KIP-796: Interactive Query v2

Status

Current state: *Accepted*

Discussion thread: https://lists.apache.org/thread/vkn4njjhb0s6w582798boffn67z9spdb

Vote thread: https://lists.apache.org/thread/jmfqp5rycbcztn1ksvn4b7c9p34xgftk

JIRA: *Unable to render Jira issues macro, execution error.*

POC PR: https://github.com/apache/kafka/pull/11406

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

KIP Tracking

To control the scope of this long-term project, we chose to defer much of the details to later KIPs. Here's a roundup of the related work:

- KIP-805: Add range and scan query support in IQ v2
- KIP-806: Add session and window query over kv-store in IQv2

Motivation

Kafka Streams supports an interesting and innovative API for "peeking" into the internal state of running stateful stream processors from outside of the application, called Interactive Query (IQ). This functionality has proven invaluable to users over the years for everything from debugging running applications to serving low latency queries straight from the Streams runtime.

However, the actual interfaces for IQ were designed in the very early days of Kafka Streams, before the project had gained significant adoption, and in the absence of much precedent for this kind of API in peer projects. With the benefit of hindsight, we can observe several problems with the original design that we hope to address in a revised framework that will serve Streams users well for many years to come.

Problems

1. IQ serves queries by presenting callers with a composite store interface, which encapsulates the fact that stores will in general be partitioned, and that a given instance may only host a subset of those partitions (if any).
    a. The cost of constructing this store interface is non-trivial, though many real-world use cases will only use the store to run one query and then discard it.
    b. The creation of the store is subject to any number of error conditions, so callers need to handle exceptions on calling `KafkaStreams.store()`.
    c. Once you have an IQ store reference, it is still subject to any number of transient and permanent conditions (such as rebalances having moved partitions off the local instance, Streams changing state, etc.), so callers also need to handle exceptions while running queries on their store, and be prepared to rebuild the store if it becomes invalid.
    d. Users desiring to query custom state stores need to produce a fairly detailed implementation of QueryableStoreType that details how to compose results from multiple partitions into one.
        i. In particular, if you want to plug a store with special query capabilities in to the Streams DSL (for example as the materialization of a KTable), the store must extend the appropriate store interface, for example `KeyValueStore<Bytes,byte[]>`. However, when it comes to exposing it in IQ, you will have to implement `QueryableStoreType<YourStore>`, and it requires you to produce an instance of YourStore that delegates to a list of `ReadOnlyKeyValueStore<Bytes,byte[]>` returned by `storeProvider.stores`. However, the catch is that those delegate stores are *not* instances of your store type! They will be instances of the internal class `MeteredTimestampedKeyValueStore`, which wraps other internal store types, which after a few more layers of wrapping contain YourStore at the bottom. Your only choice will be to create a *completely separate implementation* of the YourStore interface that delegates all the ReadOnlyKeyValueStore methods to those wrapper layers, and then for the methods that are special to YourStore, you'll have to repeatedly cast and type-check the wrappers until you manage to get all the way down to the actual YourStore that can serve the query.
        ii. In other words, it's so complicated that it might as well be impossible. Which makes it not surprising that no one has actually tried. I suspect that if we made it easier to extend the storage framework, we would see a bunch of new use cases pop up building on IQ in the future.
2. IQ composes all locally present partitions into a unified response. For example, for queries that return an iterator, it builds a composite iterator that collates all locally available partitions' iterators into one.
    a. While this is useful for trivial use cases, it destroys useful information about the response:
        i. Callers don't know which partitions were included in the response.
        ii. After iterating for some time, callers can't tell when individual partitions' iterations are complete. This is important if we experience a failure: partitions that are already complete don't need to repeat the query.
    b. In practice, partitions' responses can succeed or fail independently, but the composite response couples all responses to any individual partition's failure.
3. Because IQ responses are simply the result type of whatever store method you invoke, it is not possible to attach extra metadata that is useful in the IQ context, specifically.
    a. Eg. We might want to add detailed trace information about which store layers or segments participated in the query, including execution time, cache hit/miss, etc. This kind of feature would be particularly useful when debugging performance, or when IQ is backing a service that uses distributed tracing, etc.
    b. Eg. We might want to add information about the precise state of the view we served: what was the input offset we last wrote into the store? What is the "current stream time" of the view we served? What was the state of the StreamTask when we served the query? Etc.
    c. These are just examples from various conversations about potentially useful IQ response metadata. The point is to illustrate the fact that we are missing opportunities by restricting the IQ response to be the simple value returned by the store methods that serve the query.
4. Supporting new types of queries to the "standard" store interfaces is onerous, since it requires adding new methods to the store interfaces, which need to be overridden in dozens of utility implementations throughout the Streams codebase.
    a. Example: KIP-617: Allow Kafka Streams State Stores to be iterated backwards . This change involved four PRs (https://github.com/apache/kafka/pull/9137, https://github.com/apache/kafka/pull/9138, https://github.com/apache/kafka/pull/9139/files, https://github.com/apache/kafka/pull/9239), totaling **108 files** and **6,000+ lines of code** changed.
    b. Another example: KIP-614: Add Prefix Scan support for State Stores (which only edits the KeyValueStore). This change took two PRs (https://github.com/apache/kafka/pull/9508 and https://github.com/apache/kafka/pull/10052), totaling **19 files and 600+ lines of code** changed.
5. IQ forces all queries to compose the contents of the Record Cache (the write buffer for Processors) with the underlying bytes stores.
    a. Despite its name, the Record Cache is a write buffer, not a traditional read cache. Perhaps not surprisingly, its performance is not very good for arbitrary queries, since its primary purpose is to ensure that Processors always read their own writes while delaying operations like sending writes to the changelog and the underlying stores.
    b. We could invest in optimizing the Record Cache for queries, but we would probably find that the better approach is to separate the read and write workloads.
    c. Regardless of potential future optimizations in the Record Cache, merging the buffered writes with the underlying stores (as in the case of a table scan) will always require extra work, and it would be good to have an option to skip the record cache for IQ users.
    d. In contrast to Processors, IQ interactions can never write, so do not need any concept of "read your writes".

