FAQ

Please help us keep this FAQ up-to-date. If there is an answer that you think can be improved, please help improve it. If you look for an answer that isn't here, and later figure it out, please add it. You don't need permission, it's a wiki. 🙂

Streams

When I commit my process state, what does the Streams library do and how it affects my application's performance?

Streams application's state can be committed either periodically based on the `commit.interval` config, or whenever users call `context.commit` in their application code.

When commit operation is triggered, Streams library will make the following steps:
1. Flush the state store, and write the checkpoint file for the current log end offset of the store.
2. Flush the producer to make sure any batched records has been sent and acked by the brokers.
3. Send a commit offset request to brokers in sync mode to the broker for the input topics.
4*. If exactly_once mode is turned on, there are extra steps needed to make sure the above steps are done atomically.
As one can see, these steps have non-negligible overhead on the throughput, since they need to be done synchronously while no records are being processed at the mean time. On the other hand, the shorter the commit interval less time is needed when streams application is re-bootstrapped. In practice users need to consider tuning commit frequency so that it is not affecting their application performance during normal runs, while still have a fast failover / restart cost.

My application failed to start, with the a rocksDB exception raised as "java.lang.ExceptionInInitializerError.. Unable to load the RocksDB shared libraryjava...". How to resolve this?

Streams API uses RocksDB as the default local persistent key-value store. And RocksDB JNI would try to statically load the sharedlibs into java.io.tmpdir. On Unix-like platforms, the default value of this system environment property is typically `"/tmp"`, or `"/var/tmp";` On Microsoft Windows systems the property is typically `"C:\\WINNT\\TEMP"`.

If your application does not have permission to access these directories (or for Unix-like platforms if the pointed location is not mounted), the above error would be thrown. To fix this, you can either grant the permission to access this directory to your applications, or change this property when executing your application like "java -Djava.io.tmpdir=..".

My application's memory usage keeps increasing when running until it hits an OOM. Why?

The most common cause of this scenario is that you did not close an iterator from the state stores after completed using it. For persistent stores like RocksDB, an iterator is usually backed by some physical resources like open file handlers and in-memory caches. Not closing these iterators would effectively causing resource leaks.

For more details please read this javadoc.

I saw exceptions running my Streams app for the first time as "Extracted timestamp value is negative, which is not allowed." What does this mean?

This error means that the timestamp extractor of your Kafka Streams application failed to extract a valid timestamp from a record. Typically, this points to a problem with the record (e.g., the record does not contain a timestamp at all), but it could also indicate a problem or bug in the timestamp extractor used by the application.

When does a record not contain a valid timestamp:

- If you are using the default FailOnInvalidTimestamp (for v0.10.0 and v0.10.1 `ConsumerRecordTimestampExtractor`), it is most likely that your records do not carry an embedded timestamp (embedded record timestamps got introduced in Kafka's message format in Kafka `0.10`). This might happen, if for example, you consume a topic that is written by old Kafka producer clients (i.e., version `0.9` or earlier) or by third-party producer clients. Another situation where this may happen is after upgrading your Kafka cluster from `0.9` to `0.10`, where all the data that was generated with `0.9` does not include the `0.10` message timestamps.
- If you are using a custom timestamp extractor, make sure that your extractor is properly handling invalid (negative) timestamps, where "properly" depends on the semantics of your application. For example, you can return a default or an estimated timestamp if you cannot extract a valid timestamp (maybe the timestamp field in your data is just missing).
- As of Kafka 0.10.2, there are two alternative extractors namely LogAndSkipOnInvalidTimestamp and UsePreviousTimeOnInvalidTimestamp that handle invalid record timestamps more gracefully (but with potential data loss or semantical impact).
- You can also switch to processing-time semantics via `WallclockTimestampExtractor`; whether such a fallback is an appropriate response to this situation depends on your use case.

How to write a custom `TimestampExtractor` ?

If you want to provide a custom timestamp extractor, you have to note, that the extractor is applied globally, i.e., to all user and Streams-internal topics. Because, Kafka Streams relies on record metadata timestamps for all its internal topics, you need to write you extractor in a flexible way and return different timestamps for different topics. In detail, you should branch (`if-else`) within your code and only apply custom logic for records from your input topics you care about (you can check the topic via `ConsumerRecord#topic()`). For all other topic (inclusive all Streams internal topics) you should return the record's metadata timestamp you can access via `ConsumerRecord#timestamp()`.

How to scale a Streams app, i.e., increase number of input partitions?

