KIP-67: Queryable state for Kafka Streams

Status

Current state: *Accepted*

Discussion thread: *here*

JIRA: *KAFKA-3909*

Released: 0.10.1.0

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

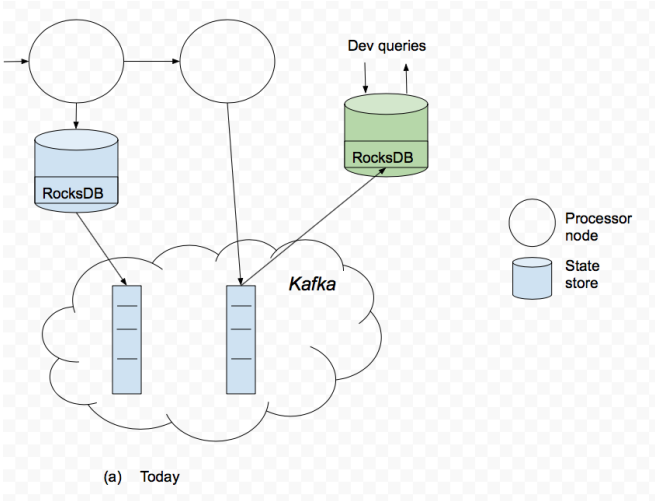
Motivation

Today a Kafka Streams application will implicitly create state. This state is used for storing intermediate data such as aggregation results. The state is also used to store KTable's data when they are materialized. The problem this document addresses is that this state is hidden from application developers and they cannot access it directly. The DSL allows users to make a copy of the data (using the through operator) but this leads to a doubling in the amount of state that is kept. In addition, this leads to extra IOs to external databases/key value stores that could potentially slow down the entire pipeline.

Here is a simple example that illustrates the problem:

```
1 KTable<String, Long> wordCounts = textLine

2    .flatMapValues(value ->Arrays.asList(value.toLowerCase().split("\\W+")))

3    .map((key, word) -> new KeyValue<>(word, word)

4    .countByKey("StoreName")

5 wordCounts.to(Serdes.String(), Serdes.Long(), "streams-wordcount-output");
```

In line 4, the aggregation already maintains state in a store called StoreName, however that store cannot be directly queried by the developer. Instead, the developer makes a copy of the data in that store into a topic called streams-wordcount-output. Subsequently, the developer might instantiate its own database after reading the data from that topic (this step is not shown above). This is shown in illustration (a):



Proposed changes

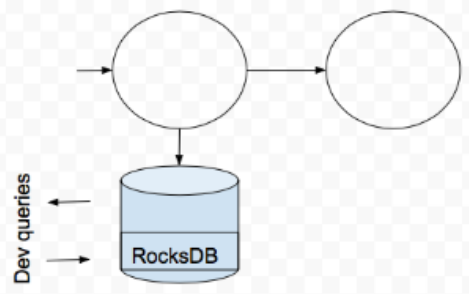
We propose to make the Kafka Streams internal state stores visible to external queries. The internal state stores include all processor node stores used implicitly (through DSL) or explicitly (through low level Processor API). Benefits of this approach include:

- Avoid duplicating data
- Avoid extra IOs
- Fewer moving pieces in the end-to-end architecture by avoiding unneeded state stores
- Ability to colocate data and processing (e.g., in situations where many rows are scanned per operation).

The proposal can be thought as having two distinct steps as described below.

Step 1 in proposal: expose state store names to DSL and local queries

The stream store namespace is local to a KStreams instance, i.e., it is part of the same process that the KStreams instance is in. Conceptually the code to access such a store would look like this:

