KIP-932: Queues for Kafka

Status

Current state: *Under Discussion*

Discussion thread: https://lists.apache.org/thread/9wdxthfsbm5xf01y4xvq6qtlg0gq96lq

JIRA: https://issues.apache.org/jira/browse/KAFKA-16092

Please keep the discussion on the mailing list rather than commenting on the wiki (wiki discussions get unwieldy fast).

Motivation

Apache Kafka has achieved great success as a highly scalable event-streaming platform. The way that consumer groups assign partitions to members of the group gives a powerful combination of ordering and scalability, but it does introduce coupling between the number of consumers in a consumer group and the number of partitions. Users of Kafka often have to “over-partition” simply to ensure they can have sufficient parallel consumption to cope with peak loads.

There are plenty of situations in which consumers could cooperatively consume from a stream of events without needing to be assigned exclusive access to specific topic-partitions. This, together with per-message acknowledgement and delivery counts, enables a class of use-cases traditionally built around the concept of a queue. For example, a queue is perfect for a situation in which messages are independent work items that can be processed concurrently by a pool of applications, and individually retried or acknowledged as processing completes. This is much easier to achieve using a queue rather than a partitioned topic with a consumer group.

This KIP introduces the concept of a **share group** as a way of enabling cooperative consumption using Kafka topics. It does not add the concept of a “queue” to Kafka per se, but rather that introduces cooperative consumption to accommodate these queuing use-cases using regular Kafka topics. Share groups make this possible. You can think of a share group as roughly equivalent to a “durable shared subscription” in existing systems.

This is indeed Queues for Kafka - queues done in a Kafka way, with no maximum queue depth and the ability to reset to a specific time for point-in-time recovery.

Proposed Changes

Share groups allow Kafka consumers to work together cooperatively consuming and processing the records from topics. They are an alternative to consumer groups for situations in which finer-grained sharing is required.

The fundamental differences between a share group and a consumer group are:

- The consumers in a share group cooperatively consume records with partitions that may be assigned to multiple consumers
- The number of consumers in a share group can exceed the number of partitions
- Records are acknowledged on an individual basis, although the system is optimized to work in batches for improved efficiency
- Delivery attempts to consumers in a share group are counted to enable automated handling of unprocessable records

Share groups are a new type of group, alongside the existing consumer groups, adding `"share"` to the existing group types of `"consumer"` and `"classic"` .

All consumers in the same share group subscribed to the same topic cooperatively consume the records of that topic. If a topic is accessed by consumers in more than one share group, each share group cooperatively consumes from that topic independently of the other share groups.

Each consumer can dynamically set the list of topics it wants to subscribe to. In practice, all of the consumers in a share group will usually subscribe to the same topic or topics.

When a consumer in a share-group fetches records, it receives **available** records from any of the topic-partitions that match its subscriptions. Records are **acquired** for delivery to this consumer with a time-limited acquisition lock. While a record is acquired, it is not available for another consumer. By default, the lock duration is 30s, but it can also be controlled using the group `group.share.record.lock.duration.ms` configuration parameter. The idea is that the lock is automatically released once the lock duration has elapsed, and then the record is available to be given to another consumer. The consumer which holds the lock can deal with it in the following ways:

- The consumer can **acknowledge** successful processing of the record
- The consumer can **release** the record, which makes the record available for another delivery attempt
- The consumer can **reject** the record, which indicates that the record is unprocessable and does not make the record available for another delivery attempt
- The consumer can do nothing, in which case the lock is automatically released when the lock duration has elapsed

The cluster limits the number of records acquired for consumers for each topic-partition in a share group. Once the limit is reached, fetching records will temporarily yield no further records until the number of acquired records reduces, as naturally happens when the locks time out. This limit is controlled by the broker configuration property `group.share.record.lock.partition.limit` . By limiting the duration of the acquisition lock and automatically releasing the locks, the broker ensures delivery progresses even in the presence of consumer failures.

Concepts

There are some concepts being introduced to Kafka to support share groups.

The **group coordinator** is now responsible for coordination of share groups as well as consumer groups. The responsibility for being coordinator for the cluster's share groups is distributed among the brokers, exactly as for consumer groups. For share groups, the group coordinator has the following responsibilities:

- It maintains the list of share-group members.
- It manages the topic-partition assignments for the share-group members using a server-side partition assignor. An initial, trivial implementation would be to give each member the list of all topic-partitions which matches its subscriptions and then use the pull-based protocol to fetch records from all partitions. A more sophisticated implementation could use topic-partition load and lag metrics to distribute partitions among the consumers as a kind of autonomous, self-balancing partition assignment, steering more consumers to busier partitions, for example. Alternatively, a push-based fetching scheme could be used.

A **share-partition** is a topic-partition with a subscription in a share group. For a topic-partition subscribed in more than one share group, each share group has its own share-partition.

A **share-partition leader** is a component of the broker which manages the share-group's view of a topic-partition. It is co-located with the topic-partition leader, and the leadership of a share-partition follows the leadership of the topic-partition. The share-partition leader has the following responsibilities:

- It fetches the records from the replica manager from the local replica
- It manages and persists the states of the in-flight records

This means that the fetch-from-follower optimization is not supported by share-groups. The KIP does however include rack information so that consumers could preferentially fetch from share-partitions whose leadership is in the same rack.

Relationship with consumer groups

Consumer groups and share groups exist in the same namespace in a Kafka cluster. As a result, if there's a consumer group with a particular name, you cannot create a share group with the same name, and vice versa. But consumer groups and share groups are quite different in terms of use, so attempts to perform operations for one kind of group on a group of the incorrect type will fail with a `GroupIdNotFoundException` . The new `AdminClient.listGroups` method gives a way of listing groups of all types.

Because consumer groups and share groups are both created automatically on first use, the type of group that is created depends upon how the group name was first used. As a result, it is helpful to be able to ensure that a group of a particular name can only be created with a particular type. This is achieved by defining a group configuration property `group.type` , using the `kafka-configs.sh` tool or the `AdminClient.incrementalAlterConfigs` method. For example, you could use the following command to ensure that the group ID `"G1"` is to be used for a share group only.

```
$ bin/kafka-configs.sh --bootstrap-server localhost:9092 --entity-name group --entity-name G1 --alter --add-
config group.type=share
```

If a regular Kafka consumer then attempts to use "`G1`" as a consumer group, the exception "`InconsistentGroupProtocolException`" will be thrown.

Share group membership

This KIP builds upon the new consumer group protocol in KIP-848: The Next Generation of the Consumer Rebalance Protocol.

Share group membership is controlled by the group coordinator. Consumers in a share group use the heartbeat mechanism to join, leave and confirm continued membership of the share group, using the new `ShareGroupHeartbeat` RPC. Share-partition assignment is also piggybacked on the heartbeat mechanism. Share groups only support server-side assignors, which implement the new internal `org.apache.kafka.coordinator.group.assignor.SharePartitionAssignor` interface.

This KIP introduces just one assignor, `org.apache.kafka.coordinator.group.assignor.SimpleShareAssignor` , which assigns all partitions of all subscribed topics to all members. In the future, a more sophisticated share group assignor could balance the number of consumers assigned to the partitions, and it may well revoke partitions from existing members in order to improve the balance. The simple assignor isn't that smart.

For a share group, a rebalance is a much less significant event than for a consumer group because there's no fencing. When a partition is assigned to a member of a share group, it's telling the member that it should fetch records from that partition, which it may well be sharing with the other members of the share group. The members are not aware of each other, and there's no synchronization barrier or fencing involved. The group coordinator, using the server-side assignor, is responsible for telling the members which partitions they are assigned and revoked. But the aim is to give every member useful work, rather than to keep the members' assignments safely separated.

For a share group, the group coordinator does not need to persist the assignments, but it does need to persist the assignment epoch so that it doesn't move backwards if the group coordinator changes.

The reconciliation process for a share group is very simple because there is no fencing - the group coordinator revokes the partitions which are no longer in the target assignment of the member and assigns the new partitions to the member at the same time. There's no need for the revocations to be acknowledged before new partitions are assigned. The member acknowledges changes to its assignment, but the group coordinator does not depend upon receiving the acknowledgement to proceed.

Data model

This is the data model maintained by the group coordinator for share groups.

Share Group and Member

The group and members represent the current state of a share group. This is reminiscent of a simplified consumer group.

Share Group		
Name	**Type**	**Description**
Group ID	string	The group ID as configured by the consumer. The ID uniquely identifies the group.
Group Epoch	int32	The current epoch of the group. The epoch is incremented by the group coordinator when a new assignment is required for the group.
Server Assignore	string	The server-side assignor used by the group.
Members	[]Member	The set of members in the group.
Partitions Metadata	[]PartitionMetadata	The metadata of the partitions that the group is subscribed to. This is used to detect partition metadata changes.
Member		
Name	**Type**	**Description**
Member ID	string	The unique identifier of the member. It is generated by the coordinator upon the first heartbeat request and must be used throughout the lifetime of the member.
Rack ID	string	The rack ID configured by the consumer.
Client ID	string	The client ID configured by the consumer.
Client Host	string	The client host of the consumer.
Subscribed Topic Names	[]string	The current set of subscribed topic names configured by the consumer.

Target Assignment

The target assignment of the group. This represents the assignment that all the members of the group will eventually converge to. It is a declarative assignment which is generated by the assignor based on the group state.

Target Assignment		
Name	**Type**	**Description**
Group ID	string	The group ID as configured by the consumer. The ID uniquely identifies the group.
Assignment Epoch	int32	The epoch of the assignment. It represents the epoch of the group used to generate the assignment. It will eventually match the group epoch.
Assignment Error	int8	The error reported by the assignor.
Members	[]Member	The assignment for each member.
Member		
Name	**Type**	**Description**
Member ID	string	The unique identifier of the member.
Partitions	[] TopicIdPartition	The set of partitions assigned to this member.
Metadata	bytes	The metadata assigned to this member.

Current Assignment

The current assignment of a member.

Current Assignment		
Name	**Type**	**Description**
Group ID	string	The group ID as configured by the consumer. The ID uniquely identifies the group.
Member ID	string	The member ID of this member.
Member Epoch	int32	The current epoch of this member. The epoch is the assignment epoch of the assignment currently used by this member.
Error	int8	The error reported by the assignor.
Partitions	[] TopicIdPartition	The current partitions used by the member.
Version	int16	The version used to encode the metadata.
Metadata	bytes	The current metadata used by the member.

Rebalance process

The rebalance process is driven by the group coordinator and revolves around three kinds of epochs: the group epoch, the assignment epoch and the member epoch. This is intentionally very similar to how the process works for consumer groups in KIP-848.

Group epoch - Trigger a rebalance

The group coordinator is responsible for triggering a rebalance of the group when the metadata of the group changes. The group epoch represents the generation of the group metadata. It is incremented whenever the group metadata is updated. This happens in the following cases:

- A member joins or leaves the group.
- A member updates its subscriptions.
- A member is removed from the group by the group coordinator.
- The partition metadata is updated, such as when a new partition is added or a topic is created or deleted.

In all these cases, a new version of the group metadata is calculated by the group coordinator with an incremented group epoch. **For a share group, the group coordinator does not persist the group metadata.** The new version of the group metadata signals that a new assignment is required for the group.

Assignment epoch - Compute the group assignment

Whenever the group epoch is larger that the target assignment epoch, the group coordinator triggers the computation of a new target assignment based on the latest group metadata using a server-side assignor. For a share group, the group coordinator does not persist the assignment. The assignment epoch becomes the group epoch of the group metadata used to compute the assignment.

Member epoch - Reconciliation of the group

Each member independently reconciles its current assignment with its new target assignment, converging with the target epoch and assignment.

The group coordinator revokes the partitions which are no longer in the target assignment of the member, and assigns the partitions which have been added to the target assignment of the member. It provides the new assignment to the member in its heartbeat response until the member acknowledges the assignment change in a heartbeat request.

By assigning and revoking partitions for the members of the group, the group coordinator can balance the partitions across the members of the group.

The member provides the rebalance timeout to the group coordinator when it joins the group. This is the timeout for the group coordinator waiting for the member to acknowledge that it has adopted the target assignment. If the member does not confirm the target assignment within the rebalance timeout, the group coordinator removes the member from the group.

Member ID

Every member is uniquely identified by the UUID called the member ID. This UUID is generated by the group coordinator and given to the member when it joins the group. It is used in all communication with the group coordinator and must be kept during the entire lifespan of the member.

Heartbeat and session

The member uses the new `ShareGroupHeartbeat` API to establish a session with the group coordinator. The member is expected to heartbeat every `group.share.heartbeat.interval.ms` in order to keep its session opened. If it does not heartbeat at least once within the `group.share.session.timeout.ms`, the group coordinator will remove the member from the group. The member is told the heartbeat interval in the response to the `ShareGroupHeartbeat` API.

If a member is removed from the group because it fails to heartbeat, because there’s intentionally no fencing, at the protocol level, the consumer does not lose the ability to fetch and acknowledge records. A failure to heartbeat is most likely because the consumer has died. If the consumer just failed to heartbeat due to a temporary pause, it could in theory continue to fetch and acknowledge records. When it finally sends a heartbeat and realises it’s been kicked out of the group, it should stop fetching records because its assignment has been revoked, and rejoin the group.

Static membership

Share groups do not support static membership. Because the membership of a share group is more ephemeral, there’s less advantage to maintaining an assignment when a member has temporarily left but will rejoin within the session timeout.

Share group states

Share groups do not have the ASSIGNING state because only server-side assignors are supported, and do not need the RECONCILING state because there’s no need for all members to converge before the group enters the STABLE state.

- **EMPTY** - When a share group is created or the last member leaves the group, the share group is EMPTY.
- **STABLE** - When a share group has active members, the share group is STABLE.
- **DEAD** - When the share group remains EMPTY for a configured period, the group coordinator transitions it to DEAD to delete it.

Persistence and fail-over

For a share group, the group coordinator only persists a single record which essentially reserves the group's ID as a share group in the namespace of groups.

When the group coordinator fails over, the newly elected coordinator loads the state from the `__consumer_offsets` partition. This means a share group will remain in existence across the fail-over. However, the members of the groups and their assignments are not persisted. This means that existing members will have to rejoin the share group following a coordinator failover.

In-flight records

For each share-partition, the share group adds some state management for the records being consumed. The starting offset of records which are eligible for consumption is known as the **share-partition start offset** (SPSO), and the last offset of records which are eligible for consumption is known as the **share-partition end offset** (SPEO). The records between starting at the SPSO and up to the SPEO are known as the **in-flight records**. So, a share-partition is essentially managing the consumption of the in-flight records.

The SPEO is not necessarily always at the end of the topic-partition and it just advances freely as records are fetched beyond this point. The segment of the topic-partition between the SPSO and the SPEO is a sliding window that moves as records are consumed. The share-partition leader limits the distance between the SPSO and the SPEO. The upper bound is controlled by the broker configuration `group.share.record.lock.partition.limit`. Unlike existing queuing systems, there’s no “maximum queue depth”, but there is a limit to the number of in-flight records at any point in time.

The records in a share-partition are in one of four states:

State	Description
Available	The record is available for a consumer
Acquired	The record has been acquired for a specific consumer, with a time-limited acquisition lock
Acknowledged	The record has been processed and acknowledged by a consumer
Archived	The record is not available for a consumer

All records before the SPSO are in **Archived** state. All records after the SPEO are in **Available** state, but not yet being delivered to consumers.

The records also have a **delivery count** in order to prevent unprocessable records being endlessly delivered to consumers. If a record is repeatedly causing exceptions during its processing, it is likely that it is a “poison message”, perhaps with a formatting or semantic error. Every time that a record is acquired by a consumer in a share group, its delivery count increments by 1. The first time a record is acquired, its delivery count is 1.

The state transitions look like this:

```
+--------------+
|  Available   |<------------------+
+--------------+                   |
       |                           |
       | acquired for consumer     | - if (delivery count < group.share.delivery.attempt.limit)
       | (delivery count++)        |     - released by consumer
       |                           |     - acquisition lock elapsed
       V                           |
+--------------+                   |
|   Acquired   |-------------------+
+--------------+                   |
       |                           |
       |                           | - if (delivery count == group.share.delivery.attempt.limit)
       | accepted by consumer      |     - released by consumer
       |                           |     - acquisition lock elapsed
       V                           | OR
+--------------+                   | - rejected by consumer as unprocessable
| Acknowledged |                   |
+--------------+                   |
       |                           |
       |                           |
       | SPSO moves past record    |
       |                           |
       V                           |
+--------------+                   |
|   Archived   |<------------------+
+--------------+
```

The share-partition leader persists the states and delivery counts. These updates are not performed with exactly-once semantics, so the delivery count cannot be relied upon to be precise in all situations. It is intended as a way to protect against poison messages, rather than a precise count of the number of times a record is delivered to a consumer.