Basically, Kafka Streams does not allow to change the number of input topic partitions during its "life time". If you stop a running Kafka Streams application, change the number of input topic partitions, and restart your app it will most likely break with an exception as described in FAQ *"What does exception "Store <someStoreName>'s change log (<someStoreName>-changelog) does not contain partition <someNumber>" mean?"* It is

tricky to fix this for production use cases and it is highly recommended to **not** change the number of input topic partitions (cf. comment below). For POC/demos it's not difficult to fix though.

In order to fix this, you should reset your application using Kafka's application reset tool: Kafka Streams Application Reset Tool. Using the application reset tool, has the disadvantage that you wipe out your whole application state. Thus, in order to get your application into the same state as before, you need to reprocess the whole input topic from beginning. This is of course only possible, if all input data is still available and nothing got deleted by brokers that applying topic retention time/size policy. Furthermore you should note, that adding partitions to input topics changes the topic's partitioning schema (be default hash-based partitioning by key). Because Kafka Streams assumes that input topics are correctly partitioned by key, if you use the reset tool and reprocess all data, you might get wrong result as "old" data is partitioned differently than "new" data (ie, data written after adding the new partitions). For production use cases, you would need to read all data from your original topic and write it into a new topic (with increased number of partitions) to get your data partitioned correctly (or course, this step might change the ordering of records with different keys -- what should not be an issue usually -- just wanted to mention it). Afterwards you can use the new topic as input topic for your Streams app. This repartitioning step can be done easily within your Streams application that read and write's back the date without any further processing

In general however, it is recommended to over partition your topics for production use cases, such that you will never need to change the number of partitions later on. The overhead of over partitioning is rather small and saves you a lot of hassle later on. This is a general recommendation if you work with Kafka -- it's not limited to Streams use cases.

> *One more remark:*
>
> *Some people might suggest to increase the number of partitions of Kafka Streams internal topics manually. First, this would be a* hac k *and is* ***not recommended*** *for certain reasons.*
>
> 1. *It might be tricky to figure out what the right number is, as it depends on various factors (as it's a Stream's internal implementation detail).*
> 2. *You also face the problem of breaking the partitioning scheme, as described in the paragraph above. Thus, you application most likely ends up in an inconsistent state.*
>
> *In order to avoid inconsistent application state, Streams does not delete any internal topics or changes the number of partitions of internal topics automatically, but fails with the error message you reported. This ensure, that the user is aware of all implications by doing the "cleanup" manually.*

What does exception "Store <someStoreName>'s change log (<someStoreName>-changelog) does not contain partition <someNumber>" mean?

It means that a Streams internal changelog topic does not have the expected number of topic partitions. This can happen, if you add partitions to an input topic. For more details, see FAQ "How to scale a Streams app, i.e., increase the number of input topic partitions".

I get a locking exception similar to "Caused by: java.io.IOException: Failed to lock the state directory: /tmp/kafka-streams/<app-id>/0_0". How can I resolve this?

Kafka 0.10.0 and 0.10.1 does have some bugs with regard to multi-threading. If you cannot (or don't want to) update, you can apply the following workaround:

- switch to single threaded execution
- if you want to scale your app, start multiple instances (instead of going multi-threaded with one instance)
- if you start multiple instances on the same host, use a different state directory (`state.dir` config parameter) for each instance (to "isolate" the instances from each other)

It might also be necessary, to delete the state directory manually before starting the application. This will not result in data loss – the state will be recreated from the underlying changelog topic.0

How can I access record metadata?

Record metadata is accessible via `ProcessorContext` that is provided via `Processor#init()`, `Transformer#init()`, and `ValueTransformer#init()`. The context object is updated under the hood and contains the metadata for the currently processed record each time `process()`, `transform()`, or `transformValue()` is called.

Accessing record metadata via ProcessorContext

```
public class MyProcessor implements Processor<KEY_TYPE, VALUE_TYPE> {
  private ProcessorContext context;

  @Override
  public void init(final ProcessorContext context) {
    this.context = context;
  }

  @Override
  void process(final KEY_TYPE key, final VALUE_TYPE value) {
      this.context.offset(); // returns the current record's offset
      // you can also access #partition(), #timestamp(), and #topic()
  }

  // other methods omitted for brevity
}
```

Why do I get an `IllegalStateException` when accessing record metadata?

If you attach a new `Processor/Transformer/ValueTransformer` to your topology using a corresponding supplier, you need to make sure that the supplier returns a *new* instance each time `get()` is called. If you return the same object, a single `Processor/Transformer/ValueTransformer` would be shared over multiple tasks resulting in an `IllegalStateException` with error message `"This should not happen as topic() should only be called while a record is processed"` (depending on the method you are calling it could also be `partition()`, `offset()`, or `timestamp()` instead of `topic()`).

Producers

How should I set metadata.broker.list?

