KIP-954: expand default DSL store configuration to custom types

Status

Current state: *Accepted*

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

1. **Supporting Custom Stores.** We want to complete the work outlined in KIP-591, specifically extending the functionality to cover custom state stores. This KIP expands on and makes feasible the "Rejected Alternative" in KIP-591, which was rejected in favor of reducing scope. Without this KIP, DSL users must carefully examine the output of their topology description and wire in custom state stores (using the `Materialized` API) at every location that uses state. This is both tedious and error prone; if `Materialized` is used in an incorrect location an unnecessary state store may be created. In the inverse scenario, missing a `Materialized` call would silently create a RocksDB store.
2. **Supporting Default Stores for Stream-Stream Joins.** We would like to address a gap in the API for stream-stream joins (really 3 separate issues). For a quick refresher, these are a special case among DSL operators that cannot be configured via `Materialized` and are instead the lone instance of an operator configuration accepting a StoreSupplier directly (rather than indirectly, eg via a StoreBuilder or Materialized). For this reason, they are completely missed by the *default.dsl.store* config as it currently operates exclusively through the Materialized class. The only way to configure the underlying state store implementation is with the StreamJoined configuration class.
3. **Support Custom Stores for Stream-Stream Joins.** As described in KAFKA-14976, some stream-stream join types now use an additional store which is not windowed but rather a KeyValueStore. This is not customizable and not exposed to the user in any way – it simply hardcodes a RocksDBStore or InMemoryStore depending on what the join's WindowStoreSupplier(s) return for the StateStore#persistent API. This means no matter what one does, there is no way to use all (and only) custom state store implementations for any application including these stream-stream joins!

The 2nd problem follows from the 1st: because you are forced to configure these stores through the StreamJoined class, which in turn requires a WindowStoreSupplier (or two), it's up to the user to figure out the correct values for all three parameters (windowSize, retentionPeriod, and retainDuplicates). And not only is there indeed a "correct" answer for all three (unlike other operators which let users freely choose some or all of the store parameters), it is difficult to determine what those expected values are. For example, retainDuplicates must always be true for stream-stream joins, while the retentionPeriod is fixed at the windowSize + gracePeriod. Most frustratingly, however, is that the windowSize itself is not what you might assume or expect: if you used the old JoinWindows#of (or new JoinWindows#ofTimeDifferenceAndNoGrace), the WindowStoreSupplier will actually require twice the "time difference" passed into your JoinWindows. And the only way to find out is trial and error or checking the code after you get a runtime exception for passing in the wrong value(s).

Public Interfaces

New Interfaces

In order to allow users to implement their own default stores, we will need to provide the following interface. This interface will be used when instantiating a new store when no StoreSupplier is provided (either via `Materialized` or `StoreSupplier` APIs):

DSLStoreProvider.java

```
/**
 * {@code DslStoreSuppliers} defines a grouping of factories to construct
 * stores for each of the types of state store implementations in Kafka
 * Streams. This allows configuration of a default store supplier beyond
 * the builtin defaults of RocksDB and In-Memory.
 *
 * <p>There are various ways that this configuration can be supplied to
 * the application (in order of precedence):
 * <ol>
 *     <li>Passed in directly to a DSL operator via either
 *     {@link org.apache.kafka.streams.kstream.Materialized#as(DslStoreSuppliers)},
 *     {@link org.apache.kafka.streams.kstream.Materialized#withStoreType(DslStoreSuppliers)}, or
 *     {@link org.apache.kafka.streams.kstream.StreamJoined#withDslStoreSuppliers(DslStoreSuppliers)}</li>
 *
 *     <li>Passed in via a Topology configuration override (configured in a
 *     {@link org.apache.kafka.streams.TopologyConfig} and passed into the
 *     {@link org.apache.kafka.streams.StreamsBuilder#StreamsBuilder(TopologyConfig)} constructor</li>
 *
 *     <li>Configured as a global default in {@link org.apache.kafka.streams.StreamsConfig} using
 *     the {@link org.apache.kafka.streams.StreamsConfig#DSL_STORE_SUPPLIERS_CLASS_CONFIG}</li>
 *     configuration.
 * </ol>
 *
 * <p>Kafka Streams is packaged with some pre-existing {@code DslStoreSuppliers}
 * that exist in {@link BuiltInDslStoreSuppliers}
 */
public interface DslStoreSuppliers extends Configurable {

    KeyValueBytesStoreSupplier keyValueStore(final DslKeyValueParams params);

    WindowBytesStoreSupplier windowStore(final DslWindowParams params);

    SessionBytesStoreSupplier sessionStore(final DslSessionParams params);
}

// the below are all "struct"-like classes with the following fields
class DslKeyValueParams(String name);
class DslWindowParams(String name, Duration retentionPeriod, Duration windowSize, boolean retainDuplicates,
EmitStrategy emitStrategy, boolean isSlidingWindow, boolean isTimestamped);
class DslSessionParams(String name, Duration retentionPeriod, EmitStrategy emitStrategy);
```

Note on Evolving API: a concern raised on KIP-591 about having such an interface is that the increased API surface area would mean introducing new store implementations would cause custom state store implementations to throw compile time errors. Introducing the *Params classes will prevent such issues unless an entirely new store type is added.