Unintended Consequences

In a nutshell, all those issues result in a system that isn't ideal from anyone's perspective:

1. People using IQ:
    a. Have to call two methods to do one query (KafkaStreams#store and then the actual query method) and have to deal with exceptions from both of those methods
    b. Lose important information about which partitions were included in the response, and when individual partitions complete during the process of consuming results
    c. Get worse performance than necessary due to the overhead of building the intermediate store abstraction
2. People adding new stores:
    a. Have to implement prohibitively complex logic to expose their store's capabilities through IQ (see Problem 1d).

3. People contributing to existing store interfaces:
    a. Have to jump through a bunch of hoops to add a new method to the store interfaces.
    b. Have no way to know they did everything right unless they test every combination of store configurations with both the PAPI and IQ
4. People maintaining Streams:
    a. Have a significant burden reviewing KIPs and PRs because there are so many complexities involved in properly changing store interfaces
    b. Have to deal with a long-tail of bug reports that trickle in when some contribution inevitably overlooks some "minor" point like verifying a new method works via IQ or is properly handled in the cache, etc.

In conclusion, and to clarify: IQ itself has been extremely successful and valuable to many people. I only harp on the above points to demonstrate a number of flaws that I think we can improve on to make it even more valuable and successful in the future.

Goals

To address the above pain points while preserving the positive aspects of IQ, I'd propose the following goals for this KIP:

1. We should continue to offer a mechanism to peek into the internal state of Kafka Streams's stateful operations.
2. We should recognize that querying state via IQ is a different use case from using a store in a Processor, and a different interface should therefore be on the table.
3. Simple use cases should continue to be easy.
    a. Particularly, it should continue to be easy to just "query the store" and not worry about individual partitions, etc.
4. More complex use cases should be possible and not too hard.
    a. Particularly, it should be possible to pick and choose partitions to query, and to handle independent partitions' responses independently.
    b. It should also be possible to define new store types and queries with a manageable level of complexity.
    c. It should be possible to tune queries for maximum performance.
5. Contributing to and maintaining the code base should be relatively straightforward.

Proposed Changes

This KIP proposes a new framework for IQ, which we will refer to as "IQv2". It is outside the scope of this KIP to propose new mechanisms for every query that IQ supports today, as part of the purpose of this design is to be easily extensible (and we want to keep the KIP focused on the framework). However, we do propose to add a few specific queries, to flesh out the proposal and to give some purpose to the initial batch of implementation work.

The basic design of IQv2 is to add a mechanism for KafkaStreams (and TopologyTestDriver, which we'll omit for brevity in the discussion) to execute a "query" on the caller's behalf (as opposed to constructing a store for the caller to query).

- This addresses Problem 1 (and Unintended Consequence 1a) because each time a user wants to query the store, they just call one method and have no store lifecycle to maintain.

The query itself will be (almost) completely opaque to KafkaStreams, and will effectively be a protocol between the IQv2 caller and the underlying state store.