The broker list provided to the producer is only used for fetching metadata. Once the metadata response is received, the producer will send produce requests to the broker hosting the corresponding topic/partition directly, using the ip/port the broker registered in ZK. Any broker can serve metadata requests. The client is responsible for making sure that at least one of the brokers in metadata.broker.list is accessible. One way to achieve this is to use a VIP in a load balancer. If brokers change in a cluster, one can just update the hosts associated with the VIP.

Why do I get QueueFullException in my producer when running in async mode?

This typically happens when the producer is trying to send messages quicker than the broker can handle. If the producer can't block, one will have to add enough brokers so that they jointly can handle the load. If the producer can block, one can set queue.enqueueTimeout.ms in producer config to -1. This way, if the queue is full, the producer will block instead of dropping messages.

I am using the ZK-based producer in 0.7 and I see data only produced on some of the brokers, but not all, why?

This is related to an issue in Kafka 0.7.x (see the discussion in http://apache.markmail.org/thread/c7tdalfketpusqkg). Basically, for a new topic, the producer bootstraps using all existing brokers. However, if a topic already exists on some brokers, the producer never bootstraps again when new brokers are added to the cluster. This means that the producer won't see those new broker. A workaround is to manually create the log directory for that topic on the new brokers.

Why are my brokers not receiving producer sent messages?

This happened when I tried to enable gzip compression by setting compression.codec to 1. With the code change, not a single message was received by the brokers even though I had called producer.send() 1 million times. No error printed by producer and no error could be found in broker's kafka-request.log. By adding log4j.properties to my producer's classpath and switching the log level to DEBUG, I captured the java.lang.NoClassDefFoundError: org/xerial/snappy/SnappyInputStream thrown at the producer side. Now I can see this error can be resolved by adding snappy jar to the producer's classpath.

Why is data not evenly distributed among partitions when a partitioning key is not specified?

In Kafka producer, a partition key can be specified to indicate the destination partition of the message. By default, a hashing-based partitioner is used to determine the partition id given the key, and people can use customized partitioners also.

To reduce # of open sockets, in 0.8.0 (https://issues.apache.org/jira/browse/KAFKA-1017), when the partitioning key is not specified or null, a producer will pick a random partition and stick to it for some time (default is 10 mins) before switching to another one. So, if there are fewer producers than partitions, at a given point of time, some partitions may not receive any data. To alleviate this problem, one can either reduce the metadata refresh interval or specify a message key and a customized random partitioner. For more detail see this thread http://mail-archives.apache.org/mod_mbox/kafka-dev/201310.mbox/%3CCAFbh0Q0aVh%2Bvqxfy7H-%2BMnRFBt6BnyoZk1LWBoMspwSmTqUKMg%40mail.gmail.com%3E

Is it possible to delete a topic?

Deleting a topic is supported since 0.8.2.x. You will need to enable topic deletion (setting *delete.topic.enable* to true) on all brokers first.

Consumers

Why does my consumer never get any data?

By default, when a consumer is started for the very first time, it ignores all existing data in a topic and will only consume new data coming in after the consumer is started. If this is the case, try sending some more data after the consumer is started. Alternatively, you can configure the consumer by setting auto.offset.reset to "earliest" for the new consumer in 0.9 and "smallest" for the old consumer.

Why does my consumer get InvalidMessageSizeException?

This typically means that the "fetch size" of the consumer is too small. Each time the consumer pulls data from the broker, it reads bytes up to a configured limit. If that limit is smaller than the largest single message stored in Kafka, the consumer can't decode the message properly and will throw an InvalidMessageSizeException. To fix this, increase the limit by setting the property "fetch.size" (0.7) / "fetch.message.max.bytes" (0.8) properly in config/consumer.properties. The default fetch.size is 300,000 bytes. For the new consumer in 0.9, the property to adjust is "max.partition.fetch.bytes," and the default is 1MB.

Should I choose multiple group ids or a single one for the consumers?

If all consumers use the same group id, messages in a topic are distributed among those consumers. In other words, each consumer will get a non-overlapping subset of the messages. Having more consumers in the same group increases the degree of parallelism and the overall throughput of consumption. See the next question for the choice of the number of consumer instances. On the other hand, if each consumer is in its own group, each consumer will get a full copy of all messages.

Why some of the consumers in a consumer group never receive any message?

Currently, a topic partition is the smallest unit that we distribute messages among consumers in the same consumer group. So, if the number of consumers is larger than the total number of partitions in a Kafka cluster (across all brokers), some consumers will never get any data. The solution is to increase the number of partitions on the broker.

Why are there many rebalances in my consumer log?

A typical reason for many rebalances is the consumer side GC. If so, you will see Zookeeper session expirations in the consumer log (grep for Expired). Occasional rebalances are fine. Too many rebalances can slow down the consumption and one will need to tune the java GC setting.