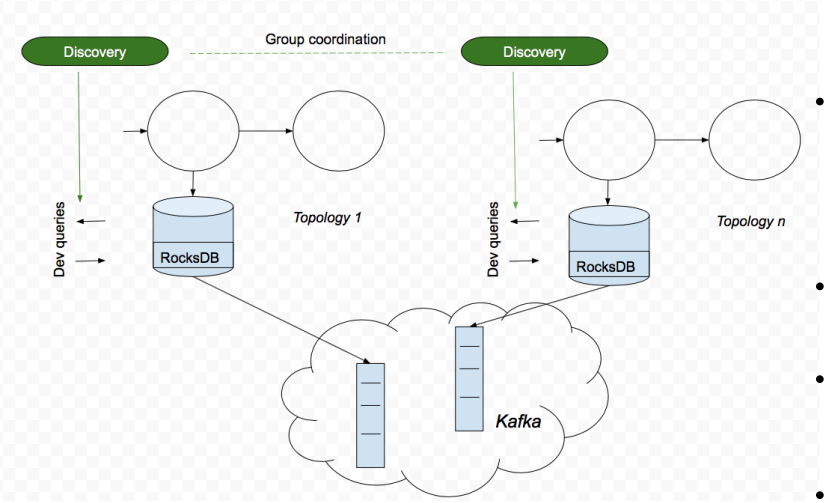



```
KafkaStreams streams = new KafkaStreams(..);

ReadOnlyKeyValueStore store = streams.store("storeName", QueryableStoreTypes.keyValueStore(
);
```

The state store is discovered by querying the KafkaStreams instance. The query operations will be read-only, i.e., no updates. The query method is calling methods of the StateStore object.

Step 2 in proposal: global discovery of state stores

The second step adds to the first by having the namespace be global. The figure below shows two KStream instances on potentially different servers and two state stores each backed up by their own Kafka topic. An additional proposed component in the diagram is a logical discovery component. Conceptually this component provides a lookup API that keeps track of the mapping between a state store name and the KafkaStreams instance that owns that state store.



Hence, the discovery API is part of the KafkaStreams instance. The API will provide four methods:

- Collection<StreamsMetadata> KafkaStreams.allMetadata()) where StreamsMetadata has fields such as list of assigned partitions, list of state store names and HostInfo that includes hostname / port, etc. The port is the listening port of a user-defined listener that users provide to listen for queries (e.g., using REST APIs). More on the user-defined agent below.
- Collection<StreamsMetadata> KafkaStreams.allMetadataForStore(String /* storeName */) would return only the StreamsMetadata that include the given store name.
- StreamsMetadata KafkaStreams.metadataWithKey(String storeName, K key, Serializer<K> serializer) would return the StreamsMetadata that has the store that might have the key (the key might not exist in any store). The default StreamPartitioner will be used for key partitioning.
- StreamsMetadata KafkaStreams.metadataForKey(String storeName, K key, StreamPartitioner<K, ?> partitioner) same as above but will use the provided StreamPartitioner

An additional configuration parameter, StreamsConfig.APPLICATION_SERVER_CONFIG, will be added. This is a host:port pair supplied by the streams developer and should map to a Server running in the same instance of the KafkaStreams application. The supplied host:port pair will form part of the StreamsMetadata returned from the the above mentioned API calls.

The individual KafkaStreams instances will know about all mappings between state store names, keys and KafkaStream instances. We propose that they keep track of this information by piggybacking on the consumer group membership protocol. In particular, we propose to add the above mapping Map<HostInfo, Set<TopicPartition>> to StreamPartitionAssignor so that each consumer knows about all the other tasks metadata and host states in the system.

Bootstrapping. To bootstrap the discovery, a user can simply query the discovery instance of any one of the KafkaStream instances she operates, i.e., bootstrap happens within a single KafkaStream instance.

User-defined listener. This design option simply provides a discovery API and the ability to find a state store. Once a state store is found, then the store is queried using the APIs in step 1. However, this KIP stops short of providing a listener on each KafkaStream instance that actually listens for remote queries (e.g., through REST). It is assumed that the user will write such a listener themselves. So the scope for Apache Kafka is the discovery API (through existing consumer membership protocol) and the queries as specified in the step 1.

Operations on state stores

The focus is on querying state stores, not updating them. It is not clear what it would mean to update a state store from outside the stream processing framework. Such updates are likely to introduce undefined behavior to the framework. This KIP does not support updates.

The operations of the state store will be the query-only operations of whatever state store the user has plugged in. Some examples (full API in a subsequent section):

- If the plugged in state store is a RocksDB key-value state store: V get(K key);
- If the plugged in state store is a RocksDB-backed window store: WindowStoreIterator<V> fetch(K key, long timeFrom, long timeTo);

Handling failures

Tasks, and thus StateStores, can be re-assigned to threads anytime. This could happen after a failure of a server on which KafkaStreams is running, or for load balancing. During any re-assignment of state stores, they will not be available for querying until they are fully restored and ready to be used by the KafkaStreams instance.

