KIP-535: Allow state stores to serve stale reads during rebalance

JIRA: ~~KAFKA-6144~~ **- Allow serving interactive queries from in-sync Standbys** RESOLVED

Discussion: here

Motivation

Currently Streams interactive queries (IQ) fail during the time period where there is a rebalance in progress.

Consider the following scenario in a three-node Streams cluster with node A, node S and node R, executing a stateful sub-topology/topic group with 1 partition and `*num.standby.replicas=1*`

- **t0**: A is the active instance owning the partition, B is the standby that keeps replicating the A's state into its local disk, R just routes streams IQs to active instance using StreamsMetadata
- **t1**: IQs pick node R as router, R forwards query to A, A responds back to R which reverse forwards back the results.
- **t2:** Active A instance is killed and rebalance begins. IQs start failing to A
- **t3**: Rebalance assignment happens and standby B is now promoted as active instance. IQs continue to fail
- **t4**: B fully catches up to changelog tail and rewinds offsets to A's last commit position, IQs continue to fail
- **t5**: IQs to R, get routed to B, which is now ready to serve results. IQs start succeeding again

Depending on Kafka consumer group session/heartbeat timeouts, step t2,t3 can last a few seconds (~10 seconds based on defaults values). Depending on how laggy the standby B was prior to A being killed, t4 can take few seconds-minutes. While this behavior favors consistency over availability at all times, the long unavailability window might be undesirable for certain classes of applications (e.g simple caches or dashboards). These applications might also be able tolerate eventual consistency.

This KIP aims to also expose additional metadata, that would help performing highly available interactive queries against Streams state. Specifically

- Exposes information about the current lag of the state stores, at each Kafka Streams instance locally.
- AssignmentInfo also provides host information about standbys for all topic partitions globally

Thus an application relying on IQs can

- Route queries to standby replicas for better availability, if active is currently down (window t2-t4 discussed above)
- Implement a control plane to exchange lag information (e.g piggybacked on requests or heartbeats) to choose the best standby to route to
- Or even choose to fail the query if all the replicas are lagging behind very much (e.g node S was down for a long time before time t0, and was still catching up when node A went down.)

It's worth noting that with this change, Streams API can be used to build stateful application that offer different tradeoffs for consistency and availability.

Public Interfaces

A single StreamsMetadata object represents the state of a Kafka Streams instance. It already has methods to obtain host/port of the instance, stores/topic partitions it hosts as an active. We will extend this class to also add standby stores/topic partitions, which will be filled in based on the AssignmentInfo object obtained during rebalance.

StreamsMetadata.java

```
package org.apache.kafka.streams.state;

public class StreamsMetadata {
...
    private final Set<TopicPartition> standbyTopicPartitions;
    private final Set<String> standbyStateStoreNames;

    public Set<TopicPartition> standbyTopicPartitions() {
        return standbyTopicPartitions;
    }

    public Set<String> standbyStateStoreNames() {
        return standbyStateStoreNames;
    }
...
}
```

We will introduce a new class **KeyQueryMetadata,** which contains all the routing information corresponding to a queried key. Notably it contains the partition value to which the key belongs, so that it can used to correlate with the lag information

KeyQueryMetadata

```
package org.apache.kafka.streams;

public class KeyQueryMetadata {
    // Active streams instance for key
    private HostInfo activeHost;
    // Streams instances that host the key as standbys
    private List<HostInfo> standbyHosts;
    // Store partition corresponding to the key.
    private int partition;

    // Standard getters/setters will be added.
    ...

}
```

We will also introduce a **LagInfo**, class that encapsulates the lag information

```
package org.apache.kafka.streams;

public class LagInfo {

    private final long currentOffsetPosition;

    private final long endOffsetPosition;

    private final long offsetLag;
    ...
}
```

