

# Large-Scale and Multi-Structured Databases

## ***Key – Value Databases: Redis***

### ***Part 1***

Prof Pietro Ducange

Eng. José Corcuera

# Objective of this Class

- To learn what Redis is and how it works.
- To learn the **best practices to define keys**.
- To review the different **datatypes** existing in Redis.
- To review the **commands** offered by Redis.
- To learn what a **Pipelining** is and when to use it.
- To learn how to start a **Transaction**.
- **Java connector for Redis**.
- Use case scenarios.
- Exercises.

# What is Redis?

- Redis is an in-memory data structure store used as a database.
- It can be used also as a cache, message broker and streaming engine.
- Redis provides different data structures such as strings, lists, etc.
- It offers atomic operations like incrementing the value in a hash, pushing an element to a list, etc.
- Periodically data persistence feature (dumping the dataset to disk). This feature can be disabled.
- Redis also includes: Transactions, Pub/Sub, keys with limited time-to-live, replication, scaling and many other features.

# Installing Redis

- Redis installer is available for Linux, MacOS and Windows.
- Visit this link to get instructions of how to install Redis: <https://redis.io/docs/getting-started/installation/>
- By default, Redis allows connections from 127.0.0.1. To allow connections from anywhere, update your **/etc/redis/redis.conf** file:
  - Replace "bind 127.0.0.1" with "bind 0.0.0.0"
  - Once you update your **redis.conf** file, it is mandatory to restart the service.
- To establish a connection to your Redis server make use of **redis-cli** program.
  - Run: "**redis-cli -h**" to see all the parameters this program accepts.

# Redis Keys (1)

- Keys are unique in Redis. They are used to locate a specific value.
- Keys are binary safe (you can use a string or the content of a JPEG file).
- Empty string is also valid a key.
- Use a namespace to identify your applications keys (sometimes the same Redis instance can be used by other applications). Example:
  - **sales-app:invoice:10012:status** = "PAID"
  - **shopping-cart:1002** = "[{product\_id: 1000, quantity: 10}]"
- Very long keys are not a good idea (Use hashing, i.e.: SHA1).

# Redis Keys (1)

- Keys must be readable. For example: "I need to define a key to store the quantity of followers for a user"
  - **Bad: u1000flw**
  - **Good: user:1000:followers**
- Dots or dashes are often used for multi-word fields:
  - comment:4321:reply.to
  - shopping-cart:5431:updated-date
- The maximum allowed key size is 512 MB (**do not reach that size please**).

# Redis datatypes

- Redis supports different datatypes:

hello world	String
011011010110111101101101	Bitmap
{23334}{6634728}{916}	Bitfield
{a: "hello", b: "world"}	Hash
[A>B>C>C]	List
{A<B<C}	Set
{A:1, B:2, C:3}	Sorted set
{A: (50.1, 0,5)}	Geospatial
01101101 01101111 01101101	Hyperlog
{id1=time1.seq(( a: "foo", a: "bar")) }	Stream

We are going to work with only String datatype.



What is the difference between Lists and Sets?

Image taken from: <https://architecturenotes.co/redis/>

# Redis commands (1)

- **To define** a new key-value pair:  
**set** <key> <value>
- **To retrieve** the value from a key:  
**get** <key>
- **To define the expiration on a key:**  
**expire** <key> <seconds> **NX**  
**set** <key> <value> **EX** <seconds>
- **To see the time to live of a key:**  
**ttl** <key>
- **To delete a key:**  
**del** <key>

```
jose@uss-defiant:~$ redis-cli
127.0.0.1:6379> set hello 'hello world from redis'
OK
127.0.0.1:6379> get hello
"hello world from redis"
127.0.0.1:6379> expire hello 10
(integer) 1
127.0.0.1:6379>
127.0.0.1:6379> get hello
(nil)
127.0.0.1:6379> █
```



# Redis commands (2)

- **Scenario:** There is a counter entry with the value 10 and two clients want to increase this value at the same time.
- **Problem:** Both clients could read the value 10 at the same time and set the value to 11 when the expected value should be 12.
- **Solution:** To make use of atomic operations like INCR, INCRBY, DECR and DECRBY or TRANSACTIONS (we are going to see it in a while).

```
127.0.0.1:6379> set counter 10
OK
127.0.0.1:6379> incr counter
(integer) 11
127.0.0.1:6379> incr counter
(integer) 12
127.0.0.1:6379> █
```

*All available commands can be found here: <https://redis.io/commands/>*

# Pipelining

- It is a technique for improving performance.
- Multiple commands are sent at once without waiting for the response to each of them.

