KIP-213 Support non-key joining in KTable

Status

Current state: *Accepted*
Discussion thread: *here*
JIRA: *KAFKA-3705* Please keep the discussion on the mailing list rather than commenting on the wiki (wiki discussions get unwieldy fast).

Note: This KIP was previously worked on by Jan. The current proposal is at the top, with Jan's portion preserved at the end of the document.

Motivation

Close the gap between the semantics of KTables in streams and tables in relational databases. It is common practice to capture changes as they are made to tables in a RDBMS into Kafka topics (JDBC-connect, Debezium, Maxwell). These entities typically have multiple one-to-many relationship. Usually RDBMSs offer good support to resolve this relationship with a join. Streams falls short here and the workaround (group by - join - lateral view) is not well supported as well and is not in line with the idea of record based processing.

This KIP makes relational data liberated by connection mechanisms far easier for teams to use, smoothing a transition to natively-built event-driven services.

Features

1. Perform KTable-to-KTable updates from both sides of the join
2. Resolves out-of-order processing due to foreignKey changes
3. Scalable as per normal joins

Public Interfaces

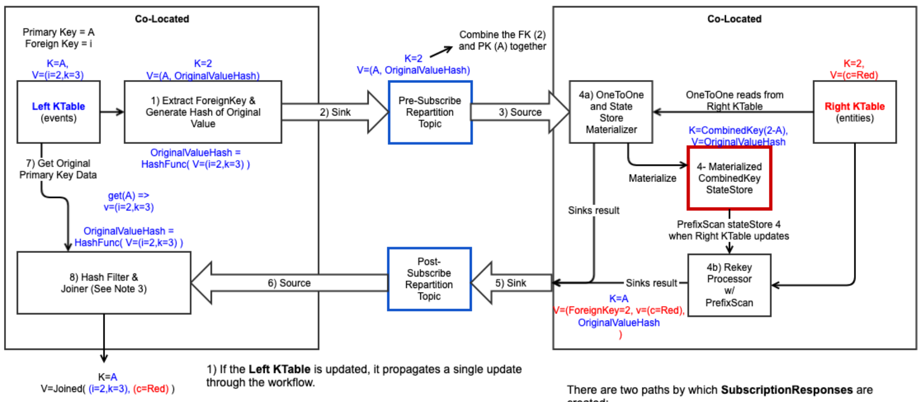
```
    /**
     * Join records of this {@code KTable} with another {@code KTable} using non-windowed inner join.
     * <p>
     * This is a foreign key join, where the joining key is determined by the {@code foreignKeyExtractor}.
     *
     * @param other               the other {@code KTable} to be joined with this {@code KTable}. Keyed by KO.
     * @param foreignKeyExtractor a {@link Function} that extracts the key (KO) from this table's value (V). If
the
     *                            result is null, the update is ignored as invalid.
     * @param joiner              a {@link ValueJoiner} that computes the join result for a pair of matching
records
     * @param <VR>                the value type of the result {@code KTable}
     * @param <KO>                the key type of the other {@code KTable}
     * @param <VO>                the value type of the other {@code KTable}
     * @return a {@code KTable} that contains the result of joining this table with {@code other}
     */
    <VR, KO, VO> KTable<K, VR> join(final KTable<KO, VO> other,
                                    final Function<V, KO> foreignKeyExtractor,
                                    final ValueJoiner<V, VO, VR> joiner);

    /**
     * Join records of this {@code KTable} with another {@code KTable} using non-windowed inner join.
     * <p>
     * This is a foreign key join, where the joining key is determined by the {@code foreignKeyExtractor}.
     *
     * @param other               the other {@code KTable} to be joined with this {@code KTable}. Keyed by KO.
     * @param foreignKeyExtractor a {@link Function} that extracts the key (KO) from this table's value (V). If
the
     *                            result is null, the update is ignored as invalid.
     * @param joiner              a {@link ValueJoiner} that computes the join result for a pair of matching
records
     * @param named               a {@link Named} config used to name the processor in the topology
     * @param <VR>                the value type of the result {@code KTable}
     * @param <KO>                the key type of the other {@code KTable}
     * @param <VO>                the value type of the other {@code KTable}
     * @return a {@code KTable} that contains the result of joining this table with {@code other}
     */
    <VR, KO, VO> KTable<K, VR> join(final KTable<KO, VO> other,
                                    final Function<V, KO> foreignKeyExtractor,
                                    final ValueJoiner<V, VO, VR> joiner,
                                    final Named named);

    /**
     * Join records of this {@code KTable} with another {@code KTable} using non-windowed inner join.
     * <p>
     * This is a foreign key join, where the joining key is determined by the {@code foreignKeyExtractor}.
     *
     * @param other               the other {@code KTable} to be joined with this {@code KTable}. Keyed by KO.
     * @param foreignKeyExtractor a {@link Function} that extracts the key (KO) from this table's value (V). If
the
     *                            result is null, the update is ignored as invalid.
     * @param joiner              a {@link ValueJoiner} that computes the join result for a pair of matching
records
     * @param materialized        a {@link Materialized} that describes how the {@link StateStore} for the
resulting {@code KTable}
     *                            should be materialized. Cannot be {@code null}
     * @param <VR>                the value type of the result {@code KTable}
     * @param <KO>                the key type of the other {@code KTable}
     * @param <VO>                the value type of the other {@code KTable}
     * @return a {@code KTable} that contains the result of joining this table with {@code other}
     */
    <VR, KO, VO> KTable<K, VR> join(final KTable<KO, VO> other,
                                    final Function<V, KO> foreignKeyExtractor,
                                    final ValueJoiner<V, VO, VR> joiner,
                                    final Materialized<K, VR, KeyValueStore<Bytes, byte[]>> materialized);

    /**
     * Join records of this {@code KTable} with another {@code KTable} using non-windowed inner join.
     * <p>
     * This is a foreign key join, where the joining key is determined by the {@code foreignKeyExtractor}.
     *
     * @param other               the other {@code KTable} to be joined with this {@code KTable}. Keyed by KO.
     * @param foreignKeyExtractor a {@link Function} that extracts the key (KO) from this table's value (V). If
the
     *                            result is null, the update is ignored as invalid.
     * @param joiner              a {@link ValueJoiner} that computes the join result for a pair of matching
```

```
records
     * @param named                a {@link Named} config used to name the processor in the topology
     * @param materialized         a {@link Materialized} that describes how the {@link StateStore} for the
resulting {@code KTable}
     *                             should be materialized. Cannot be {@code null}
     * @param <VR>                 the value type of the result {@code KTable}
     * @param <KO>                 the key type of the other {@code KTable}
     * @param <VO>                 the value type of the other {@code KTable}
     * @return a {@code KTable} that contains the result of joining this table with {@code other}
     */
    <VR, KO, VO> KTable<K, VR> join(final KTable<KO, VO> other,
                                    final Function<V, KO> foreignKeyExtractor,
                                    final ValueJoiner<V, VO, VR> joiner,
                                    final Named named,
                                    final Materialized<K, VR, KeyValueStore<Bytes, byte[]>> materialized);

    /**
     * Join records of this {@code KTable} with another {@code KTable} using non-windowed left join.
     * <p>
     * This is a foreign key join, where the joining key is determined by the {@code foreignKeyExtractor}.
     *
     * @param other                the other {@code KTable} to be joined with this {@code KTable}. Keyed by KO.
     * @param foreignKeyExtractor a {@link Function} that extracts the key (KO) from this table's value (V). If
the
     *                             result is null, the update is ignored as invalid.
     * @param joiner               a {@link ValueJoiner} that computes the join result for a pair of matching
records
     * @param <VR>                 the value type of the result {@code KTable}
     * @param <KO>                 the key type of the other {@code KTable}
     * @param <VO>                 the value type of the other {@code KTable}
     * @return a {@code KTable} that contains only those records that satisfy the given predicate
     */
    <VR, KO, VO> KTable<K, VR> leftJoin(final KTable<KO, VO> other,
                                        final Function<V, KO> foreignKeyExtractor,
                                        final ValueJoiner<V, VO, VR> joiner);

    /**
     * Join records of this {@code KTable} with another {@code KTable} using non-windowed left join.
     * <p>
     * This is a foreign key join, where the joining key is determined by the {@code foreignKeyExtractor}.
     *
     * @param other                the other {@code KTable} to be joined with this {@code KTable}. Keyed by KO.
     * @param foreignKeyExtractor a {@link Function} that extracts the key (KO) from this table's value (V) If
the
     *                             result is null, the update is ignored as invalid.
     * @param joiner               a {@link ValueJoiner} that computes the join result for a pair of matching
records
     * @param named                a {@link Named} config used to name the processor in the topology
     * @param <VR>                 the value type of the result {@code KTable}
     * @param <KO>                 the key type of the other {@code KTable}
     * @param <VO>                 the value type of the other {@code KTable}
     * @return a {@code KTable} that contains the result of joining this table with {@code other}
     */
    <VR, KO, VO> KTable<K, VR> leftJoin(final KTable<KO, VO> other,
                                        final Function<V, KO> foreignKeyExtractor,
                                        final ValueJoiner<V, VO, VR> joiner,
                                        final Named named);

    /**
     * Join records of this {@code KTable} with another {@code KTable} using non-windowed left join.
     * <p>
     * This is a foreign key join, where the joining key is determined by the {@code foreignKeyExtractor}.
     *
     * @param other                the other {@code KTable} to be joined with this {@code KTable}. Keyed by KO.
     * @param foreignKeyExtractor a {@link Function} that extracts the key (KO) from this table's value (V). If
the
     *                             result is null, the update is ignored as invalid.
     * @param joiner               a {@link ValueJoiner} that computes the join result for a pair of matching
records
     * @param materialized         a {@link Materialized} that describes how the {@link StateStore} for the
resulting {@code KTable}
```

```
     *                              should be materialized. Cannot be {@code null}
     * @param <VR>                  the value type of the result {@code KTable}
     * @param <KO>                  the key type of the other {@code KTable}
     * @param <VO>                  the value type of the other {@code KTable}
     * @return a {@code KTable} that contains the result of joining this table with {@code other}
     */
    <VR, KO, VO> KTable<K, VR> leftJoin(final KTable<KO, VO> other,
                                        final Function<V, KO> foreignKeyExtractor,
                                        final ValueJoiner<V, VO, VR> joiner,
                                        final Materialized<K, VR, KeyValueStore<Bytes, byte[]>> materialized);

    /**
     * Join records of this {@code KTable} with another {@code KTable} using non-windowed left join.
     * <p>
     * This is a foreign key join, where the joining key is determined by the {@code foreignKeyExtractor}.
     *
     * @param other                 the other {@code KTable} to be joined with this {@code KTable}. Keyed by KO.
     * @param foreignKeyExtractor a {@link Function} that extracts the key (KO) from this table's value (V) If
the
     *                              result is null, the update is ignored as invalid.
     * @param joiner                a {@link ValueJoiner} that computes the join result for a pair of matching
records
     * @param named                 a {@link Named} config used to name the processor in the topology
     * @param materialized          a {@link Materialized} that describes how the {@link StateStore} for the
resulting {@code KTable}
     *                              should be materialized. Cannot be {@code null}
     * @param <VR>                  the value type of the result {@code KTable}
     * @param <KO>                  the key type of the other {@code KTable}
     * @param <VO>                  the value type of the other {@code KTable}
     * @return a {@code KTable} that contains the result of joining this table with {@code other}
     */
    <VR, KO, VO> KTable<K, VR> leftJoin(final KTable<KO, VO> other,
                                        final Function<V, KO> foreignKeyExtractor,
                                        final ValueJoiner<V, VO, VR> joiner,
                                        final Named named,
                                        final Materialized<K, VR, KeyValueStore<Bytes, byte[]>> materialized);
```