When records are fetched for a consumer, the share-partition leader starts at the SPSO and finds **Available** records. For each record it finds, it moves it into **Acquired** state, bumps its delivery count and adds it to a batch of acquired records to return to the consumer. The consumer then processes the records and acknowledges their consumption. The delivery attempt completes successfully and the records move into **Acknowledged** state.

Alternatively, if the consumer cannot process a record or its acquisition lock elapses, the delivery attempt completes unsuccessfully and the record’s next state depends on the delivery count. If the delivery count has reached the cluster’s share delivery attempt limit (5 by default), the record moves into **Archived** state and is not eligible for additional delivery attempts. If the delivery count has not reached the limit, the record moves back into **Available** state and can be delivered again.

This means that the delivery behavior is at-least-once.

Ordering

Share groups focus primarily on sharing to allow consumers to be scaled independently of partitions. The records in a share-partition can be delivered out of order to a consumer, in particular when redeliveries occur.

For example, imagine two consumers in a share group consuming from a single-partition topic. The first consumer fetches records 100 to 109 inclusive and then crashes. At the same time, the second consumer fetches, processes and acknowledges records 110 to 119. When the second consumer fetches again, it gets records 100 to 109 with their delivery counts set to 2 because they are being redelivered. That’s exactly what you want, but the offsets do not necessarily increase monotonically in the same way as they do for a consumer group.

The records returned in a batch for particular share-partition are guaranteed to be in order of increasing offset. There are no guarantees about the ordering of offsets between different batches.

Managing the SPSO and SPEO

The consumer group concepts of seeking and position do not apply to share groups. The SPSO for each share-partition can be initialized for an empty share group and the SPEO naturally moves forwards as records are consumed.

When a topic subscription is added to a share group for the first time, the SPSO is initialized for each share-partition. By default, the SPSO for each share-partition is initialized to the latest offset for the corresponding topic-partitions.

Alternatively, there is an administrative action available using either `AdminClient.alterShareGroupOffsets` or the `kafka-share-groups.sh` tool to reset the SPSO for an empty share group with no active members. This can be used to "reset" a share group to the start of a topic, a particular timestamp or the end of a topic. It can also be used to initialize the share group to the start of a topic. Resetting the SPSO discards all of the in-flight record state and delivery counts.

For example, to start using a share group S1 to consume for the first time from the earliest offset of a topic T1, you could use:

```
$ kafka-share-groups.sh --bootstrap-server localhost:9092 --group S1 --topic T1 --reset-offsets --to-earliest --
execute
```

If the number of partitions is increased for a topic with a subscription in a share group, the SPSO for the newly created share-partitions is initialized to 0 (which is of course both the earliest and latest offset for an empty topic-partition). This means there is no doubt about what happens when the number of partitions is increased.

If the SPSO is reset to an offset that has been tiered to remote storage (KIP-405: Kafka Tiered Storage), there will be a performance impact just as for existing consumers fetching records from remote storage.

Log retention

The SPSO for each share-partition is bounded by the log start offset (LSO) of the topic-partition, which is itself managed by the retention policy.

If log segments are being retained based on time, when an inactive log segment's age exceeds the configured time, the LSO advances to the start of the next log segment and the old segment is deleted. If the SPSO is within the log segment that is deleted, it will also advance to the next log segment. This is roughly equivalent to message-based expiration in other messaging systems.

If log segments are being retained based on size, when the log exceeds the configured size, the LSO advances to the start of the next log segment and the old segment is deleted. If the SPSO is within the log segment that is deleted, it will also advance to the next log segment. This keeps control of the space consumed by the log, but it does potentially silently remove records that were eligible for delivery. When using share groups with log retention based on size, it is important to bear this in mind.

When the SPSO advances because of the LSO moving, the in-flight records past which the SPSO moves logically move into **Archived** state. The exception is that records which are already **Acquired** for delivery to consumers can be acknowledged with any `AcknowledgeType`, at which point they logically transition into **Archived** state too; there's no need to throw an exception for a consumer which has just processed a record which is about to become **Archived**.

Note that because the share groups are all consuming from the same log, the retention behavior for a topic applies to all of the share groups consuming from that topic.

Log compaction

When share groups are consuming from compacted topics, there is the possibility that in-flight records are cleaned while being consumed. In this case, the delivery flow for these records continues as normal because the disappearance of the cleaned records will only be discovered when they are next fetched from the log. This is analogous to a consumer group reading from a compacted topic - records which have been fetched by the consumer can continue to be processed, but if the consumer tried to fetch them again, it would discover they were no longer there.

When fetching records from a compacted topic, it is possible that record batches fetched have offset gaps which correspond to records the log cleaner removed. This simple results in gaps of the range of offsets of the in-flight records.

Reading transactional records

Each consumer in a consumer group has its own isolation level which controls how it handles records which were produced in transactions. For a share group, the concept of isolation level applies to the entire group, not each consumer.

The isolation level of a share group is controlled by the group configuration `group.share.isolation.level`.

For the `read_uncommitted` isolation level, which is the default, the share group consumes all transactional and non-transactional records.

For the `read_committed` isolation level, the share group only consumes committed records. The share-partition leader itself is responsible for keeping track of the commit and abort markers and filtering out transactional records which have been aborted. So, the set of records which are eligible to become in-flight records are non-transactional records and committed transactional records only. The SPEO can only move up to the last stable offset.

This processing has to occur on the broker because none of the clients receives all of the records. It can be performed with shallow iteration of the log.

In-flight records example

An example of a share-partition showing the states looks like this:

```
+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+
|   0   |   1   |   2   |   3   |   4   |   5   |   6   |   7   |   8   |   9   |  ...  | <- offset
| Archv | Archv | Acqrd | Avail | Acqrd | Acked | Archv | Avail | Avail | Avail | Avail | <- state
|       |       |   1   |   2   |   1   |       |       |       |       |       |       | <- delivery count
+-------+-------+---^---+-------+-------+-------+-------+-------+-------+---^---+-------+
                    |                                                       |
                    +-- Share-partition start offset (SPSO)                 +-- Share-partition end offset
(SPEO)
```

The share group is currently managing the consumption of the in-flight records, which have offsets 2 to 8 inclusive.

- All records earlier than offset 2 are in **Archived** state and are not in-flight
- Records 2 and 4 have been acquired for consumption by a consumer, and their delivery counts have been incremented to 1
- Record 3 has previously been acquired twice for consumption by a consumer, but went back into **Available** state
- Record 5 has been acknowledged
- Record 6 has previously been acquired for consumption by a consumer, was rejected because it cannot be processed, and is in **Archived** state
- Records 7 and 8 are available for consumption by a consumer
- All records starting with offset 9 and later are in **Available** state

The cluster records this information durably. In this example, the durable state contains the SPSO position, the non-zero delivery count for offset 3, the **Acknowledged** state of offset 5, and the **Archived** state of offset 6.

Batching

Cooperative consumption is inherently record-based, but the expectation is that batching is used to maximise performance. For example, in the case where all records in a batch are processed successfully:

- When a consumer fetches records, the share-partition leader prefers to return complete record batches.
- In the usual and optimal case, all of the records in a batch will be in **Available** state and can all be moved to **Acquired** state with the same acquisition lock time-out.
- When the consumer has processed the fetched records, it can acknowledge delivery of all of the records as a single batch, transitioning them all into **Acknowledged** state.

So, when a bunch of consumers are cooperatively consumed from a topic using a share group, the natural unit of sharing is the record batch. The processing loop is roughly:

- Fetch record batch
- Process records
- Acknowledge all records in batch

In the situation where some records in a batch have been released or rejected separately, subsequent fetches of those records are more likely to have gaps.

Fetching and acknowledging records

Share groups introduce two new APIs in the Kafka protocol for fetching and acknowledging records.

- `ShareFetch` for fetching records from share-partition leaders
- `ShareAcknowledge` for acknowledging delivery with share-partition leaders

The `ShareFetch` API works very much like incremental fetch using a concept called a **share session**. Each share session contains a set of topic-partitions which are managed in the share-partition leaders. The share-partition leader manages the fetching of records and the in-flight record state for its share-partitions. The consumer adds and removes topic-partitions from its share session using the `ShareFetch` API just like the `Fetch` API is used for incremental fetch. With the `Fetch` API, the consumer specifies the fetch offset. With the `ShareFetch` API, the consumer just fetches records and the share-partition leader decides which records to return.

In order to ensure no share-partitions are starved from records being fetched, the share-partition leader rotates the order of share-partitions for which it returns partition information. This ensures that it eventually returns data about all partitions for which data is available.

When a batch of records is first read from the log and added to the in-flight records for a share-partition, the broker does not know whether the set of records between the batch's base offset and the last offset contains any gaps, as might occur for example as a result of log compaction. When the broker does not know which offsets correspond to records, the batch is considered an **unmaterialized record batch**. Rather than forcing the broker to iterate through all of the records in all cases, which might require decompressing every batch, the broker can send unmaterialized record batches to consumers. It initially assumes that all offsets between the base offset and the last offset correspond to records. When the consumer processes the batch, it may find gaps and it reports these using the `ShareAcknowledge` API. This means that the presence of unmaterialized record batches containing gaps might temporarily inflate the number of in-flight records, but this will be resolved by the consumer acknowledgements.

Client programming interface

A new interface `KafkaShareConsumer` is introduced for consuming from share groups. It looks very similar to `KafkaConsumer` trimmed down to the methods that apply to share groups.

In order to retain similarity with `KafkaConsumer` and make it easy for applications to move between the two interface, `KafkaShareConsumer` follows the same threading rules as `KafkaConsumer` . It is not thread-safe and only one thread at a time may called the methods of `KafkaShareConsumer` . Unsynchronized access will result in `ConcurrentModificationException` . The only exception to this rule is `KafkaShareConsumer.wakeup()` which may be called from any thread.

To join a share group, the client application instantiates a `KafkaShareConsumer` using the configuration parameter `group.id` to give the ID of the share group. Then, it uses `KafkaShareConsumer.subscribe(Collection<String> topics)` to provide the list of topics that it wishes to consume from. The consumer is not allowed to assign partitions itself.

Each call to `KafkaShareConsumer.poll(Duration)` fetches data from any of the topic-partitions for the topics to which it subscribed. It returns a set of in-flight records acquired for this consumer for the duration of the acquisition lock timeout. For efficiency, the consumer preferentially returns complete record sets with no gaps. The application then processes the records and acknowledges their delivery, either using explicit or implicit acknowledgement.

If the application calls the `KafkaShareConsumer.acknowledge(ConsumerRecord, AcknowledgeType)` method for any record in the batch, it is using explicit acknowledgement. The calls to `KafkaShareConsumer.acknowledge(ConsumerRecord, AcknowledgeType)` must be issued in the order in which the records appear in the `ConsumerRecords` object, which will be in order of increasing offset for each share-partition. In this case:

- The application calls `KafkaShareConsumer.commitSync/Async()` which commits the acknowledgements to Kafka. If any records in the batch were not acknowledged, they remain acquired and will be presented to the application in response to a future poll.
- The application calls `KafkaShareConsumer.poll(Duration)` without committing first, which commits the acknowledgements to Kafka asynchronously. In this case, no exception is thrown by a failure to commit the acknowledgement. If any records in the batch were not acknowledged, they remain acquired and will be presented to the application in response to a future poll.
- The application calls `KafkaShareConsumer.close()` which attempts to commit any pending acknowledgements and releases any remaining acquired records.

If the application does not call `KafkaShareConsumer.acknowledge(ConsumerRecord, AcknowledgeType)` for any record in the batch, it is using implicit acknowledgement. In this case:

- The application calls `KafkaShareConsumer.commitSync/Async()` which implicitly acknowledges all of the delivered records as processed successfully and commits the acknowledgements to Kafka.
- The application calls `KafkaShareConsumer.poll(Duration)` without committing, which also implicitly acknowledges all of the delivered records and commits the acknowledgements to Kafka asynchronously. In this case, no exception is thrown by a failure to commit the acknowledgements.
- The application calls `KafkaShareConsumer.close()` which releases any acquired records without acknowledgement.

The `KafkaShareConsumer` guarantees that the records returned in the `ConsumerRecords` object for a specific share-partition are in order of increasing offset. For each share-partition, the share-partition leader guarantees that acknowledgements for the records in a batch are performed atomically. This makes error handling significantly more straightforward because there can be one error code per share-partition.

When the share-partition leader receives a request to acknowledge delivery, which can occur as a separate RPC or piggybacked on a request to fetch more records, it checks that the records being acknowledged are still in the **Acquired** state and acquired by the share group member trying to acknowledge them. If a record had reached its acquisition lock timeout and reverted to **Available** state, the attempt to acknowledge it will fail with `org.apache.kafka.common.errors.TimeoutException`, but the record may well be re-acquired for the same consumer and returned to it again.

Acknowledge commit callback

Acknowledgements errors are delivered to a new kind of callback called an **acknowledge commit callback** which can optionally be registered with a `KafkaShareConsumer.wakeup()` .

- If the application uses `KafkaShareConsumer.commitSync()` to commit its acknowledgements, the results of the acknowledgements is returned to the application
- If the application uses `KafkaShareConsumer.commitAsync()` or `KafkaShareConsumer.poll(Duration)` to commit its acknowledgements, the results of the acknowledgements are only delivered if there is an acknowledge commit callback registered.

The acknowledge commit callback is called on the application thread and it is not permitted to call the methods of `KafkaShareConsumer` with the exception of `KafkaShareConsumer.wakeup()` .

Example - Acknowledging a batch of records (implicit acknowledgement)

In this example, a consumer using share group `"myshare"` subscribes to topic `"foo"`. It processes all of the records in the batch and then calls `KafkaShareConsumer.commitSync()` which implicitly marks all of the records in the batch as successfully consumed and commits the acknowledgement synchronously with Kafka. Asynchronous commit would also be acceptable.

```
Properties props = new Properties();
props.setProperty("bootstrap.servers", "localhost:9092");
props.setProperty("group.id", "myshare");

KafkaShareConsumer<String, String> consumer = new KafkaShareConsumer<>(props, new StringDeserializer(), new
StringDeserializer());
consumer.subscribe(Arrays.asList("foo"));
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));    // Returns a batch of
acquired records
    for (ConsumerRecord<String, String> record : records) {
        doProcessing(record);
    }
    consumer.commitSync();                                                              // Commit the
acknowledgement of all the records in the batch
}
```

Behind the scenes, the `KafkaShareConsumer` fetches records from the share-partition leader. The leader selects the records in **Available** state, and will return complete record batches (https://kafka.apache.org/documentation/#recordbatch) if possible. It moves the records into **Acquired** state, increments the delivery count, starts the acquisition lock timeout, and returns them to the `KafkaShareConsumer` . Then the `KafkaShareConsumer` keeps a map of the state of the records it has fetched and returns a batch to the application.

When the application calls `KafkaShareConsumer.commitSync()`, the `KafkaConsumer` updates the state map by marking all of the records in the batch as **Acknowledged** and it then commits the acknowledgements by sending the new state information to the share-partition leader. For each share-partition, the share-partition leader updates the record states atomically.

Example - Per-record acknowledgement (explicit acknowledgement)

In this example, the application uses the result of processing the records to acknowledge or reject the records in the batch.

```
Properties props = new Properties();
props.setProperty("bootstrap.servers", "localhost:9092");
props.setProperty("group.id", "myshare");

KafkaShareConsumer<String, String> consumer = new KafkaShareConsumer<>(props, new StringDeserializer(), new
StringDeserializer());
consumer.subscribe(Arrays.asList("foo"));
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));    // Returns a batch of
acquired records
    for (ConsumerRecord<String, String> record : records) {
        try {
            doProcessing(record);
            consumer.acknowledge(record, AcknowledgeType.ACCEPT);                       // Mark the record as
processed successfully
        } catch (Exception e) {
            consumer.acknowledge(record, AcknowledgeType.REJECT);                       // Mark the record as
unprocessable
        }
    }
    consumer.commitAsync();                                                             // Commit the
acknowledgements of all the records in the batch
}
```

In this example, each record processed is separately acknowledged using a call to the new `KafkaShareConsumer.acknowledge(ConsumerRecord, AcknowledgeType)` method. The `AcknowledgeType` argument indicates whether the record was processed successfully or not. In this case, the bad records are rejected meaning that they're not eligible for further delivery attempts. For a permanent error such as a deserialization error, this is appropriate. For a transient error which might not affect a subsequent processing attempt, the `AcknowledgeType.RELEASE` is more appropriate because the record remains eligible for further delivery attempts.

The calls to `KafkaShareConsumer.acknowledge(ConsumerRecord, AcknowledgeType)` are simply updating the state map in the `KafkaConsume` . It is only once `KafkaShareConsumer.commitAsync()` is called that the acknowledgements are committed by sending the new state information to the share-partition leader.

