KIP-182: Reduce Streams DSL overloads and allow easier use of custom storage engines

Status

Current state: *Adopted (1.0)*

Discussion thread: [DISCUS] KIP-182: Reduce Streams DSL overloads and allow easier use of custom storage engines

JIRA: *KAFKA-5651*

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

Motivation

As the Kafka Streams DSL has evolved, some of the APIs have become very overload heavy. For example, we have 8 different overloads for KStream#print. As we add more overloads it becomes harder for a developer using a modern IDE to discover the interfaces hence interrupting the flow and becoming an API usability issue.

Further, we'd like to provide users with a way to override certain *StateStore* features on a per operator basis, for example, enable caching or logging, for some but not all *StateStore*s. Without a change in approach to the DSL this would add yet more overloads for every operation. Additionally, it should be simple to use the KafkaStreams Caching and Logging wrappers with custom *StateStores*.

Before we go and add many more overloaded methods it is worth while exploring other options to see if we can provide a more concise and intuitive API.

Public Interfaces

New methods added to existing interfaces:

KStream

```
void print(final Printed<K, V> printed);

KStream<K, V> through(final String topic, final Produced<K, V> partitioned);

void to(final String topic, final Produced<V, V> partitioned);

KGroupedStream<K, V> groupByKey(final Serialized<K, V> serialized);

<KR> KGroupedStream<KR, V> groupBy(final KeyValueMapper<? super K, ? super V, KR> selector, Serialized<KR, V>
serialized);

<VO, VR> KStream<K, VR> join(final KStream<K, VO> other, final ValueJoiner<? super V, ? super VO, ? extends VR>
joiner, final JoinWindows windows, final Joined<K, V, VO> options);

<VT, VR> KStream<K, VR> join(final KTable<K, VT> other, final ValueJoiner<? super V, ? super VT, ? extends VR>
joiner, final Joined<K, V, VT> options);

<VO, VR> KStream<K, VR> leftJoin(final KStream<K, VO> other, final ValueJoiner<? super V, ? super VO, ? extends
VR> joiner, final JoinWindows windows, final Joined<K, V, VO> options);

<VT, VR> KStream<K, VR> leftJoin(final KTable<K, VT> other, final ValueJoiner<? super V, ? super VT, ? extends
VR> joiner, final Joined<K, V, VT> options);

<VO, VR> KStream<K, VR> outerJoin(final KStream<K, VO> other, final ValueJoiner<? super V, ? super VO, ?
extends VR> joiner, final JoinWindows windows, final Joined<K, V, VO> options);
```

KTable

```
<KR, VR> KGroupedTable<KR, VR> groupBy(final KeyValueMapper<? super K, ? super V, KeyValue<KR, VR>> selector,
Serialized<KR, VR> serialized);

KTable<K, V> filter(final Predicate<? super K, ? super V> predicate, final Materialized<K, V,
KeyValueStore<Bytes, byte[]>> materialized);

KTable<K, V> filterNot(final Predicate<? super K, ? super V> predicate, final Materialized<K, V,
KeyValueStore<Bytes[], byte[]>> materialized);

<VR> KTable<K, VR> mapValues(final ValueMapper<? super V, ? extends VR> mapper, final Materialized<K, V,
KeyValueStore<Bytes[], byte[]>> materialized);

<VO, VR> KTable<K, VR> join(final KTable<K, VO> other,
                            final ValueJoiner<? super V, ? super VO, ? extends VR> joiner,
                            final Materialized<K, VR, KeyValueStore<Bytes, byte[]>> materialized);

<VO, VR> KTable<K, VR> leftJoin(final KTable<K, VO> other,
                                final ValueJoiner<? super V, ? super VO, ? extends VR> joiner,
                                final Materialized<K, VR, KeyValueStore<Bytes, byte[]>> materialized);

<VO, VR> KTable<K, VR> outerJoin(final KTable<K, VO> other,
                                 final ValueJoiner<? super V, ? super VO, ? extends VR> joiner,
                                 final Materialized<K, VR, KeyValueStore<Bytes, byte[]>> materialized);
```

