KIP-150 - Kafka-Streams Cogroup

Status

Current state: Accepted (vote)

Discussion thread: here

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

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

Motivation

When multiple streams aggregate together to form a single larger object (eg. A shopping website may have a cart stream, a wish list stream, and a purchases stream. Together they make up a Customer.), it is very difficult to accommodate this in the Kafka-Streams DSL. It generally requires you to group and aggregate all of the streams to KTables then make multiple outerjoin calls to end up with a KTable with your desired object. This will create a state store for each stream and a long chain of ValueJoiners that each new record must go through to get to the final object.
Creating a cogroup method where you use a single state store will:

1. Reduce the number of gets from state stores. With the multiple joins when a new value comes into any of the streams a chain reaction happens where ValueGetters keep calling ValueGetters until we have accessed all state stores.
2. Slight performance increase. As described above all ValueGetters are called also causing all ValueJoiners to be called forcing a recalculation of the current joined value of all other streams, impacting performance.

Example with Current API:

```
KTable<K, V1> table1 = builder.stream("topic1").groupByKey().aggregate(initializer1, aggregator1,
materialized1);
KTable<K, V2> table2 = builder.stream("topic2").groupByKey().aggregate(initializer2, aggregator2,
materialized2);
KTable<K, V3> table3 = builder.stream("topic3").groupByKey().aggregate(initializer3, aggregator3,
materialized3);
KTable<K, CG> cogrouped = table1.outerJoin(table2, joinerOneAndTwo).outerJoin(table3, joinerOneTwoAndThree);
```

As you can see this creates 3 StateStores, and in the Materialized parameter requires 3 initializers, and 3 aggValueSerdes. This also adds the pressure to user to define what the intermediate values are going to be (V1, V2, V3). They are left with a couple choices, first to make V1, V2, and V3 all the same as CG and the two joiners are more like mergers, or second make them intermediate states such as Topic1Map, Topic2Map, and Topic3Map and the joiners use those to build the final aggregate CG value. This is something the user could avoid thinking about with this KIP.

When a new input arrives lets say at "topic1" it will first go through a KStreamAggregate grabbing the current aggregate from storeName1. It will produce this in the form of the first intermediate value and get sent through a KTableKTableOuterJoin where it will look up the current value of the key in storeName2. It will use the first joiner to calculate the second intermediate value, which will go through an additional KTableKTableOuterJoin. Here it will look up the current value of the key in storeName3 and use the second joiner to build the final aggregate value.

If you think through all possibilities for incoming topics you will see that no matter which topic it comes in through all three stores are queried and all of the joiners must get used.

Topology wise for N incoming streams this creates N KStreamAggregates, 2*(N-1) KTableKTableOuterJoins, and N-1 KTableKTableJoinMergers.

Example with Proposed API:

```
KGroupedStream<K, V1> grouped1 = builder.stream("topic1").groupByKey();
KGroupedStream<K, V2> grouped2 = builder.stream("topic2").groupByKey();
KGroupedStream<K, V3> grouped3 = builder.stream("topic3").groupByKey();
KTable<K, CG> cogrouped = grouped1.cogroup(aggregator1)
        .cogroup(grouped2, aggregator2)
        .cogroup(grouped3, aggregator3)
        .aggregate(initializer1, materialized1);
```

As you can see this creates 1 StateStore, and in the Materialized parameter requires 1 initializer, and 1 aggValueSerde. The user no longer has to worry about the intermediate values and the joiners. All they have to think about is how each stream impacts the creation of the final CG object. The idea is that you can collect many grouped streams with overlapping key spaces and any kind of value types. Once aggregated its value will be reduced into one type. This is why the user need only one initializer. Each aggregator will need to integrate the new value with the new object made in the initializer.

When a new input arrives lets say at "topic1" it will first go through a KStreamAggreagte and grab the current aggregate from storeName1. It will add its incoming object to the aggregate, update the store and pass the new aggregate on. This new aggregate goes through the KStreamCogroup which is pretty much just a pass through processor and you are done.

Topology wise for N incoming streams the new api will only every create N KStreamAggregates and 1 KStreamCogroup.

Concrete Example:

```
public class Customer {
    List<Item> cart;
    List<Item> purchases;
    List<Item> wishList;
}
```

There are 3 streams: cart, purchases, and wish-list.

