Key-value database-Redis

Agenda

- What are key-value databases
- Introduction to Redis
- Download and install Redis
- Redis commands (strings, lists, keys)

redis

REmote **DI**ctionary **S**erver

What is Redis

- Redis is an open-source, in-memory data structure store, used as a database, cache, and message broker. It's known for its speed and flexibility, making it popular in various real-life applications.

Example

- Imagine you have a social media platform where users can follow each other. To efficiently handle this scenario, you can use Redis to store the followers of each user. Here's how it works:
- **Storing Followers**: Each user has a unique ID, and you can use Redis Hashes to store followers. For example, you can have a hash called "followers:user_id" where the keys are the follower IDs and the values are timestamps indicating when they followed. This allows for quick access to a user's followers.
- **Retrieving Followers**: When a user wants to see their followers, Redis can quickly retrieve the data from the hash. Since Redis is in-memory, the retrieval is fast, even with large numbers of followers.
- **Updating Followers**: When a user follows or unfollows another user, Redis allows for quick updates to the data. You can simply add or remove entries in the hash, ensuring that the user's follower list is always up to date.
- **Cache Invalidation**: Redis can also be used as a cache to store frequently accessed data, like user profiles or posts. This reduces the load on your primary database and speeds up response times for users.
- In this example, Redis acts as a high-performance database and cache, efficiently handling user-related data and improving the overall performance of the social media platform.

Message Broker:

- Redis can be used as a message broker by leveraging its Pub/Sub (Publish/Subscribe) feature. This allows different parts of an application to communicate asynchronously. Here's an example:
- Imagine you have a chat application where users can send messages to each other. You can use Redis Pub/Sub to handle real-time message delivery. When a user sends a message, it's published to a channel corresponding to the recipient's user ID. The recipient, who is subscribed to that channel, receives the message instantly.
- Example:
- User A sends a message to User B.
- The message is published to a Redis channel named "user_b_messages".
- User B, who is subscribed to the "user_b_messages" channel, receives the message instantly.

- **Publishing Messages**: When a user sends a message in a chat application, you can use the PUBLISH command to publish the message to a specific channel. Each channel represents a conversation or a recipient.
- Example:
- PUBLISH user_b_messages "Hello, User B! This is a message from User A."

- **Subscribing to Channels**: Users who want to receive messages from a specific conversation or recipient can subscribe to the corresponding channel using the SUBSCRIBE command. This command establishes a persistent connection to the channel, ensuring that the user receives messages in real-time.
- Example:
- SUBSCRIBE user_b_messages

- **Listening for Messages**: Once subscribed, users continuously listen for new messages on the subscribed channels. Redis automatically delivers incoming messages to subscribers via the established connections.
- Example (pseudo code for listening to messages):
- while (true):
- message = RECEIVE_MESSAGE_FROM_CHANNEL(user_b_messages)
- DISPLAY_MESSAGE_TO_USER(message)

- **Unsubscribing from Channels**: When a user wants to stop receiving messages from a particular conversation or recipient, they can unsubscribe from the corresponding channel using the UNSUBSCRIBE command.
- Example:
- UNSUBSCRIBE user_b_messages
- These Redis commands enable the implementation of a Pub/Sub (Publish/Subscribe) mechanism, allowing real-time communication between different parts of an application. Redis efficiently handles message delivery, making it suitable for use as a message broker in various scenarios such as chat applications, notifications systems, and real-time updates.
-

Cache:

- Redis is often used as a cache due to its speed and versatility. It can store frequently accessed data in memory, reducing the need to fetch it from a slower disk-based database. Here's an example:
- Suppose you have a website where users can search for products. You can use Redis to cache the search results for faster retrieval. When a user searches for a product, you first check if the results are already cached in Redis. If they are, you return the cached results; otherwise, you fetch the results from the primary database, cache them in Redis, and then return them to the user.
- Example:
- User searches for "red shoes".
- The application checks Redis for cached search results.
- If cached results exist, they are returned to the user.
- If not, the application fetches the results from the primary database, caches them in Redis, and then returns them to the user.

- **Storing Cached Data**: When a user searches for a product and the results are not found in the cache, you would typically fetch the data from the primary database and then store it in Redis for future requests. You can use the SET command to store the search results as a string in Redis, along with an expiration time to ensure the cache is refreshed periodically.
- Example:
- SET product_search:red_shoes "Cached search results for red shoes"
- EXPIRE product_search:red_shoes 3600 # Expires in 1 hour

- **Retrieving Cached Data**: When a user searches for the same product again, you first check if the data exists in the cache using the GET command. If it exists, you return the cached results; otherwise, you fetch the data from the primary database.
- Example:
- GET product_search:red_shoes

Database:

- Redis can also be used as a primary database, especially for scenarios where data needs to be stored and accessed with low latency. While Redis is primarily an in-memory data store, it can persist data to disk for durability. Here's an example:
- Consider a leaderboard for an online game where you need to store and update players' scores. You can use Redis to store the leaderboard data. Each player's score is stored as a sorted set in Redis, allowing you to quickly retrieve the top players or update a player's score.
- Example:
- Player A scores 100 points.
- Player B scores 150 points.
- Player C scores 80 points.
- The leaderboard is stored as a sorted set in Redis.
- You can retrieve the top players or update a player's score efficiently using Redis commands.

- **Storing Player Scores**: You can use the ZADD command to add or update a player's score in the sorted set representing the leaderboard. Each player's score acts as the score value, and their unique identifier (e.g., username or player ID) serves as the member.
- Example:
- ZADD leaderboard 100 PlayerA
- ZADD leaderboard 150 PlayerB
- ZADD leaderboard 80 PlayerC

- **Retrieving Top Players**: To retrieve the top players from the leaderboard, you can use the ZREVRANGE command, specifying the start and stop indexes to fetch a range of elements from the sorted set in descending order. You can also use the WITHSCORES option to retrieve both the members and their scores.
- Example:
- ZREVRANGE leaderboard 0 2 WITHSCORES
- This command retrieves the top 3 players (indexes 0 to 2) with their scores.

- **Updating Player Scores**: If a player's score changes, you can update it in the leaderboard using the ZADD command. Redis automatically updates the score if the member already exists in the sorted set.
- Example (updating PlayerA's score to 120):
- ZADD leaderboard 120 PlayerA
- These are some basic Redis commands for managing the leaderboard data. With Redis, you can efficiently store, update, and retrieve data like player scores, making it suitable for real-time applications such as online games.

What are key value databases/store

- Databases store the data as collection of key value pairs
- Examples

Key	Value
John	997734xxxx
Peter	983388xxxx

- Key(simple string) should be unique; value can be anything(arbitrary data field)

Key	Value
Name	Joe Bloggs
Age	42
Occupation	Stunt Double
Height	175cm
Weight	77kg

Key	Value
John	john@example.com
Doe	doe@example.com
Worf	worf@example.com

What are key value databases/store

- This type of NoSQL database implements a hash table to store unique keys along with the pointers to the corresponding data values.
- A value can be stored as an integer, a string, JSON, or an array—with a key used to reference that value.
- It typically offers excellent performance and can be optimized to fit an organization’s needs.
- Key-value stores have no query language but they do provide a way to add and remove key-value pairs.
- Values cannot be queried or searched upon. Only the key can be queried.

When to use a key-value database

- When your application needs to handle lots of small continuous reads and writes, that may be volatile. Key-value databases offer fast in-memory access.
- When storing basic information, such as customer details; storing webpages with the URL as the key and the webpage as the value; storing shopping-cart contents, product categories, e-commerce product details
- For applications that don’t require frequent updates or need to support complex queries.

Examples of Popular Key-Value Databases

- Redis
- Amazon DynamoDB
- Memcached
- Riak
- Aerospike
- Oracle NoSQL
- InfinityDB

What is Redis

- Redis is a flexible, open-source , in-memory data structure store, used as database, cache, and message broker

  (refer https://tsh.io/blog/message-broker/.)
- Redis is a NoSQL database so it facilitates users to store huge amount of data without the limit of a Relational database.
- Redis supports various types of data structures like strings, hashes, lists, sets, sorted sets, bitmaps, hyperlogs and geospatial indexes with radius queries.
- Redis has built-in replication, transaction, different levels of non disk persistence(https://redis.io/topics/persistence).
- Sharding is supported in redis.

What is In-memory database?

- Database management systems which stores database on computers main memory rather than a hard disk
- This gives a very quicker response time
- The databases are used in applications that demand a quicker response time like online interactive gaming

Popular use cases of Redis

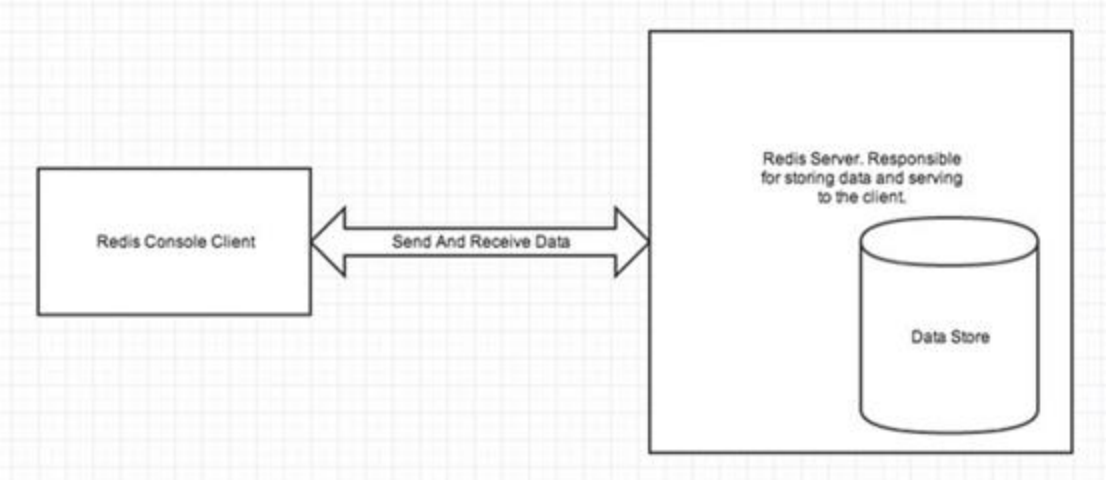
- Caching
- Chat, messaging, and queues
- Gaming leaderboards
- Session store
- Rich media streaming
- Geospatial
- Machine Learning
- Real-time analytics

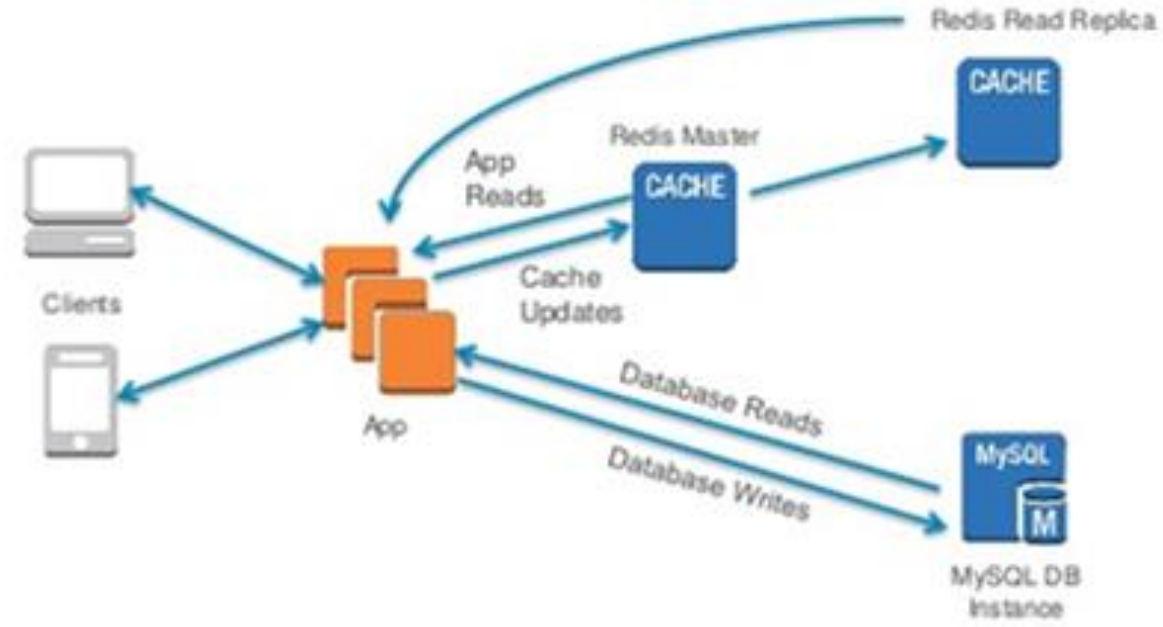
Who uses Redis?

