TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

Macroarea di Ingegneria
Dipartimento di Ingegneria Civile e Ingegneria Informatica

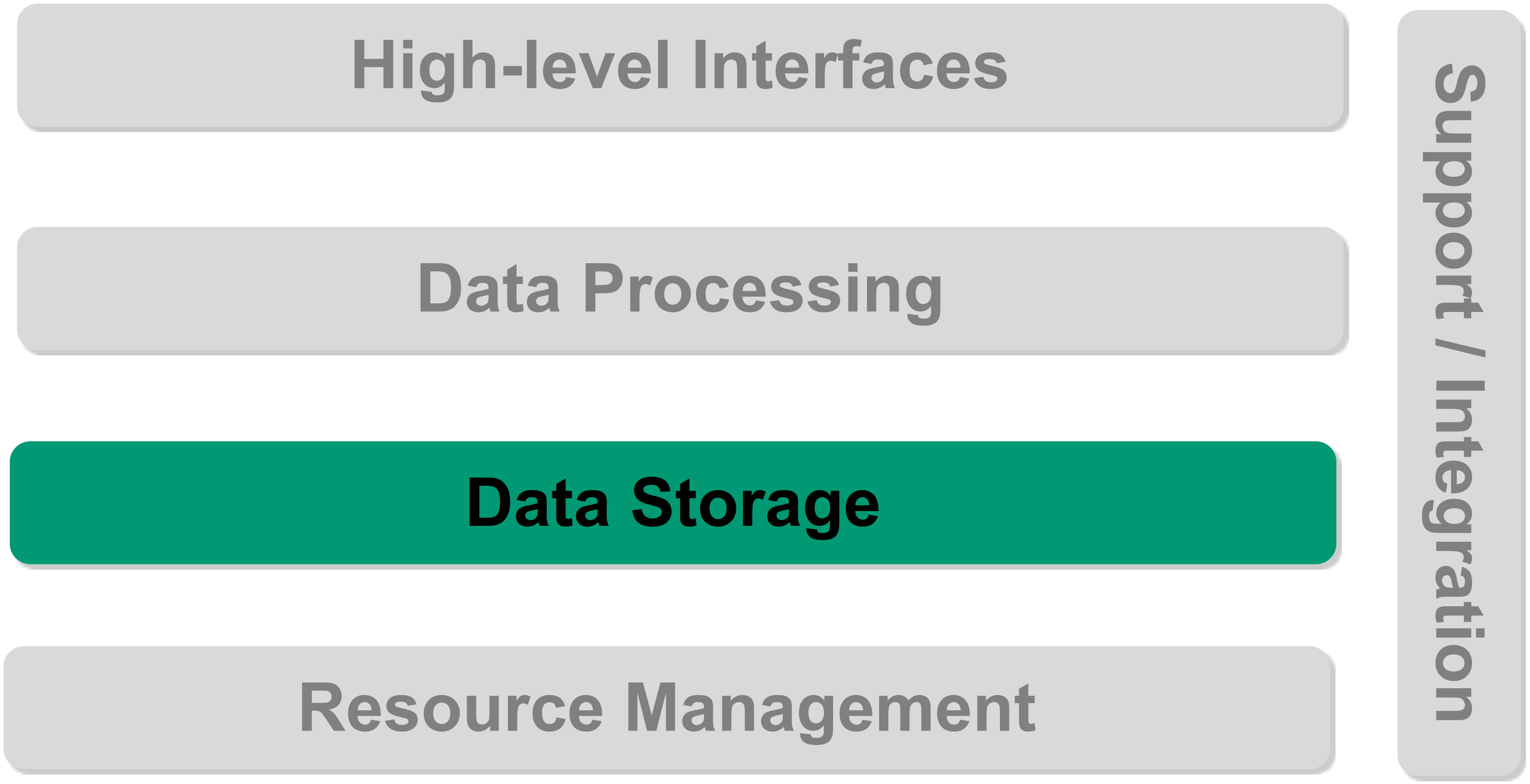
NoSQL: Redis

A.A. 2023/24

Matteo Nardelli

Laurea Magistrale in
Ingegneria Informatica - II anno

The reference Big Data stack



NoSQL data stores

Main features of NoSQL (**Not Only SQL**) data stores:

- Support flexible schema
- Scale horizontally
- Provide scalability and high availability by storing and replicating data in distributed systems
- Do not typically support ACID properties, but rather BASE

Simple APIs

- Low-level data manipulation and selection methods
- Queries capabilities are often limited

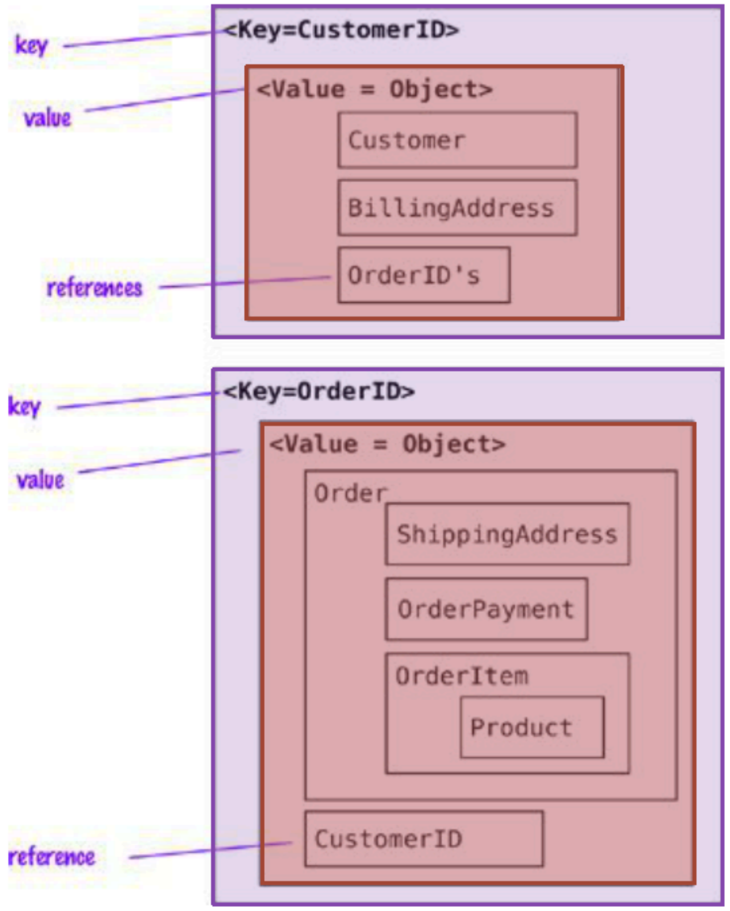
Data models for NoSQL systems:

- Aggregate-oriented models:
  key-value, **document**, and **column-family**
- **Graph-based** models

Key-value data model

- Simple data model:
  - data as a collection of key-value pairs
- Strongly aggregate-oriented
  - A set of <key,value> pairs
  - Value: an aggregate instance
  - A value is mapped to a **unique** key
- The aggregate is opaque to the database
  - Values do not have a known structure
  - Just a big blob of mostly meaningless bit
- Access to an aggregate:
  - Lookup based on its key
- Richer data models can be implemented on top

