KIP-820: Extend KStream process with new Processor API

Status

Current state: Implemented and merged

Discussion thread: here

JIRA: here

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

Motivation

KIP-478, the new strongly-typed Processor API, brings the option to reconsider the abstractions around custom processing in the Kafka Streams DSL.

Currently, multiple `Transformers` and a final `Processor` operation are the options available to implement custom processing with access to the Record context; including record metadata (e.g. topic, partition, offset), timestamp, and headers; and reference to state stores.

There have been discussions on how to refactor these APIs:

- https://issues.apache.org/jira/browse/KAFKA-8396
- https://issues.apache.org/jira/browse/KAFKA-8410
- https://issues.apache.org/jira/browse/KAFKA-10603

`Transformers` are currently limited to the *old* `ProcessorContext`, and there have been multiple extensions to support value processing without repartitioning and one-to-many record processing. With the addition of the new `Processor` API, `KStream` can be extended to access the newer, typed API and effectively replace most of the operations `*Transforms` offer at the moment, with more open control to forward records.

This KIP could be considered as a step towards deprecating `Transforms`, though that should be discussed in a follow-up KIP.

Public Interfaces

Modified methods

- `KStream<KOut,VOut> KStream#process(ProcessorSupplier<K, V, KOut, VOut> processorSupplier, String... stateStoreNames)`
  - `from void KStream#process(ProcessorSupplier<K, V, Void, Void> processorSupplier, ...)`
- `KStream<KOut,VOut> KStream#process(ProcessorSupplier<K, V, KOut, VOut> processorSupplier, Named named, String... stateStoreNames)`
  - `from void KStream#process(ProcessorSupplier<K, V, Void, Void> processorSupplier, ...)`

New methods

Processors without forcing repartitioning:

- `KStream<K,VOut> KStream#processValues(FixedKeyProcessorSupplier<K, V, K, VOut> processorSupplier, String... stateStoreNames)`
- `KStream<K,VOut> KStream#processValues(FixedKeyProcessorSupplier<K, V, K, VOut> processorSupplier, Named named, String... stateStoreNames)`

Internal changes

- Infrastructure for Fixed Key Records:
  - `FixedKeyRecord`:
    - Not a `Record` sub/superclass to casting to `Record`.
    - Private constructor to avoid reconstructing record and change key.
  - `FixedKeyProcessor(Supplier)` and `FixedKeyProcessorContext` interfaces for `FixedKeyRecord`.
    - `FixedKeyContextualProcessor` abstract class, similar to `ContextualProcessor`.

Deprecated methods

All transform operations on the KStream will be deprecated in favor of `process` and `processValues` operations:

- `KStream#transform`
- `KStream#flatTransform`
- `KStream#transformValues`
- `KStream#flatTransformValues`

Proposed Changes

`KStream#process` and `KStream#processValues` to replacing most `Transformers`

With the ability to manage `forward` calls as part of the `Processor` itself, `transform`, `valueTransform`, `flatTransform`, and `flatValueTransform` can be replaced by a `process`/`processValues`:

```
Topology topology() {
  final var builder = new StreamsBuilder();
  builder.stream("words", Consumed.with(Serdes.String(), Serdes.String()));
      .processValues(() -> new FixedKeyContextualProcessor<String, String, String>() {
          @Override
          public void process(FixedKeyRecord<String, String> record) {
              for (final var word : record.value().split(",")) {
                  context().forward(record.withValue("Hello " + word));
              }
          }}, Named.as("process-values-without-repartitioning"))
      .process(() -> new ContextualProcessor<String, String, String, String>() {
          @Override
          public void process(Record<String, String> record) {
              for (final var word : record.value().split(",")) {
                  context().forward(record.withKey(word).withValue("Hello " + word));
              }
          }}, Named.as("process-with-partitioning"))

      .to("output", Produced.with(Serdes.String(), Serdes.String()));
  return builder.build();
}
```

Infrastructure for Fixed Key Records

`FixedKeyRecord`

Record with immutable key.

```
public final class FixedKeyRecord<K, V> {

    private final K key;
    private final V value;
    private final long timestamp;
    private final Headers headers;

    FixedKeyRecord(final K key, final V value, final long timestamp, final Headers headers) {
        this.key = key;
        this.value = value;
        if (timestamp < 0) {
            throw new StreamsException(
                "Malformed Record",
                new IllegalArgumentException("Timestamp may not be negative. Got: " + timestamp)
            );
        }
        this.timestamp = timestamp;
        this.headers = new RecordHeaders(headers);
    }

    public K key() { return key; }

    public V value() { return value; }

    public long timestamp() { return timestamp; }

    public Headers headers() { return headers; }

    public <NewV> FixedKeyRecord<K, NewV> withValue(final NewV value) { return new FixedKeyRecord<>(key, value,
timestamp, headers); }

    public FixedKeyRecord<K, V> withTimestamp(final long timestamp) { return new FixedKeyRecord<>(key, value,
timestamp, headers); }

    public FixedKeyRecord<K, V> withHeaders(final Headers headers) { return new FixedKeyRecord<>(key, value,
timestamp, headers); }
}
```