The implication is that the user's code will not have a direct reference to the underlying StateStores, but rather an Object that knows how to locate and query them. Whenever operations are made against this object it must first check if the underlying StateStore is valid. If the StateStore is not valid the operation needs to return an exception InvalidStateStore. On catching this exception the user can try again. If the rebalance has finished and the StateStores are open, subsequent operations will be successful.

State store names

Today, some state store names are hidden from the developer and are internal to Kafka Streams. This makes it difficult for a developer to query them. Hence, as part of this proposal we add the need to expose all relevant state store names to the user. This means three things:

1. Explicitly require names for all operations that create a state store, like aggregate.
2. Explicitly require names for all created KTables (thus their changelog state stores)
3. Materialize all KTables (today some KTables are materialized and some are not)

Public Interfaces

Query API

We propose adding an additional method to the KafkaStreams public API:

```
/**
 * Find the store with the provided storeName and of the type as
 * specified by QueryableStoreType. Will throw an UnknownStoreException
 * if a store with the given name and type is not found
 */
public <T> T store(final String storeName,
                      final QueryableStoreType<T> queryableStoreType)
```

The QueryableStoreType interface, below, can be used to 'plug-in' different StateStore implementations to Queryable State. Developers using the Process or API and supplying custom StateStore's can get access to their StateStores with the same API method as above. Implementations of this interface for the StateStores that are part of the KafkaStreams library will be provided by this KIP.

```
public interface QueryableStoreType<T> {
    /**
     * Called when searching for {@StateStore}s to see if they
     * match the type expected by implementors of this interface
     * @param stateStore    The stateStore
     * @return true if it is a match
     */
    boolean accepts(final StateStore stateStore);
    /**
     * Create an instance of T (usually a facade) that developers can use
     * to query the Underlying {@StateStore}s
     * @param storeProvider     provides access to all the underlying StateStore instances of type T
     * @param storeName         The name of the Store
     * @return  T usually a read-only interface over a StateStore @see {@link   QueryableStoreTypes.
KeyValueStoreType}
     */
    T create(final StateStoreProvider storeProvider, final String storeName);
}
```

A class that provides implementations of the QueryableStoreTypes that are part of KafkaStreams,i.e.,

Two new interfaces to restrict StateStore access to Read Only (note this only applies to implementations that are part of Kafka Streams)

```
/* A window store that only supports read operations
*
* @param <K> Type of keys
* @param <V> Type of values
*/
public interface ReadOnlyWindowStore<K, V> {
   /**
    * Get all the key-value pairs with the given key and the time range from all
    * the existing windows.
    *
    * @return an iterator over key-value pairs {@code <timestamp, value>}
    */
   WindowStoreIterator<V> fetch(K key, long timeFrom, long timeTo);
}
/**
* A key value store that only supports read operations
* @param <K> the key type
* @param <V> the value type
*/
public interface ReadOnlyKeyValueStore<K, V> {
   /**
    * Get the value corresponding to this key
    *
    * @param key The key to fetch
    * @return The value or null if no value is found.
    * @throws NullPointerException If null is used for key.
    */
   V get(K key);
  /**
    * Get an iterator over a given range of keys.
    * This iterator MUST be closed after use.
    *
    * @param from The first key that could be in the range
    * @param to The last key that could be in the range
    * @return The iterator for this range.
    * @throws NullPointerException If null is used for from or to.
    */
   KeyValueIterator<K, V> range(K from, K to);
   /**
    * Return an iterator over all keys in the database.
    * This iterator MUST be closed after use.
    *
    * @return An iterator of all key/value pairs in the store.
    */
   KeyValueIterator<K, V> all();
}
```

We can then use the above API to get access to the stores like so:

```
final ReadOnlyKeyValueStore<String, Long>
   myCount = kafkaStreams.store("my-count", QueryableStoreTypes.<String, Long>keyValueStore());
final ReadOnlyWindowStore<String, String>
   joinOther =
   kafkaStreams.store("join-other", QueryableStoreTypes.<String, String>windowStore());
```

