Capacity planning and sizing

Kafka Streams is simple, powerful streaming library built on top of Apache Kafka®. Under the hood, there are several key considerations to account for when provisioning your resources to run Kafka Streams applications. This section addresses questions like:

- How many Kafka brokers do I need to process 1 million records/second with a Kafka Streams application?
- How many client machines do I need to process those records with a Kafka Streams application?
- How much memory/CPU/network will a Kafka Streams application typically require?

Background and context

Here we recap some fundamental building blocks that will be useful for the rest of this section. This is mostly a summary of aspects of the Kafka Streams architecture that impact performance. You might want to refresh your understanding of sizing just for Kafka first, by revisiting notes on Production Deployment and how to choose the number of topics/partitions.

Kafka Streams uses Kafka's producer and consumer APIs under the hood a Kafka Streams application has Kafka producers and consumers, just like a typical Kafka client. So when it will come time to add more Kafka Streams instances, think of that as adding more producers and consumers to your app.

Unit of parallelism is a task In Kafka Streams the basic unit of parallelism is a stream task. Think of a task as consuming from a single Kafka partition per topic and then processing those records through a graph of processor nodes. If the processing is stateful, then the task writes to state stores and produces back to one or more Kafka partitions. To improve the potential parallelism, there is just one tuning knob: choose a higher number of partitions for your topics. That will automatically lead to a proportional increase in number of tasks.

Task placement matters: Increasing the number of partitions/tasks increases the potential for parallelism, but we must still decide where to place those tasks physically. There are two options: scale up, by putting all the tasks on a single server. This is useful when the app is CPU bound and one server has a lot of CPUs. You can do this by having an app with lots of threads (`num.stream.threads` config option, with a default of 1) or equivalently have clones of the app running on the same machine, each with 1 thread. There should not be any performance difference between the two. The second option is to scale out, by spreading the tasks across more than one machine. This is useful when the app is network, memory or disk bound, or if a single server has a limited number of CPU cores.

Load balancing is automatic: Once you decide how many partitions you need and where to start Kafka Streams instances, the rest is automatic. The load is balanced across the tasks with no user involvement because of the consumer group management feature that is part of Kafka. Kafka Streams benefits from it, because, as we mentioned earlier, it is a client of Kafka too in this context.

Armed with this information, let's look at a couple of key scenarios.

Stateless vs. stateful

Stateless applications

Here the CPU and network resources are key. These are applications that don't need to keep any state around. For example, they could be filtering, or performing some logic on the streaming data as it flows through the processor nodes, such as data conversion. Optionally these apps might write the final output back to a Kafka topic, however they do not have state stores (no aggregates or joins). Several stateless application examples can be found in the section on stream transformations.

In a way, sizing for stateless applications is similar to what you are doing today just with Kafka when sizing for adding more clients. T
difference is that the clients you are adding are Kafka Streams instances that may consume extra CPU resources because they are a
processing each record (in addition to just consuming it). So whereas before you were mainly concerned with the network, now you need to think

about the CPU as well.

For these apps, first determine whether the CPU or the network is a bottleneck by monitoring the OSes performance counters. If the CPU is the bottleneck, use more CPUs by adding more threads to your app (or equivalently by starting several clone instances on the same machine). If the network is a bottleneck, you need to add another machine and run a clone of your Kafka Streams app there. Kafka Streams will automatically balance the load among all the tasks running on all the machines.

Stateful applications

Here we need to monitor another two resources, local disks and memory. These are applications that perform aggregates and joins. Furthermore, local state can be queried by external applications any time through Interactive Queries. Several stateful application examples can be found in the section on stream transformations.