Key-value data model: example



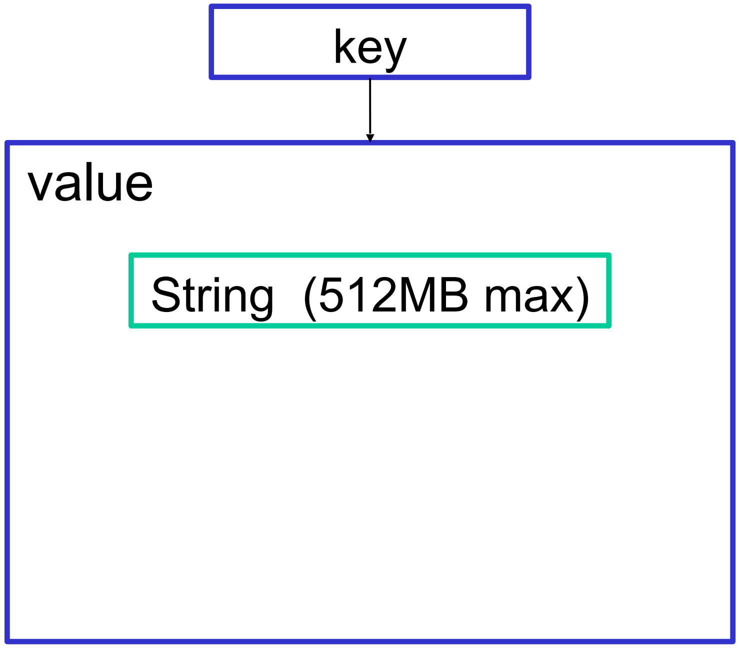
Redis

- REmote DIrectory Server
  - An (in-memory) key-value store.

- Redis was the most popular implementation of a key-value database as of March 2022, according to DB-Engines Ranking (link).

Data Model

- Key: Printable ASCII
- Value:
  - Primitives: **Strings**
  - Containers (of strings):
    - Hashes
    - Lists
    - Sets
    - Sorted Sets



https://redis.io/topics/data-types

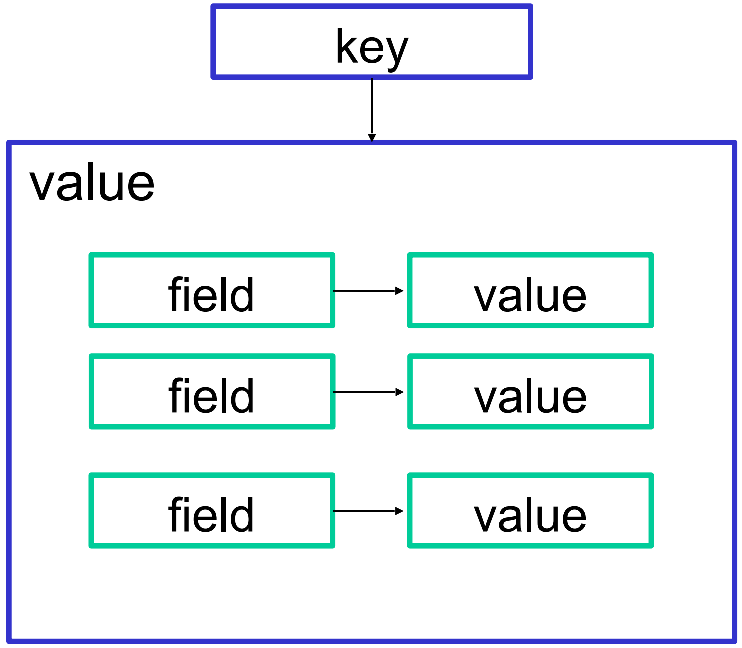
Redis

- REmote DIrectory Server
  - An (in-memory) key-value store.

- Redis was the most popular implementation of a key-value database as of March 2022, according to DB-Engines Ranking (link).