Changes to **org.apache.kafka.streams.KafkaStreams**, to introduce two methods for querying the metadata specific to a given key/store and one method for fetching offset lag information for all stores hosts locally on that streams instance. In addition, we will add deprecated annotation to the existing `metadataFoKey()` methods and their javadocs will clearly reflect that it returns active replica information. Also, we introduce a new method to get stores for querying with IQ that includes a flag indicating whether to allow queries over standbys and restoring stores, or only on running active ones. The behavior of the pre-existing "store(name, type)" method is unchanged: it only allows querying running active stores.

```
package org.apache.kafka.streams;
```

```
public class KafkaStreams {

...

  /**
   * Returns {@link KeyQueryMetadata} containing all metadata about hosting the given key for the given store.
   */
  public <K> KeyQueryMetadata queryMetadataForKey(final String storeName,
                                                  final K key,
                                                  final Serializer<K> keySerializer) {}

  /**
   * Returns {@link KeyQueryMetadata} containing all metadata about hosting the given key for the given store,
using the
   * the supplied partitioner
   */
  public <K> KeyQueryMetadata queryMetadataForKey(final String storeName,
                                                  final K key,
                                                  final StreamPartitioner<? super K, ?> partitioner) {}
...

 /**
   * Deprecating this function to get more users accepted to queryMetadataForKey.
   */
 public <K> StreamsMetadata metadataForKey(final String storeName,
                                           final K key,
                                           final Serializer<K> keySerializer) {}

 /**
   * Deprecating this function to get more users accepted to queryMetadataForKey.
   */
 public <K> StreamsMetadata metadataForKey(final String storeName,
                                           final K key,
                                           final StreamPartitioner<? super K, ?> partitioner) {}

  /**
   * Returns mapping from store name to another map of partition to offset lag info, for all stores local to
this Streams instance. It includes both active and standby store partitions, with active partitions always
reporting 0 lag in RUNNING state and actual lag from changelog end offset when RESTORING.
   */
  public Map<String, Map<Integer, LagInfo>> allLocalStorePartitionLags() {}

  /**
   * Get a facade wrapping the local {@link StateStore} instances with the provided {@code storeName} if the
Store's
   * type is accepted by the provided {@link QueryableStoreType#accepts(StateStore) queryableStoreType}.
   * The returned object can be used to query the {@link StateStore} instances.
   *
   * @param storeName           name of the store to find
   * @param queryableStoreType  accept only stores that are accepted by {@link QueryableStoreType#accepts
(StateStore)}
   * @param includeStaleStores      If false, only permit queries on the active replica for a partition, and
only if the
   *                            task for that partition is running. I.e., the state store is not a standby
replica,
   *                            and it is not restoring from the changelog.
   *                            If true, allow queries on standbys and restoring replicas in addition to active
ones.
   * @param <T>                 return type
   * @return A facade wrapping the local {@link StateStore} instances
   * @throws InvalidStateStoreException if Kafka Streams is (re-)initializing or a store with {@code storeName}
and
   * {@code queryableStoreType} doesn't exist
   */
  public <T> T store(final String storeName,
                     final QueryableStoreType<T> queryableStoreType,
                     final boolean includeStaleStores) {}

...
```

```
}
```

Below is some pseudo code to showing sample usage of these two APIs by a real Streams application.

```
// Global map containing latest lag information across hosts. This is collected by the Streams application
instances outside of Streams, just relying on the local lag APIs in each.
// KafkaStreams#allMetadata() is used by each application instance to discover other application instances to
exchange this lag information.
// Exact communication mechanism is left to the application. Some options : a gossip protocol, maintain another
single partition Kafka topic to implement lag broadcast
// Accumulated lag information from other streams instances.
final Map<HostInfo, Map<String, Map<Integer, Long>>> globalLagInformation;

// Key which needs to be routed
final K key;

// Store to be queried
final String storeName;

// Fetch the metadata related to the key
KeyQueryMetadata queryMetadata = queryMetadataForKey(store, key, serializer);

// Acceptable lag for the query
final long acceptableOffsetLag = 10000;


if (isAlive(queryMetadata.getActiveHost()) {
  // always route to active if alive
  query(store, key, queryMetadata.getActiveHost());
} else {
  // filter out all the standbys with unacceptable lag than acceptable lag & obtain a list of standbys that are
in-sync
  List<HostInfo> inSyncStandbys = queryMetadata.getStandbyHosts().stream()
      // get the lag at each standby host for the key's store partition
      .map(standbyHostInfo -> new Pair(standbyHostInfo, globalLagInforation.get(standbyHostInfo).get(storeName).
get(queryMetadata.partition())).offsetLag())
      // Sort by offset lag, i.e smallest lag first
      .sorted(Comaparator.comparing(Pair::getRight())
      .filter(standbyHostLagPair -> standbyHostLagPair.getRight() < acceptableOffsetLag)
      .map(standbyHostLagPair -> standbyHostLagPair.getLeft())
      .collect(Collectors.toList());

  // Query standbys most in-sync to least in-sync
  for (HostInfo standbyHost : inSyncStandbys) {
    try {
      query(store, key, standbyHost);
    } catch (Exception e) {
      System.err.println("Querying standby failed");
    }
  }
}
```

