# KIP-X: Introduce a cooperative consumer processing semantic



## Status

**Current state**: *Draft*

**Discussion thread**: TBD

**JIRA**: *TBD*

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

## Motivation

Consumer semantics is very useful for distributed data processing in Kafka, however the granularity of parallelism doesn't sometimes satisfy the business need. To avoid peak traffic affecting production, Kafka users would normally do the capacity planning beforehand to allow 5X ~ 10X future traffic increase. This aims to avoid hitting the scalability bottleneck at the best effort, but still by chance that eventually the traffic goes beyond the original planning (which may be a good thing 🙂). One solution people have considered is to do online partition expanding. The proposal was not continuing to evolve due to its complexity and operation overhead. A second painful option is to switch input topic on the fly. To do that, job owner needs to feed a new topic with more number of partitions, setup a mirrored job and make sure the metrics are aligned for the A/B jobs, eventually make the switch. As of today, this whole process is manual, cumbersome and error-prone for most companies that have critical SLAs.

In the infra cost perspective, pre-define an impractical high number of partitions will definitely increase the network traffic as more metadata and replication will be needed. Besides extra money paid, the operation overhead increases while maintaining broker cluster in good shape with more topic partitions beyond necessity. It's been a known pain point for Kafka streaming processing scalability which is of great value to be resolved, as the number of processing tasks match the number of input partitions. Therefore, Kafka Streams jobs' overhead increases with the number of partition increase at the same time.

Also in reality for use cases such as Kafka Streams, online partition expansion is impossible for stateful operations, as the number of partition for changelog /repartition topics are fixed. This means we couldn't change the input partitions gracefully once the job is up and running. To be able to scale up, we have to cleanup any internal topics to continue processing, which means extra downtime and data loss. By scaling on the client side directly, the operation becomes much easier.

Today most consumer based processing model honors the partition level ordering. However, ETL operations such as join, aggregation and so on are per-key level, so the relative order across different keys does not require to be maintained, except for user customized operations. Many organizations are paying more system guarantee than what they actually need.

The proposal here, is to decouple the consumption threads and physical partitions count, by making consumers capable of collaborating on individual topic partition. There are a couple of benefits compared with existing model:

1. Data consume and produce scales are no longer coupled. This means we could save money by configuring a reasonable topic with decent amount of partitions to save cost.
2. Better avoid partition level hotkeys. When a specific key is processing really slow, the decoupled key based consumption could bypass it and make progress on other keys.

3. No operation overhead for scaling out in extreme cases. Users just need to add more consumer/stream capacity to unblock even there are a few partitions available.

# Proposed Changes

We want to clarify beforehand that this KIP would be a starting point of a transformational change on the consumer consumption semantics. It's not possible to have all the design details rolling out in one shot. Instead, the focus is to define a clear roadmap of what things need to be done, and illustrate the long term plan while getting some concrete tasks in starting steps. There will also be follow-up KIPs and design docs, so stay tuned.

## Use Case

As stated above, the scaling cap for consumer based application is the number of input partitions. In an extreme scenario when there is one single input partition with two consumers, one consumer must remain idle. If the single box consumer **could not keep up the speed of processing**, there is no other solution but lagging. It would be ideal we could co-process data within one partition by two consumers when **the partition level order is not required**, such that we could add as many consumer instances as we want.

So this cooperative consumption model applies with following limitations:

1. The bottleneck is on the application processing, not data consumption throughput which could be caused by other reasons such as network saturation of broker.
2. The processing semantic does not require partition level order, otherwise only one consumer could work on the input sequentially without parallelism.

For pt 2, there are also different requirements for stateless and stateful operations, such as whether **key level ordering** is required or not. Naturally speaking, with a requirement of key level ordering, the broker needs to allocate the same message key to the same consumer within one generation as a must. For stateless operation which doesn't care about the ordering at all, broker could just do a  round robin assignment of records to fetch requests.