Data Model

- Key: Printable ASCII
- Value:
  - Primitives: Strings
  - Containers (of strings):
    - **Hashes**
    - Lists
    - Sets
    - Sorted Sets



https://redis.io/topics/data-types

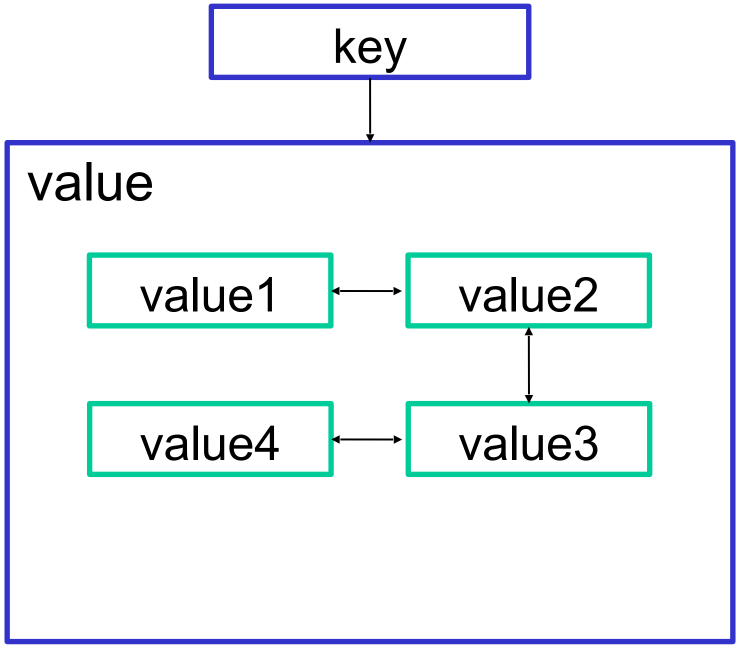
Redis

- REmote DIrectory Server
  - An (in-memory) key-value store.

- Redis was the most popular implementation of a key-value database as of March 2022, according to DB-Engines Ranking (link).

Data Model

- Key: Printable ASCII
- Value:
  - Primitives: Strings
  - Containers (of strings):
    - Hashes
    - **Lists**
    - Sets
    - Sorted Sets



https://redis.io/topics/data-types

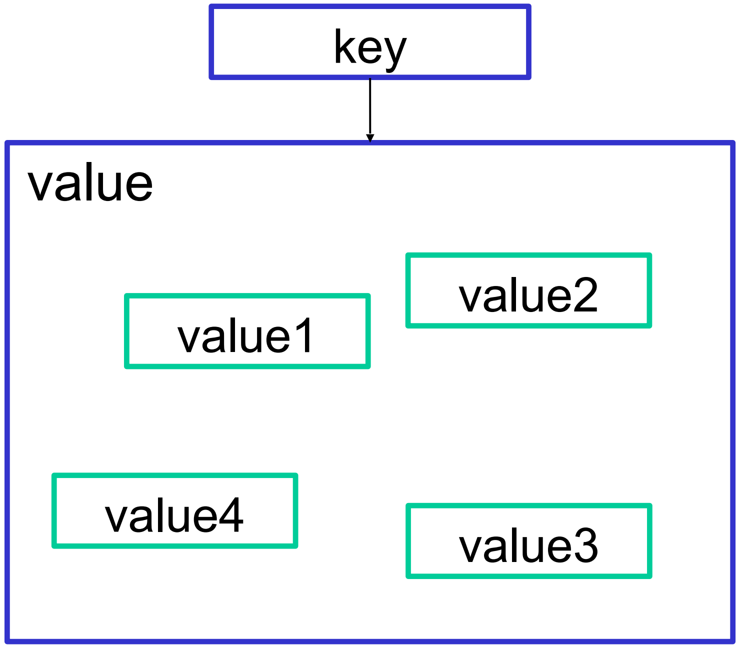
Redis

- REmote DIrectory Server
  - An (in-memory) key-value store.

- Redis was the most popular implementation of a key-value database as of March 2022, according to DB-Engines Ranking (link).

