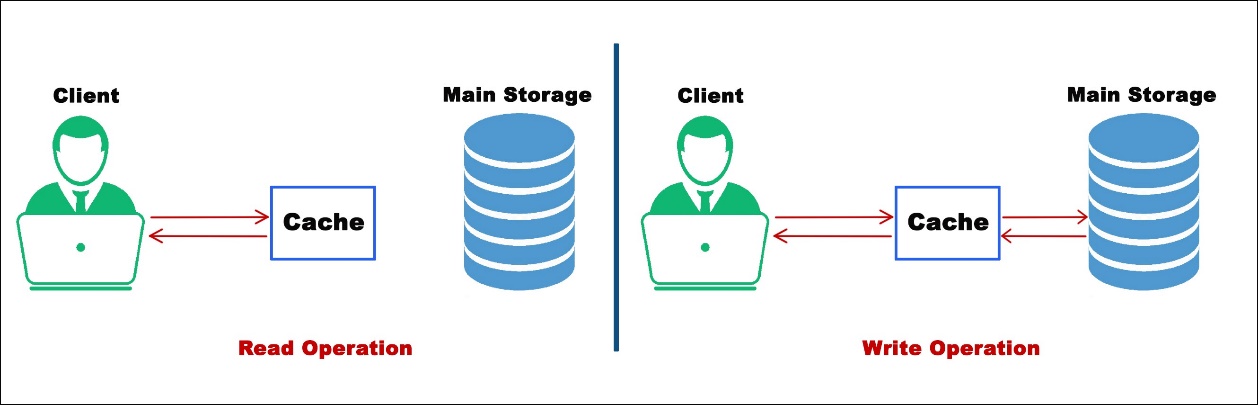
A **distributed cache** is a system that pools together the random-access memory (RAM) of multiple networked computers into a single in-memory data store used as a data cache to provide fast access to data. While most caches are traditionally in one physical server or hardware component, a distributed cache can grow beyond the memory limits of a single computer by linking together multiple computers

A Cache is like short-term memory. It is typically faster than the origin data source. You know Accessing data from RAM is always faster than accessing it from the hard drive.

**Cache Invalidation**

It does require some maintenance for keeping cache **coherent** with the source of truth (e.g., database). If any data is modified in the database, it should be invalidated or modified in the cache also; if it is not so, this can cause inconsistent application behaviour.  
A cache can be invalidated in the following ways.

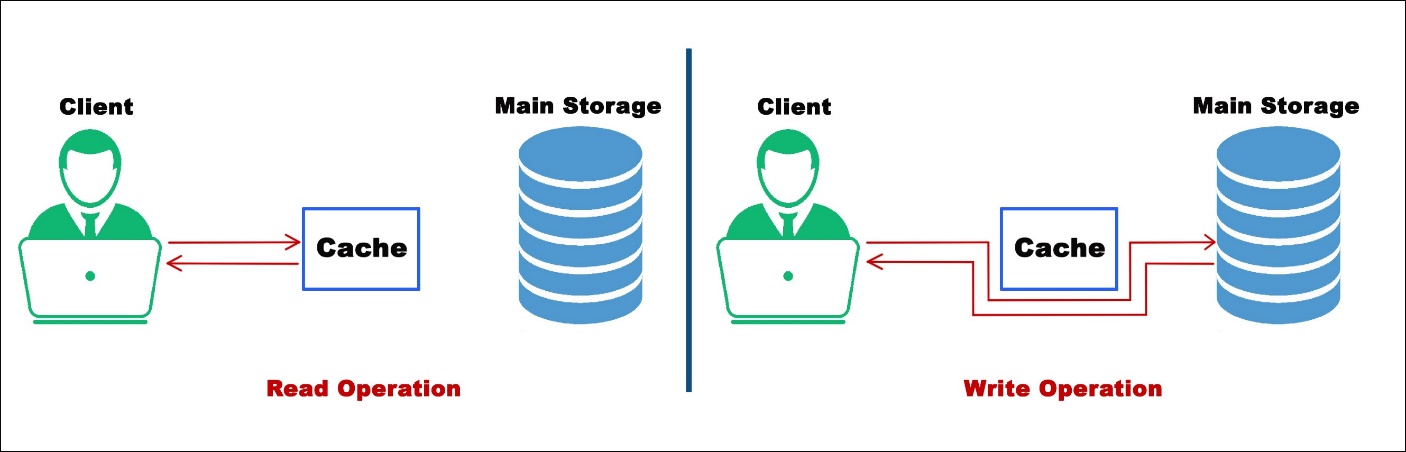
**Read-Through →** The data is written into the cache and the corresponding database at the same time. This scheme maintains the complete data ***consistency*** between the cache and the main storage. Even this scheme also ensures that nothing will get lost in case of a crash, power failure, or any other system disruptions.