## Proposed Roadmap

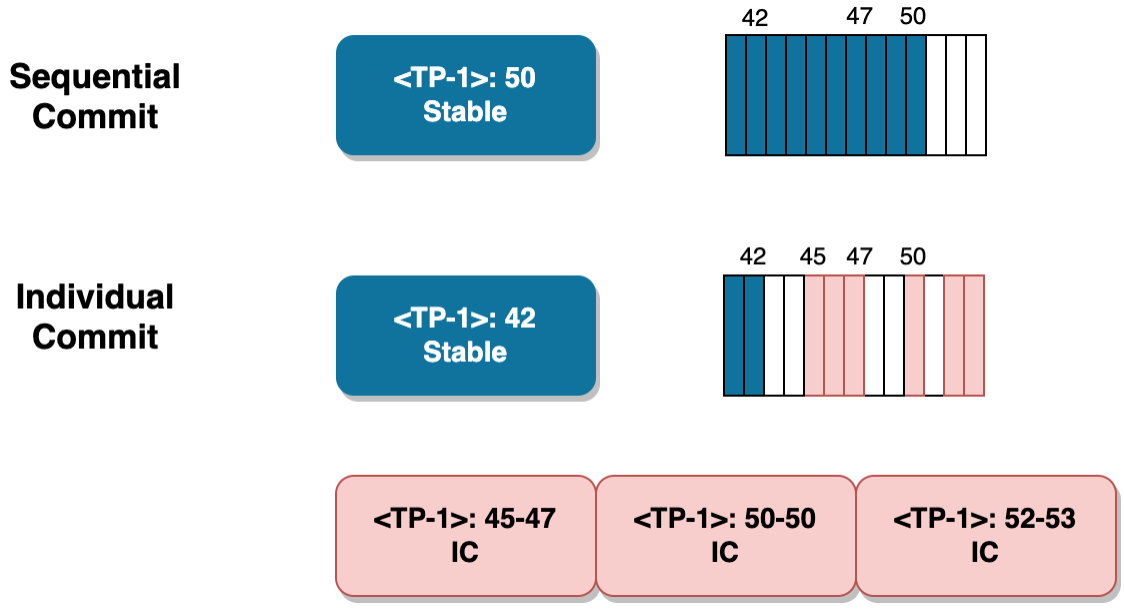
We would start from supporting a new offset commit semantics as this unblocks the potential to process data regardless of partition level ordering. The stateful operation support is of more value in nature, so concurrently we could add supports on key based filtering with fetch calls. The optimized stateless and transactional supports have to be built on top of the first two steps.

| Stage name | Goal | Dependency |
|---|---|---|
| Individual commit | Create a generic offset commit model beyond current partition  single offset mapping. | No |
| Key filtering based fetch | Add capability to FetchRequest with specific hashed key range or specific keys | No |
| Rebalance support for concurrent assignment | Add assignor support to allow splitting single topic partition with different hash range | Key based filtering |
| Transactional support | Incorporate individual commit into the transaction processing model | Key based filtering |
| Support cooperative fetch on broker | Round robin assign data to cooperative consumer fetches when there is no key-range specified | Rebalance support |

## High Level Design

### Individual Commit

*** Note that we will use term **IC** to refer to the **individual commit** for followup discussion.




The logic behind individual commit is very straightforward. For sequential commit each topic partition will only point to one offset number, while under individual commit it is possible to have "gaps" in between. The "stable offset" is a borrowed concept from transaction semantic which is just for illustration purpose, whose purpose is to mark the position where all the messages before it are already committed. The `IC` blocks refer to the individual commits that are marking records already processed.

In the individual commit mode, the offset metadata shall grow much quicker and harder to predict. The log needs to be highly compacted to avoid disk waste, which is different requirement from the existing consumer offset topic. Thus, we propose to add another internal topic called `__individual_commit_offsets` which stores the individual commits specifically, and call the current __consumed_offsets topic the primary offset topic. Furthermore, this isolation should make the IC feature rollout more controllable by avoiding messing up stable offset in primary and achieve at-least-once when we need to delete the corrupted IC offset topic. The individual commit offset topic shall be required to co-locate with the __consumed_offsets topic, which means it has to share the configuration of number of partitions, replication factor and min replicas as primary offset topic.

The offset commit and offset fetch workflow will be changed under individual commit mode.

