Kafka Streams Join Semantics

Kafka Streams offers a variety of different join operators with three different types:

- sliding window KStream-KStream join
- KStream-KTable join
- KTable-KTable join

Furthermore, there are the different "variants" of joins, namely **inner**, **left**, and **outer** join (not each stream type offers every variant though).

Join semantics are inspired by SQL join semantics, however, because Kafka Streams offers *stream* instead of batch processing, semantics do no align completely. In the following, we give a details explanation of the offered join semantics in Kafka Streams.

Below, we describe the semantics of each operator on two input streams/tables. We assume that all messages have the same key in these examples and thus omit the key to improve readability. For window joins, we assume that all records belong to a single window. Nevertheless, time (and processing order) is an important factor in stream-joins and thus we also show the timestamp for each record and assume that all records are processed in timestamp order. The format below will be `ts:value` for each record and *null* indicates a missing value.

- STREAM_1: 1:null, 3:A, 5:B 7:null, 9:C, 12:null, 15:D
- STREAM_2: 2:null, 4:a, 6:b, 8:null, 10:c, 11:null, 13:null, 14:d

Pay attention, that both streams are use as examples for KStream (ie, record stream) and KTable (ie, changelog stream) with different semantics. For KTable, so-called tombstone records with format `key:null` are of special interest, as they delete a key (those records are shown as **null** in all examples to highlight tombstone semantics). Last but not least, in Kafka Streams each join is "customized" by the user with a `ValueJoiner` function that compute the actual result. Hence, we show output records as *"X - Y"* with X and Y being the left and right value, respectively, given to the value joiner. If the output is shown as **null** (ie, tombstone message), `ValueJoiner` will not be called because a result record will be deleted.

Improved Left/Outer Stream-Stream Join (v`3.1.x` and newer)

> Prior to version `3.1.x` Kafka Streams might emit so called "spurious" left/outer join result. In this section we only explain the different new behavior that avoids spurious left/outer stream-stream join results. See New Join Semantics below that describe all joins in more details, including spurious left/output join behavior in versions `0.10.2.x` to `3.0.x`.

(See Unable to render Jira issues macro, execution error.)

KStream-KStream Join

This is a sliding window join, ie, all tuples that are "close" to each other with regard to time (ie, time difference up to window size) are joined. The result is a KStream. The table below shows the output (for each processed input record) for left and outer join only (as inner joins are not subject to spurious join results). Pay attention, that some input records do not produce output records, and that left/outer output record are emitted with some "delay" (ie, only emitted after grace-period passed). Also note, that the new behavior requires to set a **gracePeriod** in the window definition to specify when left/outer join result should be emitted via `ofTimeDifferenceNoGrace()` or `ofTimeDifferenceWithGrace(...)` (setting the grace period using the old and now deprecated API, `JoinWindows.of(...).grace(...)`, will not result in this new behavior, but will produce the same result as in older releases, `0.10.2.x` to `3.0.x`).

In contrast to the later examples, we assume a window size of 15, and a grace period of 5.

ts	STREAM_1 (left)	STREAM_2 (right)	innerJoin (same as in older versions)	leftJoin	outerJoin
1	null				
2		null			

3	A				
4		a	A - a	A - a	A - a
5	B		B - a	B - a	B - a
6		b	A - b B - b	A - b B - b	A - b B - b
7	null				
8		null			
9	C		C - a C - b	C - a C - b	C - a C - b
10		c	A - c B - c C - c	A - c B - c C - c	A - c B - c C - c
11		null			
12	null				
13		null			
14		d	A - d B - d C - d	A - d B - d C - d	A - d B - d C - d
15	D		D - a D - b D - c D - d	D - a D - b D - c D - d	D - a D - b D - c D - d
...					
40	E				
...					
60	F			E - null	E - null
...					
80		f		F - null	F - null
...					
100	G				null - f

New Join Semantics (`v0.10.2.x` and newer)

This section describes the new join semantics as of version `0.10.2.x`. For old join semantics (version `0.10.0.x` and `0.10.1.x`) see Old Join Semantics below.

