KIP-314: KTable to GlobalKTable Bi-directional Join

Status

Current state: Under Discussion

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).

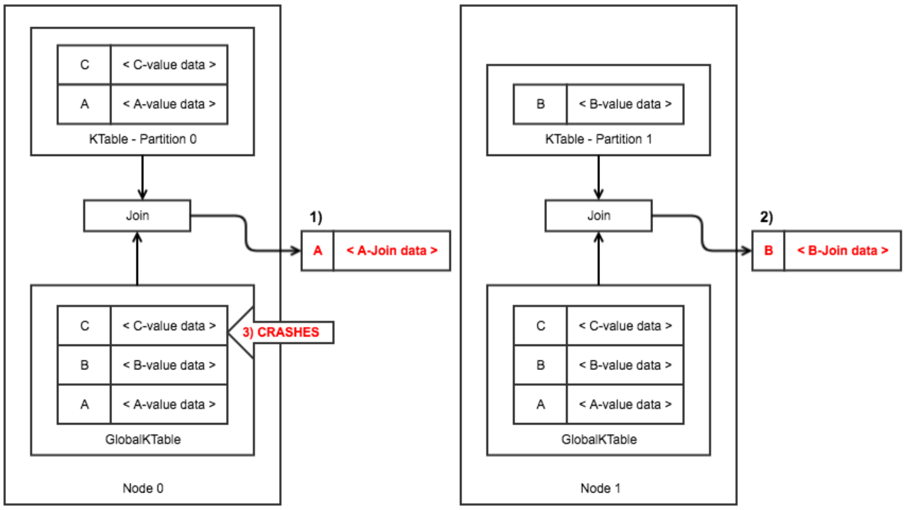
Unresolved Impediments

There are two major impediments to resolving this KIP in the way that is proposed in the initial design. These impediments are external to how the joins are performed, but are instead related to managing data consistency between various failure modes.

Impediment 1 - Rebalancing and Offset Management

When a partition is rebalanced from one node to another, the "virtual offset" of the GlobalKTable is not maintained. The term "virtual offset" is used to indicate the exact events which were processed by a single node's GlobalKTable and those which were not (see the Impediment 3, Problems with a GlobalKTable Virtual Offset). This piece of data is missing and there is no reasonably easy way to store it between failures. This "virtual offset" would need to be assigned to the partitions that are reshuffled. This is problematic as nothing in the current Kafka standard accommodates this piece of information, and refactoring of the GlobalKTable implementation would need to occur to accommodate this.

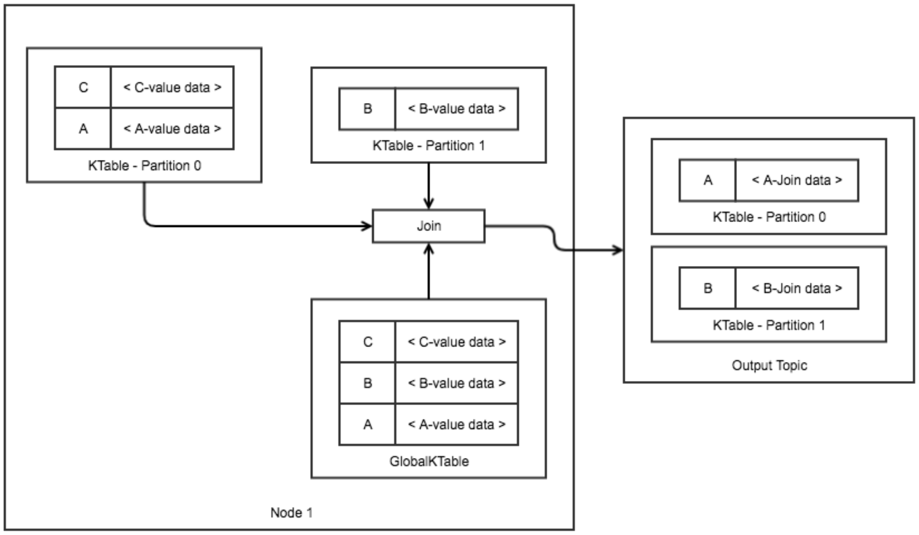
This issue is illustrated in the following example. In the next two images, it can be seen that despite expectations that the "C" event added to the GlobalKTable should join with the KTable, depending on failure timings the event could be completely skipped.



