UNIVERSITÉ LIBRE DE BRUXELLES

INFO-H-415 Advanced Databases

Key-value stores and Redis

Fatemeh Shafiee	000454718
Raisa Uku	000456485

December 2017

Contents

List of Figures

List of Tables

1 Introduction

Relational or SQL databases have been the main mechanism for storing, managing and retrieving data for decades. For a relational database to be effective, the data has to be in transactional mode, consistent and structured. be structured in a very organized way in order to be stored (Das, 2015, p. 4).Some examples of these databases are MySQL, Oracle, IMB DB2, MS SQL Server, PostgreSQL and Microsoft Azure.

On the other hand, if the amount of provided data is huge, unstructured and it has not a fixed schema (such as articles, photos, social media data, videos, etc.), a relational database can not store, manage, retrieve or analyze all of this data and the result will be a complicated data model to store simple data. NoSQL databases, which offer ease of access and much greater flexibility than relational databases can deal with massive amounts of unstructured data. Exponentially increase in the amount of data, which leads to the era of big data is the real NoSQL motivator, as can do things that traditional relational databases cannot.

The purpose of this report is to introduce Redis which is a key-value NoSQL database, its use cases, data types, supported languages and compare its performance with a traditional relational database. A Twitter-like social network application has been implemented in order to compare Redis as a key-value store to a relational database, MySQL.

2 NoSQL Databases

2.1 Introduction to NoSQL Databases

NoSQL which stands for "NotOnlySQL", is a term that defines an approach to design non-relational databases and work with large sets of distributed data. These databases are more flexible as they do not rely on predefined structures such as tables, columns or rows to organize and retrieve data. When comparing relational databases and NoSQL databases, it can be concluded that NoSQL databases are more scalable and provide flexible data modeling, higher availability and performance. Therefore, NoSQL databases are getting popular in big data management and analysis since less organized data need to be stored in an efficient way.

NoSQL databases are sometimes referred to as cloud databases, non-relational databases or Big Data databases. These databases have provided a competitive advantage in many industries that are dealing with huge amount of data and also in modern business applications.

There are four different varieties of NoSQL databases with their own specific attributes:

Key-value stores: Key-value databases implement a data model that pairs a unique key with an associated value. Every value stored in the database has a key which can be used to search for values. These values can be text, video, JSON, document, etc. Key-value databases are extremely scalable for session management and caching in web applications such as social networking, gaming, media sharing and Q&A portals. Some popular choices of these stores are Riak, Aerospike, Berkeley DB, MemchacheDB and Redis (Das, 2015, p. 15-16).

Document stores: These databases are designed to store and retrieve document-oriented information or data that do not have a fixed schema, also known as semi-structured data, typically in BSON, JSON or XML format. Document stores are also called "Document databases" and are used for content management and mobile application data handling. These NoSQL databases are based on the idea of key-value stores which means each document is assigned a unique key and contains complex data. This assigned key is used to retrieve the document. Examples of document stores include MongoDB, Couchbase Server, CouchDB , DocumentDB, MarkLogic, Terrastore, OrientDB and RavenDB.

Graph stores: Graph databases are based on graph theory. They store data as nodes (vertices) and relations (edges or directed links). Both, nodes and relations can have properties associated with them. The structure of these databases provides interesting patterns between the nodes. Graph databases are mostly used in systems such as CRM or reservation systems where mapping the relationships is required. Some of the popular graph stores are AllegroGraph, IBM Graph, Neo4j and Titan.

Wide-column stores: These databases store data in tables but they are designed with the philosophy of storing data in columns rather than rows. Wide-column stores can organize and retrieve large data volumes faster than conventional relational databases and in a very cost-effective way. This type of NoSQL database is used for recommendation engines, catalogs, fraud detection and other types of data processing. Google BigTable, Amazon DynamoDB, Cassandra, HBase, BigTable and HyperTable are examples of wide-column stores (Das, 2015, p. 11-12).
.

