Lecture notes: Scaling through partitioning in the Cloud

M2 MOSIG: Cloud Computing, from infrastructure to applications

Thomas Ropars

2023

To increase the throughput of a system using multiple nodes, we need to distribute the load. This is done by partitioning the data and assigning the partitions to different nodes.

In this course, we will study techniques to partition data, and we will look at the problems that need to be solved in a partitioned system.

1 Context

With partitioning approaches, each piece of data is included in a single partition. To some extend, each partition can be seen as an independent system. However, there might still be interactions between the partitions. Some operations on data (*e.g.*, range queries) may require accessing more than one partition. Also, if the size (or the composition) of the state evolves over time, re-partitioning may be required.

In this course, we are going to look at the following questions:

- What techniques are used to partition data?
- How to repartition a system?
- How to route client requests to the right node?

In the previous lecture, we studied replication. Replication can be applied independently in each partition. Considering a single-leader replication scheme, the typical way of combining replication and partitioning is the following:

- Each node will host multiple partitions (more precisely one replica of multiple partitions)
- A node will be the leader for some of the partitions it is hosting, and a follower for other.

In the following, we will ignore replication as it is mostly orthogonal to partitioning. Note that the word *sharding* is also often used to refer to partitioning.

2 Partitioning techniques

To study partitioning techniques, we will assume a simple model where each data item is identified by a unique key.

The objective of partitioning is to spread the load over the nodes that store partitions of the data. To optimize performance, the load should be spread evenly, that is, all nodes should serve a similar number of requests. If there are some *hot spots* in the system, *i.e.*, nodes that serve much more requests than the others, then the system will under-perform. The throughput will be limited by the hot spots while other nodes will be mostly idle.

One can imagine a simple approach to avoid hot spots: distributing the data randomly over the nodes. However, in this case, it becomes difficult for a client to know which node hosts a data item. Instead, we are going to study two approaches:

- Partitioning by key range
- Partitioning by hash keys

2.1 Partitioning by key range

With this approach, a range of the keys is assigned to each node. If a client knows the boundaries between the partitions and the schema for assigning partitions to nodes, it can easily figure out which node to contact to access a given item.

The example below illustrates a partitioning scheme based on the alphabetical ordering:

```
+---------------+---------------+---------------+---------------+
|  partition 1  |  partition 2  |  partition 3  |  partition 4  |
+---------------+---------------+---------------+---------------+
|      A-E      |      F-L      |      M-R      |      S-Z      |
+---------------+---------------+---------------+---------------+
```

In practice the load might be skewed: some keys may receive much more requests than others. For instance, if the keys are words from the natural language, it is probably the case that there will be more requests for the range `A-E` than for the range `V-Z`, although they cover the same number of letters. To avoid hot spots, the partitioning scheme may be adapted, as the example above shows.

To illustrate the advantages and drawbacks of partitioning by key range, we can consider a dataset composed on time series. We have multiple sensors that generate data continuously, and we need to partition these data. To this end, we use as key the timestamp associated with each data item, and we apply a partitioning scheme where the data for each day is stored in a different partition:

- The main advantage of such an approach is that range queries are very simple to implement. If one wants to access all data generated between 8AM and 11AM yesterday morning, she just needs to contact one node.
- The main drawback is that there will be a hot spot for data insertions since it it the same node that should store all the data from the current day.

2.2 Partitioning using a hash of the key

In this approach, a hash of each key is computed, and the data are partitioned based on the hash keys.

The hash function does not have to be *strong* from cryptographic point of view, but it should ensure that the generated values are evenly distributed over the range of possible hash keys. It means that the hash function should behave like a pseudo random number generator with an uniform distribution (but the function should obviously always generate the same hash for a given key).

Since the generated hash keys are uniformly distributed, each partition can be of the same size, as illustrated below assuming 2^{12} hash keys.

partition 1	partition 2	partition 3	partition 4
0-1023	1024-2047	2048-3071	3072-4095

Since the partitioning scheme is simple, it is easy to figure out where to find a data item based on the computed hash of the key, assuming no rebalancing can occur in the system. We will discuss request routing in more details later.

Such an approach has been adopted by many systems in practice. The main advantage and drawback are:

- This solution is a good approach to avoid hot spots
- Because a hash function is applied to the keys, range queries become difficult to implement. Several systems based on such an approach do not support range queries.

Compound primary keys To deal with the limitations of hash keys with respect to range queries, Apache Cassandra has adopted an approach based on *compound primary keys*. In this approach a key is composed of multiple fields, and the partitioning based on hashing is only applied to a subset of these fields.

To illustrate the approach, let us consider an example adapted from the one presented in `https://www.instaclustr.com/cassandra-data-partitioning/`. We assume that we need to store the logs generated by a set of sensors. Each log is identified with a timestamp. Different strategies can be applied for the definition of the key, leading to different results. In this exemple, the key is composed of 2 fields: The first field is used for hash partitioning (*partition key*), while the second field is used to order the data in each partition (*clustering key*).

- `key = < log_hour,` $\perp$ `>`: In this case, all logs generated during the same hour have the same *partition key*. All these logs will be in the same partition, simplifying range queries.
- `key = < log_hour, log_level >`: Same partitioning as before but the logs inside one partition are ordered based on their severity.
- `key = < (log_hour, sensor),` $\perp$ `>`: Logs are partitions based on the hour when they were generated and based on the sensor that generated them.
- `key = < (log_hour, sensor), log_level >`: Logs are partitioned based on time and sensor identifier, and are ordered inside each partition based on the severity.

