KIP-914: DSL Processor Semantics for Versioned Stores

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

KIP-889 introduced the interface for versioned key-value stores in Kafka Streams, and a RocksDB-based implementation, but intentionally deferred additional functionality including usage of the new store type from DSL processors. This KIP addresses usage of versioned key-value stores from existing DSL processors.

As described in KIP-889, a large part of the motivation for introducing versioned stores is to address the semantic gap in stream-table joins today. When an out-of-order record arrives on the stream side, the record should be joined against the table state at the current record's timestamp, rather than the latest record seen so far. With versioned stores, this is now possible by performing a timestamped lookup into the table store.

Additionally, versioned stores differ from non-versioned key-value stores in that `get(key)` returns the latest record by timestamp rather than the latest record by offset. This difference has implications for other processors as well, such as the table-table join processors. When a new record arrives at a table-table join processor, the processor calls `get(key)` on the opposite state store (i.e., materialization) to find the record to join with. If the state store is versioned, the result could be different compared to for a non-versioned store in the presence of out-of-order data. Analogous implications exist for table aggregate processors, and a specific optimization in the table filter processor is relevant as well.

This KIP aims to clarify and define intended processor behavior with versioned stores.

Public Interfaces

DSL Processors

No changes to any public interfaces themselves, but the behavior of these existing public interfaces will change when relevant state stores are versioned:

- `KStream#join(KTable, ...)` : stream-table join processors will be updated to perform timestamped lookups if the table is versioned
- `KTable#join(KTable, ...)`: table-table join processors, including foreign key joins, will be updated to not produce new join results on out-of-order records (by key) from versioned tables
- `KTable#filter(...)` : table filter processors today have an optimization to drop null results if the previous result for the same key was also null. This optimization will be disabled if the table being filtered is versioned
- `KGroupedTable#aggregate()` , and related methods `KGroupedTable#count()` and `KGroupedTable#reduce()` : table aggregate processors will be updated to ignore out-of-order records (by key) when aggregating a versioned table
- `KTable#suppress()` : this method will be disabled for versioned tables. A `TopologyException` will be thrown.

For the precise definition of what counts as a "versioned table," see the first paragraph of the "Proposed Changes" section.

Versioned Stores

Also a couple updates to versioned stores themselves, to aid in shoring up processor semantics:

- GlobalKTables will no longer be allowed to be versioned
- `VersionedKeyValueStore#put(...)` will now have a return type: a long which is the validTo timestamp of the newly put record, with two special values to indicate either that no such timestamp exists (because the record is the latest for its key) or that the put did not take place (because grace period has elapsed). This also requires a corresponding change in `VersionedBytesStore` , specifically, a new `put(key, value, timestamp)` method will be added.

```
public interface VersionedKeyValueStore<K, V> extends StateStore {

  long put(K key, V value, long timestamp); // <-- replaces the existing `void put(K key, V value, long
timestamp);`

  // other existing methods unchanged
}

public interface VersionedBytesStore extends KeyValueStore<Bytes, byte[]>, TimestampedBytesStore {

  long put(Bytes key, byte[] value, long timestamp); // <-- new method

  // other existing methods unchanged
}
```

Proposed Changes

The changes proposed to processor semantics above only apply to processors operating on versioned tables. A table is versioned iff a table is materialized as a versioned store upstream, with no materialization as an unversioned store or stateful transformation (aggregation or join) in between. In other words, a table will no longer be considered versioned downstream if any of the following occur:

- an explicit materialization, i.e., passing a `Materialized` instance into a method which produces a resulting KTable, as an unversioned store
- a stateful operation (aggregation or join) without an explicit `Materialized` instance
    - for aggregations, this happens because all aggregations are materialized, and implicit materialization today is always unversioned
    - for foreign-key table-table joins, this happens because all foreign-key joins are materialized, and implicit materialization today is always unversioned
    - for primary-key table-table joins, the result is not materialized by default but the result table should not be considered versioned today (see section on table-table joins below), and this design choice helps make this condition about stateful operations easier to reason about
- converting to a stream and calling `KStream#toTable()` to convert back. Converting a table to a stream causes the table to lose its versioned /unversioned status. In order to preserve such a table as versioned, users can pass an explicit `Materialized` instance into the `KStream#toTable()` call to materialize the table with a versioned store.

