Discussion: Joins (as of 0.10.0.0)

Preliminary Remarks

Stream

Each element in a stream is a key-value pair. A stream is partitioned by the key.

- Key = Partitioning Key

Join

Our JOIN operators are basically equi-joins on the keys of streams. Streams being joined must have the same key type. Also, they must be copartitioned.

- Join Key = Key = Partitioning Key

KTable

KTable is a particular interpretation of a stream as a change log of a conceptual (or underlying) table. KTable's key is the primary key of the table. The corresponding topic is likely to be compaction enabled.

- Key = Primary Key = Partitioning Key = Compaction Key

KTable Aggregate

A KTable can be aggregated by a non-key field. The resulting stream is another KTable keyed by the aggregation key.

- Aggregation by a non key field
- The result is another KTable
  - Key = Aggregation Key = Primary Key = Compaction Key = Partitioning Key
  - Value = Aggregate Value (ex. count, sum)

Windowed Aggregate

A stream can be aggregated by a window. Data within a window can be aggregated by key (ex. a page view count for each page id by day from a page view stream).

The aggregation key is the key of the stream. If a user want to aggregate the stream by non-key, he/she has to repartition the stream by using map(). through() or map().to() before aggregation.

The key of the result stream is the primary key of the produced stats, thus it is desired to be:

- key = aggregation key + window id

Note that the partitioning key is still the aggregation key. Although this breaks a basic assumption that the key is the partitioning key, this is advantageous in joining over the aggregation key, which may be more useful than joining over the combined key (aggregation key + window id). For example, a user may want to compare today's page views with the last week's. In this case join is performed on the aggregation key, however "keys" are different since the window ids do not match.

As long as we contain this exception in our local operation, it should not break the system. For that reason a new distinct type of a stream, WTable, is introduced below.

WTable<K, V, W>

WTable cannot be directly written to a topic.

- no to() or through()

A user has to convert WTable to KStream with some transformation before persisting to a topic.

- wtable .toStream ((K, V, W) (K1, V1)).to(topic)

It is up to a user, but some reasonable conversions may be:

- (K, W) K1, V V2
    - K1 is a primary key
    - can be read as KTable
- K K1, (V, W) V1
    - joinable by the source key
    - make no sense to read as KTable

Instances of WTable are created only by the framework as the result of windowed aggregate.

- no topic can be read as WTable

JOIN operators

Join Types

A join combines two streams. The first stream is called the primary stream, and the second stream is called the secondary stream.

primary-stream × *secondary-stream*

We have three types of joins.

- Inner Join
- Outer Join
- Left Join (no Right Join)

Available join types depends on the types of join streams.

primary \ secondary	KStream	KTable	WTable
KStream	LOI †	L	~~L~~ ?
KTable		LOI	~~L~~ ?
WTable		~~L~~	LOI

I: inner join, O: outer join, L: left join

†: windowed joins

?: any use case?

The result types of join are as follows.

primary \ secondary	KStream	KTable	WTable
KStream	KStream	KStream	~~KStream~~

KTable		KTable	~~KTable~~
WTable		~~WTable~~	WTable

Join Processing

KTable-KTable Join

A join is performed when a record arrives at the join operator. The new record in one stream is matched with records in a materialized table of the other stream. All types of joins are driven by both streams.

When tables (KTable, WTable) are joined, the result is also a table. Let *T* be the change log, *t* be the materialization of *T*, and Function *f: T t* be the materialization function.

If

$t_1 = f(T_1)$
$t_2 = f(T_2)$

then

$innerJoin(t_1, t_2) = f(innerJoin(T_1, T_2))$
$outerJoin(t_1, t_2) = f(outerJoin(T_1, T_2))$
$leftJoin(t_1, t_2) = f(leftJoin(T_1, T_2)).$

Thus, in this sense, ***table-table join*s are eventually consistent**. This gives a kind of resilience to late arrival of records. A late arrival in either stream can "update" the join result.

KStream-KStream Join

A join is performed when a record arrives at the join operator. The new record in one stream is matched with buffered records of the other stream. The inner join and the outer join are driven by both streams, thus both streams must have a buffer. On the other hand, unlike table-table left join, the stream-stream left join is driven only by the primary stream, so only the secondary stream is required to have a buffer. This style of processing affects the consistency of join results.

We will consider a single window of a windowed join. Let *S* be a stream, *s* be the set of records, and *g: S s* such that *s* contains exactly all records in *S*.

If

$s_1 = g(S_1)$
$s_2 = g(S_2)$

then

$innerJoin(s_1, s_2) = g(innerJoin(S_1, S_2))$

Thus, in this sense, ***inner join* is eventually consistent.** However, *outer join* and *left join* do not possess this property. There is no general way for a late arrival to "update" earlier result because a stream, unlike a table, does not have a primary key.

Output of Join

Let us consider the inner/outer join processing with hopping windows, where sequence of windows come and go. One record may belong to multiple windows. Suppose two records r_1, r_2 , whose keys are same, are associated with sets windows (w_1, w_2, w_3) and (w_2, w_3, w_4), respectively. What should the join outputs look like?

Matching windows, windows which has both r_1 and r_2, are (w_2, w_3). Do we want to emit an output record for each matching window? It will be confusing because the number of duplicate records depends on the number of matching windows of two records.

A window join is a join by a time difference. It makes sense to emit a single output record no matter how many matching windows inputs have. (If we follow this direction, it should not be a user's concern that the join use which type of windowing, like hopping window or sliding window. It is an implementation detail.)