These examples show that keys have to be chosen carefully taking into account the kind of queries that are expected for the data.

About skewed workloads If some keys are very popular compared to other keys in the system, even an approach based on hashing cannot prevent hot spots.

Consider a social network such as Twitter that may store data based on user identifiers. It has been stated that in 2010 the account of one famous pop-star was responsible for 3% of the total traffic on Twitter. Hot spots seem unavoidable with such a pattern. A possible solution is to add a suffix to the key that could be a random number. Let say that we choose a random number between 0 and 100, it means that the hot data can now be partitioned over 100 nodes. The downside is that read queries (`e.g.`, fetching the most recent tweets from this user) now require contacting 100 nodes.

3 Rebalancing partitions

Moving partitions between nodes is called *rebalancing*. Rebalancing may be required in different situations:

- The number of operations executed by the system increases (or decreases), and servers need to be added (or removed) to deal with these changes in the load.
- The dataset size increases (or decreases), requiring again changing the number of servers.
- Some server fails and other servers need to be integrated in the system to replace them.

The main requirements for rebalancing are the following:

- After rebalancing, the load should be fairly distributed over the servers
- The amount of data moved between servers should be minimized

3.1 Partitioning based on modulo

When using hash keys, a simple solution to adapt the partitioning scheme to the number of partitions is to use the modulo operator. In a configuration with N partitions, the partition hosting hash key K can be computed as $K \mod N$.

However such a solution is not efficient because lots of data need to be moved between partitions when the number of partitions changes. Consider an item with K=28:

- With 4 partitions, the item is in partition 0
- With 8 partitions, the item is in partition 4
- With 16 partitions, the item is in partition 12

3.2 Fixed number of partitions

A simple solution to deal with rebalancing is to initially create a number of partitions that is much larger than the number of nodes that is used.

In this case, if new nodes need to be added to the system, they can *steal* some partitions from existing nodes. The distribution of the partitions on the nodes can take into account the load in each partition. The relative capacity of nodes may also be taken into account in the assignment of partitions.

Several systems adopt such an approach. One important choice is then to select an appropriate number of partitions at the initialization of the system:

- Creating too few partitions limit the ability of the system to scale out
- Creating too many partitions creates overhead in the management of the partitions

A good trade-off needs to be found.

Having a fix number of partitions works well when partitioning is based on hash keys because of the uniform distribution that is guaranteed. However, with key range partitioning, it might be more difficult to apply because it might be difficult to decide on the boundaries of partitions a priori, and be sure to avoid hot spots.

3.3 Dynamic partitioning

An alternative approach is to use dynamic partitioning where decisions to create or merge partitions are taken dynamically based on the load.

One has to define two rules: i) the condition under which a partition should be split into two; ii) the condition under which two partitions should be merged. These conditions are typically based on the amount of data stored in the partitions.

Each partition is stored in one node and a node can store multiple partitions. When two partitions are created by splitting a large existing partition, one can be migrated to another node.

The advantage of dynamic partitioning is that the number of partitions adapts to the load. It avoids the overhead of having plenty of partitions to manage a small dataset like it can be the case with the previous approach. On the other hand, it might not provide enough parallelism if the dataset is small and the criteria to create multiple partitions are not met. One approach to deal with this issue is to initialize the system with multiple partitions from the beginning but in this case, it means that the boundaries of the initial partitions need again to be chosen carefully to avoid hot spots.

Dynamic partitioning can be used with key range partitioning and hash partitioning. MongoDB is an example of system that implements dynamic partitioning and supports key range and hash keys partitioning approaches.

4 Request routing

The last question related to partitioning that we are going to study is request routing. A client wants to access the item with key `ABC`. How do we ensure that the client request reaches the server hosting key `ABC`? Since the system might rebalance partitions, the question is not as simple as it might look at first sight.

The first point to discuss is who takes the routing decisions. There are three main approaches:

- The client can be in charge of learning about the partitions and of contacting the right server directly.
- An intermediate routing service can be introduced. All requests are sent to this service that should know which server can handle each request.
- The client can select a server randomly, and this server is then in charge of routing the request to the right server.

The main challenge that remains to be solved is to let the component in charge of taking routing decisions obtain information about the partitions.

Using a coordination service A coordination service is a database with strong consistency guarantees. Apache Zookeeper is an example of such a coordination service[1].

Several systems rely on a coordination service to manage partitions. The servers publish information about the partitions they are hosting in the coordination service. The routing agent can subscribe to the coordination service to be informed of any change in the configuration.

Using a gossip protocol Some systems, such as Apache Cassandra, prefer to avoid relying on an external service for dealing with the routing problem. Instead they allow a client to send a request to any node in the system. Then, this node should have enough information to forward the request to the right node.

To disseminate information about the configuration of the system to all the nodes, a gossiping protocol is used. A gossiping protocol basically works as follows:

- At regular time interval, a node select a few other nodes in the system (typically 3) to which it sends a message containing information about its current configuration.
- In each message, a node includes not only its own configuration, but also the configuration information it received from other nodes through gossip messages.

Such a mechanism ensures that all nodes become very fast aware of all the configuration changes in the system.

To go further

Some references can complement the material presented in these lecture notes:

- Chapter 6 of *Designing Data-Intensive Applications* by Martin Kleppmann
- A post about data partitioning in Cassandra:
  `https://www.instaclustr.com/cassandra-data-partitioning/`

[1]Note that Zookeeper does not provide linearizability