Kafka Notes  -

--------------

Website Events  - Depending on kind of website

                             Tweets - Tweet

                             Post - Facebook

                             Products Addition - E-commerce Portal

                             Views, Clicks, Engagement Time

Pricing data -  Taxi Prices

                        Product Prices

                        RTB - campaign prices

                        Financial Exchange - Stock prices

                        Cryptocurrency Exchange - Prices

Financial Transactions - Payments on e-commerce websites

User Interactions - Like, Subscribe, Follow , Addition/Removal from shopping Cart

                              other form of User Actions

Decoupling -

--------------

Protocol - Common protocol - Preferably long poll over http .. 😀😀

Data Format - Custom Serializer

Schema Evolution - schema may change - schema registry - Kafka Takes Binary Data 😀😀

Increased Load on Individual Connections

Kafka -Distributed queue

Source System  ===> Producers

Consumers ===> Target System

Role of zookeeper for kafka - ❤❤❤

Zookeeper - 1)manages brokers keeps a list of them.

                     2)helps in performing leader election for partitions

                     3)Zookeeper send notifications to Kafka..

                         new topic, broker dies, broker comes up, deletes topic (Keeps a watch on each znode corresponding to topic, broker)

                     4)Design operates with odd number of servers - 3,5,7

                     5)Has a leader (handle writes) rest of the servers are followers(handle reads)

                     6)Does not store consumer offsets with Kafka > v0.10

Zookeeper connections with kafka brokers.

Some zookeeper servers connect with some kafka brokers, write requests such as topic addition may go to a follower and propagated to leader where znode is added and replication occurs. ❤❤

Kafka client connect to needed bootstrap broker

Delivery Semantics for consumers

Atmost Once - Messages can be sent again if not received by consumer

Exactly Once  - Messages sent only once - Kafka  ==> kafka Workflows like streams API 💙💙

Atleast Once Usually preferred

1`

Consumer Offsets  -

Two consumers in a consumer group trying to read topic concurrently.

Reads and commits are atomic, two consumers in a group can't commit same offset.

Offsets commited in a topic - consumer offsets

Once consumer in a group has commited offset another consumer in a group reads ==> might read from a different broker.. information sync across brokers.. ??

Do consumers in a group read sequentially.. once consumer offset has been committed..??🧡🧡

Consumers in a group == read data from exclusive partitions

Uses group co-ordinator and consumer co-ordinator to bind consumers to partition, there may be multiple partitions assigned to a consumer. 💚💚

Distributed Systems - Kafka Partition Data replicated .... Just Like File Blocks in HDFS ... Different color codes in parition different sequence of messages.

C1 - P1 , P2  - Message sequence splitted across partition.

C2  - P3      -  Possibility that another consumer may read advance sequence of messages. 💖💖 ??

C3 -  P4, P5

Partition One Leader and Corresponding InSync Replication --- >

One partition will receive data and partitions **(In Sync Replicas)** in other brokers will synchronize. 💛💛 😀😀

Can a partition have multiple leaders and not all replicas be insync ?? 😃😃

Anonymous class -- New Callback() ( On completion method override )  being passed as a function argument 😄😄

Using reflection in binding String Serilaizer .get Class .getName()  key.serializer, Value.Serializer in Producer Properties 😀😀

@Nullable Annotation @NotNull Annotation

in Producer record constructor doing casting on NULL like (Long)null, (Integer)null) 😀😀

Calling another constructors using this() in Producer Record Constructor

Consumer Demo --->

using fixed size in list in consumers.subscribe(Arrays.asList(topic))

Look at callback code later on.. ??

producer.send code later on.. ??

Consumer demo - Countdown latch latch.await.. how to handle Interrupted Exception thrown by Latch.await in general. 🤑🤑

//waiting up to the specified wait time - LinkedBLockingQueue

Abstract Processor of Twitter - HosebirdMessageProcessor

queue.offer(msg, offerTimeoutMillis, TimeUnit.***MILLISECONDS***); 😀😀😃😃

for Blocking queue default timeout value -

**public** **static** **final** **long** ***DEFAULT\_OFFER\_TIMEOUT\_MILLIS*** = 500; 🤑🤑🤑

For consumption uses take api unless element becomes available as per twitter documentation.

//waits until element becomes available

E take() throws InterruptedException;

Implementation in twitter producer changes with poll with a timeout 5 seconds.

/\*\*

     \* Creates a {@code LinkedBlockingQueue} with the given (fixed) capacity.