Example - Per-record acknowledgement, ending processing of the batch on an error (explicit acknowledgement)

In this example, the application stops processing the batch when it encounters an exception.

```
Properties props = new Properties();
props.setProperty("bootstrap.servers", "localhost:9092");
props.setProperty("group.id", "myshare");

KafkaShareConsumer<String, String> consumer = new KafkaShareConsumer<>(props, new StringDeserializer(), new
StringDeserializer());
consumer.subscribe(Arrays.asList("foo"));
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));    // Returns a batch of
acquired records
    for (ConsumerRecord<String, String> record : records) {
        try {
            doProcessing(record);
            consumer.acknowledge(record, AcknowledgeType.ACCEPT);                       // Mark the record as
processed successfully
        } catch (Exception e) {
            consumer.acknowledge(record, AcknowledgeType.REJECT);                       // Mark this record as
unprocessable
            break;
        }
    }
    consumer.commitAsync();                                                             // Commit the
acknowledgements of the acknowledged records only
}
```

There are the following cases in this example:

1. The batch contains no records, in which case the application just polls again. The call to `KafkaShareConsumer.commitAsync()` just does nothing because the batch was empty.
2. All of the records in the batch are processed successfully. The calls to `KafkaShareConsumer.acknowledge(ConsumerRecord, AcknowledgeType.ACCEPT)` marks all records in the batch as successfully processed.
3. One of the records encounters an exception. The call to `KafkaShareConsumer.acknowledge(ConsumerRecord, AcknowledgeType.REJECT)` rejects that record. Earlier records in the batch have already been marked as successfully processed. The call to `KafkaShareConsumer.commitAsync()` commits the acknowledgements, but the records after the failed record remain **Acquired** as part of the same delivery attempt and will be presented to the application in response to another poll.

Access control

Share group access control is performed on the `GROUP` resource type, just the same as consumer groups, with the same rules for the actions checked. A share group is just a new kind of group.

- Operations which read information about a share group need permission to perform the `DESCRIBE` action on the named group resource
- Operations which change information about a share group (such as consuming a record) need permission to perform the `READ` action on the named group resource

Managing durable share-partition state

The share-partition leader is responsible for recording the durable state for the share-partitions it leads. For each share-partition, we need to be able to recover:

- The Share-Partition Start Offset (SPSO)
- The state of the in-flight records
- The delivery counts of records whose delivery failed

The delivery counts are only maintained approximately and the **Acquired** state is not persisted. This minimises the amount of share-partition state that has to be logged. The expectation is that most records will be fetched and acknowledged in batches with only one delivery attempt.

Examples

Operation	State changes	Cumulative state
Starting state of topic-partition with latest offset 100	SPSO=100, SPEO=100	SPSO=100, SPEO=100
In the batched case with successful processing, there’s a state change per batch to move the SPSO forwards		
Fetch records 100-109	SPEO=110, records 100-109 (acquired, delivery count 1)	SPSO=100, SPEO=110, records 100-109 (acquired, delivery count 1)

Acknowledge 100-109	SPSO=110	SPSO=110, SPEO=110
With a messier sequence of release and acknowledge, there’s a state change for each operation which can act on multiple records		
Fetch records 110-119 Consumer 1 gets 110-112, consumer 2 gets 113-118, consumer 3 gets 119	SPEO=120, records 110-119 (acquired, delivery count 1)	SPSO=110, SPEO=120, records 110-119 (acquired, delivery count 1)
Release 110 (consumer 1)	record 110 (available, delivery count 1)	SPSO=110, SPEO=120, record 110 (available, delivery count 1), records 111-119 (acquired, delivery count 1)
Acknowledge 119 (consumer 3)	record 110 (available, delivery count 1), records 111-118 acquired, record 119 acknowledged	SPSO=110, SPEO=120, record 110 (available, delivery count 1), records 111-118 (acquired, delivery count 1), record 119 acknowledged
Fetch records 110, 120 (consumer 1)	SPEO=121, record 110 (acquired, delivery count 2), record 120 (acquired, delivery count 1)	SPSO=110, SPEO=121, record 110 (acquired, delivery count 2), records 111-118 (acquired, delivery count 1), record 119 acknowledged, record 120 (acquired, delivery count 1)
Lock timeout elapsed 111, 112 (consumer 1's records)	records 111-112 (available, delivery count 1)	SPSO=110, SPEO=121, record 110 (acquired, delivery count 2), records 111-112 (available, delivery count 1), records 113-118 (acquired, delivery count 1), record 119 acknowledged, record 120 (acquired, delivery count 1)
Acknowledge 113-118 (consumer 2)	records 113-118 acknowledged	SPSO=110, SPEO=121, record 110 (acquired, delivery count 2), records 111-112 (available, delivery count 1), records 113-119 acknowledged, record 120 (acquired, delivery count 1)
Fetch records 111,112 (consumer 3)	records 111-112 (acquired, delivery count 2)	SPSO=110, SPEO=121, record 110-112 (acquired, delivery count 2), records 113-119 acknowledged, record 120 (acquired, delivery count 1)
Acknowledge 110 (consumer 1)	SPSO=111	SPSO=111, SPEO=121, record 111-112 (acquired, delivery count 2), records 113-119 acknowledged, record 120 (acquired, delivery count 1)
Acknowledge 111,112 (consumer 3)	SPSO=120	SPSO=120, SPEO=121, record 120 (acquired, delivery count 1)

Control records

The durable share-partition state is recorded using control records, in a similar way to the transaction markers introduced in KIP-98 - Exactly Once Delivery and Transactional Messaging. These control records are written onto the topic-partition whose delivery they reflect. This is important for performance reasons because it means the share-partition leader is able to read and write them directly on the topic-partition for which it is of course also the leader.

Two new control record types are introduced: **SHARE_CHECKPOINT** (5) and **SHARE_DELTA** (6). They are written into separate message sets with the Control flag set. This flag indicates that the records are not intended for application consumption. Indeed, these message sets are not returned to any consumers at all since they are just intended for the share-partition leader.

When a control record is written as a result of an operation such as a `ShareAcknowledge` RPC, the control record must be written and fully replicated before the RPC response is sent.

SHARE_CHECKPOINT

A **SHARE_CHECKPOINT** record contains a complete checkpoint of the share-partition state. It contains:

- The group ID
- The checkpoint epoch, which is an integer that increments with each **SHARE_CHECKPOINT**
- The SPSO
- The SPEO
- An array of `[BaseOffset, LastOffset, State, DeliveryCount]` tuples where each tuple contains information for a sequence of records with the same state and delivery count

Here are some examples of how the cumulative state from the previous table would be represented in **SHARE_CHECKPOINT** records:

Cumulative state	SHARE_CHECKPOINT

SPSO=100, SPEO=100	{
	  "GroupId": "G1",
	  "CheckpointEpoch": 1,
	  "StartOffset": 100,
	  "EndOffset": 100,
	  "States": []
	}

SPSO=110, SPEO=121, record 110 (acquired, delivery count 2), records 111-112 (available, delivery count 1), records 113-118 (acquired, delivery count 1), record 119 acknowledged, record 120 (acquired, delivery count 1)	{
	  "GroupId": "G1",
	  "CheckpointEpoch": 1,
	  "StartOffset": 110,
	  "EndOffset": 121,
	  "States": [
	    {
	      "BaseOffset": 110,
	      "LastOffset": 110,
	      "State": 0 (Available),
	      "DeliveryCount": 1
	    },
	    {
	      "BaseOffset": 111,
	      "LastOffset": 112,
	      "State": 0 (Available),
	      "DeliveryCount": 1
	    },
	    {
	      "BaseOffset": 113,
	      "LastOffset": 118,
	      "State": 0 (Available),
	      "DeliveryCount": 0
	    },
	    {
	      "BaseOffset": 119,
	      "LastOffset": 119,
	      "State": 2 (Acknowledged),
	      "DeliveryCount": 1 (whatever it was when it was acknowledged)
	    },
	    {
	      "BaseOffset": 120,
	      "LastOffset": 120,
	      "State": 0 (Available),
	      "DeliveryCount": 0
	    }
	  ]
	}

Note that the **Acquired** state is not recorded because it’s transient. As a result, an **Acquired** record with a delivery count of 1 is recorded as **Available** with a delivery count of 0. In the unlikely event of a share-partition leader crash, memory of the in-flight delivery will be lost.

SHARE_DELTA

A SHARE_DELTA record contains a partial update to the share-partition state. It contains:

- The group ID
- The checkpoint epoch of the SHARE_CHECKPOINT it applies to
- An array of `[BaseOffset, LastOffset, State, DeliveryCount]` tuples

Examples with control records

Here are the previous examples, showing the control records which record the cumulative state durably. Note that any **SHARE_DELTA** could be replaced with a **SHARE_CHECKPOINT**. This example omits the details about consumer instances.

<table>
<tr><th>Operation</th><th>State changes</th><th>Cumulative state</th><th>Control records</th></tr>
<tr><td>Starting state of topic-partition with latest offset 100</td><td>SPSO=100, SPEO=100</td><td>SPSO=100, SPEO=100</td><td><pre>
SHARE_CHECKPOINT offset 130:
{
  "GroupId": "G1",
  "CheckpointEpoch": 1,
  "StartOffset": 110,
  "EndOffset": 110,
  "States": []
}
</pre></td></tr>
<tr><td colspan="3"><b>In the batched case with successful processing, there’s a state change per batch to move the SPSO forwards</b></td><td></td></tr>
<tr><td>Fetch records 100-109</td><td>SPEO=110, records 100-109 (acquired, delivery count 1)</td><td>SPSO=100, SPEO=110, records 100-109 (acquired, delivery count 1)</td><td></td></tr>
<tr><td>Acknowledge 100-109</td><td>SPSO=110</td><td>SPSO=110, SPEO=110</td><td><pre>
SHARE_DELTA offset 131:
{
  "GroupId": "G1",
  "CheckpointEpoch": 1,
  "States": [
    {
      "BaseOffset": 100,
      "LastOffset": 109,
      "State": 2 (Acknowledged),
      "DeliveryCount": 1
    }
  ]
}
</pre></td></tr>
<tr><td colspan="3"><b>With a messier sequence of release and acknowledge, there’s a state change for each operation which can act on multiple records</b></td><td></td></tr>
<tr><td>Fetch records 110-119</td><td>SPEO=120, records 110-119 (acquired, delivery count 1)</td><td>SPSO=110, SPEO=120, records 110-119 (acquired, delivery count 1)</td><td></td></tr>
<tr><td>Release 110</td><td>record 110 (available, delivery count 1)</td><td>SPSO=110, SPEO=120, record 110 (available, delivery count 1), records 111-119 (acquired, delivery count 1)</td><td><pre>
SHARE_DELTA offset 132:
{
  "GroupId": "G1",
  "CheckpointEpoch": 1,
  "States": [
    {
      "BaseOffset": 110,
      "LastOffset": 110,
      "State": 0 (Available),
      "DeliveryCount": 1
    }
  ]
}
</pre>Note that the SPEO in the control records is 111 at this point. All records after this are in their first delivery attempt so this is an acceptable situation.</td></tr>
</table>

Acknowledge 119	record 110 (available, delivery count 1), records 111-118 acquired, record 119 acknowledged	SPSO=110, SPEO=120, record 110 (available, delivery count 1), records 111-118 (acquired, delivery count 1), record 119 acknowledged	SHARE_DELTA offset 133:
			{
			"GroupId": "G1",
			"CheckpointEpoch": 1,
			"States": [
			{
			"BaseOffset": 111,
			"LastOffset": 118,
			"State": 0 (Available),
			"DeliveryCount": 0
			},
			{
			"BaseOffset": 119,
			"LastOffset": 119,
			"State": 2 (Acknowledged),
			"DeliveryCount": 1
			}
			]
			}

Fetch records 110, 120	SPEO=121, record 110 (acquired, delivery count 2), record 120 (acquired, delivery count 1)	SPSO=110, SPEO=121, record 110 (acquired, delivery count 2), records 111-118 (acquired, delivery count 1), record 119 acknowledged, record 120 (acquired, delivery count 1)	

Lock timeout elapsed 111, 112	records 111-112 (available, delivery count 1)	SPSO=110, SPEO=121, record 110 (acquired, delivery count 2), records 111-112 (available, delivery count 1), records 113-118 (acquired, delivery count 1), record 119 acknowledged, record 120 (acquired, delivery count 1)	SHARE_DELTA offset 134:
			{
			"GroupId": "G1",
			"CheckpointEpoch": 1,
			"States": [
			{
			"BaseOffset": 111,
			"LastOffset": 112,
			"State": 0 (Available),
			"DeliveryCount": 1
			}
			]
			}

Acknowledge 113-118	records 113-118 acknowledged	SPSO=110, SPEO=121, record 110 (acquired, delivery count 2), records 111-112 (available, delivery count 1), records 113-119 acknowledged, record 120 (acquired, delivery count 1)	SHARE_DELTA offset 135:
			{
			"GroupId": "G1",
			"CheckpointEpoch": 1,
			"States": [
			{
			"BaseOffset": 113,
			"LastOffset": 118,
			"State": 2 (Acknowledged),
			"DeliveryCount": 1
			}
			]
			}

Fetch records 111,112	records 111-112 (acquired, delivery count 2)	SPSO=110, SPEO=121, record 110-112 (acquired, delivery count 2), records 113-119 acknowledged, record 120 (acquired, delivery count 1)	

<table>
<tr>
<td>Acknowledge 110</td>
<td>SPSO=111</td>
<td>SPSO=111, SPEO=121, record 111-112 (acquired, delivery count 2), records 113-119 acknowledged, record 120 (acquired, delivery count 1)</td>
<td><pre>
SHARE_DELTA offset 136:
{
  "GroupId": "G1",
  "CheckpointEpoch": 1,
  "States": [
    {
      "BaseOffset": 110,
      "LastOffset": 110,
      "State": 2 (Acknowledged),
      "DeliveryCount": 2
    }
  ]
}
</pre></td>
</tr>
<tr>
<td>Acknowledge 111,112</td>
<td>SPSO=120</td>
<td>SPSO=120, SPEO=121, record 120 (acquired, delivery count 1)</td>
<td><pre>
SHARE_DELTA offset 137:
{
  "GroupId": "G1",
  "CheckpointEpoch": 1,
  "States": [
    {
      "BaseOffset": 111,
      "LastOffset": 112,
      "State": 2 (Acknowledged),
      "DeliveryCount": 2
    }
  ]
}
</pre>
or alternatively, taking a new checkpoint:
<pre>
SHARE_CHECKPOINT offset 137:
{
  "GroupId": "G1",
  "CheckpointEpoch": 2,
  "StartOffset": 120,
  "EndOffset": 120,
  "States": []
}
</pre>
Note that the delivery of 120 has not been recorded yet because it is the first delivery attempt and it is safe to recover the SPEO back to offset 120 and repeat the attempt.</td>
</tr>
</table>

Recovering share-partition state and interactions with log cleaning

A share-partition is a topic-partition with a subscription in a share group. The share-partition is essentially a view of the topic-partition, managed by the share-partition leader, with durable state stored on the topic-partition in **SHARE_CHECKPOINT** and **SHARE_DELTA** control records.

In order to recreate the share-partition state when a broker becomes the leader of a share-partition, it must read the most recent **SHARE_CHECKPOINT** and any subsequent **SHARE_DELTA** control records, which will all have the same checkpoint epoch. In order to minimise the amount of log scanning required, it’s important to write **SHARE_CHECKPOINT** records frequently, and also to have an efficient way of finding the most recent **SHARE_CHECKPOINT** record.

For each share-partition, the offset of the most recent **SHARE_CHECKPOINT** record is called the **Share Checkpoint Offset (SCO)**. The **Earliest Share Offset (ESO)** is the earliest of the share checkpoint offsets for all of the share groups with a subscription in a share group.

- The log cleaner can clean all **SHARE_CHECKPOINT** and **SHARE_DELTA** records before the SCO.
- The log cleaner must not clean **SHARE_CHECKPOINT** and **SHARE_DELTA** records after the SCO.

In practice, the ESO is used as the cut-off point for cleaning of these control records.

Administration

Several components work together to create share groups. The group coordinator is responsible for assignment, membership and the state of the group. The share-partition leaders are responsible for delivery and acknowledgement. The following table summarises the administration operations and how they work.

Operation	Supported by	Notes
Create share group	Group coordinator	This occurs as a side-effect of a ShareGroupHeartbeat. The group coordinator writes a record to the consumer offsets topic to persist the group's existence.
List share groups	Group coordinator	
List share group offsets	Group coordinator and share-partition leaders	
Describe share group	Group coordinator	
Alter share group offsets	Share-partition leaders	The share-partition leader makes a durable share-partition state update for each share-partition affected.
Delete share group offsets	Share-partition leaders	The share-partition leader makes a durable share-partition state update for each share-partition affected.
Delete share group	Group coordinator working with share-partition leaders	Only empty share groups can be deleted. However, the share-partition leaders need to delete share group offsets, and then delete the share group. It is not an atomic operation. The share-partition leader makes a durable share-partition state update for each share-partition affected. The group coordinator writes a tombstone record to the consumer offsets topic to persist the group deletion.