(See KIP-77: Improve Kafka Streams Join Semantics)

Kafka Streams offers the follow join operators (operators in **bold font** were added in current trunk, compared to 0.10.1.x and older):

	inner join	**left join**	**outer join**
KStream-KStream	yes	yes	yes
KStream-KTable	**yes**	yes	no
KTable-KTable	yes	yes	yes

KStream-KStream Join

This is a sliding window join, ie, all tuples that are "close" to each other with regard to time (ie, time difference up to window size) are joined. The result is a KStream. The table below shows the output (for each processed input record) for all three join variants. Pay attention, that some input records do not produce output records.

The table below marks so called "spurious" left/outer join results, that are in the result in version `0.10.2.x` to `3.0.x`, in bold face. Compare Improved left /outer stream-stream join above for version `3.1.x` that avoids spurious results.

ts	STREAM_1 (left)	STREAM_2 (right)	innerJoin	leftJoin	outerJoin
1	null				
2		null			
3	A			**A - null**	**A - null**
4		a	A - a	A - a	A - a
5	B		B - a	B - a	B - a
6		b	A - b B - b	A - b B - b	A - b B - b
7	null				
8		null			
9	C		C - a C - b	C - a C - b	C - a C - b
10		c	A - c B - c C - c	A - c B - c C - c	A - c B - c C - c
11		null			
12	null				
13		null			
14		d	A - d B - d C - d	A - d B - d C - d	A - d B - d C - d
15	D		D - a D - b D - c D - d	D - a D - b D - c D - d	D - a D - b D - c D - d

KStream-KTable Join

This is an asymmetric non-window join. The basic semantics is a KTable lookup for each KStream record (while each KTable input record updates the current KTable view but does never yield any result record). The result is a KStream. Pay attention, that the KTable lookup is done on the *current* KTable state, and thus, out-of-order records can yield non-deterministic result. Furthermore, in older version of Kafka Streams there is no guarantee that all records will be processed in timestamp order (even if processing records in timestamp order is the goal, it is only best effort). The table below shows the output (for each processed input record) for both offered join variants. Pay attention, that some input records do not produce output records.

In newer versions, Kafka Streams improved timestamp synchronization significantly:

- `2.1.x` and newer: improvements in processing order and introducing `max.task.idle.ms` config to allow for partial blocking if one input is empty (cf. KIP-353: Improve Kafka Streams Timestamp Synchronization)
- `3.0.x` and newer: stronger synchronization guarantees to avoid race conditions due to unpredictable consumer fetch behavior (cf. KIP-695: Further Improve Kafka Streams Timestamp Synchronization)

ts	STREAM_1 (left)	STREAM_2 (right)	leftJoin	innerJoin
1	null			
2		**null**		
3	A		A - null	
4		a		
5	B		B - a	B - a
6		b		
7	null			
8		**null**		

9	C		C - null	
10		c		
11		**null**		
12	null			
13		**null**		
14		d		
15	D		D - d	D - d

KTable-KTable Join

This is a symmetric non-window join. The basic semantics is a KTable lookup in the "other" stream for each KTable update. The result is a (continuously updating) KTable (ie, a changelog stream that can contain tombstone message with format `<key:null>`; those tombstone are shown as **null** in the result in contrast to results *"X - null"* indicating a valid join result with only one join partner). Pay attention, that the KTable lookup is done on the *current* KTable state, and thus, out-of-order records can yield non-deterministic result. Furthermore, in older versions of Kafka Streams there is no guarantee that all records will be processed in timestamp order (even if processing records in timestamp order is the goal, it is only best effort).