1) Keyed event "A" is inserted into the GlobalKTable. Updates are propagated from node 0 matches.

2) Keyed event "B" is inserted into the GlobalKTable. Updates are propagated from node 1 matches.

3) Keyed event "C" is inserted into the GlobalKTable. Node 1 processes the event (with no output), but node 0 crashes. Note that at this point, both Partition 0 and Partition 1 of the KTable source topic are at the head - there are no new events to consume.



1) Partition 0 is rebalanced on to Node 1. GlobalKTable has already processed all previous updates, and the offset for Partition 0 is also at the head node, as per when Node 0 crashed.

Note: The output data is inconsistent with what we would expect. The updated GlobalKTable-driven value for C is not present in the output topic, and there is no way to tell that it has been missed.

Impediment 2 - Restoring the GKT vs Replaying the entire topic

In this scenario, a node is brought up and a GlobalKTable driving a KTable join is created. Ideally we would want to start exactly where the GlobalKTable driver left off, but this information is absent. Without "virtual offsets", there is no way to know where a GlobalKTable left off processing, and so the GlobalKTable needs to be built up from the start so that we do not miss any events. Failure to reproduce this exact state for each processing node would result in the issue identified above in Impediment #1, where data could be missed.

Without a virtual offset, the alternative is that each new node that is brought online creates a new GlobalKTable and processes data from the beginning of the GlobalKTable. This would cause a significant amount of duplicate processing and old, temporarily inconsistent state to be published and propagated. While it would end up being eventually consistent, it is very noisy and would be awful to deal with in practice.

Impediment 3 - Problems with a GlobalKTable Virtual Offset

The ideal globalKTable virtual offset would be a monotonically increasing index that is identical between all nodes. A given index value would have a specific event, regardless of which node the index was referenced from. Unfortunately, each GlobalKTable consumer will consume messages as in the order they become available to that particular node. Under reasonable volume, or given different starting times, there is no guarantee that any virtual index generated on one node would match a virtual index generated on another node. This is just fundamentally lacking from GlobalKTable implementation, **and to me it seems that this is a fatal flaw to this entire idea**.

There is, however, perfect determinism at the offset level of a given *partition*, which is the entire basis of Kafka's distribution methodology. This suggests that the solution to this problem is really to implement KIP-213 Support non-key joining in KTable.