### Offset Commit

The IC offset will be appended into corresponding offset log as normal, however the broker shall maintain an in-memory state to keep track of the individual commits that are written but not yet cleaned up. To eventually compact this topic, each IC record will go through the coordinator memory to check whether there could be records deleted. In the illustration scenario,

- If there is an IC [48-49] then we no longer need to maintain IC ranges [45-47] and [50-50]. We shall append 2 null records to delete [45-47] and [50-50] and augment the original IC into [45-50]
- If there is an IC [43-44], we are good to move stable offset forward. The operation is like:
    - Push stable offset to 47 in the primary offset topic
    - Append a record to remove IC [45-47] in the IC offset topic

There is also a math we need to do which is how frequent we should commit. The default max request size for Kafka is 1MB, which means we could not include more than 1MB / 16b  60,000 ranges within a request. This means if we are in auto commit mode, the consumer has to breakdown commits if it already accumulates a large number of ranges. It's also undesirable to let stable offset make no progress for a long time. For the sake of smooth progress, we define a new config on the max ranges being collected before triggering a commit call.

Also we need to take care of the memory bound for maintaining the individual offsets. If there are too many of them, we eventually couldn't store all in the heap. One solution is to block progress of the active group when the number of IC offsets are too large, which usually indicates a dead member within the group. The coordinator will callout rebalance in this extreme case to make sure progress could be made again.

### Offset Fetch

During a consumer startup in sequential commit mode, it will attempt to fetch the stored offsets before resuming work. This would be the same step for IC mode, only the consumer will be aware of both the individual offsets and stable offsets now. The individual offsets serve as a reference when user calls #poll(), such that if there is any record that's already committed, it will be filtered and the polled records will only contain committed ones. When doing the offset commit, consumer will do a "gap amending" too.

Imagine a scenario when a consumer fetches based on below setup:

```
Offset: stable: 40, IC: [43-45], [48-49]
Fetched range: [40-50]
```

Consumer will commit only the processed offsets:

```
[41-42], [46-47], [50-50]
```

which matches the offset commit semantic on always committing offsets that are processed within current batch.

In order to cleanup unnecessary IC offsets, the OffsetCommitResponse shall also contain the information about the latest stable offset, so that consumers could safely delete any IC offsets that fall behind stable offset.

## Key Based Filtering

In order to allow sharing between the consumers, broker has to be able to distinguish data records by keys. For a generic purpose, the easy way to assign records by keys is to get a hashing range and allow range split when having multiple consumers talking to the same partition. This would be our starting point to support data sharing between consumers. Suppose we do a key hashing to a space of [0, Long.MAX] while two consumers are assigned to the same partition, they will each get key range [0, Long.MAX/2 - 1] and [Long.MAX/2, Long.MAX]. This hash range assignment decision will be made during rebalance phase.

With the key based hashing, brokers will load topic partitions data into the main memory and do the partitioning work based on the key hashed range. When a FetchRequest comes with specific key range, broker will only reply the ranges that meet the request need. This workflow could be implemented as a generic server side filtering, and KIP-283 is already a good example where we also attempt to load data into memory for down-conversion.

There is obvious performance penalty compared with zero-copy mechanism for existing consumer fetching, however the value of unblocking new use cases motivates us to figure out more optimizations in the long run. In the near term, we could consider cache the filter results of one batch of data so that when other fetch requests fall into this range, we could serve in-memory result instead of doing a repetitive fetch. For example, if a fetch request coming in to ask for range [40-50], but only [40, 44] are satisfying its hash range. The result of [45, 49] will be saved as message keys,  available for next fetch request instead of evicting immediately. We also recommend a config to control how much memory we would like to spend on this feature, which is called *max.bytes.key.range.hashing.* When the cache is sufficient with space and the Fetch request is hanging for new data, producer results could write directly into the cache and do the filtering in order to further save the disk read time.

Also user could opt to choose either load the message key or the entire message into the main memory depending on the understanding of business needs since the variation of message value could be huge. This option would be introduced as a topic level config. Either way, once a cached result is being sent out, it will be evicted immediately to leave room for the current or future key range filter results.

