Kafka Streams Application to perform Join and Benchmarking

Key Concepts:

Kafka is a distributed, partitioned, replicated commit log service. Kafka Streams was built with the goal to handle and process real time data feeds of companies to produce useful derived feeds. Kafka uses Zookeeper for synchronizing its cluster, where there are multiple brokers for load balancing. Each Kafka broker instance can handle hundreds of thousands of reads and writes per second.

Topics and Partitions: Data is fed into topics which can have multiple partitions. Each partition of a topic has a leader broker which is responsible for reading writing to it. Each partition is also replicated on other brokers for fault tolerance in case any broker crashes. Messages are first sent to the leader of a partition and then replicated to the remaining brokers. This can be synchronous or asynchronous replication. This guarantees that any successfully published message will not be lost and can be consumed, even when there are server failures. The zookeeper maintains the information about: list of brokers; list of topics and a list of partitions for each topic.

Having more partitions increases I/O parallelism for writes and also increases the degree of parallelism for consumers (since partition is the unit for distributing data to consumers). Also, more partitions add some overhead: (a) there will be more files and thus more open file handlers; (b) there are more offsets to be checkpointed by consumers which can increase the load on ZooKeeper.

Kafka Streams: is a library for creating streams processing applications. Built upon the fundamental components of Kafka, streams also provide fault tolerance, partitioning, scalability, ordering, and load balancing. Kafka streams can be created from one or more kafka topics using the KstreamBuilder. Kafka streams are of two types: (a) KStream: record stream (b) KTable : changelog stream

Many of the streams applications work on streams table duality, since while performing operations one might be needed to convert to another. A KStream is an abstraction of a record stream, where each data record represents a self-contained datum. Data records in a record stream are always interpreted as “inserts” because no record replaces an existing row with the same key. A KTable is an abstraction of a changelog stream, where each data record represents an update. the value in a data record is considered to be an update of the last value for the same record key, if any (if a corresponding key doesn’t exist yet, the update will be considered a create). A stream can be considered a changelog of a table, where each data record in the stream captures a state change of the table. A table can be considered a snapshot, at a point in time, of the latest value for each key in a stream.

Stream Processing: Why stream processing is gaining so much popularity can be explained with the following example. Applications backing an online shop might need to access multiple, updating database tables (e.g. sales prices, inventory, customer information) when processing a new record. These applications can be implemented such that they work on the tables changelog streams directly, i.e. without requiring to make a database query over the network for each record. This would give so much performance gain as well as save network and storage resources.

Some stream processing applications don’t require state, which means the processing of a message is independent from the processing of all other messages such as filter, map, branch. Whereas some are stateful transformations such as joins, aggregations where the processing logic require accessing an associated state. Ability to maintain state provides possibilities for sophisticated stream processing applications. For example a windowed state is retention of records that fall within its interval, so that the processing logic can consider the little late arriving records as well, which is required for the join and similar operations, because the records of the two streams might not come perfectly synchronized. A local state store is needed to store the recently received records within a window which is specified by the *STATE_DIR_CONFIG* configuration property. The state store is a RocksDB database, which is an embeddable persistent key-value store for fast storage.

Timestamps are automatically embedded into Kafka messages. This can be the event time(creation time) or processing time depending on the configured setting for *log.messagetimestamp.type*. It can be customized too. The timestamp extractor in Kafka Streams will retrieve these embedded timestamps based on which they can decide if a record falls within a particular window.

Join operation produces new joined records when the join operator finds some records with the same key in the other stream / materialized store. Joins can be of different types:

KStream-to-KStream Joins: a newly received record from one of the streams is joined with the other stream's records within the specified window interval to produce one result for each matching pair based on the logic provided in the ValueJoiner. The output is a new stream instance.

KTable-to-KTable Joins: are join operations designed to be consistent with the ones in relational databases. Here, both changelog streams are materialized into local state stores to represent the latest snapshot of the their data table duals and join is performed for each matching pair depending on the ValueJoiner logic producing a new KTable.

KStream-to-KTable Joins allow you to perform table lookups against a changelog stream (KTable) upon receiving a new record from the KStream. Only records received from the record stream will trigger the join via a ValueJoiner, records received from the changelog stream will be used only to update the materialized state store. Result is a KStream.