List of well-known companies using Redis

- Twitter.
- GitHub.
- Weibo.
- Pinterest.
- Snapchat.
- Craigslist.
- StackOverflow.
- Flickr

Redis Architecture





Features of Redis

Speed:

- Redis stores the whole dataset in primary memory that's why it is extremely fast.
- It loads up to 110,000 SETs/second and 81,000 GETs/second can be retrieved in an entry level Linux box.
- Redis supports Pipelining of commands and facilitates you to use multiple values in a single command to speed up communication with the client libraries.

Pipelining:Pipelining facilitates a client to send multiple requests to the server without waiting for the replies at all, and finally reads the replies in a single step.

- Redis is a TCP server which supports request/response protocol. In Redis, a request is completed in two steps:
- The client sends a query to the server usually in a blocking way for the server response.
- The server processes the command and sends the response back to the client.

Persistence:

- While all the data lives in memory, changes are asynchronously saved on disk using flexible policies based on elapsed time and/or number of updates since last save.

Data Structures:

Redis supports various types of data structures such as strings, hashes, sets, lists, sorted sets with range queries, bitmaps, hyperloglogs and geospatial indexes with radius queries.

Features of Redis

Atomic Operations:

- Redis operations working on the different Data Types are atomic, so it is safe to set or increase a key, add and remove elements from a set, increase a counter etc.

Supported Languages:

- Redis supports a lot of languages such as ActionScript, C, C++, C#, Clojure, Common Lisp, D, Dart, Erlang, Go, Haskell, Haxe, Io, Java, JavaScript (Node.js), Julia, Lua, Objective-C, Perl, PHP, Pure Data, Python, R, Racket, Ruby, Rust, Scala, Smalltalk and Tcl.

Features of Redis

Master/Slave Replication:

- Redis follows a very simple and fast Master/Slave replication. It takes only one line in the configuration file to set it up

Sharding:

- Redis supports sharding. It is very easy to distribute the dataset across multiple Redis instances, like other key-value store.

Portable:

- Redis is written in ANSI C and works in most POSIX systems like Linux, BSD, Mac OS X, Solaris, and so on. Redis is reported to compile and work under WIN32 if compiled with Cygwin, but there is no official support for Windows currently.

LRU eviction in redis with example

- LRU (Least Recently Used) eviction is a cache eviction policy used in Redis to automatically remove least recently used items from a cache when the maximum memory limit is reached.

Here's how LRU eviction works in Redis :

1. **Configuring Redis for LRU Eviction**:
2. **Tracking Usage with a Time Stamp**:Redis tracks the time of the last access (read or write) for each key in the cache.
3. **Selecting Keys for Eviction**:

When Redis needs to evict keys to free up memory, it selects keys based on their access time.

The keys that have not been accessed for the longest time (least recently used) are selected for eviction

Example

- Let's consider a Redis cache with a maximum memory limit of 100 MB.

- Key Value
- ----------
- key1 value1
- key2 value2
- key3 value3
- key4 value4

Eviction Process:

Suppose key3 and key4 have not been accessed for the longest time according to their time stamps.
Redis selects key3 and key4 for eviction as they are the least recently used keys.
It removes key3 and key4 from the cache, freeing up memory for new items.

Continued Operation:
Redis continues to serve read and write requests as usual, with the remaining key-value pairs in the cache.
Accessing any key updates its access time, ensuring that recently used keys are not evicted.

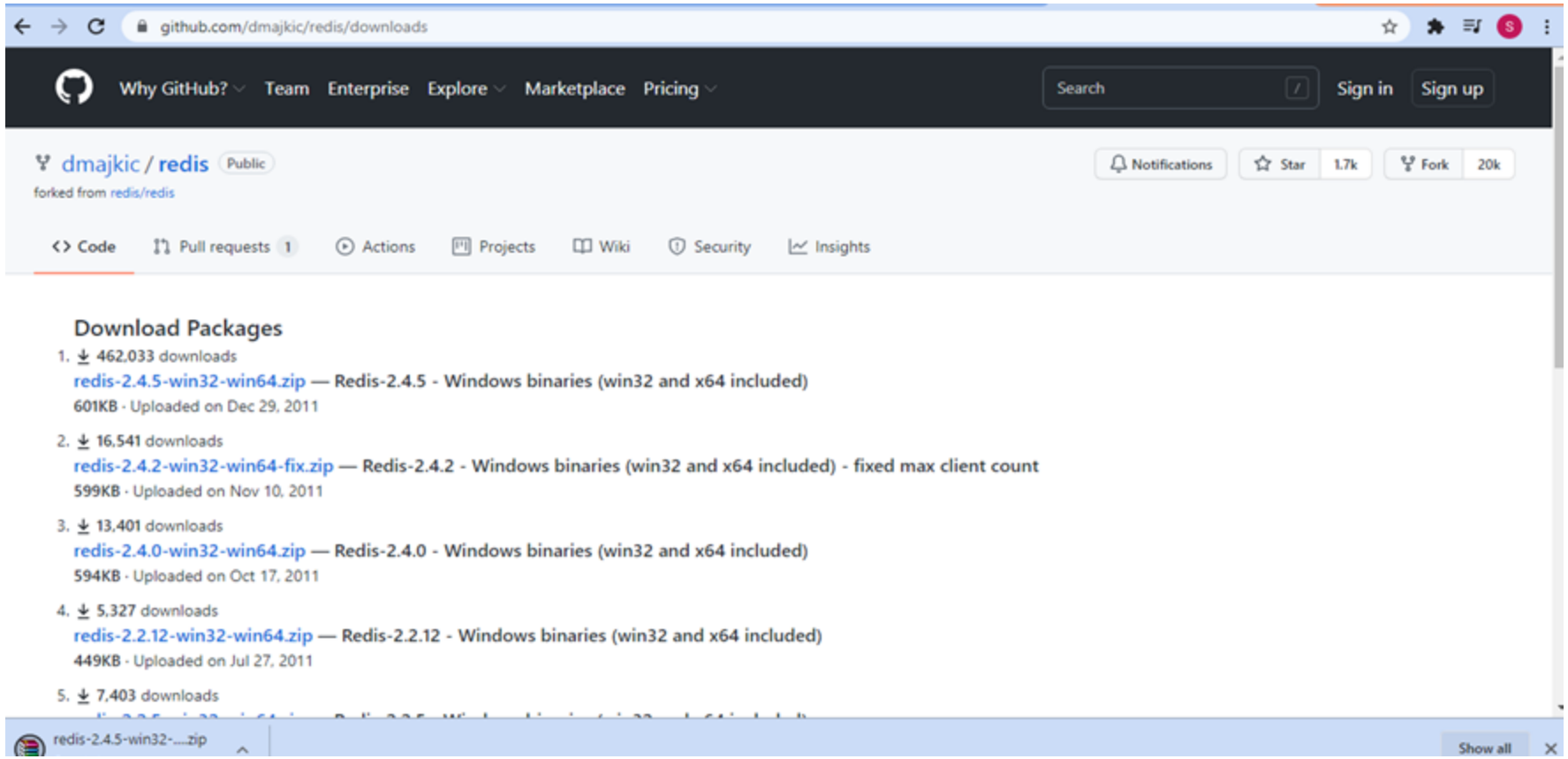
Configuring LRU Parameters:
In Redis, you can configure various parameters related to LRU eviction, such as the maximum memory limit, eviction policy, and behavior when the memory limit is exceeded.

Redis vs RDBMS

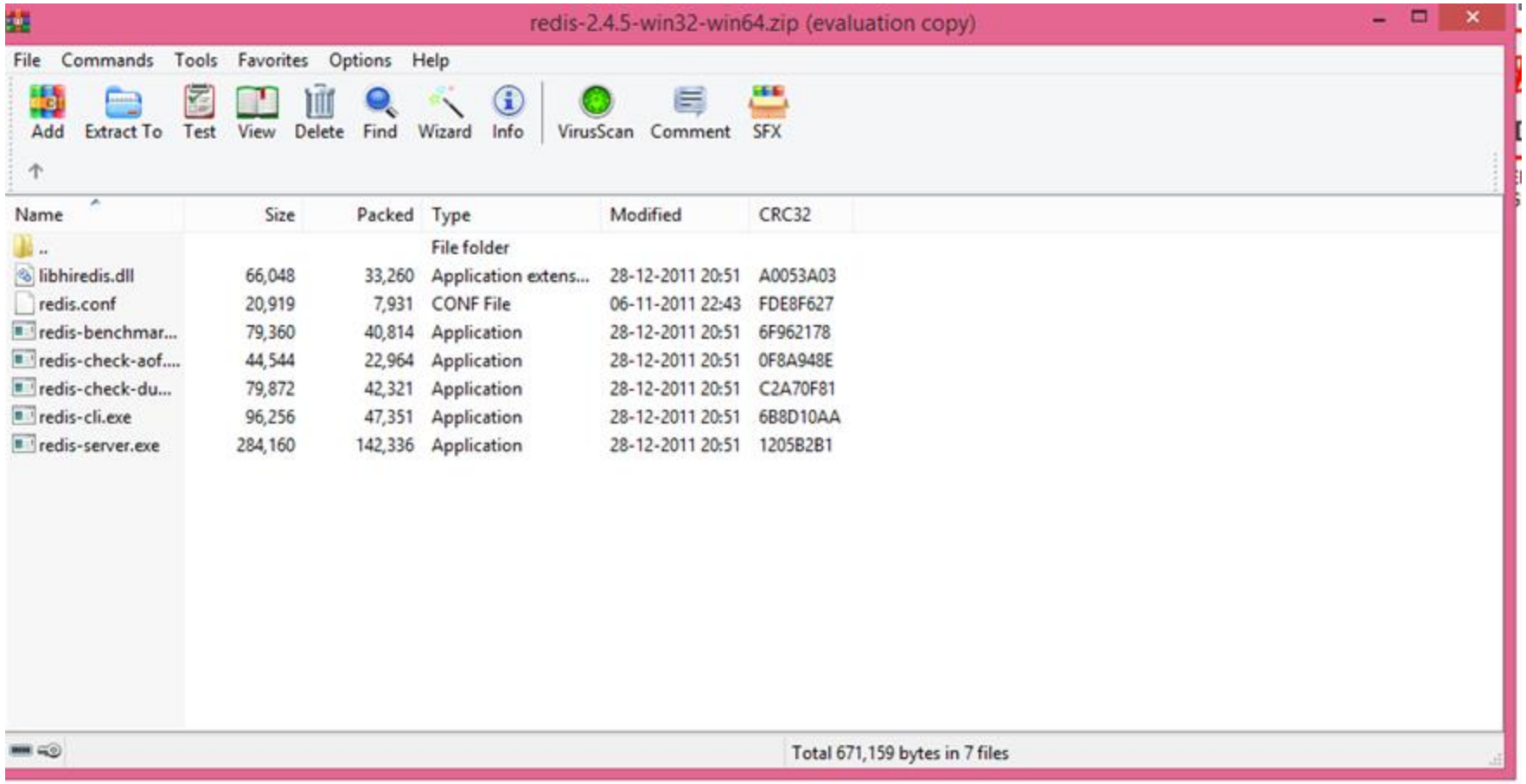
Redis	RDBMS
Redis stores everything in primary memory.	RDBMS stores everything in secondary memory.
In Redis, Read and Write operations are extremely fast because of storing data in primary memory.	In RDBMS, Read and Write operations are slow because of storing data in secondary memory.
Primary memory is in lesser in size and much expensive than secondary so, Redis cannot store large files or binary data.	Secondary memory is in abundant in size and cheap than primary memory so, RDBMS can easily deal with these type of files.
Redis is used only to store those small textual information which needs to be accessed, modified and inserted at a very fast rate. If you try to write bulk data more than the available memory then you will receive errors.	RDBMS can hold large data which has less frequently usage and not required to be very fast.