Can I predict the results of the consumer rebalance?

During the rebalance process, each consumer will execute the same deterministic algorithm to range partition a sorted list of topic-partitions over a sorted list of consumer instances. This makes the whole rebalancing process deterministic. For example, if you only have one partition for a specific topic and going to have two consumers consuming this topic, only one consumer will get the data from the partition of the topic; and even if the consumer named "Consumer1" is registered after the other consumer named "Consumer2", it will replace "Consumer2" gaining the ownership of the partition in the rebalance.

Range partitioning works on a per-topic basis. For each topic, we lay out the available partitions in numeric order and the consumer threads in lexicographic order. We then divide the number of partitions by the total number of consumer streams (threads) to determine the number of partitions to allocate to each consumer. If it does not evenly divide, then the first few consumers will have one extra partition. For example,

suppose there are two consumers C1 and C2 with two streams each, and there are five available partitions (p0, p1, p2, p3, p4). So each consumer thread will get at least one partition and the first consumer thread will get one extra partition. So the assignment will be: p0 -> C1-0, p1 -> C1-0, p2 -> C1-1, p3 -> C2-0, p4 -> C2-1

My consumer seems to have stopped, why?

First, try to figure out if the consumer has really stopped or is just slow. You can use our tool

ConsumerOffsetChecker

```
bin/kafka-run-class.sh kafka.tools.ConsumerOffsetChecker --group consumer-group1
--zkconnect zkhost:zkport --topic topic1
consumer-group1,topic1,0-0 (Group,Topic,BrokerId-PartitionId)
Owner = consumer-group1-consumer1
Consumer offset = 70121994703
= 70,121,994,703 (65.31G)
Log size = 70122018287
= 70,122,018,287 (65.31G)
Consumer lag = 23584
= 23,584 (0.00G)
```

In 0.8, you can also monitor the MaxLag and the MinFetch jmx bean (see http://kafka.apache.org/documentation.html#monitoring).

If consumer offset is not moving after some time, then consumer is likely to have stopped. If consumer offset is moving, but consumer lag (difference between the end of the log and the consumer offset) is increasing, the consumer is slower than the producer. If the consumer is slow, the typical solution is to increase the degree of parallelism in the consumer. This may require increasing the number of partitions of a topic.

The high-level consumer will block if

- there are no more messages available
    - The ConsumerOffsetChecker will show that the log offset of the partitions being consumed does not change on the broker
- the next message available is larger than the maximum fetch size you have specified
    - One possibility of a stalled consumer is that the fetch size in the consumer is smaller than the largest message in the broker. You can use the DumpLogSegments tool to figure out the largest message size and set fetch.size in the consumer config accordingly.
- your client code simply stops pulling messages from the iterator (the blocking queue will fill up).
    - One of the typical causes is that the application code that consumes messages somehow died and therefore killed the consumer thread. We recommend using a try/catch clause to log all Throwable in the consumer logic.
- consumer rebalancing fails (you will see ConsumerRebalanceFailedException): This is due to conflicts when two consumers are trying to own the same topic partition. The log will show you what caused the conflict (search for "conflict in ").
    - If your consumer subscribes to many topics and your ZK server is busy, this could be caused by consumers not having enough time to see a consistent view of all consumers in the same group. If this is the case, try Increasing rebalance.max.retries and reb alance.backoff.ms.
    - Another reason could be that one of the consumers is hard killed. Other consumers during rebalancing won't realize that consumer is gone after zookeeper.session.timeout.ms time. In the case, make sure that rebalance.max.retries * rebalance.backoff.ms > zookeeper.session.timeout.ms.

Why messages are delayed in my consumer?

This could be a general throughput issue. If so, you can use more consumer streams (may need to increase # partitions) or make the consumption logic more efficient.

Another potential issue is when multiple topics are consumed in the same consumer connector. Internally, we have an in-memory queue for each topic, which feed the consumer iterators. We have a single fetcher thread per broker that issues multi-fetch requests for all topics. The fetcher thread iterates the fetched data and tries to put the data for different topics into its own in-memory queue. If one of the consumer is slow, eventually its corresponding in-memory queue will be full. As a result, the fetcher thread will block on putting data into that queue. Until that queue has more space, no data will be put into the queue for other topics. Therefore, those other topics, even if they have less volume, their consumption will be delayed because of that. To address this issue, either making sure that all consumers can keep up, or using separate consumer connectors for different topics.

How to improve the throughput of a remote consumer?

If the consumer is in a different data center from the broker, you may need to tune the socket buffer size to amortize the long network latency. Specifically, for Kafka 0.7, you can increase socket.receive.buffer in the broker, and socket.buffersize and fetch.size in the consumer. For Kafka

0.8, the consumer properties are socket.receive.buffer.bytes and fetch.message.max.bytes.