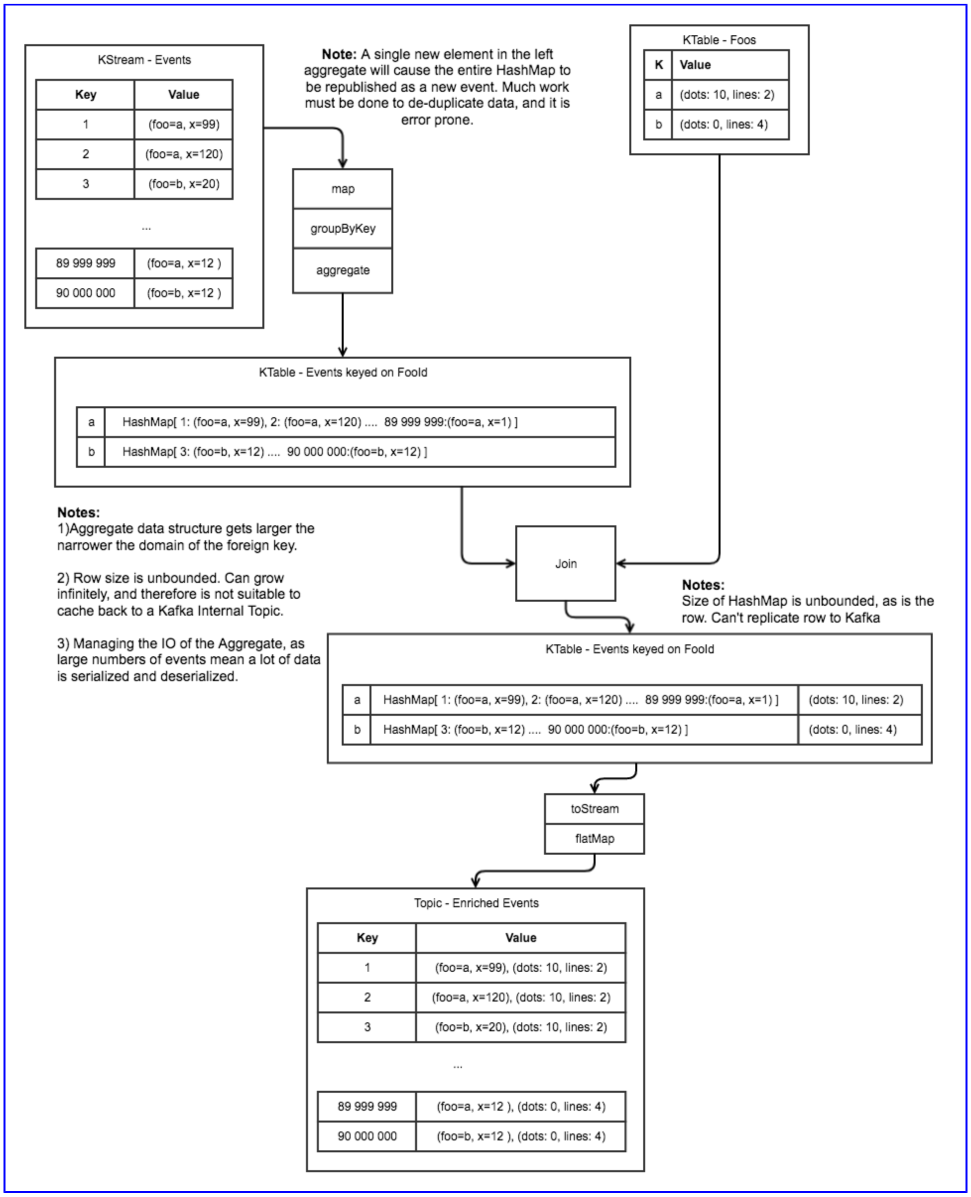
Motivation

There are a number of tools which liberate relational data into the event streaming domain, such as Kafka Connect and Debezium. These pull the relational rows from a database and convert them into events for use in downstream Kafka Stream consumers. One of the main issues with this is that the data is often still highly normalized, and these normalized relationships prove difficult to use with existing Kafka Streams. At this point, it is often desirable to denormalize some of these relationships using the Kafka Streams API.

One main requirement that I have run into several times is that the denormalized data be updated should any of its components change. This is required for a consistent view of the data for all downstream consumers. To do this, we must perform stateful joins using KTable to KTables. Unfortunately, this approach also requires that we rekey the data on the foreign key, which while supported, introduces a number of its own issues. Furthermore, solutions to this approach become untenable as the data volume grows. Further details into the problems with this approach can be seen by this excellent blog post by Dan Lebrero.

http://danlebrero.com/2017/05/07/kafka-streams-ktable-globalktable-joining-reference-data/

For relational data with a very narrow domain, rekeying on the foreign key becomes prohibitively expensive in terms of maintaining the data structures. Additional complexities about managing order, duplicate outputs and race conditions also begin to apply. In fact, the narrower the domain of the foreign keyed data, the worse the problem becomes. However, with KTable joins to GlobalKTables, the narrower the domain of the data the easier it is for the globalKTable to provide easy and performant joins without managing all of this overhead. A basic illustration of this problem is given in the following image.



KStream - Events

Key	Value
1	(foo=a, x=99)
2	(foo=a, x=120)
3	(foo=b, x=20)
...	
89 999 999	(foo=a, x=12)
90 000 000	(foo=b, x=12)

Note: A single new element in the left aggregate will cause the entire HashMap to be republished as a new event. Much work must be done to de-duplicate data, and it is error prone.

KTable - Foos

K	Value
a	(dots: 10, lines: 2)
b	(dots: 0, lines: 4)

KTable - Events keyed on FooId

a	HashMap[1: (foo=a, x=99), 2: (foo=a, x=120) .... 89 999 999:(foo=a, x=1)]
b	HashMap[3: (foo=b, x=12) .... 90 000 000:(foo=b, x=12)]

Notes:
1)Aggregate data structure gets larger the narrower the domain of the foreign key.

2) Row size is unbounded. Can grow infinitely, and therefore is not suitable to cache back to a Kafka Internal Topic.

3) Managing the IO of the Aggregate, as large numbers of events mean a lot of data is serialized and deserialized.