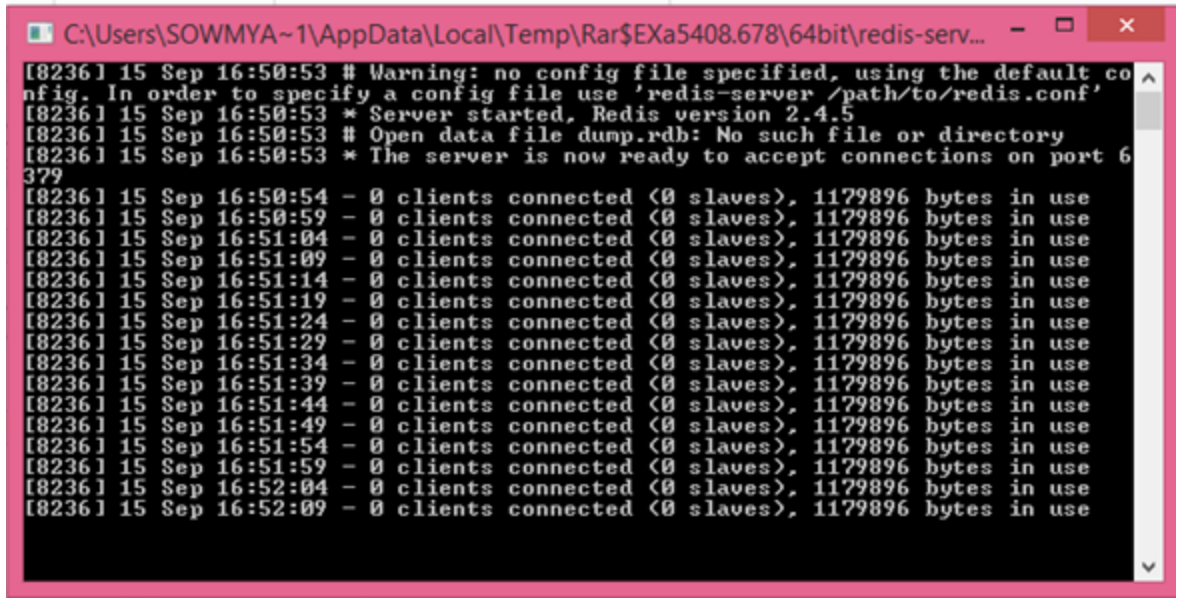
Installation

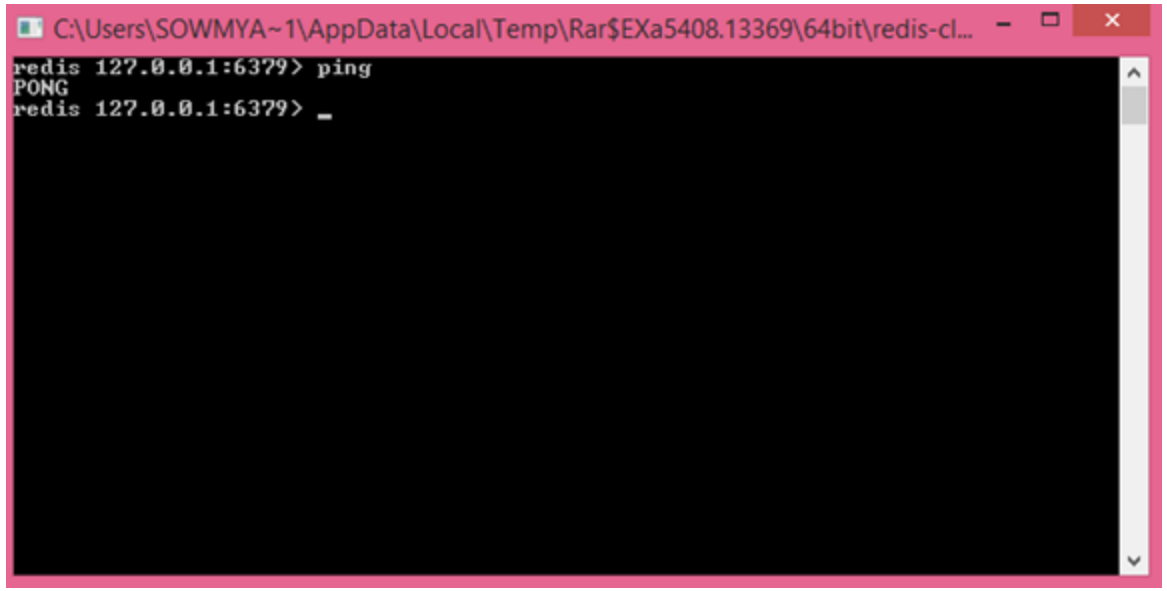
- https://github.com/dmajkic/redis/downloads



Run server and client exe




C:\Users\SOWMYA~1\AppData\Local\Temp\Rar$EXa5408.678\64bit\redis-serv...
[8236] 15 Sep 16:50:53 # Warning: no config file specified, using the default co
nfig. In order to specify a config file use 'redis-server /path/to/redis.conf'
[8236] 15 Sep 16:50:53 * Server started, Redis version 2.4.5
[8236] 15 Sep 16:50:53 # Open data file dump.rdb: No such file or directory
[8236] 15 Sep 16:50:53 * The server is now ready to accept connections on port 6
379
[8236] 15 Sep 16:50:54 - 0 clients connected (0 slaves), 1179896 bytes in use
[8236] 15 Sep 16:50:59 - 0 clients connected (0 slaves), 1179896 bytes in use
[8236] 15 Sep 16:51:04 - 0 clients connected (0 slaves), 1179896 bytes in use
[8236] 15 Sep 16:51:09 - 0 clients connected (0 slaves), 1179896 bytes in use
[8236] 15 Sep 16:51:14 - 0 clients connected (0 slaves), 1179896 bytes in use
[8236] 15 Sep 16:51:19 - 0 clients connected (0 slaves), 1179896 bytes in use
[8236] 15 Sep 16:51:24 - 0 clients connected (0 slaves), 1179896 bytes in use
[8236] 15 Sep 16:51:29 - 0 clients connected (0 slaves), 1179896 bytes in use
[8236] 15 Sep 16:51:34 - 0 clients connected (0 slaves), 1179896 bytes in use
[8236] 15 Sep 16:51:39 - 0 clients connected (0 slaves), 1179896 bytes in use
[8236] 15 Sep 16:51:44 - 0 clients connected (0 slaves), 1179896 bytes in use
[8236] 15 Sep 16:51:49 - 0 clients connected (0 slaves), 1179896 bytes in use
[8236] 15 Sep 16:51:54 - 0 clients connected (0 slaves), 1179896 bytes in use
[8236] 15 Sep 16:51:59 - 0 clients connected (0 slaves), 1179896 bytes in use
[8236] 15 Sep 16:52:04 - 0 clients connected (0 slaves), 1179896 bytes in use
[8236] 15 Sep 16:52:09 - 0 clients connected (0 slaves), 1179896 bytes in use


C:\Users\SOWMYA~1\AppData\Local\Temp\Rar$EXa5408.13369\64bit\redis-cl...
redis 127.0.0.1:6379> ping
PONG
redis 127.0.0.1:6379>

Redis commands

- Strings
- Lists
- Sets
- Sorted Sets
- Hashes

Strings-list of commands

command	use
SET	Set the string value of a key
GET	Get the value of the key
MSET	Set multiple keys to multiple values
MGET	Get the values of all given keys
APPEND	Append a value to a key
INCR	Increment the integer value of a key by one
DECR	Decrement the integer value of a key by one

- Get Range of value:

```
redis 127.0.0.1:6379> SET mykey "This is my test key“
 OK
redis 127.0.0.1:6379> GETRANGE mykey 0 3
"This"
redis 127.0.0.1:6379> GETRANGE mykey 0 -1
"This is my test key"
```

- get the values of all specified keys

```
redis 127.0.0.1:6379> SET key1 "hello“
 OK
redis 127.0.0.1:6379> SET key2 "world"
OK
redis 127.0.0.1:6379> MGET key1 key2 someOtherKey
1)  "Hello"
2)  2) "World"
3)  3) (nil)
```

- Strlen:

```
redis 127.0.0.1:6379> SET key1 "Hello World"
OK
redis 127.0.0.1:6379> STRLEN key1
(integer) 11
redis 127.0.0.1:6379> STRLEN key2
(integer) 0
```

Storing multiple values in a single key

```
redis 127.0.0.1:6379> HMSET myhash field1 "foo" field2 "bar“
 OK
redis 127.0.0.1:6379> HGET myhash field1
"foo"
redis 127.0.0.1:6379> HGET myhash field2
"bar"
```

```
redis 127.0.0.1:6379> HVALS myhash
1) "foo“
2)  2) "bar"
```

Redis Cache-like Behaviour

Example

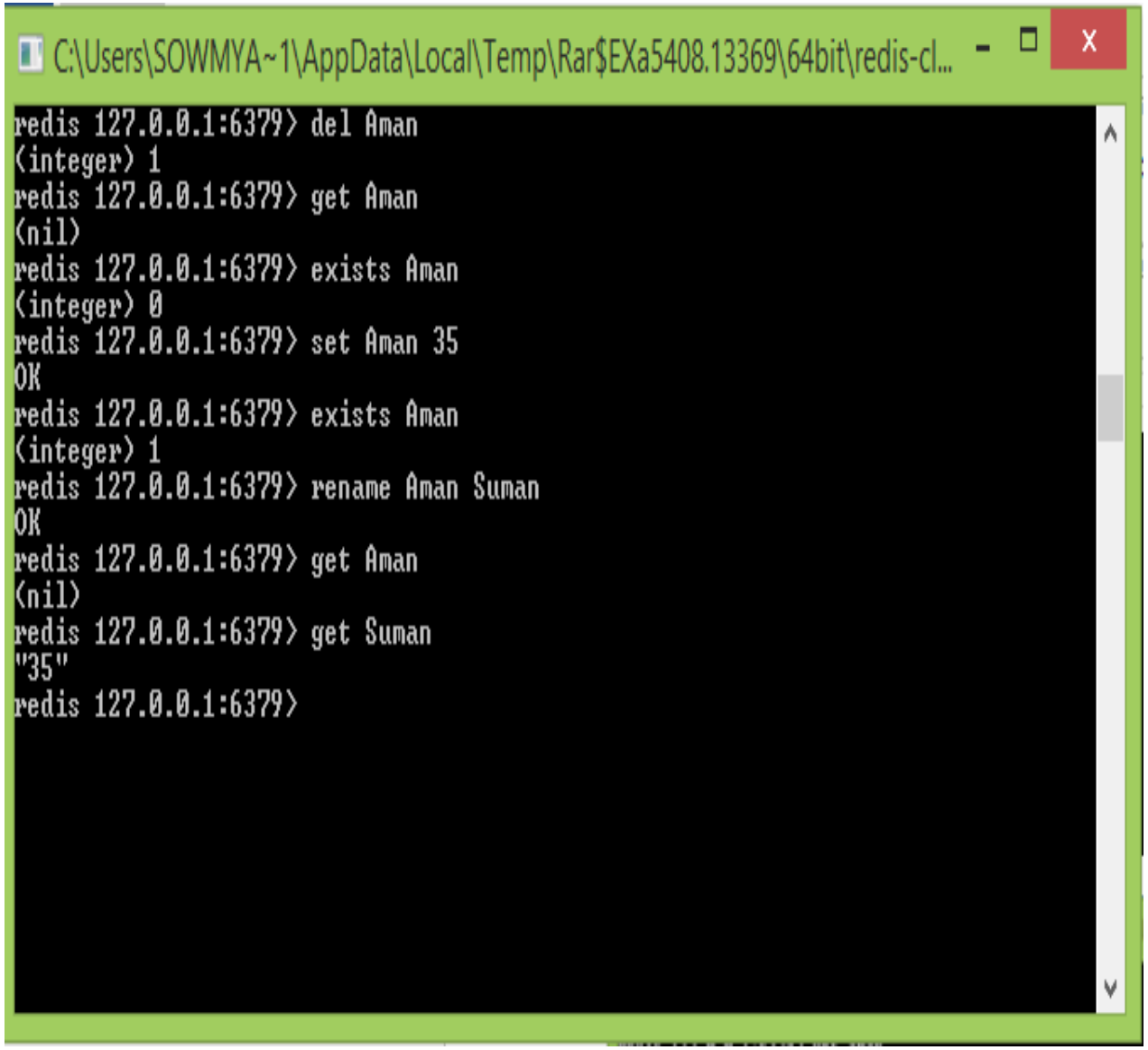
```
>SET cookie:google hello
OK
> EXPIRE cookie:google 30
(integer) 1
> TTL cookie:google          // time to live
(integer) 23
> GET cookie:google
„hello“                      // still some time to live
> TTL cookie:google
(integer) -1                 // key has expired
> GET cookie:google
(nil)                        // and was deleted
```