Discovery API

Exposed APIs from Kafka Streams:

```
/**
 * A new config will be added to StreamsConfig
 * A user defined endpoint that can be used to connect to remote KafkaStreams instances.
 * Should be in the format host:port
 */
public static final String APPLICATION_SERVER_CONFIG = "application.server";



public class HostInfo {
   String hostname;    /* hostname for instance that contains state store */
   int port;           /* listening port for instance that contains state store */
}
public class StreamsMetadata {
    private final HostInfo hostInfo; /* hostInfo from above */
    private final Set<String> stateStores; /* state stores on this instance */
    private final Set<TopicPartition> topicPartitions; /* TopicPartitions on this instance */
}

/**
 * Find all of the instances of {@link StreamsMetadata} in a {@link KafkaStreams application}
 * Note: this is a point in time view and it may change due to partition reassignment.
 * @return collection containing all instances of {@link StreamsMetadata} in this application
 */
public Collection<StreamsMetadata> allMetadata();

/**
 * Find the instances {@link StreamsMetadata} for a given storeName
 * Note: this is a point in time view and it may change due to partition reassignment.
 * @param storeName the storeName to find metadata for
 * @return  A collection containing instances of {@link StreamsMetadata} that have the provided storeName
 */
public Collection<StreamsMetadata> allMetadataForStore(final String storeName);

/**
 * Find the {@link StreamsMetadata} for a given storeName and key.
 * Note: the key may not exist in the {@link org.apache.kafka.streams.processor.StateStore},
 * this method provides a way of finding which host it would exist on.
 * Note: this is a point in time view and it may change due to partition reassignment.
 * @param storeName         Name of the store
 * @param key               Key to use to for partition
 * @param keySerializer     Serializer for the key
 * @param <K>               key type
 * @return  The {@link StreamsMetadata} for the storeName and key
 */
public <K> StreamsMetadata metadataForKey(final String storeName,
                                                                                                   final K key,
                                                   final Serializer<K> keySerializer);

/**
 * Find the {@link StreamsMetadata} for a given storeName and key.
 * Note: the key may not exist in the {@link org.apache.kafka.streams.processor.StateStore},
 * this method provides a way of finding which host it would exist on.
 * Note: this is a point in time view and it may change due to partition reassignment.
 * @param storeName         Name of the store
 * @param key               Key to use to for partition
 * @param partitioner       Partitioner for the store
 * @param <K>               key type
 * @return  The {@link StreamsMetadata} for the storeName and key
 */
public <K> StreamsMetadata metadataForKey(final String storeName,
                                                   final K key,
                                                   final StreamPartitioner<K, ?> partitioner);
```