We add some new helper methods to *Stores* so people can conveniently and quickly create basic *StateStoreSuppliers* for use in the DSL or PAPI. We will also deprecate the existing *Stores.create(...)*

Stores

```
public static <K, V> KeyValueBytesStoreSupplier persistentKeyValueStore(final String name)

public static <K, V> KeyValueBytesStoreSupplier inMemoryKeyValueStore(final String name)

public static <K, V> KeyValueBytesStoreSupplier lruMap(final String name)

public static <K, V> WindowBytesStoreSupplier persistentWindowStore(final String name,

final long retentionPeriod,

final int numSegments,

final long windowSize,

final boolean retainDuplicates)
public static <K, V> SessionBytesStoreSupplier persistentSessionStore(final String name, final long
retentionPeriod)
/**
 *  The following methods are for use with the PAPI. They allow building of StateStores that can be wrapped with
 *  caching, logging, and any other convenient wrappers provided by the KafkaStreams library
 */
public <K, V> StateStoreBuilder<WindowStore<K, V>> windowStoreBuilder(final WindowBytesStoreSupplier supplier,

final Serde<K> keySerde,

final Serde<V> valueSerde)

public <K, V> StateStoreBuilder<KeyValueStore<K, V>> keyValueStoreBuilder(final KeyValueBytesStoreSupplier
supplier,

final Serde<K> keySerde,

final Serde<V> valueSerde)

public <K, V> StateStoreBuilder<SessionStore<K, V>> sessionStoreBuilder(final SessionBytesStoreSupplier
supplier,

final Serde<K> keySerde,

final Serde<V> valueSerde)
```

Topology

```
public synchronized <K, V> Topology addGlobalStore(final StateStoreBuilder storeSupplier,
                                                   final String sourceName,
                                                   final TimestampExtractor timestampExtractor,
                                                   final Deserializer keyDeserializer,
                                                   final Deserializer valueDeserializer,
                                                   final String topic,
                                                   final String processorName,
                                                   final ProcessorSupplier stateUpdateSupplier)

public synchronized final Topology addStateStore(final StateStoreBuilder supplier, final String...
processorNames)
```

KGroupedStream

```
<W extends Window> TimeWindowedKStream<K, V> windowedBy(Windows<W> timeWindows);

SessionWindowedKStream<K, V> windowedBy(SessionWindows sessionWindows);

KTable<K, Long> count(final Materialized<K, Long> materialized);

KTable<K, V> reduce(final Reducer<V> reducer, final Materialized<K, V, KeyValueStore<Bytes, byte[]>>
materialized);

<VR> KTable<K, VR> aggregate(final Initializer<VR> initializer,
                             final Aggregator<? super K, ? super V, VR> aggregator,
                             final Materialized<K, VR, KeyValueStore<Bytes, byte[]>> materialized);
```

KGroupedTable

```
KTable<K, Long> count(final Materialized<K, V, KeyValueStore<Bytes, byte[]>> materialized);

KTable<K, V> reduce(final Reducer<V> adder, final Reducer<V> subtractor, final Materialized<K, V,
KeyValueStore<Bytes, byte[]>> materialized);

<VR> KTable<K, VR> aggregate(final Initializer<VR> initializer,
                             final Aggregator<? super K, ? super V, VR> aggregator,
                             final Aggregator<? super K, ? super V, VR> subtractor,
                             final Materialized<K, VR, KeyValueStore<Bytes, byte[]>> materialized);
```

For *StreamsBuilder* we remove all *stream*, *table*, and *globalTable* overloads that take more than a single argument and replace them with:

StreamsBuilder

```
public synchronized <K, V> KStream<K, V> stream(final String topic)

public synchronized <K, V> KStream<K, V> stream(final String topic, final Consumed<K, V> options)

public synchronized <K, V> KStream<K, V> stream(final Collection<String> topic, final Consumed<K, V> options)

public synchronized <K, V> KStream<K, V> stream(final Collection<String> topic)

public synchronized <K, V> KStream<K, V> stream(final Pattern pattern, final Consumed<K, V> options)

public synchronized <K, V> KTable<K, V> table(final String topic, final Consumed<K, V> consumed)

public synchronized <K, V> KTable<K, V> table(final String topic, final Consumed<K, V> consumed, final
Materialized<K, V, KeyValueStore<Bytes, byte[]>> materialized)

public synchronized <K, V> KTable<K, V> table(final String topic, final Materialized<K, V, KeyValueStore<Bytes,
byte[]>> materialized)

public synchronized <K, V> GlobalKTable<K, V> globalTable(final String topic, final Consumed<K, V> consumed)

public synchronized <K, V> GlobalKTable<K, V> globalTable(final String topic, final Consumed<K, V> consumed,
final Materialized<K, V, KeyValueStore<Bytes, byte[]>> materialized)

public synchronized <K, V> GlobalKTable<K, V> globalTable(final String topic, final Materialized<K, V,
KeyValueStore<Bytes, byte[]>> materialized)
```

New classes and interfaces:

WindowedKStream

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

    KTable<Windowed<K>, Long> count();

    KTable<Windowed<K>, Long> count(final Materialized<K, Long, WindowStore<Bytes, byte[]>> materializedAs);

    <VR> KTable<Windowed<K>, VR> aggregate(final Initializer<VR> initializer,
                                           final Aggregator<? super K, ? super V, VR> aggregator);

    <VR> KTable<Windowed<K>, VR> aggregate(final Initializer<VR> initializer,
                                           final Aggregator<? super K, ? super V, VR> aggregator,
                                           final Materialized<K, VR, WindowStore<Bytes, byte[]>>
materializedAs);


    KTable<Windowed<K>, V> reduce(final Reducer<V> reducer);

    KTable<Windowed<K>, V> reduce(final Reducer<V> reducer,
                                  final Materialized<K, V, WindowStore<Bytes, byte[]>> materializedAs);


}
```

SessionWindowedKStream

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

    KTable<Windowed<K>, Long> count();

    KTable<Windowed<K>, Long> count(final Materialized<K, Long, SessionStore<Bytes, byte[]>> materializedAs);

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

    <VR, T> KTable<Windowed<K>, VR> aggregate(final Initializer<VR> initializer,
                                              final Aggregator<? super K, ? super V, VR> aggregator,
                                              final Merger<? super K, T> sessionMerger,
                                              final Materialized<K, VR, SessionStore<Bytes, byte[]>>
materializedAs);


    KTable<Windowed<K>, V> reduce(final Reducer<V> reducer);

    KTable<Windowed<K>, V> reduce(final Reducer<V> reducer,
                                  final Materialized<K, V, SessionStore<Bytes, byte[]>> materializedAs);
}
```

Materialized

```
/**
 * Used when materializing a state store, i.e, during an aggregation operation or KTable operations
 */
public class Materialized<K, V, S extends StateStore> {

        public static <K, V, S extends StateStore> Materialized<K, V, S> as(final String storeName)

    public static <K, V> Materialized<K, V, WindowStore<Bytes, byte[]>> as(final WindowBytesStoreSupplier
supplier);

    public static <K, V> Materialized<K, V, SessionStore<Bytes, byte[]>> as(final SessionBytesStoreSupplier
supplier);

    public static <K, V> Materialized<K, V, KeyValueStore<Bytes, byte[]>> as(final KeyValueBytesStoreSupplier
supplier)

    public Materialized<K, V, S> withValueSerde(final Serde<V> valueSerde)

    public Materialized<K, V, S> withKeySerde(final Serde<K> valueSerde)

    public Materialized<K, V, S> withLoggingEnabled(final Map<String, String> topicConfig)

    public Materialized<K, V, S> withLoggingDisabled()

    public Materialized<K, V, S> withCachingEnabled()

    public Materialized<K, V, S> withCachingDisabled()

        public String storeName();

        public StoreSupplier<S> storeSupplier()

        public Serde<K> keySerde()

        public Serde<V> valueSerde()

        public boolean loggingEnabled()

        public Map<String, String> logConfig()

        public boolean cachingEnabled()

}
```

