

Redis

1. Caching best practice

Local caches

A local cache stores your frequently-used data within your application. This makes data retrieval faster than with other caching architectures because it removes network traffic that is associated with retrieving data.

A major disadvantage is that among your applications, each node has its own resident cache working in a disconnected manner. **The information that is stored in an individual cache node (whether it's cached database rows, web content, or session data) can't be shared with other local caches. This creates challenges in a distributed environment where information sharing is critical to support scalable dynamic environments.**

Because most applications use multiple application servers, coordinating the values across them becomes a major challenge if each server has its own cache. In addition, when outages occur, the data in the local cache is lost and must be rehydrated, which effectively negates the cache. The majority of these disadvantages are mitigated with remote caches.

Remote caches

A remote cache (or *side cache*) is a separate instance (or separate instances) dedicated for storing the cached data in-memory. Remote caches are stored on dedicated servers and are typically built on key/value NoSQL stores, such as [Redis](#) and [Memcached](#). They provide hundreds of thousands of requests (and up to a million) per second per cache node. Many solutions, such as [Amazon ElastiCache for Redis](#), also provide the high availability needed for critical workloads.

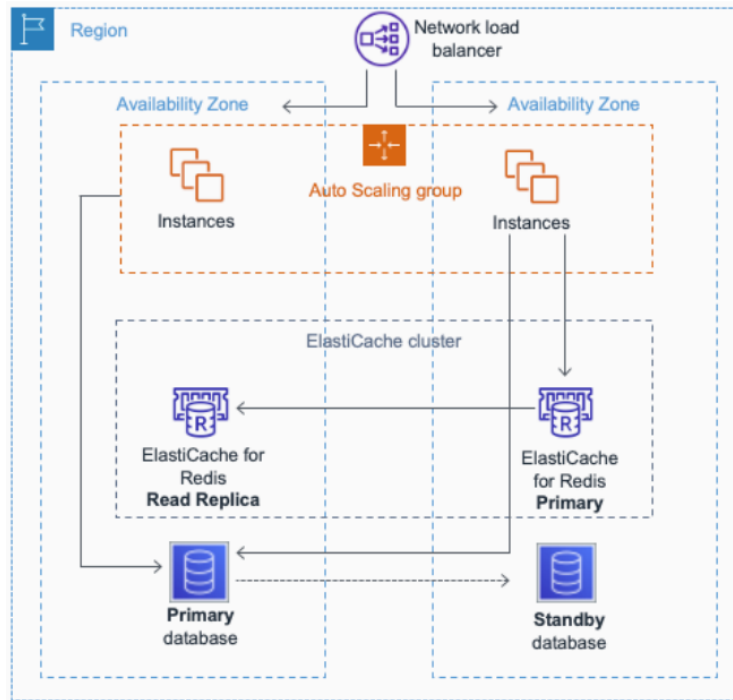
The average latency of a request to a remote cache is on the sub-millisecond timescale, which, in the order of magnitude, is faster than a request to a disk-based database. At these speeds, local caches **are seldom necessary**. Remote caches are ideal for distributed environments because they work as a connected cluster that all your disparate systems can utilize. However, when network latency is a concern, you can apply a two-tier caching strategy that uses a local and remote cache together. This

Caching patterns

When you are caching data from your database, there are caching patterns for [Redis](#) and [Memcached](#) that you can implement, including proactive and reactive approaches. The patterns you choose to implement should be directly related to your caching and application objectives.

Two common approaches are cache-aside or **lazy loading (a reactive approach)** and **write-through (a proactive approach)**. A cache-aside cache is updated after the data is requested. A write-through cache is updated immediately when the primary database is updated. **With both approaches, the application is essentially managing what data is being cached and for how long.**

The following diagram is a typical representation of an architecture that uses a remote [distributed cache](#).



```

...
// rs contains the ResultSet, key contains the SQL statement
if (rs != null) { //lets write-through to the cache
    CachedRowSet cachedRowSet = new CachedRowSetImpl();
    cachedRowSet.populate(rs, 1);
    ByteArrayOutputStream bos = new ByteArrayOutputStream();
    ObjectOutputStream out = new ObjectOutputStream(bos);
    out.writeObject(cachedRowSet);
    byte[] redisRSValue = bos.toByteArray();
    jedis.set(key.getBytes(), redisRSValue);
    jedis.expire(key.getBytes(), ttl);
}
...

```

2. Redis data structure

- STRINGS
- LISTS
- SETs
- HASHes
- ZSETs.

