hazelcast.org

Compare Redis and Hazelcast • Hazelcast

11-13 minutes

Rethinking Redis?

Hazelcast and Redis at first glance are very similar. They can tackle similar use cases, so it may be hard to decide which to use. On this page, we'll try to describe some of the differences.

We'll focus in on 5 important categories;

1. Caching
2. Clustering
3. Querying
4. Streaming
5. Compute

Firstly we should say that Redis is a very popular open-source project and it is used widely. If your use case is very simple caching and you do not need clustering, querying & compute then you should stick with Redis. As a simple single instance cache, it is a great choice.

Obviously, we think Hazelcast is pretty special as well. So let's try to look into how things are different and provide some information that may help you in your evaluation.

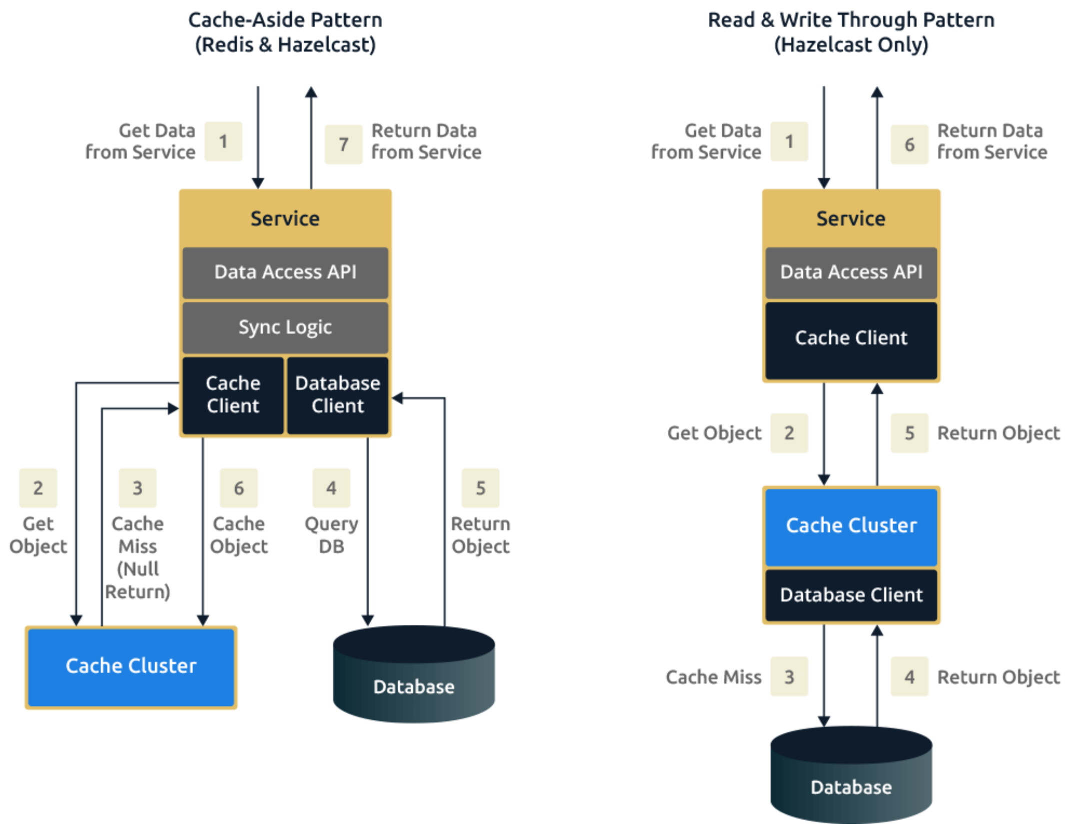
Caching

Hazelcast provides a wider variety of caching patterns.

The most common use case for Hazelcast IMDG and Redis is caching. When we talk about caching, we usually mean we are holding data in-memory that comes from a slower store which is usually disk-bound. This could be a Relational Database, Mainframe, NoSQL Database or another Application API. Because we would like to speed up access to this data we cache the items once we have read them from the slower store.

The biggest difference between Hazelcast and Redis for caching use cases is that Redis forces the use of one caching pattern, whilst Hazelcast provides a number of patterns. Using Redis as a cache over another store like a database forces the use of the cache-aside pattern; this introduces extra network hops. Hazelcast, by comparison, can be configured to handle read-through on a cache miss and write-through on updates. With Redis, the responsibility is on the developer to write the cache/database synchronization logic and also the code to update/read the database. For Hazelcast, only the update/read logic is required, which makes the code base much cleaner and easier to understand. Hazelcast is capable of handling cache-aside if required, but can also handle read-through, write-through, and write-behind caching patterns. Redis only allows for cache-aside.

The diagram below shows the request flow using a cache-aside pattern versus that of read-through and write-through. The important thing to note is that the amount of logic and libraries required in your service accessing the cache is greatly reduced with read-through and write-through. The cache process itself handles this logic and the connections to the database with read-through and write-through, whereas cache-aside means your service has to handle this. Imagine many different applications all having to handle access to the database and the cache, this could get very complicated and hard to handle in production. Instead, it is much simpler operationally to let the distributed cache handle this. These patterns make it much easier to implement a Cache as a Service



Hazelcast Cache-Aside vs. Read-Write
Note the extra network hop and also the extra cache/DB sync logic required in the cache-aside pattern.

Lastly, let's talk about write-behind. This is a caching pattern that is available in Hazelcast. Think of it as an extension to the write-through pattern. Write-behind solves the problem of writing to a slow backing store, for example, an overloaded relational database. With write-behind, the service or application submits the data to the distributed cache and it returns an acknowledgment once the data arrives but before it is committed to the backing store. The distributed cache places the data in a queue. This queue is read by a separate thread at some point later and the distributed cache takes care of submitting the data to the backing store such as the relational database. In this way, a slow backing store does not introduce latency to the application. In contrast, the write-through pattern blocks the acknowledgment back to the caller until the backing store has confirmed its commit. Because Redis is single-threaded it is unable to provide any of these options.