- This is the key to addressing Problem 4, and it resolves Unintended Consequence 2 (because new stores don't need to do anything except handle queries to be integrated with IQ) and Unintended Consequence 3 (because the scope of a new capability is only limited to adding a new Query type and adding handlers for it in the desired store). It also resolves Unintended Consequence 4 for the same reason as 3, since the scope of adding a new query is so much smaller.
- This design also addresses Problem 5 because the Caching state store layers will have the opportunity to handle known queries before passing them down to lower store layers. So, if desired, we can define a well-known KeyQuery that has a flag controlling whether the cache should handle it or not, while a custom query type would naturally bypass the cache, since the cache doesn't have knowledge of the query type.

IQv2 will include "container" request and response objects, enabling refinements and controls to be added onto queries and also enabling additional metadata to be accompany results.

- This addresses Problem 3 because we can attach all the extra information we need "around" the core query and result.
- It also creates a mechanism for future extensions to IQ

The response object will not attempt to compose individual partitions' responses. Instead, the response object will provide an API to get the responses for each partition. Additionally, we will provide some utilities to compose partition responses.

- This addresses Problem 2 because responses aren't required to be composable
- It also creates room for partitions to report successful or failure responses independently, which addresses Unintended Consequence 1b
- It also addresses Unintended Consequence 1c because we don't do extra work to combine partions' results unless we need to.

Special interface stability protocol

All of the new methods and classes will be marked as `@Evolving` ("Compatibility may be broken at minor release") to allow for more rapid interaction as we flesh out the first generation of queries. For such changes, we will simply notify the KIP discussion thread and update this document. Our goal is to stabilize the interface as quickly as possible, ideally in the next major release.

Public Interfaces

This KIP will add several new methods, classes, and interfaces.

Documentation

In addition to the Java interfaces below, we want to call out that we should add a new section to the documentation detailing how to use the IQv2 extension points (adding new store types and queries), which would also cover how to contribute new Query implementations to Apache Kafka so that users will actually be able to realize the extensibility benefits of this proposal.

KafkaStreams modifications

IQv2 will continue to present its primary API via the KafkaStreams interface. The `query` method itself is this API.

Note that exceptional conditions that prevent the query from being executed at all will be thrown as exceptions. These include Streams not having been started, and invalid requests like querying a store that doesn't exist or specifying a partition to query that doesn't exist.

There is a second class of failure, which today also is thrown as an exception, that will instead be reported on a per-partition basis as a QueryResult for which `isFailure` is true, and which contains the reason for the failure along with other information. This would include conditions for which only part of the query may fail, such as a store being offline, a partition not being locally assigned, a store not being up to the desired bound, etc.

KafkaStreams

```
public class KafkaStreams implements AutoCloseable {
  ...

  /**
    * Run an interactive query against a state store.
    * <p>
    * This method allows callers outside of the Streams runtime to
    * access the internal state of stateful processors. See
    * https://kafka.apache.org/documentation/streams/developer-guide/interactive-queries.html
    * for more information.
    * @throws StreamsNotStartedException If Streams has not yet been started. Just call {@link
KafkaStreams#start()} and then retry this call.
    * @throws StreamsStoppedException If Streams is in a terminal state like PENDING_SHUTDOWN, NOT_RUNNING,
PENDING_ERROR, or ERROR. The caller should discover a new instance to query.
    * @throws UnknownStateStoreException If the specified store name does not exist in the topology.
    * /
  @Evolving
  public <R> StateQueryResult<R> query(StateQueryRequest<R> request);

  ...
}
```

StateQueryRequest

This is the main request object for IQv2. It contains all of the information required to execute the query. Note that, although this class is similar to the IQv1 request object `StoreQueryParameters` , we are proposing a new class to avoid unnecessary coupling between the old and new APIs' design goals.

This class implements a progressive builder pattern in an attempt to avoid the pitfalls we identified in Kafka Streams DSL Grammar. The progressive builder pattern allows us to supply arguments via semantically meaningful static and instance methods AND check required arguments at compile time.

The way it works is that the constructor is private and the first required argument is a static method. That method returns an intermediate interface that contains the next builder method, which either returns a new intermediate interface or the final class, depending on whether the following arguments are required or not. All of the optional arguments can be instance methods in the final interface.