By default, all tables in an application are unversioned. Only once a user opts in by passing a `Materialized` instance to materialize a table with a versioned store does the first table become versioned. Downstream tables implicitly become versioned until one of the conditions above causes a downstream table to become unversioned, after which subsequent derived tables will be unversioned as well (until another table is explicitly materialized to be versioned).

Stream-Table Joins Perform Timestamped Lookups on Versioned Tables

Today, when a new record (from the stream-side) arrives at a stream-table join processor, the processor performs a `get(key)` lookup on the table materialization to find the record to join with. If the table store is versioned, the behavior will be to call `get(key, timestamp)` instead, where `timestamp` is the stream-side record's timestamp. This change provides proper temporal join semantics, when users opt in to using versioned stores.

In the event that the stream-side record timestamp is older than the versioned store's history retention and `get(key, timestamp)` returns null, the join processor will handle this null in the same way as other nulls – for inner joins, no join result will be produced, whereas for left joins a join result with null table value will be produced. (The RocksDB-based versioned store implementation introduced in KIP-889 will log a warning on calls to `get(key, timestamp)` which have exceeded history retention, but there will be no other indication that this has occurred.)

For inner joins, this is the best option because any other table-side record that we might try to join the stream-side record with would result in incorrect temporal join semantics. For left joins, we could also consider the option of dropping the stream-side record (for which there is no table-side record due to history retention having elapsed) entirely rather than emitting a join result with null. Both approaches have their merits; there could be use cases which require/prefer stream-side records never be dropped from the join, while other use cases might prefer to exclude such records from the join result. I think it's best to have the processor err on the side of including the records and allowing users to identify and filter them out downstream if needed, rather than the reverse where users will have no way of getting the dropped records back. (There are also implementation challenges associated with having the processor determine whether a null returned from `get(key, timestamp)` is due to the store's history retention having been exceeded or not. To address these would be outside the scope of this KIP and likely require a KIP of its own. If this alternative behavior turns out to be preferable, we can introduce a config for it in the future.)

Table-Table Joins Do Not Produce Join Results on Out-of-order Records from Versioned Tables

This change is more nuanced and warrants additional context.

Background

The way that primary-key table-table joins work today is, whenever a new record arrives at the processor, a `get(key)` lookup is performed on the opposite table's materialization to find the record to join with, and the join result is emitted with the larger timestamp of the two records being joined. The latest (by offset) join result is always the join of the latest (by offset) records from each of the two tables.

If there are no out-of-order records, then the timestamps of the join results will be non-decreasing as well, which means that the latest-by-offset join result is also the latest-by-timestamp join result. If there are out-of-order records, it is no longer true that the join result timestamps will be non-decreasing; it is possible that consumers of the join results will see time "move backwards." When using non-versioned stores, this is (perhaps) acceptable since state store semantics are offset-based, and the join respects that -- a join result is produced from the latest-by-offset records from each table, and this join result is the latest-by-offset record in the result topic as well.

Versioned stores aim to provide timestamp-based semantics instead. A table-table join operating on versioned tables (i.e., tables materialized with versioned stores) should join the latest-by-timestamp records from each table, and this resulting join record should be the latest-by-timestamp in the result topic.

We cannot guarantee this without changes to the table-table join processor logic today. Consider the following example of A join B where both stores are versioned, and all records are for the same key:

```
A: (timestamp = 0, value = a0)
A: (timestamp = 4, value = a4)
B: (timestamp = 2, value = b2) -> emits (a4, b2) with timestamp=max(4, 2)=4
B: (timestamp = 1, value = b1) -> emits (a4, b1) with timestamp=max(4, 1)=4
```

The arrival of the out-of-order record on the B side has triggered a new join result (a4, b1) with the same timestamp as the previous join result (a4, b2), which means the new join result replaces the previous one, even though the previous one is the join of the latest-by-timestamp records from each table. There are a number of ways we can try to fix this.

Attempt 1

One idea is to say, when joining versioned tables, if the new record is not out-of-order, emit the latest join result. Else, perform a single timestamped lookup into the other store and emit an older result.