Read-Through Cache

→ The disadvantage of this scheme is the Higher Latency, since every write operation is done twice before returning success to the client😔

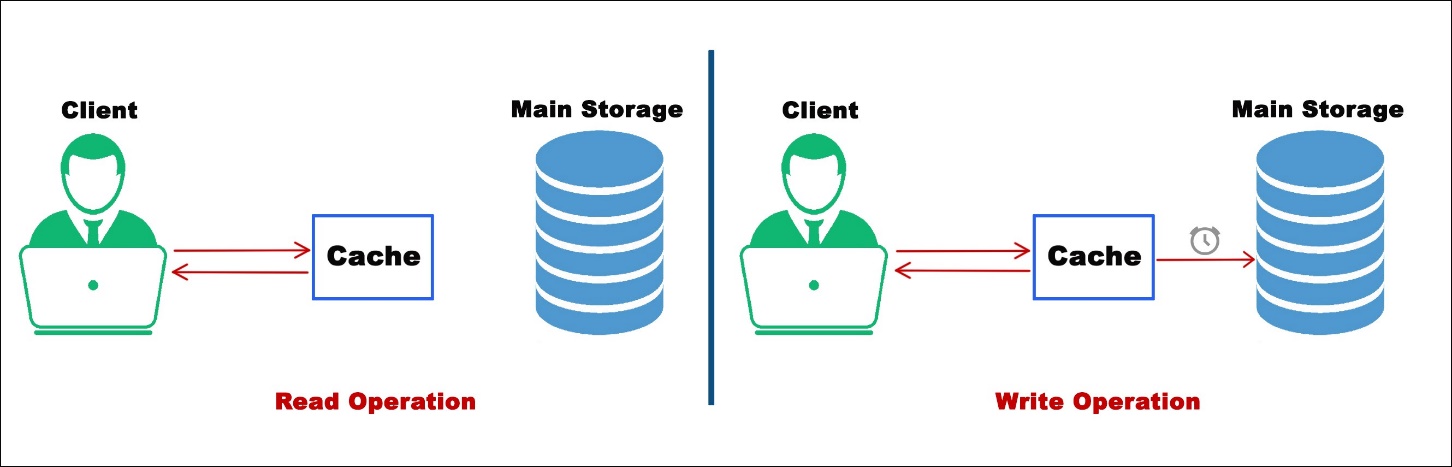
**Write-Through →** In this strategy, every information directly written to the database just bypassing the cache.



Write-Through-Cache

→ The disadvantage of this scheme is the Higher Latency if in case of a read request for recently written data will create a **“cache miss”**😯 and then the read request must be made on back-end storage.

**Write-Back →**The data is written to cache only and completion is immediately confirmed to the client. The write to the permanent storage is done after specified intervals or under certain conditions. This results in low latency and high throughput for write-intensive applications.



Write-Back-Cache

→ However, this speed comes with the risk of data loss in case of a crash or other adverse event because the only copy of the written data is in the cache.

# Cache Eviction Policies

What if our Cache is full? So, A cache eviction algorithm is a way of deciding which element to evict when the cache is full. The following are some of the most common cache eviction policies:

1. **First In First Out (FIFO):** The cache evicts the first block accessed first without any regard to how often or how many times it was accessed before.
2. **Last In First Out (LIFO):** The cache evicts the block accessed most recently first without any regard to how often or how many times it was accessed before.
3. **Least Recently Used (LRU):** Discards the least recently used items first.  
   I mostly prefer to LRU only. 🙂
4. **Most Recently Used (MRU):** Discards, in contrast to LRU, the most recently used items first.
5. **Least Frequently Used (LFU):** Counts how often an item is needed. Those that are used least often are discarded first.

The popular distributed caches used in the industry are **Eh-cache, Memcache, Redis, Riak, Hazelcast**.

**Redis Cache**

**RE**mote **DI**ctionary **S**erver

Redis is an open source, advanced key-value store and an apt solution for building high performance, scalable web applications

SET and GET Operations:

redis 127.0.0.1:6379> SET name "tutorialspoint"

OK

redis 127.0.0.1:6379> GET name

"tutorialspoint"

**Key value pair:** name is key value is tutorialspoint

**Note** − A string value can be at max 512 megabytes in length.

## **Hashes**

A Redis hash is a collection of key value pairs. Redis Hashes are maps between string fields and string values. Hence, they are used to represent objects.

redis 127.0.0.1:6379> HMSET user:1 username tutorialspoint password tutorialspoint points 200

**OK**

redis 127.0.0.1:6379> HGETALL user:1

1) "username"

2) "tutorialspoint"

3) "password"

4) "tutorialspoint"

5) "points"

6) "200"

Every hash can store up to 232 - 1 field-value pairs (more than 4 billion).