Below is an example of how a developer might use the Discover API

```
public static void main(String[] args) throws Exception {
  Properties streamsConfiguration = new Properties();
```

```
  ...
  /**
   * To use the Discovery API the developer must provide an host:port pair that
   * maps to an embedded service listening on this address. i.e.,
   * it is up to the developer to define the protocol and create the service
   */
  streamsConfiguration.put(StreamsConfig.APPLICATION_SERVER_CONFIG, "localhost:7070");
  ...
  KafkaStreams streams = new KafkaStreams(builder, streamsConfiguration);
  streams.start();

  /**
   * Start your embedded service listening on the endpoint
   * provided above
   */
  new QueryableStateProxy(streams).start(7070);
}

/**
 * Example Rest Proxy that uses the Discovery API to locate the
 * KafkaStreamsInstances StateStores are running on. A developer would first connect
 * to a well-known instance to find where a particular store, or store with key,
 * is located. They'd then use the returned KafkaStreamsInstances
 * to connect to the appropriate instances and perform queries, i.e, :
 * KafkaStreamsInstance instance = http.get("http://well-known-host:8080/state/instance/my-store/my-key");
 * Long result = http.get("http://" + instance.host() + ":" + instance.port() + "/state/stores/my-store/my-
key");
 */
@Path("state")
public class QueryableStateProxy {
   /**
    * The KafkaStreams instance knows about all of the other instances
    * in the application. This is maintained by the StreamPartitionAssignor
    * on partition assignments (rebalances)
 */
   private final KafkaStreams streams;
   public QueryableStateProxy(final KafkaStreams streams) {
      this.streams = streams;
   }

   @GET()
   @Path("/instances")
   public Response streamsMetadata() {
         // get the current collection of StreamsMetadata
     final Collection<StreamsMetadata> metadata = streams.allMetadata();
     return respondWithMetadata(metadata);
   }

   @GET()
   @Path("/instances/{storeName}")
   public Response streamsMetadataForStore(@PathParam("storeName") String store) {
      // find all the metadata that have the provided store
      final Collection<StreamsMetadata> metadata = streams.allMetadataForStore(store);
      return respondWithMetadata(metadata);
   }

   @GET()
   @Path("/instance/{storeName}/{key}")
   public Response streamsMetadataForStoreAndKey(@PathParam("storeName") String store,
                                                 @PathParam("key") String key) {

         // locate the instance that would have the store with the provided key (if the key exists)
      final StreamsMetadata metadata = streams.metadataForKey(store, key, new StringSerializer());
      if (instance == null) {
        return Response.noContent().build();
      }
      return Response.ok(metadata.toString()).build();
   }

   @GET()
   @Path("/stores/{storeName}/{key}")
```

```
   public Response byKey(@PathParam("storeName") String storeName, @PathParam("id") String key) {
          // Get a handle on the Store for the provides storeName
      final ReadOnlyKeyValueStore<String, Long> store = streams.store(storeName,
                                                                      QueryableStoreTypes.keyValueStore());
      // store may not exist or might not exist yet, i.e, if partitions haven't been assigned or
      // a rebalance is in process
      if (store == null) {
        return Response.noContent().build();
      }

          // we have a store so we can get the result
      final Long result = store.get(key);
      if (result == null) {
        return Response.noContent().build();
      }
      return Response.ok(result).build();
    }

        public void start(int port) {
     // start your favourite http server
     ...
    }
}
```

Developer Guide for Custom Stores

If you want to make a customized store Queryable you'll need to implement the QueryableStoreType interface. Below are example implementations for Key ValueStore and WindowStore. You may also want to provide an interface to restrict operations to read-only and a Composite type for providing a faade over the potentially many instances of the underlying store (see example CompositeReadOnlyKeyValueStore below).

```
public class QueryableStoreTypes {
   public static <K, V> QueryableStoreType<ReadOnlyKeyValueStore<K, V>> keyValueStore() {
       return new KeyValueStoreType<>();
   }
   public static <K, V> QueryableStoreType<ReadOnlyWindowStore<K, V>> windowStore() {
       return new WindowStoreType<>();
   }
   public static abstract class QueryableStoreTypeMatcher<T> implements QueryableStoreType<T> {
       private final Class matchTo;
       public QueryableStoreTypeMatcher(Class matchTo) {
           this.matchTo = matchTo;
       }
       @SuppressWarnings("unchecked")
       @Override
       public boolean accepts(final StateStore stateStore) {
           return matchTo.isAssignableFrom(stateStore.getClass());
       }
   }
   private static class KeyValueStoreType<K, V> extends
                                            QueryableStoreTypeMatcher<ReadOnlyKeyValueStore<K, V>> {
       KeyValueStoreType() {
           super(ReadOnlyKeyValueStore.class);
       }
       @Override
       public ReadOnlyKeyValueStore<K, V> create(
           final UnderlyingStoreProvider<ReadOnlyKeyValueStore<K, V>> storeProvider,
           final String storeName) {
           return new CompositeReadOnlyStore<>(storeProvider, storeName);
       }
   }
   private static class WindowStoreType<K, V> extends QueryableStoreTypeMatcher<ReadOnlyWindowStore<K,V>> {
       WindowStoreType() {
           super(ReadOnlyWindowStore.class);
       }
```

```
        @Override
        public ReadOnlyWindowStore<K, V> create(
            final UnderlyingStoreProvider<ReadOnlyWindowStore<K, V>> storeProvider,
            final String storeName) {
            return new CompositeReadOnlyWindowStore<>(storeProvider, storeName);
        }
    }
}

public interface StateStoreProvider {
         /**
     * Find instances of StateStore that are accepted by {@link QueryableStoreType#accepts} and
     * have the provided storeName.
     *
     * @param storeName             name of the store
     * @param queryableStoreType    filter stores based on this queryableStoreType
     * @param <T>                   The type of the Store
     * @return  List of the instances of the store in this topology. Empty List if not found
     */
    <T> List<T> stores(String storeName, QueryableStoreType<T> queryableStoreType);
}

/**
 * A wrapper over the underlying {@link ReadOnlyKeyValueStore}s found in a {@link
 * org.apache.kafka.streams.processor.internals.ProcessorTopology}
 *
 * @param <K> key type
 * @param <V> value type
 */
public class CompositeReadOnlyKeyValueStore<K, V> implements ReadOnlyKeyValueStore<K, V> {

    private final StateStoreProvider storeProvider;
    private final QueryableStoreType<ReadOnlyKeyValueStore<K, V>> storeType;
    private final String storeName;

    public CompositeReadOnlyKeyValueStore(final StateStoreProvider storeProvider,
                                          final QueryableStoreType<ReadOnlyKeyValueStore<K, V>> storeType,
                                          final String storeName) {
        this.storeProvider = storeProvider;
        this.storeType = storeType;
        this.storeName = storeName;
    }

    @Override
    public V get(final K key) {
        final List<ReadOnlyKeyValueStore<K, V>> stores = storeProvider.getStores(storeName, storeType);
        for (ReadOnlyKeyValueStore<K, V> store : stores) {
            V result = store.get(key);
            if (result != null) {
                return result;
            }
        }
        return null;
    }

   //...
}
```