Without pipelining:

*Client: INCR X*

*Server: 1*

*Client: INCR X*

*Server: 2*

*Client: INCR X*

*Server: 3*

*Client: INCR X*

*Server: 4*

VS.

With pipelining:

*Client: INCR X*

*Client: INCR X*

*Client: INCR X*

*Client: INCR X*

*Server: 1*

*Server: 2*

*Server: 3*

*Server: 4*

**RTT is very low.**

**Round Trip Time (RTT):** Time of a packet to travel from a client to a server and back from the server to a client to carry the response.

# Transactions (1)

- Execution of a group of commands in a single step.
- All the commands defined in a transaction are serialized and executed sequentially.
- Commands to be used: **MULTI**, **EXEC**, **DISCARD** and **WATCH**.
- **EXEC** triggers the execution of all commands in a transaction.
- Redis does not support rollbacks. What you can do is to **DISCARD** a previous sent command.
- Keys involved in a transaction may be modified by other transactions. Use **WATCH** to monitor the state of these keys and, in case some of them are modified during the execution of the transaction, this will be aborted.

# Transactions (2)

```
jose@uss-defiant:~$ redis-cli
127.0.0.1:6379> watch my-key
OK
127.0.0.1:6379> MULTI
OK
127.0.0.1:6379> set my-key 20
QUEUED
127.0.0.1:6379> set my-key "modified by client 1"
QUEUED
127.0.0.1:6379> EXEC
(nil)
127.0.0.1:6379> get my-key
"modified by client 2"
127.0.0.1:6379> █
```

Client 1

```
jose@uss-defiant:~$ redis-cli
127.0.0.1:6379> watch my-key
OK
127.0.0.1:6379> MULTI
OK
127.0.0.1:6379> set my-key 12
QUEUED
127.0.0.1:6379> set my-key "modified by client 2"
QUEUED
127.0.0.1:6379> EXEC
1) OK
2) OK
127.0.0.1:6379> get my-key
"modified by client 2"
127.0.0.1:6379> █
```

Client 2

- Client 2, queue a set of commands and execute them first.
- After client 2 commands execution, client 1 executes their commands.
- Since my-key was modified first by client 2, client 1 transaction is aborted.

# Jedis: Java client for Redis

- It is a Java library for connecting to Redis.
- Designed for performance and ease to use.
- GitHub repository: <https://github.com/redis/jedis>
- Commands are implemented in methods defined the **redis.clients.jedis.Jedis** class.
- The API specification can be found here: <https://javadoc.io/doc/redis.clients/jedis/latest/index.html>.
- It is available in Maven too.

```
<dependency>  
  <groupId>redis.clients</groupId>  
  <artifactId>jedis</artifactId>  
  <version>4.3.0</version>  
</dependency>
```

# Jedis in action (1)

- Establishing a connection to a Redis server:

```
import redis.clients.jedis.Jedis;
```

```
String redisHost = "localhost";
```

```
Integer redisPort = 6379;
```

```
Jedis jedis = new Jedis(redisHost , redisPort);
```

```
// do stuff here
```

```
jedis.close();
```

Obtaining a single connection to Redis.

Remember to close the connection.  
Also, you can use try-with-resources.

# Jedis in action (2)

- **Connecting to Redis server by using a connection pool:**

```
import redis.clients.jedis.Jedis;
```

```
String redisHost = "localhost";
```

```
Integer redisPort = 6379;
```

```
JedisPool pool = new JedisPool(redisHost, redisPort);
```

```
try (Jedis jedis = pool.getResource()) {
```

```
    /*
```

```
    do stuff here
```

```
    */
```

```
}
```

```
...
```

```
pool.close();
```

Using a pool,  
improve the  
performance of  
Redis operations.