How can I rewind the offset in the consumer?

With the new consumer in 0.9, we have added a seek API to set to next position that will be fetched. You can either seek to the earliest position with seekToBeginning(), the latest with seekToEnd(), or to an arbitrary offset with seek().

If you are using the older high level consumer, currently there is no api to reset the offsets in the consumer. The only way is to stop all consumers and reset the offsets for that consumer group in ZK manually. We do have an import/export offset tool that you can use (bin/kafka-run-class.sh kafka.tools.ImportZkOffsets and bin/kafka-run-class.sh kafka.tools.ExportZkOffsets). To get the offsets for importing, we have a GetOffsetShell tool (bin/kafka-run-class.sh kafka.tools.GetOffsetShell) that allows you to get the offsets before a give timestamp. The offsets returned there are the offsets corresponding to the first message of each log segment. So the granularity is very coarse.

I don't want my consumer's offsets to be committed automatically. Can I manually manage my consumer's offsets?

You can turn off the autocommit behavior (which is on by default) by setting `auto.commit.enable=false` in your consumer's config. There are a couple of caveats to keep in mind when doing this:

- You will manually commit offsets using the consumer's `commitOffsets` API. Note that this will commit offsets for all partitions that the consumer currently owns. The consumer connector does not currently provide a more fine-grained commit API.
- If a consumer rebalances for any reason it will fetch the last committed offsets for any partitions that it ends up owning. If you have not yet committed any offsets for these partitions, then it will use the latest or earliest offset depending on whether `auto.offset.reset` is set to largest or smallest (respectively).

For releases prior to 0.9, if you need more fine-grained control over offsets, you will need to use the SimpleConsumer and manage offsets on your own. In 0.9, we have added a commitOffsets() API to the old consumer which takes a map with the specific offsets to be committed. A similar API is also available for the new consumer (commitSync() and commitAsync()).

What is the relationship between fetch.wait.max.ms and socket.timeout.ms on the consumer?

fetch.wait.max.ms controls how long a fetch request will wait on the broker in the normal case. The issue is that if there is a hard crash on the broker (host is down), the client may not realize this immediately since TCP will try very hard to maintain the socket connection. By setting socket.timeout.ms, we allow the client to break out sooner in this case. Typically, socket.timeout.ms should be set to be at least fetch.wait.max.ms or a bit larger. It's possible to specify an indefinite long poll by setting fetch.wait.max.ms to a very large value. It's not recommended right now due to https://issues.apache.org/jira/browse/KAFKA-1016. The consumer-config documentation states that "The actual timeout set will be max.fetch.wait + socket.timeout.ms." - however, that change seems to have been lost in the code a while ago. https://issues.apache.org/jira/browse/KAFKA-1147 is filed to fix it.

How do I get exactly-once messaging from Kafka?

Exactly once semantics has two parts: avoiding duplication during data production and avoiding duplicates during data consumption.

There are two approaches to getting exactly once semantics during data production:

1. Use a single-writer per partition and every time you get a network error check the last message in that partition to see if your last write succeeded
2. Include a primary key (UUID or something) in the message and deduplicate on the consumer.

If you do one of these things, the log that Kafka hosts will be duplicate-free. However, reading without duplicates depends on some co-operation from the consumer too. If the consumer is periodically checkpointing its position then if it fails and restarts it will restart from the checkpointed position. Thus if the data output and the checkpoint are not written atomically it will be possible to get duplicates here as well. This problem is particular to your storage system. For example, if you are using a database you could commit these together in a transaction. The HDFS loader Camus that LinkedIn wrote does something like this for Hadoop loads. The other alternative that doesn't require a transaction is to store the offset with the data loaded and deduplicate using the topic/partition/offset combination.

I think there are two improvements that would make this a lot easier:

1. Producer idempotence could be done automatically and much more cheaply by optionally integrating support for t his on the server.
2. The existing high-level consumer doesn't expose a lot of the more fine grained control of offsets (e.g. to reset your position). We will be working on that soon

Why can't I specify the number of streams parallelism per topic map using wildcard stream as I use static stream handler?

The reason we do not have per-topic parallelism specification in wildcard is that with the wildcard topicFilter, we will not know exactly which topics to consume at the construction time, hence no way to specify per-topic specs.

How to consume large messages?

