KIP-221: Enhance DSL with Connecting Topic Creation and Repartition Hint

- Status
- Motivation
- Public interfaces
- Proposed Changes
- Backward Compatibility
- Rejected Alternatives
  - Repartition "hint" in groupBy operations
  - Why not use Produced operation for specifying number of partitions?

Status

Current state: *Accepted*

Voting thread: https://www.mail-archive.com/dev@kafka.apache.org/msg99680.html

Discussion thread: https://www.mail-archive.com/dev@kafka.apache.org/msg99111.html

JIRA: KAFKA-6037 - Getting issue details... STATUS KAFKA-8611 - Getting issue details... STATUS KAFKA-10003 - Getting issue details... STATUS

PR: https://github.com/apache/kafka/pull/7170

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

Motivation

Today the downstream sub-topology's parallelism (aka the number of tasks) are purely dependent on the upstream sub-topology's parallelism, which ultimately depends on the source topic's num.partitions. However this does not work perfectly with dynamic scaling scenarios. By delegating the stream topology power to create repartition topic with customized number of partitions gives user more flexibility. Also, at this point, when using DSL in Kafka Streams, data re-partition happens only when key-changing operation is followed by stateful operation. On the other hand, in DSL, stateful computation can happen using *transform()* operation as well. Problem with this approach is that, even if any upstream operation was key-changing before calling *transform()*, no auto-repartition is triggered. If repartitioning is required, a call to *through(String)* should be performed before *transform()*. With the current implementation, burden of managing and creating the topic falls on user and introduces extra complexity of managing Kafka Streams application.

Public interfaces

With current API contract in Kafka Streams DSL we have possibility to control and specify *producer level* configuration using **Produced<K, V>** class, which we can pass to various operators like KStream#to, KStream#through etc. However, in current Kafka Streams DSL, there's no possibility to specify *topic level* configurations for different operators that potentially may create *internal topics*. In order to give the user possibility to control parallelism of sub-topologies and potentially make internal topic configuration more flexible, we shall add following configuration class.