Try-with-resources.

Closing the pool.

# Jedis in action (3)

- Once you get an instance of a Jedis object, you can execute many commands:

```
String key = "my-app:some-key";
```

```
jedis.set(key, "Hello world!");
```

SET command.

```
jedis.expire(key, 10);
```

EXPIRE command.

```
String value = jedis.get(key);
```

GET command.

```
long ttl = jedis.ttl(key);
```

TTL command.



# Jedis in action (4)

- We can send multiple commands within a transaction and abort it in case watched keys have been modified.

```
String key1 = "my-app:some-key";  
String key2 = "my-app:key-2";
```

**WATCH command.**

If this key is modified, the transaction is aborted.

```
jedis.watch(key1);
```

**MULTI command.**

```
Transaction transaction = jedis.multi();  
transaction.set(key1, "New value for key1");  
transaction.set(key2, "New value for key2");
```

**EXEC command.**

It will return a list with output values of each command sent in the transaction.

```
List<Object> result = transaction.exec();
```

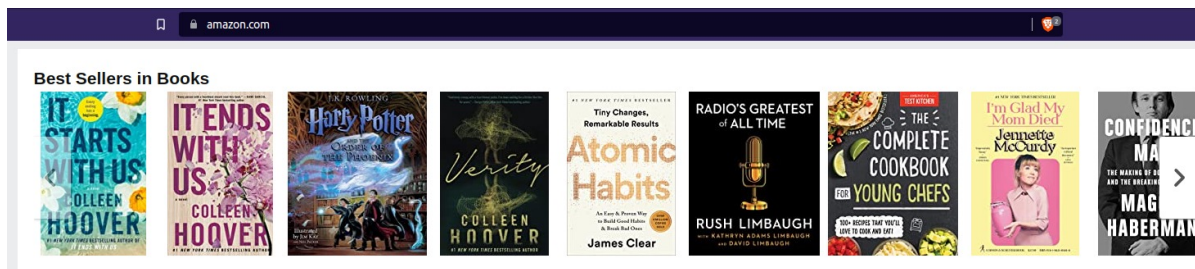
**The execution of the transaction returns null when keys marked with WATCH have been modified.**

# Jedis in action - Exercise

- Create a Java application that defines 3 keys with these possible values:
  - For key1: any numeric value.
  - For key2: any String.
  - For key3: the JSON string serialization of a Java object (make use of the GSON library).
- The namespace of your application is "app1".
- After creating these keys, get their values from Redis and print them on the console.
- You don't need to add GSON maven dependency since Jedis already includes it.

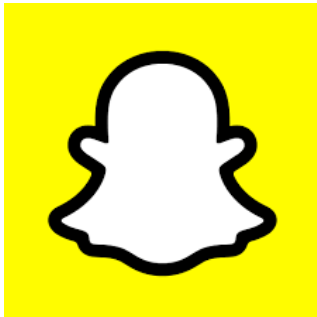
# When to use Redis?

- When an application has to deal with **volatile data** (i.e: shopping cart).
- When the data is always queried but **hardly ever modified** (i.e: list of countries).
- When you have to **access data by using only an identifier and you want to reduce response time** (i.e. accessing to a popular product detail page in Amazon, "best sellers in books section").



# Who's using Redis?

- You can check here: <https://techstacks.io/tech/redis>



craigslist

# Use case scenario 1: Shopping cart

The screenshot shows the Trenitalia shopping cart interface. On the left, a sidebar titled 'CART' contains a 'JOURNEY' section for 'Pisa Centrale - Roma Termini' on '14.10.2022 (15:09)' with a price of '66,90€'. Below this is a '+ ADD ANOTHER JOURNEY' button. At the bottom of the sidebar, the 'TOTAL CART' is '66,90€ (VAT included)' and a 'GO TO PAYMENT' button is present. The main area displays the selected journey: 'Pisa Centrale (15:09) - Roma Termini (18:10)' for '14 Oct \* 1 Adult'. It lists two ticket types: 'FAST REGIONAL - 4040' for '8,90€' and 'FRECCIARGENTO - 8519' for '58,00€'. Both are for '1 Adult' in '2ª Classe'. The total price '66,90€' is shown in red at the top right of the main area. A 'Time left' indicator shows '08:54' with a red circle and an 'X' icon, indicating a 10-minute timer. A blue arrow points from the text 'Tickets in the shopping cart are removed after 10 minutes.' to this timer.