With key based filtering + IC offset support, user is already capable of doing standalone mode of partition data sharing. By leveraging cloud operation technology such as Kubernetes, user could generate a set of pod configurations that defines which consumer subscribing to which topic partition range, and do easy hot swap when certain nodes go down. We shall discuss the consumer API changes to support this feature in the public interface section.

Thinking in even longer term, this work could be more generic for broker side filtering. Client could specify: 1. key ranges, 2. blacklist keys, 3. whitelist keys, 4. max/min record size, etc to further reduce the network cost with a penalty on throughput and latency.

## Rebalance support for concurrent assignment

### Multiple restrictions

To satisfy different administration requirements to turn off this feature at any time, we are placing several configurations on:

1. Broker level, to determine whether to allow IC request,
2. Topic level, to determine whether this topic is allowed to be cooperatively processed
3. Consumer level, to determine if this consumer opts to share processing with someone else

### Abstract Assignor

To determine the hash range for every consumer generation, in the group assignment phase the leader will evaluate the situation and decide whether to trigger the sharing mechanism. For the very first version, a very intuitive criteria is comparing the relative size of topic partitions vs number of consumers that are opting into the key sharing mode. For example if number of consumers m > number of partitions n, we would do the assignment based off partitions instead of consumers. If 5 consumers subscribing to 1 topic with 3 partitions, the final assignment would be:

```
N = Long.MAX
M1: tp1[0, N/2 - 1],
M2: tp2[0, N/2 - 1],
M3: tp3,
M4: tp1[N/2, N]
M5: tp2[N/2, N]
```

The new assignment comes from the fact that partitions are playing a reverse mapping to consumers. So in partition perspective, our assignment looks like a round robin assignment based off partitions:

```
tp1: M1, M4
tp2: M2, M5
tp3: M3
```

The assigned ranges could then be used by the consumer to make their fetch more fruitful. We don't plan to introduce a separate assignor strategy, but instead we would like to rollout this strategy as a plugin to the existing assignment assignors. As of today the ConsumerPartitionAssignor interface looks like below:

**ConsumerPartitionAssignor.java**

```
public ConsumerPartitionAssignor {
  public Map<String, List<TopicPartition>> assign(Map<String, Integer> partitionsPerTopic,
                                                  Map<String, Subscription> subscriptions)
}
```

Take round robin assignor as an example while merging the restriction requirements, an abstract assignor that could take partition split into consideration looks like below:

**KeyRangeBasedRoundRobinAssignor**

```
partitionsPerTopic := Map<String, Integer>
topicAllowKeyRangeFetch := Map<String, Boolean>
Subscriptions := Map<String, Subscription>
coordinatorAllowIC := Boolean
finalAssignment := Map<String, Map<String, Map<String, List<Tuple<Integer, KeyRange>>>>

groupSubscriptionByTopic
Step one: group subscription by topics
Group consumers based on their topic subscription. Inside the value, we put partition count and consumer.ids
inside
subscriptionByTopic := Map<String, Tuple<Integer, List<String>>>
for topic, value in subscriptionByTopic:
          numPartitions := value.1
        consumerList := value.2
    topicFinalAssignment := Map<String, List<Tuple<Integer, KeyRange>>
        consumersAllowKeyRange := Map<String, Subscriptions> // Set of Consumers who could allow KeyRange fetch
and commit
        consumersAllowKeyRange = getConsumersThatAllowKeyRange(topic) // must be a consumer who subscribes to
this topic
        if numPartitions < consumersAllowKeyRange.size() and coordinatorAllowIC and topicAllowKeyRangeFetch
[topic]:
            consumerPosition := 0
            partitionToConsumersMap := Map<Integer, List<String>>
            currentPartition := 0
            for c, _ in consumersAllowKeyRange:
                    partitionToConsumersMap[currentPartition].add(c);
                    currentPartition = (currentPartition + 1) % numPartitions

            for partition, assignedConsumers in partitionToConsumersMap:
                    keyRanges := List<KeyRange>
                     keyRanges = splitRangesByAvgSize(assignedConsumers.size()) // split key ranges by Range.
MAX and number of shares
            k = 0
                    for c in assignedConsumers:
                           topicFinalAssignment[c].add(new Tuple<>(partition, keyRanges[k])
                        k += 1
        else:
        do normal round robin assignment
    finalAssignment[topic] = topicFinalAssignment

return finalAssignment
```

Similarly for other assignment strategy such as range assignor, we always attempt to do a `reverse-assign` when consumers out-number any topic's total partitions.