Workflow

The overall process is outlined below.


Co-Located
Primary Key = A
Foreign Key = i
K=A,
V=(i=2,k=3)
Left KTable
(events)
K=2
V=(A, OriginalValueHash)
1) Extract ForeignKey & Generate Hash of Original Value
OriginalValueHash =
HashFunc(V=(i=2,k=3))
7) Get Original Primary Key Data
get(A) =>
v=(i=2,k=3)
OriginalValueHash =
HashFunc(V=(i=2,k=3))
8) Hash Filter & Joiner (See Note 3)
K=A
V=Joined((i=2,k=3), (c=Red))
2) Sink
Combine the FK (2) and PK (A) together
K=2
V=(A, OriginalValueHash)
Pre-Subscribe Repartition Topic
3) Source
6) Source
Post-Subscribe Repartition Topic
5) Sink
Co-Located
4a) OneToOne and State Store Materializer
OneToOne reads from Right KTable
K=2,
V=(c=Red)
Right KTable
(entities)
K=CombinedKey(2-A),
V=OriginalValueHash
Materialize
4- Materialized CombinedKey StateStore
Sinks result
PrefixScan stateStore 4 when Right KTable updates
4b) Rekey Processor w/ PrefixScan
Sinks result
K=A
V=(ForeignKey=2, v=(c=Red),
OriginalValueHash
)

1) If the **Left KTable** is updated, it propagates a single update through the workflow.