Proposed implementation outline

The implementation has three logical parts. First, expose all state store names to the DSL. Note that we don't need to do this for the processor API since the state stores are already explicit there. Second, implement the interfaces that query a state store.

For the first part, an implementation outline follows:

1. Expose stream names to the DSL: see KAFKA-3870 PR.
2. Always enforce KTable materialization upon creation (i.e. even for "builder.table(..).to()" we also materialize it into a state store; this will indeed incur unnecessary overhead but should be very rare, and it helps for consistency of materialization).
3. We must also document the valid characters for a state store name. Since the name is directly used to name underlying changelog topics, for example, only such characters are currently valid that are also valid characters to name a Kafka topic.

For the second part, an implementation outline follows:

1. Implement read-only interface and QueryableStoreType for key value state store
2. Implement read-only interface and QueryableStoreType for windowed state store
3. Ensure store operations are thread-safe. I.e., a single state store could have concurrent access, e.g. one stream worker thread updating the store, and one user thread querying the store.
4. Ensure failure and rebalancing is handled correctly. I.e., because the "read-only" state store interface could actually wrap multiple state stores underneath, since one instance could contain multiple tasks and hence multiple state stores with the same names. Rebalance could happen migrating state stores in / out while the user is interacting with the interface.
5. See Expose Local Stores for how this could be done

For the third part:

1. Update group membership data to include discovery endpoints
2. Enable discovery

Compatibility, Deprecation, and Migration Plan

- *The query and discovery APIs will not affect existing users.*
- *However, exposing state store names to the API will affect existing users, since the current interfaces will change. The new interfaces will not be backwards compatible.*
- *As we now need to handle concurrent access to the State Stores, this may incur some overhead on streams applications while querying is on-going. We will measure the overhead in benchmarks.*

Rejected Alternatives

Querying directly from Kafka topic. We considered allowing users to query directly from a Kafka topic, instead of a state store. The namespace would be global in this case, but instead of worrying about the StateStore namespace, we would be interested in the topic names instead (each state store is often backed into a Kafka topic). A user would create a RocksDb StateStore and instruct it to cache data from the desired topic. From then on, queries are performed on that local state store using the query method of direct calls, just like the above KIP proposal. Discovery with this option is implicitly done by the consumer that reads from the topic.

One potential drawback of this method is lack of consistency. In particular, the latest records for a topic might be residing on a remote StateStore (that acts like a cache for that topic). Hence, the user might not see the latest value for their data. Another limitation is that it might not be possible to run this on a single machine. A large keyspace could result in too much data to fit in memory and/or on local disk. It might be hard to provision the downstream machine correctly. This would then require another ‘cluster’ to host the Rocks DB caches. Furthermore, if you have N applications that need to query the state, you need to duplicate the state N times. Another limitation is that state stores that do not persist to Kafka (e.g., an in-memory state store) cannot be queried using this design.