Implementation:

Source Topics -> Source Streams:
user-clicks (user, clicks) → userclicks
user-location (user, region) → userregion

Derived Stream → Output Topic
regionclicks(region, clicks) → outputTopic1 (Join)
regionclicks(region, clicks) → outputTopic2 (WindowedJoin)

i) **Join.java** (KStream - KTable Join): As the records arrive, their changelog is updated in the table by the *leftjoin* function using a ValueJoiner. *map* function maps a record with the given key and value to a new key, value using a KeyValueMapper. A reduce function is also added to the processor, which combines values of the stream by key into a new instance of ever-updating KTable. In a KStream-KTable left join, a KStream record will only join a KTable record if the KTable record has arrived before the KStream record and is in the KTable. Otherwise, the join result will be null. That is why we get UNKNOWN at the start of the output stream. Output stream is written to topic *outputTopic1.*

ii) WindowedJoin.java (KStream - KStream Join): Joining two data streams which are not synchronized in time using *join* function with a ValueJoiner. *map* function maps a record with the given key and value to a new key, value using a KeyValueMapper. KSream-KStream join needs to be windowed otherwise the number of records that must be maintained for performing the join may grow indefinitely. Used *JoinWindows* provided in the KStream API with an interval of 10 seconds. Output stream is written to topic *outputTopic2*.

iii) Producer.java (Kafka Producer): to write data to the source topics (*user-clicks, user-location*) from which input streams are built.

iv) Consumer.java (Kafka Consumer): to consume the output created after the join operations from the topic (*outputTopic1 / outputTopic2*).

The producer and consumer needed to be implemented because the console-producer and consumer provided can serialize and deserialize records with key and value of String type only.

Serialization Deserialization: When a producer writes to a topic, records need to be serialized. When building a stream from a topic or consuming from a topic we need to provide correct deserializers for the key and value. Also when using a logged state store for example the stateful operations such as joins and aggregations, correct serde(serialization + deserialization) is needed which can be explicitly provided when calling API methods for the transformations. Default serde to be used can be provide via StreamsConfig.

Gradle: as the build tool

BENCHMARKING:

Kafka is fast. A single node can handle hundreds of read/writes from multiple clients in real time. Kafka is also distributed and scalable. It creates and takes down nodes in an elastic manner, without incurring any downtime. Data streams are split into partitions and spread over different brokers for parallelism and redundancy. The following are the observations of few tests conducted:

Cluster with 3 brokers
Each input topics had 3 partition
This configuration creates three stream tasks

Throughput (joins / second):

We can extract the timestamp of a record using *timestamp()* function. Found the minimum creation time of all records which is the creation time of the first record. And noted the unix time when the last record was consumed. This gave the approximate total for the entire process from producing to joining to consuming. Since the *Consumer* and the *WindowedJoin* were already started before the producer, the time calculated would be pretty close to actual join time. Had the consumer been started after a timegap which could be the case for certain applications the calculated time won't be valid. For such cases, we can track the first and last record and extract their timestamps to calculate the difference.

(Number of records produced*1000/calculated time in milliseconds) gives the joins per second.
Throughput observed = 1000000*1000/226520 = 4414 joins per second

Latency
The default timestamp is creation time, which can be extracted from each consumer record. While consuming records, stored the unix time for each record. Their difference gives the latency. Calculated the average of all latencies.

Latency observed = 7.86 milliseconds
For single broker, single partition configuration, latency was much less around 2.5 milliseconds

Parallelism
As mentioned earlier, created topics with three partitions each:
```
$ bin/kafka-topics.sh --zookeeper localhost:2181  --create --topic user-clicks --partitions 3 --replication-factor 3
```

Kafka creates 3 logical partitions for the topic and elects a leader(broker) for each partition. A total of three replicas per partition are created and distributed across the alive brokers.

```
$ bin/kafka-topics.sh --zookeeper localhost:2181 --describe --topic user-clicks
```

```
    Topic:user-clicks     PartitionCount:3      ReplicationFactor:3   Configs:
    Topic: user-clicks    Partition: 0    Leader: 1     Replicas: 1,0,2       Isr: 0,1,2
    Topic: user-clicks    Partition: 1    Leader: 1     Replicas: 2,1,0       Isr: 1,0,2
    Topic: user-clicks    Partition: 2    Leader: 0     Replicas: 0,2,1       Isr: 0,1,2
```

Fault tolerance:
Shutting down a broker, distributes the tasks among other alive brokers transparent to the consumers. Running the above command reassigns the partitions to alive brokers and elects of new leaders for needed partitions.

```
    Topic:user-clicks     PartitionCount:3      ReplicationFactor:3   Configs:
    Topic: user-clicks    Partition: 0    Leader: 1     Replicas: 1,0,2       Isr: 0,1
    Topic: user-clicks    Partition: 1    Leader: 1     Replicas: 2,1,0       Isr: 1,0
    Topic: user-clicks    Partition: 2    Leader: 0     Replicas: 0,2,1       Isr: 0,1
```

The stream processing task is reassigned to another active broker.