Notes:
Size of HashMap is unbounded, as is the row. Can't replicate row to Kafka

KTable - Events keyed on FooId

a	HashMap[1: (foo=a, x=99), 2: (foo=a, x=120) .... 89 999 999:(foo=a, x=1)]	(dots: 10, lines: 2)
b	HashMap[3: (foo=b, x=12) .... 90 000 000:(foo=b, x=12)]	(dots: 0, lines: 4)

Topic - Enriched Events

Key	Value
1	(foo=a, x=99), (dots: 10, lines: 2)
2	(foo=a, x=120), (dots: 10, lines: 2)
3	(foo=b, x=20), (dots: 10, lines: 2)
...	
89 999 999	(foo=a, x=12), (dots: 0, lines: 4)
90 000 000	(foo=b, x=12), (dots: 0, lines: 4)

By allowing for a KTable-to-GlobalKTable join, ***driven by both sides***, this problem space can be greatly simplified and result in a higher performance, simpler way to handle highly relational data. I have personally worked on several Kafka Streams applications that follow this approach described above, and it is only due to the relatively friendly nature of much of our data that we have managed to get a semblance of functionality out of it. For highly normalized data, such a pattern is not tenable. As it currently stands in Kafka Streams there is no easy way to handle multiple joins on small, normalized tables in a way that is easy to reason about and easy to implement.

Public Interfaces

streams/kstreams/KTable.java

```
public interface KTable<K, V> {
        <GK, GV, RV> KTable<K, RV> join(final GlobalKTable<GK, GV> globalKTable,
                                  final KeyValueMapper<? super K, ? super V, ? extends GK> keyMapper,
                                  final ValueJoiner<? super V, ? super GV, ? extends RV> joiner);

        <GK, GV, RV> KTable<K, RV> leftJoin(final GlobalKTable<GK, GV> globalKTable,
                                      final KeyValueMapper<? super K, ? super V, ? extends GK> keyMapper,
                                      final ValueJoiner<? super V, ? super GV, ? extends RV> joiner);
}
```