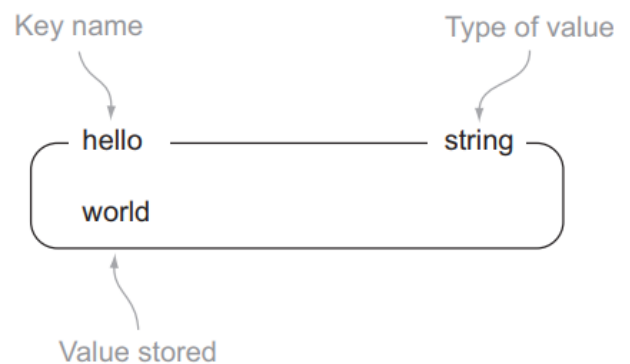
Table 1.2 The five structures available in Redis

Structure type	What it contains	Structure read/write ability
STRING	Strings, integers, or floating-point values	Operate on the whole string, parts, increment/decrement the integers and floats
LIST	Linked list of strings	Push or pop items from both ends, trim based on offsets, read individual or multiple items, find or remove items by value
SET	Unordered collection of unique strings	Add, fetch, or remove individual items, check membership, intersect, union, difference, fetch random items
HASH	Unordered hash table of keys to values	Add, fetch, or remove individual items, fetch the whole hash
ZSET (sorted set)	Ordered mapping of string members to floating-point scores, ordered by score	Add, fetch, or remove individual values, fetch items based on score ranges or member value

Strings:

Strings in Redis

In Redis, **STRINGS** are similar to strings that we see in other languages or other key-value stores. Generally, when I show diagrams that represent keys and values, the diagrams have the key name and the type of the value along the top of a box, with the value inside the box. I've labeled which part is which as an example in figure 1.1, which shows a **STRING** with key `hello` and value `world`.



Command	What it does
GET	Fetches the data stored at the given key
SET	Sets the value stored at the given key
DEL	Deletes the value stored at the given key (works for all types)

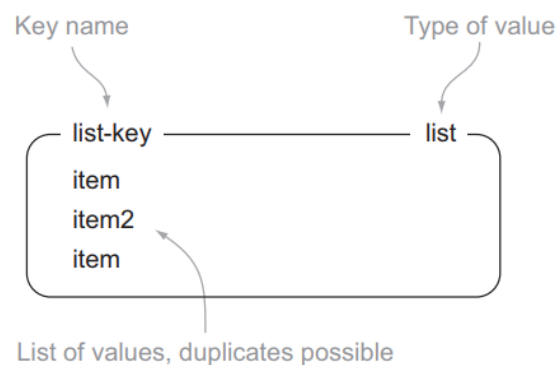
Command	Example use and description
INCR	INCR key-name—Increments the value stored at the key by 1
DECR	DECR key-name—Decrements the value stored at the key by 1

Command	Example use and description
INCRBY	INCRBY key-name amount—Increments the value stored at the key by the provided integer value
DECRBY	DECRBY key-name amount—Decrements the value stored at the key by the provided integer value
INCRBYFLOAT	INCRBYFLOAT key-name amount—Increments the value stored at the key by the provided float value (available in Redis 2.6 and later)

Lists:

In the world of key-value stores, Redis is unique in that it supports a linked-list structure. LISTS in Redis store an ordered sequence of strings, and like STRINGS, I represent figures of LISTS as a labeled box with list items inside. An example of a LIST can be seen in figure 1.2.

The operations that can be performed on LISTS are typical of what we find in



Command	What it does
RPUSH	Pushes the value onto the right end of the list
LRANGE	Fetches a range of values from the list
LINDEX	Fetches an item at a given position in the list
LPOP	Pops the value from the left end of the list and returns it

Sets in Redis

Command	What it does
SADD	Adds the item to the set
SMEMBERS	Returns the entire set of items
SISMEMBER	Checks if an item is in the set
SREM	Removes the item from the set, if it exists