Repartitioned.java

```
/**
 * This class is used to provide the optional parameters for internal repartitioned topics when using:
 * - {@link KStream#repartition(Repartitioned)}
 * - {@link KStream#repartition(KeyValueMapper, Repartitioned)}
 *
 * @param <K> key type
 * @param <V> value type
 */
public class Repartitioned<K, V> implements NamedOperation<Repartitioned<K, V>> {

        protected final String name;

        protected final Serde<K> keySerde;

        protected final Serde<V> valueSerde;
```

```
        protected final Integer numberOfPartitions;

        protected final StreamPartitioner<? super K, ? super V> partitioner;

        private Repartitioned(String name,
                              Serde<K> keySerde,
                              Serde<V> valueSerde,
                              Integer numberOfPartitions,
                              StreamPartitioner<? super K, ? super V> partitioner) {
                this.name = name;
                this.keySerde = keySerde;
                this.valueSerde = valueSerde;
                this.numberOfPartitions = numberOfPartitions;
                this.partitioner = partitioner;
        }

        /**
         * Create a {@link Repartitioned} instance with the provided name used as part of the repartition topic
if required.
         *
         * @param name the name used as a processor named and part of the repartition topic name if required.
         * @param <K>  key type
         * @param <V>  value type
         * @return A new {@link Repartitioned} instance configured with processor name and repartition topic
name
         * @see KStream#repartition(Repartitioned)
         * @see KStream#repartition(KeyValueMapper, Repartitioned)
         */
        public static <K, V> Repartitioned<K, V> as(final String name) {
                return new Repartitioned<>(name, null, null, null, null);
        }

        /**
         * Create a {@link Repartitioned} instance with provided key serde and value serde.
         *
         * @param keySerde   Serde to use for serializing the key
         * @param valueSerde Serde to use for serializing the value
         * @param <K>        key type
         * @param <V>        value type
         * @return A new {@link Repartitioned} instance configured with key serde and value serde
         * @see KStream#repartition(Repartitioned)
         * @see KStream#repartition(KeyValueMapper, Repartitioned)
         * @see KStream#groupByKey(Repartitioned)
         * @see KStream#groupBy(KeyValueMapper, Repartitioned)
         */
        public static <K, V> Repartitioned<K, V> with(final Serde<K> keySerde,
                                                      final Serde<V> valueSerde) {
                return new Repartitioned<>(null, keySerde, valueSerde, null, null);
        }

        /**
         * Create a {@link Repartitioned} instance with provided partitioner.
         *
         * @param partitioner the function used to determine how records are distributed among partitions of
the topic,
         *                    if not specified and the key serde provides a {@link WindowedSerializer} for the
key
         *                    {@link WindowedStreamPartitioner} will be used—otherwise {@link
DefaultPartitioner} will be used
         * @param <K>         key type
         * @param <V>         value type
         * @return A new {@link Repartitioned} instance configured with partitioner
         * @see KStream#repartition(Repartitioned)
         * @see KStream#repartition(KeyValueMapper, Repartitioned)
         */
        public static <K, V> Repartitioned<K, V> streamPartitioner(final StreamPartitioner<? super K, ? super
V> partitioner) {
                return new Repartitioned<>(null, null, null, null, partitioner);
        }

        /**
```

```
         * Create a {@link Repartitioned} instance with provided number of partitions for repartition topic if
required.
         *
         * @param numberOfPartitions number of partitions used when creating repartition topic if required
         * @param <K>                key type
         * @param <V>                value type
         * @return A new {@link Repartitioned} instance configured number of partitions
         * @see KStream#repartition(Repartitioned)
         * @see KStream#repartition(KeyValueMapper, Repartitioned)
         */
        public static <K, V> Repartitioned<K, V> numberOfPartitions(final int numberOfPartitions) {
                return new Repartitioned<>(null, null, null, numberOfPartitions, null);
        }

        /**
         * Create a new instance of {@link Repartitioned} with the provided name used as part of repartition
topic and processor name.
         * Note that Kafka Streams creates repartition topic only if required.
         *
         * @param name the name used for the processor name and as part of the repartition topic name if
required
         * @return a new {@link Repartitioned} instance configured with the name
         */
        @Override
        public Repartitioned<K, V> withName(final String name) {
                return new Repartitioned<>(name, keySerde, valueSerde, numberOfPartitions, partitioner);
        }

        /**
         * Create a new instance of {@link Repartitioned} with the provided number of partitions for
repartition topic.
         * Note that Kafka Streams creates repartition topic only if required.
         *
         * @param numberOfPartitions the name used for the processor name and as part of the repartition topic
name if required
         * @return a new {@link Repartitioned} instance configured with the number of partitions
         */
        public Repartitioned<K, V> withNumberOfPartitions(final int numberOfPartitions) {
                return new Repartitioned<>(name, keySerde, valueSerde, numberOfPartitions, partitioner);
        }

        /**
         * Create a new instance of {@link Repartitioned} with the provided key serde.
         *
         * @param keySerde Serde to use for serializing the key
         * @return a new {@link Repartitioned} instance configured with the key serde
         */
        public Repartitioned<K, V> withKeySerde(final Serde<K> keySerde) {
                return new Repartitioned<>(name, keySerde, valueSerde, numberOfPartitions, partitioner);
        }

        /**
         * Create a new instance of {@link Repartitioned} with the provided value serde.
         *
         * @param valueSerde Serde to use for serializing the value
         * @return a new {@link Repartitioned} instance configured with the value serde
         */
        public Repartitioned<K, V> withValueSerde(final Serde<V> valueSerde) {
                return new Repartitioned<>(name, keySerde, valueSerde, numberOfPartitions, partitioner);
        }

        /**
         * Create a new instance of {@link Repartitioned} with the provided partitioner.
         *
         * @param partitioner the function used to determine how records are distributed among partitions of
the topic,
         *                    if not specified and the key serde provides a {@link WindowedSerializer} for the
key
         *                    {@link WindowedStreamPartitioner} will be used—otherwise {@link
DefaultPartitioner} wil be used
         * @return a new {@link Repartitioned} instance configured with provided partitioner
```

```
         */
        public Repartitioned<K, V> withStreamPartitioner(final StreamPartitioner<? super K, ? super V>
partitioner) {
                return new Repartitioned<>(name, keySerde, valueSerde, numberOfPartitions, partitioner);
        }
}
```

New **KStream#repartition** operations shall be introduced in order to give the user control over parallelism for sub-topologies. Additionally, we deprecate **KStream#through** in favor of the new **#repartition** methods.