Public Interfaces

This KIP introduces extensive additions to the public interfaces.

Client API changes

KafkaShareConsumer

This KIP introduces a new interface for consuming records from a share group called `org.apache.kafka.clients.consumer.ShareConsumer` with an implementation called `org.apache.kafka.clients.consumer.KafkaShareConsumer` . The interface stability is `Evolving` .

```
@InterfaceStability.Evolving
public interface ShareConsumer<K, V> {

    /**
     * Get the current subscription. Will return the same topics used in the most recent call to
     * {@link #subscribe(Collection)}, or an empty set if no such call has been made.
     *
     * @return The set of topics currently subscribed to
     */
    Set<String> subscription();

    /**
     * Subscribe to the given list of topics to get dynamically assigned partitions.
     * <b>Topic subscriptions are not incremental. This list will replace the current
     * assignment, if there is one.</b> If the given list of topics is empty, it is treated the same as {@link
#unsubscribe()}.
     *
     * <p>
     * As part of group management, the coordinator will keep track of the list of consumers that belong to a
particular
     * group and will trigger a rebalance operation if any one of the following events are triggered:
     * <ul>
     * <li>A member joins or leaves the share group
     * <li>An existing member of the share group is shut down or fails
     * <li>The number of partitions changes for any of the subscribed topics
     * <li>A subscribed topic is created or deleted
     * </ul>
     *
     * @param topics The list of topics to subscribe to
     *
     * @throws IllegalArgumentException If topics is null or contains null or empty elements
     * @throws KafkaException for any other unrecoverable errors
     */
    void subscribe(Collection<String> topics);

    /**
     * Unsubscribe from topics currently subscribed with {@link #subscribe(Collection)}.
     *
     * @throws KafkaException for any other unrecoverable errors
     */
    void unsubscribe();

    /**
     * Fetch data for the topics specified using {@link #subscribe(Collection)}. It is an error to not have
     * subscribed to any topics before polling for data.
     *
     * <p>
     * This method returns immediately if there are records available. Otherwise, it will await the passed
timeout.
     * If the timeout expires, an empty record set will be returned.
     *
     * @param timeout The maximum time to block (must not be greater than {@link Long#MAX_VALUE} milliseconds)
     *
     * @return map of topic to records since the last fetch for the subscribed list of topics
     *
     * @throws AuthenticationException if authentication fails. See the exception for more details
     * @throws AuthorizationException if caller lacks Read access to any of the subscribed
     *             topics or to the configured groupId. See the exception for more details
     * @throws InterruptException if the calling thread is interrupted before or while this method is called
     * @throws InvalidTopicException if the current subscription contains any invalid
     *             topic (per {@link org.apache.kafka.common.internals.Topic#validate(String)})
     * @throws WakeupException if {@link #wakeup()} is called before or while this method is called
     * @throws KafkaException for any other unrecoverable errors (e.g. invalid groupId or
     *             session timeout, errors deserializing key/value pairs,
     *             or any new error cases in future versions)
     * @throws IllegalArgumentException if the timeout value is negative
     * @throws IllegalStateException if the consumer is not subscribed to any topics
     * @throws ArithmeticException if the timeout is greater than {@link Long#MAX_VALUE} milliseconds.
     */
    ConsumerRecords<K, V> poll(Duration timeout);

    /**
     * Acknowledge successful delivery of a record returned on the last {@link #poll(Duration)} call.
     * The acknowledgement is committed on the next {@link #commitSync()}, {@link #commitAsync()} or
     * {@link #poll(Duration)} call.
     *
     * <p>
     * Records for each topic-partition must be acknowledged in the order they were returned from
     * {@link #poll(Duration)}. By using this method, the consumer is using
     * <b>explicit acknowledgement</b>.
     *
     * @param record The record to acknowledge
     *
     * @throws IllegalArgumentException if the record being acknowledged doesn't meet the ordering requirement
     * @throws IllegalStateException if the record is not waiting to be acknowledged, or the consumer has
already
     *                               used implicit acknowledgement
     */
    void acknowledge(ConsumerRecord<K, V> record);

    /**
     * Acknowledge delivery of a record returned on the last {@link #poll(Duration)} call indicating whether
     * it was processed successfully. The acknowledgement is committed on the next {@link #commitSync()},
     * {@link #commitAsync()} or {@link #poll(Duration)} call. By using this method, the consumer is using
     * <b>explicit acknowledgement</b>.
     *
     * <p>
     * Records for each topic-partition must be acknowledged in the order they were returned from
     * {@link #poll(Duration)}.
     *
     * @param record The record to acknowledge
     * @param type The acknowledge type which indicates whether it was processed successfully
     *
     * @throws IllegalArgumentException if the record being acknowledged doesn't meet the ordering requirement
     * @throws IllegalStateException if the record is not waiting to be acknowledged, or the consumer has
already
     *                               used implicit acknowledgement
     */
    void acknowledge(ConsumerRecord<K, V> record, AcknowledgeType type);

    /**
     * Commit the acknowledgements for the records returned. If the consumer is using explicit acknowledgement,
     * the acknowledgements to commit have been indicated using {@link #acknowledge(ConsumerRecord)} or
     * {@link #acknowledge(ConsumerRecord, AcknowledgeType)}. If the consumer is using implicit acknowledgement,
     * all the records returned by the latest call to {@link #poll(Duration)} are acknowledged.
     * <p>
     * This is a synchronous commit and will block until either the commit succeeds, an unrecoverable error is
     * encountered (in which case it is thrown to the caller), or the timeout expires.
     *
     * @return A map of the results for each topic-partition for which delivery was acknowledged.
     *         If the acknowledgement failed for a topic-partition, an exception is present.
     *
     * @throws InterruptException If the thread is interrupted while blocked.
     * @throws KafkaException for any other unrecoverable errors
     */
    Map<TopicIdPartition, Optional<KafkaException>> commitSync();

    /**
     * Commit the acknowledgements for the records returned. If the consumer is using explicit acknowledgement,
     * the acknowledgements to commit have been indicated using {@link #acknowledge(ConsumerRecord)} or
     * {@link #acknowledge(ConsumerRecord, AcknowledgeType)}. If the consumer is using implicit acknowledgement,
     * all the records returned by the latest call to {@link #poll(Duration)} are acknowledged.
     * <p>
     * This is a synchronous commit and will block until either the commit succeeds, an unrecoverable error is
     * encountered (in which case it is thrown to the caller), or the timeout expires.
     *
     * @param timeout The maximum amount of time to await completion of the acknowledgement
     *
     * @return A map of the results for each topic-partition for which delivery was acknowledged.
     *         If the acknowledgement failed for a topic-partition, an exception is present.
     *
     * @throws IllegalArgumentException If the {@code timeout} is negative.
     * @throws InterruptException If the thread is interrupted while blocked.
     * @throws KafkaException for any other unrecoverable errors
     */
    Map<TopicIdPartition, Optional<KafkaException>> commitSync(Duration timeout);

    /**
     * Commit the acknowledgements for the records returned. If the consumer is using explicit acknowledgement,
     * the acknowledgements to commit have been indicated using {@link #acknowledge(ConsumerRecord)} or
     * {@link #acknowledge(ConsumerRecord, AcknowledgeType)}. If the consumer is using implicit acknowledgement,
     * all the records returned by the latest call to {@link #poll(Duration)} are acknowledged.
     *
     * @throws KafkaException for any other unrecoverable errors
     */
    void commitAsync();

    /**
     * Sets the acknowledge commit callback which can be used to handle acknowledgement completion.
     *
     * @param callback The acknowledge commit callback
     */
    void setAcknowledgeCommitCallback(AcknowledgeCommitCallback callback);

    /**
```

```
     * Determines the client's unique client instance ID used for telemetry. This ID is unique to
     * this specific client instance and will not change after it is initially generated.
     * The ID is useful for correlating client operations with telemetry sent to the broker and
     * to its eventual monitoring destinations.
     * <p>
     * If telemetry is enabled, this will first require a connection to the cluster to generate
     * the unique client instance ID. This method waits up to {@code timeout} for the consumer
     * client to complete the request.
     * <p>
     * Client telemetry is controlled by the {@link ConsumerConfig#ENABLE_METRICS_PUSH_CONFIG}
     * configuration option.
     *
     * @param timeout The maximum time to wait for consumer client to determine its client instance ID.
     *                The value must be non-negative. Specifying a timeout of zero means do not
     *                wait for the initial request to complete if it hasn't already.
     *
     * @return The client's assigned instance id used for metrics collection.
     *
     * @throws IllegalArgumentException If the {@code timeout} is negative.
     * @throws IllegalStateException If telemetry is not enabled because config `{@code enable.metrics.push}`
     *                               is set to `{@code false}`.
     * @throws InterruptException If the thread is interrupted while blocked.
     * @throws KafkaException If an unexpected error occurs while trying to determine the client
     *                        instance ID, though this error does not necessarily imply the
     *                        consumer client is otherwise unusable.
     */
    Uuid clientInstanceId(Duration timeout);

    /**
     * Get the metrics kept by the consumer
     */
    Map<MetricName, ? extends Metric> metrics();

    /**
     * Close the consumer, waiting for up to the default timeout of 30 seconds for any needed cleanup.
     * This will commit acknowledgements if possible within the default timeout.
     * See {@link #close(Duration)} for details. Note that {@link #wakeup()} cannot be used to interrupt close.
     *
     * @throws InterruptException If the thread is interrupted before or while this method is called
     * @throws KafkaException for any other error during close
     */
    void close();

    /**
     * Tries to close the consumer cleanly within the specified timeout. This method waits up to
     * {@code timeout} for the consumer to complete acknowledgements and leave the group.
     * If the consumer is unable to complete acknowledgements and gracefully leave the group
     * before the timeout expires, the consumer is force closed. Note that {@link #wakeup()} cannot be
     * used to interrupt close.
     *
     * @param timeout The maximum time to wait for consumer to close gracefully. The value must be
     *                non-negative. Specifying a timeout of zero means do not wait for pending requests to
complete.
     *
     * @throws IllegalArgumentException If the {@code timeout} is negative.
     * @throws InterruptException If the thread is interrupted before or while this method is called
     * @throws KafkaException for any other error during close
     */
    void close(Duration timeout);

    /**
     * Wake up the consumer. This method is thread-safe and is useful in particular to abort a long poll.
     * The thread which is blocking in an operation will throw {@link WakeupException}.
     * If no thread is blocking in a method which can throw {@link WakeupException},
     * the next call to such a method will raise it instead.
     */
   void wakeup();
}
```

The following constructors are provided for `KafkaShareConsumer` .

Method signature	Description
`KafkaShareConsumer(Map<String, Object> configs)`	Constructor
`KafkaShareConsumer(Properties properties)`	Constructor

`KafkaShareConsumer(Map<String, Object> configs,`
`  Deserializer<K> keyDeserializer,`
`  Deserializer<V> valueDeserializer)`
	Constructor

`KafkaShareConsumer(Properties properties,`
`  Deserializer<K> keyDeserializer,`
`  Deserializer<V> valueDeserializer)`
	Constructor

AcknowledgeCommitCallback

The new `org.apache.kafka.clients.consumer.AcknowledgeCommitCallback` can be implemented by the user to execute when acknowledgement completes. It is called on the application thread and is not permitted to called the methods of `KafkaShareConsumer` with the exception of `KafkaShareConsumer.wakeup()`.

Method signature	Description
`void onComplete(Map<TopicIdPartition, Set<OffsetAndMetadata>> offsets, Exception exception)`	A callback method the user can implement to provide asynchronous handling of request completion. Parameters: offsets - A map of the offsets that this callback applies to. exception - The exception thrown during processing of the request, or null if the acknowledgement completed successfully. Exceptions: WakeupException - if KafkaShareConsumer.wakeup() is called. InterruptException - if the calling thread is interrupted. AuthorizationException - if not authorized to the topic or group. KafkaException - for any other unrecoverable errors.

ConsumerRecord

Add the following method on the `org.apache.kafka.client.consumer.ConsumerRecord` class.

Method signature	Description
`Optional<Short> deliveryCount()`	Get the delivery count for the record if available.

The delivery count is available for records delivered using a share group and `Optional.empty()` otherwise.

A new constructor is also added:

```
    /**
      * Creates a record to be received from a specified topic and partition
      *
      * @param topic The topic this record is received from
      * @param partition The partition of the topic this record is received from
      * @param offset The offset of this record in the corresponding Kafka partition
      * @param timestamp The timestamp of the record.
      * @param timestampType The timestamp type
      * @param serializedKeySize The length of the serialized key
      * @param serializedValueSize The length of the serialized value
      * @param key The key of the record, if one exists (null is allowed)
      * @param value The record contents
      * @param headers The headers of the record
      * @param leaderEpoch Optional leader epoch of the record (may be empty for legacy record formats)
      * @param deliveryCount Optional delivery count of the record (may be empty when deliveries not counted)
      */
     public ConsumerRecord(String topic,
                           int partition,
                           long offset,
                           long timestamp,
                           TimestampType timestampType,
                           int serializedKeySize,
                           int serializedValueSize,
                           K key,
                           V value,
                           Headers headers,
                           Optional<Integer> leaderEpoch,
                           Optional<Short> deliveryCount)
```

AcknowledgeType

The new `org.apache.kafka.clients.consumer.AcknowledgeType` enum distinguishes between the types of acknowledgement for a record consumer using a share group.

Enum constant	Description
`ACCEPT` (0)	The record was consumed successfully
`RELEASE` (1)	The record was not consumed successfully. Release it for another delivery attempt.
`REJECT` (2)	The record was not consumed successfully. Reject it and do not release it for another delivery attempt.

AdminClient

Add the following methods on the `org.apache.kafka.client.admin.AdminClient` interface.

Method signature	Description
`AlterShareGroupOffsetsResult alterShareGroupOffsets(String groupId, Map<TopicPartition, OffsetAndMetadata> offsets)`	Alter offset information for a share group.
`AlterShareGroupOffsetsResult alterShareGroupOffsets(String groupId, Map<TopicPartition, OffsetAndMetadata> offsets, AlterShareGroupOffsetsOptions options)`	Alter offset information for a share group.
`DeleteShareGroupOffsetsResult deleteShareGroupOffsets(String groupId, Set<TopicPartition> partitions)`	Delete offset information for a set of partitions in a share group.
`DeleteShareGroupOffsetsResult deleteShareGroupOffsets(String groupId, Set<TopicPartition> partitions, DeleteShareGroupOffsetsOptions options)`	Delete offset information for a set of partitions in a share group.
`DeleteShareGroupResult deleteShareGroups(Collection<String> groupIds)`	Delete share groups from the cluster.
`DeleteShareGroupResult deleteShareGroups(Collection<String> groupIds, DeleteShareGroupOptions options)`	Delete share groups from the cluster.
`DescribeShareGroupsResult describeShareGroups(Collection<String> groupIds)`	Describe some share groups in the cluster.

`DescribeShareGroupsResult describeShareGroups(Collection<String> groupIds, DescribeShareGroupsOptions options)`	Describe some share groups in the cluster.
`ListShareGroupOffsetsResult listShareGroupOffsets(Map<String, ListShareGroupOffsetsSpec> groupSpecs)`	List the share group offsets available in the cluster for the specified share groups.
`ListShareGroupOffsetsResult listShareGroupOffsets(Map<String, ListShareGroupOffsetsSpec> groupSpecs, ListShareGroupOffsetsOptions options)`	List the share group offsets available in the cluster for the specified share groups.
`ListShareGroupsResult listShareGroups()`	List the share groups available in the cluster.
`ListShareGroupsResult listShareGroups(ListShareGroupsOptions options)`	List the share groups available in the cluster.
`ListGroupsResult listGroups()`	List the groups available in the cluster.
`ListGroupsResult listGroups(ListGroupsOptions options)`	List the groups available in the cluster.