## **Sets**

Redis Sets are an unordered collection of strings. In Redis, you can add, remove, and test for the existence of members in O(1) time complexity.

redis 127.0.0.1:6379> sadd tutoriallist redis

(integer) 1

redis 127.0.0.1:6379> sadd tutoriallist mongodb

(integer) 1

redis 127.0.0.1:6379> sadd tutoriallist rabitmq

(integer) 1

redis 127.0.0.1:6379> sadd tutoriallist rabitmq

(integer) 0

redis 127.0.0.1:6379> smembers tutoriallist

1) "rabitmq"

2) "mongodb"

3) "redis"

**Note** − In the above example, **rabitmq** is added twice, however due to unique property of the set, it is added only once.

To run commands on Redis remote server, you need to connect to the server by the same client **redis-cli**

### **Syntax**

$ redis-cli -h host -p port -a password

**Using java:**

* You need to include the **jedis.jar** into your classpath.
* import redis.clients.jedis.Jedis;
* public class RedisJava {
* public static void main(String[] args) {
* //Connecting to Redis server on localhost
* Jedis jedis = new Jedis("localhost");
* System.out.println("Connection to server sucessfully");
* //check whether server is running or not
* System.out.println("Server is running: "+jedis.ping());
* }
* //Connecting to Redis server on localhost
* Jedis jedis = new Jedis("localhost");
* System.out.println("Connection to server sucessfully");
* //store data in redis list
* jedis.lpush("tutorial-list", "Redis");
* jedis.lpush("tutorial-list", "Mongodb");
* jedis.lpush("tutorial-list", "Mysql");
* // Get the stored data and print it
* List<String> list = jedis.lrange("tutorial-list", 0 ,5);
* for(int i = 0; i<list.size(); i++) {
* System.out.println("Stored string in redis:: "+list.get(i));
* }
* }

**Spring boot with Redis:**

@EnableCaching in main start application

Whatever we need to cache we use

@Cacheable(value = "itemCache")

**public** Item **getItemForId**(String id) {

**return** itemRepository.findById(id).orElseThrow(RuntimeException::**new**);

}

1. **For remote connection:**   
    Jedis jedis=new Jedis("192.168.31.237",6379);
2. public class RedisJava {
3. public static void main(String[] args) {
4. *//Connecting to Redis server on localhost*
5. Jedis jedis = new Jedis("192.168.56.129",6379);
6. jedis.auth("199701");
7. System.out.println("Connection to server sucessfully");
8. *//check whether server is running or not*
9. System.out.println("Server is running: "+jedis.ping());
10. }
11. }

|  |  |
| --- | --- |
| **Memcached** | **Redis** |
| Memcached only does caching information. | It provides some more functionalities like replication and persistence along with caching information. |
| Memcached supports the functionality of LRU (Least Recently Used) eviction of values. | LRU is not supported by Redis. |
| In Memcached, when they overflow memory, the one you have not used recently (LRU- Least Recently Used) will get deleted. | In Redis, there is a time set for each function, Three keys are maintained, the one, which is closest to expiry, will get deleted. |
| CAS (Check and Set) is supported by Memcached. | CAS is not supported by Redis. |
| Array objects are needed to be serialized in order to get saved. We need to unserialize them for their retrieval. | Redis has got stronger data structures; it can handle strings, binary safe strings, list of binary safe strings, sorted lists, etc. |
| Memcached has at most 250 bytes length. | Redis has at most 2 GB key length. |
| It is Multi-threaded | It is single threaded. |

* **What is the difference between RDB and AOF in Redis?**

**RDB is snapshot-based; AOF logs each write. AOF is safer for durability, RDB is faster for recovery.**

What is Redis Sentinel?

Sentinel provides high availability — monitors, notifies, and auto-fails over masters.

How do you implement distributed locks in Redis?

Use SET key value NX PX timeout or libraries like RedLock.

What is a Redis Stream and how is it used?

Redis Streams are for event streaming, like Kafka-lite: consumer groups, IDs, etc.

How does Redis handle concurrency?

Redis is single-threaded; commands are atomic. However, clients must manage race conditions.