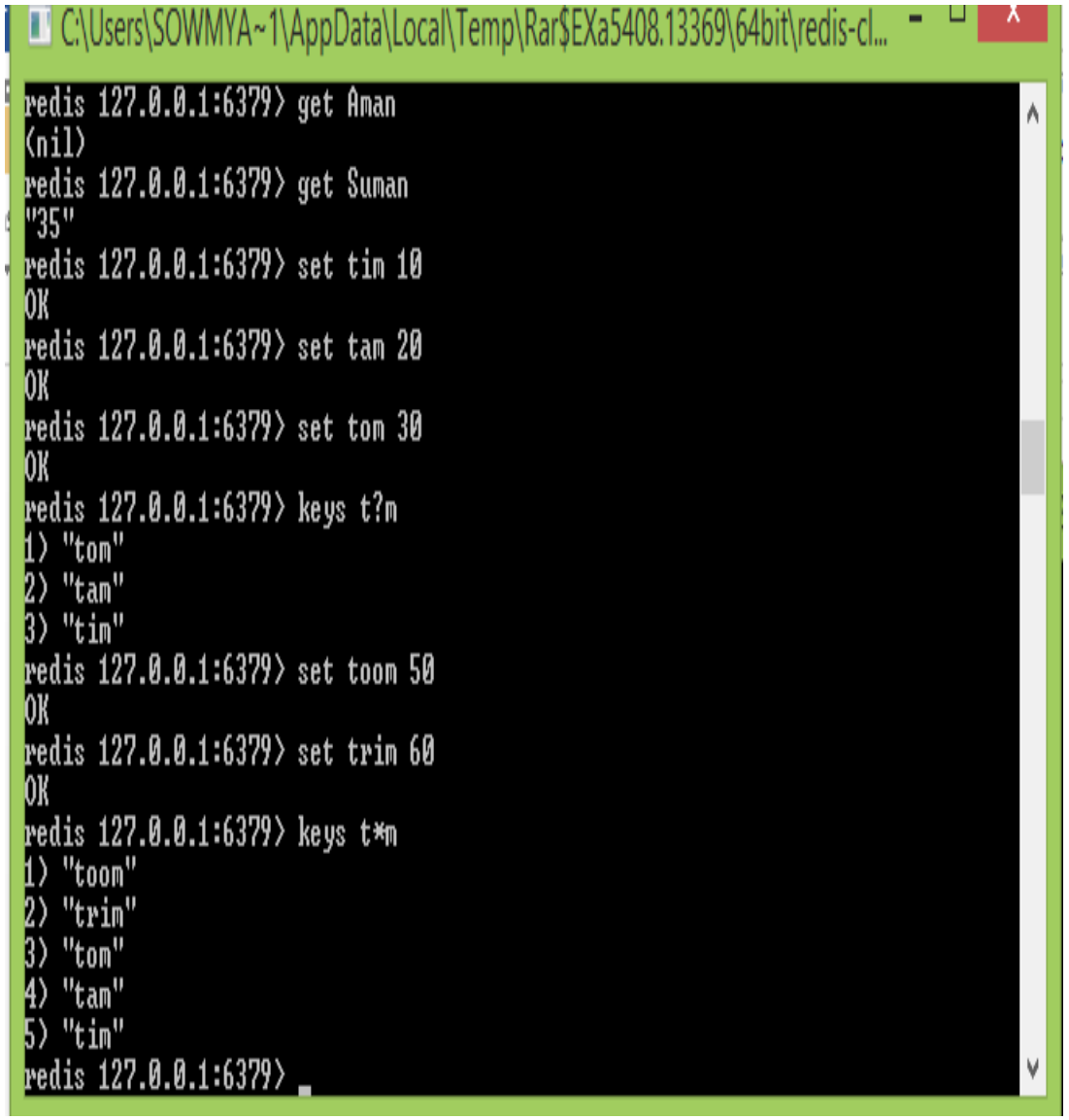
Strings-list of commands

Command	use
INCRBY	Increment the integer value of a key by the given amount
DECRBY	Decrement the integer value of a key by the given amount
SETEX	Set the value and expiration of the key
SETNX	Set the value of the key only if the key does not exists
STRLEN	Get the length of the value stored in a key

Strings-list of commands

Command	use
DEL	Delete a key
EXISTS	Check if a key exists
RENAME	Rename a key
TTL	Time to live for a key
RANDOMKEY	Return a random key from the keyspace
KEYS Pattern	Search for key using regular expressions


C:\Users\SOWMYA~1\AppData\Local\Temp\Rar$EXa5408.13369\64bit\redis-cl...
redis 127.0.0.1:6379> del Aman
(integer) 1
redis 127.0.0.1:6379> get Aman
(nil)
redis 127.0.0.1:6379> exists Aman
(integer) 0
redis 127.0.0.1:6379> set Aman 35
OK
redis 127.0.0.1:6379> exists Aman
(integer) 1
redis 127.0.0.1:6379> rename Aman Suman
OK
redis 127.0.0.1:6379> get Aman
(nil)
redis 127.0.0.1:6379> get Suman
"35"
redis 127.0.0.1:6379>


C:\Users\SOWMYA~1\AppData\Local\Temp\Rar$EXa5408.13369\64bit\redis-cl...
redis 127.0.0.1:6379> get Aman
(nil)
redis 127.0.0.1:6379> get Suman
"35"
redis 127.0.0.1:6379> set tim 10
OK
redis 127.0.0.1:6379> set tam 20
OK
redis 127.0.0.1:6379> set tom 30
OK
redis 127.0.0.1:6379> keys t?m
1) "tom"
2) "tam"
3) "tim"
redis 127.0.0.1:6379> set toom 50
OK
redis 127.0.0.1:6379> set trim 60
OK
redis 127.0.0.1:6379> keys t*m
1) "toom"
2) "trim"
3) "tom"
4) "tam"
5) "tim"
redis 127.0.0.1:6379>

Lists in Redis

- Redis lists are nothing but a list of strings that are sorted by insertion order. A list can contain more than 4 billion elements.

Command	Use
LPUSH	Insert all specified values at the head of the list
RPUSH	Insert all specified values at the tail of the list
LPOP	Removes and returns first element of the list
RPOP	Removes and returns last element of the list
LRANGE	Returns the specified elements of the list

Lists in Redis

Command	Use
LLEN	Returns the length of the list
LSET	Sets the list element at index to element
LREM	count>0: remove elements equal to element moving from head to tail Count<0: remove elements equal to element moving from tail to head Count=0: remove all elements equal to element
LINDEX	Get an element from the list by its index

Redis Data Types

List

- Operations:
  - Add element(s) to the list:
    - LPUSH (to the head)
    - RPUSH (to the tail)
    - LINSERT (inserts before or after a specified element)
    - LPUSHX (push only if the list exists, do not create if not)
  - Remove element(s): LPOP, RPOP, LREM (remove elements specified by a value)
  - LRANGE (get a range of elements), LLEN (get length), LINDEX (get an element at index)
  - BLPOP, BRPOP remove an element or block until one is available
    - Blocking version of LPOP/RPOP

Redis Data Types

List – Example

```
> LPUSH animals dog
(integer) 1      // number of elements in the list
> LPUSH animals cat
(integer) 2
>RPUSH animals horse (integer) 3
> LRANGE animals 0 -1 // -1 = the end
1) „cat“
2) „dog“
3) „horse“
> RPOP animals
„horse“
>LLEN animals (integer) 2
```

Redis Sets

What is a SET? How sets are different from lists?

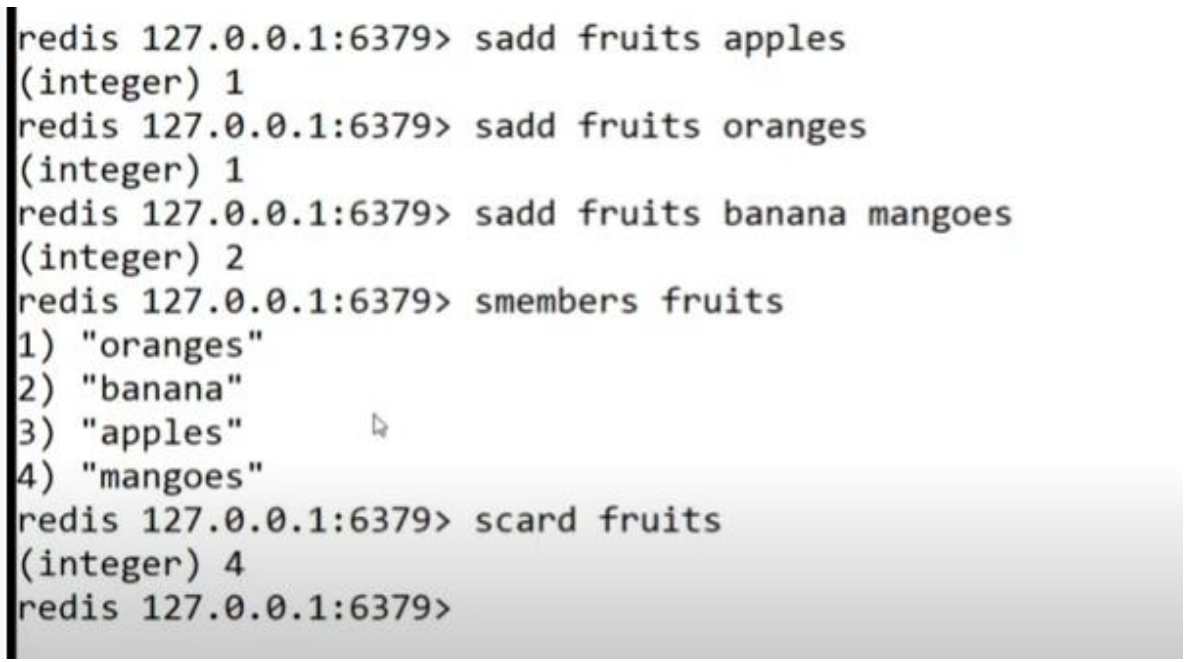
- Unordered collection of strings
- Sets do not allow duplicate strings