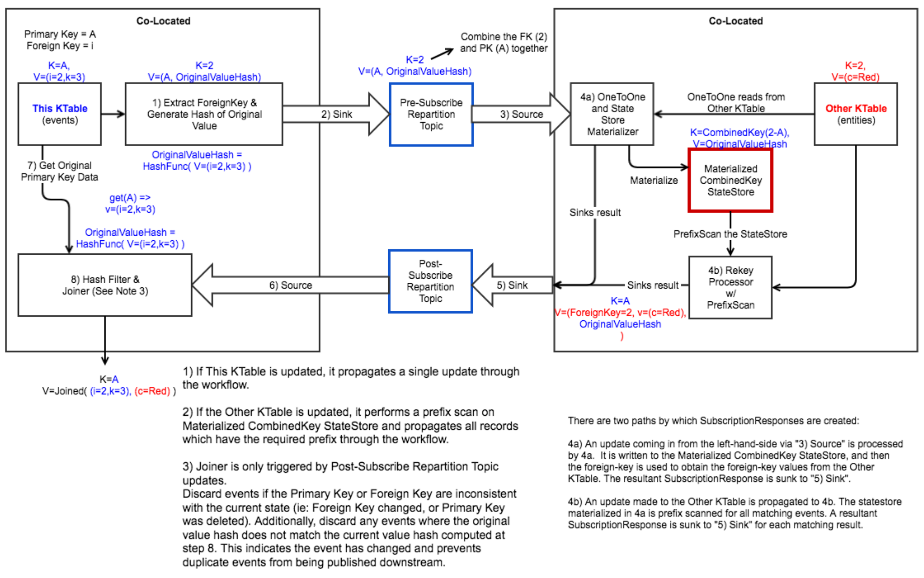
2) If the **Right KTable** is updated, it performs a prefixScan on **Materialized CombinedKey StateStore** and propagates all records which have the required prefix through the workflow.

3) **Joiner (8)** is only triggered by **Post-Subscribe Repartition Topic** updates.
It discards events if the Primary Key or Foreign Key are inconsistent with the current state (ie: Foreign Key changed, or Primary Key was deleted). Additionally, it will discard any event where the original value hash does not match the current hash value. This indicates the event has changed and prevents inconsistent output.

There are two paths by which **SubscriptionResponses** are created:

4a) An update coming in from the left-hand-side via **3) Source** is processed by **4a**. It is written to the **Materialized CombinedKey StateStore.** Next, the foreign-key is used to obtain the key-values from the **Right KTable**. The resultant **SubscriptionResponse** is sunk to **5) Sink**.

4b) An update made to the **Right KTable** is propagated to **4b**. **Materialized CombinedKey StateStore** is **prefix scanned** for all matching events. A **SubscriptionResponse** is sunk to **5) Sink**" for each matching record result.


Co-Located
Primary Key = A
Foreign Key = i
K=A,
V=(i=2,k=3)
This KTable
(events)
K=2
V=(A, OriginalValueHash)
1) Extract ForeignKey & Generate Hash of Original Value
OriginalValueHash =
HashFunc(V=(i=2,k=3))
2) Sink
K=2
V=(A, OriginalValueHash)
Combine the FK (2) and PK (A) together
Pre-Subscribe Repartition Topic
3) Source
7) Get Original Primary Key Data
get(A) =>
v=(i=2,k=3)
OriginalValueHash =
HashFunc(V=(i=2,k=3))
8) Hash Filter & Joiner (See Note 3)
6) Source
Post-Subscribe Repartition Topic
5) Sink
K=A
V=Joined((i=2,k=3), (c=Red))
Co-Located
4a) OneToOne and State Store Materializer
OneToOne reads from Other KTable
K=2,
V=(c=Red)
Other KTable
(entities)
K=CombinedKey(2-A),
V=OriginalValueHash
Materialize
Materialized CombinedKey StateStore
Sinks result
PrefixScan the StateStore
4b) Rekey Processor w/ PrefixScan
Sinks result
K=A
V=(ForeignKey=2, v=(c=Red),
OriginalValueHash
)

1) If This KTable is updated, it propagates a single update through the workflow.

2) If the Other KTable is updated, it performs a prefix scan on Materialized CombinedKey StateStore and propagates all records which have the required prefix through the workflow.

3) Joiner is only triggered by Post-Subscribe Repartition Topic updates.
Discard events if the Primary Key or Foreign Key are inconsistent with the current state (ie: Foreign Key changed, or Primary Key was deleted). Additionally, discard any events where the original value hash does not match the current value hash computed at step 8. This indicates the event has changed and prevents duplicate events from being published downstream.

There are two paths by which SubscriptionResponses are created:

4a) An update coming in from the left-hand-side via "3) Source" is processed by 4a. It is written to the Materialized CombinedKey StateStore, and then the foreign-key is used to obtain the foreign-key values from the Other KTable. The resultant SubscriptionResponse is sunk to "5) Sink".

4b) An update made to the Other KTable is propagated to 4b. The statestore materialized in 4a is prefix scanned for all matching events. A resultant SubscriptionResponse is sunk to "5) Sink" for each matching result.

- One optimization possibility is to materialize the data provided to the original node at Step 6. This would allow for changes to an event in This KTable which does not change the foreign key to shortcut sending an event to the Other node. The tradeoff is the need to maintain a materialized state store.

More Detailed Implementation Details

Prefix-Scanning using a CombinedKey

CombinedKey is a simple tuple wrapper that stores the primary key and the foreign key together. The CombinedKey serde requires the usage of both the primary key and foreign key serdes. The CombinedKey is serialized as follows:

```
{4-byte foreignKeyLength}{foreignKeySerialized}{primaryKeySerialized}
```

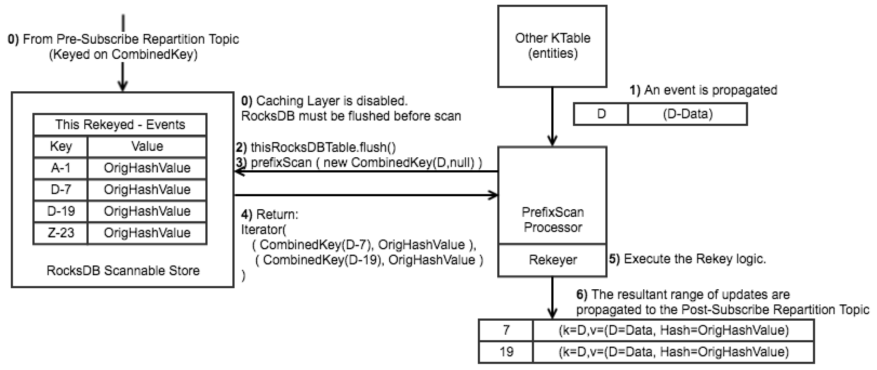
This can support up to MAXINT size of data in bytes per key, which currently far exceeds the realistic sizing of messages in Kafka.