This means for 5 consumers subscribing to 2 topics with 2 and 3 partitions each, the final assignment will look like (in partition perspective):

```
t1p0: c1, c2, c3
t1p1: c4, c5
-----
t2p0: c1, c2
t2p1: c3, c4
t2p2: c5
```

This step unblocks the potential to allow a dynamic consumer group scaling beyond partition level capping.

### Supporting Kafka Streams Rebalance

Kafka Streams have built a customized partition assignment semantics. For stateful DSL operations, there is a subtle requirement to make sure that the change in key distribution under the same partition will not encounter `key not found` error on the stream instance state. An example is like two stream instances A, B subscribing to one topic partition and do a per key aggregation. Record key *m* belongs to instance A for the first generation, however gets reassigned to instance B in the second generation. How does instance B know the previous states of record *m*? The answer is to replay the changelog topics. So in order to make Streams work with IC semantics, we could not split the stream changelogs because that would make it hard to maintain the key mapping. If multiple stream threads subscribing to the same partition, **the changelog shall not be split**. When the stream thread realizes a key range changes during rebalance, the restore consumer shall apply the new standalone mode key range assignment where the key based filtering rule could also be applied during restoration.

One more detail on stream is about task definition. The task will still be associated with partition number, but internally we mark it as a "Shared" task and put the key range into it. During rebalance, we will try to group shared tasks associating based on their partition and assign them onto the same stream instance, and the state store for each topic partition could allow concurrent access by shared tasks.

For example, if we have 2 partitions tp-0, tp-1 and 2 stream instances, with each stream instance configuring with 2 threads. In total, we have 4 consumers subscribing to them, and create four shared tasks:

```
task-0-[0, N/2 - 1],
task-0-[N/2, N],
task-1-[0, N/2 - 1],
task-1-[N/2 , N],
```

Normally for non-shared tasks, we would create two local state stores state-store-0, state-store-1. During rebalance, we will target to make the following assignments:

```
M1: task-0-[0, N/2 - 1], task-0-[N/2, N]
M2: task-1-[0, N/2 - 1], task-1-[N/2, N]
```

The benefit of arranging such assignment is to reduce state store creation. If unfortunately we have an assignment such like:

```
M1: task-0-[0, N/2 - 1], task-0-[N/2, N]
M2: task-1-[0, N/2 - 1], task-1-[N/2, N]
```

We need to create 2 state stores on each machine, which in total is 4.

Another issue will be on the interactive query, as a specific record key may be assigned to any shared task. So StreamsMetadata needs

## Look into the future: Transactional Support

Eventually IC semantic has to be compatible with transaction support. Despite the ongoing discussion of any change to the transaction semantics, we are providing a rough plan on integrating with current transaction model in this KIP as well. The current transaction model uses transaction markers to specify whether all the records before it are ready to be revealed in downstream. So inherently, this is a design that obeys partition level ordering. To avoid over-complicating the transaction semantic, we will introduce a new delayed queue or purgatory for transactional IC semantic, where the commit shall be blocked until it could hit the stable offset. Take Kafka Streams as an example, if we have a chunk of data [40-49] with stable offset at 39, with two stream threads A and B are turning on EOS at the same time in the sharing mode:

1. Stream thread A did a fetch to get [40, 44]
2. Stream thread B did a fetch to get [45, 49]
3. Stream thread B finished processing and issue a transactional commit of [45, 49]
4. As the offset range [40, 44] is held by someone at the moment, transactional commit request will be put in a purgatory to wait for stable offset advance
5. Stream thread A finished processing and issue a transactional commit of [40, 44]
6. We first advance the stable offset and reply stream thread A with successful commit
7. Then we search purgatory to reply stream thread B and allow it to proceed

This high level workflow is blocking in nature, due to the nature of Kafka transaction model. Another approach is to add the current transaction model capability to track key level ordering which is also very tricky. The design is still open to discussion and may change as the Kafka transaction model evolves.