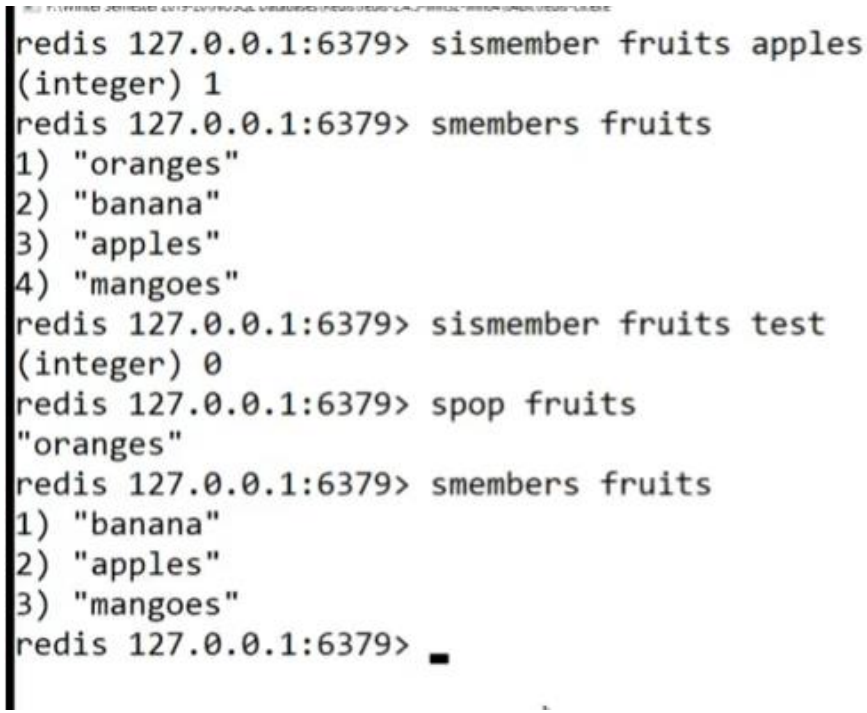
SET commands

Command	use
SADD	Add one or more string to SET
SISMEMBER	Determine if given value is member of SET
SCARD	Get the number of members in a SET
SMEMBERS	Get all members in a SET
SPOP	Remove and return one or multiple random members from a SET

SET commands

SUNION	Returns the result of union of 2 sets
SINTER	Intersects multiple sets
SDIFF	Subtracts multiple sets
SMOVE	Move a member from one SET to another
SRANDMEMBER	Get one or multiple random members from a SET


redis 127.0.0.1:6379> sadd fruits apples
(integer) 1
redis 127.0.0.1:6379> sadd fruits oranges
(integer) 1
redis 127.0.0.1:6379> sadd fruits banana mangoes
(integer) 2
redis 127.0.0.1:6379> smembers fruits
1) "oranges"
2) "banana"
3) "apples"
4) "mangoes"
redis 127.0.0.1:6379> scard fruits
(integer) 4
redis 127.0.0.1:6379>


redis 127.0.0.1:6379> sismember fruits apples
(integer) 1
redis 127.0.0.1:6379> smembers fruits
1) "oranges"
2) "banana"
3) "apples"
4) "mangoes"
redis 127.0.0.1:6379> sismember fruits test
(integer) 0
redis 127.0.0.1:6379> spop fruits
"oranges"
redis 127.0.0.1:6379> smembers fruits
1) "banana"
2) "apples"
3) "mangoes"
redis 127.0.0.1:6379>

Sunion:

```
redis 127.0.0.1:6379> sadd sub1 eng maths phy
(integer) 3
redis 127.0.0.1:6379> smembers sub1
1) "phy"
2) "maths"
3) "eng"
redis 127.0.0.1:6379>
```

```
1) "phy"
2) "maths"
3) "eng"
redis 127.0.0.1:6379> sadd sub2 eng chem geo
(integer) 3
redis 127.0.0.1:6379> smembers sub2
1) "geo"
2) "chem"
3) "eng"
redis 127.0.0.1:6379> sunion sub1 sub2
1) "geo"
2) "chem"
3) "maths"
4) "phy"
5) "eng"
redis 127.0.0.1:6379>
```

```
redis 127.0.0.1:6379> sunion sub1 sub2
1) "geo"
2) "chem"
3) "maths"
4) "phy"
5) "eng"
redis 127.0.0.1:6379> sinter sub1 sub2
1) "eng"
redis 127.0.0.1:6379> sdiff sub1 sub2
1) "maths"
2) "phy"
redis 127.0.0.1:6379> sunionstore result sub1 sub2
(integer) 5
redis 127.0.0.1:6379>
```

Redis Data Types

Set

- Unordered collection of non-repeating strings
- Possible to add, remove, and test for existence of members in *O(1)*
- Max number of members: $2^{32} - 1$
- Operations:
  - Add element: SADD, remove element: SREM
  - Classical set operations: SISMEMBER, SDIFF, SUNION, SINTER
  - The result of a set operation can be stored at a specified key (SDIFFSTORE, SINTERSTORE, ...)
  - SCARD (element count), SMEMBER (get all elements)
  - Operations with a random element: SPOP (remove and return random element), SRANDMEMBER (get a random element)
  - SMOVE (move element from one set to another)

Redis Data Types

Set – Example

```
>SADD friends:Lisa Anna
(integer) 1
> SADD friends:Dora Anna Lisa
(integer) 2
> SINTER friends:Lisa friends:Dora
1) „Anna“
> SUNION friends:Lisa friends:Dora
1) „Lisa“
2) „Anna“
> SISMEMBER friends:Lisa Dora
(integer) 0
>SREM friends:Dora Lisa
(integer) 1
```

Sorted Sets

- Sorted Sets are similar to sets in Redis but each member in the set is associated with a score .
- The Set can be sorted using score.

Game1{Sandy 3, Sam 14, James 20, Renny 4}

- Members should be unique but scores can have duplicate.

Sorted Sets commands

command	use
ZADD	Add one or more members to a sorted Set. Update its score if it already exists
ZRANK key member	Determine index of member in a sorted Set
ZRANGE key start stop [WITH SCORES]	Return a range of members in a Sorted Set, by index
ZREVRANGE key start stop [WITH SCORES]	Return a range of members in a Sorted Set, by index, with scores ordered from high to low
ZINCRBY key increment member	Increment the score of a member in a sorted Set

Sorted Sets commands

Command	Use
ZRANGEBYSCORE	Return a range of members in a sorted Set by score
ZSCORE key member	Get the score associated with the given member in the sorted set
ZCARD key	Get the number of members in a sorted set
ZREM by member	Remove one or member from a sorted set
ZREMRANGEBYRANK key start stop	Remove all members in a sorted set within the given indexes
ZREMRANGEBYSCORE key min max	Remove all members in a sorted set within the given scores
ZCOUNT by min max	Count the members in a sorted set with scores within the given values

Redis Hashes: Representing Objects

- Objects have a lot of properties
- Properties are field value pairs
- For instance every student objects have following properties

Name

Address

Gender

Phone

- Redis hashes are perfect datatypes to represent objects
- A hash can hold many field value pairs.

Hash Commands

command	Use
HSET/HMSET	Set the string value of a hash field /set multiple hash fields to multiple values
HGET/HMGET	Get the value of a hash field/get the values of all the given hash fields
HKEYS/HVALS	Get all the fields in a hash/get all the values in a hash
HGETALL	Get all the fields and values in a hash
HLEN	Get the number of field in a hash /get the length of the value of a hash field
HINCRBY	Increment the integer value of a hash field by the given number
HDEL	Delete one or more hash fields
HEXISTS	Determine if a hash field exists

Redis Transaction

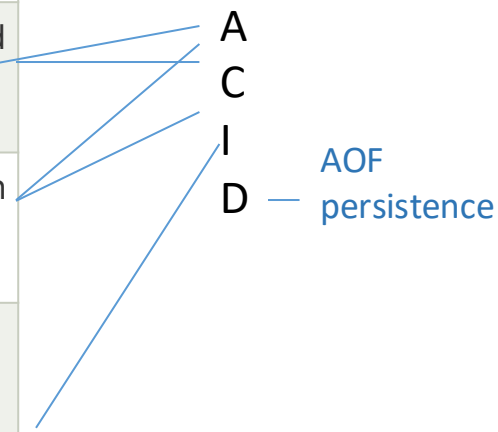
- Redis transaction is used to facilitates users to execute group of commands in a single step.

There are two properties of execution:

- All commands in a transaction are sequentially executed as a single isolated operation.
- It can never happen that a request issued by another client is served in the middle of the execution of a Redis transaction
- Redis transaction is also atomic. Atomic means either all of the commands or none are processed.

Redis Transaction Commands

Command	Description	
1	DISCARD	It is used to discard all commands issued after MULTI
2	EXEC	It is used to execute all commands issued after MULTI
3	MULTI	It is used to mark the start of a transaction block
4	UNWATCH	It is used to forget about all watched keys
5	WATCH key [key ...]	It is used to watch the given keys to determine the execution of the MULTI/EXEC block



Persistence mechanisms

- Redis offers two main types of persistence mechanisms :

RDB (Redis Database Snapshot):

- RDB persistence performs point-in-time snapshots of your dataset at specified intervals.
- It saves the dataset to disk as a binary file (dump.rdb) in a specified directory.
- RDB is useful for backups and disaster recovery but may result in some data loss if the system crashes between snapshots.
- Example: Let's say you configure Redis to save an RDB snapshot every 15 minutes. If Redis crashes, you can restart it and load the latest RDB snapshot to recover most of the data up to the point of the last snapshot.

Here's how it works

- **Snapshot Creation (Fork)**:
- When Redis triggers an RDB snapshot (either manually or based on configured intervals), it forks a child process from the main Redis process.
- The child process is an exact copy of the parent process at the point of forking.
- The child process is responsible for creating the RDB snapshot while the parent process continues serving client requests.
- **Snapshot Creation (Child Process)**:
  - The child process iterates through the dataset in memory, serializes it, and writes it to a temporary RDB file (dump.rdb.tmp).
- **Atomic File Swap**:
  - Once the snapshot creation is complete, the child process renames the temporary RDB file to the final RDB file name (dump.rdb).
  - This atomic file swap ensures that the RDB file is always in a consistent state, even if the snapshot creation process is interrupted.

- **Continue Serving Requests (Parent Process)**:
  - While the child process is creating the snapshot, the parent process continues serving client requests without interruption.

- **Completion and Cleanup**:
  - Once the snapshot creation is successful, the child process exits.
  - If there was an error during snapshot creation, the child process exits with an error status, and the main Redis process handles the situation accordingly.
  - Redis may also perform additional housekeeping tasks, such as deleting old RDB files if configured to do so.

AOF (Append-Only File):

- AOF persistence logs every write operation to the Redis dataset in an append-only file.
- It's like a transaction log, and Redis can replay these commands to rebuild the dataset.
- AOF is more durable than RDB because it logs every write operation, making it less prone to data loss in case of crashes.
- However, AOF files can become large over time, and Redis periodically rewrites them to optimize space and performance.
- Example: Redis appends every write operation to the AOF file. If Redis crashes, it replays the commands from the AOF file to reconstruct the dataset up to the point of the crash.
- Both persistence mechanisms can be used together for extra durability, although they have different trade-offs in terms of performance and disk space usage. The choice between RDB and AOF, or a combination of both, depends on factors like the importance of data durability, recovery time objectives, and resource constraints.

- **Initial AOF File**:
  - Initially, the AOF file contains a sequential log of every write operation performed on the Redis dataset.
- **AOF Rewrite Trigger**:
  - Redis monitors the size of the AOF file and triggers a rewrite when it exceeds a certain threshold (configured by the user).
- **AOF Rewrite Process**:
  - During the rewrite process, Redis reads the current dataset from memory.
  - It replays all the write operations from the original AOF file sequentially, but instead of appending them directly to a new AOF file, it constructs a new AOF file with optimized commands.
  - For example, instead of logging individual SET commands for each key, Redis might consolidate multiple SET commands into a single MSET command, reducing redundancy and optimizing space.
  - Additionally, Redis may apply further optimizations like removing redundant commands, using shorter representations for commands, and other space-saving techniques.
- **New AOF File**:
  - Once the rewrite process is complete, Redis replaces the old AOF file with the new optimized AOF file.
  - The new AOF file contains a more compact representation of the dataset, reducing disk space usage and improving performance during AOF playback.
- **Continued Logging**:
  - Redis continues to append new write operations to the new AOF file as usual.
  - When the AOF file size exceeds the threshold again, Redis triggers another rewrite process to further optimize the file.