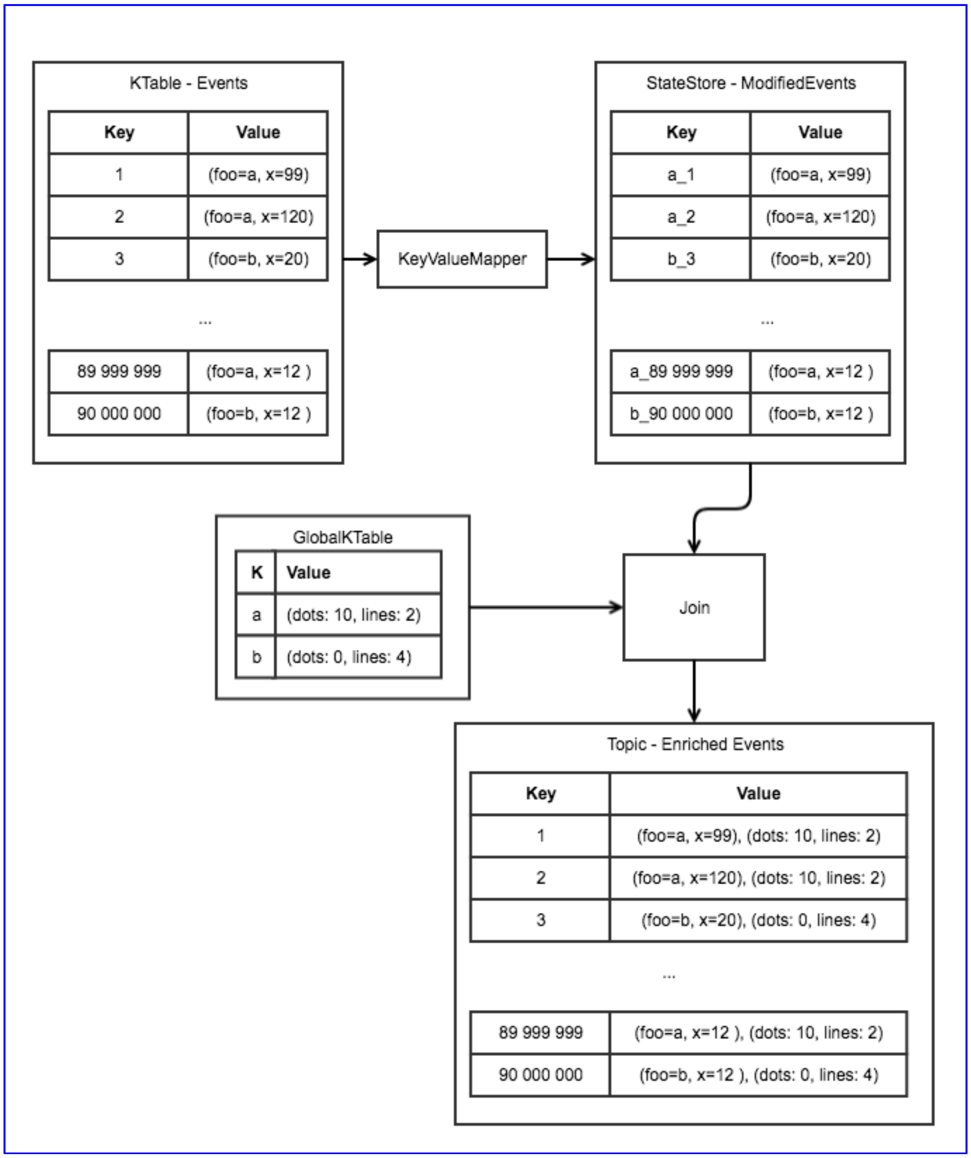
Proposed Changes

Summary

- A new KTable to GlobalKTable inner and left join. Note that the GlobalKTable would always be on the right side of the join. There would be no GlobalKTable-to-KTable left join.
- GlobalKTable's output can be attached to a processor, to drive the join from the right side of the KTable-to-GlobalKTable join.
- Use RocksDB prefix RangeScan to join from the GlobalKTable, using a the re-keyed table.

Proposal

The above is illustrated at a high level in the following diagram. A new KTable is created with the results of the KeyValueMapper as a prefix of the state store Key. This in turn allows for updated KTable rows to be joined directly by using specific prefix joiner logic, which will be included in this JIRA. In addition, the GlobalKTable can also attach itself to a processor, and induce joins on the KTable. This is necessary to fulfill the requirement of keeping all stateful data up to date, which is a normal operation when updating data in a relational database. This can be seen in the diagram following the next.



KTable - Events

Key	Value
1	(foo=a, x=99)
2	(foo=a, x=120)
3	(foo=b, x=20)
...	
89 999 999	(foo=a, x=12)
90 000 000	(foo=b, x=12)

StateStore - ModifiedEvents

Key	Value
a_1	(foo=a, x=99)
a_2	(foo=a, x=120)
b_3	(foo=b, x=20)
...	
a_89 999 999	(foo=a, x=12)
b_90 000 000	(foo=b, x=12)

GlobalKTable

K	Value
a	(dots: 10, lines: 2)
b	(dots: 0, lines: 4)

Topic - Enriched Events

Key	Value
1	(foo=a, x=99), (dots: 10, lines: 2)
2	(foo=a, x=120), (dots: 10, lines: 2)
3	(foo=b, x=20), (dots: 0, lines: 4)
...	
89 999 999	(foo=a, x=12), (dots: 10, lines: 2)
90 000 000	(foo=b, x=12), (dots: 0, lines: 4)

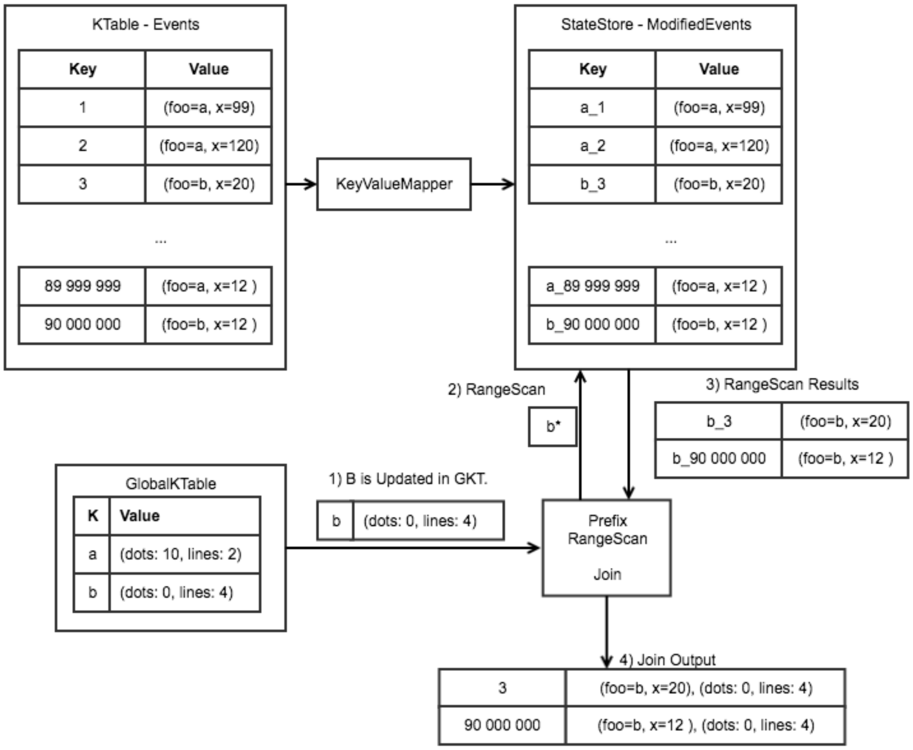
GlobalKTable as Driver, joined on KTable join mechanism.