First you need to make sure these large messages can be accepted at Kafka brokers. The broker property **`message.max.bytes`** controls the maximum size of a message that can be accepted at the broker, and any single message (including the wrapper message for compressed message set) whose size is larger than this value will be rejected for producing. Then you need to make sure consumers can fetch such large messages from brokers. For the old consumer, you should use the property **`fetch.message.max.bytes`**, which controls the maximum number of bytes a consumer issues in one fetch. If it is less than a message's size, the fetching will be blocked on that message keep retrying. The property for the new consumer is **`max.partition.fetch.bytes`**.

How do we migrate to committing offsets to Kafka (rather than Zookeeper) in 0.8.2?

(Answer provided by Jon Bringhurst on mailing list)

A summary of the migration procedure is:

1) Upgrade your brokers and set dual.commit.enabled=false and offsets.storage=zookeeper (Commit offsets to Zookeeper Only).
2) Set dual.commit.enabled=true and offsets.storage=kafka and restart (Commit offsets to Zookeeper and Kafka).
3) Set dual.commit.enabled=false and offsets.storage=kafka and restart (Commit offsets to Kafka only).

Brokers

How does Kafka depend on Zookeeper?

Starting from 0.9, we are removing all the Zookeeper dependency from the clients (for details one can check this page). However, the brokers will continue to be heavily depend on Zookeeper for:

1. Server failure detection.
2. Data partitioning.
3. In-sync data replication.

Once the Zookeeper quorum is down, brokers could result in a bad state and could not normally serve client requests, etc. Although when Zookeeper quorum recovers, the Kafka brokers should be able to resume to normal state automatically, there are still a few corner cases the they cannot and a hard kill-and-recovery is required to bring it back to normal. Hence it is recommended to closely monitor your zookeeper cluster and provision it so that it is performant.

Also note that if Zookeeper was hard killed previously, upon restart it may not successfully load all the data and update their creation timestamp. To resolve this you can clean-up the data directory of the Zookeeper before restarting (if you have critical metadata such as consumer offsets you would need to export / import them before / after you cleanup the Zookeeper data and restart the server).

Why do I see error "Should not set log end offset on partition" in the broker log?

Typically, you will see errors like the following.

kafka.common.KafkaException: Should not set log end offset on partition [test,22]'s local replica 4
ERROR [ReplicaFetcherThread-0-6], Error for partition [test,22] to broker 6:class kafka.common.UnknownException(kafka.server.ReplicaFetcherThread)

A common problem is that more than one broker registered the same host/port in Zookeeper. As a result, the replica fetcher is confused when fetching data from the leader. To verify that, you can use a Zookeeper client shell to list the registration info of each broker. The Zookeeper path and the format of the broker registration is described in Kafka data structures in Zookeeper. You want to make sure that all the registered brokers

have unique host/port.

Why does controlled shutdown fail?

If a controlled shutdown attempt fails, you will see error messages like the following in your broker logs

```
WARN [Kafka Server 0], Retrying controlled shutdown after the previous attempt failed...
(kafka.server.KafkaServer)
```

```
WARN [Kafka Server 0], Proceeding to do an unclean shutdown as all the controlled shutdown attempts failed
```

In addition to these error messages, if you also see SocketTimeoutExceptions, it indicates that the controller could not finish moving the leaders for all partitions on the broker within controller.socket.timeout.ms. The solution is to increase controller.socket.timeout.ms as well as increase controlled.shutdown.retry.backoff.ms and controlled.shutdown.max.retries to give enough time for the controlled shutdown to complete. If you don't see SocketTimeoutExceptions, it could indicate a problem in your cluster state or a bug as this happens when the controller is not able to move the leaders to another broker for several retries.

Why can't my consumers/producers connect to the brokers?

When a broker starts up, it registers its ip/port in ZK. You need to make sure the registered ip is consistent with what's listed in metadata.broker.list in the producer config. By default, the registered ip is given by InetAddress.getLocalHost.getHostAddress. Typically, this should return the real ip of the host. However, sometimes (e.g., in EC2), the returned ip is an internal one and can't be connected to from outside. The solution is to explicitly set the host ip to be registered in ZK by setting the "hostname" property in server.properties. In another rare case where the binding host/port is different from the host/port for client connection, you can set advertised.host.name and advertised.port for client connection.

Why partition leaders migrate themselves some times?

During a broker soft failure, e.g., a long GC, its session on ZooKeeper may timeout and hence be treated as failed. Upon detecting this situation, Kafka will migrate all the partition leaderships it currently hosts to other replicas. And once the broker resumes from the soft failure, it can only act as the follower replica of the partitions it originally leads.

To move the leadership back to the brokers, one can use the preferred-leader-election tool here. Also, in 0.8.2 a new feature will be added which periodically trigger this functionality (details here).

To reduce Zookeeper session expiration, either tune the GC or increase zookeeper.session.timeout.ms in the broker config.

How many topics can I have?