- SET key1 value1
- SET key2 value2
- SET key3 value3
- After optimization with MSET command:
- **MSET key1 value1 key2 value2 key3 value3**

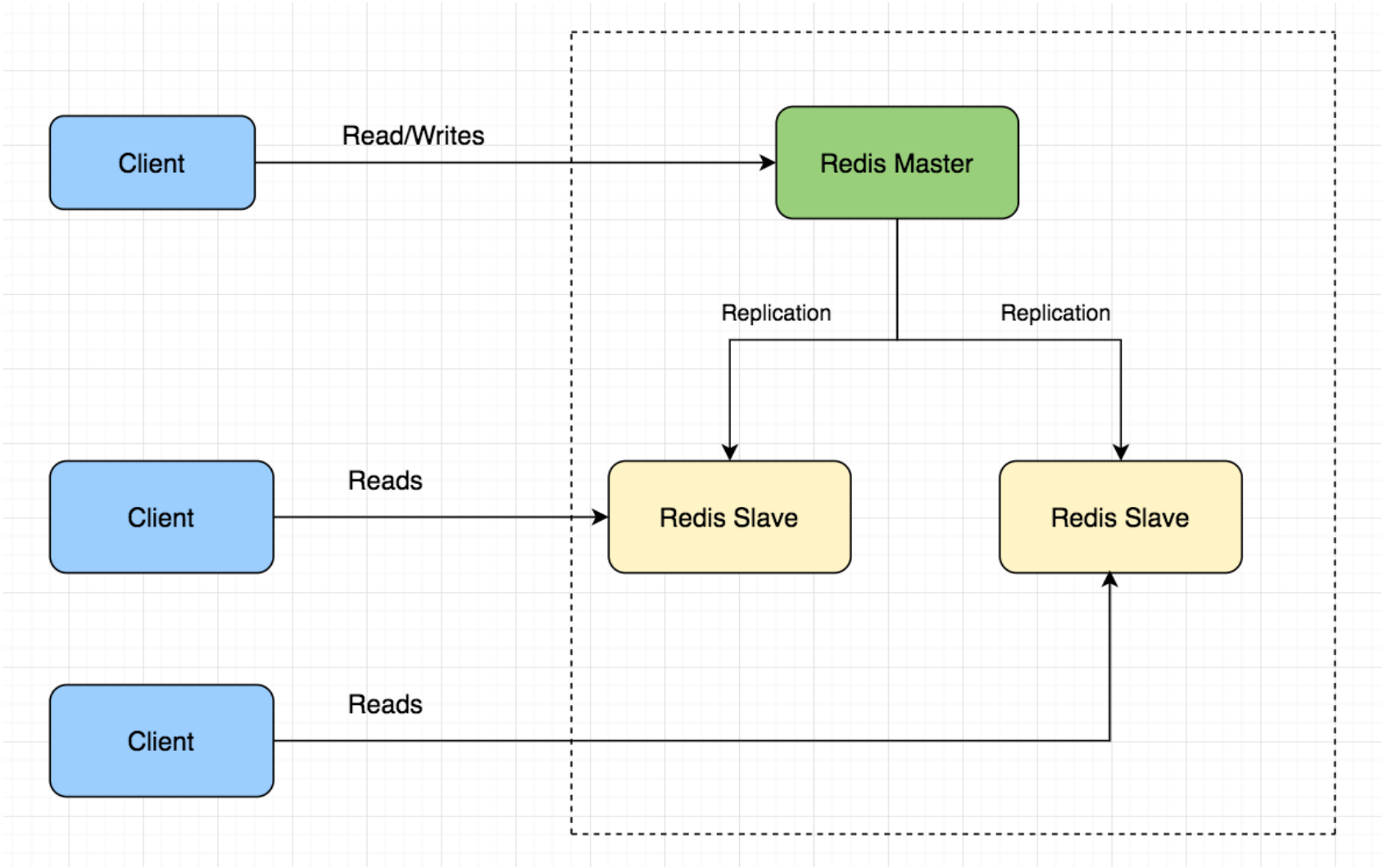
LPUSH list1 value1
LPUSH list1 value2
LPUSH list1 value3
After optimization
LPUSH list1 value1 value2 value3

HSET hash1 field1 value1
HSET hash1 field2 value2
HSET hash1 field3 value3
HMSET hash1 field1 value1 field2 value2 field3 value3

How it works:

- Redis forks a child process to perform AOF rewriting.
- The child process reads the dataset from memory and constructs a new AOF file with optimized commands.
- Meanwhile, the main Redis process continues serving client requests.
- Once AOF rewriting is complete, the child process atomically replaces the original AOF file with the new optimized AOF file.
- The main Redis process continues serving client requests with the updated AOF file, ensuring durability and optimal performance.
- This process of AOF rewriting ensures efficient disk space utilization and maintains optimal performance for write operations in Redis.

Redis: Master-Slave Architecture



What if Redis Master goes Down?

Now here we have two choices:

- Add new machine as Redis Master
- Make any existing Redis Slave as New Redis Master
- Problem with first approach where we add new machine as new Redis Master is that this will sync/replicate Zero data to all Redis Slaves means we loose all data.
- Second approach is recommended as existing slave will already have all data and once we make this Redis Slave as new Redis Master, this will sync/replicate data to all Redis Slaves which means we did not loose data.

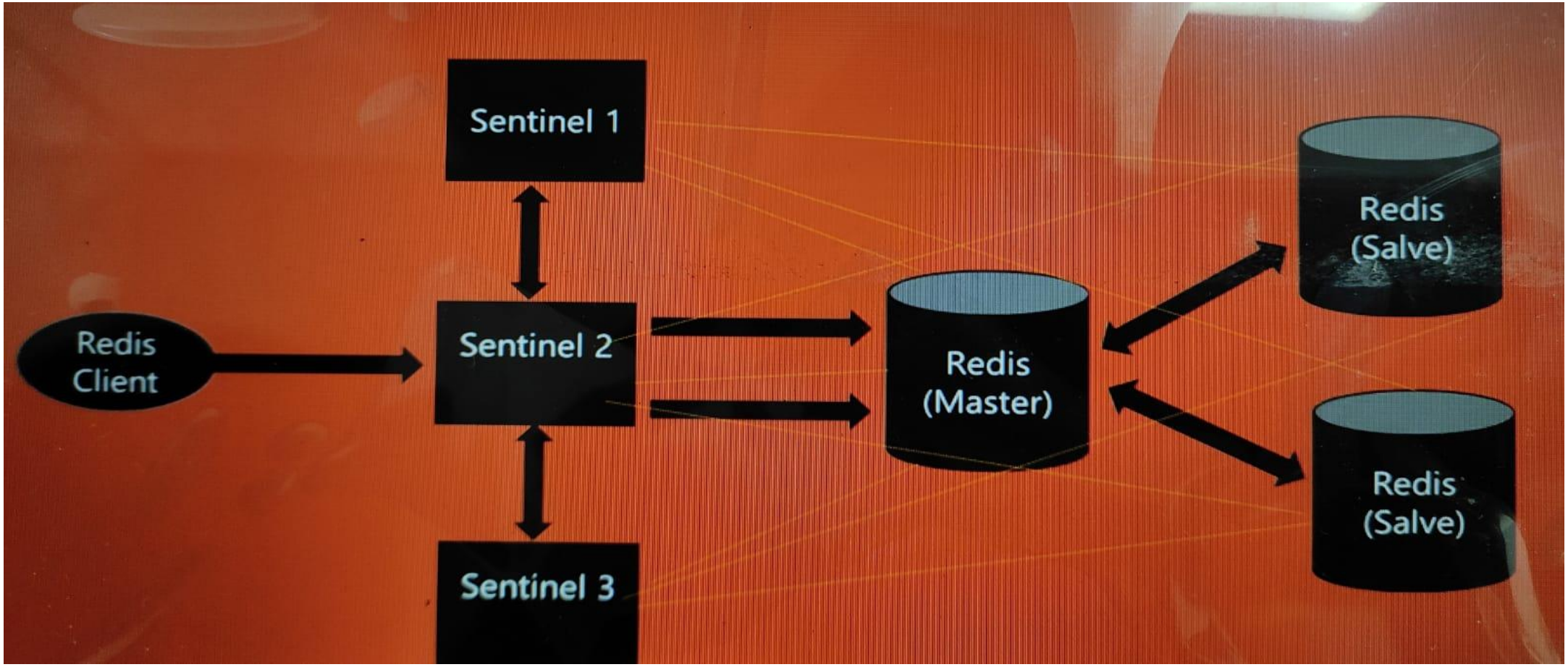
What if Redis Slave goes Down?

- Since all Redis Slaves are having latest set of data (by replication from Redis Master), if any of the Redis Slave goes down, other Redis Slave will serve read requests.

High Availability with Sentinel

- Redis sentinel is a system, designed to help managing Redis instances.
- The primary purpose of using sentinel is to provide a high availability system, by monitoring, notifying and providing instances failover.
- **Monitoring**. Sentinel constantly checks if your master and replica instances are working as expected.
- **Notification**. Sentinel can notify the system administrator, or other computer programs, via an API, that something is wrong with one of the monitored Redis instances.
- **Automatic failover**. If a master is not working as expected, Sentinel can start a failover process where a replica is promoted to master, the other additional replicas are reconfigured to use the new master, and the applications using the Redis server are informed about the new address to use when connecting.
- **Configuration provider**. Sentinel acts as a source of authority for clients service discovery: clients connect to Sentinels in order to ask for the address of the current Redis master responsible for a given service. If a failover occurs, Sentinels will report the new address.

Sentinel:

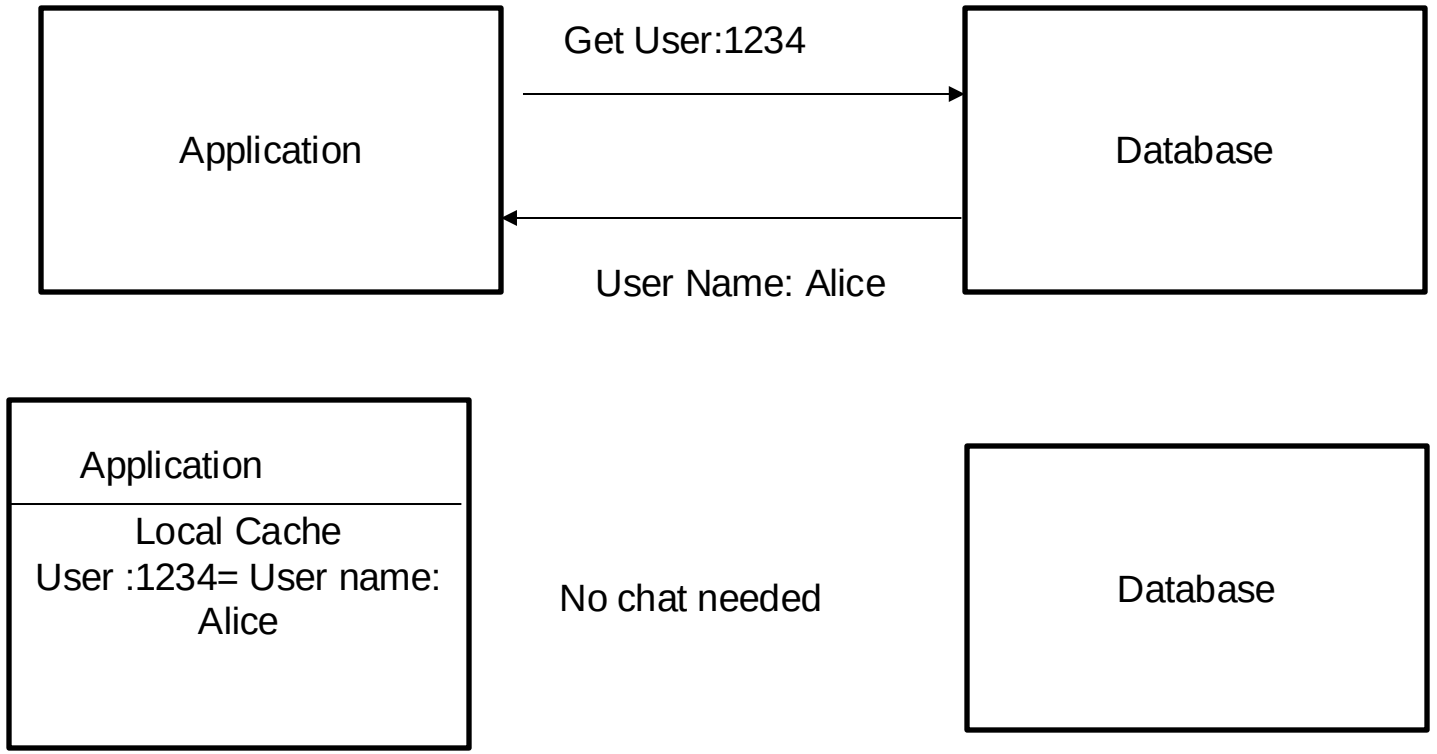

Sentinel 1
Redis Client
Sentinel 2
Sentinel 3
Redis (Master)
Redis (Salve)
Redis (Salve)