The following image illustrates the GlobalKTable join mechanism.

1. The GlobalKTable receives an update from its upstream topic. This exercises the join logic on the ModifiedEvents table.

2. A Range Scan on the GKT key is performed on the ModifiedEvents store. This will enable use to retrieve all of the elements from the KTable that satisfy the prefix.
3. The Range Scan results are returned to the join processor, where the join logic is executed.
4. The processor joins the data together using **inner-join** logic. The subsequent events are output from the processor to the downstream consumers.

Note: The GlobalKTable cannot be on the left in a Left-Join. This would result in each instance producing an event on a GKT update, which is not supported.



KTable - Events

Key	Value
1	(foo=a, x=99)
2	(foo=a, x=120)
3	(foo=b, x=20)
...	
89 999 999	(foo=a, x=12)
90 000 000	(foo=b, x=12)

StateStore - ModifiedEvents

Key	Value
a_1	(foo=a, x=99)
a_2	(foo=a, x=120)
b_3	(foo=b, x=20)
...	
a_89 999 999	(foo=a, x=12)
b_90 000 000	(foo=b, x=12)

3) RangeScan Results

b_3	(foo=b, x=20)
b_90 000 000	(foo=b, x=12)

GlobalKTable

K	Value
a	(dots: 10, lines: 2)
b	(dots: 0, lines: 4)

1) B is Updated in GKT.

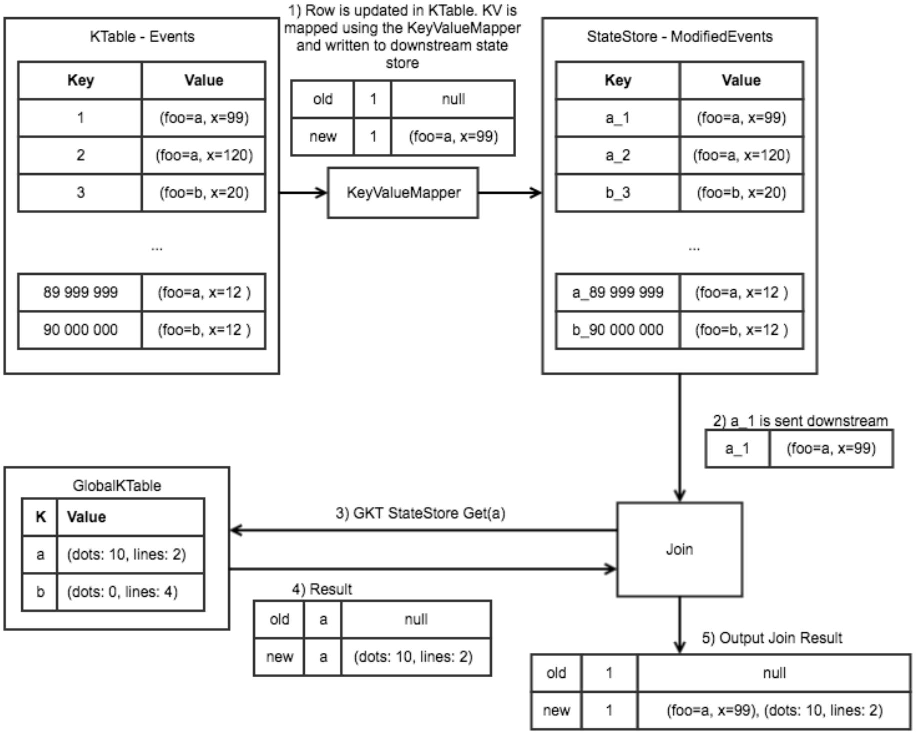
b	(dots: 0, lines: 4)

