Kafka Low-Level Design

Kafka Design Motivation Scale

Tos

- To scale Kafka is
  - distributed,
  - supports sharding
  - load balancing
- Scaling needs inspired Kafka partitioning and consumer model
- Kafka scales writes and reads with partitioned, distributed, commit logs

Persistence: Embrace Filesystem

Tos

- Kafka relies heavily on filesystem for storing and caching messages/records
- Disk performance of hard drives performance of sequential writes is fast
  - JBOD with six 7200rpm SATA RAID-5 array clocks at 600MB/sec
  - Heavily optimized by operating systems
- Ton of cache: Operating systems use available of main memory for disk caching
- JVM GC overhead is high for caching objects OS file caches are almost free
- Kafka greatly simplifies code for cache coherence by using OS page cache
- Kafka disk does sequential reads easily optimized by OS page cache

Big fast HDDs and long sequential access

Tos

- Like Cassandra, LevelDB, RocksDB, and others, Kafka uses long sequential disk access for read and writes
- Kafka uses tombstones instead of deleting records right away
- Modern Disks have somewhat unlimited space and are fast
- Kafka can provide features not usually found in a messaging system like holding on to old messages for a really long time
  - This flexibility allows for interesting application of Kafka

Kafka Record Retention Redux

Tos

- Kafka cluster retains all published records
  - Time based – configurable retention period
  - Size based - configurable based on size
  - Compaction - keeps latest record
- Kafka uses Topic ***Partitions***
- Partitions are broken down into ***Segment*** files

Broker Log Config

Tos

NAME	DESCRIPTION	DEFAULT
log.dir	topic logs will be stored in this or log.dirs	/tmp/kafka – logs
log.dirs	The directories where the Topics logs are kept used for JBOD	
log.flush.interval.messages	Accumulated messages	9,223,372,036,854
log.flush.interval.ms	Maximum time that a topic message is kept in memory before flushed to disk. If not set uses log.flush.scheduler.interval.ms	
log.flush.offset.checkpoint.interval.ms	Interval to flush log recovery point.	60,000
log.flush.scheduler.interval.ms	Interval that topic messages are periodically flushed from memory to log.	9,223,372,036,854,780,000

Broker Log Retention Config

Tos

log.retention.bytes	Delete log records by size. The maximum size of the log before deleting its older records	long	-1
Log.retention.hours	Delete log records by time hours. Hours to keep a log file before deleting older records(in hours),tertiary to log.retention.ms property.	int	168
log.retention.minutes	Delete log records by time minutes. Minutes to keep a log file before deleting it,secondary to log.retention.ms property. If not set,use log.retention.hours is used	Int	Null
log.retention.ms	Delete log records by time milliseconds. Milliseconds to keep a log file before deleting it, If not set, use log.retention.minutes.	Long	Null

Broker Log Segment File Config

Tos

Kafka Broker Config Log Segment

NAME	DESCRIPTION	TYPE	DEFAULT
Log.roll.hours	Time period before rolling a new topic log segment.(secondary to log.roll.ms property)	Int	1
log.roll.ms	Time period in milliseconds before rolling a new log segment. If not,uses log.roll.hours.	Long	
log.segment.bytes	The maximum size of a single log segment file.	Int	1,073,741,82
log.segment.delete.delay.ms	Time period to wait before deleting a segment file from the filesystem	Long	60,000

Kafka Producer Load Balancing

Tos

- Producer sends records directly to Kafka broker partition leader
- Producer asks Kafka broker for metadata about which Kafka broker has which topic partitions leaders - thus no routing layer needed
- Producer client controls which partition it publishes messages to
- Partitioning can be done by key, round-robin or using a custom semantic partitioner

Kafka Producer Record Batching

Tos

- Kafka producers support record batching.
  - by the size of records and auto-flushed based on time
- Batching is good for network IO throughput.
- Batching speeds up throughput drastically.
- Buffering is configurable
  - lets you make a tradeoff between additional latency for better throughput.
- Producer sends multiple records at a time which equates to fewer IO requests instead of lots of one by one sends

