KIP-99: Add Global Tables to Kafka Streams

- Motivation
  - Example
- Public Interfaces
  - GlobalKTable
  - KStream
  - KTable
  - KStreamBuilder
  - TopologyBuilder
- Compatibility, Deprecation, and Migration Plan
- Test Plan
- Rejected Alternatives and Future Work

Status

Current state: *Accepted*

Discussion thread: *here*

JIRA: *KAFKA-4490*

Released: 0.10.2

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

Motivation

In streams applications it is common to chain multiple joins together in-order to enrich a large dataset with some, often, smaller side-data. For example: enriching an Order with customer data. This also maps into the relational world where it is common to join some fact tables with some dimensional data. The facts are often large, with frequent updates and/or new data arrivals, i.e., a Purchases stream, whereas the dimensional data is generally much smaller with a potentially less frequent update rate, i.e., Products.

In Kafka Streams we can join the fact and dimension tables today, by partitioning both the fact stream & dimension table on the same key. This works ok if you only ever need to join the fact with one dimension, however joining to multiple dimensions requires repartitioning of the fact stream once for each dimension. This is quite expensive in terms of network I/O, disk usage, and processing latency, i.e., every re-partitioning operation results in duplicating the input stream to another topic. This requires consuming the input topic, re-keying and then writing to an intermediate topic, and then subsequently consuming from the intermediate topic. Further, this requires that the dimension tables be co-partitioned with the fact table, i.e. they are all required to have the same number of partitions. This can obviously result in over-the-top partitioning for small dimension data sets.

A convenient solution, where the dimensions are of a manageable size, is to replicate the dimension data, in their entirety, onto each Kafka Streams node. This then allows joins to be performed without the prerequisite that the fact stream be partitioned by the dimension table's key, i.e., all dimension data is available to all partitions of the fact table, so we could perform non-key based joins from a fact to many dimensions regardless of their partitioning key, all at low cost.

Example

We have a bespoke website for selling some custom made goods. When a purchase is made we want to do some data enrichment to immediately produce, and, send detailed invoices to customers. We have three inputs:

Purchase(account_id, product_id, …)

Product(name, description, price, …)

Customer(name, address, …)

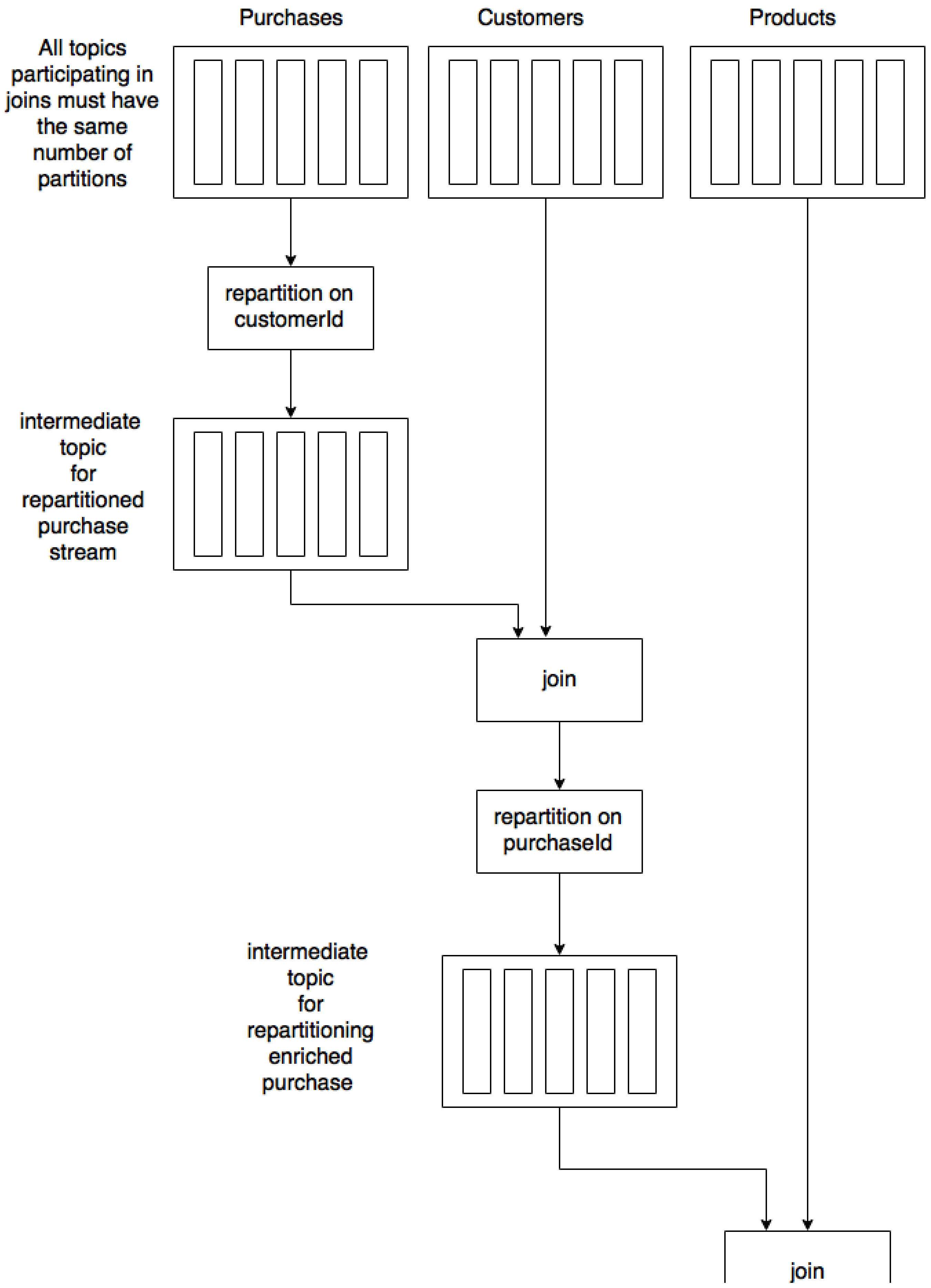
For this example, Purchase is our fact, Product and Customer are our dimensions.