- **Local storage space (for fast queries)** Kafka Streams uses embedded databases to store local data. The default storage engine is RocksDB. If the local storage is running out of disk space, that is an indication that you need to scale out your application onto another machine. Adding more threads will not help in this case.
- **Global storage space (for fault tolerance)** In addition to writing locally, the data is also backed up to so-called changelog topics for which log compaction is enabled, so that if the local state fails, the data is recoverable. Hence, you need to provision Kafka to take into account traffic from S total state stores. Each state store adds one topic in Kafka (R-way replicated where the replication factor is controlled by setting `replication.factor`). Hence, you need to provision Kafka the same way you need to provision if a client were to write to S new topics. This might require adding more brokers to sustain the traffic (cf. examples below).
- **Global storage space (for internal operations)** Kafka Streams might store internal topics that are critical for its operations. One example of such internal topics is what is known as `repartition topics` that hold intermediate data (e.g., as a result of automatic re-partitioning of topics). Each time you remap the keys of a topic or join on a remapped key, a new repartition topic will be created with the same approximate size as the original topic whose keys are being remapped. Hence, you need to provision Kafka to take into account traffic from repartition topics. Note that repartition topics can also be replicated by setting `replication.factor`.
- **Memory (for performance)**: each state store has some memory overhead for buffering (seeMemory management). In addition, RocksDB also utilizes some off-heap memory to do compaction and its own buffering. If you find that one machine runs out of memory, you may have to increase the available RAM of the client machine, or scale out your application onto other machines. Adding more threads will not help in this case.
- **Standby replicas (for high availability)** To improve availability during failure, Kafka Streams tasks can be replicated by setting `num.standby.replicas` (the default is 0). Thestandby replicas do not perform any processing. They only consume from the changelog topics that back the state stores that the original task utilizes. Standby replicas place extra load on the system. They utilize network and storage resources client-side, as well as network resources at the brokers. You should consider enabling standby replicas if you are using Interactive Queries. Because, during task migration, your app won't be able to serve Interactive Queries for the migrated task and the state it manages. Standby replicas help to minimize the failover time in this situation.

Examples

Let's look at a couple of concrete sizing questions for stateless and stateful applications:

Scenario 1: Say you need to filter a stream of messages based on a predicate (e.g., that checks whether a string is a valid IP address) where each message is 100 bytes. Filtering is a stateless transformation. Assume you want to support a filtering rate of 10 million messages/second, i.e., approximately 1 GBps (in this scenario we assume Kafka is properly provisioned and can sustain that rate, and we focus on the Kafka Streams application). If each client machine is limited to a network rate of 500 MBps, then two client machines would be needed, each running one instance of the Kafka Streams application. Kafka Streams would automatically load balance the processing between the two instances.

Scenario 2: This is identical to scenario 1, except that in this case the Kafka Streams application outputs all the filtered messages to a new topic called `matchTopic`, by using the `.to()` operator. Assume that approximately half of the original messages are filtered. That means that 5 million messages/second need to be output to `matchTopic`. In this scenario, the application still demands an ingestion rate of 1 GBps, as well as an output rate of 0.5 GBps, for a total network load of 1.5 GBps. In this scenario, three Kafka Streams instances will be needed, on three different client machines.

Scenario 3: Say you need to process 1 million distinct keys with a key of size 8 bytes (a long) and a value of String (average 92 bytes) so we get about 100 bytes per message. For 1 million messages, you need 100 million bytes, i.e., roughly 100 MB to hold the state.

For window operations, you also need to take the number of windows plus retention time into account, e.g., assume a hopping time window with advance = 1 minute and retention time of 1 hour. Thus, you need to hold 60 windows per key. With the data from above, you end up with 60 times 100 MB which is roughly 6 GB.

So if you have 20 stores like this, you would need about 120 GB. If you have 100 partitions you end up up with 1.2 GB per partition on average assuming that data is evenly distributed across partitions. Thus, if you want to hold the state in main-memory and you have a 16 GB machine, you can put about 10 partitions per machine -- to be conservative and leave some headroom for other memory usage. Thus, you will need about 10 machines to handle all 100 partitions.

Note that in this scenario we wanted all the data in-memory. A similar calculation happens when the data is persisted (in RocksDB, for example). In that case, the relevant metric would be disk storage space.

