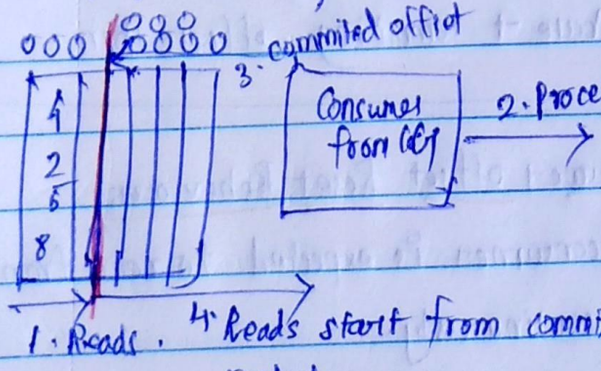
At least Once

→ offsets are commited after the message is processed.

→ If the processing goes wrong, the message will be read again.

→ This can result in duplicate processing of messages.

→ Make sure your processing is idempotent. (Processing again won't impact the system).



Exactly once: Can be achieved for kafka => kafka workflows using kafka streams API.
For kafka => Sink workflow use an idempotent consumer.

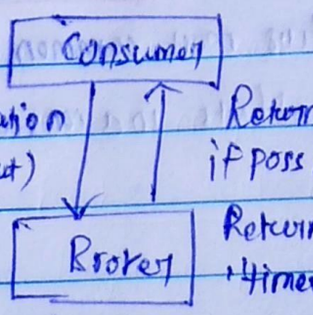
Bottom Line:

For most application you should use at least once processing & ensure transformation / Processing are idempontent.

Consumer Poll Behaviour:

⇒ Kafka consumer have a 'poll' model, while many other messaging bus in enterprises have a push model.

→ This allow consumers to control where in the log they want to consume, how fast & gives them ability to replay events.



Consumer Poll Behaviour:

⇒ Fetch.min.bytes (default 1):

* Controls how much data you want to pull at least on each request.
* Helps improving the throughput & decreasing request number.
* At the cost of latency.

⇒ Max.poll.records (default 500):

* controls how many records to receive per poll request.
* Increase if your message are very small & have a lot of available RAM.

Max.partitions.fetch.bytes (default 1MB):

* Maximum data returned by the broker per partition.
* If you read from 100 partitions, you will need a lot of memory (RAM).

Fetch.max.bytes (default 50MB):-

* Maximum data returned for each fetch request (covers multiple partitions).
* The consumer performs multiple fetches in parallel.
* Change these settings only if your consumer maxed out on throughput already.

Consumer offset Commit Strategies:-

There are two most common patterns for commiting offsets in a consumer application.

2 strategies:-

(easy) enable.auto.commit = true & synchronous processing of batches.

(medium) enable.auto.commit = false & manual commit of offsets.

True

```
while (true) {
    -- consumer.poll(--);
    doSomethingSynchronous(..);
}
```

→ with auto commit, offset will be commited automatically, for you at regular interval. (auto.commit.interval.ms = 5000 by default) every time you call .poll()

→ If you don't use synchronous processing, you will be "at most once" behaviour because offset will be commited before your data is processed.

False:-

enable.auto.commit = false

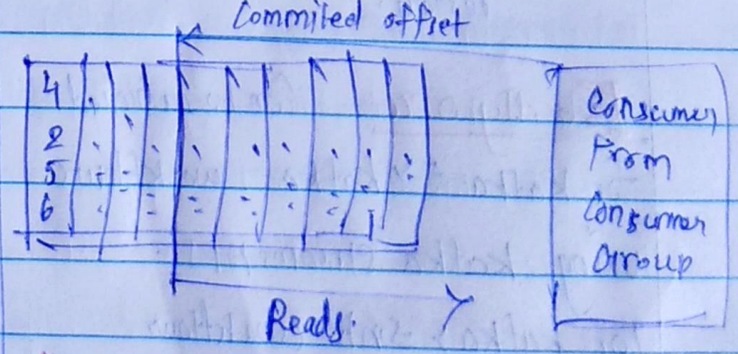
```
while (true) {
    batch += consumer.poll(..);
    if isReady(batch) {
        doSomethingSynchronous(batch);
        consumer.commitSync();
    }
}
```

→ you control when you commit offsets & what condition for commiting them.

→ Example: accumulating records into buffer and then flushing the buffer to a database + commiting offsets then.

Consumer offset Reset Behaviour:-

→ A consumer is expected to read from a log continuously.



→ But if your application has a bug, your consumer can be down.

→ If kafka has a retention of 7 days & your consumer is down for more than 7 days, the offsets are invalid.

auto.offset.reset values:

latest → will read from the end of the log

earliest → will read from the start of the log

none → will throw an exception if no offset is found.

Additionally, consumer offsets can be lost:

→ If consumer has not read new data in 1 day (kafka < 2.0)

→ If consumer has not read new data in 7 days (kafka >= 2.0).

This can be controlled by the broker setting offset.retention.minutes.

Replaying data for a consumer group:

→ Take all the consumers from a specific group down.

→ Use 'kafka-consumer-group' command to set offset to what you want.

→ Restart consumers.

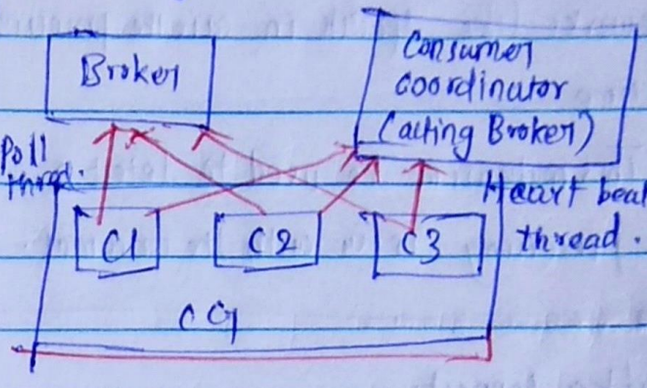
Bottom line:

→ Set proper data retention period & offset retention period.

→ Ensure the auto offset reset behavior is the one you expect/want.

→ Use replay capability in case of unexpected behaviour.

Controlling Consumer Liveliness:



→ Consumers in a group talk to a Consumer Group Coordinator.

→ To detect consumers that are down, there is a "heartbeat mechanism" and a "poll mechanism".

→ To avoid issues, consumers are encouraged to process data fast & poll often.

Consumer Heartbeat thread

session.timeout.ms (10 sec):

→ Heartbeats are sent periodically to the broker.

→ If no heartbeat is sent during that period, the consumer is considered dead.

→ Set even lower to faster consumer rebalances.

Heartbeat.interval.ms (3 sec)

→ How often to send heartbeats.

→ Usually set to 1/3rd of session.timeout.ms.

This mechanism is used to detect a consumer application being down.

Consumer Poll Thread:-

max.poll.intravelors (5 min):

→ Maximum amount of time b/w two .poll() calls before declaring the consumer dead.

→ This particullary relevant for BigData frameworks like Spark in case the processin take time.

→ This mechanism is used to detect a data processing issue with the consumer.

Kafka connect:-

→ Code & Connectors re-use.

CQRS - Common Query Responsibility Segregation.