We would construct 3 aggregators in which we add the item to the appropriate list. One of these would look like:

```
private static final Aggregator<String, Item, Customer> CART_AGGREGATOR = new Aggregator<String, Item,
Customer>() {
    @Override
    public Patient apply(String key, Item value, Customer aggregate) {
        aggregate.cart.add(value);
        return aggregate;
    }
};
```

Then we would create the topology:

```
KGroupedStream<Long, Item> groupedCart = builder.stream("cart").groupByKey();
KGroupedStream<Long, Item> groupedPurchases = builder.stream("purchases").groupByKey();
KGroupedStream<Long, Item> groupedWishList = builder.stream("wish-list").groupByKey();
KTable<Long, Customer> customers = groupedCart.cogroup(CART_AGGREGATOR)
        .cogroup(groupedPurchases, PURCHASE_AGGREGATOR)
        .cogroup(groupedWishList, WISH_LIST_AGGREGATOR)
        .aggregate(initializer, materialized);
customers.to("customers");
```

Now imagine the streams get the following values:

Stream "cart":

1L, Item[no:01]
2L, Item[no:02]
1L, Item[no:03]
1L, Item[no:04]
2L, Item[no:05]

Stream "purchases":

2L, Item[no:06]
1L, Item[no:07]
1L, Item[no:08]
2L, Item[no:09]
2L, Item[no:10]

Stream "wish-list":

1L, Item[no:11]
2L, Item[no:12]
2L, Item[no:13]
2L, Item[no:14]
2L, Item[no:15]

After all items have flown through the topology, you could expect to see the following outputs in "customers":

1L, Customer[
cart:{Item[no:01], Item[no:03], Item[no:04]},
purchases:{Item[no:07], Item[no:08]},
wishList:{Item[no:11]}
]
2L, Customer[
cart:{Item[no:02], Item[no:05]},
purchases:{Item[no:06], Item[no:09], Item[no:10]},
wishList:{Item[no:12], Item[no:13], Item[no:14], Item[no:15]}
]

It is important to note that intermediate values would also be produced, unless they are processed closely enough together that caching prevents this. (eg. After first item is processed from "cart" stream customer 1L would be output with only that first item in its cart and no items in the purchases or wishlist.)

