KIP-X: Native TopK support in Kafka Streams

Status

Current state: *Draft*

Discussion thread: TBD

JIRA: TBD

Please keep the discussion on the mailing list rather than commenting on the wiki (wiki discussions get unwieldy fast).

Motivation

TopK is a very commonly supported feature in SQL-like queries, and widely used in tech and non-tech industry alike. For example, a blog site owner would like to know the top 5 active authors posting blogs:

```
SELECT * FROM authors ORDER BY num_posts DESC LIMIT 5
```

similarly a professor wants to know the top 10 students in their final, a salesman wants to know best selling products, etc. These needs are important to be satisfied in traditional database. In the streaming SQL, this is also a valuable problem to be solved natively on Streams to natively support `ORDER BY` and `LIMIT` semantics on KSQL, instead of relying on customized implementations.

To breakdown, the TopK operator is aiming at globally sort all the emitted key-value pairs from upstream operator, and only retain the top K values within its state as a table. We also see cases where the TopK query is partitioned, such as a salesperson wants to know the best sellers in different categories. Considering the nature of ever updating streams, it is also necessary to support windowed TopK as well.

Public Interfaces

We are proposing a new operator called top() on KTable interface to create a materialized view for global ranking of data records in either ascending or descending order, with only limited number of withholding elements:

KTable.java

```
<P, VO> KTable<P, Map<Integer, VO>> top(Ranker<? super K, ? super V, P, VO> ranker);
```

The reason for only supporting KTable operator is that the top() operator does not serve any aggregation purpose, so its comparison is purely record based. For a KStream instance, it is advised to do the aggregation first to create a changing agg result as KTable.

The Ranker is a generic interface to be implemented. It covers the following necessary definitions for topK operator:

1. The ranking order (ascending | descending)
2. The element limit
3. The partition key (if needed)
4. The output value type

Ranker

```
/**
 * The helper instance to create the topK ranking based on streaming in events or table updates.
 *
 * @param <K> input record key
 * @param <V> input record value
 * @param <P> input partition key
 * @param <VO> the output value type
 */
public interface Ranker<K, V, P,  VO> {

    enum Order {
      ASCENDING,
      DESCENDING
    }

    /**
     * The order of ranking, either ascending or descending.
     */
    Order order();

    /**
     * The number of retained elements in given order.
     *
     * @return the total number of top records to be preserved.
     */
    int limit();

    /**
     * Compare the two key-value pairs rank.
     *
     * @param key1       the key of the first record
     * @param value1     the value of the first record
     * @param key2       the key of the second record
     * @param value2     the value of the second record
     *
     * @return the comparison result, 0 means equal, negative means record 1 < record 2, positive means record
1 > record 2
     */
    int compare(final K key1, final V value1, final K key2, final V value2);

    /**
     * Get the partition key for different ranking categories
     *
     * @param key key of the record
     * @param value value of the record
     * @return the partition key
     */
    P partition(final K key, final V value);

    /**
     * The output value type to be saved in the top ranking result.
     *
     * @param key record key
     * @param value record value
     * @return output value
     */
    VO output(final K key, final V value);
}
```

The output flow here is an ever-updating KTable with key as the ranking number and value as the user-defined type. Thus we could support very complex query on the Ksql layer as:

```
SELECT * FROM (

SELECT *, ROW_NUMBER() OVER (PARTITION BY category ORDER BY sales DESC) as row_num

FROM ShopSales)

WHERE row_num <= 5
```

And a sample streaming query to compute the most recent 20 logged-in users would be look like:

SampleStream.java

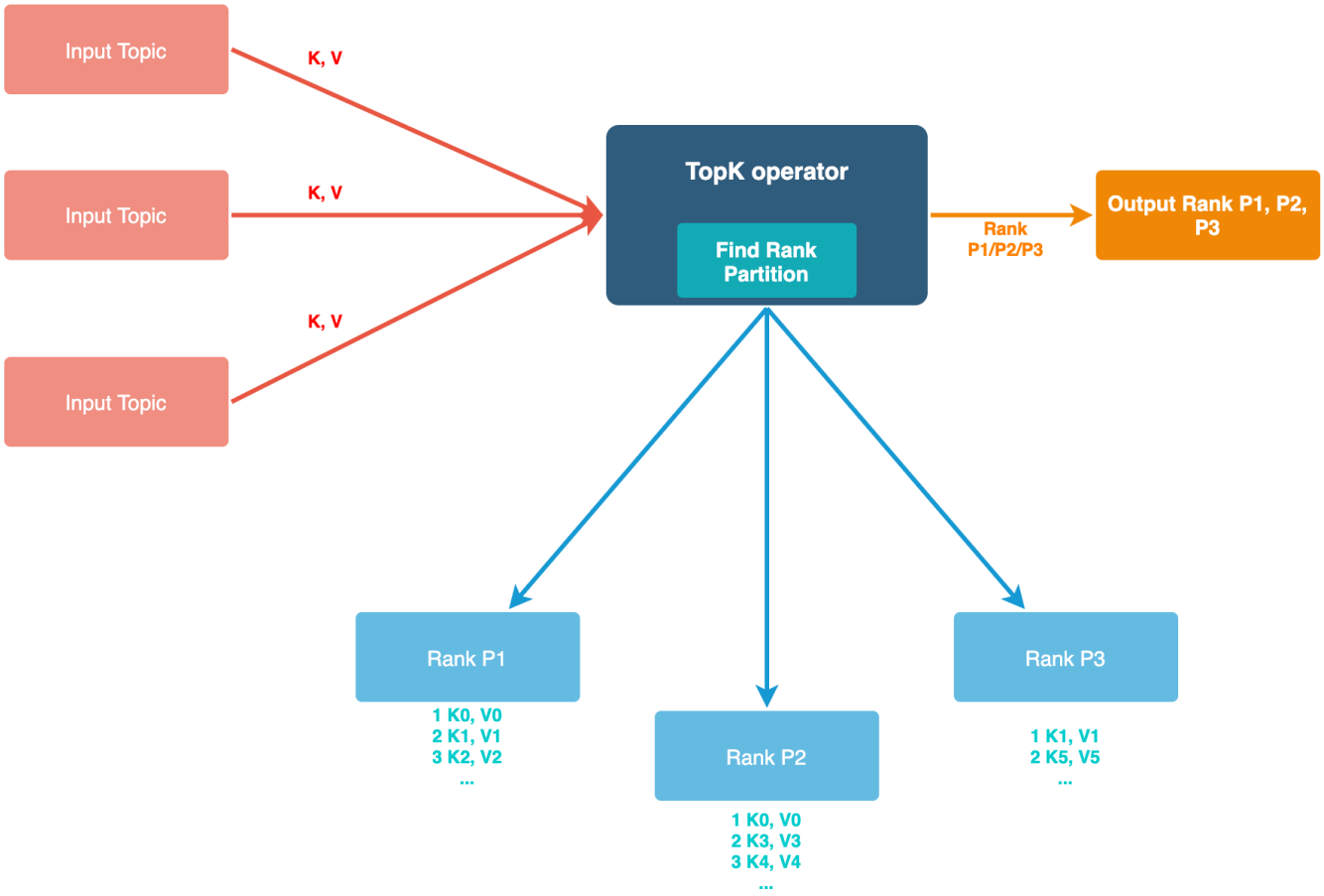
```
final StreamsBuilder builder = new StreamsBuilder();

final KTable<String, UserProfile> users = builder.table("streams-userprofile-input", Consumed.with(Serdes.
String(), new JSONSerde<>()));

final KTable<String, UserAggStats> userActivities = users.groupByKey(...).aggregate(...); // get the
aggregation result of user comments, blog posts, etc.

userActivities.top(new Ranker<String, UserAggStats, String, UserProfile>() {
    @Override
    public Order order() {
        return Order.ASCENDING;
    }

    @Override
    public int limit() {
        return 20;
    }

    @Override
    public int compare(String key1, UserAggStats stats1, String key2, UserAggStats stats2) {
        return new Long(stats1.numComments - stats2.numComments).intValue();
    }

    @Override
    public String partition(String key, UserAggStats stat) {
        return stat.profile().region; // Partition users by region
    }

    @Override
    public UserProfile output(String key, UserAggStats stat) {
        return stat.profile();
    }
});
```

Proposed Changes

We are expecting to roll out this change in 3 phases. This KIP will focus on phase 1 and 2 design.

Basic In-memory Ranking Store Support (Phase 1)

We would start from using an in-memory structure to maintain the ever changing rank as a single node. The operator will serve as a special task which subscribes all upstream partitions and compute the result as a mapping from user defined partition towards the top ranked values. The architecture will look like:



Scale Limit

In the first version we are not trying to tackle the scalability problem. We will be restricting the total number of maintained elements to 10000, which means if you have 10 customized categories, each domain should have at most 1000 top elements to be maintained.

Optimizations (Phase 2)

Two-level TopK

When the number of unique key grows, it is no longer trivial to maintain a single node architecture to compute all the incoming changes on the fly. Thus we propose to adopt a similar strategy from Flink as two-level TopK: first on each upstream partition, there would be a first level sorting and topK values computation, and then gets sent to the intermediate repartitioned topic, which will be processed by a separate second level TopK operator. This will greatly reduce the amount of work on the single node as the pre-filtering is working.

Incremental Update

To reduce the amount of data transmitted through the wire, we could optionally choose to do incremental updates of the ranking instead of a full update to push to the downstream. This means that we maintain the same output format as Map<Integer, VOut>, but do the updates in the meantime as well. The approach is to add an API to the Ranker to define whether the given operator sends partial or full updates.

Extension (Phase 3)

Windowed table is a special type of KTable where its key is windowed. The current proposal hasn't touched the complicated nesting logic between windowed table and topK definition. In the perspective of the end users, a topK operator connecting with windowed table should be defined as individual rank order for each time window with retention. The data state volume will be really huge with many separate rankings based on time window. We decide to postpone the design for windowed table upstream before we collect enough supporting user stories and industry use cases to properly define its semantic. By then, the design for window scalability is making sense.

Compatibility, Deprecation, and Migration Plan

N/A

Rejected Alternatives

N/A

Reference

Flink Top-N operator: https://ci.apache.org/projects/flink/flink-docs-master/dev/table/sql/queries.html#top-n