- **Where does Redis fit in CAP theorem?**
- CAP: Consistency, Availability and Partition Tolerance.
- Redis is **AP** system. Lets understand why Redis doesn't provide strong Consistency.
- What happens when Redis Master receives write request from Client:
- It does acknowledge to Client.
- Redis Master replicates the write request to 1 or more slaves. (Depends on Replication factor).
- Here you can see, Redis Master does not wait for replication to be completed on slaves and does acknowledgment to client immediately.
- Now lets us assume, Redis Master acknowledged to client and then got crashed. Now one of the Redis Slave (that did not receive the write) will get promoted to Redis Master, loosing the write forever.

Client-side caching in Redis

- Server-assisted, client-side caching in Redis
- Client-side caching is a technique used to create high performance services.
- It exploits the memory available on application servers, servers that are usually distinct computers compared to the database nodes, to store some subset of the database information directly in the application side.
- Normally when data is required, the application servers ask the database about such information, like in the following diagram:

- When client-side caching is used, the application will store the reply of popular queries directly inside the application memory, so that it can reuse such replies later, without contacting the database again:



- While the application memory used for the local cache may not be very big, the time needed in order to access the local computer memory is orders of magnitude smaller compared to accessing a networked service like a database.
- Since often the same small percentage of data are accessed frequently, this pattern can greatly reduce the latency for the application to get data and, at the same time, the load in the database side.

Advantages Of Client side caching

- Usually the two key advantages of client-side caching are:

1. Data is available with a very small latency.
2. The database system receives less queries, allowing it to serve the same dataset with a smaller number of nodes.

Problem with local cache:

- A problem with the above pattern is how to invalidate the information that the application is holding, in order to avoid presenting stale data to the user.
- For example after the application above locally cached the information for user:1234, Alice may update her username to Flora. Yet the application may continue to serve the old username for user:1234.
- Sometimes, depending on the exact application we are modeling, this isn't a big deal, so the client will just use a fixed maximum "time to live" for the cached information.
- Once a given amount of time has elapsed, the information will no longer be considered valid

The Redis implementation of client-side caching

- The Redis client-side caching support is called *Tracking*,
- In the ***default mode***, the server remembers what keys a given client accessed, and sends invalidation messages when the same keys are modified. This costs memory in the server side, but sends invalidation messages only for the set of keys that the client might have in memory.
- In the ***broadcasting*** mode, the server does not attempt to remember what keys a given client accessed, so this mode costs no memory at all in the server side. Instead clients subscribe to key prefixes such as object: or user:, and receive a notification message every time a key matching a subscribed prefix is touched.

Tracking

1. Clients can enable tracking if they want. Connections start without tracking enabled.
2. When tracking is enabled, the server remembers what keys each client requested during the connection lifetime (by sending read commands about such keys).
3. When a key is modified by some client, or is evicted because it has an associated expire time, or evicted because of a *maxmemory* policy, all the clients with tracking enabled that may have the key cached, are notified with an *invalidation message*.
4. When clients receive invalidation messages, they are required to remove the corresponding keys, in order to avoid serving stale data.

Pipelining in redis

- In Redis, pipelining is a technique used to improve the efficiency of executing multiple commands in a batch. Normally, when you send a command to Redis, it processes that command and sends a response back before you can send the next command. This round-trip communication can introduce latency, especially when you need to execute multiple commands in succession.
- Pipelining allows you to send multiple commands to Redis in a single batch, without waiting for individual responses. Redis queues up these commands and processes them in a single batch, reducing the round-trip time and improving overall performance. After sending all the commands, you can then read the responses from Redis.

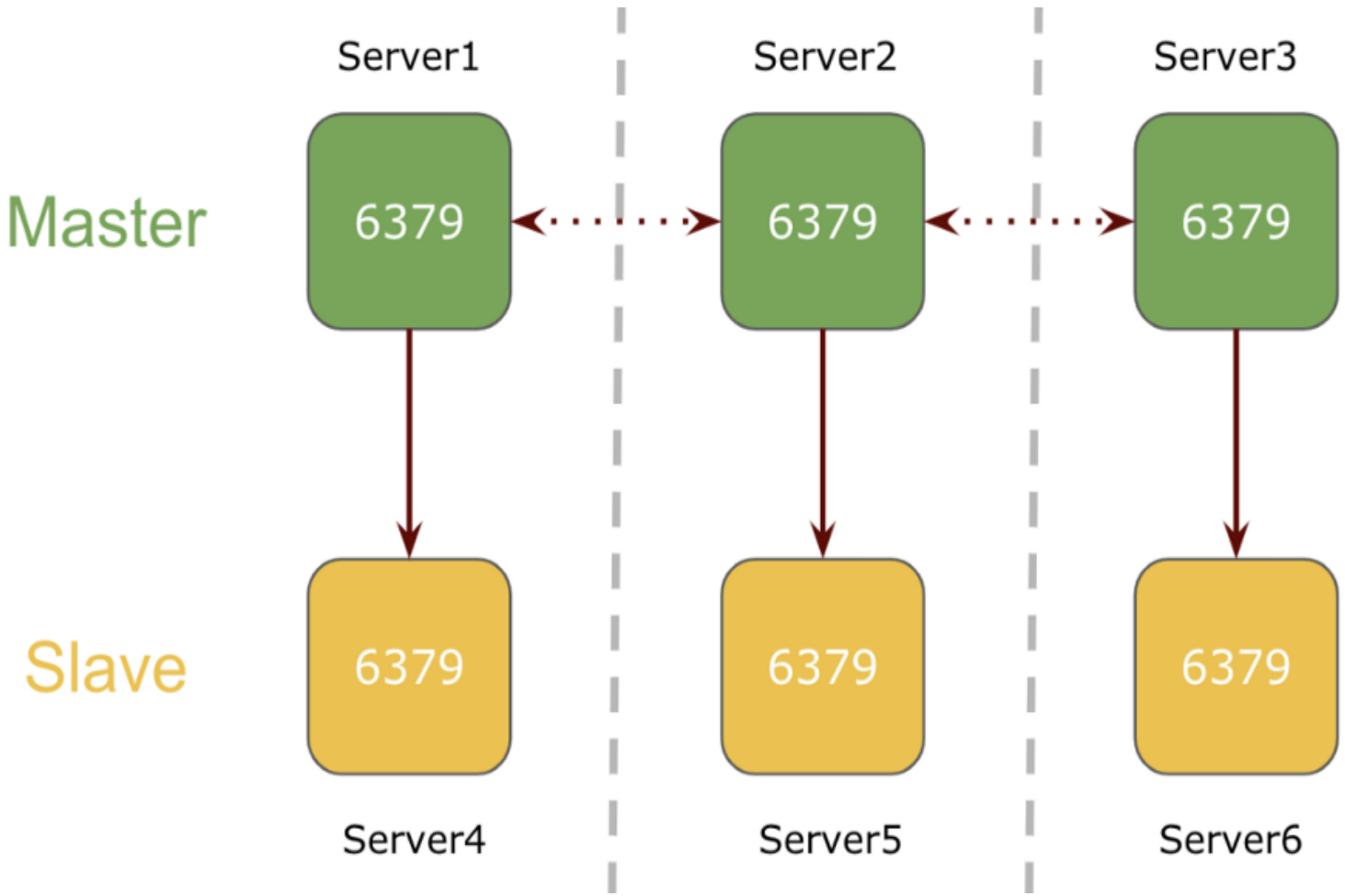
- Here's how you can use pipelining in Redis:
- **Send Multiple Commands**: Instead of sending one command at a time and waiting for the response, you send multiple commands sequentially without waiting for a response for each command.
- **Read Responses**: After sending all the commands, you read the responses in the same order you sent the commands. Redis will provide the responses for each command in the same order.

```
import redis
# Connect to Redis
r = redis.Redis(host='localhost', port=6379, db=0)
# Start pipelining
pipe = r.pipeline()
# Queue up multiple commands
pipe.set('key1', 'value1')
pipe.set('key2', 'value2')
pipe.get('key1')
pipe.get('key2')
# Execute all queued commands in a single batch
responses = pipe.execute()
# Print responses
print(responses)  # You will get the responses for the get commands
```

Sharding

- Redis sharded data automatically into the servers.
  Redis has a concept of the **hash slot** in order to split data. All the data are divided into slots.
  There are 16384 slots. These slots are divided by the number of servers.
- If there are 3 servers; A, B and C then
- Server 1 contains hash slots from 0 to 5500.
- Server 2 contains hash slots from 5501 to 11000.
- Server 3 contains hash slots from 11001 to 16383.

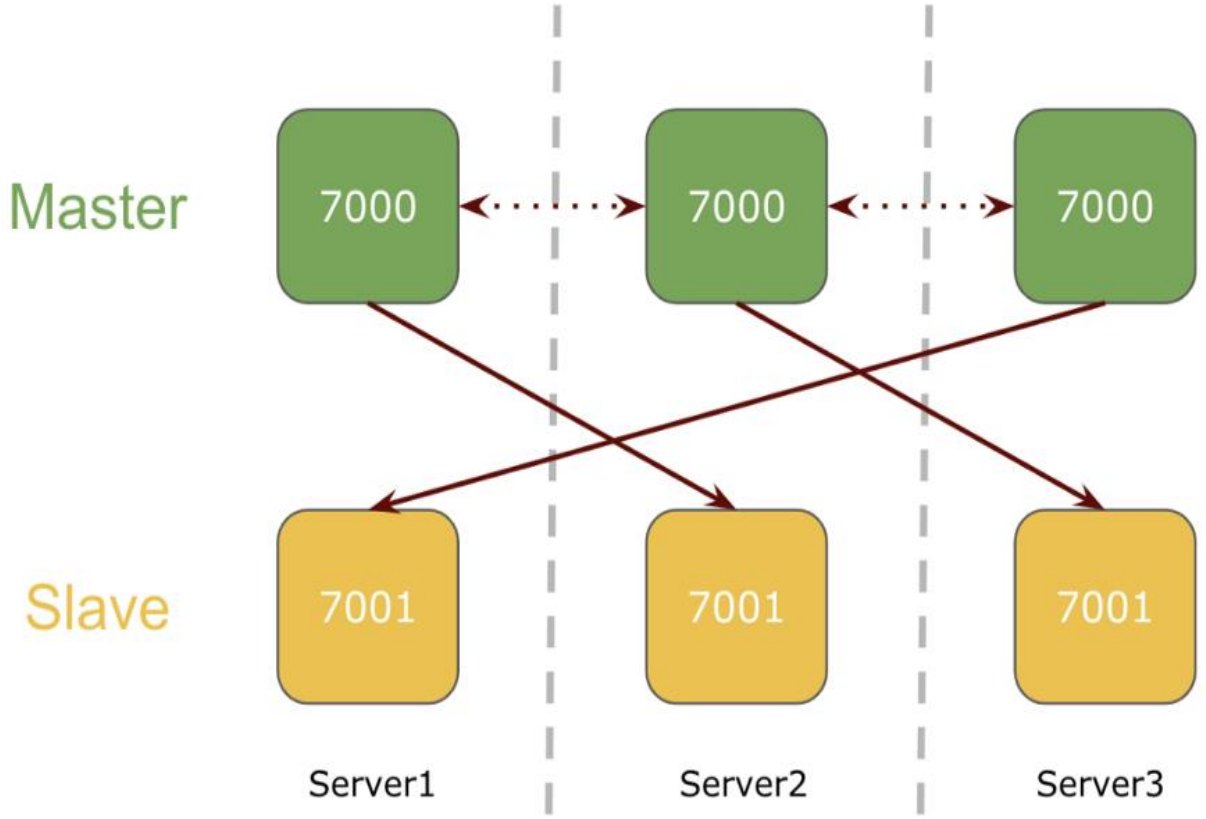
6 Node M/S Cluster



In a 6 node cluster mode, 3 nodes will be serving as a master and the 3 nodes will be their respective slave. Here, Redis service will be running on port 6379 on all servers in the cluster. Each master server is replicating the keys to its respective Redis slave node assigned during the cluster creation process.