Currently in Kafka Streams to enrich a Purchase with both Product and Customer information, we'd do something like:

```
final KStream<Long, Purchase> purchases = builder.stream("purchase");
final KTable<Long, Customer> customers = builder.table("customer", "customer-store");
final KTable<Long, Product> products = builder.table("product", "product-store");


// re-key purchases stream on customerId
purchases.map((key, purchase) -> KeyValue.pair(purchase.customerId(), purchase))
        // join to customers. This will create an intermediate topic to repartition
        // the purchases stream based on the customerId
        .leftJoin(customers, EnrichedPurchase::new)
        // re-key enrichedPurchase based on productId
        .map((key, enrichedPurchase) ->
                   KeyValue.pair(enrichedPurchase.productId(), enrichedPurchase))
        // join to products. This will create an intermediate topic to repartition
        // the previous intermediate topic based on productId
        .leftJoin(products, EnrichedPurchase::withProduct);
```

As described in the code above and shown in the diagram below, there are multiple repartitioning phases and intermediate topics. Also, we need to co-partition all of the topics involved in the joins.

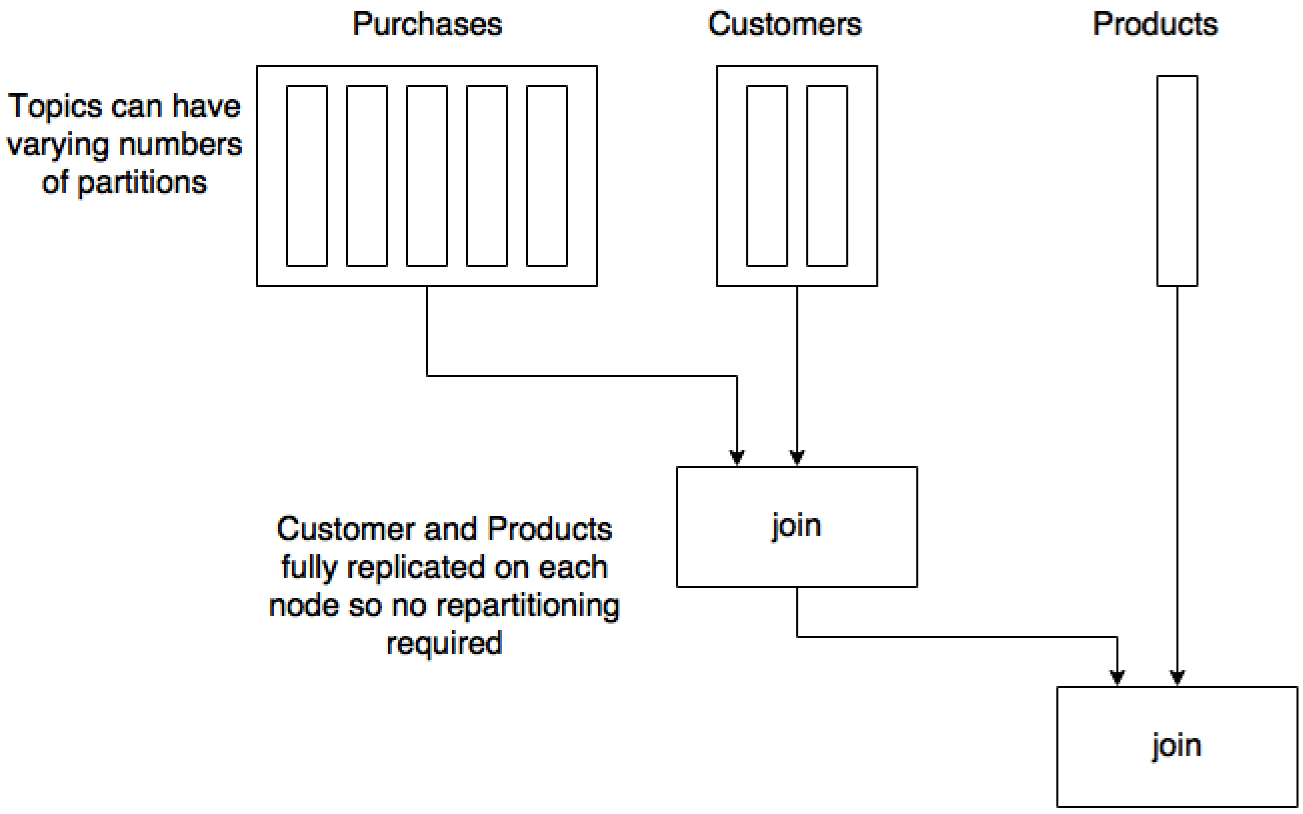

Purchases
Customers
Products
All topics participating in joins must have the same number of partitions
repartition on customerId
intermediate topic for repartitioned purchase stream
join
repartition on purchaseId
intermediate topic for repartitioning enriched purchase
join

With the introduction of global tables to Kafka Streams we eliminate the intermediate topics used for repartitioning.

```
final KStream<Long, Purchase> purchases = builder.stream("purchase");
final GlobalKTable<Long, Customer> customers = builder.globalTable("customer", "customer-store");
final GlobalKTable<Long, Product> products = builder.globalTable("product", "product-store");


// join to the customers table by providing a mapping to the customerId
purchases.leftJoin(customers,
                            ((key, purchase) -> KeyValue.pair(purchase.customerId(), purchase),
                      EnrichedPurchase::new)
          // join to the products table by providing a mapping to the productId
          .leftJoin(products,
                    KeyValue.pair(enrichedPurchase.productId(), enrichedPurchase),
                    EnrichedPurchase::withProduct);