More producer settings for performance

Tos

KafkaExample.java

KafkaExample

```
21      private static Producer<Long, String> createProducer() {
22          Properties props = new Properties();
23          props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, BOOTSTRAP_SERVERS);
24          props.put(ProducerConfig.CLIENT_ID_CONFIG, "KafkaExampleProducer");
25          props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, LongSerializer.class.getName());
26          props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName(
27
28          //The batch.size in bytes of record size, 0 disables batching
29          props.put(ProducerConfig.BATCH_SIZE_CONFIG, 32768);
30
31          //Linger how much to wait for other records before sending the batch over the network.
32          props.put(ProducerConfig.LINGER_MS_CONFIG, 20);
33
34          // The total bytes of memory the producer can use to buffer records waiting to be sent
35          // to the Kafka broker. If records are sent faster than broker can handle than
36          // the producer blocks. Used for compression and in-flight records.
37          props.put(ProducerConfig.BUFFER_MEMORY_CONFIG, 67_108_864);
38
39          //Control how much time Producer blocks before throwing BufferExhaustedException.
40          props.put(ProducerConfig.MAX_BLOCK_MS_CONFIG, 1000);
```

For higher throughput, Kafka Producer allows buffering based on time and size.

Multiple records can be sent as a batches with fewer network requests. Speeds up throughput drastically.

Kafka Compression

Tos

- Kafka provides ***End-to-end Batch Compression***
- Bottleneck is not always CPU or disk but often network bandwidth
  - especially in cloud, containerized and virtualized environments
  - especially when talking datacenter to datacenter or WAN
- Instead of compressing records one at a time, compresses whole batch
- Message batches can be compressed and sent to Kafka broker/server in one go
- Message batch get written in compressed form in log partition
  - don't get decompressed until they consumer
- GZIP, Snappy and LZ4 compression protocols supported

Kafka Compression Config

Compression.type	Configures compression type for topics. Can be set to codecs ‘gzip’,’snappy’,’Iz4’or ‘uncompressed’. If set to ‘producer’ then it retains compression codec set by the producer(so it does not have to be uncompressed and then recompressed).	Default producer

Pull vs. Push/Streams: Pull

Tos

- With Kafka consumers ***pull*** data from brokers
- Other systems are push based or stream data to consumers
- Messaging is usually a pull-based system (SQS, most MOM is pull)
  - if consumer fall behind, it catches up later when it can
- Pull-based can implement aggressive batching of data
- Pull based systems usually implement some sort of ***long poll***
  - long poll keeps a connection open for response after a request for a period
- Pull based systems have to pull data and then process it
  - There is always a pause between the pull

Pull vs. Push/Streams: Push

Tos

- Push based push data to consumers (scribe, flume, reactive streams, RxJava, Akka)
  - push-based have problems dealing with slow or dead consumers
  - push system consumer can get overwhelmed
  - push based systems use back-off protocol (back pressure)
    - consumer can indicate it is overwhelmed, (http://www.reactive-streams.org/)
- Push-based streaming system can
  - send a request immediately or accumulate request and send in batches
- Push-based systems are always pushing data or streaming data
  - Advantage: Consumer can accumulate data while it is processing data already sent
  - Disadvantage: If consumer dies, how does broker know and when does data get resent to another consumer (harder to manage message acks; more complex)

MOM Consumer Message State

Tos

- With most MOM it is brokers responsibility to keep track of which messages have been consumed
- As message is consumed by a consumer, broker keeps track
  - broker may delete data quickly after consumption
- Trickier than it sounds (acknowledgement feature), lots of state to track per message, sent, acknowledge

Kafka Consumer Message State

Tos

- Kafka topic is divided into ordered partitions - A topic partition gets read by only one ***consumer*** per ***consumer group***
- Offset data is not tracked per message - ***a lot less data to track***
  - just stores offset of each ***consumer group, partition pairs***
  - Consumer sends offset Data periodically to Kafka Broker
  - Message acknowledgement is cheap compared to MOM
- Consumer can rewind to older offset (replay)
  - If bug then fix, rewind consumer and replay

Message Delivery Semantics

Tos

- At most once
  - Messages may be lost but are never redelivered
- At least once
  - Messages are never lost but may be redelivered
- Exactly once
  - this is what people actually want, each message is delivered once and only once