With this approach, the above example becomes:

```
A: (timestamp = 0, value = a0)
A: (timestamp = 4, value = a4)
B: (timestamp = 2, value = b2) -> latest record on B side, joins with latest on A side to emit (a4, b2) with
timestamp=max(4, 2)=4
B: (timestamp = 1, value = b1) -> out-of-order record on B side, timestamped lookup on A side finds a0 to emit
(a0, b1) with timestamp=max(0, 1)=1
```

This ensures that the latest (by timestamp) join result will indeed always be the join of the latest (by timestamp) records from each table, so the original concern is fixed. However, the value of the out-of-order join results emitted is dubious, as they are not guaranteed to be correct. Consider a different example:

```
A: (timestamp = 0, value = a0)
B: (timestamp = 0, value = b0) -> latest record on B side, emits (a0, b0) with timestamp=0
A: (timestamp = 4, value = a4) -> latest record on A side, emits (a4, b0) with timestamp=4
B: (timestamp = 3, value = b3) -> latest record on B side, emits (a4, b3) with timestamp=4
B: (timestamp = 2, value = b2) -> out-of-order record on B side, emits (a0, b2) with timestamp=2
A: (timestamp = 1, value = a1) -> out-of-order record on A side, emits (a1, b0) with timestamp=1, but now the
(a0, b2) for timestamp=2 above is incorrect.
```

Everything was fine up until the last record came. The (a1, b0) join result triggered by the last record is correct, but the (a0, b2) record triggered by the previous record is now incorrect; it should be (a1, b2) instead. In order to properly emit (a1, b2) in addition to (a1, b0) when then a1 record arrives, we have to perform a scan for versions to join with in the B store, instead of performing a single timestamped lookup.

Attempt 2

Suppose we want to maintain that older join results are always accurate. As noted above, this requires performing a scan for old record versions in the versioned store. This certainly works, but it quickly becomes complicated and the number of join results that need to be emitted grows quickly as well.

```
A: (timestamp = 0, value = a0)
A: (timestamp = 5, value = a5)
B: (timestamp = 2, value = b2) -> emits two join results: (a0, b2) for ts=2, and (a5, b2) for ts=5
B: (timestamp = 3, value = b3) -> emits two join results: (a0, b3) for ts=3, and (a5, b3) for ts=5
B: (timestamp = 4, value = b4) -> emits two join results: (a0, b4) for ts=4, and (a5, b4) for ts=5
A: (timestamp = 1, value = a1) -> emits three join results: (a1, b2) for ts=2 and (a1, b3) for ts=3 and (a1,
b4) for ts=4
```

... And we can see how the complexity grows rather quickly. In the future it could be nice to add an option for the join processor to emit this complete older version history for the join result, to support use cases which require it, but this volume of updates does not make sense as the default. Adding such an option is deferred to a future KIP.

Proposal

Handling out-of-order records by performing a single timestamped lookup as in Attempt 1 does not guarantee correctness of the older join result, and performing a scan to guarantee correct older version history as in Attempt 2 is expensive. In light of this, I think it's best to simply not produce older join results for now. Specifically, when joining versioned tables, if the new record is not out-of-order, emit the latest join result. Else, emit nothing. The example above becomes:

```
A: (timestamp = 0, value = a0)
A: (timestamp = 5, value = a5)
B: (timestamp = 2, value = b2) -> latest record on B side, emits (a5, b2) with timestamp=5
B: (timestamp = 3, value = b3) -> latest record on B side, emits (a5, b3) with timestamp=5
B: (timestamp = 4, value = b4) -> latest record on B side, emits (a5, b4) with timestamp=5
A: (timestamp = 1, value = a1) -> out-of-order record on A side, no new join result emitted
```

A consequence of this is that the join result timestamps are once again guaranteed to be non-decreasing. The latest-by-timestamp join result is also the latest-by-offset result, which makes downstream consumption of the join results table suitable even for consumers which don't already handle versioning.

Note that it's still possible there will be older (by timestamp and by offset) join results in the join results topic, and these results are not guaranteed to be correct. The correctness guarantee only extends to the latest (by timestamp and offset) result.

```
A: (timestamp = 0, value = a0)
B: (timestamp = 2, value = b2) -> latest record on B side, emits (a0, b2) with timestamp=2
A: (timestamp = 5, value = a5) -> latest record on A side, emits (a5, b2) with timestamp=5
A: (timestamp = 1, value = a1) -> out-of-order record on A side, no new join result emitted. the older (a0, b2)
join result is no longer correct.
```

For users that require that all older join results are correct, we will have to introduce an option in the future for implementing Attempt 2 above.