Note: the "position bound" part of the proposal is an evolution on IQv1 and potentially controversial. The idea is to remove the need for the semantically loose `StoreQueryParameters#enableStaleStores` method. Instead of choosing between querying only the active store and querying unboundedly stale restoring and standby stores, callers can choose to query only the active store (latest), query *any* store (no bound), or to query a store that is at least past the point of our last query/queries. If this idea is controversial, we can separate it from this KIP and propose it separately.

```
/**
  * The request object for Interactive Queries.
  * This is an immutable builder class for passing all required and
  * optional arguments for querying a state store in Kafka Streams.
  * <p>
  * @param <R> The type of the query result.
  */
@Evolving
public class StateQueryRequest<R> {
```

```
/**
  * First required argument to specify the name of the store to query
  */
public static InStore inStore(final String name);

public static class InStore {

  /**
    * Second required argument to provide the query to execute.
    */
  public <R> StateQueryRequest<R> withQuery(final Query<R> query);
}

/**
  * Optionally bound the current position of the state store
  * with respect to the input topics that feed it. In conjunction
  * with {@link StateQueryResult#getPosition}, this can be
  * used to achieve a good balance between consistency and
  * availability in which repeated queries are guaranteed to
  * advance in time while allowing reads to be served from any
  * replica that is caught up to that caller's prior observations.
  * <p>
  * Note that the set of offsets provided in the bound does not determine
  * the partitions to query. For that, see {@link withPartitionsToQuery}.
  * Unrelated offsets will be ignored, and missing offsets will be treated
  * as indicating "no bound".
  */
public StateQueryRequest<R> withPositionBound(PositionBound positionBound);

/**
  * Specifies that this query should only run on partitions for which this instance is the leader
  * (aka "active"). Partitions for which this instance is not the active replica will return
  * {@link FailureReason#NOT_ACTIVE}.
  */
public StateQueryRequest<R> requireActive();

/**
  * Optionally specify the partitions to include in the query.
  * If omitted, the default is to query all locally available partitions
  */
public StateQueryRequest<R> withPartitions(Set<Integer> partitions);

/**
  * Query all locally available partitions
  */
public StateQueryRequest<R> withAllPartitions();


/**
  * Instruct Streams to collect detailed information during query
  * execution, for example, which stores handled the query, how
  * long they took, etc.
  */
public StateQueryRequest<R> enableExecutionInfo();

// Getters are also proposed to retrieve the request parameters

public String getStoreName();
public Query<R> getQuery();
public PositionBound getPositionBound();
public boolean executionInfoEnabled()

/**
  * empty set indicates that no partitions will be fetched
  * non-empty set indicate the specific partitions that will be fetched (if locally available)
  * throws UnsupportedOperationException if the request is to all partitions (isAllPartitions() == true)
  */
public Set<Integer> getPartitions();

/**
```

```
    * indicates that all locally available partitions will be fetched
    */
  public boolean isAllPartitions();

}
```

StateQueryResult

This is the main response object for IQv2. It wraps the individual results, as well as providing a vehicle to deliver metadata relating to the result as a whole.

```
/**
  * The response object for interactive queries.
  * It wraps the individual results, as well as providing a
  * vehicle to deliver metadata relating to the result as a whole.
  * <p>
  * @param <R> The type of the query result.
  */
@Evolving
public class StateQueryResult<R> {


  /**
    * Set the result for a global store query. Used by Kafka Streams and available for tests.
    */
  public void setGlobalResult(final QueryResult<R> r);

  /**
    * The query's result for global store queries. Is {@code null} for non-global (partitioned)
    * store queries.
    */
  public QueryResult<R> getGlobalResult();

  /**
    * Set the result for a partitioned store query. Used by Kafka Streams and available for tests.
    */
  public void addResult(final int partition, final QueryResult<R> r);

  /**
    * The query's result for each partition that executed the query.
    */
  public Map<Integer /*partition*/, QueryResult<R>> getPartitionResults();

  /**
    * Asserts that only one partition returns a result and extract the result.
    * Useful with queries that expect a single result.
    */
  public QueryResult<R> getOnlyPartitionResult()


  /**
    * The position of the state store at the moment it executed the
    * query. In conjunction
    * with {@link StateQueryRequest#withPartitionBound}, this can be
    * used to achieve a good balance between consistency and
    * availability in which repeated queries are guaranteed to
    * advance in time while allowing reads to be served from any
    * replica that is caught up to that caller's prior observations.
    */
  public Position getPosition();
}
```

StateStore modifications

This is the essence of the proposal. Rather than modifying each store interface to allow new kinds of queries, we introduce a generic capability of stores to execute query objects. This allows stores to choose whether they accept or reject queries of a given type, the introduction of new queries, etc.