## Look into the future: Cooperative Fetch

Take a look back at the stateless operations like filter or map, there is no necessity to honor the consumer  key mappings during the processing. From KIP-283, we already know it's very costly and inefficient to copy data around. Based on client's need, broker could do a random assignment when receiving fetch requests without key level granularity. It will keep track of who has been fetched to which position. This means we don't need to load any data into the main memory and the total of IC offsets will be significantly dropped.

For example if we have a chunk of data [40-49], where offsets 41, 43, 45, 47, 49 belong to key A and 40, 42, 44, 46, 48 belong to key B, consumers owning key A will commit 5 IC offsets, same as key B owner. If we don't maintain that mapping, for consumer fetching we will first return data range [40-44] and advance the in-memory marker, and then reply [45, 49] even potentially this consumer is trying to fetch starting from offset 40, with the hope that someone else will take care of [40-44] eventually.

Cooperative fetch is more like an optimization upon our stateless use case scenario, instead of a new feature. To make it right, we have to encode more metadata within the consumer fetch request such as consumer generation and consumer id, wisely refusing advancing the marker unless necessary when 1. the consumer is zombie, 2. the same consumer is retrying the fetch 3. session timeout. And there should also be a max wait time for commit gaps so that we don't lose the chance to process the data when consumer whoever did the fetch encounters hard failures already. It could be set the same as consumer configured session timeout.

The design is still open to discussion as the trade-off between improvements over complexity is still not clear.

# Public Interfaces

The offset commit protocol will be changed to allow IC semantic. The committed offset will include a list of offset ranges indicating the acknowledged messages, and in the response a LastStableOffset will be added to let consumer purge local IC offsets.

```
OffsetCommitRequest => GroupId Offsets GenerationId MemberId GroupInstanceId
  GroupId             => String
  Offsets             => Map<TopicPartition, CommittedOffset>
  GenerationId        => int32, default -1
  MemberId            => nullable String
  GroupInstanceId     => nullable String

CommittedOffset => offset, metadata, leaderEpoch, offsetRanges
  Offset              => int64, default -1
  Metadata            => nullable String
  LeaderEpoch         => Optional<int32>
  OffsetRanges        => List<Tuple<int64, int64>> // NEW

OffsetCommitResponse => ThrottleTimeMs Topics
  Topics         => List<OffsetCommitResponseTopic>
  ErrorCode      => int16

OffsetCommitResponseTopic => Name Partitions
  Name           => int32
  Partitions     => List<OffsetCommitResponsePartition>

OffsetCommitResponsePartition => PartitionIndex ErrorCode LastStableOffset
  PartitionIndex => int32
  ErrorCode => int16
  LastStableOffset => int64 // NEW
```

When the offset range is not empty, broker will only handle offset ranges and ignore the plain offset field. Optionally broker will check if the original offset field is set to -1 to make sure there is no data corruption in the request. Trivially, transaction offset commit will also include the offset range.

The offset fetch request will also be augmented to incorporate the key hash range. Using a list instead of one hash range allows future extensibility.

```
FetchRequest => MaxWaitTime ReplicaId MinBytes IsolationLevel FetchSessionId FetchSessionEpoch [Topics]
[RemovedTopics] KeyRanges GenerationId MemberId SessionTimeout
  MaxWaitTime => int32
  ReplicaId => int32
  MinBytes => int32
  IsolationLevel => int8
  FetchSessionId => int32
  FetchSessionEpoch => int32
  Topics => TopicName Partitions
    TopicName => String
    Partitions => [Partition FetchOffset StartOffset LeaderEpoch MaxBytes]
        Partition => int32
        CurrentLeaderEpoch => int32
        FetchOffset => int64
        StartOffset => int64
        MaxBytes => int32
  RemovedTopics => RemovedTopicName [RemovedPartition]
    RemovedTopicName => String
    RemovedPartition => int32
  KeyRanges => List<Tuple<int64, int64>> // NEW
```