Data Model

- Key: Printable ASCII
- Value:
  - Primitives: Strings
  - Containers (of strings):
    - Hashes
    - Lists
    - **Sets**
    - Sorted Sets



https://redis.io/topics/data-types

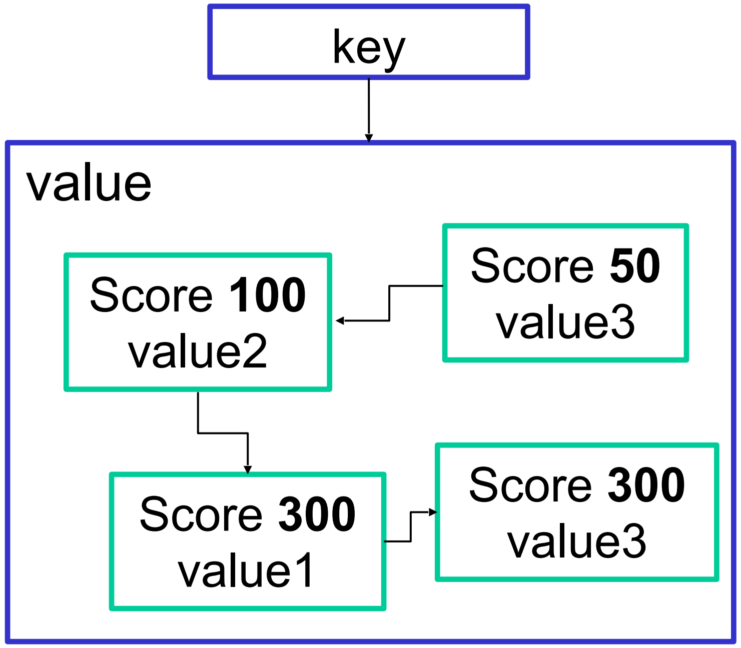
Redis

- REmote DIrectory Server
  - An (in-memory) key-value store.

- Redis was the most popular implementation of a key-value database as of March 2022, according to DB-Engines Ranking (link).

Data Model

- Key: Printable ASCII
- Value:
  - Primitives: Strings
  - Containers (of strings):
    - Hashes
    - Lists
    - Sets
    - **Sorted Sets**



https://redis.io/topics/data-types

Hands-on Redis
(Docker image)

Redis with Dockers

- We use a lightweight container with redis preconfigured

```
$ docker pull sickp/alpine-redis
```

- create a small network named redis_network with one redis server and one client

```
$ docker network create redis_network


$ docker run --rm --network=redis_network --name=redis-server
sickp/alpine-redis


$ docker run --rm --net=redis_network -it sickp/alpine-redis redis-cli
-h redis-server
```

Redis with Dockers

- Use the command line interface on the client to connect to the redis server

```
$ redis-cli -h redis-server [-p (port-number)]
```

Atomic Operations: Strings

Main operations, implemented in an atomic manner:

```
redis> GET key
redis> SET key value [EX expiration-period-secs]
redis> APPEND key value
redis> EXISTS key
redis> DEL key
redis> KEYS pattern          # use SCAN in production
```

```
# set if key does not exist
redis> SETNX key value
# Get old value and set a new one
redis> GETSET key value
# Set a timeout after which the key will be deleted
redis> EXPIRE key seconds
```

Details on Redis commands: https://redis.io/commands/

Atomic Operations: Hashes

Main operations, implemented in an atomic manner:

```
redis> HGET key field
redis> HSET key field value
redis> HEXISTS key field
redis> HDEL key field
```

```
# Get all field names of the hash stored at key
redis> HKEYS key
# Get all values of the hash stored at key
redis> HVALS key
```

Details on Redis commands: https://redis.io/commands/

Case Study (1)

- **Problem:** We need to implement a recommendation system for a radio station that suggests the next song according to the history of played songs for the same genre. To this end, we need to trace the number of reproductions (a counter) for each genre played by the user.

Case Study (2)

- **Problem:** We need to implement a recommendation system for a radio station that suggests the next song according to the history of played songs for the same genre. To this end, we need to trace the number of reproductions (a counter) for each genre played by the user.

- **Solution:** To store the counter per genre, we can resort to a hashmap. To store such a data structure for each userX, we simply save it under the key "userXcounter".