Note that we are proposing a method with multiple arguments instead of a "parameters" object because this is an interface that users should implement to provide their own state stores. Following the strategy of adding/deprecating new method overloads allows us to modify this API over time by adding new methods with `default` implementations and control what happens in the default case (falling back to another method or returning an error), whereas if we had a "parameters" object, Streams would have no control or knowledge over whether stores use the new members or not, which makes API evolution more difficult to manage.

```
public interface StateStore {
  ...

  /**
    * Execute a query. Returns a QueryResult containing either result data or
    * a failure.
    * <p>
    * If the store doesn't know how to handle the given query, the result
    * will be a {@link FailureReason#UNKNOWN_QUERY_TYPE}.
    * If the store couldn't satisfy the given position bound, the result
    * will be a {@link FailureReason#NOT_UP_TO_BOUND}.
    * @param query The query to execute
    * @param offsetBound The position the store must be at or past
    * @param collectExecutionInfo Whether the store should collect detailed execution info for the query
    * @param <R> The result type
    */
  @Evolving
  default <R> QueryResult<R> query(Query<R> query,
                                   PositionBound positionBound,
                                   boolean collectExecutionInfo) {
    // If a store doesn't implement a query handler, then all queries are unknown.
    return QueryResult.forUnknownQueryType(query, this);
  }

    /**
     * Returns the position the state store is at with respect to the input topic/partitions
     */
    @Evolving
    default Position getPosition() {
        throw new UnsupportedOperationException(
            "getPosition is not implemented by this StateStore (" + getClass() + ")"
        );
    }

     ...
}
```

Query

This is the interface that all queries must implement. Part of what I'm going for here is to place as few restrictions as possible on what a "query" or its "result" is, so the only thing in this interface is a generic type indicating the result type, which lets Streams return a fully typed response without predetermining the response type itself.

```
@Evolving
public interface Query<R> { }
```

Using this interface, store implementers can create any query they can think of and return any type they like, from a single value to an iterator, even a future. While writing the POC, I implemented three queries for KeyValue stores, which I'll include here as examples:

Proposed Query: KeyQuery

This query implements the functionality of the current `KeyValueStore#get(key)` method:

```
@Evolving
public class KeyQuery<K, V> implements Query<V> {
  // static factory to create a new KeyQuery, given a key
  public static <K, V> KeyQuery<K, V> withKey(final K key);

  // getter for the key
  public K getKey();
}

// ====================================
// example usage in IQv2:

Integer key = 1;

// note that "mystore" is a KeyValueStore<Integer, ValueAndTimestamp<Integer>>,
// hence the result type
StateQueryRequest<ValueAndTimestamp<Integer>> query =
  inStore("mystore")
    .withQuery(KeyQuery.withKey(key));

// run the query
StateQueryResult<ValueAndTimestamp<Integer>> result = kafkaStreams.query(query);

// In this example, it doesn't matter which partition ran the query, so we just
// grab the result from the partition that returned something
// (this is what IQ currently does under the covers).

Integer value = result.getOnlyPartitionResult().getResult().value();

// ====================================
// For comparison, here is how the query would look in current IQ:
// (note, this glosses over many of the problems described above.
//  the example is here to illustrate the different ergonomics
//  of the two APIs.)

// create the query parameters
StoreQueryParameters<ReadOnlyKeyValueStore<Integer, ValueAndTimestamp<Integer>>> storeQueryParameters =
  StoreQueryParameters.fromNameAndType(
    "mystore",
    QueryableStoreTypes.timestampedKeyValueStore()
  );

// get a store handle to run the query
ReadOnlyKeyValueStore<Integer, ValueAndTimestamp<Integer>> store =
  kafkaStreams.store(storeQueryParameters);

// query the store
Integer value1 = store.get(key).value();
```

We propose to add KeyQuery as part of the implementation of this KIP, in order to have at least one query available to flesh out tests, etc., for the framework. Other queries are deferred to later KIPs.

Example Query: RawKeyQuery

We could later propose to add a "raw" version of the KeyQuery, which simply takes the key as a byte array and returns the value as a byte array. This could allow callers to bypass the serialization logic in the Metered stores entirely, potentially saving valuable CPU time and memory.

```
@Evolving
public class RawKeyQuery implements Query<byte[]> {
  public static RawKeyQuery withKey(final Bytes key);
  public static RawKeyQuery withKey(final byte[] key);
  public Bytes getKey();
}
```

Example query: RawScanQuery

This example demonstrates two variations on the first example:

1. The ability to define queries handling "raw" binary data for keys and values
2. The ability for a query to return multiple results (in this case, an iterator)

I'm only bundling those for brevity. We can also have typed, iterable queries and raw single-record queries.