Types	**Performance**	**Scalability**	**Flexibility**	**Complexity**
Key-Value Store	**high**	**high**	**high**	none
Column Store	**high**	**high**	moderate	low
Document	**high**	variable(hight)	**high**	low
Graph Database	variable	variable	**high**	**hight**

Table 1: Types of NoSQL databases
(Woudehouse, 2016)

2.2 Key-Value Databases

A Key-value store is designed for storing, retrieving, and managing data in form of key and value pairs. Therefore, data is represented as a collection of key-value pairs. The key in a each pair must be unique to allow a user to access the value associated with that key. The value can be anything, it might be a long or short text, a number, an image, a programming or markup code, etc. For instance, for a search engine, data is stored in a form to associate each keyword (key) to a list of documents (values) containing that keyword. A key-value database can be used in various domains, such as (*What is a Key-Value Database?*, 2016):

Ecommerce: shopping cart contents, product categories, product details, product reviews.

General Web/Computers: user profiles, session information, article/blog comments, Emails, status messages.

Networking/Data Maintenance: telecom directories, Internet Protocol (IP) forwarding tables, Data deduplication.

3 Redis

Redis (REmote DIctionary Server, written by Salvatore Sanfilippo in 2006) is a NoSQL, advanced and popular **key-value data store**. It support powerful data structures, such as Strings, Lists, Sets, Sorted Sets and Hashes and it is sometimes called: **the data structure server**. Key must be a String, but value can be any of the data types mentioned. Given that interesting set of data structures, Redis is known as Swiss Army knife of data type storage (Silva & Tavares, 2015).

Furthermore, Redis is an in-memory database, meaning that all data by default are saved in the primary memory (RAM) in form of key-value pairs. This makes read and write operations for Redis very fast compared to Relational Databases. On the other hand, the limitation in size of the Primary Memory, limits the Redis data-store size, forcing it to store only data that needs to be accessed, modified and inserted very rapidly.

It is not very common to use Redis as as a standalone database but sometimes it can be considered as the main primary database in some applications. In order to improve the performance, Redis is usually used along with a relational database (Carlson, 2013) or even with other non-relational database such as MongoDB for caching (Cummings, Eftekhary, & House, 2015). It can be concluded that it is not necessary to switch to Redis in order to take advantage of this key-value store, but rather use it in existing environments to do things that were not possible before or to fix existing problems.

According to redis.io, there are multiple factors having direct consequences on Redis performance. Some of these factors are included in the following (*How fast is Redis?*, 2017):

Network bandwidth and latency: It is a good practice to use the ping program to quickly check the latency between the client and server hosts is normal. Regarding the bandwidth, it is generally useful to estimate the throughput in Gbit/s and compare it to the theoretical bandwidth of the network.

CPU: Being single-threaded, Redis favors fast CPUs with large caches and not many cores. At this game, Intel CPUs are currently the winners. When client and server run on the same box, the CPU is the limiting factor with redis-benchmark.