The examples above discussed primary-key table-table joins, but the same reasoning and analysis extends to foreign-key joins, and this KIP proposes to make the analogous updates to foreign-key joins as well.

In the case of a table-table join where one side is versioned and the other is not, out-of-order records from the versioned side will not trigger a new join result, but out-of-order records from the unversioned side will. This is inline with the understanding that a versioned table follows timestamp-based semantics, while an unversioned table follows offset-based semantics. The join result guarantees that the latest-by-offset result is the join of the latest-by-timestamp record from the versioned side with the latest-by-offset record from the unversioned side, so it is valid to interpret the join result as an unversioned table. It is not recommended to interpret the join result of a join with mixed versioning as a versioned table, since there are no guarantees about the latest-by-timestamp result in the presence of out-of-order records from the unversioned source table.

Finally, it's worth nothing that even when joining versioned tables, the table-table join processors will only ever call `get(key)` and not `get(key, timestamp)` . The table history retentions still have an indirect implication for this use case, though, since history retention doubles as grace period for the store today (subject to change in the future). Because grace period is per store instance, which has task-level granularity, that means if grace period is set too low then the latest record for one key could be dropped from the store if another key has already advanced the store's observed stream time past the grace period by the time that this record is seen. In light of this, users should still set history retention high enough to capture records from different keys which may arrive out-of-order, even though `get(key, timestamp)` may never be called.

Table Filter Processors Applied to Versioned Tables Do Not Drop Duplicate Tombstones

The DSL table filter processor today has an optimization to drop null values if the previous (filtered) result for the same key is also null. The purpose of this optimization is to avoid sending unnecessary tombstones downstream. When filtering a versioned table, however, duplicate tombstones may substantively change the result table in the presence of out-of-order data, if the result table is also interpreted as a versioned table.

Consider the following series of filtered records, where all records have the same key:

```
Filter                                    Downstream table
----------------------------------------
(v1, ts=1)                -->             (v1, ts=1)
(null, ts=2)        -->               (null, ts=2)
(null, ts=4)        -->               [n/a]
(v2, ts=3)                -->             (v2, ts=3)
```

With the optimization enabled, `(null, ts=4)` is not forwarded downstream because the previous result (with timestamp 2) is also a tombstone. If the downstream table is interpreted as being versioned, the result is now incorrect because it appears that the latest value is `v2` with timestamp 3, when the latest value should actually be `null` with timestamp 4. In order to address this, I propose to disable the optimization for dropping duplicate tombstones when filtering versioned tables.

There is a case to be made that whether the optimization to ignore duplicate tombstones is enabled or not should depend on whether a table is materialized as a versioned table downstream, rather than upstream, since if a table is not materialized as a versioned table downstream then it doesn't matter whether extra tombstones are preserved in order to maintain correct version history. However, this is not true in the case of stream-table joins. If a table is materialized with a versioned store and then a filter is applied prior to joining with a stream, then whether the filter preserves proper version history affects the result of the stream-table join. Additionally, the existing proposal to have processor semantics for versioned tables apply only based on what happens upstream and not downstream (i.e., it can be determined whether a table is versioned or not by only considering the upstream topology, and not anything that happens downstream) for all processors is more consistent/easier to reason about.

Table Aggregations Ignore Out-of-order Records from Versioned Tables

When aggregating a table, updates to the table are accounted for in the aggregation by first removing the old value from the aggregate, and then adding the new value into the aggregate. "Old" and "new" are currently determined based on record offsets, but for versioned tables they should be determined based on record timestamps instead. In other words, if a table is versioned and there is out-of-order data for a particular key, the aggregation result should include the record with the latest timestamp, not the record with the latest key.

In order to achieve this, we should update table aggregate processors to ignore out-of-order records when aggregating versioned tables. This includes `KTable#count()` and `KTable#reduce()` in addition to `KTable#aggregate()` .

```
Record (all with same key)               Effect on unversioned aggregate                      Effect on
versioned aggregate
-------------------------------------------------------------------------------------------------------
(v1, ts=1)                                                 add:
v1                                                                          add: v1
(v2, ts=10)                                                 remove: v1 ; add:
v2                                       remove: v1 ; add: v2
(v3, ts=5)                                                 remove: v2 ; add:
v3                                       nothing -- v3 has an older timestamp than v2
```

Suppress Cannot Be Applied to Versioned Tables

The `KTable#suppress()` operation suppresses table updates (per key) and only forwards records downstream periodically. The purpose of a versioned table is to store a complete record history (subject to history retention time) for each key, so it's unclear what the semantics around suppressing updates should be.