Unlike many messaging systems Kafka topics are meant to scale up arbitrarily. Hence we encourage fewer large topics rather than many small topics. So for example if we were storing notifications for users we would encourage a design with a single notifications topic partitioned by user id rather than a separate topic per user.

The actual scalability is for the most part determined by the number of total partitions across all topics not the number of topics itself (see the question below for details).

How do I choose the number of partitions for a topic?

There isn't really a right answer, we expose this as an option because it is a tradeoff. The simple answer is that the partition count determines the maximum consumer parallelism and so you should set a partition count based on the maximum consumer parallelism you would expect to need (i.e. over-provision). Clusters with up to 10k total partitions are quite workable. Beyond that we don't aggressively test (it should work, but we can't guarantee it).

Here is a more complete list of tradeoffs to consider:

- A partition is basically a directory of log files.
- Each partition must fit entirely on one machine. So if you have only one partition in your topic you cannot scale your write rate or retention beyond the capability of a single machine. If you have 1000 partitions you could potentially use 1000 machines.
- Each partition is totally ordered. If you want a total order over all writes you probably want to have just one partition.
- Each partition is not consumed by more than one consumer thread/process in each consumer group. This allows to have each process consume in a single threaded fashion to guarantee ordering to the consumer within the partition (if we split up a partition of ordered messages and handed them out to multiple consumers even though the messages were stored in order they would be processed out of order at times).
- Many partitions can be consumed by a single process, though. So you can have 1000 partitions all consumed by a single process.
- Another way to say the above is that the partition count is a bound on the maximum consumer parallelism.
- More partitions will mean more files and hence can lead to smaller writes if you don't have enough memory to properly buffer the writes and coalesce them into larger writes
- Each partition corresponds to several znodes in zookeeper. Zookeeper keeps everything in memory so this can eventually get out of

hand.
- More partitions means longer leader fail-over time. Each partition can be handled quickly (milliseconds) but with thousands of partitions this can add up.
- When we checkpoint the consumer position we store one offset per partition so the more partitions the more expensive the position checkpoint is.
- It is possible to later expand the number of partitions BUT when we do so we do not attempt to reorganize the data in the topic. So if you are depending on key-based semantic partitioning in your processing you will have to manually copy data from the old low partition topic to a new higher partition topic if you later need to expand.

Note that I/O and file counts are really about #partitions/#brokers, so adding brokers will fix problems there; but zookeeper handles all partitions for the whole cluster so adding machines doesn't help.

Why do I see lots of Leader not local exceptions on the broker during controlled shutdown?

This happens when the producer clients are using num.acks=0. When the leadership for a partition is changed, the clients (producer and consumer) gets an error when they try to produce or consume from the old leader when they wait for a response. The client then refreshes the partition metadata from zookeeper and gets the new leader for the partition and retries. This does not work for the producer client when ack = 0. This is because the producer does not wait for a response and hence does not know about the leadership change. The client would end up loosing messages till the shutdown broker is brought back up. This issue is fixed in KAFKA-955

How to reduce churns in ISR? When does a broker leave the ISR ?

ISR is a set of replicas that are fully sync-ed up with the leader. In other words, every replica in ISR has all messages that are committed. In an ideal system, ISR should always include all replicas unless there is a real failure. A replica will be dropped out of ISR if it diverges from the leader. This is controlled by two parameters: replica.lag.time.max.ms and replica.lag.max.messages. The former is typically set to a value that reliably detects the failure of a broker. We have a min fetch rate JMX in the broker. If that rate is n, set the former to a value larger than 1/n * 1000. The latter is typically set to the observed max lag (a JMX bean) in the follower. Note that if replica.lag.max.messages is too large, it can increase the time to commit a message. If latency becomes a problem, you can increase the number of partitions in a topic.

If a replica constantly drops out of and rejoins isr, you may need to increase replica.lag.max.messages. If a replica stays out of ISR for a long time, it may indicate that the follower is not able to fetch data as fast as data is accumulated at the leader. You can increase the follower's fetch throughput by setting a larger value for num.replica.fetchers.

After bouncing a broker, why do I see LeaderNotAvailable or NotLeaderForPartition exceptions on startup?

If you don't use controlled shutdown, some partitions that had leaders on the broker being bounced go offline immediately. The controller takes some time to elect leaders and notify the brokers to assume the new leader role. Following this, clients take some time to send metadata requests and discover the new leaders. If the broker is stopped and restarted quickly, clients that have not discovered the new leader keep sending requests to the newly restarted broker. The exceptions are throws since the newly restarted broker is not the leader for any partition.

How to replace a failed broker?