The equivalence between the consumer group and share group interfaces is clear. There are some differences:

- Altering the offsets for a share group resets the Share-Partition Start Offset for topic-partitions in the share group (share-partitions)
- The members of a share group are not assigned distinct sets of partitions
- A share group has only three states - `EMPTY` , `STABLE` and `DEAD`

Here are the method signatures.

```
    /**
     * Alters offsets for the specified group. In order to succeed, the group must be empty.
     *
     * <p>This is a convenience method for {@link #alterShareGroupOffsets(String, Map,
AlterShareGroupOffsetsOptions)} with default options.
     * See the overload for more details.
     *
     * @param groupId The group for which to alter offsets.
     * @param offsets A map of offsets by partition with associated metadata.
     * @return The AlterShareGroupOffsetsResult.
     */
    default AlterShareGroupOffsetsResult alterShareGroupOffsets(String groupId, Map<TopicPartition,
OffsetAndMetadata> offsets) {
        return alterShareGroupOffsets(groupId, offsets, new AlterShareGroupOffsetsOptions());
    }

    /**
     * Alters offsets for the specified group. In order to succeed, the group must be empty.
     *
     * <p>This operation is not transactional so it may succeed for some partitions while fail for others.
     *
     * @param groupId The group for which to alter offsets.
     * @param offsets A map of offsets by partition with associated metadata. Partitions not specified in the
map are ignored.
     * @param options The options to use when altering the offsets.
     * @return The AlterShareGroupOffsetsResult.
     */
    AlterShareGroupOffsetsResult alterShareGroupOffsets(String groupId, Map<TopicPartition, OffsetAndMetadata>
offsets, AlterShareGroupOffsetsOptions options);

    /**
     * Delete offsets for a set of partitions in a share group with the default
     * options. This will succeed at the partition level only if the group is not actively
     * subscribed to the corresponding topic.
     *
     * <p>This is a convenience method for {@link #deleteShareGroupOffsets(String, Map,
DeleteShareGroupOffsetsOptions)} with default options.
     * See the overload for more details.
     *
     * @return The DeleteShareGroupOffsetsResult.
     */
    default DeleteShareGroupOffsetsResult deleteShareGroupOffsets(String groupId, Set<TopicPartition>
partitions) {
        return deleteShareGroupOffsets(groupId, partitions, new DeleteShareGroupOffsetsOptions());
    }
    /**
     * Delete offsets for a set of partitions in a share group. This will
     * succeed at the partition level only if the group is not actively subscribed
     * to the corresponding topic.
     *
     * @param options The options to use when deleting offsets in a share group.
     * @return The DeleteShareGroupOffsetsResult.
     */
    DeleteShareGroupOffsetsResult deleteShareGroupOffsets(String groupId,
        Set<TopicPartition> partitions,
        DeleteShareGroupOffsetsOptions options);

    /**
     * Delete share groups from the cluster with the default options.
     *
     * <p>This is a convenience method for {@link #deleteShareGroups(Collection<String>,
DeleteShareGroupsOptions)} with default options.
     * See the overload for more details.
     *
     * @param groupIds The IDs of the groups to delete.
     * @return The DeleteShareGroupsResult.
     */
    default DeleteShareGroupsResult deleteShareGroups(Collection<String> groupIds) {
        return deleteShareGroups(groupIds, new DeleteShareGroupsOptions());
    }

    /**
     * Delete share groups from the cluster.
     *
     * @param groupIds The IDs of the groups to delete.
     * @param options The options to use when deleting a share group.
     * @return The DeleteShareGroupsResult.
     */
    DeleteShareGroupsResult deleteShareGroups(Collection<String> groupIds, DeleteShareGroupsOptions options);

    /**
     * Describe some share groups in the cluster, with the default options.
     *
     * <p>This is a convenience method for {@link #describeShareGroups(Collection, DescribeShareGroupsOptions)}
     * with default options. See the overload for more details.
     *
     * @param groupIds The IDs of the groups to describe.
     * @return The DescribeShareGroupsResult.
     */
    default DescribeShareGroupsResult describeShareGroups(Collection<String> groupIds) {
        return describeShareGroups(groupIds, new DescribeShareGroupsOptions());
    }

    /**
     * Describe some share groups in the cluster.
     *
     * @param groupIds The IDs of the groups to describe.
     * @param options  The options to use when describing the groups.
     * @return The DescribeShareGroupsResult.
     */
    DescribeShareGroupsResult describeShareGroups(Collection<String> groupIds,
                                                 DescribeShareGroupsOptions options);

    /**
     * List the share group offsets available in the cluster for the specified share groups with the default
options.
     *
     * <p>This is a convenience method for {@link #listShareGroupOffsets(Map, ListShareGroupOffsetsOptions)}
     * to list offsets of all partitions for the specified share groups with default options.
     *
     * @param groupSpecs Map of share group ids to a spec that specifies the topic partitions of the group to
list offsets for.
     * @return The ListShareGroupOffsetsResult
     */
    default ListShareGroupOffsetsResult listShareGroupOffsets(Map<String, ListShareGroupOffsetsSpec>
groupSpecs) {
        return listShareGroupOffsets(groupSpecs, new ListShareGroupOffsetsOptions());
    }

    /**
     * List the share group offsets available in the cluster for the specified share groups.
     *
     * @param groupSpecs Map of share group ids to a spec that specifies the topic partitions of the group to
list offsets for.
     * @param options The options to use when listing the share group offsets.
     * @return The ListShareGroupOffsetsResult
     */
    ListShareGroupOffsetsResult listShareGroupOffsets(Map<String, ListShareGroupOffsetsSpec> groupSpecs,
ListShareGroupOffsetsOptions options);

    /**
     * List the share groups available in the cluster with the default options.
     *
     * <p>This is a convenience method for {@link #listShareGroups(ListShareGroupsOptions)} with default
options.
     * See the overload for more details.
     *
     * @return The ListShareGroupsResult.
     */
    default ListShareGroupsResult listShareGroups() {
        return listShareGroups(new ListShareGroupsOptions());
    }

    /**
     * List the share groups available in the cluster.
     *
     * @param options The options to use when listing the share groups.
     * @return The ListShareGroupsResult.
     */
    ListShareGroupsResult listShareGroups(ListShareGroupsOptions options);

    /**
     * List the groups available in the cluster with the default options.
     *
     * <p>This is a convenience method for {@link #listGroups(ListGroupsOptions)} with default options.
     * See the overload for more details.
     *
     * @return The ListGroupsResult.
     */
    default ListGroupsResult listGroups() {
        return listGroups(new ListGroupsOptions());
    }

    /**
     * List the groups available in the cluster.
     *
     * @param options The options to use when listing the groups.
     * @return The ListGroupsResult.
     */
    ListGroupsResult listGroups(ListGroupsOptions);
```

AlterShareGroupOffsetsResult

```
package org.apache.kafka.clients.admin;

/**
 * The result of the {@link Admin#alterShareGroupOffsets(String groupId, Map<TopicPartition,
OffsetAndMetadata>), AlterShareGroupOffsetsOptions)} call.
 * <p>
 * The API of this class is evolving, see {@link Admin} for details.
 */
@InterfaceStability.Evolving
public class AlterShareGroupOffsetsResult {
    /**
     * Return a future which succeeds if all the alter offsets succeed.
     */
    public KafkaFuture<Void> all() {
    }

    /**
     * Return a future which can be used to check the result for a given partition.
     */
    public KafkaFuture<Void> partitionResult(final TopicPartition partition) {
    }
}
```

AlterShareGroupOffsetsOptions

```
package org.apache.kafka.client.admin;

/**
 * Options for the {@link Admin#alterShareGroupOffsets(String groupId, Map<TopicPartition, OffsetAndMetadata>),
AlterShareGroupOffsetsOptions)} call.
 * <p>
 * The API of this class is evolving, see {@link Admin} for details.
 */
@InterfaceStability.Evolving
public class AlterShareGroupOffsetsOptions extends AbstractOptions<AlterShareGroupOffsetsOptions> {
}
```

DeleteShareGroupOffsetsResult

```
package org.apache.kafka.clients.admin;

/**
 * The result of the {@link Admin#deleteShareGroupOffsets(String, Set<TopicPartition>,
DeleteShareGroupOffsetsOptions)} call.
 * <p>
 * The API of this class is evolving, see {@link Admin} for details.
 */
@InterfaceStability.Evolving
public class DeleteShareGroupOffsetsResult {
    /**
     * Return a future which succeeds only if all the deletions succeed.
     */
    public KafkaFuture<Void> all() {
    }

    /**
     * Return a future which can be used to check the result for a given partition.
     */
    public KafkaFuture<Void> partitionResult(final TopicPartition partition) {
    }
}
```

DeleteShareGroupOffsetsOptions

```
package org.apache.kafka.client.admin;

/**
 * Options for the {@link Admin#deleteShareGroupOffsets(String, Set<TopicPartition>,
DeleteShareGroupOffsetsOptions)} call.
 * <p>
 * The API of this class is evolving, see {@link Admin} for details.
 */
@InterfaceStability.Evolving
public class DeleteShareGroupOffsetsOptions extends AbstractOptions<DeleteShareGroupOffsetsOptions> {
}
```

DeleteShareGroupsResult

```
package org.apache.kafka.clients.admin;

/**
 * The result of the {@link Admin#deleteShareGroups(Collection<String>, DeleteShareGroupsOptions)} call.
 * <p>
 * The API of this class is evolving, see {@link Admin} for details.
 */
@InterfaceStability.Evolving
public class DeleteShareGroupsResult {
    /**
     * Return a future which succeeds only if all the deletions succeed.
     */
    public KafkaFuture<Void> all() {
    }

    /**
     * Return a map from group id to futures which can be used to check the status of individual deletions.
     */
    public Map<String, KafkaFuture<Void>> deletedGroups() {
    }
}
```

DeleteShareGroupsOptions

```
package org.apache.kafka.client.admin;

/**
 * Options for the {@link Admin#deleteShareGroups(Collection<String>, DeleteShareGroupsOptions)} call.
 * <p>
 * The API of this class is evolving, see {@link Admin} for details.
 */
@InterfaceStability.Evolving
public class DeleteShareGroupsOptions extends AbstractOptions<DeleteShareGroupsOptions> {
}
```

DescribeShareGroupsResult

```
package org.apache.kafka.clients.admin;

/**
 * The result of the {@link Admin#describeShareGroups(Collection<String>, DescribeShareGroupsOptions)} call.
 * <p>
 * The API of this class is evolving, see {@link Admin} for details.
 */
@InterfaceStability.Evolving
public class DescribeShareGroupsResult {
    /**
     * Return a future which yields all ShareGroupDescription objects, if all the describes succeed.
     */
    public KafkaFuture<Map<String, ShareGroupDescription>> all() {
    }

    /**
     * Return a map from group id to futures which yield group descriptions.
     */
    public Map<String, KafkaFuture<ShareGroupDescription>> describedGroups() {
    }
}
```

ShareGroupDescription

This class does indeed reuse the `MemberDescription` class intended for consumer groups. It is sufficiently general to work for share groups also.

```
package org.apache.kafka.client.admin;

import org.apache.kafka.common.Node;
import org.apache.kafka.common.ShareGroupState;
import org.apache.kafka.common.acl.AclOperation;

/**
 * A detailed description of a single share group in the cluster.
 * <p>
 * The API of this class is evolving, see {@link Admin} for details.
 */
@InterfaceStability.Evolving
public class ShareGroupDescription {
  public ShareGroupDescription(String groupId, Collection<MemberDescription> members, ShareGroupState state,
Node coordinator);
  public ShareGroupDescription(String groupId, Collection<MemberDescription> members, ShareGroupState state,
Node coordinator, Set<AclOperation> authorizedOperations);

  /**
   * The id of the share group.
   */
  public String groupId();

  /**
   * A list of the members of the share group.
   */
  public Collection<MemberDescription> members();

  /**
   * The share group state, or UNKNOWN if the state cannot be parsed.
   */
  public ShareGroupState state();

  /**
   * The share group coordinator, or null if the coordinator is not known.
   */
  public Node coordinator();

  /**
   * The authorized operations for this group, or null if that information is not known.
   */
  public Set<AclOperation> authorizedOperations();
}
```

DescribeShareGroupsOptions

```
package org.apache.kafka.client.admin;

/**
 * Options for {@link Admin#describeShareGroups(Collection<String>, DescribeShareGroupsOptions)}.
 *
 * The API of this class is evolving, see {@link Admin} for details.
 */
@InterfaceStability.Evolving
public class DescribeShareGroupsOptions extends AbstractOptions<DescribeShareGroupsOptions> {
    public DescribeShareGroupsOptions includeAuthorizedOperations(boolean includeAuthorizedOperations);

    public boolean includeAuthorizedOperations();
}
```

ListShareGroupOffsetsResult

The offset returned for each topic-partition is the share-partition start offset (SPSO).

```
package org.apache.kafka.clients.admin;

/**
 * The result of the {@link Admin#listShareGroupOffsets(Map<String, ListShareGroupOffsetsSpec>,
ListShareGroupOffsetsOptions)} call.
 * <p>
 * The API of this class is evolving, see {@link Admin} for details.
 */
@InterfaceStability.Evolving
public class ListShareGroupOffsetsResult {
    /**
     * Return a future which yields all Map<String, Map<TopicPartition, OffsetAndMetadata> objects, if requests
for all the groups succeed.
     */
    public KafkaFuture<Map<String, Map<TopicPartition, OffsetAndMetadata>>> all() {
    }

    /**
     * Return a future which yields a map of topic partitions to OffsetAndMetadata objects for the specified
group.
     */
    public KafkaFuture<Map<TopicPartition, OffsetAndMetadata>> partitionsToOffsetAndMetadata(String groupId) {
    }
}
```

ListShareGroupOffsetsOptions

```
package org.apache.kafka.client.admin;

/**
 * Options for {@link Admin#listShareGroupOffsets(Map<String, ListShareGroupOffsetsSpec>,
ListShareGroupOffsetsOptions)}.
 * <p>
 * The API of this class is evolving, see {@link Admin} for details.
 */
@InterfaceStability.Evolving
public class ListShareGroupOffsetsOptions extends AbstractOptions<ListShareGroupOffsetsOptions> {
}
```

ListShareGroupOffsetsSpec

```
package org.apache.kafka.client.admin;

/**
 * Specification of share group offsets to list using {@link Admin#listShareGroupOffsets(Map<String,
ListShareGroupOffsetsSpec>, ListShareGroupOffsetsOptions)}.
 * <p>
 * The API of this class is evolving, see {@link Admin} for details.
 */
@InterfaceStability.Evolving
public class ListShareGroupOffsetsSpec {
  public ListShareGroupOffsetsSpec();

  /**
   * Set the topic partitions whose offsets are to be listed for a share group.
   */
  ListShareGroupOffsetsSpec topicPartitions(Collection<TopicPartition> topicPartitions);

  /**
   * Returns the topic partitions whose offsets are to be listed for a share group.
   */
  Collection<TopicPartition> topicPartitions();
}
```

ListShareGroupsResult

```
package org.apache.kafka.clients.admin;

/**
 * The result of the {@link Admin#listShareGroups(ListShareGroupsOptions)} call.
 * <p>
 * The API of this class is evolving, see {@link Admin} for details.
 */
@InterfaceStability.Evolving
public class ListShareGroupsResult {
    /**
     * Returns a future that yields either an exception, or the full set of share group listings.
     */
    public KafkaFuture<Collection<ShareGroupListing>> all() {
    }

    /**
     * Returns a future which yields just the valid listings.
     */
    public KafkaFuture<Collection<ShareGroupListing>> valid() {
    }

    /**
     * Returns a future which yields just the errors which occurred.
     */
    public KafkaFuture<Collection<Throwable>> errors() {
    }
}
```

ShareGroupListing

```
package org.apache.kafka.client.admin;

import org.apache.kafka.common.ShareGroupState;

/**
 * A listing of a share group in the cluster.
 * <p>
 * The API of this class is evolving, see {@link Admin} for details.
 */
@InterfaceStability.Evolving
public class ShareGroupListing {
  public ShareGroupListing(String groupId);
  public ShareGroupListing(String groupId, Optional<ShareGroupState> state);

  /**
   * The id of the share group.
   */
  public String groupId();

  /**
   * The share group state.
   */
  public Optional<ShareGroupState> state();
}
```

ListShareGroupsOptions

```
package org.apache.kafka.client.admin;

import org.apache.kafka.common.ShareGroupState;

/**
 * Options for {@link Admin#listShareGroups(ListShareGroupsOptions)}.
 *
 * The API of this class is evolving, see {@link Admin} for details.
 */
@InterfaceStability.Evolving
public class ListShareGroupsOptions extends AbstractOptions<ListShareGroupsOptions> {
    /**
     * If states is set, only groups in these states will be returned. Otherwise, all groups are returned.
     */
    public ListShareGroupsOptions inStates(Set<ShareGroupState> states);

    /**
     * Return the list of States that are requested or empty if no states have been specified.
     */
    public Set<ShareGroupState> states();
}
```

ListGroupsResult

```
package org.apache.kafka.clients.admin;

/**
 * The result of the {@link Admin#listGroups(ListGroupsOptions)} call.
 * <p>
 * The API of this class is evolving, see {@link Admin} for details.
 */
@InterfaceStability.Evolving
public class ListGroupsResult {
    /**
     * Returns a future that yields either an exception, or the full set of group listings.
     */
    public KafkaFuture<Collection<GroupListing>> all() {
    }

    /**
     * Returns a future which yields just the valid listings.
     */
    public KafkaFuture<Collection<GroupListing>> valid() {
    }

    /**
     * Returns a future which yields just the errors which occurred.
     */
    public KafkaFuture<Collection<Throwable>> errors() {
    }
}
```

GroupListing

```
package org.apache.kafka.client.admin;

import org.apache.kafka.common.ShareGroupState;

/**
 * A listing of a group in the cluster.
 * <p>
 * The API of this class is evolving, see {@link Admin} for details.
 */
@InterfaceStability.Evolving
public class GroupListing {
  public GroupListing(String groupId, GroupType type);

  /**
   * The id of the group.
   */
  public String groupId();

  /**
   * The group type.
   */
  public GroupType type();
}
```

ListGroupsOptions

```
package org.apache.kafka.client.admin;

import org.apache.kafka.common.GroupType;

/**
 * Options for {@link Admin#listGroups(ListGroupsOptions)}.
 *
 * The API of this class is evolving, see {@link Admin} for details.
 */
@InterfaceStability.Evolving
public class ListGroupsOptions extends AbstractOptions<ListGroupsOptions> {
    /**
     * If types is set, only groups of these types will be returned. Otherwise, all groups are returned.
     */
    public ListGroupsOptions types(Set<GroupType> types);

    /**
     * Return the list of types that are requested or empty if no types have been specified.
     */
    public Set<GroupType> types();
}
```

GroupType

Another case is added to the `org.apache.kafka.common.GroupType` enum:

Enum constant	Description
`SHARE("share")`	Share group

ShareGroupState

A new enum `org.apache.kafka.common.ShareGroupState` is added:

Enum constant
`DEAD`
`EMPTY`

`STABLE`
`UNKNOWN`

Its definition follows the pattern of `ConsumerGroupState` with fewer states.