We shall also put offset ranges as part of OffsetFetchResponse:

```
OffsetFetchResponse => ThrottleTimeMs Topics ErrorCode
  ThrottleTimeMs => int32
  Topics         => List<OffsetFetchResponseTopic>
  ErrorCode      => int16

OffsetFetchResponseTopic => Name Partitions
  Name           => int32
  Partitions     => List<OffsetFetchResponsePartition>

OffsetFetchResponsePartition => PartitionIndex CommittedOffset CommittedLeaderEpoch Metadata ErrorCode
  PartitionIndex => int32
  CommittedOffset => int64
  CommittedLeaderEpoch => int32
  Metadata => string
  ErrorCode => int16
  CommittedRangeOffsets => List<Tuple<int64, int64>> // NEW
```

To leverage the key based filtering in a standalone mode, we also define a new consumer API in standalone mode:

**Consumer.java**

```
public Consumer {
  ...

 /**
  * Manually assign a list of partitions with specific key ranges to this consumer. If a partition maps to an
empty list,
  * that means a full ownership of the partition.
  */
  void assign(Map<TopicPartition, List<Tuple<int64, int64>>> partitionWithKeyRanges);
}
```

To recognize whether a consumer is allowed to subscribe as key-share, we shall put a new boolean field in Subscription to indicate that:

**Subscription.java**

```
public Subscription {
  ...
  public boolean allowKeyShare();
}
```

To leverage the key based filtering in a group subscription mode, we will also add callbacks the ConsumerRebalanceListener to populate key range assignments if necessary:

**Consumer.java**

```
public ConsumerRebalanceListener {

        @Deprecated
        void onPartitionsRevoked(Collection<TopicPartition> partitions);

         @Deprecated
    void onPartitionsAssigned(Collection<TopicPartition> partitions);

    @Deprecated
    default void onPartitionsLost(Collection<TopicPartition> partitions) {
        onPartitionsRevoked(partitions);
    }

   /**
    * Note that if a partition maps to an empty list, that means a full ownership of the partitions.
    */
        void onPartitionWithKeyRangesAssigned(Map<TopicPartition, List<Tuple<int64, int64>>
partitionWithKeyRanges); // NEW

        void onPartitionWithKeyRangesRevoked(Map<TopicPartition, List<Tuple<int64, int64>>
partitionWithKeyRanges); // NEW

        void onPartitionWithKeyRangesLost(Map<TopicPartition, List<Tuple<int64, int64>>
partitionWithKeyRanges); // NEW
}
```

For better visibility, we shall include the key range information in StreamsMetadata:

**StreamsMetadata.java**

```
public class OffsetAndMetadata {

  public HostInfo hostInfo();

  public Set<String> stateStoreNames();

  @Deprecated
  public Set<TopicPartition> topicPartitions();

  public Map<TopicPartition, List<Tuple<Long, Long>>> topicPartitionsWithKeyRange();

  public String host();

  public int port();
}
```

For better visibility, we shall include the range offsets in the ListConsumerGroupOffsetsResult:

**ListConsumerGroupOffsetsResult.java**

```
public class ListConsumerGroupOffsetsResult {
        ...
    final KafkaFuture<Map<TopicPartition, OffsetAndMetadata>> future;
        ...
}

public class OffsetAndMetadata {

    public long offset();

        public List<Tuple<Long, Long>> rangeOffsets(); // NEW

    public String metadata();

    public Optional<Integer> leaderEpoch();
}
```

## New Exceptions

We are going to define a series of exceptions which are thrown as 1. access control was hit, 2. inconsistency

**Errors.java**

```
INDIVIDUAL_COMMIT_NOT_ALLOWED(88, "The broker hosting coordinator doesn't allow individual commit",
IndividualCommitNotAllowedException::new);
TOPIC_RANGE_FETCH_NOT_ALLOWED(89, "The topic is not allowed to perform range fetch with",
TopicRangeFetchNotAllowedException::new);
CONSUMER_RANGE_FETCH_NOT_ACCEPTED(90, "The consumer is not allowed to do the range fetch",
ConsumerRangeFetchNotAccepted::new);
INDIVIDUAL_COMMIT_TOO_OLD(91, "The individual commit failed due to attempting to commit some entry behind LSO");
MAX_INDIVIDUAL_COMMIT_REACHED(92, "The maximum number of individual commits coordinator could hold reaches the
cap");
```