Serialized

```
/**
 * Optional params that can be passed to groupBy and groupByKey operations
 */
public class Serialized<K, V> {

    public static <K, V> Serialized<K, V> with(final Serde<K> keySerde, final Serde<V> valueSerde)

    public Serialized<K, V> withKeySerde(final Serde<K> keySerde)

    public Serialized<K, V> withValueSerde(final Serde valueSerde)

        public Serde<K> keySerde()

    public Serde<V> valueSerde()
}
```

Joined

```
/**
 * Optional params that can be passed to join, leftJoin, outerJoin operations
 */
public class Joined<K, V, VO> {

    public static <K, V, VO> Joined<K, V, VO> with(final Serde<K> keySerde, final Serde <V> valueSerde, final
Serde<VO> otherValueSerde)

    public static <K, V, VO> Joined<K, V, VO> keySerde(final Serde<K> keySerde)

    public static <K, V, VO> Joined<K, V, VO> valueSerde(final Serde<V> valueSerde)

    public static <K, V, VO> Joined<K, V, VO> otherValueSerde(final Serde<V> valueSerde)

    public Joined<K, V, VO> withKeySerde(final Serde<K> keySerde)

    public Joined<K, V, VO> withValueSerde(final Serde<V> valueSerde)

    public Joined<K, V, VO> withOtherValueSerde(final Serde<VO> otherValueSerde)

        public Serde<K> keySerde()

    public Serde<V> valueSerde()

        public Serde<VO> otherValueSerde()
}
```

Produced

```
/**
 * Optional arguments that can be specified when doing to and through operations
 */
public static <K, V> Produced<K, V> with(final Serde<K> keySerde, final Serde<V> valueSerde)

public static <K, V> Produced<K, V> with(final Serde<K> keySerde, final Serde<V> valueSerde, final
StreamPartitioner<K, V> partitioner)

public static <K, V> Produced<K, V> keySerde(final Serde<K> keySerde)

public static <K, V> Produced<K, V> valueSerde(final Serde<V> valueSerde)

public static <K, V> Produced<K, V> streamPartitioner(final StreamPartitioner<K, V> partitioner)

public Produced<K, V> withStreamPartitioner(final StreamPartitioner<K, V> partitioner)
public Produced<K, V> withValueSerde(final Serde<V> valueSerde)
public Produced<K, V> withKeySerde(final Serde<K> keySerde)
public Serde<K> keySerde()
public Serde<K> valueSerde()
public StreamPartitioner<K, V> streamPartitioner()
```

Printed

```
/**
 * options that can be used when printing to stdout our writing to a file
 */
public class Printed<K, V> {
    public static <K, V> Printed<K, V> toFile(final String filepath)

    public static <K, V> Printed<K, V> toSysOut()

    public Printed<K, V> withLabel(final String label)

    public Printed<K, V> withKeyValueMapper(final KeyValueMapper<? super K, ? super V, String> mapper)

    public ProcessorSupplier<K, V> build(final String processorName)
}
```

Consumed

```
/**
 * Options for consuming a topic as a KStream or KTable
 */
public class Consumed<K, V> {
    public static <K, V>  Consumed<K, V> with(final Serde<K> keySerde, final Serde<V> valueSerde, final
TimestampExtractor extractor, final Topology.AutoOffsetReset resetPolicy)
    public static <K, V>  Consumed<K, V> with(final Serde<K> keySerde, final Serde<V> valueSerde)
    public static <K, V>  Consumed<K, V> with(final TimestampExtractor extractor)
    public static <K, V>  Consumed<K, V> with(final Topology.AutoOffsetReset resetPolicy)

    public Consumed<K, V> withKeySerde(final Serde<K> keySerde)

    public Consumed<K, V> withValueSerde(final Serde<V> valueSerde)

    public Consumed<K, V> withTimestampExtractor(final TimestampExtractor timestampExtractor)

    public Consumed<K, V> withOffsetResetPolicy(final Topology.AutoOffsetReset resetPolicy)

        public Serde<K> keySerde()

        public Serde<V> valueSerde()

        public TimestampExtractor timestampExtractor()

        public Toploogy.AutoOffsetReset offsetResetPolicy()
}
```