Hashes in Redis

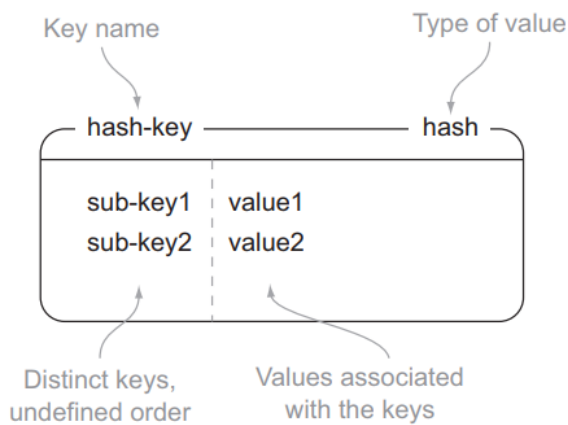


Table 1.6 Commands used on HASH values

Command	What it does
HSET	Stores the value at the key in the hash
HGET	Fetches the value at the given hash key
HGETALL	Fetches the entire hash
HDEL	Removes a key from the hash, if it exists

Publish/subscribe

Table 3.11 Commands for handling pub/sub in Redis

Command	Example use and description
SUBSCRIBE	<code>SUBSCRIBE channel [channel ...]</code> —Subscribes to the given channels
UNSUBSCRIBE	<code>UNSUBSCRIBE [channel [channel ...]]</code> —Unsubscribes from the provided channels, or unsubscribes all channels if no channel is given
PUBLISH	<code>PUBLISH channel message</code> —Publishes a message to the given channel
PSUBSCRIBE	<code>PSUBSCRIBE pattern [pattern ...]</code> —Subscribes to messages broadcast to channels that match the given pattern
PUNSUBSCRIBE	<code>PUNSUBSCRIBE [pattern [pattern ...]]</code> —Unsubscribes from the provided patterns, or unsubscribes from all subscribed patterns if none are given

3. Persistence options

Within Redis, there are two different ways of persisting data to disk. One is a method called **snapshotting** that takes the data as it exists at one moment in time and writes it to disk. The other method is called **AOF, or append-only file**, and it works by copying incoming write commands to disk as they happen. These methods can be used together, separately, or not at all in some circumstances. Which to choose will depend on your data and your application.

Listing 4.1 Options for persistence configuration available in Redis

```
save 60 1000
stop-writes-on-bgsave-error no
rdbcompression yes
dbfilename dump.rdb
```

**Snapshotting
persistence options**

```
appendonly no
appendfsync everysec
no-appendfsync-on-rewrite no
auto-aof-rewrite-percentage 100
auto-aof-rewrite-min-size 64mb
```

**Append-only file
persistence options**

**Shared option,
where to store
the snapshot or
append-only file**

```
dir ./
```

DEVELOPMENT

For my personal development server, I'm mostly concerned with minimizing the overhead of snapshots. To this end, and because I generally trust my hardware, I have a single rule: **save 900 1**. The save option tells Redis that it should perform a BGSAVE operation based on the subsequent two values. In this case, if at least one write has occurred in at least **900 seconds (15 minutes) since the last BGSAVE**, Redis will automatically start a new BGSAVE.

If you're planning on using snapshots on a production server, and you're going to be storing a lot of data, you'll want to try to run a development server with the same or similar hardware, the same save options, a similar set of data, and a similar expected load. By setting up an environment equivalent to what you'll be running in production, you can make sure that you're not snapshotting too often (wasting resources) or too infrequently (leaving yourself open for data loss).

4. Redis Persistence

RDB	AOF	Hybrid (RDB + AOF)
Performs point in time snapshot of data at the specified interval.	Keeps log for every write operation performed by Redis	
save 60 1000	Define fsync policies , always, everysec	

Within Redis, there are two different ways of persisting data to disk. One is a method called *snapshotting* that takes the data as it exists at one moment in time and writes it to disk. The other method is called *AOF*, or *append-only file*, and it works by copying incoming write commands to disk as they happen. These methods can be used together, separately, or not at all in some circumstances. Which to choose will depend on your data and your application.

Listing 4.1 Options for persistence configuration available in Redis

```
save 60 1000
stop-writes-on-bgsave-error no
rdbcompression yes
dbfilename dump.rdb

appendonly no
appendfsync everysec
no-appendfsync-on-rewrite no
auto-aof-rewrite-percentage 100
auto-aof-rewrite-min-size 64mb

dir ./
```

**Snapshotting
persistence options**

**Append-only file
persistence options**

**Shared option,
where to store
the snapshot or
append-only file**




Snapshots:

Advantages
Compact, Single file of Redis Data
Perfect for backups
Performance is good
Allows Faster Restarts
Disadvantage
Chances of data loss
Fork Process, may impact write operations for few milliseconds

AOF:

Advantages

- High Durability
- Allows different fsync policies
- Append only logs, no chances of corruption
- Corrupted file can be fixed by redis-check-aof tool
- AOF File can be parsed, lines can be removed 

Disadvantage

- Size is bigger than RDB
- Slower than RDB

Snapshotting

By default Redis saves snapshots of the dataset on disk, in a binary file called `dump.rdb`. You can configure Redis to have it save the dataset every N seconds if there are at least M changes in the dataset, or you can manually call the `SAVE` or `BGSAVE` commands.