When performing the prefix scan on the RocksDB instance, we simply drop the primary key component, such that the serialized combined key looks like:

```
{4-byte foreignKeyLength}{foreignKeySerialized}
```

Other/Entity PrefixScan Processor Behaviour

- Requires the specific serialized format detailed above
- Requires a RocksDB instance storing all of the This/Event data


0) From Pre-Subscribe Repartition Topic
(Keyed on CombinedKey)
RocksDB Scannable Store
Other KTable
(entities)
1) An event is propagated
PrefixScan
Processor
Rekeyer
5) Execute the Rekey logic.
6) The resultant range of updates are propagated to the Post-Subscribe Repartition Topic

This Rekeyed - Events

Key	Value
A-1	OrigHashValue
D-7	OrigHashValue
D-19	OrigHashValue
Z-23	OrigHashValue

D	(D-Data)

0) Caching Layer is disabled.
RocksDB must be flushed before scan

2) thisRocksDBTable.flush()
3) prefixScan (new CombinedKey(D,null))

4) Return:
Iterator(
(CombinedKey(D-7), OrigHashValue),
(CombinedKey(D-19), OrigHashValue)
)

7	(k=D,v=(D=Data, Hash=OrigHashValue)
19	(k=D,v=(D=Data, Hash=OrigHashValue)

Joiner Propagation of Stale Data

Rapid changes to a foreign Key (perhaps due capturing changes from a relational database undergoing such operations) can cause a race condition. The other/Entity foreign keyed table may be quite congested with subscription requests on one node, and lightly subscribed on another due to the nature of the key distribution. As such, there is no guarantee the order in which the subscription updates will arrive in the post-subscribe repartition topic. It is for this reason why we must compare the current state of the primary key event to ensure that the data is still consistent with the extracted foreign key.

While it is possible for a stale event to be propagated (ie: matches the foreign key, but is stale), the up-to-date event will be propagated when it eventually arrives. Entity changes do not cause the same race conditions that event-changes do, and so are not of a concern.

Original Value Hash

A hash value is computed for each message propagated from This KTable. It is passed through the entire process and returns to the co-located KTable in the final step. Before joining, a lookup is done to validate that the hash value is identical. If the hash is not identical, this indicates that the event has since changed, even if the primary and foreign key remain the same. This indicates that the value should be discarded, lest a duplicate be printed.

Example:

```
Start with two empty tables:
LHS is This KTable
RHS is Other KTable
1- RHS is updated to Y|bar
2- LHS is updated to A|Y,1
   -> sends Y|A+ subscription message to RHS
3- LHS is updated to A|Y,2
   -> sends Y|A- unsubscribe message to RHS
   -> sends Y|A+ subscription to RHS
4- RHS processes first Y|A+ message
   -> sends A|Y,bar back
5- LHS processes A|Y,bar and produces result record A|Y,2,bar.
6- RHS processes Y|A- unsubscribe message (update store only)
7- RHS processes second Y|A+ subscribe message
   -> sends A|Y,bar back
8- LHS processes A|Y,bar and produces result record A|Y,2,bar
Thus, the first result record, that should have been `A|Y,1,bar`, is now
`A|Y,2,bar`. Furthermore, we update result table from `A|Y,2,bar` to
`A|Y,2,bar`.
By
 using a hash function to compare in step 5, we can instead see that the
 message is stale and should be discarded, lest we produce double output
 (or more, in the case of a rapidly changing set of events).
```

Tombstones & Foreign Key Changes

In the event of a deletion on the LHS, a tombstone record is sent to the RHS. This will see the state deleted in the RHS state store, and the null will be propagated back to the LHS via the Post-Subscribe Repartition Topic.

```
1) LHS sends (FK-Key, null) to RHS
2) RHS deletes Key from state store
3) RHS sends (Key, null) back to LHS
4) LHS validates Key is still null in LHS-state store, propagates tombstone downstream
```

Each event from the LHS has a set of instructions sent with it to the RHS. These instructions are used to help manage the resolution of state.

Changing LHS (Key, FK-1) to (Key, FK-2).

```
1) LHS sends (CombinedKey(FK-1,Key), SubscriptionWrapper(value=null, instruction=DELETE_KEY_NO_PROPAGATE)) to
RHS-1
2) LHS sends (CombinedKey(FK-2,Key), SubscriptionWrapper(value=NewValueHash,
instruction=PROPAGATE_NULL_IF_NO_FK_VAL_AVAILABLE)) to RHS-2
3) RHS-1 deletes CombinedKey(FK-1,Key) from state store, but does not propagate any other event.
4) RHS-2 updates the local Key-Value store with (CombinedKey(FK-2,Key), NewValueHash).
5) RHS-2 looks up FK-2 in the RHS-2 materialized state store:
   a) If a non-null result is obtained, RHS-2 propagates to LHS: (Key, SubscriptionResponseWrapper
(NewValueHash, RHS-Result, propagateIfNull=false))
   b) If a null result is obtained, RHS-2 propagates to LHS: (Key, SubscriptionResponseWrapper(NewValueHash,
null, propagateIfNull=true))
      - This is done to ensure that the old join results are wiped out, since that join result is now stale
/incorrect.
6) LHS validates that the NewValueHash from the SubscriptionResponseWrapper matches the hash of the current key
in the LHS table.
   a) If the hash doesn't match, discard the event and return.
   b) If the hash does match, proceed to next step.
7) LHS checks if RHS-result == null and propagateIfNull is true.
   a) If yes, then propagate out a (Key, null) and return
   b) If no, proceed to next step.
8) Reminder: RHS-result is not null, and NewValueHash is valid and current. LHS performs the join logic on (LHS-
Value and RHS-Result), and outputs the resultant event (Key, joinResult(LHS-Value, RHS-Result))
```

The workflow of LHS-generated changes to outputs is shown below. Each step is cumulative with the previous step. LEFT and INNER join outputs are shown below.