StateStoreBuilder

```
/**
 * Implementations of this will provide the ability to wrap a given StateStore
 * with or without caching/loggging etc.
 */
public interface StoreBuilder<T extends StateStore> {

    StoreBuilder<T> withCachingEnabled();
    StoreBuilder<T> withLoggingEnabled(Map<String, String> config);
    StoreBuilder<T> withLoggingDisabled()
    T build();
    Map<String, String> logConfig();
    boolean loggingEnabled();
}
```

StoreSupplier

```
public interface StoreSupplier<T extends StateStore> {

    /**
     * Return the name of this state store supplier.
     * This must be a valid Kafka topic name; valid characters are ASCII alphanumerics, '.', '_' and '-'
     *
     * @return the name of this state store supplier
     */
    String name();

    /**
     * Return a new {@link StateStore} instance.
     *
     * @return a new {@link StateStore} instance of type T
     */
    T get();

        /**
     * Return a String that is used as the scope for metrics recorded by Metered stores
           * @return metricsScope
           */
    String metricsScope();

}
```

WindowBytesStoreSupplier

```
/**
 * A store supplier that can be used to create one or more {@link WindowStore} instances of type &lt;Byte, byte
[]&gt;
 */
public interface WindowBytesStoreSupplier extends StoreSupplier<WindowStore<Bytes, byte[]>> {
    /**
     * The number of segments the store has. If your store is segmented then this should be the number of
segments
     * in the underlying store. It is also used to reduce the amount of data that is scanned when caching is
enabled
     *
     * @return number of segments
     */
    int segments();

    /**
     * The size of the windows any store created from this supplier is creating
     * @return window size
     */
    long windowSize();

    /**
     * Whether or not this store is retaining duplicate keys. Usually only true if the store is being used
     * for joins. Note this should return false if caching is enabled
     * @return true if duplicates should be retained
     */
    boolean retainDuplicates();

    /**
     * The time period for which the {@link WindowStore} will retain historic data
     * @return retentionPeriod
     */
    long retentionPeriod();
}
```

KeyValueBytesStoreSupplier

```
/**
 * A store supplier that can be used to create one or more {@link KeyValueStore} instances of type &lt;Byte,
byte[]&gt;
 */
public interface KeyValueBytesStoreSupplier extends StoreSupplier<KeyValueStore<Bytes, byte[]>> {

}
```

```
/**
 * A store supplier that can be used to create one or more {@link SessionStore} instances of type &lt;Byte, byte
[]&gt;
 */
public interface SessionBytesStoreSupplier extends StoreSupplier<SessionStore<Bytes, byte[]>> {

    /**
     * The size of a segment, in milliseconds. Used when caching is enabled to segment the cache
     * and reduce the amount of data that needs to be scanned when performing range queries
     *
     * @return segmentInterval in milliseconds
     */
    long segmentIntervalMs();
}
```

Proposed Changes

Add the above methods, interfaces, classes to the DSL. Deprecate existing overloads on *KStream, KTable,* and *KGroupedStream* that take more than the required parameters, for example, *KTable#filter(Predicate, String)* and *KTable#filter(Predicate, StateStoreSupplier)* will be deprecated. *StateStoreSupplier* will also be deprecated. All versions of *KTable#through* and *KTable#to* will be deprecated in favour of using *KTable#toStream()#through and KTable#toStream()#to*

The new Interface *BytesStoreSupplier* supersedes the existing *StateStoreSupplier* (which will remain untouched). This so we can provide a convenient way for users creating custom state stores to wrap them with caching/logging etc if they chose. In order to do this we need to force the inner most store, i.e, the custom store, to be a store of type `<Bytes, byte[]>`.

Compatibility, Deprecation, and Migration Plan

- What impact (if any) will there be on existing users?
    - None - we will deprecate the existing methods so that existing users can continue until they decide to change

Rejected Alternatives

- Using a more fluent api: this approach always results in intermediate stages that require a final *build* or *apply* call to create the underlying *KStream/KTable* etc. We felt that this wasn't quite right.
- Builder for all a params: Rather than specifying the required params and optional params separately we could make each method take a *Builder* that has all of the params. It was felt that this is a but onerous for users that just want to use the required params and don't care about the options