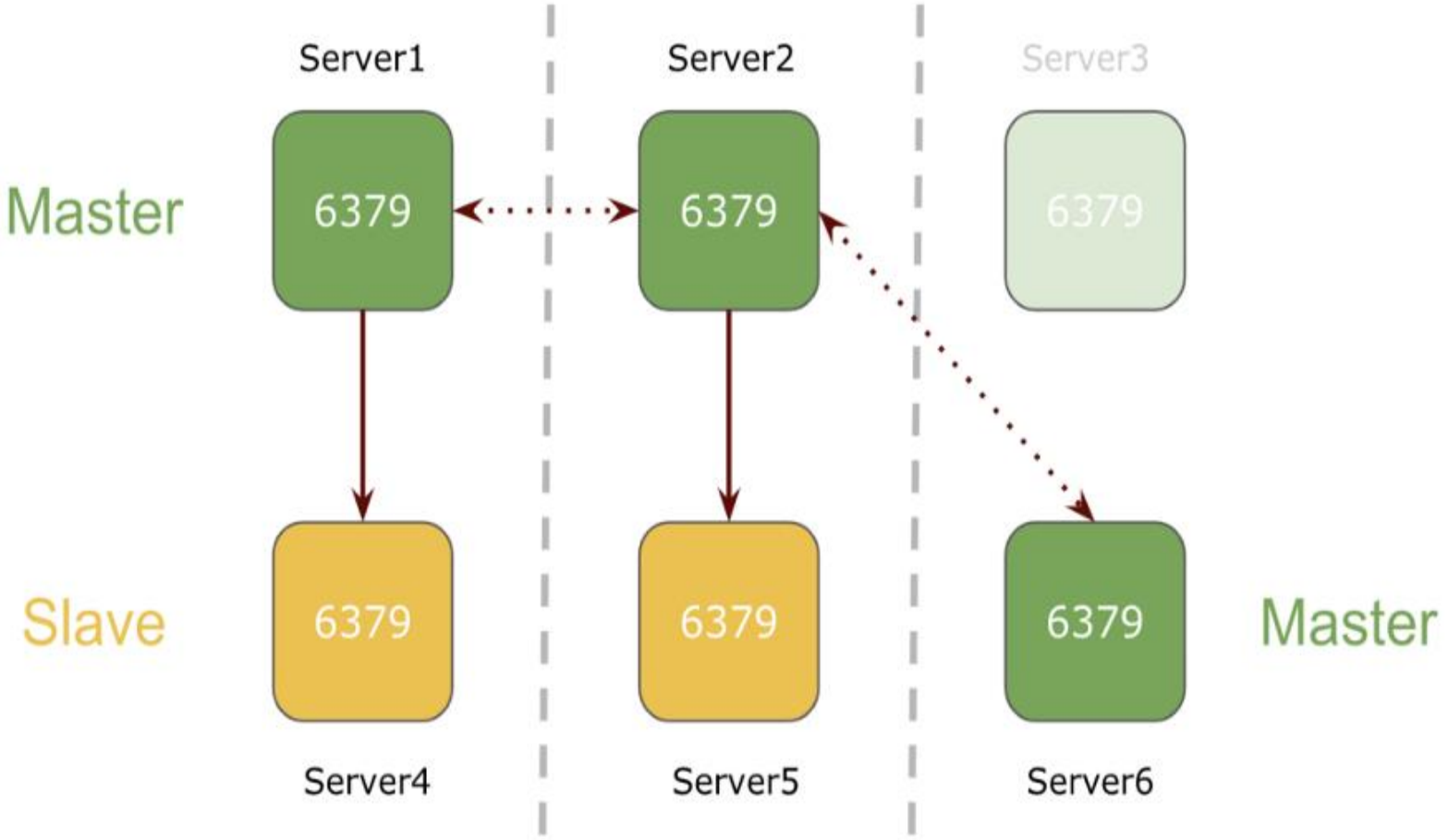
3 Node M/S Cluster

In a 3 node cluster mode, there will be 2 redis services running on each server on different ports. All 3 nodes will be serving as a master with redis slave on cross nodes. Here, two redis services will be running on each server on two different ports and each master is replicating the keys to its respective redis slave running on other nodes.



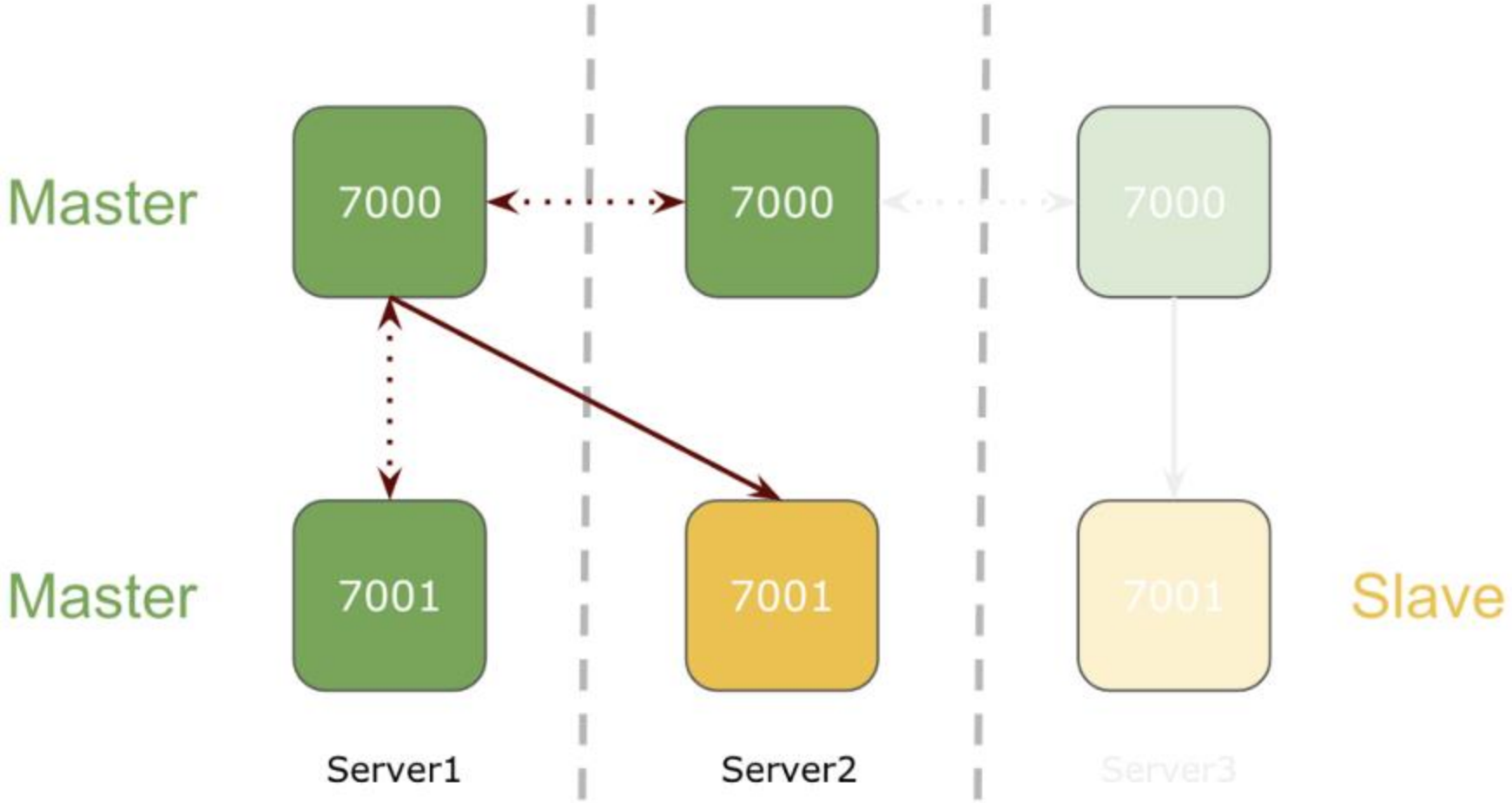
WHAT IF Redis Goes Down
1 node goes down in a 6 node Redis Cluster

If one of the nodes goes down in the Redis 6-node cluster setup, its respective slave will be promoted as the master.
In the above example, master Server3 goes down and its slave Server6 is promoted as the master.



1 node goes down in a 3 node Redis Cluster

If one of the nodes goes down in the Redis 3-node cluster setup, its respective slave running on the separate node will be promoted to master.
In the above example, Server 3 goes down and slave running on Server1 is promoted to master.



Redis replication

- This system works using three main mechanisms:

1. When a master and a replica instances are well-connected, the master keeps the replica updated by sending a stream of commands to the replica to replicate the effects on the dataset happening in the master side due to: client writes, keys expired or evicted, any other action changing the master dataset.
2. When the link between the master and the replica breaks, for network issues or because a timeout is sensed in the master or the replica, the replica reconnects and attempts to proceed with a partial resynchronization: it means that it will try to just obtain the part of the stream of commands it missed during the disconnection.
3. When a partial resynchronization is not possible, the replica will ask for a full resynchronization. This will involve a more complex process in which the master needs to create a snapshot of all its data, send it to the replica, and then continue sending the stream of commands as the dataset changes.

Important Topics

- What is a key-value database, and how does it differ from traditional relational databases?
- Difference between k? n and k* n commands in redis
- Write the command to insert and retrive the value in Redis for the following table

SID	SNAME	SAGE
S1	ABC	20
S2	CDE	24

- List down any 5 examples of key value databases.
- Explain the concept of key-value storage in Redis.
- What is in-memory database.
- What are the basic data types available in Redis?
- Explain “expire “ and “TTL” commands in Redis

- working of sentinel and list its functions.
- Redis Vs RDBMS databases
- Justify why Redis does not provide strong consistency with its architecture
- Redis String commands with example
- List and explain the modes in Redis Tracking method
- Explain the three main mechanism of Redis replication
- Explain Where does Redis fit in CAP theorem?
- Explain the problems in local cache and how to solve that problem

- Explain sharding in Redis
- Explain client side caching in redis
- a. Provide examples of scenarios where Redis is a suitable choice as a data store.
- b. Convert the below Employee relation into key- value database schema
- Explain all “Hash” commands with example
- Elaborate Redis architecture
- Explain commands in Redis with Use , Syntax and Example
- Discuss pub/sub design pattern in redis?
- Explain Pipelining in Redis