In newer versions, Kafka Streams improved timestamp synchronization significantly:

- `2.1.x` and newer: improvements in processing order and introducing `max.task.idle.ms` config to allow for partial blocking if one input is empty (cf. KIP-353: Improve Kafka Streams Timestamp Synchronization)
- `3.0.x` and newer: stronger synchronization guarantees to avoid race conditions due to unpredictable consumer fetch behavior (cf. KIP-695: Further Improve Kafka Streams Timestamp Synchronization)

> **KTable Cache**
>
> If you want to observe the below described behavior, you will most likely need to disable `KTable` deduplication cache, by setting `cache.max.bytes.buffering=0` in `StreamsConfig`. Otherwise, the deduplication cache will "swallow" many of the produced result records and it will be hard to reason about the actual join behavior.

ts	left	right	innerJoin	leftJoin	outerJoin
1	**null**				
2		**null**			
3	A			A - null	A - null
4		a	A - a	A - a	A - a
5	B		B - a	B - a	B - a
6		b	B - b	B - b	B - b
7	**null**		**null**	**null**	null - b
8		**null**			**null**
9	C			C - null	C - null
10		c	C - c	C - c	C - c
11		**null**	**null**	C - null	C - null
12	**null**			**null**	**null**
13		**null**			
14		d			null - d
15	D		D - d	D - d	D - d
16					
17		d	D - d	D - d	D - d

KTable-KTable Foreign-Key Join (v`2.4.x` and newer)

This is a symmetric non-window join. There are two streams involved in this join, the left stream and the right stream, each of which are usually keyed on different key types. The left stream is keyed on the primary key, whereas the right stream is keyed on the foreign key. Each element in the left stream has a foreign-key extractor function applied to it, which extracts the foreign key. The resultant left-event is then joined with the right-event keyed on the corresponding foreign-key. Updates made to the right-event will also trigger joins with the left-events containing that foreign-key. It can be helpful to think of the left-hand materialized stream as *events* containing a foreign key, and the right-hand materialized stream as *entities* keyed on the foreign key.

KTable lookups are done on the *current* KTable state, and thus, out-of-order records can yield non-deterministic result. Furthermore, in older versions of Kafka Streams there is no guarantee that all records will be processed in timestamp order (even if processing records in timestamp order is the goal, it is only best effort).

In newer versions, Kafka Streams improved timestamp synchronization significantly:

- `2.1.x` and newer: improvements in processing order and introducing `max.task.idle.ms` config to allow for partial blocking if one input is empty (cf. KIP-353: Improve Kafka Streams Timestamp Synchronization)
- `3.0.x` and newer: stronger synchronization guarantees to avoid race conditions due to unpredictable consumer fetch behavior (cf. KIP-695: Further Improve Kafka Streams Timestamp Synchronization)

The workflow of LHS-generated changes to outputs is shown below. Each step is cumulative with the previous step. Only LEFT and INNER joins are supported, and their outputs are shown below.

ts		LHS-Stream (K, extracted-FK)	RHS-Stream State (FK,V)	Inner-Join Output	Left-Join Output
1	Publish event to LHS	(k,1)	(1,foo)	(k,1,foo)	(k,1,foo)
2	Change LHS fk	(k,2)	(1,foo)	(k,null)	(k,2,null)
3	Change LHS fk	(k,3)	(1,foo)	(k,null)	(k,3,null)
4	Publish RHS entity	-	(1,foo) (3,bar)	(k,3,bar)	(k,3,bar)
5	Delete k	(k,null)	(1,foo) (3,bar)	(k,null)	(k,null,null)
6	Publish original event again	(k,1)	(1,foo) (3,bar)	(k,1,foo)	(k,1,foo)
7	Publish event to LHS	(q,10)	(1,foo) (3,bar)	-	(q,null,10)
8	Publish RHS entity	-	(1,foo) (3,bar) (q,baz)	(q,10,baz)	(q,10,baz)

Old Join Semantics (`v0.10.0.x` and `v0.10.1.x`)

