KIP-591: Add Kafka Streams config to set default state store

Status

Current state: *"Accepted"*

Discussion thread:

1. [DISCUSS] KIP-591: Add Kafka Streams config to set default store type

2. discuss: https://lists.apache.org/thread/238rstw9zj49vyhzzoh67d8b74vz4nbm

3. vote: https://lists.apache.org/thread/qz8s06t2brjcv928bo26qqtz06ysg3ln

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

Released:

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

Motivation

Kafka Streams supports RocksDB stores as well as in-memory stores out of the box. By default all DSL operators use RocksDB stores. Currently, it is only possible to switch out RocksDB stores with in-memory store on a per operator basis what is tedious if all stores should be switched (for example in a Kuber netes deployment without local disks).

Furthermore, the current store interface via `Materialized` is very generic as it allows to pass in any custom state store implementation with the side effect that a user needs to name the store and thus also makes it queryable via "Interactive Queries".

We propose to simplify switching the store type via a new config to Kafka Streams that allows to set the default built-in store types for the whole program as well as an improved API for `Materialized`.

Public Interfaces

We propose to add a new configuration parameter `default.dsl.store` that defines the default store type used by the DSL accepting two values `"rocksDB"` (default) and `"in_memory"` .

```
public class StreamsConfig {
    public static final String DEFAULT_DSL_STORE_CONFIG = "default.dsl.store";
    private static final String DEFAULT_DSL_STORE_DOC = "The default store implementation type used by DSL
operators.";

    public static final ROCKS_DB = "rocksDB";
    public static final IN_MEMORY = "in_memory";

    .define(DEFAULT_DSL_STORE_CONFIG,
            Type.STRING,
            ROCKS_DB,
            in(ROCKS_DB, IN_MEMORY),
            Importance.LOW,
                        DEFAULT_DSL_STORE_DOC)
}
```

In addition, we propose to extend `Materialized` config object with a corresponding option to specify the store type:

```
public class Materialized<K, V, S extends StateStore> {

  public enum StoreImplType {
    ROCKS_DB,
    IN_MEMORY
  }


  /**
   * Materialize a {@link StateStore} with the given {@link StoreType}.
   *
   * @param storeType  the type of the state store
   * @param <K>       key type of the store
   * @param <V>       value type of the store
   * @param <S>       type of the {@link StateStore}
   * @return a new {@link Materialized} instance with the given storeName
   */
  public static <K, V, S extends StateStore> Materialized<K, V, S> as(StoreImplType storeType);

  /**
   * Set the type of the materialized {@link StateStore}.
   *
   * @param storeType  the store type {@link StoreType} to use.
   * @return itself
   */
  public Materialized<K, V, S> withStoreType(StoreImplType storeType);
}
```

In order to leverage the new configuration, users will need to call the new proposed API in KAFKA-13281, to provide `topology-config` while construct the `StreamBuilder` . Currently, the internal implementation for `topology-config` is all set. In this KIP, we'll publish the API to allow pass the `topology-config` into `StreamBuilder` .

```
/**
 * Streams configs that apply at the topology level. The values in the {@link StreamsConfig} parameter of the
 * {@link org.apache.kafka.streams.KafkaStreams} or {@link KafkaStreamsNamedTopologyWrapper} constructors will
 * determine the defaults, which can then be overridden for specific topologies by passing them in when
creating the
 * topology builders via the {@link org.apache.kafka.streams.StreamsBuilder()} method.
 */
public class TopologyConfig extends AbstractConfig {
    private static final ConfigDef CONFIG;
    static {
        CONFIG = new ConfigDef()
             .define(BUFFERED_RECORDS_PER_PARTITION_CONFIG,
                     Type.INT,
                     null,
                     Importance.LOW,
                     BUFFERED_RECORDS_PER_PARTITION_DOC)
             .define(CACHE_MAX_BYTES_BUFFERING_CONFIG,
                    Type.LONG,
                    null,
                    Importance.MEDIUM,
                    CACHE_MAX_BYTES_BUFFERING_DOC)
             .define(DEFAULT_DESERIALIZATION_EXCEPTION_HANDLER_CLASS_CONFIG,
                    Type.CLASS,
                    null,
                    Importance.MEDIUM,
                    DEFAULT_DESERIALIZATION_EXCEPTION_HANDLER_CLASS_DOC)
             .define(DEFAULT_TIMESTAMP_EXTRACTOR_CLASS_CONFIG,
                     Type.CLASS,
                     null,
                     Importance.MEDIUM,
                     DEFAULT_TIMESTAMP_EXTRACTOR_CLASS_DOC)
             .define(MAX_TASK_IDLE_MS_CONFIG,
                     Type.LONG,
                     null,
                     Importance.MEDIUM,
                     MAX_TASK_IDLE_MS_DOC)
             .define(TASK_TIMEOUT_MS_CONFIG,
                     Type.LONG,
                     null,
                     Importance.MEDIUM,
                     TASK_TIMEOUT_MS_DOC);
             .define(DEFAULT_DSL_STORE_CONFIG,
                                  Type.STRING,
                            ROCKS_DB,
                             in(ROCKS_DB, IN_MEMORY),
                            Importance.LOW,
                             DEFAULT_DSL_STORE_DOC);
    }

         public TopologyConfig(final StreamsConfig globalAppConfigs) {
        this(null, globalAppConfigs, new Properties());
    }

    public TopologyConfig(final String topologyName, final StreamsConfig globalAppConfigs, final Properties
topologyOverrides) {}

        // check whether this is a named topology
          public boolean isNamedTopology(){}

        // get this task config with the topology config
        public TaskConfig getTaskConfig() {}
```

```
public class StreamsBuilder {
    public StreamsBuilder() {}

        /**
     * Create a {@code StreamsBuilder} instance.
         *
         * @param topologyConfigs    the streams configs that apply at the topology level. Please refer to
{@link TopologyConfig} for more detail
     */
        public StreamsBuilder(final TopologyConfig topologyConfigs) {} // new added
```