Note this query is purely an example of what is possible. It will not be added as part of this KIP.

```
public class RawScanQuery implements Query<KeyValueIterator<Bytes, byte[]>> {
    private RawScanQuery() { }

    public static RawScanQuery scan() {
        return new RawScanQuery();
    }
}

// example usage

// since we're handling raw data, we're going to need the serdes
// this is just an example, this method is also not proposed in this KIP.
StateQuerySerdes<Integer, ValueAndTimestamp<Integer>> serdes =
  kafkaStreams.serdesForStore("mystore");

// run the "scan" query
StateQueryResult<KeyValueIterator<Bytes, byte[]>> scanResult =
  kafkaStreams.query(inStore("mystore").withQuery(RawScanQuery.scan()));

// This time, we'll get results from all locally available partitions.
Map<Integer, QueryResult<KeyValueIterator<Bytes, byte[]>>> partitionResults =
  scanResult.getPartitionResults();


// for this example, we'll just collate all the partitions' iterators
// together and print their data

List<KeyValueIterator<Bytes, byte[]>> iterators =
  partitionResults
    .values()
    .stream()
    .map(QueryResult::getResult)
    .collect(Collectors.toList());

// Using an example convenience method we could add to collate iterators' data
try (CloseableIterator<KeyValue<Bytes, byte[]>> collated = Iterators.collate(collect)) {
  while(collate.hasNext()) {
    KeyValue<Bytes, byte[]> next = collated.next();
    System.out.println(
      "|||" +
        " " + serdes.keyFrom(next.key.get()) +
        " " + serdes.valueFrom(next.value)
    );
  }
}
```

QueryResult

This is a container for a single partition's query result.

```
@Evolving
public class QueryResult<R> {
  // wraps a successful result
  public static <R> QueryResult<R> forResult(R result);

  // wraps a failure result
  public static <R> QueryResult<R> forFailure(FailureReason failureReason, String failureMessage);

  // returns a failed query result because the store didn't know how to handle the query.
  public static <R> QueryResult<R> forUnknownQueryType(Query<R> query, StateStore store);

  // returns a failed query result because the partition wasn't caught up to the desired bound.
  public static <R> QueryResult<R> notUpToBound(Position currentPosition, Position bound);

  // Used by state stores that need to delegate to another store to run a query and then
  // translate the results. Does not change the execution info or any other metadata.
  public <NewR> QueryResult<NewR> swapResult(NewR newTypedResult);

  // If requested, stores should record
  // helpful information, such as their own class, how they executed the query,
  // and the time they took.
  public void addExecutionInfo(String executionInfo);

  // records the point in the store's history that executed the query.
  public void setPosition(Position position);

  public boolean isSuccess();
  public boolean isFailure();
  public List<String> getExecutionInfo();
  public Position getPosition();
  public FailureReason getFailureReason();
  public String getFailureMessage();
  public getResult();
}
```

FailureReason

An enum classifying failures for individual partitions' failures

```
public enum FailureReason {
    /**
     * Failure indicating that the store doesn't know how to handle the given query.
     */
    UNKNOWN_QUERY_TYPE,

    /**
     * The query required to execute on an active task (via {@link StateQueryRequest#requireActive()}),
     * but while executing the query, the task was either a Standby task, or it was an Active task
     * not in the RUNNING state. The failure message will contain the reason for the failure.
     * <p>
     * The caller should either try again later or try a different replica.
     */
    NOT_ACTIVE,

    /**
     * Failure indicating that the store partition is not (yet) up to the desired bound.
     * The caller should either try again later or try a different replica.
     */
    NOT_UP_TO_BOUND,

    /**
     * Failure indicating that the requested store partition is not present on the local
     * KafkaStreams instance. It may have been migrated to another instance during a rebalance.
     * The caller is recommended to try a different replica.
     */
    NOT_PRESENT,

    /**
     * The requested store partition does not exist at all. For example, partition 4 was requested,
     * but the store in question only has 4 partitions (0 through 3).
     */
    DOES_NOT_EXIST;

    /**
     * The store that handled the query got an exception during query execution. The message
     * will contain the exception details. Depending on the nature of the exception, the caller
     * may be able to retry this instance or may need to try a different instance.
     */
    STORE_EXCEPTION; }
```

Position

A class representing a processing state position in terms of its inputs: a vector or (topic, partition, offset) components.

```
@Evolving
public class Position {

  // Create a new Position from a map of topic -> partition -> offset
  static Position fromMap(Map<String, Map<Integer, Long>> map);

  // Create a new, empty Position
  static Position emptyPosition();

  // Return a new position based on the current one, with the given component added
  Position withComponent(String topic, int partition, long offset);

  // Merge all the components of this position with all the components of the other
  // position and return the result in a new Position
  Position merge(Position other);

  // Get the set of topics included in this Position
  Set<String> getTopics();

  // Given a topic, get the partition -> offset pairs included in this Position
  Map<Integer, Long> getBound(String topic);
}
```

PositionBound

A class bounding the processing state Position during queries. This can be used to specify that a query should fail if the locally available partition isn't caught up to the specified bound. "Unbounded" places no restrictions on the current location of the partition.