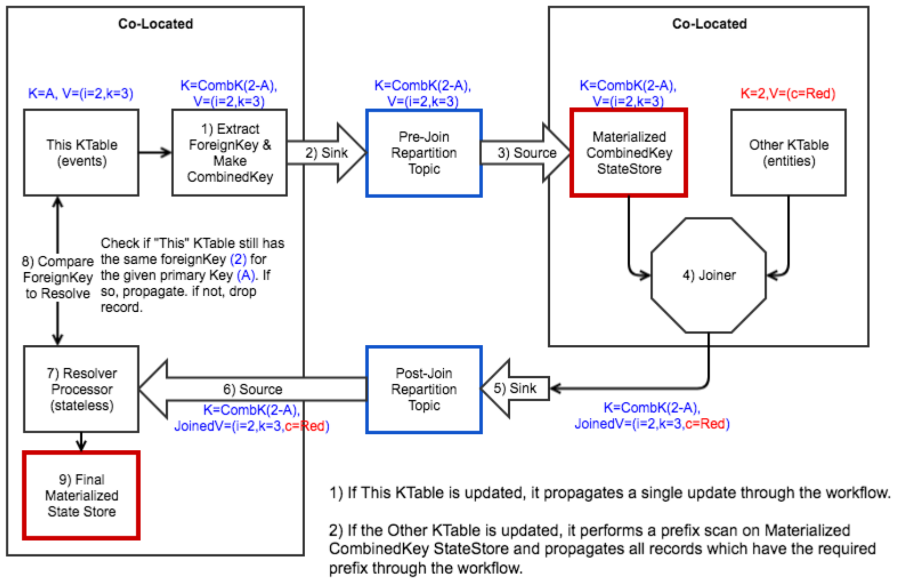
	LHS Event (key, extracted fk)	**To which RHS-partition?**	**RHS-0 State**	**RHS-1 State**	**Inner Join Output**	**Left Join Output**	**Execute Join Logic?**	**Notes**	**Inner-Join SubscriptionWrapper Instruction**
Publish new event	(k,1)	RHS-0	(1,foo)		(k,1,foo)	(k,1,foo)	Inner/Left	Normal fk-join induced by LHS event	to RHS-0: PROPAGATE_ONLY_IF_FK_VAL_AVAILABLE
Publish update to event by changing fk	(k,1) (k,2)	RHS-1	(1,foo)		(k,null)	(k,2,null)	**LEFT**	Must indicate a delete because there is currently no (fk,value) in RHS with key=2, and (k,1,foo) is no longer valid output.	to RHS-0: DELETE_KEY_NO_PROPAGATE to RHS-1: PROPAGATE_NULL_IF_NO_FK_VAL_AVAILABLE
Publish update to event by changing fk	(k,2) (k,3)	RHS-0	(1,foo)		**(k,null)**	(k,3,null)	**LEFT**	**Ideally would not publish a delete with Inner Join,** but we do not maintain sufficient state to know that the (k,2) update resulted in a null output and we don't need to do it again.	to RHS-0: DELETE_KEY_NO_PROPAGATE to RHS-1: PROPAGATE_NULL_IF_NO_FK_VAL_AVAILABLE
Publish a value to RHS-0	-	-	(1,foo) (3,bar)		(k,3,bar)	(k,3,bar)	Inner/Left	Performs prefix scan join	-
Delete k	(k,3) (k,null)	RHS-0	(1,foo) (3,bar)		(k,null)	(k,null,null)	**LEFT**	Propagate null/delete through the sub-topology	to RHS-0: DELETE_KEY_AND_PROPAGATE
Publish original event again	(k,null) (k,1)	RHS-0	(1,foo) (3,bar)		(k,1,foo)	(k,1,foo)	Inner/Left	Normal fk-join induced by LHS event	to RHS-0: PROPAGATE_ONLY_IF_FK_VAL_AVAILABLE
Publish to LHS	(q,10)	RHS-1	(1,foo) (3,bar)		Nothing	**(q,null,10)**	**LEFT**	Significant difference between Inner and Outer	-
Publish a value to RHS-1	-	-	(1,foo) (3,bar)	(q,baz)	(q,10,baz)	(q,10,baz)	Inner/Left	Normal fk-join induced by LHS event	to RHS-1: PROPAGATE_ONLY_IF_FK_VAL_AVAILABLE

Compatibility, Deprecation, and Migration Plan

- *There is no impact to existing users.*

Rejected Alternatives:

Shipping the whole payload across and having the Foreign-node perform the join.


Co-Located
Co-Located
K=A, V=(i=2,k=3)
K=CombK(2-A), V=(i=2,k=3)
K=CombK(2-A), V=(i=2,k=3)
K=CombK(2-A), V=(i=2,k=3)
K=2,V=(c=Red)
This KTable (events)
1) Extract ForeignKey & Make CombinedKey
2) Sink
Pre-Join Repartition Topic
3) Source
Materialized CombinedKey StateStore
Other KTable (entities)
4) Joiner
8) Compare ForeignKey to Resolve
Check if "This" KTable still has the same foreignKey (2) for the given primary Key (A). If so, propagate. if not, drop record.
7) Resolver Processor (stateless)
6) Source
K=CombK(2-A), JoinedV=(i=2,k=3,c=Red)
Post-Join Repartition Topic
5) Sink
K=CombK(2-A), JoinedV=(i=2,k=3,c=Red)
9) Final Materialized State Store

1) If This KTable is updated, it propagates a single update through the workflow.

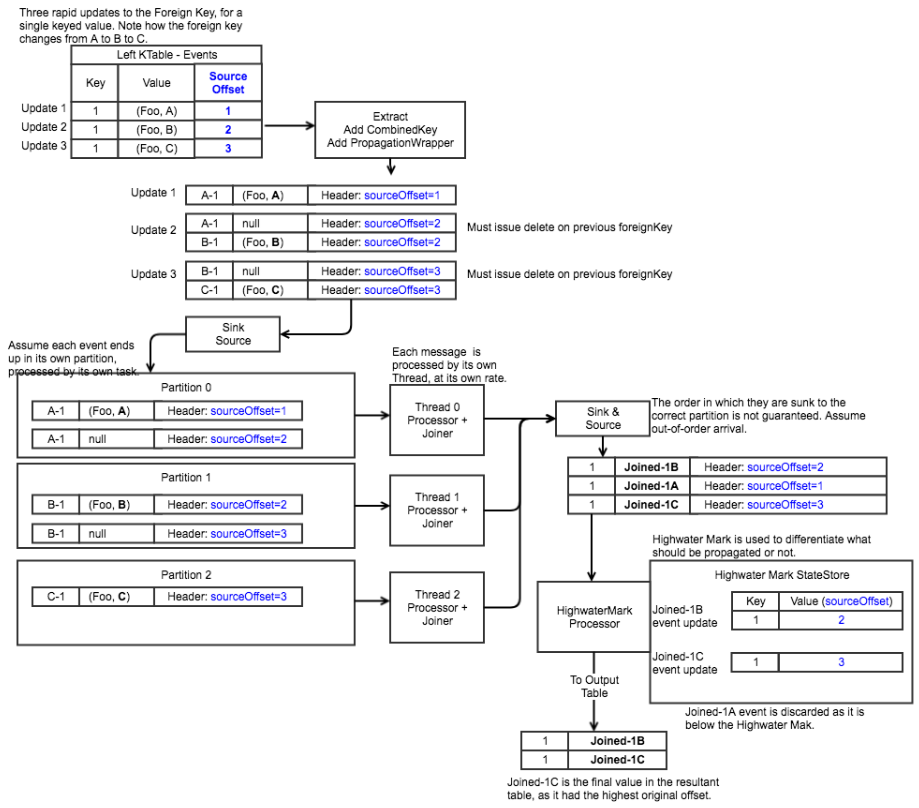
2) If the Other KTable is updated, it performs a prefix scan on Materialized CombinedKey StateStore and propagates all records which have the required prefix through the workflow.