Proposed Changes

To create a topology with a non-default store implementation, user can declare it via `new StreamBuilder(topologyConfig)` like this:

```
Properties props = new Properties();
...
// set the "default.dsl.store" config as "in-memory store"
props.put(StreamsConfig.DEFAULT_DSL_STORE_CONFIG, StreamsConfig.IN_MEMORY);
StreamsConfig streamsConfig = new StreamsConfig(props);

builder = new StreamsBuilder(new TopologyConfig(streamsConfig));
topology = builder.build();
```

If not set, or using the old `new StreamBuidler()` it'll default to use `RocksDB store` , which is the same as current behavior. So this is a backward-compatible design.

All stateful operators (like `aggregate()`, `reduce()`, `count()`, or `join()`, will use the state store in the `default.dsl.store` configuration. Of course, if the store is overwritten for an individual operator via `Materialized` the operator overwrite will be used. The improved `Materialized` interface allows to switch the store type more easily without the need to specify a store name.

```
KStream stream = ...

// current code to change from RocksDB to in-memory
stream.groupByKey().count(Materialized.as(Stores.inMemoryKeyValueStore("some-name"));

// new code, without the need to provide a name
stream.groupByKey().count(Materialized.withStoreType(StoreImplType.IN_MEMORY));
```

Compatibility, Deprecation, and Migration Plan

Because the default value of the config is `"ROCKS_DB"` there is no behavior change and thus there are not backward compatibility concerns.

Test Plan

Regular unit and integration testing is sufficient.

Rejected Alternatives

1. use store implementation class for allowing default custom store implementation.

We propose to add a new configuration parameter `default.store.impl.class` that defines the default store implementation class used by the DSL. Default to `RocksDBPersistentStoreImplementation.class`

```
public class StreamsConfig {
    public static final String DEFAULT_STORE_IMPLEMENTATION_CLASS_CONFIG = "default.store.impl.class";
    private static final String DEFAULT_STORE_IMPLEMENTATION_CLASS_DOC =
          "Store supplier implementation class to use. Default is set as RocksDBPersistentStoreImplementation.
It can be overwritten dynamically during streaming operation.";


        .define(DEFAULT_STORE_IMPLEMENTATION_CLASS_CONFIG,
            Type.CLASS,
            RocksDBPersistentStoreImplementation.class.getName(),
            Importance.MEDIUM,
            DEFAULT_STORE_IMPLEMENTATION_CLASS_DOC)
}
```