Command-line tools

kafka-share-groups.sh

A new tool called `kafka-share-groups.sh` is added for working with share groups. It has the following options:

Option	Description
--all-topics	Consider all topics assigned to a group in the \`reset-offsets\` process.
--bootstrap-server <String: server to connect to>	REQUIRED: The server(s) to connect to.
--command-config <String: command config property file>	Property file containing configs to be passed to Admin Client.
--delete	Pass in groups to delete topic partition offsets over the entire share group. For instance --group g1 --group g2
--delete-offsets	Delete offsets of share group. Supports one share group at the time, and multiple topics.
--describe	Describe share group and list offset lag (number of records not yet processed) related to given group.
--dry-run	Only show results without executing changes on share groups. Supported operations: reset-offsets.
--execute	Execute operation. Supported operations: reset-offsets.
--group <String: share group>	The share group we wish to act on.
--help	Print usage information.
--list	List all share groups.
--members	Describe members of the group. This option may be used with the '--describe' option only.
--offsets	Describe the group and list all topic partitions in the group along with their offset lag. This is the default sub-action of and may be used with the '--describe' option only.
--reset-offsets	Reset offsets of share group. Supports one share group at a time, and instances must be inactive.
--state [String]	When specified with '--describe', includes the state of the group. When specified with '--list', it displays the state of all groups. It can also be used to list groups with specific states.
--timeout <Long: timeout (ms)>	The timeout that can be set for some use cases. For example, it can be used when describing the group to specify the maximum amount of time in milliseconds to wait before the group stabilizes (when the group is just created, or is going through some changes). (default: 5000)
--to-datetime <String: datetime>	Reset offsets to offset from datetime. Format: 'YYYY-MM-DDTHH:mm:SS.sss'.
--to-earliest	Reset offsets to earliest offset.
--to-latest	Reset offsets to latest offset.
--topic <String: topic>	The topic whose share group information should be deleted or topic which should be included in the reset offset process.
--version	Display Kafka version.

Here are some examples.

To display a list of all share groups:

```
$ kafka-share-groups.sh --bootstrap-server localhost:9092 --list
```

To delete the information for topic `T1` from inactive share group `S1` , which essentially resets the consumption of this topic in the share group:

```
$ kafka-share-groups.sh --bootstrap-server localhost:9092 --group S1 --topic T1 --delete-offsets
```

To set the starting offset for consuming topic `T1` in inactive share group `S1` to a specific date and time:

```
$ kafka-share-groups.sh --bootstrap-server localhost:9092 --group S1 --topic T1 --reset-offsets --to-datetime 1999
-12-31T23:57:00.000 --execute
```

kafka-console-share-consumer.sh

A new tool called `kafka-console-share-consumer.sh` is added for reading data from Kafka topics using a share group and outputting to standard output. This is similar to `kafka-console-consumer.sh` but using a share group and supporting the various acknowledge modes. It has the following options:

Option	Description
--bootstrap-server <String: server to connect to>	REQUIRED: The server(s) to connect to.
--consumer-config <String: config file>	Consumer config properties file. Note that [consumer-property] takes precedence over this config.
--consumer-property <String: consumer_prop>	Consumer property in the form key=value.
--enable-systest-events	Log lifecycle events of the consumer in addition to logging consumed messages. (This is specific for system tests.)
--formatter <String: class>	The name of a class to use for formatting Kafka messages for display. (default: kafka.tools.DefaultMessageFormatter)
--formatter-config <String: config file>	Config properties file to initialize the message formatter. Note that [property] takes precedence of this config.
--group <String: share groud id>	The share group id of the consumer. (default: share)
--help	Print usage information.
--key-deserializer <String: deserializer for keys>	The name of the class to use for deserializing keys.
--max-messages <Integer: num_messages>	The maximum number of messages to consume before exiting. If not set, consumption is continual.

--property <String: prop>	The properties to initialize the message formatter. Default properties include:
	print.timestamp=true|false
	print.key=true|false
	print.offset=true|false
	print.delivery=true|false
	print.partition=true|false
	print.headers=true|false
	print.value=true|false
	key.separator=<key.separator>
	line.separator=<line.separator>
	headers.separator=<line.separator>
	null.literal=<null.literal>
	key.deserializer=<key.deserializer>
	value.deserializer=<value.deserializer>
	header.deserializer=<header.deserializer>
	Users can also pass in customized properties for their formatter; more specifically, users can pass in properties keyed with 'key.deserializer.', 'value.deserializer.' and 'headers.deserializer.' prefixes to configure their deserializers.

--reject	If specified, messages are rejected as they are consumed.
--reject-message-on-error	If there is an error when processing a message, reject it instead of halting.
--release	If specified, messages are released as they are consumed.

--timeout-ms <Integer: timeout_ms>	If specified, exit if no message is available for consumption for the specific interval.
--topic <String: topic>	REQUIRED: The topic to consume from.
--value-deserializer <String: deserializer for values>	The name of the class to use for deserializing values.
--version	Display Kafka version.

kafka-producer-perf-test.sh

The following enhancements are made to the `kafka-producer-perf-test.sh` tool. The changes are intended to make this tool useful for observing the operation of share groups by generating a low message rate with predictable message payloads.

Option	Description
--throughput THROUGHPUT	(Existing option) Enhanced to permit fractional rates, such as 0.5 meaning 1 message every 2 seconds.
--payload-monotonic	payload is monotonically increasing integer.

Configuration

Broker configuration

Configuration	Description	Values
`group.share.enable`	Whether to enable share groups on the broker.	Default `false` while the feature is being developed. Will become `true` in a future release.
`group.share.delivery.count.limit`	The maximum number of delivery attempts for a record delivered to a share group.	Default 5, minimum 2, maximum 10
`group.share.record.lock.duration.ms`	Share-group record acquisition lock duration in milliseconds.	Default 30000 (30 seconds), minimum 1000 (1 second), maximum 60000 (60 seconds)
`group.share.record.lock.duration.max.ms`	Share-group record acquisition lock maximum duration in milliseconds.	Default 60000 (60 seconds), minimum 1000 (1 second), maximum 3600000 (1 hour)
`group.share.record.lock.partition.limit`	Share-group record lock limit per share-partition.	Default 200, minimum 100, maximum 10000
`group.share.session.timeout.ms`	The timeout to detect client failures when using the group protocol.	Default 45000 (45 seconds)
`group.share.min.session.timeout.ms`	The minimum session timeout.	Default 45000 (45 seconds)
`group.share.max.session.timeout.ms`	The maximum session timeout.	Default 60000 (60 seconds)
`group.share.heartbeat.interval.ms`	The heartbeat interval given to the members.	Default 5000 (5 seconds)
`group.share.min.heartbeat.interval.ms`	The minimum heartbeat interval.	Default 5000 (5 seconds)
`group.share.max.heartbeat.interval.ms`	The maximum heartbeat interval.	Default 15000 (15 seconds)
`group.share.max.groups`	The maximum number of share groups.	Default 10, minimum 1, maximum 100
`group.share.max.size`	The maximum number of consumers that a single share group can accommodate.	Default 200, minimum 10, maximum 1000
`group.share.assignors`	The server-side assignors as a list of full class names. In the initial delivery, only the first one in the list is used.	A list of class names. Default `"org.apache.server.group.share.SimpleAssignor"`

Group configuration

The following dynamic group configuration properties are added. These are properties for which it would be problematic to have consumers in the same share group using different behavior if the properties were specified in the consumer clients themselves.

Configuration	Description	Values
`group.share.isolation.level`	Controls how to read records written transactionally. If set to `"read_committed"`, the share group will only deliver transactional records which have been committed. If set to `"read_uncommitted"`, the share group will return all messages, even transactional messages which have been aborted. Non-transactional records will be returned unconditionally in either mode.	Valid values `"read_committed"` and `"read_uncommitted"` (default)
`group.share.auto.offset.reset`	What to do when there is no initial offset in Kafka or if the current offset does not exist any more on the server: • `"earliest"` : automatically reset the offset to the earliest offset • `"latest"` : automatically reset the offset to the latest offset	Valid values `"latest"` (default) and `"earliest"`
`group.share.record.lock.duration.ms`	Record acquisition lock duration in milliseconds.	null, which uses the cluster configuration `group.share.record.lock.duration.ms`, minimum 1000, maximum limited by the cluster configuration `group.share.record.lock.duration.max.ms`
`group.type`	Ensures that a newly created group has the specified group type.	Valid values: `"consumer"` or `"share"`

Consumer configuration

The existing consumer configurations apply for share groups with the following exceptions:

- `auto.offset.reset` : this is handled by a dynamic group configuration `group.share.auto.offset.reset`
- `enable.auto.commit` and `auto.commit.interval.ms` : share groups do not support auto-commit
- `isolation.level` : this is handled by a dynamic group configuration `group.share.isolation.level`
- `partition.assignment.strategy` : share groups do not support client-side partition assignors
- `interceptor.classes` : interceptors are not supported

Kafka protocol changes

This KIP introduces the following new APIs:

- `ShareGroupHeartbeat` - for consumers to form and maintain share groups
- `ShareGroupDescribe` - for describing share groups
- `ShareFetch` - for fetching records from share-partition leaders
- `ShareAcknowledge` - for acknowledging delivery of records with share-partition leaders
- `AlterShareGroupOffsets` - for altering the share-partition start offsets for the share-partitions in a share group
- `DeleteShareGroupOffsets` - for deleting the offsets for the share-partitions in a share group
- `DescribeShareGroupOffsets` - for describing the offsets for the share-partitions in a share group

Error codes

This KIP adds the following error codes the Kafka protocol.

- `INVALID_RECORD_STATE` - The record state is invalid. The acknowledgement of delivery could not be completed.

ShareGroupHeartbeat API

The ShareGroupHeartbeat API is used by share group consumers to form a group. The API allows members to advertise their subscriptions and their state. The group coordinator uses it to assign partitions to and revoke partitions from members. This API is also used as a liveness check.

Request schema

The member must set all the top-level fields when it joins for the first time or when an error occurs. Otherwise, it is expected only to fill in the fields which have changed since the last heartbeat.

```
{
  "apiKey": TBD,
  "type": "request",
  "listeners": ["broker"],
  "name": "ShareGroupHeartbeatRequest",
  "validVersions": "0",
  "flexibleVersions": "0+",
  "fields": [
      { "name": "GroupId", "type": "string", "versions": "0+", "entityType": "groupId",
        "about": "The group identifier." },
      { "name": "MemberId", "type": "string", "versions": "0+",
        "about": "The member ID generated by the coordinator. The member ID must be kept during the entire
lifetime of the member." },
      { "name": "MemberEpoch", "type": "int32", "versions": "0+",
        "about": "The current member epoch; 0 to join the group; -1 to leave the group." },
      { "name": "RackId", "type": "string", "versions": "0+",  "nullableVersions": "0+", "default": "null",
        "about": "null if not provided or if it didn't change since the last heartbeat; the rack ID of consumer
otherwise." },
      { "name": "RebalanceTimeoutMs", "type": "int32", "versions": "0+", "default": -1,
        "about": "-1 if it didn't chance since the last heartbeat; the maximum time in milliseconds that the
coordinator will wait on the member to revoke its partitions otherwise." },
      { "name": "SubscribedTopicNames", "type": "[]string", "versions": "0+", "nullableVersions": "0+",
"default": "null",
        "about": "null if it didn't change since the last heartbeat; the subscribed topic names otherwise." }
  ]
}
```

Response schema

The group coordinator will only send the Assignment field when it changes.

```
{
  "apiKey": TBD,
  "type": "response",
  "name": "ShareGroupHeartbeatResponse",
  "validVersions": "0",
  "flexibleVersions": "0+",
  // Supported errors:
  // - GROUP_AUTHORIZATION_FAILED (version 0+)
  // - NOT_COORDINATOR (version 0+)
  // - COORDINATOR_NOT_AVAILABLE (version 0+)
  // - COORDINATOR_LOAD_IN_PROGRESS (version 0+)
  // - INVALID_REQUEST (version 0+)
  // - UNKNOWN_MEMBER_ID (version 0+)
  // - GROUP_MAX_SIZE_REACHED (version 0+)
  "fields": [
    { "name": "ThrottleTimeMs", "type": "int32", "versions": "0+",
      "about": "The duration in milliseconds for which the request was throttled due to a quota violation, or
zero if the request did not violate any quota." },
    { "name": "ErrorCode", "type": "int16", "versions": "0+",
      "about": "The top-level error code, or 0 if there was no error" },
    { "name": "ErrorMessage", "type": "string", "versions": "0+", "nullableVersions": "0+", "default": "null",
      "about": "The top-level error message, or null if there was no error." },
    { "name": "MemberId", "type": "string", "versions": "0+", "nullableVersions": "0+", "default": "null",
      "about": "The member ID generated by the coordinator. Only provided when the member joins with
MemberEpoch == 0." },
    { "name": "MemberEpoch", "type": "int32", "versions": "0+",
      "about": "The member epoch." },
    { "name": "HeartbeatIntervalMs", "type": "int32", "versions": "0+",
      "about": "The heartbeat interval in milliseconds." },
    { "name": "Assignment", "type": "Assignment", "versions": "0+", "nullableVersions": "0+", "default": "null",
      "about": "null if not provided; the assignment otherwise.", "fields": [
        { "name": "Error", "type": "int8", "versions": "0+",
          "about": "The assigned error." },
        { "name": "AssignedTopicPartitions", "type": "[]TopicPartitions", "versions": "0+",
          "about": "The partitions assigned to the member." }
    ]}
  ],
  "commonStructs": [
    { "name": "TopicPartitions", "versions": "0+", "fields": [
        { "name": "TopicId", "type": "uuid", "versions": "0+",
          "about": "The topic ID." },
        { "name": "Partitions", "type": "[]int32", "versions": "0+",
          "about": "The partitions." }
    ]}
  ]
}
```

ShareGroupDescribe API

The ShareGroupDescribe API is used to describe share groups.