Rejected Because:

- Transfers the maximum amount of data across a network
- Joining on the foreign node can be a bottle-neck vs. joining on the local node.

Problem: Out-of-order processing of Rekeyed data

Solution A - Hold Ordering Metadata in Record Headers and Highwater Mark Table


Three rapid updates to the Foreign Key, for a single keyed value. Note how the foreign key changes from A to B to C.
Left KTable - Events
Key
Value
Source Offset
Update 1
Update 2
Update 3
1
(Foo, A)
(Foo, B)
(Foo, C)
2
3
Extract
Add CombinedKey
Add PropagationWrapper
A-1
B-1
C-1
null
Header: sourceOffset=1
Header: sourceOffset=2
Header: sourceOffset=3
Must issue delete on previous foreignKey
Sink
Source
Assume each event ends up in its own partition, processed by its own task.
Partition 0
Partition 1
Partition 2
Each message is processed by its own Thread, at its own rate.
Thread 0 Processor + Joiner
Thread 1 Processor + Joiner
Thread 2 Processor + Joiner
Sink & Source
The order in which they are sunk to the correct partition is not guaranteed. Assume out-of-order arrival.
Joined-1B
Joined-1A
Joined-1C
Highwater Mark is used to differentiate what should be propagated or not.
Highwater Mark StateStore
Value (sourceOffset)
Joined-1B event update
Joined-1C event update
HighwaterMark Processor
To Output Table
Joined-1A event is discarded as it is below the Highwater Mak.
Joined-1C is the final value in the resultant table, as it had the highest original offset.

Rejected because:

Input Tables generated by:

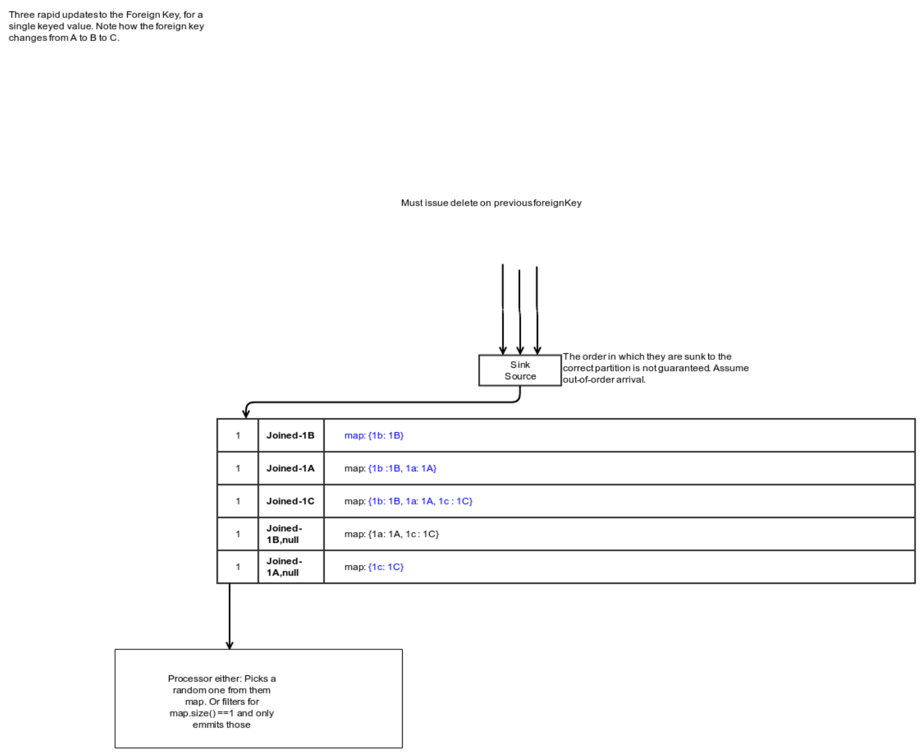
```
stream.flatMapValues().groupByKey().aggregate()
```

In this case, multiple KTable updates have the same input record and thus the same offset. Hence, there is no guarantee that offsets are unique and thus we cannot use them to resolve update conflicts.

Solution B - User-Managed GroupBy (Jan's)

A Table KTable<CombinedKey<A,B>,JoinedResult> is not a good return type. It breaks the KTable invariant that a table is currently partitioned by its key, which this table wouldn't be and the CombinedKey is not particularly useful as its a mere Kafka artifact.

With a followed up group by, we can remove the repartitioning artifact by grouping into a map. Out of order events can be hold in the map and can be dealt with, however one likes it. Either wait for some final state and propagate no changes that are "intermediate" and show artifacts or propagate directly. The eventual correctness is guaranteed in both ways. The huge advantage is further, that the group by can be by any key, resulting in a table of that key.


Three rapid updates to the Foreign Key, for a single keyed value. Note how the foreign key changes from A to B to C.
Must issue delete on previous foreignKey
Sink
Source
The order in which they are sunk to the correct partition is not guaranteed. Assume out-of-order arrival.
1 | Joined-1B | map: {1b: 1B}
1 | Joined-1A | map: {1b :1B, 1a: 1A}
1 | Joined-1C | map: {1b: 1B, 1a: 1A, 1c : 1C}
1 | Joined-1B,null | map: {1a: 1A, 1c : 1C}
1 | Joined-1A,null | map: {1c: 1C}
Processor either: Picks a random one from them map. Or filters for map.size() ==1 and only emmits those

Jan Filipiak's Original Proposal (From here to end of document)