Public Interfaces

```
KGroupedStream <K, V> {
...
        <T> CogroupedKStream<K, T> cogroup(final Aggregator<? super K, ? super V, T> aggregator);
}
```

```
/**
* {@code CogroupedKStream} is an abstraction of multiple <i>grouped</i> record streams of {@link KeyValue}
pairs.
* It is an intermediate representation of one or more {@link KStream}s in order to apply one or more aggregation
* operations on the original {@link KStream} records.
* <p>
* It is an intermediate representation after a grouping of {@link KStream}s, before the aggregations are
applied to
* the new partitions resulting in a {@link KTable}.
* <p>
* A {@code CogroupedKStream} must be obtained from a {@link KGroupedStream} via
* {@link KGroupedStream#cogroup(Initializer, Aggregator, org.apache.kafka.common.serialization.Serde, String)
cogroup(...)}.
*
* @param <K> Type of keys
* @param <V> Type of aggregate values
*/
public interface CogroupedKStream<K, Vout> {
    <Vin> CogroupedKStream<K, Vout> cogroup(final KGroupedStream<K, Vin> groupedStream,
                                            final Aggregator<? super K, ? super Vin, Vout> aggregator);

    KTable<K, Vout> aggregate(final Initializer<Vout> initializer,
                              final Materialized<K, Vout, KeyValueStore<Bytes, byte[]>> materialized);

    KTable<K, Vout> aggregate(final Initializer<Vout> initializer);

    KTable<K, Vout> aggregate(final Initializer<Vout> initializer,
                                                    final Named named,
                              final Materialized<K, Vout, KeyValueStore<Bytes, byte[]>> materialized);

    KTable<K, Vout> aggregate(final Initializer<Vout> initializer,
                                                    final Named named);

    <W extends Window> TimeWindowedCogroupedKStream<K, Vout> windowedBy(final Windows<W> windows);

    SessionWindowedCogroupedKStream<K, Vout> windowedBy(final SessionWindows sessionWindows);

}
```

```
/**
 * {@code SessionWindowedCogroupKStream} is an abstraction of a <i>windowed</i> record stream of {@link org.
apache.kafka.streams.KeyValue} pairs.
 * It is an intermediate representation of a {@link KGroupedStream} in order to apply a windowed aggregation
operation on the original
 * {@link KGroupedStream} records.
 * <p>
 * It is an intermediate representation after a grouping, cogrouping and windowing of a {@link KStream} before
an aggregation is applied to the
 * new (partitioned) windows resulting in a windowed {@link KTable}
 * (a <emph>windowed</emph> {@code KTable} is a {@link KTable} with key type {@link Windowed Windowed<K>}.
 * <p>
 * The specified {@code SessionWindows} define the gap between windows.
 * The result is written into a local windowed {@link org.apache.kafka.streams.state.KeyValueStore} (which is
basically an ever-updating
 * materialized view) that can be queried using the name provided in the {@link Materialized} instance.
 *
 * New events are added to windows until their grace period ends (see {@link TimeWindows#grace(Duration)}).
 *
 * Furthermore, updates to the store are sent downstream into a windowed {@link KTable} changelog stream, where
 * "windowed" implies that the {@link KTable} key is a combined key of the original record key and a window ID.

 * A {@code WindowedKStream} must be obtained from a {@link KGroupedStream} via {@link KGroupedStream#windowedBy
(Windows)} .
 *
 * @param <K> Type of keys
 * @param <V> Type of values
 * @see KStream
 * @see KGroupedStream
 * @see CogroupedKStream
 */
import org.apache.kafka.streams.state.SessionStore;

public interface SessionWindowedCogroupedKStream<K, V> {
    KTable<Windowed<K>, Vout> aggregate(final Initializer<V> initializer,
                                        final Merger<? super K, V> sessionMerger,
                                        final Materialized<K, V, SessionStore<Bytes, byte[]>> materialized);

    KTable<Windowed<K>, V> aggregate(final Initializer<V> initializer,
                                     final Merger<? super K, V> sessionMerger);

    KTable<Windowed<K>, Vout> aggregate(final Initializer<V> initializer,
                                                                  final Named named,
                                        final Merger<? super K, V> sessionMerger,
                                        final Materialized<K, V, SessionStore<Bytes, byte[]>> materialized);

    KTable<Windowed<K>, V> aggregate(final Initializer<V> initializer,
                                                                  final Named named,
                                     final Merger<? super K, V> sessionMerger);

}
```

```
/**
 * {@code TimeWindowedCogroupKStream} is an abstraction of a <i>windowed</i> record stream of {@link org.apache.
kafka.streams.KeyValue} pairs.
 * It is an intermediate representation of a {@link KGroupedStream} in order to apply a windowed aggregation
operation on the original
 * {@link KGroupedStream} records.
 * <p>
 * It is an intermediate representation after a grouping, cogrouping and windowing of a {@link KStream} before
an aggregation is applied to the
 * new (partitioned) windows resulting in a windowed {@link KTable}
 * (a <emph>windowed</emph> {@code KTable} is a {@link KTable} with key type {@link Windowed Windowed<K>}.
 * <p>
 * The specified {@code windows} define either hopping time windows that can be overlapping or tumbling (c.f.
 * {@link TimeWindows}) or they define landmark windows (c.f. {@link UnlimitedWindows}).
 * The result is written into a local windowed {@link org.apache.kafka.streams.state.KeyValueStore} (which is
basically an ever-updating
 * materialized view) that can be queried using the name provided in the {@link Materialized} instance.
 *
 * New events are added to windows until their grace period ends (see {@link TimeWindows#grace(Duration)}).
 *
 * Furthermore, updates to the store are sent downstream into a windowed {@link KTable} changelog stream, where
 * "windowed" implies that the {@link KTable} key is a combined key of the original record key and a window ID.

 * A {@code WindowedKStream} must be obtained from a {@link KGroupedStream} via {@link KGroupedStream#windowedBy
(Windows)} .
 *
 * @param <K> Type of keys
 * @param <T> Type of values
 * @see KStream
 * @see KGroupedStream
 * @see CogroupedKStream
 */

public interface TimeWindowedCogroupedKStream<K, V> {

    KTable<Windowed<K>, V> aggregate(final Initializer<V> initializer,
                                     final Materialized<K, V, WindowStore<Bytes, byte[]>> materialized);

    KTable<Windowed<K>, V> aggregate(final Initializer<V> initializer);

    KTable<Windowed<K>, V> aggregate(final Initializer<V> initializer,
                                                                    final Named named,
                                     final Materialized<K, V, WindowStore<Bytes, byte[]>> materialized);

    KTable<Windowed<K>, V> aggregate(final Initializer<V> initializer,
                                                                    final Named named);

}
```

Proposed Changes

1. Construct the above Public Interfaces.
2. Create an internal.KCogroupedStreamImpl that will keep track of the KeyValueSerde, AggValueSerde, Initializer, and Pairs of (KGroupedStream, Aggregator).
3. Create an internal TimeWindowedKCogroupedStreamImpl and SessionWindowedKCogroupedStreamImpl that will complete the windowed aggregations.
4. make the KStreamAggProcessorSupplier for each KGroupedStream. Additionally ensure all sources are copartitioned, processors have access to the state store.
5. create a cogroupedStreamAggregateBuilder to take the groupStream and aggregate pairs apply the aggregations

Compatibility, Deprecation, and Migration Plan

- Users must upgrade to new version if they want to use this functionality.

Rejected Alternatives

- Introducing Named in the cogroup

Rational for edits to the KIP

Due to changes to the project since KIP #150 was written there are a few items that may need to be updated.

First item that changed is the adoption of Materialized parameter.

The second item is the WindowedBy. How the KIP currently handles windowing is that it overloads the aggregate function to taking a Window object and an initializer. Currently the practice to window grouped streams is to call windowedBy and receive a windowed stream object. The same interface for a windowed stream made from a grouped stream will not work for Cogrouped streams as grouped streams SO we have to make new ones for CogroupedWindows. This is because when a cogroup is created and aggregated the aggregator is associated with each grouped stream then stored in the cogroup. When aggregating what is needed is an initializer but not an aggregator.