## New Configurations

### Broker configs

| | |
|---|---|
| accept.individual.commit | Determine whether the group coordinator allows individual commit.<br><br>Default: true |
| *max.bytes.key.range.cache* | The maximum memory allowed to hold partitioned records in-memory for next batch serving. Note that this is not the memory limit for handling one batch of data range check.<br><br>Default: 209715200 bytes |
| max.bytes.fetcher.filtering | The maximum memory allowed for each replica fetcher to do general broker side filtering, including the key hash processing.<br><br>Default: 104857600 bytes |
| individual.commit.log.segment.bytes | The segment size for the individual commit topic.<br><br>Default: 10485760 bytes |

### Topic config

| | |
|---|---|
| accept.range.fetch | Determine whether the topic is allowed to perform key range based fetch. The reason to set it false could be more like a user's call such that any subscribing application must obey the partition level ordering.<br><br>Default: false |

| | |
|---|---|
| *reserve.message.value.range.fetch* | Determine whether when caching last fetch request result, we should also keep the message value inside main memory for optimization.<br><br>Default: false |
| individual.commit.log.segment.bytes | The segment size for the individual commit topic.<br><br>Default: 10485760 bytes |

### Consumer configs

| | |
|---|---|
| allow.individual.commit | Determine whether this consumer will participate in a shared consumption mode with other consumers.<br><br>Default: true |
| max.num.ranges.to.commit | Commit the progress if accumulated offset ranges are beyond this number.<br><br>Default: 10,000 |

# Related Work

Pulsar has officially supported key share feature in 2.4, which suggests multiple consumers could share the same partition data.

The blockers for us to implement a similar feature are:

1. Our consumer model is pull based, which incurs random read if we let consumers ask for specific keyed records. Sequential read is the key performance sugar for Kafka, as otherwise we could not bypass memory copy of the data.
2. Our broker doesn't know anything about data distribution, as all the metadata is encoded and looks opaque to them. In reality, we could not afford letting consumers fetch with raw keys.
3. Consumer coordinator is at a centralized location, however we need to distribute keys in different partitions. For Pulsar their offset data is co-located with actual partitions. The burden for broadcasting the state change in Kafka would be pretty hard and very error-prone.

# Compatibility, Deprecation, and Migration Plan

As transactional support and cooperative fetch need more design discussion, we will delay putting on the upgrade path now.

To upgrade an existing consumer use case to allow individual commit + key based filtering + rebalance support, the user needs to:

1. Upgrade broker to latest, by default individual commit is turned on
2. Allow the input topic to be shared by turning on the range fetch config
3. Upgrade consumer client to allow range sharing
4. Rolling bounce the group, and the consumer group should be able to share consumption of the topic

# Failure Recovery and Test Plan

We discuss the failure scenario alongside with the test plan to better justify the value.

1. Individual Commit failure. The failure could result from:
    a. Broker restriction. If this is the case, the client will immediately fail with restriction exception.
    b. IC log not found or corrupted. As the _consumed_offsets topic need to co-partition with IC log, we need to load the replica from ISR to the current coordinator, and force a leader election to nominate the new log as the leader.
    c. Commit too old: the IC offset is smaller than the current LSO. In this case, reply with the current LSO and client will adjust its position to catch up with latest.
    d. In at least once, commit in duplicate is not a serious flag. Brokers should not response a separate exception but will do the warning log for user to debug later.
2. Range Fetch failure. The failure could result from:
    a. Topic restriction. The client will immediately crash with restriction exception.
    b. Other common fetch request errors like topic not found, shall be handled in the same way
3. Rebalance support failure. The failure could result from:
    a. Consumer restriction: Consumer will reject an assignment with range fetch if it is not allowed to.
    b. Make sure the no assignment overlap is detected in system tests.
    c. Other common rebalance failure.

# Rejected Alternatives

There are a couple of alternatives to this proposal here.

1. KIP-253 proposed physical partition expansion, which is a fairly complex implementation and could be hard to reason about correctness.

2. Some discussion around making Kafka Streams as a multi-threading model where consumers are completely decoupled from processing thread. This means we have to conquer the concurrent processing challenge and there could be more inherent work to redesign state store semantics too.