Public Interfaces

Less intrusive

We would introduce a new Method into KTable and KTableImpl

KTable.java

```
   /**
     *
     * Joins one record of this KTable to n records of the other KTable,
     * an update in this KTable will update all n matching records, an update
     * in other table will update only the one matching record.
     *
     * @param the table containing n records for each K of this table
     * @param keyExtractor a {@code ValueMapper} returning the key of this table from the others value
     * @param joinPrefixFaker a {@code ValueMapper} returning an outputkey that when serialized only produces
the
     *                             prefix of the output key which is the same as serializing K
     * @param leftKeyExtractor a {@code ValueMapper} extracting the Key of this table from the resulting Key
     * @param <KO> the resultings tables Key
     * @param <VO> the resultings tables Value
     * @param joiner
     * @return
     */
    <KO, VO, K1, V1> KTable<KO, VO> oneToManyJoin(KTable<K1, V1> other,
            ValueMapper<V1, K> keyExtractor,
            ValueMapper<K, KO> joinPrefixFaker,
            ValueMapper<KO, K> leftKeyExtractor,
            ValueJoiner<V, V1, VO> joiner,
            Serde<K1> keyOtherSerde, Serde<V1> valueOtherSerde,
            Serde<KO> joinKeySerde, Serde<VO> joinValueSerde);
```

More intrusive

We would Introduce a new Complex type: Combined Key

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

public class CombinedKey<P,S> {
    public P prefix;
    public S suffix;
}
```

The method could be rewritten

KTable.java

```
   /**
     *
     * Joins one record of this KTable to n records of the other KTable,
     * an update in this KTable will update all n matching records, an update
     * in other table will update only the one matching record.
     *
     * @param the table containing n records for each K of this table
     * @param keyExtractor a {@code ValueMapper} returning the key of this table from the others value
     * @param leftKeyExtractor a {@code ValueMapper} extracting the Key of this table from the resulting Key
     * @param <VO> the resultings tables Value
     * @param joiner
     * @return
     */
    <VO, K1, V1> KTable<CombinedKey<K,K1>,VO> oneToManyJoin(KTable<K1, V1> other,
            ValueMapper<V1, K> keyExtractor,
            ValueJoiner<V, V1, VO> joiner,
            Serde<K1> keyOtherSerde, Serde<V1> valueOtherSerde,
            Serde<VO> joinValueSerde);
```

Tradeoffs

The more intrusive version gives the user better clarity that his resulting KTable is not only keyed by the other table's key but its also keyed by this table's key. So he will be less surprised that in a theoretical later aggregation he might find the same key from the other ktable twice. On the other hand the less intrusive method doesn't need to introduce this wrapper class but let the user handle the need of having both tables keys present in the output key himself. This might lead to a deeper understanding for the user and serdes might be able to pack the data denser. An additional benefit is that the user can stick with his default serde or his standard way of serializing when sinking the data into another topic using for example to() while the CombinedKey would require an additional mapping to what the less intrusive method has.

Back and forth mapper

This is a proposal to get rid of the Type CombinedKey in the return type. We would internally use a Combined key and a Combined Key Serde and apply the mappers only at the processing boundaries (ValueGetterSupplier, context.forward). The data will still be serialized for repartitioning in a way that is specific to Kafka and might prevent users from using their default tooling with these topics.

KTable.java

```
     /**
         *
     * Joins one record of this KTable to n records of the other KTable,
     * an update in this KTable will update all n matching records, an update
     * in other table will update only the one matching record.
     *
     * @param the table containing n records for each K of this table
     * @param keyExtractor a {@code ValueMapper} returning the key of this table from the others value
     * @param customCombinedKey a {@code ValueMapper} allowing the CombinedKey to be wrapped in a custom object
         * @param combinedKey a {@code ValueMapper} allowing to unwrap the custom object again.
     * @param <VO> the resultings tables Value
     * @param joiner
     * @return
     */
    <KO VO, K1, V1> KTable<KO,VO> oneToManyJoin(KTable<K1, V1> other,
            ValueMapper<V1, K> keyExtractor,
                        ValueMapper<CombinedKey<K1,K>,KO> outputKeyCombiner,
                        ValueMapper<KO,CombinedKey<K1,K>> outputKeySpliter,
                        ValueJoiner<V, V1, VO> joiner,
            Serde<K1> keyOtherSerde,
                Serde<V1> valueOtherSerde,
            Serde<VO> joinValueSerde);
```

Custom Serde

Introducing an additional new Serde. This is the approach is the counterpart to having a back and forth mapper. With this approach it is possible to keep any Custom serialization mechanism off the wire. How to serialize is completely with the user.

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

public class CombinedKeySerde<K,K1> extends Serde<CombinedKey<K,K1>> {

        public Serializer<K> getPartialKeySerializer();
 }
```

KTable.java

```
/**
  *
  * Joins one record of this KTable to n records of the other KTable,
  * an update in this KTable will update all n matching records, an update
  * in other table will update only the one matching record.
  *
  * @param the table containing n records for each K of this table
  * @param keyExtractor a {@code ValueMapper} returning the key of this table from the others value
  * @param leftKeyExtractor a {@code ValueMapper} extracting the Key of this table from the resulting Key
  * @param <VO> the resultings tables Value
  * @param joiner
  * @return
  */
 <VO, K1, V1> KTable<CombinedKey<K,K1>,VO> oneToManyJoin(KTable<K1, V1> other,
         ValueMapper<V1, K> keyExtractor,
         ValueJoiner<V, V1, VO> joiner,
         Serde<K1> keyOtherSerde, Serde<V1> valueOtherSerde,
         Serde<VO> joinValueSerde,
                     CombinedKeySerde<K,K1> combinedKeySerde);
```

Streams

We will implement a default CombinedKeySerde that will use a regular length encoding for both fields. So calls to the "intrusive approach" would constuct a default CombinedKeySerde and invoke the Serde Overload. This would work with all serde frameworks if the user is not interested in how the data is serialized in the topics.

Protobuf / Avro / thrift / Hadoop-Writeable / Custom

Users of these frameworks should have a very easy time implementing a CombinedKeySerde. Essentially they define an object that wraps K and K1 as usual keeping K1 as an optional field. The serializer returned from getPartialKeySerializer() would do the following:

1. create such a wrapping object
2. set the value for the K field
3. serialize the wrapped object as usual.

This should work straight forward and users might implement a CombinedKeySerde that is specific to their framework and reuse the logic without implementing a new Serde for each key-pair.