Tickets in the shopping cart are removed after 10 minutes.

Other shopping carts do not remove items (i.e. Amazon).

Website: <https://www.trenitalia.com/>

# Use case scenario 2: Rate limiting

## API Plans

Fixed limit plans based on daily request quotas and billed monthly

Currency: EUR ▼


Plans	Free	Tier 1	Tier 2	Tier 3
Cost / per month	0	9 EUR	45 EUR	220 EUR
Daily Limits	Number of requests you can make per 24 hours			
Data Tools				
Email Validate	50	10K	100K	1M
Phone Validate	50	10K	100K	1M
UA Lookup	50	10K	100K	1M
Bad Word Filter	50	10K	100K	1M
WWW				
Browser Bot	10	100	1K	10K
HTML Clean	10	1K	10K	100K
URL Info	10	1K	10K	100K
Geolocation				
IP Info	50	10K	100K	1M
Geocode Address	25	1K	10K	100K
Geocode Reverse	25	1K	10K	100K

Daily request quotas per day.

Website: <https://www.neutrinoapi.com/>

# Use case scenario 3: Online auction/bid

**Lot 1** HUGE PAIR ANTIQUED CAST STONE DOGS ON PALLET  
by John Cowell Auctions



1/1

**Estimate**  
No Estimate

**Fees** ⓘ

Seleziona lingua ▼ Powered by Google Traduttore

HUGE PAIR ANTIQUED CAST STONE DOGS ON PALLET

**Timed Auction**  
Auction Ends: 14th Oct 22 from 12pm BST

**Register to Bid**

**Additional Fees** ▼

**Time Left:** 4h 51m  
**Current Bid:** 350 GBP


**My Max Bid:** None

Min: 375 GBP

**Place Bid**

**Reserve:** Not Met  
**Bids:** 1  
**Watching:** 3

**Other Lots in this Auction**



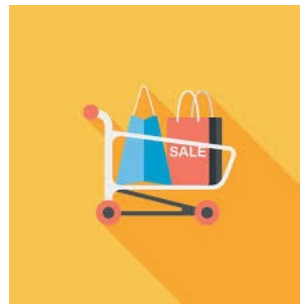
The max bid is the winner at the end of the auction.

Website: <https://www.easyliveauction.com/>

# Exercise 1: Shopping cart

***Create a Java application that registers 10 products in a shopping cart. Your application must:***

- ***Define the proper namespace.***
- ***A shopping cart is associated to a user, so use the value 1 as userId.***
- ***You can buy more than one specific product (l.e.: 10 coca cola of 33 cl).***
- ***Items in the shopping cart are removed after 60 seconds.***
- ***After adding the 10 products, implement a method to retrieve them.***
- ***Also, it is required to know when the shopping cart was updated and what the current number of items is.***





# Exercise 2: Hospital's ticketing system

*Patients in a hospital need to get a ticket to have access to medical attention services. The hospital opens from 08:00 – 12:00 every day and they can serve 100 tickets so, 100 patients. Let's implement a Java application that simulates this scenario so your Java application must:*

- *Defines a class TicketManager with a method get ticket number. This method returns an integer value starting from 1 up to 60. When there is no possible to return a ticket, it will return -1 (or an Exception).*
- *Creates 10 threads. Each of these threads is going to get a ticket (by calling the method defined in the previous point) every second in an interval of 30 seconds. The value obtained is printed in the console.*

# References

- <https://redis.io/docs/>
- <https://redis.io/docs/manual/transactions/>
- <https://redis.com/blog/5-key-takeaways-for-developing-with-redis/>