If suppress is called on a versioned table, should the operator still collapse updates based on record offset as it does today? This would produce a versioned today with incomplete version history, and it's unclear what the use cases for this would be.

Should the operator collapse updates by keeping the latest by timestamp record instead? This would be consistent with the fact that other processors use timestamp-based semantics for versioned tables, but the value of suppressing updates in this way is limited as well. It amounts to dropping out-of-order records (per key) from the versioned table, which can be done more efficiently without using suppress and its associated buffer.

Until we have a clearer understanding of possible use cases for suppressing a versioned table, it's best to disallow it in order to avoid surprising/confusing semantics.

Other Processors and the Long-Term Vision

The changes proposed to the processors above are necessary for coherent semantics when using versioned stores. This KIP does not propose changes to any of the other processors, and it's worth discussing why.

At the heart of the complexities around the table-table join semantics above is that it's not clear how tables should handle out-of-order records. Given that a `KStream#toTable()` method exists and that we fully expect streams to contain out-of-order records, it makes sense to support creating a table from a stream containing out-of-order records, particularly in light of versioned tables. But the possibility for tables with out-of-order records creates ambiguity for how table processors should handle these records. Rather than requiring each processor to handle these complexities (as loosely proposed above), I think it'd be better to introduce a "grace period" at table source nodes and in `KStream#toTable()` to reorder out-of-order records (per key) so that tables, at least the versioned ones, do not have to worry about out-of-order data at downstream processors.

That said, this is not included in the scope of this KIP, since this KIP is focused only on the minimal processor updates required for versioned stores to "make sense" with existing processors today. Reordering out-of-order data at source table nodes is a larger change, and only nice-to-have rather than necessary.

In light of this long-term vision, this KIP does not propose changes to any of the other table processors either. Table operations such as `filter` , `mapValues` , and `transformValues` should not be in the business of re-ordering or dropping records, even if a table is materialized as a versioned table before or after any of these steps. (In fact, it's only because the existing `filter` implementation does drop records (tombstones) under certain situations that the `filter` processor needs an update in this KIP in the first place.) Table aggregation operations including `count` and `reduce` already produce results with non-decreasing timestamps, and therefore do not need updates at this time beyond ensuring that the proper records are included in the aggregation. And the requisite updates for table-table join processors have already been discussed above.

In the future if we update source table nodes to reorder out-of-order data for versioned tables, then this current proposed special-handling for versioned tables in the various table processors (table-table join, table aggregate, and table filter) will no longer be needed either.

GlobalKTables Cannot Be Versioned

At the time of publishing KIP-889 for introducing versioned stores, it didn't seem like there were reasons GlobalKTables shouldn't be allowed to be versioned but in thinking about processor semantics we now have a reason. Because KStream-GlobalKTable joins today are not timestamp aware, as in, there is no guarantee on the processing order of stream-side records relative to records from the global table (due to the fact that the threads consuming from each are separate, and there is no synchronization), this means the benefit of table versioning during a KStream-GlobalKTable join is limited and also challenging to reason about. In the interest of limiting surface area for now, let's disallow global tables from being versioned since we can always enable it later on if needed.

If a user tries to pass a versioned state store as the materialization for a global table, a TopologyException will be thrown when building the topology, and the error message will make clear that this is why.

Versioned Stores Return `validTo` Timestamp on Put

The purpose of this change is to provide more information about the outcome of a call to `VersionedKeyValueStore#put(...)` . The return value will be the validTo timestamp of the newly put record, with two special values to be aware of:

- -1, used throughout Streams to indicate "no timestamp," means that the record which was put is the latest record version for the particular key, and therefore there is no validTo timestamp
- `Long.MIN_VALUE` will indicate that the grace period has elapsed, and the put did not take place.

This additional metadata about the result of the put enables processors to determine when a record is out-of-order, so that table-table joins and aggregates may drop out-of-order data as describe above. The inclusion of the special `MIN_VALUE` return value to indicate that the put did not take place is included both to disambiguate from the -1 case and also so that callers may choose to take different actions in this case.