Kafka Streams 0.10.0.x and 0.10.1.x offers the follow join operators:

	inner join	left join	outer join
KStream-KStream	yes	yes	yes
KStream-KTable	no	yes	no
KTable-KTable	yes	yes	yes

KStream-KStream Join

This is a sliding window join, ie, all tuples that are "close" to each other with regard to time (ie, time difference up to window size) are joined. The result is a KStream. The table below shows the output (for each processed input record) for all three join variants. Pay attention, that some input records do not produce output records.

ts	STREAM_1 (left)	STREAM_2 (right)	innerJoin	leftJoin	outerJoin
1	null			null - null	null - null
2		null			null - null
3	A			A - null	A - null
4		a	A - a		A - a
5	B		B - a	B - a	B - a

6		b	A - b B - b		A - b B - b
7	null		null - a null - b	null - a null - b	null - a null - b
8		null	A - null B - null		A - null B - null
9	C		C - a C - b	C - a C - b	C - a C - b
10		c	A - c B - c C - c		A - c B - c C - c
11		null	A - null B - null C - null		A - null B - null C - null
12	null		null - a null - b null - c	null - a null - b null - c	null - a null - b null - c
13		null	A - null B - null C - null		A - null B - null C - null
14		d	A - d B - d C - d		A - d B - d C - d
15	D		D - a D - b D - c D - d	D - a D - b D - c D - d	D - a D - b D - c D - d

KStream-KTable Join

This is an asymmetric non-window join. The basic semantics is a KTable lookup for each KStream record. The result is a KStream. Pay attention, that the KTable lookup is done on the *current* KTable state, and thus, out-of-order records can yield non-deterministic result. Furthermore, in practice Kafka Streams does not guarantee that all records will be processed in timestamp order (even if processing records in timestamp order is the goal, it is only best effort). The table below shows the output (for each processed input record) for both offered join variants. Pay attention, that some input records do not produce output records.

ts	STREAM_1 (left)	STREAM_2 (right)	leftJoin
1	null		null - null
2		**null**	
3	A		A - null
4		a	
5	B		B - a
6		b	
7	null		null - b
8		**null**	
9	C		C - null
10		c	
11		**null**	
12	null		null - null

13		**null**	
14		d	
15	D		D - d

KTable-KTable Join

This is a symmetric non-window join. The basic semantics is a KTable lookup in the "other" stream for each KTable update. The result is a (continuously updating) KTable (ie, a changelog stream that can contain tombstone message with format `<key:null>`; those tombstone are shown as **null** in the result in contrast to results *"X - null"* indicating a valid join result with only one join partner). Pay attention, that the KTable lookup is done on the *current* KTable state, and thus, out-of-order records can yield non-deterministic result. Furthermore, in practice Kafka Streams does not guarantee that all records will be processed in timestamp order (even if processing records in timestamp order is the goal, it is only best effort).

> **KTable Cache**
>
> If you want to observe the below described behavior, you will most likely need to disable `KTable` deduplication cache (for Kafka `0.10.1.x`), by setting `cache.max.bytes.buffering=0` in `StreamsConfig`. Otherwise, the deduplication cache will "swallow" many of the produced result records and it will be hard to reason about the actual join behavior.

ts	STREAM_1 (left)	STREAM_2 (right)	innerJoin	leftJoin	outerJoin
1	**null**		**null**	**null**	**null**
2		**null**	**null**	**null**	**null**
3	A		**null**	A - null	A - null
4		a	A - a	A - a	A - a
5	B		B - a	B - a	B - a
6		b	B - b	B - b	B - b
7	**null**		**null**	**null**	null - b
8		**null**	**null**	**null**	**null**
9	C		**null**	C - null	C - null
10		c	C - c	C - c	C - c
11		**null**	**null**	C - null	C - null
12	**null**		**null**	**null**	**null**
13		**null**	**null**	**null**	**null**
14		d	**null**	**null**	null - d
15	D		D - d	D - d	D - d
16					
17		d	D - d	D - d	D - d