Clustering

Most clustering operations are handled automatically by Hazelcast. Redis requires manual intervention.

The ways in which Redis & Hazelcast provide clustering is quite different. Clustering is a technology that provides scalability and increased space and processing power for data. It is used when one process is not powerful enough or where redundancy (often referred to as High Availability or HA) is required. Clustering works by co-ordinating many processes together to form a cluster. A Cluster can span many machines to provide redundancy against one machine failure. The processes in the cluster are often referred to as nodes or members. For Redis clustering is used to make use of multiple cores on a machine, this is encouraged because Redis is single-threaded. Hazelcast is multi-threaded, so clustering for Hazelcast is a matter of making use of more memory.

Hazelcast was built from day one as an in-memory distributed data store; Redis was not. This becomes obvious when you compare the way in which clustering and sharding/partitioning are handled. High availability, auto-partitioning, and auto-discovery are built into the core of each Hazelcast member. Read the Getting Started With Hazelcast IMDG guide to see how easy it is to create and scale a Hazelcast cluster in comparison to Redis.

Each Hazelcast member in a cluster takes care of a portion of primary and replica entries. There is no concept of a master process and backup processes. This makes reasoning about your architecture much easier. With Redis, there is the concept of master nodes and backup nodes. Redis Clustering provides HA and scalability by sharding, it splits larger datasets across multiple nodes. Hazelcast also provides this, expect it is called partitioning. However, when a node leaves a Redis cluster, manual resharding and recovery are required. Adding and removing nodes in a Redis cluster is also a manual process. All of the above are handled by Hazelcast automatically.

Querying

Hazelcast understands complex object graphs and provides querying API, Redis does not.

Redis and Hazelcast both provide Key/Value structures, but they work quite differently when you want to query. By query we mean the ability to retrieve data when you do not know the key, so you are querying by specific properties of the value.

The most fundamental difference is that Hazelcast is able to store complex objects and understand the object graph. Redis is unable to do this, in order to reason about graphs the developer has to model the graph in a series of key/value entries where part of the key represents a property and its value. Redis does not provide the ability to division data using concepts such as tables, everything is stored in one namespace, e.g. the Database. Hence the need to come up with complex namespace schemes within keys. This StackOverflow post describes this approach.

On the other hand, Hazelcast provides a Predicate API and SQL like where clauses and projections to query out data. These Querying API can be used on Complex Objects and JSON. Hazelcast also has a more flexible namespace in that you can have many Maps and name these appropriately, for example, Customer, Invoice, Orders. This then negates the need to pollute the key namespace with these concerns, your keys can just describe the actual value being saved.

Here’s a code sample of a SQL like query on a Hazelcast Map, you’ll see it’s in Java but the concept is the same in all the languages supported by Hazelcast. For more information view our documentation.

```
IMap<String, Employee> map = hazelcastInstance.getMap( "employee" );
Set employees = map.values( new SqlPredicate( "active AND age < 30" ) );
```

NOTE: Starting with Hazelcast IMDG 4.1 there will be support for full ANSI SQL Querying.

Finally, Redis does not natively support indexes, instead, the application programmer has to create their own index structures and update these themselves, as well as referring to them. This is described in the Redis Documentation.

Hazelcast has native support for indexes. They can be applied via configuration (XML|YAML) or dynamically via the API. Indexes can be applied at any level of the object graph. Compound Indexes are supported. For more information view our documentation.

Streaming

Redis supports pub-sub messaging to distribute streaming data whilst Hazelcast comes with a full streaming stack.

A Streaming platform requires multiple building blocks:

1. Messaging to distribute streaming data
2. Processing API to build continuous applications over data streams
3. Connectors to integrate with legacy non-streaming systems

Since version 5, Redis has Redis Streams: a pub-sub based messaging. Redis Streams are an append-only log-based storage that preserves insertion ordering and allows non-destructive reading. Multiple consumers can read and replay the data multiple times, receiving the same data in the same order. Replaying the ordered stream is fundamental for fault-tolerance and scalability in streaming making Redis Streams a good fit for the messaging layer in the streaming stack.

Hazelcast brings all three components that are vital for streaming. Firstly, Hazelcast Jet is a distributed stream processing engine with support for stateful operations. You can use it to join or aggregate data streams by using event-time semantics and by keeping end-to-end exactly-once processing guarantees. Jet comes with many connectors including a CDC that converts relational database transactions to a stream of change events. In a frequent use-case, the connectors stream data from databases, JMS or Kafka. Jet runs a continuous query triggered by new data events and pushes results to the cache, keeping it fresh.

Compute

Redis supports Lua Scripts whilst Hazelcast allows Java and soon Python & C++.

Aside from the language differences, both Redis & Hazelcast can direct compute functions directly to a node/member of the cluster that holds specific data. For example, in Hazelcast a Java program can be directed to the member of a cluster holding a specific key. Something similar is possible in Redis. One difference leading on from this pattern is that the Hazelcast Java program is free to reach out from the member it is running on, to access data on another member in the cluster. Redis Clustering and Lua Scripting are unable to achieve this.

Secondly, Hazelcast provides an Entry Processor pattern. This is a framework that allows a function to be run against data in a Map, Hazelcast takes care of running this function across all members of the cluster or it can be targeted to cluster members that satisfy a key predicate.

NOTE: This compute section concerns itself with Redis and Hazelcast IMDG. For more complex batch and streaming compute see the section above on Streaming that compares Hazelcast Jet to Redis Stream Data Type.