As part of updating the return type for `VersionedKeyValueStore#put(...)` , a corresponding update to `VersionedBytesStore` is needed in order to pass the return value when converting from `KeyValueStore<Bytes, byte[]>` back to `VersionedKeyValueStore` . This is largely an implementation detail since users are not expected to interact directly with `VersionedBytesStore` , and is only included as part of this KIP because `VersionedBytesStore` is technically a public interface (because it is required from `VersionedBytesStoreSupplier` , the return type from `Stores#persistentVersionedKeyValueStore(...)` .

The reason a new method is added to `VersionedBytesStore` rather than updating an existing one is because the existing `put(key, value)` method of `VersionedBytesStore` cannot be updated as the method signature is inherited from `KeyValueStore<Bytes, byte[]>` . As a workaround, a new method with signature `put(key, value, timestamp)` is added instead. This new signature also more closely aligns with the `VersionedKeyValueStore` usage of the method, which allows us to simplify the internal implementation.

Compatibility, Deprecation, and Migration Plan

If discussion for this KIP wraps up in the next couple weeks before the 3.5 KIP deadline, then there are no compatibility concerns because the changes in this KIP only affect versioned stores, versioned stores are only opt-in, and KIP-889 for introducing versioned stores is also going out in 3.5 so there are no concerns with existing apps using versioned stores either.

Test Plan

Integration testing for the new types of processor functionality using versioned stores: validating results with different combinations of in-order and out-of-order data.

Rejected Alternatives

Implement the long-term vision directly

As described above, it'd be nice if – long term – table source nodes (including `KStream#toTable()`) implemented a "grace period" to use in re-ordering out-of-order records, at least for versioned tables, as this would obviate the need for discussion about how individual table processors should handle out-of-order data.

I'd like to ship KIP-889 and versioned stores without blocking on this as it's a larger, more complex change with follow-up design discussions needed, whereas versioned stores in their current form can already provide value to users. In order to ship KIP-889, there are some minimal processor changes required, and those are the only ones proposed in this KIP.

Have table-table joins on versioned tables produce an older join result on out-of-order data

See discussion above ("Attempt 1" and "Attempt 2"). Producing a complete, accurate history of older joins results is costly. Producing an incomplete history without accuracy guarantees is not valuable to users.

Drop out-of-order records from versioned tables everywhere, not just in table-table joins and table aggregations

Rather than updating just the table-table join and table aggregate processors to not produce/update results on out-of-order data from versioned tables, we could consider extending this decision to "ignore" out-of-order data for all table processors when using versioned stores. I don't think we should do this because the out-of-order data is still valuable -- we might still want to keep it in the versioned table in certain situations. However, we do know that we don't want it for purposes of the table-table join (unless we want the table-table join to produce a complete, accurate history of older join results, which is discussed above) or table aggregate, which is why it's fine to drop out-of-order data at these specific processors.

If instead of dropping the out-of-order records entirely we re-order them, and if instead of doing this in every processor we do it once at the table source, then this suggestion is the same as the long-term vision proposed above.

Make these changes to the various processors configurable

The changes are only being applied to versioned stores, which are purely opt-in. If users don't want the changes, they don't have to opt in. If it turns out there are use cases for wanting versioned tables but not wanting these changes, we can discuss adding a config as a follow-up.

A more structured return type for `VersionedKeyValueStore#put(...)`

Instead of returning a long with a special value to indicate whether the put call was accepted or not, we could return a structured type such as "PutMetadata" which contains a long field (for the validTo timestamp) and a boolean (to indicate whether the put was accepted). This would allow more evolvability in the future, if it turns out there are other types of metadata which users have use for, but would also be more cumbersome in the meantime. Given that we don't anticipate additional pieces of metadata being valuable to return on put, let's keep the return type simple rather than building for evolvability that will likely go unused.

Additional stream-table join improvements, such as grace periods

There are additional improvements it'd be good to make to the Kafka Streams join semantics, such as introducing a "grace period" for stream-table joins so that, after a stream-side record arrives, there is time for potential out-of-order table data to arrive before emitting a join result. These improvements would be good to make but are outside the scope of this KIP. (Interestingly, using versioned tables and implementing the re-ordering change in the long-term vision means this specific improvement would not be needed either.)