```

Now we've eliminated the repartitioning, intermediate topics, and the need to co-partition all the input topics. So the flow now looks like:


Purchases
Customers
Products
Topics can have varying numbers of partitions
join
Customer and Products fully replicated on each node so no repartitioning required
join

Public Interfaces

GlobalKTable

A new interface to represent the *GlobalKTable*. This interface is deliberately restrictive as its primary purpose is to enable non-key joins without having to do any re-partitioning.

```
// Represents a Table that is fully replicated on each KafkaStreams instance
public interface GlobalKTable<K, V> {
}
```

KStream

Add overloaded methods for joining with *GlobalKTable*

```
/**
 * perform a left join with a GlobalKTable using the provided KeyValueMapper
 * to map from the (key, value) of the KStream to the key of the GlobalKTable
 */
<K1, V1, R> KStream<K, R> leftJoin(final GlobalKTable<K1, V1> replicatedTable,
                                                   final KeyValueMapper<K, V, K1> keyValueMapper,
                                                    final ValueJoiner<V, V1, R> valueJoiner);

/**
 * perform a join with a GlobalKTable using the provided KeyValueMapper
 * to map from the (key, value) of the KStream to the key of the GlobalKTable
 */
<K1, V1, V2> KStream<K, V2> join(final GlobalKTable<K1, V1> table,
                                                     final KeyValueMapper<K, V, K1> keyValueMapper,
                                                      final ValueJoiner<V, V1, V2> joiner);
```

KTable

Add overloaded methods for joining with *GlobalKTable*

```
/**
 * perform a join with a GlobalKTable using the provided KeyValueMapper
 * to map from the (key, value) of the KTable to the key of the GlobalKTable
 */
<K1, V1, R> KTable<K, R> join(final GlobalKTable<K1, V1> globalTable,
                              final KeyValueMapper<K, V, K1> keyMapper,
                              final ValueJoiner<V, V1, R> joiner);

/**
 * perform a left join with a GlobalKTable using the provided KeyValueMapper
 * to map from the (key, value) of the KTable to the key of the GlobalKTable
 */
<K1, V1, R> KTable<K, R> leftJoin(final GlobalKTable<K1, V1> globalTable,
                                  final KeyValueMapper<K, V, K1> keyMapper,
                                  final ValueJoiner<V, V1, R> joiner);
```