Consumer: Message Delivery Semantics

Tos

- "at-most-once“ - Consumer reads message, save offset, process message
  - Problem: consumer process dies after saving position but before processing message - consumer takes over starts at last position and message never processed
- “at-least-once" - Consumer reads message, process messages, saves offset
  - Problem: consumer could crash after processing message but before saving position - consumer takes over receives already processed message
- “exactly once” - need a two-phase commit for consumer position, and message process output - or, store consumer message process output in same location as last position
- Kafka offers the first two and you can implement the third or 0.11 API Support this

Kafka Producer Acknowledgement

Tos

- Kafka's offers operational predictable semantics
- When publishing a message, message get ***committed*** to the log
  - Durable as long as at least one replica lives
- If Producer connection goes down during of send
  - Producer not sure if message sent; resends until message sent ack received (log could have duplicates)
  - Important: use message keys, idempotent messages
  - Not guaranteed to not duplicate from producer retry

Kafka Replication

Tos

- Kafka replicates each topic's partitions across a configurable number of Kafka brokers
- Kafka is replicated by default not a bolt-on feature
- Each topic partition has one leader and zero or more followers
  - leaders and followers are called replicas
  - replication factor = 1 leader + N followers
- Reads and writes always go to leader
- Partition leadership is evenly shared among Kafka brokers
  - logs on followers are in-sync to leader's log - identical copy - sans un-replicated offsets
- Followers pull records in batches records from leader like a regular Kafka consumer

Kafka Broker Failover

Tos

- Kafka keeps track of which Kafka Brokers are alive (in-sync)
  - To be alive Kafka Broker must maintain a ZooKeeper session (heart beat)
  - Followers must replicate writes from leader and not fall "too far" behind
- Each leader keeps track of set of "in sync replicas" aka ISRs
- If ISR/follower dies, falls behind, leader will removes follower from ISR set - falling behind
  replica.lag.time.max.ms > lag
- Kafka guarantee: committed message not lost, as long as one live ISR - "committed" when written to all ISRs logs
- Consumer only reads committed messages

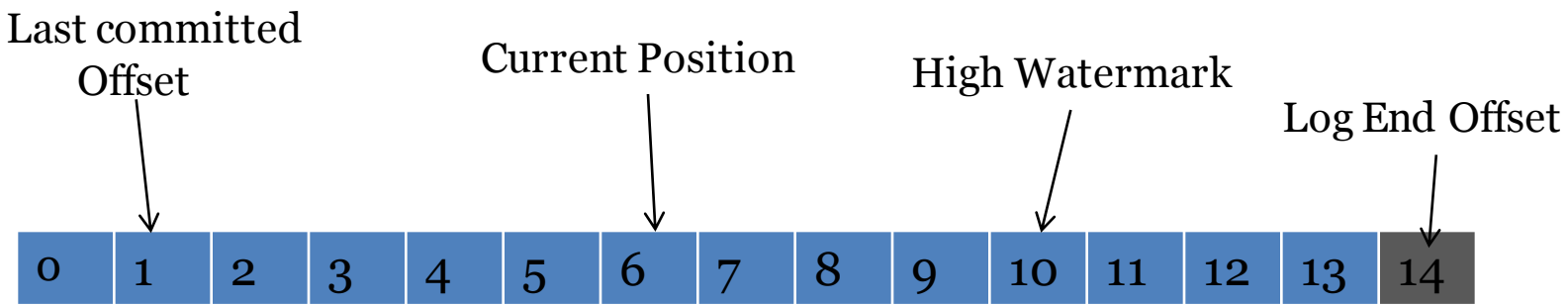
Replicated Log Partitions

Tos

- A Kafka partition is a replicated log - replicated log is a distributed data system primitive
- Replicated log useful for building distributed systems using state-machines
- A replicated log models “coming into consensus” on ordered series of values
  - While leader stays alive, all followers just need to copy values and ordering from leader
- When leader does die, a new leader is chosen from its in-sync followers
- If producer told a message is committed, and then leader fails, new elected leader must have that committed message
- More ISRs; more to elect during a leadership failure