EDIT: The KIP initially proposed to have a "latest" bound, which would require the query to run on a RUNNING Active task, but that has been moved to StateQueryRequest#requireActive. This more cleanly separates responsibilities, since StateStores are responsible for enforcing the PositionBound and Kafka Streams is responsible for handling the other details of StateQueryRequest. The problem was that the store actually doesn't have visibility into the state or type of task that contains it, so it was unable to take responsibility for the "latest" bound. It's important in particular to keep the StateStore's responsibilities clear because that is an extension point for Kafka Streams users who implement their own stores.

```
@Evolving
public class PositionBound {

  // Create a new Position bound representing "no bound"
  static PositionBound unbounded();

  // Create a new, empty Position
  static PositionBound at(Position position);

  // Check whether this is an "unbounded" Position bound
  boolean isUnbounded();

  // Get the Position (if it's not unbounded or latest)
  Position position();
}
```

StateStoreContext

To support persistence of position information across restarts (for persistent state stores, which won't read the changelog), we need to add a mechanism for the store to be notified when Streams is in a consistent state (after commit and before processing). This operation is purely a contract between the persistent, inner state store and the `{Global,Processor}StateManager` . No other components need to invoke that operation, and no components need to invoke it on the so registering a callback as part of `register` is better than

```
interface StateStoreContext {
...

// UNCHANGED EXISTING METHOD FOR REFERENCE:

     void register(final StateStore store,
                   final StateRestoreCallback stateRestoreCallback)

// NEW METHOD:


   /**
     * Registers and possibly restores the specified storage engine.
     *
     * @param store the storage engine
     * @param stateRestoreCallback the restoration callback logic for log-backed state stores upon restart
     * @param commitCallback a callback to be invoked upon successful task commit, in case the store
     *                           needs to perform any state tracking when the task is known to be in
     *                           a consistent state. If the store has no such state to track, it may
     *                           use {@link StateStoreContext#register(StateStore, StateRestoreCallback)}
instead.
     *                           Persistent stores provided by Kafka Streams use this method to save
     *                           their Position information to local disk, for example.
     *
     * @throws IllegalStateException If store gets registered after initialized is already finished
     * @throws StreamsException if the store's change log does not contain the partition
     */
    @Evolving
    void register(final StateStore store,
                  final StateRestoreCallback stateRestoreCallback,
                  final CommitCallback commitCallback);
...
}
```

Compatibility, Deprecation, and Migration Plan

Since this is a completely new set of APIs, no backward compatibility concerns are anticipated. Existing IQ usages will be free to migrate to the new IQv2 queries as they become available. Deprecation of the existing IQ API will be considered and proposed in a later KIP after IQv2 is determined to be feature complete, ergonomic, and performant enough to serve existing use cases.

Since the scope of this KIP is so large, we plan a short period of interface instability (during which all the IQv2 APIs are marked `@Evolving`) in case we discover a need to change this proposal. During this time, we will have the option to break compatibility in major (eg 4.0) or minor (eg 3.1), but not bugfix (eg 3.0.1) releases. We hope not to make use of this option, though. We plan to drop these annotations (in a later KIP) once IQv2 is complete enough to have confidence in the current design.

Since nothing is deprecated in this KIP, users have no need to migrate unless they want to.

Rejected Alternatives

Incrementally improve the current IQ API

Some of the motivating problems could be addressed within the scope of the current IQ API, but many are inherent in the current API's design. The full set of problems convinced us that it's time for a new API.

Defer the Position aspects to a later KIP

This would certainly simplify this KIP, but it would also mean we need to update the StateStore interface twice. If the Position proposal generates too much controversy, we will split it into a separate KIP.

Defer the Position functionality to individual StateStore/Query implementations

The IQv2 design leaves the door open for Queries and StateStores to coordinate to provide all kinds of functionality on top of "just" returning responses to queries. For example, new Queries and StateStores can be implemented to return asynchronous `Future<R>` results. So perhaps it is also a good idea to push the idea of bounding the Position into the "user space" of Query/StateStore implementations as well. We think that it could be done, but there are also advantages to making it a framework feature. The biggest one is simply uniformity. The framework Position tracking and bounding can ensure a consistent API across all query and store implementations, whereas pushing it to the implementations would make it merely conventional, raising future interoperability concerns. Position data also needs to be transmitted to standby replicas and made fault-tolerant, all of which becomes more complex in "user space" than in the framework.

Improve Query dispatch performance.

This was inspired by the fact that we don't have a great mechanism to dispatch Queries to their execution methods inside the state store. Presently, we have to either have a sequence of if/else type checks or a switch statement in the `StateStore#execute` implementation, or some other approach like a HashMap of Query method. Java 17 has the ability to switch over the class itself (see https://openjdk.java.net/jeps/406), but that's obviously a long way off for Apache Kafka.