FixedKeyProcessorSupplier

```
@FunctionalInterface
public interface FixedKeyProcessorSupplier<KIn, VIn, VOut> extends ConnectedStoreProvider,
Supplier<FixedKeyProcessor<KIn, VIn, VOut>> {
    FixedKeyProcessor<KIn, VIn, VOut> get();
}
```

FixedKeyProcessor

```
public interface FixedKeyProcessor<KIn, VIn, VOut> {

    default void init(final FixedKeyProcessorContext<KIn, VOut> context) {}

    void process(FixedKeyRecord<KIn, VIn> record);

    default void close() {}
}
```

FixedKeyContextualProcessor

Helper, same as `ContextualProcessor`.

```
public abstract class FixedKeyContextualProcessor<KIn, VIn, VOut> implements FixedKeyProcessor<KIn, VIn, VOut> {

    private FixedKeyProcessorContext<KIn, VOut> context;

    protected FixedKeyContextualProcessor() {}

    @Override
    public void init(final FixedKeyProcessorContext<KIn, VOut> context) {
        this.context = context;
    }

    protected final FixedKeyProcessorContext<KIn, VOut> context() {
        return context;
    }
}
```

ProcessingContext

To be extended by `FixedKeyProcessorContext` and `ProcessorContext`:

```
interface ProcessingContext {

    String applicationId();

    TaskId taskId();

    Optional<RecordMetadata> recordMetadata();

    Serde<?> keySerde();

    Serde<?> valueSerde();

    File stateDir();

    StreamsMetrics metrics();

    <S extends StateStore> S getStateStore(final String name);

    Cancellable schedule(final Duration interval,
                         final PunctuationType type,
                         final Punctuator callback);

    void commit();

    Map<String, Object> appConfigsWithPrefix(final String prefix);
}
```

FixedKeyProcessorContext

```
public interface FixedKeyProcessorContext<KForward, VForward> extends ProcessingContext {

    <K extends KForward, V extends VForward> void forward(FixedKeyRecord<K, V> record);

    <K extends KForward, V extends VForward> void forward(FixedKeyRecord<K, V> record, final String childName);
}
```

Compatibility, Deprecation, and Migration Plan

`KStreams#process` Return type change

Changing return type from `void` to `KStream<KOut, VOut>` is source-compatible, but not binary-compatible. It will require users to recompile the application to use the latest version of the library.

Though users will not require to change their code as the current returned value is void, and the input processor supplier types will include `Void` that is the current type for output key and value.

`KStream#*transform*` deprecations

This KIP is including the deprecation of the `transform` operations on `KStream` to propose using the latest Processor API operations.

`Transform` API is not marked as deprecated yet, as it requires additional considerations:

- `Transform` API is broadly adopted for custom processing. Even though most functionality is possible to implement with new `Processor`, this migration is not straightforward.
- `KTable#transformValues` is an interesting use-case for `ValueTransformerWithKey` as the internal value is `Change<V>` — an internal type to handle record changes on KTable — not exposed as public API. An approach would be to deprecate this method in favor of `.toStream().processValues().toTable()`.

A new KIP should be proposed to continue the deprecation of `Transformer` APIs.

Rejected Alternatives

Migrate Transform APIs to the latest ProcessContext

This would involve extending KStream/KTable APIs even more with specific Transform variants. If a new Processor can support most `Transform` in the long-term becoming the adopted option to transform values, flat transform, etc.; then we should start getting the new `Processor` properly adopted in the DSL and let the usage drive the next steps.

Runtime key validation on `KStream#processValues` and changing key output type to Void

On one hand, with Void type for key output, we force the users to cast to Void and change the key to null, though this can be documented on the API, so the users are aware of the peculiarity of forwarding within `processValues`.

On the other hand, keeping the key type as output doesn't *require* to do any change of keys, but this could lead to key-checking runtime exceptions when changing keys.

This KIP proposes adding a new Record type for Fixed Keys, to remove the need for runtime checked and reduce as much as possible the ability to change the key when implementing the processor.

References

- Cleanup old Processor work: https://issues.apache.org/jira/browse/KAFKA-12939
- Reference implementation: https://github.com/jeqo/kafka/tree/kstream-new-process
- Draft implementation of fixed record key: https://github.com/apache/kafka/pull/11854