Possible Implementations of Join window

Single RocksDB Instances for Each Stream

- a rocksdb store for each stream
  - the key is the concatenation of the record key and the timestamp
  - the value is the value of the record
  - set TTL to the window retention period to physically remove expired windows from rocksdb
    - bootstrapping may temporarily take a large amount of storage since TTL is by the system time.
- for each incoming record *r*

1. store the pair *(r.key + r.timestamp, r.value)* in the corresponding rocksdb store
2. compute a valid range *(min-time, max-time) = (r.timestamp - window-size, r.timestamp + window-size)*
3. do a range search by the range *(r.key + min-time, r.key + max-time)* on the other rocksdb store
4. combine the range search result with *r* and produce outputs

Multiple RocksDB Instances for each Stream

- multiple rocksdb instances for each stream
  - the key is the concatenation of the record key and the timestamp
  - the value is the value of the record
  - TTL is not used. Instead multiple rocksdb instances are used for "rolling"
    - The two-instance config requires 2x storage overhead. The three-instance config requires 1.5x.
    - we can control rolling by the stream time, thus we can keep the storage requirement low while bootstrapping.
- processing is similar to the single rocksdb method
  - it only needs to pick the right instance to get data.
  - at the boundary of rolling the range search has to split into two ranges and query two instances

KStream-KTable Joins

A join is performed when a record from the primary stream arrives at the join operator. The new record in the primary stream is matched with records in a materialized table of the secondary stream. Only the left join is defined. Unlike table-table left join, the stream-table left join is driven only by the primary stream. This style of processing does not guarantee the consistency of join results.

Join API

Symbols

- K: key type
- V: value type
- W: window type
- $S_{K,V}$: an instance of KStream<K, V>
- $T_{K,V}$: an instance of KTable<K, V>
- $WT_{K,V,W}$: an instance of WTable<K,V,W>

KTable-KTable Join

$T_{K,V1}$ .join ($T_{K,V2}$, (V1, V2)V3)

- returns $T_{K,V3}$

$T_{K,V1}$ .outerJoin ($T_{K,V2}$, (V1, V2)V3)

- returns $T_{K,V3}$
- V1 is null when $T_{K,V1}$ does not have a corresponding record
- V2 is null when $T_{K,V2}$ does not have a corresponding record

$T_{K,V1}$ .leftJoin ($T_{K,V2}$, (V1, V2)V3)

- returns $T_{K,V3}$
- V2 is null when $T_{K,V2}$ does not have a corresponding record

KStream-KTable Join

$S_{K,V1}$ .leftJoin ($T_{K,V2}$, (V1, V2)V3)

- returns $S_{K,V3}$
- V2 is null when $S_{K,V2}$ does not have a corresponding record

KStream-KStream Join with Windowing

$S_{K,V1}$ .leftJoin ($S_{K,V2}$, (V1, V2)V3 , WindowSpec) --- *any use case?*

- returns $S_{K,V3}$
- V2 is null when $S_{K,V2}$ does not have a corresponding record

$S_{K,V1}$ .join ($S_{K,V2}$, (V1, V2)V3 , WindowSpec)

- returns $S_{K,V3}$

$S_{K,V1}$ .outerJoin ($S_{K,V2}$, (V1, V2)V3 , WindowSpec)

- returns $S_{K,V3}$
- V1 is null when $S_{K,V1}$ does not have a corresponding record
- V2 is null when $S_{K,V2}$ does not have a corresponding record

~~KStream-WTable Join~~

~~$S_{K,V1}$ .leftJoin ($WT_{K,V2,W}$, **(K, V1)W** , (V1, V2)V3) --- *any use case?*~~

- ~~returns $S_{K,V3}$~~
- ~~(K, V1)W specifies which window to look~~
- ~~V2 is null when $S_{K,V2}$ does not have a corresponding record~~

~~KTable-WTable Join~~

~~$T_{K,V1}$ .leftJoin ($WT_{K,V2,W}$, **(K, V1)W** , (V1, V2)V3) --- *any use case?*~~

- ~~returns $T_{K,V3}$~~
- ~~(K, V1)W specifies which window to look~~
- ~~V2 is null when $T_{K,V2}$ does not have a corresponding record~~

~~WTable-KTable Join~~

~~$WT_{K,V1,W}$ .leftJoin ($T_{K,V2}$, (V1, V2)V3)~~

- ~~returns $WT_{K,V3,W}$~~
- ~~V2 is null when $T_{K,V2}$ does not have a corresponding record~~

WTable-WTable Join

$WT_{K,V1,W}$ .leftJoin ($WT_{K,V2,W}$, (V1, V2) V3)

- returns $WT_{K,V3,W}$
- joins on the same window
- V2 is null when $WT_{K,V2,W}$ does not have a corresponding record

$WT_{K,V1,W}$ .leftJoin ($WT_{K,V2,W}$, **WW**, (V1, V2) V3)

- returns $WT_{K,V3,W}$
- WW is a window shifter. It generates a window id for lookup from the window id from the primary $WT_{K,V1,W}$. (ex. shifting W to one week back).
- V2 is null when $WT_{K,V2,W}$ does not have a corresponding record

$WT_{K,V1,W}$ .join ($WT_{K,V2,W}$, (V1, V2) V3)

- returns $WT_{K,V3,W}$
- joins on the same window

$WT_{K,V1,W}$ .outerJoin ($WT_{K,V2,W}$, (V1, V2) V3)

- returns $WT_{K,V3,W}$
- joins on the same window
- V1 is null when $WT_{K,V1,W}$ does not have a corresponding record
- V2 is null when $WT_{K,V2,W}$ does not have a corresponding record