If an entirely new state store type (beyond KV/Windowed/Session) is added - I think it is valid to have new store types have a default implementation that throws `new UnsupportedOperationException()` as it is unlikely that users that specify a custom state store as the default will want a different (e.g. ROCKS_DB) store created without them knowing. This also seems unlikely since these three have been there for many years and they've been the only three for that duration.

OOTB Store Type Specs

We will provide a default implementations of the above interfaces:

- org.apache.kafka.streams.RocksDbDslStoreSuppliers
- org.apache.kafka.streams.InMemoryDslStoreSuppliers

Configuration

These interfaces will be specified by means of a new config, defined below. This takes precedence over the old default.dsl.store config, which will be deprecated to provide clear signal on the preferred method. Further discussion on this point is under "Rejected Alternatives":

```
public static final String DSL_STORE_SUPPLIERS_CLASS_CONFIG = "dsl.store.suppliers.class";
public static final String DSL_STORE_SUPPLIERS_CLASS_DOC = "Defines which store implementations to plug in to
DSL operators. Must implement the <code>org.apache.kafka.streams.state.DslStoreSuppliers</code> interface.";
public static final Class DSL_STORE_SUPPLIERS_CLASS_DEFAULT = org.apache.kafka.streams.RocksDbDslStoreSuppliers.
class;

// we also add this constant for the old configuration so that it is easily referenceable
// across the code base
public static final String DEFAULT_DSL_STORE = ROCKS_DB;
```

Example usage of this configuration:

```
dsl.store.suppliers.class = com.my.company.MyCustomStoreSuppliers
```

> **Behavior Change for default.dsl.store**
>
> The new `dsl.store.suppliers.class` will respect the configurations when passed in via `KafkaStreams#new(Topology, StreamsConfig)` (and other related constructors) instead of only being respected when passed in to the initial `StoreBuilder#new(TopologyConfig)`, which is what the old default.`dsl.store` configuration required.

TopologyConfig

We will expose the following method from `TopologyConfig` :

```
    /**
     * @return the DslStoreSuppliers if the value was explicitly configured (either by
     *         {@link StreamsConfig#DEFAULT_DSL_STORE} or {@link
StreamsConfig#DSL_STORE_SUPPLIERS_CLASS_CONFIG})
     */
    public Optional<DslStoreSuppliers> resolveDslStoreSuppliers() {
        if (isTopologyOverride(DSL_STORE_SUPPLIERS_CLASS_CONFIG, topologyOverrides) || globalAppConfigs.
originals().containsKey(DSL_STORE_SUPPLIERS_CLASS_CONFIG)) {
            return Optional.of(Utils.newInstance(dslStoreSuppliers, DslStoreSuppliers.class));
        } else if (isTopologyOverride(DEFAULT_DSL_STORE_CONFIG, topologyOverrides) || globalAppConfigs.
originals().containsKey(DEFAULT_DSL_STORE_CONFIG)) {
            return Optional.of(MaterializedInternal.parse(storeType));
        } else {
            return Optional.empty();
        }
```

This allows us to determine whether or not a DslStoreSuppliers was configured or not, and also respects the hierarchy of resolution described in the javadocs to DslStoreSuppliers.

Materialized API

In the existing code, users are able to override the default store type by leveraging the existing `Materialized.as` and `Materialized.withStoreType` methods. We will change the signature of these methods (see below on how we maintain compatibility) to take in a `DslStoreSuppliers` instead of the `StoreType` enum:

Materialized.java

```
public class Materialized {

    public static <K, V, S extends StateStore> Materialized<K, V, S> as(final DslStoreSuppliers storeSuppliers);

    public Materialized<K, V, S> withStoreType(final DslStoreSuppliers storeSuppliers);
}
```

In order to ensure code compatibility and smooth upgrade paths, we will leave the old enums in place and have them implement DslStoreSuppliers:

Stores.java

```
package org.apache.kafka.streams.kstream;

public class Materialized {
        ...
        public enum StoreType implements DslStoreSuppliers {
                // DefaultDslStoreSuppliers will itself be a delegate to two different implementations
                // which will be stored in static fields ROCKS_DB and IN_MEMORY, choosing which one
                // to delegate to depending on the default.dsl.store configuration
            ROCKS_DB(DefaultDslStoreSuppliers.ROCKS_DB),
            IN_MEMORY(DefaultDslStoreSuppliers.IN_MEMORY);

                private final DslStoreSuppliers delegate;

                StoreTypeSpec(final DslStoreSuppliers delegate) {
                        this.delegate = delegate;
                }

                // delegate all methods to delegate
        }
        ...
}
```

StreamJoined API

Finally, to address the three problems with configuring state stores for stream-stream joins today, we propose one last new method that will operate similarly to the new Materialized APIs but for StreamJoined:

StreamJoined.java

```
public class StreamJoined {

    /**
     * Creates a StreamJoined instance with the given {@link DslStoreSuppliers}. The store plugin
     * will be used to get all the state stores in this operation that do not otherwise have an
     * explicitly configured {@link org.apache.kafka.streams.state.DslStoreSuppliers}.
     *
     * @param storeSuppliers         the store plugin that will be used for state stores
     * @param <K>                 the key type
     * @param <V1>                this value type
     * @param <V2>                other value type
     * @return                    {@link StreamJoined} instance
     */
    public static <K, V1, V2> StreamJoined<K, V1, V2> with(final DslStoreSuppliers storeSuppliers);

    /**
     * Configure with the provided {@link DslStoreSuppliers} for all state stores that are not
     * configured with a {@link org.apache.kafka.streams.state.DslStoreSuppliers} already.
     *
     * @param storeSuppliers  the store plugin to use for plugging in state stores
     * @return             a new {@link StreamJoined} configured with this store plugin
     */
    public StreamJoined<K, V1, V2> withstoreSuppliers(final DslStoreSuppliers storeSuppliers);
}
```

Proposed Changes

There will be no functional changes as part of this KIP. Implementations for all of the above interfaces will be provided. Note the following discussion points:

- **Terminology: "Store Type" vs "Store Implementation**". A store "type" is a top level DSL store type - there are currently only three: KV, Windowed and Session. A "store implementation" is the chosen implementation of the specified store type. Kafka Streams provides RocksDB and in-memory store implementations OOTB.

- **Versioned Tables,** like timestamped tables, are an implementation detail of the chosen plugin when providing a store type (usually KV stores). In the future, the DefaultDslStoreSuppliers is free to switch its default implementation of the KV store to return a versioned implementation instead. This KIP does not propose promoting Versioned tables to the top level API here.

Compatibility, Deprecation, and Migration Plan

- *What impact (if any) will there be on existing users?*

Existing users will see deprecation warnings if they are using the old *default.dsl.store* configuration in their code. When 4.0 is released this configuration will no longer be respected.

- *If we are changing behavior how will we phase out the older behavior?*

N/A

- *If we need special migration tools, describe them here.*

N/A

- *When will we remove the existing behavior?*

4.0

Test Plan

Existing test coverage for the default.dsl.store configuration will be migrated to use the new configuration and ensure that the default behavior can be changed.

Rejected Alternatives

Support Both Configs

Don't deprecate the old *default.dsl.store* config and instead maintain it alongside the new config. The advantage here is obviously that we minimize disruption to anyone using this config already. However I believe the existing userbase is likely to be quite small, as in its current form this feature is only useful to those who (a) have many state stores to the point overriding them separately is cumbersome, and (b) also wishes to override all (or most) of their stores to be in-memory specifically. This seems to be a relatively uncommon pattern already, not least of which being that it's a recipe for OOM disaster unless their applications are extremely small and/or low traffic. In the absence of any evidence that this config is already in use despite such a niche application, it seems best to take this opportunity to deprecate it now, before it gains any more traction, and have everyone converge on the same, single configuration going forward. Whenever there are two configs affecting the same underlying feature, it's inevitable there will be confusion over their interaction/hierarchy which can often lead to misconfiguration. Deprecating the old config is a clear sign to users which of the two should be preferred and will take precedence.

Add CUSTOM Enum to Existing Config

Instead of positioning the new config as a full replacement for *default.dsl.store*, we could instead introduce it as a complementary config for custom stores specifically. For example, add a 3rd StoreType to the enum definition, such as *CUSTOM*. If (and only if) the *CUSTOM* StoreType is used, the value of the new *default.dsl.store.type.spec* config will be looked at and used to obtain the actual StoreSupplier instances of this custom type. This option has the same advantages as the above, though I would qualify it in the same way. In general, having two configs that do the same thing or are tied to the same feature will be confusing to new (or old!) users. However, between these two alternatives I would personally advocate for this one as the better option, as it at least solves the concern over potential misconfiguration due to unclear interaction between the two configs.

Deprecating StoreType Enum, Materialized.as(StoreType) and Materialized.withStoreType(StoreType) APIs

We could deprecate the `Materialized.StoreType` enum and introduce new methods specific to the new store type spec:

```
public class Materialized {

        // new methods
    public static <K, V, S extends StateStore> Materialized<K, V, S> as(final DslStoreSuppliers storeSuppliers);
    public Materialized<K, V, S> withStoreSuppliers(final DslStoreSuppliers storestoreSuppliers);

        // deprecate old methods
        @Deprecated
    public static <K, V, S extends StateStore> Materialized<K, V, S> as(final StoreType storeType);
        @Deprecated
    public Materialized<K, V, S> withStoreType(final StoreType storeType);
}
```

This is a viable strategy, but rejected for two reasons:

1. Using an enum is more ergonomic for users in the code (compare Materialized.withStoreType(ROCKS_DB) with Materialized.withStoreTypeSpec (new RocksDbStoreTypeSpec())
2. It causes code-level deprecation, which is annoying for users to migrate away from