Scenario 4: How many additional Kafka brokers do I need to process those 1 million requests/second (aggregate, join, etc) with a Kafka Streams application? This depends on the exact logic of the application and on the load of the current brokers. Let's say the number of current brokers is B, and the load that is being put on these B brokers to collect and store the (as yet un-processed) 1 million requests/second is L. A reasonable estimate is that the application changes this load L to anywhere between L (i.e., no increase, e.g., when the app is stateless) and 2 * L (e.g. when the app does some stateful processing). Now, the question how many additional brokers you might need depends on the current utilization of the existing brokers:

- If the existing brokers are 50% utilized, then the same brokers could sustain the new load with no additional brokers needed (note that, in practice, it might make sense to allow for some free space headroom and not utilize storage 100%).
- If the existing brokers are 100% utilized by handling the load L -- i.e. if there is no spare network and storage capacity at all in the existing Kafka brokers -- then you might need from 0 (L -> L) to B (L -> 2*L) additional brokers to store the new data from the state stores.

Scenario 5: How many client machines do I need to process those records with a Kafka Streams application? For good parallelism, the number of Kafka Streams instances must match the max number of partitions of the topics being processed. Say that number is P. If each machine has C cores, it can run one instance per core, so we would need P/C machines. However, the network could be a bottleneck, so -- unless the user is in a position to upgrade the networking setup, for example -- we would need the same number of client machines as the number of Kafka brokers that host the current partitions P in Kafka. So an upper bound is probably max(P/C, current number of brokers that hold P).

Scenario 6: Imagine a streaming use case (e.g. fraud detection, i.e. a typical "fast data lookup" scenario; or an "aggregation" use case, which requires tracking state) where, in the status quo, a traditional database cluster manages 10 TB of data replicated say 3 times (for a total space requirement of 30 TB). If we instead use Interactive Queries, how many application instances do we need?

First, the application now has a total state of 10 TB to locally manage across app instances (assuming no standby replicas). If each client machine has storage space of 1 TB, we would need at least 10 client machines. If failures are common, we might need more, because if a client machine fails, another application instance automatically takes over its load. So if we want to tolerate 1 failure, we'd need 11 client machines (or alternatively upgrade the local storage space to using disks larger than 1 TB).

In addition to local storage, the data is also stored on Kafka, and is replicated with the same replication factor as the original database cluster, i.e., Kafka brokers would need 30 TB of storage.

So at a first glance it looks like with Interactive Queries we are using more storage than with a remote database (10 TB local client-size state + 30 TB server-side Kafka state = 40 TB with Interactive Queries) vs. 30 TB of database state. However, keep in mind that the traditional database cluster is probably also using Kafka as an ingest engine (in the worst-case using an additional 30 TB of Kafka state) as well as using an internal database log (in the worst-case another 30 TB). Hence, in practice, the data stored with Interactive Queries is likely to have a smaller footprint than with a traditional database.

Troubleshooting

The store/RocksDB performance appears low. The workload might be IO bound. This happens especially when using a hard disk drive, instead

of an SSD. However, if you already use SSDs, check your client-side CPU utilization. If it is very high, it is likely you may need more cores for higher performance. The reason is that using state stores means writing to both RocksDB and producing to Kafka since state stores use changelog topics by default.

RocksDB's file sizes appear larger than expected RocksDB tends to allocate sparse files, hence although the file sized might appear large, the actual amount of storage consumed might be low. Check the real storage consumed (in Linux with the `du` command). If the amount of storage consumed is indeed higher than the amount of data written to RocksDB, then write amplification might be happening. For measuring and tuning write amplification see RocksDB's Tuning Guide.

The app's memory utilization seems high If you have many stores in your topology, there is a fixed per-store memory cost. E.g., if RocksDB is your default store, it uses some off-heap memory per store. Either consider spreading your app instances on multiple machines or consider lowering RocksDb's memory usage using the RocsDBConfigSetter class.

If you take the latter approach, note that RocksDB exposes several important memory configurations as first described in the section on other memory usage. In particular, these settings include `block_cache_size` (16 MB by default), `write_buffer_size` (32 MB by default) `write_buffer_count` (3 by default). With those defaults, the estimate per RocksDB store (let's call it `estimate per store`) is (`write_buffer_size_mb` * `write_buffer_count`) + `block_cache_size_mb` (112 MB by default).

Then if you have 40 partitions and using a windowed store (with a default of 3 segments per partition), the total memory consumption is 40 * 3 * `estimate per store` (in this example that would be 13440 MB).

Please report any inaccuracies on this page or suggest an edit.

Rate this page

Last updated on Sep 10, 2019.