     \*

     \* **@param** capacity the capacity of this queue

     \* **@throws** IllegalArgumentException if {@code capacity} is not greater

     \*         than zero

     \*/

**public** LinkedBlockingQueue(**int** capacity) {

**if** **(capacity** **<= 0)** **throw** **new** **IllegalArgumentException(); 😊😊**

**this**.capacity = capacity;

        last = head = **new** Node<E>(**null**);

    }

**The producer will attempt to batch records together into fewer requests whenever multiple records are being sent to the same partition**. **This helps performance on both the client and the server. This configuration controls the default batch size in bytes.**

**No attempt will be made to batch records larger than this size.**

**Requests sent to brokers will contain multiple batches 😻😻😻**, **one for each partition with data available to be sent.**

A small batch size will make batching less common and may reduce throughput (a batch size of zero will disable batching entirely). **A very large batch size may use memory a bit more wastefully as we will always allocate a buffer of the specified batch size in anticipation of additional records.😻😻😻**

Thread delay after indexing and committing offsets to simulate processing.

Log cleaner for log compaction -

Recall that a Kafka topic has a log. A log is broken up into partitions and partitions are divided into segments which contain records which have keys and values.

The *Kafka Log Cleaner* does log compaction. The *Log cleaner* has a pool of background compaction threads. These threads recopy log segment files, removing older records whose key reappears recently in the log. Each compaction thread chooses topic log that has the highest ratio of log head to log tail. Then the compaction thread recopies the log from start to end removing records whose keys occur later in the log.

As the log cleaner cleans log partition segments, the segments get swapped into the log partition immediately replacing the older segments. This way compaction does not require double the space of the entire partition as additional disk space required is just one additional log partition segment - divide and conquer.

Deletion - log retention hours

                    log retention bytes

Compaction - log segment ms

                       log segment size

                       Compaction lag for a message

                        After compaction redundant data is deleted with a lag - Delete retention

**log**.cleaner.min.**compaction**.lag.ms

Clean up configured per topic..

Cleaning up kafka queue...

-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Consumers works the same way for compacted topics as non-compacted ones. During compaction if there are offsets with the same keys then only the latest key stays but the compacted offsets are never deleted it stays pointing to the latest key Eg. if offsets 10,11,12 have the same keys, then only the 12th offset key would be retained after compaction and any consumer fetching offset 10,11 or 12 , it would fetch the same result i.e. key-value stored at offset 12 ( as key-value for 10,11 are deleted)

What Is Log Compaction Good For?

Since Log compaction retains last known value **it is a full snapshot of the latest records** it is useful for restoring state after a crash or system failure for an in-memory service, a persistent data store, or reloading a cache. It allows downstream consumers to restore their state. 😍😍

Coming to your questions -

1.Consumer can fetch from the desired offset , only thing is that if some offsets are compacted ,you will get the latest value for the compacted offsets

2.As explained consumers will continue to fetch from the last committed offset and if the offsets to be fetched has compacted you might get duplicate messages . So idempotency is almost always important as consumer may be reading from a compacted topic.

Refer to the compaction logic in detail in the below kafka link

What happens when a broker is down depends on your configuration. It mostly depends on these configuration settings:

* min.insync.replicas
* default.replication.factor
* unclean.leader.election.enable

Kafka does not create a new replica when a broker goes down.

If the offline broker was a leader, a new leader is elected from the replicas that are in-sync. If no replicas are in-sync it will only elect an out of sync replica if unclean.leader.election.enable is true, otherwise the partition will be offline.

If the offline broker was a follower, it will be marked a out of sync by the leader.

When restarting the broker, it will try to get back in sync. Once done, whether it stays a follower or becomes the leader depends if it is the prefered replica.

Finally if you know a broker will be offline for a long time and still require a replica, you can use the reassignment tool kafka-reassign-partitions.sh to move partitions to online brokers.

Every broker has a information about the list of topics(and partitions) and their leaders which will be kept up to date by the zoo keeper whenever the new leader is elected or when the number of partition changes.

Thus, when the producer makes a call to one of the brokers, it responds with this information list. Once the producer receives this information, it caches this and uses it to connect to the leader. So next time when it wants to send the message to that particular topic (and partition) it will use this cached information.

Lets assume there was only one leader and there are no replicas for that topic/ partition duo and it got crushed. In this case it will try to connect to that leader and it fails. It will try to fetch the leader from the other brokers list which it has cached to check if there is any leader for this topic! As it does not find any, it will try to hit to the same leader(that is dead) and after reaching a maximum no of retries it will throw an exception !!