Request schema

```
{
  "apiKey": NN,
  "type": "request",
  "listeners": ["broker"],
  "name": "ShareGroupDescribeRequest",
  "validVersions": "0",
  "flexibleVersions": "0+",
  "fields": [
    { "name": "GroupIds", "type": "[]string", "versions": "0+", "entityType": "groupId",
      "about": "The ids of the groups to describe" },
    { "name": "IncludeAuthorizedOperations", "type": "bool", "versions": "0+",
      "about": "Whether to include authorized operations." }
  ]
}
```

Response schema

```
{
  "apiKey": NN,
  "type": "response",
  "name": "ShareGroupDescribeResponse",
  "validVersions": "0",
  "flexibleVersions": "0+",
  // Supported errors:
  // - GROUP_AUTHORIZATION_FAILED (version 0+)
  // - NOT_COORDINATOR (version 0+)
  // - COORDINATOR_NOT_AVAILABLE (version 0+)
  // - COORDINATOR_LOAD_IN_PROGRESS (version 0+)
  // - INVALID_REQUEST (version 0+)
  // - INVALID_GROUP_ID (version 0+)
  // - GROUP_ID_NOT_FOUND (version 0+)
  "fields": [
    { "name": "ThrottleTimeMs", "type": "int32", "versions": "0+",
      "about": "The duration in milliseconds for which the request was throttled due to a quota violation, or
zero if the request did not violate any quota." },
    { "name": "Groups", "type": "[]DescribedGroup", "versions": "0+",
      "about": "Each described group.",
      "fields": [
        { "name": "ErrorCode", "type": "int16", "versions": "0+",
          "about": "The describe error, or 0 if there was no error." },
        { "name": "ErrorMessage", "type": "string", "versions": "0+", "nullableVersions": "0+", "default":
"null",
          "about": "The top-level error message, or null if there was no error." },
        { "name": "GroupId", "type": "string", "versions": "0+", "entityType": "groupId",
          "about": "The group ID string." },
        { "name": "GroupState", "type": "string", "versions": "0+",
          "about": "The group state string, or the empty string." },
        { "name": "GroupEpoch", "type": "int32", "versions": "0+",
          "about": "The group epoch." },
        { "name": "AssignmentEpoch", "type": "int32", "versions": "0+",
          "about": "The assignment epoch." },
        { "name": "AssignorName", "type": "string", "versions": "0+",
          "about": "The selected assignor." },
        { "name": "Members", "type": "[]Member", "versions": "0+",
          "about": "The members.",
          "fields": [
            { "name": "MemberId", "type": "string", "versions": "0+",
              "about": "The member ID." },
            { "name": "InstanceId", "type": "string", "versions": "0+", "nullableVersions": "0+", "default":
"null",
              "about": "The member instance ID." },
            { "name": "RackId", "type": "string", "versions": "0+", "nullableVersions": "0+", "default": "null",
              "about": "The member rack ID." },
            { "name": "MemberEpoch", "type": "int32", "versions": "0+",
              "about": "The current member epoch." },
            { "name": "ClientId", "type": "string", "versions": "0+",
              "about": "The client ID." },
            { "name": "ClientHost", "type": "string", "versions": "0+",
              "about": "The client host." },
            { "name": "SubscribedTopicNames", "type": "[]string", "versions": "0+", "entityType": "topicName",
              "about": "The subscribed topic names." },
            { "name": "Assignment", "type": "Assignment", "versions": "0+",
              "about": "The current assignment." }
          ]},
        { "name": "AuthorizedOperations", "type": "int32", "versions": "0+", "default": "-2147483648",
          "about": "32-bit bitfield to represent authorized operations for this group." }
      ]
    }
  ],
  "commonStructs": [
    { "name": "TopicPartitions", "versions": "0+", "fields": [
      { "name": "TopicId", "type": "uuid", "versions": "0+",
        "about": "The topic ID." },
      { "name": "TopicName", "type": "string", "versions": "0+", "entityType": "topicName",
        "about": "The topic name." },
      { "name": "Partitions", "type": "[]int32", "versions": "0+",
        "about": "The partitions." }
    ]},
    { "name": "Assignment", "versions": "0+", "fields": [
      { "name": "TopicPartitions", "type": "[]TopicPartitions", "versions": "0+",
        "about": "The assigned topic-partitions to the member." },
      { "name": "Error", "type": "int8", "versions": "0+",
        "about": "The assigned error." },
      { "name": "MetadataVersion", "type": "int32", "versions": "0+",
        "about": "The assignor metadata version." },
      { "name": "MetadataBytes", "type": "bytes", "versions": "0+",
        "about": "The assignor metadata bytes." }
    ]}
  ]
}
```

ShareFetch API

The ShareFetch API is used by share group consumers to fetch acquired records from share-partition leaders. It is also possible to piggyback acknowledgements in this request to reduce the number of round trips.

The first request from a share consumer to a share-partition leader broker establishes a share session by setting `MemberId` to the member ID it received from the group coordinator and `ShareSessionEpoch` to 0. Then each subsequent `ShareFetch` or `ShareAcknowledge` request specifies the `MemberId` and increments the `ShareSessionEpoch` by one. When the share consumer wishes to close the share session, it sets `MemberId` to the member ID and `ShareSessionEpoch` to -1.

Request schema

```
{
  "apiKey": NN,
  "type": "request",
  "listeners": ["broker"],
  "name": "ShareFetchRequest",
  "validVersions": "0",
  "flexibleVersions": "0+",
  "fields": [
    { "name": "GroupId", "type": "string", "versions": "0+", "nullableVersions": "0+", "default": "null",
"entityType": "groupId",
      "about": "null if not provided or if it didn't change since the last fetch; the group identifier
otherwise." },
    { "name": "MemberId", "type": "string", "versions": "0+", "nullableVersions": "0+",
      "about": "The member ID." },
    { "name": "ShareSessionEpoch", "type": "int32", "versions": "0+",
      "about": "The current share session epoch: 0 to open a share session; -1 to close it; otherwise
increments for consecutive requests." },
    { "name": "MaxWaitMs", "type": "int32", "versions": "0+",
      "about": "The maximum time in milliseconds to wait for the response." },
    { "name": "MinBytes", "type": "int32", "versions": "0+",
      "about": "The minimum bytes to accumulate in the response." },
    { "name": "MaxBytes", "type": "int32", "versions": "0+", "default": "0x7fffffff", "ignorable": true,
      "about": "The maximum bytes to fetch.  See KIP-74 for cases where this limit may not be honored." },
    { "name": "Topics", "type": "[]FetchTopic", "versions": "0+",
      "about": "The topics to fetch.", "fields": [
      { "name": "TopicId", "type": "uuid", "versions": "0+", "ignorable": true, "about": "The unique topic
ID."},
      { "name": "Partitions", "type": "[]FetchPartition", "versions": "0+",
        "about": "The partitions to fetch.", "fields": [
        { "name": "PartitionIndex", "type": "int32", "versions": "0+",
          "about": "The partition index." },
        { "name": "CurrentLeaderEpoch", "type": "int32", "versions": "0+", "default": "-1", "ignorable": true,
          "about": "The current leader epoch of the partition." },
        { "name": "PartitionMaxBytes", "type": "int32", "versions": "0+",
          "about": "The maximum bytes to fetch from this partition.  See KIP-74 for cases where this limit may
not be honored." },
        { "name": "AcknowledgementBatches", "type": "[]AcknowledgementBatch", "versions": "0+",
          "about": "Record batches to acknowledge.", "fields": [
          { "name": "StartOffset", "type": "int64", "versions": "0+",
            "about": "Start offset of batch of records to acknowledge."},
          { "name": "LastOffset", "type": "int64", "versions": "0+",
            "about": "Last offset (inclusive) of batch of records to acknowledge."},
          { "name": "GapOffsets", "type": "[]int64", "versions": "0+",
            "about": "Array of offsets in this range which do not correspond to records."},
          { "name": "AcknowledgeType", "type": "int8", "versions": "0+", "default": "0",
            "about": "The type of acknowledgement - 0:Accept,1:Release,2:Reject."}
        ]}
    ]},
    { "name": "ForgottenTopicsData", "type": "[]ForgottenTopic", "versions": "0+", "ignorable": false,
      "about": "The partitions to remove from this share session.", "fields": [
      { "name": "TopicId", "type": "uuid", "versions": "0+", "ignorable": true, "about": "The unique topic
ID."},
      { "name": "Partitions", "type": "[]int32", "versions": "0+",
        "about": "The partitions indexes to forget." }
    ]}
  ]
}
```

Response schema

```
{
  "apiKey": NN,
  "type": "response",
  "name": "ShareFetchResponse",
  "validVersions": "0",
  "flexibleVersions": "0+",
  // Supported errors:
  // - GROUP_AUTHORIZATION_FAILED (version 0+)
  // - TOPIC_AUTHORIZATION_FAILED (version 0+)
  // - UNKNOWN_TOPIC_OR_PARTITION (version 0+)
  // - NOT_LEADER_OR_FOLLOWER (version 0+)
  // - UNKNOWN_TOPIC_ID (version 0+)
  // - INVALID_RECORD_STATE (version 0+)
  // - KAFKA_STORAGE_ERROR (version 0+)
  // - CORRUPT_MESSAGE (version 0+)
  // - INVALID_REQUEST (version 0+)
  // - UNKNOWN_SERVER_ERROR (version 0+)
  "fields": [
    { "name": "ThrottleTimeMs", "type": "int32", "versions": "0+", "ignorable": true,
      "about": "The duration in milliseconds for which the request was throttled due to a quota violation, or
zero if the request did not violate any quota." },
    { "name": "ErrorCode", "type": "int16", "versions": "0+", "ignorable": true,
      "about": "The top level response error code." },
    { "name": "Responses", "type": "[]ShareFetchableTopicResponse", "versions": "0+",
      "about": "The response topics.", "fields": [
      { "name": "TopicId", "type": "uuid", "versions": "0+", "ignorable": true, "about": "The unique topic
ID."},
      { "name": "Partitions", "type": "[]PartitionData", "versions": "0+",
        "about": "The topic partitions.", "fields": [
        { "name": "PartitionIndex", "type": "int32", "versions": "0+",
          "about": "The partition index." },
        { "name": "ErrorCode", "type": "int16", "versions": "0+",
          "about": "The error code, or 0 if there was no fetch error." },
        { "name": "AcknowledgeErrorCode", "type": "int16", "versions": "0+",
          "about": "The acknowledge error code, or 0 if there was no acknowledge error." },
        { "name": "CurrentLeader", "type": "LeaderIdAndEpoch", "versions": "0+", "fields": [
          { "name": "LeaderId", "type": "int32", "versions": "0+",
            "about": "The ID of the current leader or -1 if the leader is unknown." },
          { "name": "LeaderEpoch", "type": "int32", "versions": "0+",
            "about": "The latest known leader epoch." }
        ]},
        { "name": "Records", "type": "records", "versions": "0+", "nullableVersions": "0+", "about": "The
record data."},
        { "name": "AcquiredRecords", "type": "[]AcquiredRecords", "versions": "0+", "about": "The acquired
records.", "fields":  [
          {"name": "BaseOffset", "type":  "int64", "versions": "0+", "about": "The earliest offset in this
batch of acquired records."},
          {"name": "LastOffset", "type": "int64", "versions": "0+", "about": "The last offset of this batch of
acquired records."},
          {"name": "DeliveryCount", "type": "int16", "versions": "0+", "about": "The delivery count of this
batch of acquired records."}
        ]}
      ]}
    ]},
    { "name": "NodeEndpoints", "type": "[]NodeEndpoint", "versions": "0+",
      "about": "Endpoints for all current leaders enumerated in PartitionData with error
NOT_LEADER_OR_FOLLOWER.", "fields": [
      { "name": "NodeId", "type": "int32", "versions": "0+",
        "mapKey": true, "entityType": "brokerId", "about": "The ID of the associated node." },
      { "name": "Host", "type": "string", "versions": "0+",
        "about": "The node's hostname." },
      { "name": "Port", "type": "int32", "versions": "0+",
        "about": "The node's port." },
      { "name": "Rack", "type": "string", "versions": "0+", "nullableVersions": "0+", "default": "null",
        "about": "The rack of the node, or null if it has not been assigned to a rack." }
    ]}
  ]
}
```

ShareAcknowledge API

The ShareAcknowledge API is used by share group consumers to acknowledge delivery of records with share-partition leaders.

Request schema

```
{
  "apiKey": NN,
  "type": "request",
  "listeners": ["broker"],
  "name": "ShareAcknowledgeRequest",
  "validVersions": "0",
  "flexibleVersions": "0+",
  "fields": [
    { "name": "MemberId", "type": "string", "versions": "0+", "nullableVersions": "0+",
      "about": "The member ID." },
    { "name": "ShareSessionEpoch", "type": "int32", "versions": "0+",
      "about": "The current share session epoch: 0 to open a share session; -1 to close it; otherwise
increments for consecutive requests." },
    { "name": "Topics", "type": "[]AcknowledgeTopic", "versions": "0+",
      "about": "The topics containing records to acknowledge.", "fields": [
      { "name": "TopicId", "type": "uuid", "versions": "0+", "about": "The unique topic ID."},
      { "name": "Partitions", "type": "[]AcknowledgePartition", "versions": "0+",
        "about": "The partitions containing records to acknowledge.", "fields": [
        { "name": "PartitionIndex", "type": "int32", "versions": "0+",
          "about": "The partition index." },
        { "name": "AcknowledgementBatches", "type": "[]AcknowledgementBatch", "versions": "0+",
          "about": "Record batches to acknowledge.", "fields": [
          { "name": "StartOffset", "type": "int64", "versions": "0+",
            "about": "Start offset of batch of records to acknowledge."},
          { "name": "LastOffset", "type": "int64", "versions": "0+",
            "about": "Last offset (inclusive) of batch of records to acknowledge."},
          { "name": "GapOffsets", "type": "[]int64", "versions": "0+",
            "about": "Array of offsets in this range which do not correspond to records."},
          { "name": "AcknowledgeType", "type": "int8", "versions": "0+", "default": "0",
            "about": "The type of acknowledgement - 0:Accept,1:Release,2:Reject."}
        ]}
      ]}
    ]}
  ]
}
```

Response schema

```
{
  "apiKey": NN,
  "type": "response",
  "name": "ShareAcknowledgeResponse",
  "validVersions": "0",
  "flexibleVersions": "0+",
  // Supported errors:
  // - GROUP_AUTHORIZATION_FAILED (version 0+)
  // - TOPIC_AUTHORIZATION_FAILED (version 0+)
  // - UNKNOWN_TOPIC_OR_PARTITION (version 0+)
  // - NOT_LEADER_OR_FOLLOWER (version 0+)
  // - UNKNOWN_TOPIC_ID (version 0+)
  // - INVALID_RECORD_STATE (version 0+)
  // - KAFKA_STORAGE_ERROR (version 0+)
  // - INVALID_REQUEST (version 0+)
  // - UNKNOWN_SERVER_ERROR (version 0+)
  "fields": [
    { "name": "ThrottleTimeMs", "type": "int32", "versions": "0+", "ignorable": true,
      "about": "The duration in milliseconds for which the request was throttled due to a quota violation, or
zero if the request did not violate any quota." },
    { "name": "ErrorCode", "type": "int16", "versions": "0+", "ignorable": true,
      "about": "The top level response error code." },
    { "name": "Responses", "type": "[]ShareAcknowledgeTopicResponse", "versions": "0+",
      "about": "The response topics.", "fields": [
      { "name": "TopicId", "type": "uuid", "versions": "0+", "ignorable": true, "about": "The unique topic
ID."},
      { "name": "Partitions", "type": "[]PartitionData", "versions": "0+",
        "about": "The topic partitions.", "fields": [
        { "name": "PartitionIndex", "type": "int32", "versions": "0+",
          "about": "The partition index." },
        { "name": "ErrorCode", "type": "int16", "versions": "0+",
          "about": "The error code, or 0 if there was no error." },
        { "name": "CurrentLeader", "type": "LeaderIdAndEpoch", "versions": "0+", "fields": [
          { "name": "LeaderId", "type": "int32", "versions": "0+",
            "about": "The ID of the current leader or -1 if the leader is unknown." },
          { "name": "LeaderEpoch", "type": "int32", "versions": "0+",
            "about": "The latest known leader epoch." }
        ]}
      ]}
    ]},
    { "name": "NodeEndpoints", "type": "[]NodeEndpoint", "versions": "0+",
      "about": "Endpoints for all current leaders enumerated in PartitionData with error
NOT_LEADER_OR_FOLLOWER.", "fields": [
      { "name": "NodeId", "type": "int32", "versions": "0+",
        "mapKey": true, "entityType": "brokerId", "about": "The ID of the associated node." },
      { "name": "Host", "type": "string", "versions": "0+",
        "about": "The node's hostname." },
      { "name": "Port", "type": "int32", "versions": "0+",
        "about": "The node's port." },
      { "name": "Rack", "type": "string", "versions": "0+", "nullableVersions": "0+", "default": "null",
        "about": "The rack of the node, or null if it has not been assigned to a rack." }
    ]}
  ]
}
```

AlterShareGroupOffsets API

The AlterShareGroupOffsets API is used to alter the share-partition start offsets for the share-partitions in a share group. The share-partition leader handles this API.