Proposed Changes

In the current code, t0 and t1 serve queries from Active(Running) partition. For case t2, we are planning to return List<StreamsMetadata> such that it returns <StreamsMetadata(A), StreamsMetadata(B)> so that if IQ fails on A, the standby on B can serve the data by enabling serving from replicas. This still does not solve case t3 and t4 since B has been promoted to active but it is in Restoring state to catchup till A's last committed position as we don't serve from Restoring state in active and new replica on R is building itself from scratch. Both these cases can be solved if we start serving from Restoring state of active as well, since it is almost equivalent to previous active.

The new plan is to enhance the serving query capabilities to include standbys as well and have minimal changes in the core streams code to expose local lag information to the streams application, such that query routing can happen in the following way

1. queries get routed to any random streams instance in the cluster ("router" here on)
2. router then uses Streams metadata to pick active/standby instances for that key's store/partition
3. router instance also maintains global lag information for all stores and all their partitions, by a gossip/broadcast/heartbeat protocol (done outside of Streams framework), but using KafkaStreams#allMetadata() for streams instance discovery.

4. router then uses information in 2 & 3 to determine which instance to send the query to : always picks active instance if alive or the most in-sync live standby otherwise.

With this KIP, the onus is left on a user to decide how much lag the queries are willing to tolerate, since there is the capability of serving from a restoring active as well as running standby task.

Implementation will incur the following changes

- **AssignmentInfo** changes to include *Map<HostInfo, Set<TopicPartition>> standbyPartitionsByHost*; so that each machine knows which machine holds which *standby* partitions along with the active partitions(which they currently do). Consequently AssignmentInfo version will be bumped up to **VERSION 6**
- Changing signature of *setPartitionsByHostState(partitionsByHostState, standbyPartitionsByHost)* and to *onChange()* and further *rebuildMetadata()* to add *Set<TopicPartition> standbyTopicPartitions* in **StreamsMetadata**. This will add standby partitions in the metadata.
- Addition of StreamsMetadataState**::**getStandbyMetadataListForKey() to returns a list of StreamsMetadata which contains all the standbys available in the system for the partition. We would have access to allMetadata containing activePartitions as well as standby partitions in the StreamsMetadataState.java with the above changes.
- Overloading KafkaStreams#store() to add a new boolean parameter includeStaleStores which will be false by default if current KafkaStreams#store() is called.
- Renaming partitionsForHost to activePartitionsForHost in StreamsMetadataState.java and partitionsByHostState to activePartitionsByHostState in StreamsPartitionAssignor.java
- We also might need to make changes to make the offset lag information tracked in-memory accessible for the lag APIs.

Compatibility, Deprecation, and Migration Plan

- This KIP affects StreamsMetadata and AssignmentInfo. Our changes to StreamsMetadata will be compatible with existing streams apps. AssignmentInfo changes will add a *standbyPartitionsByHost map,* we would need to upgrade AssignmentInfo version and we expect the existing version probing mechanism to handle the migration path.
- Two methods(KafkaStreams#metadataForKey()) are deprecated and users can migrate their code to use the two new methods (KafkaStreams#queryMetadataForKey()) instead.

Rejected Alternatives

- Adding a StreamsConfig to have a universal *enableStandbyServing* flag for the application. This would restrict the functionality as there could be multiple stores in an application and we need to have the flexibility to extend different consistency guarantees in such cases, which would be restricted by having a StreamsConfig.
- Making the Streams APIs lag aware e.g only return standbys within a certain time/offset lag. It did not add a lot of value to push this into Streams. Instead, the KIP keeps Streams agnostic of what acceptable values for lag is and provides the application/user flexibility to choose.
- Propagating lag information using the Rebalance protocol. Although it seemed like logical thing to do, with KIP-441 in mind, we decided against it due to various reasons. Foremost, the lag information is quickly outdated
- We decided against making multiple calls to StreamsMetadataState to fetch active tasks, standby tasks and the partition to which key belongs(as it is required to fetch lag) and have a new public class KeyQueryMetadata as it makes the code leaner and returns in a single fetch.
- Since it's sufficient to just support offsetLag initially without needing time-based lag information right away and adding time-based lag needs broker changes to be implemented efficiently, we decided against implementing time-based offsets for now.