JSON

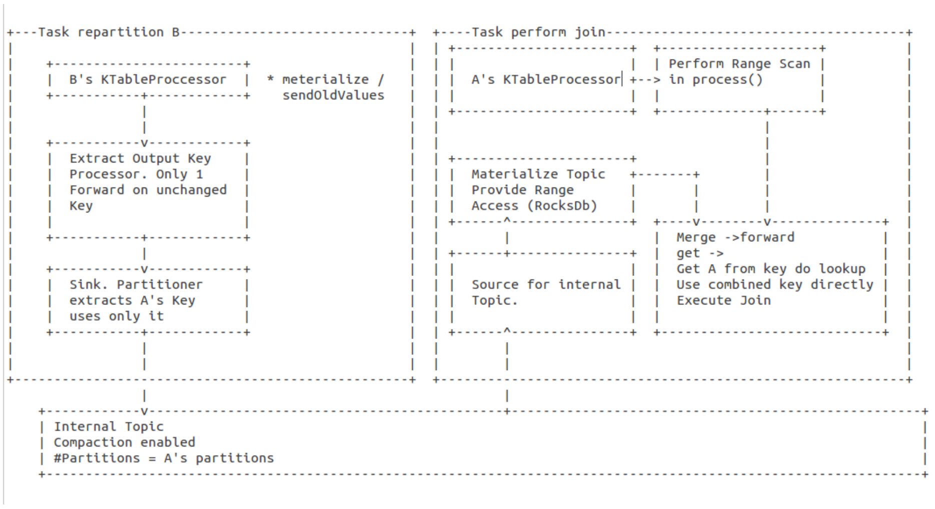
Implementing a CombinedKeySerde depends on the specific framework with json. A full key would look like this "{ "a" :{ "key":"a1" }, "b": {"key":"b5" } }" to generate a true prefix one had to generate "{ "a" :{ "key":"a1"", which is not valid json. This invalid Json will not leave the jvm but it might be more or less tricky to implement a serializer generating it. Maybe we could provide users with a utility method to make sure their serde statisfies our invariants.

Proposed Changes

Goal

With the two relations A,B and there is one A for each B and there may be many B's for each A. A is represented by the KTable the method described above gets invoked on, while B is represented by that methods first argument. We want to implement a Set of processors that allows a user to create a new KTable where A and B are joined based on the reference to A in B. A and B are represented as KTable B being partitioned by B's key and A being partitioned by A's key.

Algorithm


Task repartition B
B's KTableProccessor
* meterialize / sendOldValues
Extract Output Key Processor. Only 1 Forward on unchanged Key
Sink. Partitioner extracts A's Key uses only it
Task perform join
A's KTableProcessor
Perform Range Scan in process()
Materialize Topic Provide Range Access (RocksDb)
Merge ->forward get -> Get A from key do lookup Use combined key directly Execute Join
Source for internal Topic.
Internal Topic Compaction enabled #Partitions = A's partitions

ascii_raw_KIP-213

- Call enable sendOldValues() on sources with "*"
- Register a child of B
  - extract A's key and B's key as key and B as value.
    - forward(key, null) for old
    - forward(key, b) for new
    - skip old if A's key didn't change (ends up in same partition)
- Register sink for internal repartition topic (number of partitions equal to A, if a is internal prefer B over A for deciding number of partitions)
  - in the sink, only use A's key to determine partition
- Register source for intermediate topic
  - co-partition with A's sources
  - materialize
  - serde for rocks needs to serialize A before B. ideally we use the same serde also for the topic
- Register processor after above source.
  - On event extract A's key from the key
  - look up A by it's key
  - perform the join (as usual)
- Register processor after A's processor
  - On event uses A's key to perform a Range scan on B's materialization
  - For every row retrieved perform join as usual
- Register merger
  - Forward join Results
  - On lookup use full key to lookup B and extract A's key from the key and lookup A. Then perform join.
- Merger wrapped into KTable and returned to the user.

Step by Step

TOPOLOGY INPUT A	TOPOLOGY INPUT B	STATE A MATERIALZED	STATE B MATERIALIZE	INTERMEDIATE RECORDS PRODUCED	STATE B OTHER TASK	Output A Source / Input Range Proccesor	OUTPUT RANGE PROCESSOR	OUTPUT LOOKUP PROCESSOR
key: A0 value: [A0 ...]		key: A0 value: [A0 ...]				Change<null,[A0 ...]>	invoked but nothing found. Nothing forwarded	
key: A1 value: [A1 ...]		key: A0 value: [A0 ...] key: A1 value: [A1 ...]				Change<null,[A1 ...]>	invoked but nothing found. Nothing forwarded	
	key: B0 : value [A2,B0 ...]	key: A0 value: [A0 ...] key: A1 value: [A1 ...]	key: B0 : value [A2,B0 ...]	partition key: A2 key: A2B0 value: [A2,B0 ...]	key: A2B0 : value [A2, B0 ...]			invoked but nothing found Nothing forwarded
	key: B1 : value [A2,B1 ...]	key: A0 value: [A0 ...] key: A1 value: [A1 ...]	key: B0 : value [A2,B0 ...] key: B1 : value [A2,B1 ...]	partition key: A2 key: A2B1 value [A2, B1 ...]	key: A2B0 : value [A2, B0 ...] key: A2B1 : value [A2, B1 ...]			invoked but nothing found Nothing forwarded

key: A2 value: [A2 ...]		key: A0 value: [A0 ...] key: A1 value: [A1 ...] key: A2 value: [A2 ...]	key: B0 : value [A2,B0 ...] key: B1 : value [A2,B1 ...]		key: A2B0 : value [A2, B0 ...] key: A2B1 : value [A2, B1 ...]	Change<null,[A2 ...]>	key A2B0 value: Change<null,join([A2 ...],[A2,B0 ...]) key A2B1 value: Change<null,join([A2 ...],[A2,B1...])	
	key: B1 : value null		key: B0 : value [A2,B0 ...]	partition key: A2 key: A2B1 value:null	key: A2B0 : value [A2, B0 ...]			key A2B1 value: Change<join([A2 ...],[A2,B1...],null)
	key: B3 : value [A0,B3 ...]		key: B0 : value [A2,B0 ...] key: B3 : value [A0,B3 ...]	partition key: A0 key: A0B3 value:[A0, B3 ...]	key: A2B0 : value [A2, B0 ...] key: A0B3 : value [A0, B3 ...]			key A0B3 value: Change<join(null,[A0 ...],[A0,B3...])
key: A2 value: null		key: A0 value: [A0 ...] key: A1 value: [A1 ...]	key: B0 : value [A2,B0 ...] key: B3 : value [A0,B3 ...]		key: A2B0 : value [A2, B0 ...] key: A0B3 : value [A0, B3 ...]	Change<[A2 ...],null>	key A2B0 value: Change<join([A2 ...],[A2,B0 ...],null)	

Range lookup

It is pretty straight forward to completely flush all changes that happened before the range lookup into rocksb and let it handle a the range scan. Merging rocksdb's result iterator with current in-heap caches might be not in scope of this initial KIP. Currently we at trivago can not identify the rocksDb flushes to be a performance problem. Usually the amount of emitted records is the harder problem to deal with in the first place.

Missing reference to A

B records with a 'null' A-key value would be silently dropped.

Compatibility, Deprecation, and Migration Plan

- *There is no impact to existing users.*

Rejected Alternatives

If there are alternative ways of accomplishing the same thing, what were they? The purpose of this section is to motivate why the design is the way it is and not some other way.