Scalability:
Cluster Expansion - Multibroker Single Node
We have already seen how to create a multibroker cluster on a single node. The partition reassignment tool can be used to expand this existing Kafka cluster. Simply adding a broker to a cluster won't receive any data from existing topics until this command is run:
```
$ ./bin/kafka-reassign-partitions.sh –[topics to move] –broker-list [broker list] --execute
```

We can also add partitions to existing topics with the add partition tool command :

./bin/kafka-add-partitions.sh

Multibroker Multiple Node
I was using cloudlab, where I had an account from one of my University projects. As per the instructions found online, I ran the following steps:
ssh into a remote machine and install kafka

In both nodes added the ip addresses of other zookeeper server in the *kafka/config/zookeeper.properties* file
server.1= node1address

In the broker configurations *updated the* file: *kafka/config/server.properties*
zookeeper.connect=node1IP:2181, node2IP:2181
broker.id [every broker in the cluster should have a unique id]
Then start the zookeeper on both machines, instantiate the brokers and then run the application.
But as of now, this hasn't work. It was showing some error, which I couldn't resolve because of lack of time.

As far as the concepts go, when we create a new instance of a stream application, being coordinated by zookeeper, they will become aware of each other and automatically begin to share the processing work .
* how many instances of application to run : max(no. Of partitions of the input topics)
*partitioning properly is necessary to prevent processing hot-spots on certain brokers.

Memory Usage
A stream processing application that uses the Kafka Streams library is a normal Java application.
Command: ps awx
VSZ = 27339524
RSS= 2747128

RSS is the Resident Set Size: total memory of the processes in RAM. It does not include memory that is swapped out. It includes memory from shared libraries if pages from those libraries are actually in memory. It does include all stack and heap memory.

VSZ is the Virtual Memory Size. It includes all memory that the process can access, including memory that is swapped out and memory that is from shared libraries.

Garbage collected heap statistics:
jstat -gc pid

```
Driver Class
 S0C    S1C    S0U    S1U      EC       EU        OC         OU       MC     MU    CCSC   CCSU   YGC     YGCT    FGC    FGCT     GCT
512.0  512.0   0.0    32.0  15360.0   8601.5   40960.0     2541.2   9344.0 8947.1 1152.0 1024.6    131    3.761   0      0.000    3.761
Driver Class
S0C    S1C    S0U    S1U      EC       EU        OC         OU       MC     MU    CCSC   CCSU   YGC     YGCT    FGC    FGCT     GCT
33280.0 512.0   0.0    64.0  252416.0 51655.8   48640.0    28595.0   15104.0 14880.6 1792.0 1669.9     29    3.783   1      0.578    4.361
kafka-broker 1
 S0C    S1C    S0U    S1U      EC       EU        OC         OU       MC     MU    CCSC   CCSU   YGC     YGCT    FGC    FGCT     GCT
0.0   6144.0  0.0   6144.0 49152.0  36864.0   993280.0   145448.0  23424.0 23069.3 3200.0 3097.1     29    0.645   0      0.000    0.645
kafka-broker 1
S0C    S1C    S0U    S1U      EC       EU        OC         OU       MC     MU    CCSC   CCSU   YGC     YGCT    FGC    FGCT     GCT
```

```
 0.0   4096.0  0.0   4096.0 51200.0  46080.0   993280.0   238763.0  26496.0 25918.3 3712.0 3559.8    225   16.034   0      0.000   16.034
kafka-broker 2
 S0C    S1C    S0U    S1U      EC       EU        OC         OU       MC     MU    CCSC   CCSU   YGC     YGCT    FGC    FGCT     GCT
0.0   7168.0  0.0   7168.0 48128.0  11264.0   993280.0   211719.5  25852.0 25435.9 3632.0 3484.8    192   13.038   0      0.000   13.038
Zookeeper
 S0C    S1C    S0U    S1U      EC       EU        OC         OU       MC     MU    CCSC   CCSU   YGC     YGCT    FGC    FGCT     GCT
0.0   3072.0  0.0   3072.0 24576.0   2048.0   496640.0    1680.5   10752.0 10416.0 1280.0 1188.5      2    0.208   0      0.000    0.208
```

where,

S0C	Current survivor space 0 capacity (KB).
S1C	Current survivor space 1 capacity (KB).
S0U	Survivor space 0 utilization (KB).
S1U	Survivor space 1 utilization (KB).
EC	Current eden space capacity (KB).
EU	Eden space utilization (KB).
OC	Current old space capacity (KB).
OU	Old space utilization (KB).
PC	Current permanent space capacity (KB).
PU	Permanent space utilization (KB).
YGC	Number of young generation GC Events.
YGCT	Young generation garbage collection time.
FGC	Number of full GC events.
FGCT	Full garbage collection time.
GCT	Total garbage collection time.

Disk Usage

Command: **iotop -oPa**
Disk Read : 21.54MB
Disk Write: 721.848MB

------- ------ --------
Machine Configuration:
processor: Intel 5, clock speed: 2.2 GHz, cores: 4
memory: 4GB

#There many other configuration parameters available for Kafka Producers, Consumers and Streams. Only the ones used are mentioned whenever needed.

Next Steps:

Streaming real time data feeds from twitter and wikipedia irc using kafka connect
Learning to use avro serializers, deserializers and schema registry