KStream.java

```
public interface KStream<K, V> {

    @Deprecated
    KStream<K, V> through(final String topic);

    @Deprecated
    KStream<K, V> through(final String topic, final Produced<K, V> produced);

        /**
         * Materialize this stream to a auto-generated repartition topic and creates a new {@code KStream}
         * from the auto-generated topic using default serializers, deserializers, producer's {@link
DefaultPartitioner}.
         * Number of partitions is inherited from the source topic.
         *
         * @return a {@code KStream} that contains the exact same (and potentially repartitioned) records as
this {@code KStream}
         * @see #repartition(Repartitioned)
         * @see #repartition(KeyValueMapper, Repartitioned)
         */
        KStream<K, V> repartition();

        /**
         * Materialize this stream to a auto-generated repartition topic and creates a new {@code KStream}
         * from the auto-generated topic using {@link Serde key serde}, {@link Serde value serde}, {@link
StreamPartitioner},
         * number of partitions and topic name part as defined by {@link Repartitioned}.
         *
         * @param repartitioned the {@link Repartitioned} instance used to specify {@link org.apache.kafka.
common.serialization.Serdes},
         *                      {@link StreamPartitioner} which determines how records are distributed among
partitions of the topic,
         *                      part of the topic name and number of partitions for a repartition topic, if
repartitioning is required.
         * @return a {@code KStream} that contains the exact same (and potentially repartitioned) records as
this {@code KStream}
         * @see #repartition()
         * @see #repartition(KeyValueMapper, Repartitioned)
         */
        KStream<K, V> repartition(final Repartitioned<K, V> repartitioned);
}
```

Correspondingly, the Scala API will be updated including an implicit conversation from key/value Serdes to a `Repartitioned` instance.

```
class KStream[K, V](val inner: KStreamJ[K, V]) {
  @deprecated
  def through(topic: String)(implicit produced: Produced[K, V])

  def repartition(implicit repartitioned: Repartitioned[K, V])
}

object Repartitioned {
  def `with`[K, V](implicit keySerde: Serde[K], valueSerde: Serde[V])

  def `with`[K, V](name: String)(implicit keySerde: Serde[K], valueSerde: Serde[V])

  def `with`[K, V](partitioner: StreamPartitioner[K, V])(implicit keySerde: Serde[K], valueSerde: Serde[V])

  def `with`[K, V](numberOfPartitions: Int)(implicit keySerde: Serde[K], valueSerde: Serde[V])
}

object ImplicitConversions {
  implicit def repartitionedFromSerde[K, V](implicit keySerde: Serde[K], valueSerde: Serde[V])
}
```

Proposed Changes

- For **KStream#repartition(Repartitioned)** operation, Kafka Streams application will first issue the topic lookup request and check whether the target topic is already up and running. If **Repartitioned** is configured with number of partitions, in addition, Kafka Streams application will make sure that number of partitions in the topic match with the configured value. If not, application will thrown an error and fail during startup.
- For **KStream#repartition()** operation, use upstream topic partition size as the new topic number of partitions. Topic name will be generated based on the generated processor node name.

Backward Compatibility

This is a pure KStream library change that shouldn't affect previously setup applications. Since we introduce new KStream#groupBy operations, existing ones shouldn't be affected by this change. Using using KStream#through can either switch to the new #repartition method (which should be the common use case) or rewrite their code to use #to() and StreamsBuilder#stream() (note that #through() is just syntactic sugar for those two calls anyway).

Rejected Alternatives

Repartition "hint" in groupBy operations

In the mailing thread discussion, concern was raised that adding number of partitions configuration option to group by operations, such as `KStream#groupByKey(Repartitioned)` may not be the best option. Main argument against it is that whenever user specified number of partitions for internal, repartition topics, he/she really cares that those configuration will be applied. Case with group by is that, repartitioning will not happen at all if key changing operation isn't performed, therefore number of partitions configuration specified by the user will never kick-in. Alternatively, if user cares about manual repartitioning, one may do following in order to scale up/down sub topologies:

```
builder.stream("topic")
       .repartition((key, value) -> value.newKey(), Repartitioned.withNumberOfPartitions(5))
           .groupByKey()
           .count();
```

Follow-up ticket for this feature can be found here: KAFKA-9197 - Getting issue details... STATUS

Rejected KSream interface changes:

KStream.java

```
public interface KStream<K, V> {
        /**
         * Re-groups the records of this {@code KTable} on a new key that is selected using the provided {@link
KeyValueMapper}
         * and {@link Serde}s as specified by {@link Repartitioned}.
         * Re-grouping a {@code KTable} is required before an aggregation operator can be applied to the data
         * (cf. {@link KGroupedTable}).
         * The {@link KeyValueMapper} selects a new key (which may or may not be of the same type) while
preserving the
         * original values.
         * If the new record key is {@code null} the record will not be included in the resulting {@link
KGroupedStream}.
         * <p>
         * Because a new key is selected, an internal repartitioning topic will be created in Kafka.
         * This topic will be named "${applicationId}-<name>-repartition", where "applicationId" is user-
specified in
         * {@link  StreamsConfig} via parameter {@link StreamsConfig#APPLICATION_ID_CONFIG
APPLICATION_ID_CONFIG},
     * "<name>" is either provided via {@link org.apache.kafka.streams.kstream.Repartitioned#as(String)}
     * or generated internally, and "-repartition" is a fixed suffix.
     * If number of partitions is provided via {@link org.apache.kafka.streams.kstream.
Repartitioned#withNumberOfPartitions(int)}
     * repartition topic will be generated with the specified number of partitions.
     * If not, number of partitions will be inherited from the source topic.
         * <p>
         * You can retrieve all generated internal topic names via {@link Topology#describe()}.
         *
         * <p>
         * All data of this {@code KTable} will be redistributed through the repartitioning topic by writing
all update
         * records to and rereading all updated records from it, such that the resulting {@link KGroupedTable}
is partitioned
         * on the new key.
         *
         * @param selector      a {@link KeyValueMapper} that computes a new key for grouping
         * @param repartitioned the {@link Repartitioned} instance used to specify {@link org.apache.kafka.
common.serialization.Serdes},
         *                      {@link org.apache.kafka.streams.processor.StreamPartitioner} which determines
how records are distributed among partitions of the topic,
         *                      part of the topic name and number of partitions for a repartition topic, if
repartitioning is required.
         * @param <KR>          the key type of the result {@link KGroupedStream}
         * @return a {@link KGroupedStream} that contains the grouped records of the original {@code KStream}
         */
        <KR> KGroupedStream<KR, V> groupBy(final KeyValueMapper<? super K, ? super V, KR> selector,
                                           final Repartitioned<KR, V> repartitioned);

        /**
         * Re-groups the records by their current key into a {@link KGroupedStream} while preserving the
original values
     * and using the serializers as defined by {@link Repartitioned}.
         * If the new record key is {@code null} the record will not be included in the resulting {@link
KGroupedStream}.
         * <p>
         * An internal repartitioning topic may be created in Kafka, if number of partitions provided via
     * {@link org.apache.kafka.streams.kstream.Repartitioned#withNumberOfPartitions(int)} is different compared
to source topic of this {@link KTable},
         * If repartitioned topic is created, it will be named "${applicationId}-<name>-repartition", where
"applicationId" is user-specified in
         * {@link StreamsConfig} via parameter {@link StreamsConfig#APPLICATION_ID_CONFIG
APPLICATION_ID_CONFIG}, "<name>"
     * is either provided via {@link org.apache.kafka.streams.kstream.Repartitioned#as(String)} or generated
internally, and "-repartition" is a fixed suffix.
         * <p>
         * You can retrieve all generated internal topic names via {@link Topology#describe()}.
         *
         * @param repartitioned the {@link Repartitioned} instance used to specify {@link org.apache.kafka.
common.serialization.Serdes},
         *                      {@link org.apache.kafka.streams.processor.StreamPartitioner} which determines
how records are distributed among partitions of the topic,
         *                      part of the topic name and number of partitions for a repartition topic, if
```

```
repartitioning is required.
         * @return a {@link KGroupedStream} that contains the grouped records of the original {@code KTable}
         */
        KGroupedStream<K, V> groupBy(final Repartitioned<K, V> repartitioned);
}
```

Why not use Produced operation for specifying number of partitions?

There're two main reasons why not to use Produced class for specifying number of partitions.

1) If we enhance Produced class with this configuration, this will also affect **KStream#to**. Since **KStream#to** is the final sink of the topology, it seems to be reasonable assumption that user needs to manually create sink topic in advance. And in that case, having num of partitions configuration doesn't make much sense.

2) Looking at KStream interface, seems like Produced class is for operations that work with non-internal (e.g topics that are created and managed internally by Kafka Streams) topics and this contract will break if we gonna enhance it with number of partitions configuration. Current API contract is very clear in that respect - every operation (KStream#to, KStream#through) that use Produced class work with topics that are explicitly managed by the end user.