Case Study (3)

```
redis> HSET user1counter rock 1
redis> HGET user1counter rock
redis> HEXISTS user1counter classic
redis> HGET user1counter classic
redis> HSET user1counter rock 4
redis> HGET usr1counter rock
redis> HSET user1counter jazz 2
redis> HSET user1counter pop 1
redis> HEXISTS user1counter classic
redis> HDEL user1counter classic
redis> HEXISTS user1counter classic
```

Case Study (4)

```
redis> HKEYS user1counter
                    1) “rock”
                    2) “jazz”
                    3) “pop”
redis> HVALS user1counter
                    1) “4”
                    2) “2”
                    3) “1”
```

Atomic Operations: Sets

Main operations, implemented in an atomic manner:

```
# Add a value to the set stored at key
redis> SADD key value
# Remove the value from the set stored at key
redis> SREM key value
# Get the cardinality of the set stored at key
redis> SCARD key
# Remove and return a random member of the set
redis> SPOP key
```

```
# Union, Difference, Intersection between sets
redis> SUNION keyA keyB
redis> SDIFF keyA keyB
redis> SINTER keyA keyB
```

Details on Redis commands: https://redis.io/commands/

Case Study (5)

- Problem: We also need to store which bands/singers play a specific genre.
  - We assume that a band can play several genres.
  - We might be interested in selecting bands belonging to multiple genres, or in identifying a selection of bands that play the same kind of music.

- Solution: we need to keep trace of a set of singers for each musical genre.

Case Study (6)

- Problem: We also need to store which bands/singers play a specific genre.
  - We assume that a band can play several genres.
  - We might be interested in selecting bands belonging to multiple genres, or in identifying a selection of bands that play the same kind of music.

- Solution: We can resort to "sets" and save each bands/singers under a key representing the specific musical genre.

Case Study (7)

```
redis> SADD rock "pink floyd"
redis> SADD rock "queen"
redis> SADD rock "nirvana"
redis> SADD rock "baustelle"
redis> SADD jazz "paolo conte"
redis> SADD pop "paolo conte"
redis> SADD pop "baustelle"
redis> SCARD rock                    # 4
redis> SCARD Rock                     # 0
redis> SADD pop "mozart"
redis> SREM pop "mozart"
```

Case Study (8)

```
redis> SDIFF rock pop
                      1) “pink floyd”
                      2) “queen”
                      3) “nirvana”
redis> SUNION rock jazz
                      1) “pink floyd”
                      2) “queen”
                      3) “nirvana”
                      4) “baustelle”
                      5) “paolo conte”
```

Case Study (9)

- **Problem:** The recommendation system might learn from the user behavior upon the suggested songs. Therefore, we need to identify the number of reproduction of the suggested genre, so that, in the future, we can suggest the top-K genres that have been suggested and listened by the user.

Case Study (10)

- **Problem:** The recommendation system might learn from the user behavior upon the suggested songs. Therefore, we need to identify the number of reproduction of the suggested genre, so that, in the future, we can suggest the top-K genres that have been suggested and listened by the user.

- **Solution:** We can use the "sorted sets" to store the number of reproduction of songs per genre, so that the data structure can automatically determines the top-K elements.

Atomic Operations: Sorted Sets

Sorted Sets: non repeating collections of strings.

A score is associated to each value. Values of a set are ordered, from the smallest to the greatest score. Scores may be repeated.

Main operations, implemented in an atomic manner:

```
# Add a value to the set stored at key
redis> ZADD key score value
# Remove the value from the set stored at key
redis> ZREM key value
# Get the cardinality of the set stored at key
redis> ZCARD key
# Return the score of a value in the set stored at key
redis> ZSCORE key value
```

Details on Redis commands: https://redis.io/commands/

Atomic Operations: Sorted Sets

```
redis> ZCARD urepr
redis> ZADD urepr 1 rock
redis> ZADD urepr 1 jazz
redis> ZADD urepr 1 pop
redis> ZCARD urepr                    # 3
redis> ZREM urepr pop
redis> ZCARD urepr                           # 2
redis> ZSCORE urepr jazz               # 1
```