When a broker fails, Kafka doesn't automatically re-replicate the data on the failed broker to other brokers. This is because in the common case, one brings down a broker to apply code or config changes, and will bring up the broker quickly afterward. Re-replicating the data in this case will be wasteful. In the rarer case that a broker fails completely, one will need to bring up another broker with the same broker id on a new server. The new broker will automatically replicate the missing data.

Can I add new brokers dynamically to a cluster?

Yes, new brokers can be added online to a cluster. Those new brokers won't have any data initially until either some new topics are created or some replicas are moved to them using the partition reassignment tool.

How do I accurately get offsets of messages for a certain timestamp using OffsetRequest?

Kafka allows querying offsets of messages by time and it does so at segment granularity. The timestamp parameter is the unix timestamp and querying the offset by timestamp returns the latest possible offset of the message that is appended no later than the given timestamp. There are 2 special values of the timestamp - latest and earliest. For any other value of the unix timestamp, Kafka will get the starting offset of the log segment that is created no later than the given timestamp. Due to this, and since the offset request is served only at segment granularity, the offset fetch request returns less accurate results for larger segment sizes.

For more accurate results, you may configure the log segment size based on time (log.roll.ms) instead of size (log.segment.bytes). However care

should be taken since doing so might increase the number of file handlers due to frequent log segment rolling.

Build issues

How do I get Kafka dependencies to work in Play framework?

Add the following to your build.sbt file -

```
resolvers += "Apache repo" at
"https://repository.apache.org/content/repositories/releases"
```

Sample build.sbt

```
name := "OptionsWatcher"

version := "1.0-SNAPSHOT"

scalaVersion := "2.9.3"

resolvers += "Apache repo" at
"https://repository.apache.org/content/repositories/releases"

libraryDependencies ++= Seq(
  jdbc,
  anorm,
  cache,
  "joda-time" % "joda-time" % "2.2",
    "org.joda" % "joda-convert" % "1.3.1",
   "ch.qos.logback" % "logback-classic" % "1.0.13",
   "org.mashupbots.socko" % "socko-webserver_2.9.2" % "0.2.2",
   "nl.grons" % "metrics-scala_2.9.2" % "3.0.0",
   "com.codahale.metrics" % "metrics-core" % "3.0.0",
   "io.backchat.jerkson" % "jerkson_2.9.2" % "0.7.0",
   "com.amazonaws" % "aws-java-sdk" % "1.3.8",
   "net.databinder.dispatch" %% "dispatch-core" % "0.9.5",
   "org.apache.kafka" % "kafka_2.9.2" % "0.8.0-beta1" excludeAll (
     ExclusionRule(organization = "com.sun.jdmk"),
     ExclusionRule(organization = "com.sun.jmx"),
     ExclusionRule(organization = "javax.jms"),
     ExclusionRule(organization = "org.slf4j")
  )
)
```

Unit testing

How do I write unit tests using Kafka?

First, you need to include the test stuff from Kafka. If using Maven, this does the trick:

```
<dependency>
   <groupId>org.apache.kafka</groupId>
   <artifactId>kafka_2.10</artifactId>
   <classifier>test</classifier>
   <scope>test</scope>
</dependency>
```

I used Apache Curator to get my test ZooKeeper server:

```
<dependency>
   <groupId>org.apache.curator</groupId>
   <artifactId>curator-test</artifactId>
   <scope>test</scope>
</dependency>
```

And my code looks like this:

```
import java.io.IOException;
 import java.util.Properties;

 import kafka.server.KafkaConfig;
 import kafka.server.KafkaServerStartable;
 import kafka.utils.TestUtils;

 import org.apache.curator.test.TestingServer;

 public class TestKafkaCluster {
     KafkaServerStartable kafkaServer;
     TestingServer zkServer;

     public TestKafkaCluster() throws Exception {
         zkServer = new TestingServer();
         KafkaConfig config = getKafkaConfig(zkServer.getConnectString());
         kafkaServer = new KafkaServerStartable(config);
         kafkaServer.startup();
     }

     private static KafkaConfig getKafkaConfig(final String zkConnectString) {
         scala.collection.Iterator<Properties> propsI =
             TestUtils.createBrokerConfigs(1).iterator();
         assert propsI.hasNext();
         Properties props = propsI.next();
         assert props.containsKey("zookeeper.connect");
         props.put("zookeeper.connect", zkConnectString);
         return new KafkaConfig(props);
     }

     public String getKafkaBrokerString() {
         return String.format("localhost:%d",
                 kafkaServer.serverConfig().port());
     }

     public String getZkConnectString() {
         return zkServer.getConnectString();
     }

     public int getKafkaPort() {
         return kafkaServer.serverConfig().port();
     }

     public void stop() throws IOException {
         kafkaServer.shutdown();
         zkServer.stop();
     }
 }
```