For example, this configuration will make Redis automatically dump the dataset to disk every 60 seconds if at least 1000 keys changed:

```
save 60 1000
```

This strategy is known as *snapshotting*.

Command for snapshot:

```
save 900 1    # every 15 minutes if at least one key changed
save 300 10   # every 5 minutes if at least 10 keys changed
save 60 10000 # every 60 seconds if at least 10000 keys changed
```

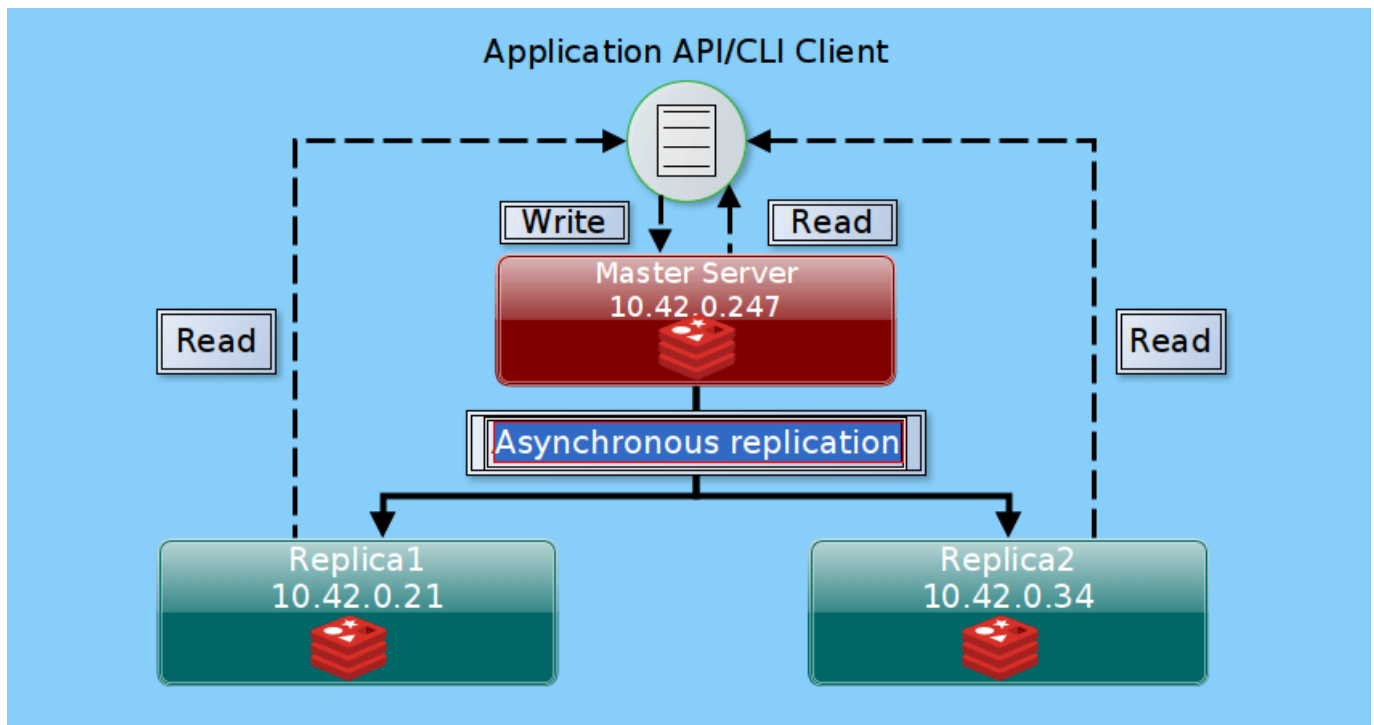
BGSAVE

```
cd /var/lib/redis
dump.rdb
```


AOF:

```
/var/lib/redis
```

5. Setup Master Slave configuration



Setup master configuration:

- Change in redis.conf

```
bind 127.0.0.1 172.31.23.8
min-replicas-to-write 2
min-replicas-max-lag 10
```

Setup slave configuration:

```
replicaof 172.31.23.8 6379
masterauth admin12345
```

Only Master can set new key, slave is read only.