Request schema

```
{
  "apiKey": NN,
  "type": "request",
  "listeners": ["broker"],
  "name": "AlterShareGroupOffsetsRequest",
  "validVersions": "0",
  "flexibleVersions": "0+",
  "fields": [
    { "name": "GroupId", "type": "string", "versions": "0+", "entityType": "groupId",
      "about": "The group identifier." },
    { "name": "Topics", "type": "[]AlterShareGroupOffsetsRequestTopic", "versions": "0+",
      "about": "The topics to alter offsets for.",  "fields": [
      { "name": "TopicName", "type": "string", "versions": "0+", "entityType": "topicName", "mapKey": true,
        "about": "The topic name." },
      { "name": "Partitions", "type": "[]AlterShareGroupOffsetsRequestPartition", "versions": "0+",
        "about": "Each partition to alter offsets for.", "fields": [
        { "name": "PartitionIndex", "type": "int32", "versions": "0+",
          "about": "The partition index." },
        { "name": "StartOffset", "type": "int64", "versions": "0+",
          "about": "The share-partition start offset." }
      ]}
    ]}
  ]
}
```

Response schema

```
{
  "apiKey": NN,
  "type": "response",
  "name": "AlterShareGroupOffsetsResponse",
  "validVersions": "0",
  "flexibleVersions": "0+",
  // Supported errors:
  // - GROUP_AUTHORIZATION_FAILED (version 0+)
  // - NOT_COORDINATOR (version 0+)
  // - COORDINATOR_NOT_AVAILABLE (version 0+)
  // - COORDINATOR_LOAD_IN_PROGRESS (version 0+)
  // - GROUP_ID_NOT_FOUND (version 0+)
  // - GROUP_NOT_EMPTY (version 0+)
  // - KAFKA_STORAGE_ERROR (version 0+)
  // - INVALID_REQUEST (version 0+)
  // - UNKNOWN_SERVER_ERROR (version 0+)
  "fields": [
    { "name": "ThrottleTimeMs", "type": "int32", "versions": "0+", "ignorable": true,
      "about": "The duration in milliseconds for which the request was throttled due to a quota violation, or
zero if the request did not violate any quota." },
    { "name": "Responses", "type": "[]AlterShareGroupOffsetsResponseTopic", "versions": "0+",
      "about": "The results for each topic.", "fields": [
      { "name": "TopicName", "type": "string", "versions": "0+", "entityType": "topicName",
        "about": "The topic name." },
      { "name": "TopicId", "type": "uuid", "versions": "0+", "ignorable": true,
        "about": "The unique topic ID." },
      { "name": "Partitions", "type": "[]AlterShareGroupOffsetsResponsePartition", "versions": "0+", "fields": [
        { "name": "PartitionIndex", "type": "int32", "versions": "0+",
          "about": "The partition index." },
        { "name": "ErrorCode", "type": "int16", "versions": "0+",
          "about": "The error code, or 0 if there was no error." },
        { "name": "ErrorMessage", "type": "string", "versions": "0+", "nullableVersions": "0+", "ignorable":
true, "default": "null",
          "about": "The error message, or null if there was no error." }
      ]}
    ]}
  ]
}
```

DeleteShareGroupOffsets API

The DeleteShareGroupOffsets API is used to delete the offsets for the share-partitions in a share group. The share-partition leader handles this API.

Request schema

```
{
  "apiKey": NN,
  "type": "request",
  "listeners": ["broker"],
  "name": "DeleteShareGroupOffsetsRequest",
  "validVersions": "0",
  "flexibleVersions": "0+",
  "fields": [
    { "name": "GroupId", "type": "string", "versions": "0+", "entityType": "groupId",
      "about": "The group identifier." },
    { "name": "Topics", "type": "[]DeleteShareGroupOffsetsRequestTopic", "versions": "0+",
      "about": "The topics to delete offsets for.",  "fields": [
      { "name": "TopicName", "type": "string", "versions": "0+", "entityType": "topicName",
        "about": "The topic name." },
      { "name": "Partitions", "type": "[]int32", "versions": "0+",
        "about": "The partitions." }
      ]}
    ]}
  ]
}
```

Response schema

```
{
  "apiKey": NN,
  "type": "response",
  "name": "DeleteShareGroupOffsetsResponse",
  "validVersions": "0",
  "flexibleVersions": "0+",
  // Supported errors:
  // - GROUP_AUTHORIZATION_FAILED (version 0+)
  // - NOT_COORDINATOR (version 0+)
  // - COORDINATOR_NOT_AVAILABLE (version 0+)
  // - COORDINATOR_LOAD_IN_PROGRESS (version 0+)
  // - GROUP_ID_NOT_FOUND (version 0+)
  // - GROUP_NOT_EMPTY (version 0+)
  // - KAFKA_STORAGE_ERROR (version 0+)
  // - INVALID_REQUEST (version 0+)
  // - UNKNOWN_SERVER_ERROR (version 0+)
  "fields": [
    { "name": "ThrottleTimeMs", "type": "int32", "versions": "0+", "ignorable": true,
      "about": "The duration in milliseconds for which the request was throttled due to a quota violation, or
zero if the request did not violate any quota." },
    { "name": "Responses", "type": "[]DeleteShareGroupOffsetsResponseTopic", "versions": "0+",
      "about": "The results for each topic.", "fields": [
      { "name": "TopicName", "type": "string", "versions": "0+", "entityType": "topicName",
        "about": "The topic name." },
      { "name": "TopicId", "type": "uuid", "versions": "0+", "ignorable": true,
        "about": "The unique topic ID." },
      { "name": "Partitions", "type": "[]DeleteShareGroupOffsetsResponsePartition", "versions": "0+", "fields":
[
        { "name": "PartitionIndex", "type": "int32", "versions": "0+",
          "about": "The partition index." },
        { "name": "ErrorCode", "type": "int16", "versions": "0+",
          "about": "The error code, or 0 if there was no error." },
        { "name": "ErrorMessage", "type": "string", "versions": "0+", "nullableVersions": "0+", "ignorable":
true, "default": "null",
          "about": "The error message, or null if there was no error." }
      ]}
    ]}
  ]
}
```

DescribeShareGroupOffsets API

The DescribeShareGroupOffsets API is used to describe the offsets for the share-partitions in a share group. The share-partition leader handles this API.

Request schema

```
{
  "apiKey": NN,
  "type": "request",
  "listeners": ["broker"],
  "name": "DescribeShareGroupOffsetsRequest",
  "validVersions": "0",
  "flexibleVersions": "0+",
  "fields": [
    { "name": "GroupId", "type": "string", "versions": "0+", "entityType": "groupId",
      "about": "The group identifier." },
    { "name": "Topics", "type": "[]DescribeShareGroupOffsetsRequestTopic", "versions": "0+",
      "about": "The topics to describe offsets for.",  "fields": [
      { "name": "TopicName", "type": "string", "versions": "0+", "entityType": "topicName",
        "about": "The topic name." },
      { "name": "Partitions", "type": "[]int32", "versions": "0+",
        "about": "The partitions." }
      ]}
    ]}
  ]
}
```

Response schema

```
{
  "apiKey": NN,
  "type": "response",
  "name": "DescribeShareGroupOffsetsResponse",
  "validVersions": "0",
  "flexibleVersions": "0+",
  // Supported errors:
  // - GROUP_AUTHORIZATION_FAILED (version 0+)
  // - NOT_COORDINATOR (version 0+)
  // - COORDINATOR_NOT_AVAILABLE (version 0+)
  // - COORDINATOR_LOAD_IN_PROGRESS (version 0+)
  // - GROUP_ID_NOT_FOUND (version 0+)
  // - INVALID_REQUEST (version 0+)
  // - UNKNOWN_SERVER_ERROR (version 0+)
  "fields": [
    { "name": "ThrottleTimeMs", "type": "int32", "versions": "0+", "ignorable": true,
      "about": "The duration in milliseconds for which the request was throttled due to a quota violation, or
zero if the request did not violate any quota." },
    { "name": "Responses", "type": "[]DescribeShareGroupOffsetsResponseTopic", "versions": "0+",
      "about": "The results for each topic.", "fields": [
      { "name": "TopicName", "type": "string", "versions": "0+", "entityType": "topicName",
        "about": "The topic name." },
      { "name": "TopicId", "type": "uuid", "versions": "0+", "ignorable": true,
        "about": "The unique topic ID." },
      { "name": "Partitions", "type": "[]DescribeShareGroupOffsetsResponsePartition", "versions": "0+",
"fields": [
        { "name": "PartitionIndex", "type": "int32", "versions": "0+",
          "about": "The partition index." },
        { "name": "StartOffset", "type": "int64", "versions": "0+",
          "about": "The share-partition start offset."},
        { "name": "ErrorCode", "type": "int16", "versions": "0+",
          "about": "The error code, or 0 if there was no error." },
        { "name": "ErrorMessage", "type": "string", "versions": "0+", "nullableVersions": "0+", "ignorable":
true, "default": "null",
          "about": "The error message, or null if there was no error." }
      ]}
    ]}
  ]
}
```

Records

This section describes the new record types.

Group metadata

In order to coexist properly with consumer groups, the group metadata records for share groups are persisted by the group coordinator to the compacted `__consumer_offsets` topic.

For each share group, a single `ConsumerGroupMetadata` record is written. When the group is deleted, a tombstone record is written.

ConsumerGroupMetadataKey

This is included for completeness. There is no change to this record.

```
{
  "type": "data",
  "name": "ConsumerGroupMetadataKey",
  "validVersions": "3",
  "flexibleVersions": "none",
  "fields": [
    { "name": "GroupId", "type": "string", "versions": "3",
      "about": "The group id." }
  ]
}
```

ConsumerGroupMetadataValue

A new version of the record value is introduced contains the `Type` field. For a share group, the type will be `"share"` . For a consumer group, the type can be omitted (null) or `"consumer"` .

```
{
  "type": "data",
  "name": "ConsumerGroupMetadataValue",
  "validVersions": "0-1",
  "flexibleVersions": "0+",
  "fields": [
    { "name": "Epoch", "type": "int32", "versions": "0+",
      "about": "The group epoch." },
    // Version 1 adds Type field
    { "name": "Type", "type": "string", "versions": "1+", "nullableVersions": "1+",
      "about": "The group type - null indicates consumer group." }
  ]
}
```

Share-partition state

The existing `ControlRecordKey` is used for the key of the **SHARE_CHECKPOINT** (Type = 5) and **SHARE_DELTA** (Type = 6) control records.

ShareCheckpointValue

```
{
  "type": "data",
  "name": "ShareCheckpointValue",
  "validVersions": "0",
  "flexibleVersions": "none",
  "fields": [
   { "name": "GroupId", "type": "string", "versions": "0",
     "about": "The group identifier." },
   { "name": "CheckpointEpoch", "type": "uint16", "versions": "0",
     "about": "The checkpoint epoch, increments with each checkpoint." },
   { "name": "StartOffset", "type": "int64", "versions": "0",
     "about": "The share-partition start offset." },
   { "name": "EndOffset", "type": "int64", "versions": "0",
     "about": "The share-partition end offset." },
   { "name": "States", "type": "[]State", "versions": "0", "fields": [
      { "name": "BaseOffset", "type": "int64", "versions": "0",
        "about": "The base offset of this state batch." },
      { "name": "LastOffset", "type": "int64", "versions": "0",
        "about": "The last offset of this state batch." },
      { "name": "State", "type": "int8", "versions": "0",
        "about": "The state - 0:Available,2:Acked,4:Archived." },
      { "name": "DeliveryCount", "type": "int16", "versions": "0",
        "about": "The delivery count." }
   ]}
  ]
}
```

ShareDeltaValue

```
{
  "type": "data",
  "name": "ShareDeltaValue",
  "validVersions": "0",
  "flexibleVersions": "none",
  "fields": [
   { "name": "GroupId", "type": "string", "versions": "0",
     "about": "The group identifier." },
   { "name": "CheckpointEpoch", "type": "uint16", "versions": "0",
     "about": "The checkpoint epoch, increments with each checkpoint." },
   { "name": "States", "type": "[]State", "versions": "0", "fields": [
      { "name": "BaseOffset", "type": "int64", "versions": "0",
        "about": "The base offset of this state batch." },
      { "name": "LastOffset", "type": "int64", "versions": "0",
        "about": "The last offset of this state batch." },
      { "name": "State", "type": "int8", "versions": "0",
        "about": "The state - 0:Available,2:Acked,4:Archived." },
      { "name": "DeliveryCount", "type": "int16", "versions": "0",
        "about": "The delivery count." }
   ]}
  ]
}
```

Index structure for locating share-partition state

More information needs to be added to describe how the index for locating the share-partition state is arranged.

Metrics

Broker Metrics

The following new broker metrics should be added:

Metric Name	Type	Group	Tags	Description	JMX Bean
group-count	Gauge	group-coordinator-metrics	`protocol: share`	The total number of share groups managed by group coordinator.	`kafka.server:type=group-coordinator-metrics, name=group-count,protocol=share`

rebalance (rebalance-rate and rebalance-count)	Meter	group-coordinator-metrics	`protocol: share`	The total number of share group rebalances count and rate.	`kafka.server:type=group-coordinator-metrics, name=rebalance-rate,protocol=share`
					`kafka.server:type=group-coordinator-metrics, name=rebalance-count,protocol=share`

num-partitions	Gauge	group-coordinator-metrics	`protocol: share`	The number of share partitions managed by group coordinator.	`kafka.server:type=group-coordinator-metrics, name=num-partitions,protocol=share`

group-count	Gauge	group-coordinator-metrics	`protocol: share`
			`state: {empty|stable|dead}`
				The number of share groups in respective state.	`kafka.server:type=group-coordinator-metrics, name=group-count,protocol=share,state={empty|stable|dead}`

share-acknowledgement (share-acknowledgement-rate and share-acknowledgement-count)	Meter	group-coordinator-metrics	`protocol: share`	The total number of offsets acknowledged for share groups.	`kafka.server:type=group-coordinator-metrics, name=share-acknowledgement-rate,protocol=share`
					`kafka.server:type=group-coordinator-metrics, name=share-acknowledgement-count,protocol=share`

record-acknowledgement (record-acknowledgement-rate and record-acknowledgement-count)	Meter	group-coordinator-metrics	`protocol: share`
			`ack-type: {accept, release, reject}`
				The number of records acknowledged per acknowledgement type.	`kafka.server:type=group-coordinator-metrics, name=record-acknowledgement-rate, protocol=share,ack-type={accept,release,reject}`
					`kafka.server:type=group-coordinator-metrics, name=record-acknowledgement-count, protocol=share,ack-type={accept,release,reject}`

partition-load-time (partition-load-time-avg and partition-load-time-max)	Meter	group-coordinator-metrics	`protocol: share`	The time taken to load the share partitions.	`kafka.server:type=group-coordinator-metrics, name=partition-load-time-avg,protocol=share`
					`kafka.server:type=group-coordinator-metrics, name=partition-load-time-max,protocol=share`

Future Work

There are some obvious extensions to this idea which are not included in this KIP in order to keep the scope manageable.

This KIP introduces delivery counts and a maximum number of delivery attempts. An obvious future extension is the ability to copy records that failed to be delivered onto a dead-letter queue. This would of course give a way to handle poison messages without them permanently blocking processing.

The focus in this KIP is on sharing rather than ordering. The concept can be extended to give key-based ordering so that partial ordering and fine-grained sharing can be achieved at the same time.

For topics in which share groups are the only consumption model, it would be nice to be able to have the SPSO of the share-partitions taken in to consideration when cleaning the log and advancing the log start offset.

It would also be possible to have share-group configuration to control the maximum time-to-live for records and automatically archive them at this time.

Finally, this KIP does not include support for acknowledging delivery using transactions for exactly-once semantics. Conceptually, this is quite straightforward but would take changes to the API.

Compatibility, Deprecation, and Migration Plan

Kafka Broker Migration

This KIP builds upon KIP-848 which introduced the new group coordinator and the new records for the `__consumer_offsets` topic. The pre-KIP-848 group coordinator will not recognize the new records, so this downgrade is not supported.

Downgrading to a software version that supports the new group coordinator but does not support share groups is supported. This KIP adds a new version for the `ConsumerGroupMetadataValue` record to include the group type. If the software version does not understand the v1 record type, it will assume the records apply to a consumer group of the same name. We should make sure this is a harmless situation.

More information need to be added here based on the share-partition persistence mechanism. Details are still under consideration here.

Test Plan

The feature will be throughly tested with unit, integration and system tests. We will also carry out performance testing both to understand the performance of share groups, and also to understand the impact on brokers with this new feature.

Rejected Alternatives

Share group consumers use KafkaConsumer

In this option, the regular `KafkaConsumer` was used by consumers to consume records from a share group, using a configuration parameter `group.type` to choose between using a share group or a consumer group. While this means that existing Kafka consumers can trivially make use of share groups, there are some obvious downsides:

1. An application using `KafkaConsumer` with a consumer group could be switched to a share group with very different semantics with just a configuration change. There is almost no chance that the application would work correctly.
2. Libraries such as Kafka Connect which embed Kafka consumers while not work correctly with share groups without code changes beyond changing the configuration. As a result, there is a risk of breaking connectors due to misconfiguration using the `group.type` configuration property.
3. More than half of the `KafkaConsumer` methods do not make sense for share groups introducing a lot of unnecessary cruft.

As a result, the KIP now proposes an entirely different class `KafkaShareConsumer` which gives a very similar interface as `KafkaConsumer` but eliminates the downsides listed above.