The `StoreImplementation` interface:

```
/**
 * A state store supplier Implementation interface for all types of {@link StateStore}.
 *
 */
public interface StoreImplementation {
    /**
     * Create a {@link KeyValueBytesStoreSupplier}.
     *
     * @param name  name of the store (cannot be {@code null})
     * @return an instance of a {@link KeyValueBytesStoreSupplier} that can be used
     * to build a persistent key-value store
     */
    KeyValueBytesStoreSupplier keyValueSupplier(String name);

    /**
     * Create a {@link WindowBytesStoreSupplier}.
     *
     * @param name                  name of the store (cannot be {@code null})
     * @param retentionPeriod       length of time to retain data in the store (cannot be negative)
     *                              (note that the retention period must be at least long enough to contain the
     *                              windowed data's entire life cycle, from window-start through window-end,
     *                              and for the entire grace period)
     * @param windowSize            size of the windows (cannot be negative)
     * @param retainDuplicates      whether or not to retain duplicates. Turning this on will automatically
disable
     *                              caching and means that null values will be ignored.
     * @return an instance of {@link WindowBytesStoreSupplier}
     * @throws IllegalArgumentException if {@code retentionPeriod} or {@code windowSize} can't be represented
as {@code long milliseconds}
     * @throws IllegalArgumentException if {@code retentionPeriod} is smaller than {@code windowSize}
        */
    WindowBytesStoreSupplier windowBytesStoreSupplier(String name,
                                                      Duration retentionPeriod,
                                                      Duration windowSize,
                                                      boolean retainDuplicates);


    /**
     * Create a {@link SessionBytesStoreSupplier}.
     *
     * @param name              name of the store (cannot be {@code null})
     * @param retentionPeriod   length of time to retain data in the store (cannot be negative)
     *                          (note that the retention period must be at least as long enough to
     *                          contain the inactivity gap of the session and the entire grace period.)
     * @return an instance of a {@link  SessionBytesStoreSupplier}
     */
    SessionBytesStoreSupplier sessionBytesStoreSupplier(String name,
                                                        Duration retentionPeriod);
}
```

For the StoreImplementation, Kakfa provides 3 built-in kinds of implementation:

RocksDBPersistentStoreImplementation.java (default)

```
/**
 * A Rocks DB persistent state store supplier Implementation.
 *
 */
public class RocksDBPersistentStoreImplementation implements StoreImplementation {
    @Override
    public KeyValueBytesStoreSupplier keyValueSupplier(final String name) {
        return Stores.persistentTimestampedKeyValueStore(name);
    }

    @Override
    public WindowBytesStoreSupplier windowBytesStoreSupplier(final String name,
                                                             final Duration retentionPeriod,
                                                             final Duration windowSize,
                                                             final boolean retainDuplicates) {
        return Stores.persistentTimestampedWindowStore(name, retentionPeriod, windowSize, retainDuplicates);
    }

    @Override
    public SessionBytesStoreSupplier sessionBytesStoreSupplier(final String name, final Duration
retentionPeriod) {
        return Stores.persistentSessionStore(name, retentionPeriod);
    }
}
```

RocksDBStoreImplementation.java

```
/**
 * A Rocks DB state store supplier Implementation.
 *
 */
public class RocksDBStoreImplementation implements StoreImplementation {
    @Override
    public KeyValueBytesStoreSupplier keyValueSupplier(final String name) {
        return Stores.persistentKeyValueStore(name);
    }

    @Override
    public WindowBytesStoreSupplier windowBytesStoreSupplier(final String name,
                                                             final Duration retentionPeriod,
                                                             final Duration windowSize,
                                                             final boolean retainDuplicates) {
        return Stores.persistentWindowStore(name, retentionPeriod, windowSize, retainDuplicates);
    }

    @Override
    public SessionBytesStoreSupplier sessionBytesStoreSupplier(final String name, final Duration
retentionPeriod) {
        return Stores.persistentSessionStore(name, retentionPeriod);
    }
}
```

InMemoryStoreImplementation.java

```
/**
 * A In-memory state store supplier Implementation.
 *
 */
public class InMemoryStoreImplementation implements StoreImplementation {
    @Override
    public KeyValueBytesStoreSupplier keyValueSupplier(final String name) {
        return Stores.inMemoryKeyValueStore(name);
    }

    @Override
    public WindowBytesStoreSupplier windowBytesStoreSupplier(final String name,
                                                             final Duration retentionPeriod,
                                                             final Duration windowSize,
                                                             final boolean retainDuplicates) {
        return Stores.inMemoryWindowStore(name, retentionPeriod, windowSize, retainDuplicates);
    }

    @Override
    public SessionBytesStoreSupplier sessionBytesStoreSupplier(final String name, final Duration
retentionPeriod) {
        return Stores.inMemorySessionStore(name, retentionPeriod);
    }
}
```

In Materialized class, we provided one more API to allow users to provide `StoreImplementation` dynamically.

```
        /**
     * Materialize a {@link StateStore} with the store implementation.
     *
     * @param storeImplementation  store implementation used to materialize the store
     * @param <K>       key type of the store
     * @param <V>       value type of the store
     * @param <S>       type of the {@link StateStore}

     * @return a new {@link Materialized} instance with the given storeImplementation
     */
    public static <K, V, S extends StateStore> Materialized<K, V, S> as(final StoreImplementation
storeImplementation)
```

in the KIP, there's a trade-off regarding the API complexity.
With the store impl, we can support default custom stores, but introduce more complexity for users, while with the enum types, users can configure default built-in store types easily, but it can't work for custom stores.

For me, I'm OK to narrow down the scope and introduce the default built-in enum store types first.
And if there's further request, we can consider a better way to support default store impl.