Kafka Consumer Replication

- What can be consumed?
- ***"Log end offset"*** is offset of last record written to log partition and where ***Producers*** write to next
- ***"High watermark"*** is offset of last record successfully replicated to all partitions followers
- ***Consumer*** only reads up to “high watermark”.
  Consumer can’t read un-replicated data



Kafka Broker Replication Config

Tos

NAME	DESCRIPTION	TYPE	DEFAULT
auto.leader.rebalance.enable	Enables auto leader balancing.	Boolean	True
leader.imbalance.check.interval.seconds	The interval for checking for partition leadership balancing	Long	300
leader.imbalance.per.broker.percentage	Leadership imbalance for each broker. If imbalance is too high then a rebalance is triggered.	Int	10
min.insync.replicas	When a producer sets acks to all(or-1). This setting is the minimum replicas count that must acknowledge a write for the write to be considered successful. If not met, then the producer will raise an exception (either NotEnoughReplicas or NotEnoughReplicasAfterAppend).	Int	1
num.replica.fetchers	Replica fetcher count. Used to replicate messages from a broker that has a leadership partition. Increase this if followers are falling behind.	Int	1

Kafka Replication Broker Config 2

Tos

NAME	DESCRIPTION
replica.high.watermark.checkpoint.interval.ms	The frequency with which the high watermark is saved out to disk used for knowing what consumers can consume. **Consumer** only reads up to “high watermark”. **Consumer can’t read un-replicated data.**
replica.lag.time.max.ms	Determines which Replicas are in the ISR set and which are not. ISR is important for acks and quorum.
replica.socket.receive.buffer.bytes	The socket receive buffer for network request
replica.socket.timeout.ms	The socket timeout for network requests. Its value should be at least replica.fetch.wait.max.ms
unclean.leader.election.enable	What happens if all of the nodes go down? Indicates whether to enable replicas not in the ISR. Replicas that are not in-sync. Set to be elected as leader as a last resort, even though doing so may result in data loss. Availability over Consistency. True is the default.

Kafka and Quorum

Tos

- Quorum is number of acknowledgements required and number of logs that must be compared to elect a leader such that there is guaranteed to be an overlap
- Most systems use a majority vote - Kafka does not use a majority vote
- Leaders are selected based on having the most complete log
- Problem with majority vote Quorum is it does not take many failure to have inoperable cluster

Kafka and Quorum 2

Tos

- If we have a replication factor of 3
  - Then at least two ISRs must be in-sync before the leader declares a sent message committed
  - If a new leader needs to be elected then, with no more than 3 failures, the new leader is guaranteed to have all committed messages
  - Among the followers there must be at least one replica that contains all committed messages

Kafka Quorum Majority of ISRs

Tos

- Kafka maintains a set of ISRs
- Only this set of ISRs are eligible for leadership election
- Write to partition is not committed until all ISRs ack write
- ISRs persisted to ZooKeeper whenever ISR set changes

All nodes die at same time. Now what?

Tos

- Kafka's guarantee about data loss is only valid if at least one replica being in-sync
- If all followers that are replicating a partition leader die at once, then data loss Kafka guarantee is not valid.
- If all replicas are down for a partition, Kafka chooses first replica (not necessarily in ISR set) that comes alive as the leader
  - Config ***unclean.leader.election.enable=true*** is default
- If ***unclean.leader.election.enable=false,*** if all replicas are down for a partition, Kafka waits for the ISR member that comes alive as new leader.

Producer Durability Acks

Tos

- Producers can choose durability by setting ***acks*** to - 0, 1 or all replicas
- acks=all is ***default***, acks happens when all current in-sync replicas (ISR) have received the message
- If durability over availability is prefer
  - Disable unclean leader election
  - Specify a minimum ISR size
    - trade-off between consistency and availability
    - higher minimum ISR size guarantees better consistency
    - but higher minimum ISR reduces availability since partition won't be unavailable for writes if size of ISR set is less than threshold

Quotas

Tos

- Kafka has quotas for Consumers and Producers
- Limits bandwidth they are allowed to consume
- Prevents Consumer or Producer from hogging up all Broker resources
- Quota is by client id or user
- Data is stored in ZooKeeper; changes do not necessitate restarting Kafka