Host the execution logic in the Query instead of the StateStore

Since the number of Query implementations will probably be much more than the number of StateStore implementations, we can move the type-checks from StateStore to Query by replacing `StateStore#execute(Query)` with `Query#executeOn(StateStore)`. Then, instead of having StateStore implementations define how to behave for each query, we would have the Queries define how to run on each StateStore.

The reason we didn't pursue this idea is that it makes it difficult for new (especially user-defined) StateStore implementations to re-use existing query types. For example, the KeyQuery/RawKeyQuery we propose to add in this KIP is very general, and presumably lots of custom stores would want to support it, but they would be unable to if doing so requires modification to the Query's implementation itself.

There's a more sophisticated version of this that allows both the Query and the StateStore to execute the query, so that the Query can try its (hopefully) more efficient dispatch first, and then defer to the StateStore's more flexible dispatch. This might be the best of both worlds in terms of dispatch performance, but it's quite complicated. We will consider this we are unable to make the current proposal perform acceptably.

Implement the Visitor pattern instead of checking the type of the Query

Although this proposal has some things in common with a Visitor-pattern situation, there are two different things that spoil this as a solution.

Following the Visitor pattern, the Query would call a method on the StateStore with itself as an argument, which allows Java to use the Query's knowledge of its own type to dispatch to the correct StateStore method. However, this means that the StateStore interface (not just any particular implementation) needs to define a method for each kind of query. We can have a hybrid model, in which the Query checks the top-level interface (say KeyValueStore) and invokes the execute method with itself as the argument, but this has many of the same downsides as the current IQ approach: namely that new queries would need to be implemented in a large number of store implementations, not just the particular bottom-level store implementation(s) that actually implement the query.

A related problem (and the reason we really need to change "a large number" of store implementations in the prior paragraph) is the "wrapped" state store design. The visitor pattern only works when you're invoking the Visitor interface that has a defined method for your concrete type. But in Streams, when you start to run a query, you're not querying (eg) a RocksDBStore. What you are really querying is probably a MeteredKeyValueStore wrapping a CachingKeyValueStore wrapping a ChangeLoggingKeyValueStore wrapping a RocksDBStore. So we'd wind either making the KeyValueStore interface to be our Visitor, and every KeyValueStore implementation would have to define handlers for all queries OR we'd automatically bypass all intervening layers and only run queries directly against the bottom-layer store. The first puts us in exactly the position we're in today: user-defined queries can't take advantage of special properties of user-defined stores, and the second means that we can't ever serve queries out of the cache.

It is possible that some kind of hybrid model here might make dispatch more efficient, but it seems like it comes at the cost of substantial complexity. As with the prior idea, we will consider this further if the performance of the current proposal seems to be too low.

Provide method to get store serdes

We previously proposed to add the following method to the KafkaStreams interface:

```
/**
  * Get a reference to the serdes used internally in a state store
  * for use with interactive queries. While many queries already
  * accept Java-typed keys and also return typed results, some
  * queries may need to handle the raw binary data stored in StateStores,
  * in which case, this method can provide the serdes needed to interpret
  * that data.
  * /
@Evolving
public <K, V> StateQuerySerdes<K, V> serdesForStore(String storeName);
```

This has been removed from the proposal because it is not necessary to implement the "parity" queries that we would implement today (to achieve parity with existing Interactive Query capabilities.

It may be necessary in the future to support high-performance "raw" queries that don't de/serialize query inputs or responses, but once we have a specific proposal for those queries, we may find that another solution is more suitable (such as returning types that wrap their serdes and can be used either way). Regardless, we can defer this decision to a later KIP and simplify this proposal.

Add a remote query capability to Kafka Streams

This is a minor point just calling out that the scope of Interactive Query has always just been to fetch locally available data, and that this IQv2 proposal does not change that scope.

We think that a remote query capability would be an interesting contribution to Kafka Streams, but it would also be a major design effort and proposal in its own right, with significant challenges to be overcome. Therefore, we propose to keep this KIP orthogonal to the idea of remote queries and focus on improving the functionality of local query fetching.

Also updating the IQ metadata APIs

There have been some problems noted with the metadata APIs that support IQ, such as queryMetadataForKey, streamsMetadataForStore, and allLocalStorePartitionLags. Updating those APIs would be valuable, but to control the scope of this proposal, we choose to treat that as future work.

Add RawKeyQuery

I originally proposed to use KeyQuery for the typed query/result and have a separate class, RawKeyQuery, for the serialized query/result that's handled by the lower store layers. During the proposal, I though this would be simpler, but during implementation and also while proposing other queries, it became apparent that it was actually more complicated. Therefore, RawKeyQuery is now only presented as an example and not actually proposed in this KIP.