4) Join Output

3	(foo=b, x=20), (dots: 0, lines: 4)
90 000 000	(foo=b, x=12), (dots: 0, lines: 4)

KTable as Driver, joined on GlobalKTable join mechanism

1. The row is updated in the KTable - Events, with a Change being sent to the downstream consumer. The Key in the ModifiedEvents StateStore is updated according to the results of the KeyValueMapper. Old values are deleted as required.
2. The updated ModifiedEvent row is sent downstream to the Join processor.
3. The prefix is stripped from the ModifiedEvent row, and the GKT is queried.
4. The GKT result is returned to the processor.
5. The processor joins the data together depending on the left/inner logic. The subsequent events are output from the processor to the downstream consumers.


KTable - Events
1) Row is updated in KTable. KV is mapped using the KeyValueMapper and written to downstream state store
KeyValueMapper
StateStore - ModifiedEvents
2) a_1 is sent downstream
GlobalKTable
3) GKT StateStore Get(a)
Join
4) Result
5) Output Join Result

KTable - Events

Key	Value
1	(foo=a, x=99)
2	(foo=a, x=120)
3	(foo=b, x=20)
...	
89 999 999	(foo=a, x=12)
90 000 000	(foo=b, x=12)

1) Row is updated in KTable. KV is mapped using the KeyValueMapper and written to downstream state store

old	1	null
new	1	(foo=a, x=99)

StateStore - ModifiedEvents

Key	Value
a_1	(foo=a, x=99)
a_2	(foo=a, x=120)
b_3	(foo=b, x=20)
...	
a_89 999 999	(foo=a, x=12)
b_90 000 000	(foo=b, x=12)

2) a_1 is sent downstream

a_1	(foo=a, x=99)

GlobalKTable

K	Value
a	(dots: 10, lines: 2)
b	(dots: 0, lines: 4)

4) Result

old	a	null
new	a	(dots: 10, lines: 2)

5) Output Join Result

old	1	null
new	1	(foo=a, x=99), (dots: 10, lines: 2)

Summary

In terms of major changes, this relies heavily on the RocksDB prefix scan, in both consistency and performance. This also relies heavily on changing the GKT to be a driver of processor join logic. In terms of data complexity, any pattern that requires us to rekey the data once is equivalent in terms of data capacity requirements, regardless of if we rekey the data to have a prefix, or rekey it such that all elements of a foreign-key are in the same row.

The mains advantages of the proposed solution here is that:

- Each row in the ModifiedEvents table is just slightly bigger than the source table row, due to the addition of the key prefix. This is contrary to the original problem, where a groupByKey on a rekeyed KTable mandates that each element with the same key must be grouped into the same row, thereby ensuring that the row size is unbounded. This risks OOM errors, difficulty in maintaining the updates over time, and an inability to back data up to the Kafka cluster when rows grow into the 10s, 100s and 1GB range.

- There is no need for re-partitioning. We want the data to remain local to each node, as only the final result is what matters. This reduces the load on the Kafka cluster and reduces both financial and processing costs.

- Updates to the GKT can be propagated out to all keyed entities that use that data. This is highly valuable in providing a way to depart from the relational structures of many change-data-capture produced events.

- Simplified logic. Having implemented solutions that follow the logic already outlined in http://danlebrero.com/2017/05/07/kafka-streams-ktable-globalktable-joining-reference-data/, I have to say that it is extremely error prone and inefficient, and having a simplification to this pattern would be very appreciated.

Compatibility, Deprecation, and Migration Plan

There should be no impact to any current users, nor any change to existing join functionality.

The only component that will require a closer look is the usage of the GlobalKTable as a processor driver. Currently, the GKT is only usable as a lookup and will not drive join logic. Aside from wishing to avoid ill-defined behaviour, I can't see any technical reasons why we cannot do this. My familiarity with this component and the history behind it is minimal though, as this is the first KIP and JIRA that I would be addressing in Kafka.

Rejected Alternatives

None currently known or described.