KStreamBuilder

```
/**
 * Add a GlobalKTable to the topology using the provided Serdes
 */
public <K, V> GlobalKTable<K, V> globalTable(final Serde<K> keySerde,
                                                                   final Serde<V> valSerde,
                                                                   final String topic,
                                             final String storeName)

/**
 * Add a GlobalKTable to the topology using default Serdes
 */
public <K, V> GlobalKTable<K, V> globalTable(final String topic,
                                             final String storeName)
```

TopologyBuilder

```
// add a Global Store that is backed by a source topic to the TopologyBuilder
public synchronized TopologyBuilder addGlobalStore(final StateStore
store,
                                                   final String sourceName,
                                                                                                      final
Deserializer keyDeserializer,
                                                                                                      final
Deserializer valueDeserializer,
                                                                                                      final
String topic,
                                                                                                      final
String processorName,
                                                                                                  final
ProcessorSupplier stateUpdateSupplier)


// All Global State Stores will be part of a single ProcessorTopology
// this provides an easy way to build that topology
public synchronized ProcessorTopology buildGlobalStateTopology();

// Retrieve the Global State Stores from the builder.
public synchronized Map<String, StateStore> builder.globalStateStores()
```

Proposed Changes

We will introduce a new type into the Streams DSL - *GlobalKTable*. The *GlobalKTable* will be fully replicated once per KafkaStreams instance. That is, each KafkaStreams instance will consume all partitions of the corresponding topic. At application start time a *GlobalKTable* will be bootstrapped to the current high watermark of the underlying partitions. Only once the bootstrapping has finished will normal stream processing begin.

The GlobalKTable will be populated and updated by a dedicated thread. This thread will be responsible for keeping all GlobalKTables up-to-date. In order to prevent stalling the updates, the thread will attempt to retry on any transient exceptions. Having a dedicated, monitored thread means we don't have to invent any custom cross-thread synchronization and rebalancing in the event that a StreamThread dies.

On close of the GlobalKTable, we will write out a checkpoint file to the global table's state directory, located under the *global* sub-directory of *StreamsConfi g.STATE_DIR_CONFIG, i*.e, /state.dir/appId/global, containing the current offset for each partition in the GlobalKTable. The checkpoint file will be used to recover during restarts. On restart we will first check for the existence of the checkpoint file, if it exists we will read in the current offsets for each partition and seek to those offsets and start consuming. In the event the checkpoint file doesn't exist, we will restore from the earliest offset.

Streams application developers will create, and use, GlobalKTables via the DSL. For example:

```
final GlobalKTable products = builder.globalTable(“product”, “products-store”);
final KStream orders = builder.stream(“orders”);
order.leftJoin(products,
              (key, value) -> value.productId(),
              (key, order, product) ->
                  new EnrichedOrder(order, product));
```

The GlobalKTable will only be used for doing lookups. That is, data arriving in the GlobalKTable will not trigger the join. Triggering the join would require *for warding* data to *processors* that are running on other threads, KafkaStreams currently assumes all nodes in a *ProcessorTopology* are running in a single thread. In using *GlobalKTables* for lookups only we can avoid any cross-thread *processor* synchronization.

The joins between *GlobalKTables* will not be materialized to a Physical *StateStore,* instead they are just a view on top of the joined *GlobalKTables.* The joi ns are resolved on demand whenever a lookup is performed on it, i.e., via an Interactive Query or via a *KStream* or *KTable join*. This means that we don't need to create yet another copy of the data on disk, and replicate it to another change-log topic. It is also reduces the memory foot print as we don't need to create another *RocksDBStore.*

Compatibility, Deprecation, and Migration Plan

- None - this is a new feature and doesn’t impact any existing usages.

Test Plan

- Unit tests to validate that all the individual components work as expected.
- Integration and/or System tests to ensure that the feature works correctly end-to-end.

Rejected Alternatives and Future Work

- Replicating per task: Each StreamTask would have its own complete replica of the table. Though this would provide the ability to do best-effort time synchronization it would be too costly in terms of resource usage.

- Replicating per thread: Doesn’t provide any benefit beyond per instance, but adds additional replication overhead
- Introduce a broadcastJoin() API that uses the existing KTable interface and automatically converts it, internally, to a global table. The feeling is that this would muddy the API, i.e. it is better to be explicit with the type.
- Supporting GlobalKTable to GlobalKTable joins: In order to fully support bi-directional joins would require KeyValueMappers , and either materializing the join in a Physical *StateStore,* i.e., RocksDB, or by having yet another interface to Map from the key of the join table to the keys of both input tables. We decided that for now we will not support *GlobalKTable/GlobalKTable* joins, but revisit in the future should the need arise.