6. Sentinel

```
sudo systemctl restart redis-sentinel
```

7. Lua Script

Some sample script:

```
local logtable = {}

local function logit(msg)
    logtable[#logtable+1] = msg
end

logit("ID from Phone")
local id = redis.call('HMGET', 'user_phone', ARGV[1])
logit('Found user from phone number: ' .. id[1])

local user = redis.call('HMGET', 'user', id[1])
logit('Found user data: ' .. user[1])

return logtable
```

```
local logtable = {}

local function logit(msg)
    logtable[#logtable+1] = msg
end

logit("All Keys-----")
for i = 1, #KEYS do
    logit(KEYS[i])
end

logit("All Args-----")
for i = 1, #ARGV do
    logit(ARGV[i])
end

return logtable
```

Run command:

```
redis-cli -a admin12345 --eval /home/ubuntu/redis/script/load_id_from_phone.lua name1
, 1111
```

8. Redis in action

Name	Type	Data storage options	Query types	Additional features
Redis	In-memory non-relational database	Strings, lists, sets, hashes, sorted sets	Commands for each data type for common access patterns, with bulk operations, and partial transaction support	Publish/Subscribe, master/slave replication, disk persistence , scripting (stored procedures)
memcached	In-memory key-value cache	Mapping of keys to values	Commands for create, read, update, delete, and a few others	Multithreaded server for additional performance
MySQL	Relational database	Databases of tables of rows, views over tables, spatial and third-party extensions	SELECT, INSERT, UPDATE, DELETE, functions, stored procedures	ACID compliant (with InnoDB), master/slave and master/master replication
PostgreSQL	Relational database	Databases of tables of rows, views over tables, spatial and third-party extensions, customizable types	SELECT, INSERT, UPDATE, DELETE, built-in functions, custom stored procedures	ACID compliant, master/slave replication, multi-master replication (third party)
MongoDB	On-disk non-relational document store	Databases of tables of schema-less BSON documents	Commands for create, read, update, delete, conditional queries, and more	Supports map-reduce operations, master/slave replication, sharding, spatial indexes

Other feature of Redis:

- Persistence:
 - Snapshot
 - Write Log
- Support Master Slave

Redis data structure:

Structure type	What it contains	Structure read/write ability
STRING	Strings, integers, or floating-point values	Operate on the whole string, parts, increment/decrement the integers and floats
LIST	Linked list of strings	Push or pop items from both ends, trim based on offsets, read individual or multiple items, find or remove items by value
SET	Unordered collection of unique strings	Add, fetch, or remove individual items, check membership, intersect, union, difference, fetch random items
HASH	Unordered hash table of keys to values	Add, fetch, or remove individual items, fetch the whole hash
ZSET (sorted set)	Ordered mapping of string members to floating-point scores, ordered by score	Add, fetch, or remove individual values, fetch items based on score ranges or member value

Redis Persistence:

- Snapshot:
- Append only

redis.conf

```

save 60 1000 # save if 60 second or 1000 write
stop-writes-on-bgsave-error no
rdbcompression yes
dbfilename dump.rdb

appendonly no
appendfsync everysec
no-appendfsync-on-rewrite no
auto-aof-rewrite-percentage 100
auto-aof-rewrite-min-size 64mb
dir ./

```

Redis Replication

Step	Master operations	Slave operations
1	(waiting for a command)	(Re-)connects to the master; issues the SYNC command
2	Starts BGSAVE operation; keeps a backlog of all write commands sent after BGSAVE	Serves old data (if any), or returns errors to commands (depending on configuration)
3	Finishes BGSAVE ; starts sending the snapshot to the slave; continues holding a backlog of write commands	Discards all old data (if any); starts loading the dump as it's received
4	Finishes sending the snapshot to the slave; starts sending the write command backlog to the slave	Finishes parsing the dump; starts responding to commands normally again
5	Finishes sending the backlog; starts live streaming of write commands as they happen	Finishes executing backlog of write commands from the master; continues executing commands as they happen

Redis transaction:

- MULTI: enter transaction
- EXEC: all command are called when EXEC is called
- DISCARD: exit transaction
- WATCH

All the commands in a transaction are serialized and executed sequentially. A request sent by another client will **never be served in the middle of the execution of a Redis Transaction**.

Execute Increment of Foo and Bar in a single atomic operation:

```
> MULTI
OK
> INCR foo
QUEUED
> INCR bar
QUEUED
> EXEC
1) (integer) 1
2) (integer) 1
```