Atomic Operations: Sorted Sets

The presence of a score enables to rank or to retrieve the elements as well as changing their order during the lifetime of the sorted set

```
# Returns the rank of value in the sorted set.
# The rank is 0-based.
redis> ZRANK key value

# Returns the values in a range of the ranking (start and stop are 0-based
indexes; -k stands for the k element from the end of the rank)
redis> ZRANGE key start stop [WITHSCORES]
# Like ZRANGE but uses the score instead of the index
redis> ZRANGEBYSCORE key min max

# Increments by increment the score of value
redis> ZINCRBY key increment value
```

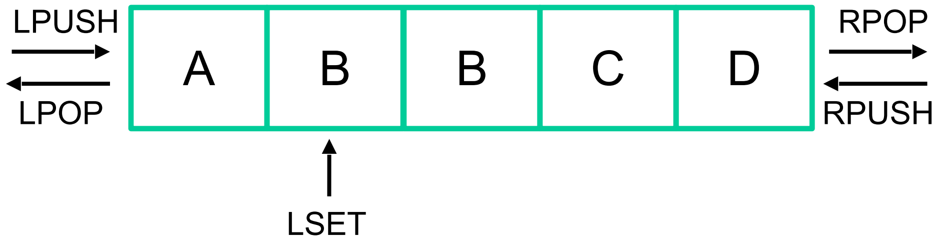
Details on Redis commands: https://redis.io/commands/

Case Study (11)

```
redis> ZRANK urepr pop
redis> ZRANK urepr rock                         # 1
redis> ZINCRBY urepr 3 rock                     # score:4
redis> ZINCRBY urepr 1 pop                      # score:1
redis> ZCARD urepr                    # 3
redis> ZRANK urepr pop                          # 1
redis> ZRANK urepr rock                         # 2
redis> ZRANGE urepr 0 1
                        1) "jazz"
                        2) "pop"
redis> ZRANGE urepr 0 -1
                        1) "jazz"
                        2) "pop"
                        3) "rock"
```

Atomic Operations: Lists

Lists are ordinary linked lists; they enable to push and pop values at both sides or in an exact position



Main operations, implemented in an atomic manner:

```
# Push value at the head/tail of the list in key
redis> LPUSH/RPUSH key value [value]
# Remove and return the head/tail of the list in key
redis> LPOP/RPOP key
# Get the length of the list
redis> LLEN key
# Returns the specified elements of the list (0-based) index
redis> LRANGE key start stop
```

Case Study (13)

- **Problem:** The music player needs to store the playlist for the user.
  - The playlist can be populated by the user by adding tracks while navigating the music store, or it can be populated by the recommendation system.
  - While the music is playing, tracks are popped out from the playlist.

Case Study (14)

- **Problem:** The music player needs to store the playlist for the user.
  - The playlist can be populated by the user by adding tracks while navigating the music store, or it can be populated by the recommendation system.
  - While the music is playing, tracks are popped out from the playlist.

- **Solution:** We can store in-memory the playlist by using a "list" data structure.

Case Study (15)

```
redis> RPUSH uplay "time"
redis> RPUSH uplay "money"
redis> LPUSH uplay "glory days"
redis> LLEN uplay                        # 3
redis> LRANGE uplay 0 -1
                              1) “glory days”
                              2) “time”
                              3) “money”
redis> LRANGE uplay -2 -1
                              1) “time”
                              2) “money”
```

Atomic Operations: Lists

```
# Removes the first count occurrences of elements equal to
value from the list stored at key

redis> LREM key count value
```

count > 0	remove elements equal to value moving from head to tail
count < 0	remove elements equal to value moving from tail to head
count = 0	remove all elements equal to value.

```
# Sets the list element at (0-based) index to value.

redis> LSET key index value
```

Details on Redis commands: https://redis.io/commands/