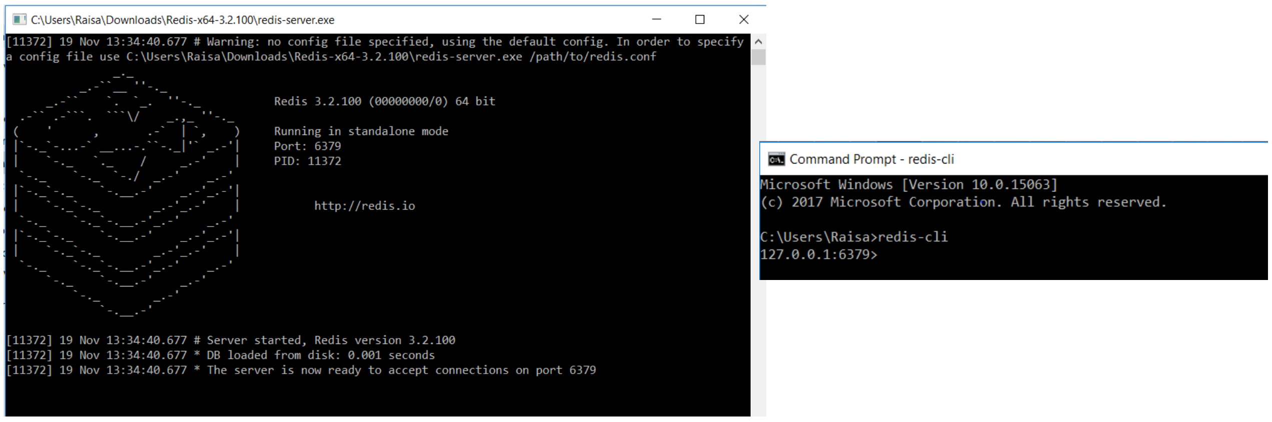
Virtual Machine: Redis runs slower on a VM compared to running without virtualization using the same hardware. If there is a chance to run Redis on a physical machine this is preferred. However this does not mean that Redis is slow in virtualized environments.

Ethernet network : When an ethernet network is used to access Redis, aggregating commands using pipelining is especially efficient when the size of the data is kept under the ethernet packet size (about 1500 bytes). Actually, processing 10 bytes, 100 bytes, or 1000 bytes queries almost result in the same throughput.

Number of client connections : Every operation is sent to Redis within the context of a connection from the client application. The maximum number of concurrent connections to Redis server is always limited, whether by operating system, Redis' configuration, or the service provider's plan. Therefore, having enough free resources will allow the connection of new application clients or an administrative session.

3.1 Architecture

The main components of Redis architecture are Redis Server and Client. Redis-server is the Redis data store and it forms the most significant part of its architecture. It is responsible for storing data and serving to the client. Redis-cli is a command-line interface that can perform any Redis command.
Figure 1 below illustrates how Redis-server is started in our Windows machine. By default, it wait for connections from the client at port 6379.


C:\Users\Raisa\Downloads\Redis-x64-3.2.100\redis-server.exe
Command Prompt - redis-cli

```
[11372] 19 Nov 13:34:40.677 # Warning: no config file specified, using the default config. In order to specify
a config file use C:\Users\Raisa\Downloads\Redis-x64-3.2.100\redis-server.exe /path/to/redis.conf

                                    Redis 3.2.100 (00000000/0) 64 bit

                                    Running in standalone mode
                                    Port: 6379
                                    PID: 11372

                                          http://redis.io

[11372] 19 Nov 13:34:40.677 # Server started, Redis version 3.2.100
[11372] 19 Nov 13:34:40.677 * DB loaded from disk: 0.001 seconds
[11372] 19 Nov 13:34:40.677 * The server is now ready to accept connections on port 6379
```

```
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\Raisa>redis-cli
127.0.0.1:6379>
```

Figure 1: Redis-server and Redis-cli

A way for datastore persistence is needed as Redis stores everything in Primary Memory, which is volatile. Redis provides two main **persistence options**: RDB persistence and AOF persistence (*Redis Documentation*, 2017). It is possible to disable persistence at all, if we are not interested for data to exist after the server stops running. Furthermore, Redis offers the opportunity to combine both this options in the same instance. The *RDB persistence* performs point-in-time snapshots of the dataset at specified intervals and save them on disk while *AOF persistence* logs every write operation received by the server, and play it again at server startup, reconstructing the original dataset. The trade-off that exists between RDB and AOF, is used to decide which option is appropriate in a specific situation. Table below lists the advantages and disadvantages of each option:

Persistence Option	**Advantages**	**Disadvantages**
RDB	1. Very compact single-file point-in-time representation of the Redis data. Perfect for backups as it allows you to easily restore different versions of the data set in case of disasters. 2. Very good for disaster recovery (a single compact file can be transferred to far data centers) 3. Compared to AOF it allows faster restarts with big datasets.	1. RDB is NOT good, when it is comes to minimizing the chance of data loss (in case Redis stops working without a correct shutdown). Data that are set after the last snapshot, can be lost. 2. As RDB needs to fork() often, this process can be time consuming if the dataset is big. This may cause Redis to stop serving clients (for some millisecond to one second). AOF also needs to fork() but in the case of AOF, it is possible to tune how often to rewrite the logs.
AOF	1. Different *fsync policies* (no fsync at all, fsync every second, fsync at every query) offered by AOF makes Redis more durable. Using the default policy (second one) only one second of writes will be lost. 2. There are no problems in case of a power outage, as AOF is an append only log. The redis-check-aof tool is used to fix any problem of this kind. Furthermore, when the AOF file gets too big, Redis is able to automatically rewrite it in background. 3. As AOF contains a log of all the performed operations one after the other, it is possible to export AOF file and remove the latest command (ex. FLUSHALL command).	1. For the same dataset AOF files are usually bigger compared to RDB files, as it writes the disk for every operation. 2. Depending on the *fsync policy* used, AOF can be slower than RDB

Table 2: RDB and AOF Persistence Options

Using a model with unified AOF and RDB into a single persistence model is recommended and this is where Redis foundation aims to end up in the future. (*Redis Documentation*, 2017)
As Redis does not provide any mechanism for datastore backup and recovery, the data will be lost in case of any hard disk or other hardware failures. Therefore, by using Redis in a **Replicated Environment** (with Master Server and Slave Servers) it is possible to handle this problem. All the Slave Servers will contains the same data as the Master Server. The Redis installation will continue working if a slave fails. Whenever a slave restart working, the master automatically updates the slave with the new data. In case of a write operation, the master replicates all the new data to all the slaves, whereas when read or sort operations occurs, master distributes them to the slaves. (*Redis Documentation*, 2017)
Furthermore, to take the advantage of storing more data and to be used for highly-available and scalable environments, **Redis Cluster** provides the opportunity to run a Redis installation where data is automatically shared across multiple nodes. Each node is a Redis Server configured as a cluster node. In addition, even with Redis Cluster, we can rely on replication to avoid losing the data due to disk crashes. To implement this, each Redis node is converted to a Master Server and every master we keep a slave(*Redis Documentation*, 2017) (Das, 2015, p. 251-260). Persistence is needed in all the above mentioned situations.

3.2 Keys Commands

In Table 3, a list of some basic commands related to keys is illustrated (*Redis Documentation*, 2017).

Command	Description
DEL key	This command deletes the key, if it exists
DUMP key	This command returns a serialized version of the value stored at the specified key
EXISTS key	This command checks whether the key exists or not
EXPIRE key (second)	Sets the expiry of the key after the specified time
EXPIREAT key timestamp	Sets the expiry of the key after the specified time. Here time is in Unix timestamp format
PEXPIRE key milliseconds	Sets the expiry of the key in Unix timestamp specified as milliseconds
KEYS pattern	Finds all keys matching the specified pattern
MOVE key db	Moves a key to another database
PERSIST key	Removes the expiration from the key
PTTL key	Gets the remaining time in keys expiry in milliseconds
TTL key	Gets the remaining time in keys expiry
RANDOMKEY	Returns a random key from Redis
RENAME key newkey	Changes the key name
RENAMENX key newkey	Renames the key, if a new key doesn't exist
TYPE key	Returns the data type of the value stored in the key

Table 3: Commands for Set Data Type

3.3 Redis Data Structures

Redis, as a Data Structure Server, supports a variety of in- build data structures, providing in this way the users with diverse mechanisms to arrange their data. This makes Redis *more than just key-value store* and differentiate it from other key-value NoSQL databases. Redis approach consist on *providing specific data structures for specific problems* and the supported data structures are: Strings, Lists, Sets, Sorted Sets and Hashes (Das, 2015, p. 37-68) (Silva & Tavares, 2015, p. 27-54).

3.3.1 Strings

The basic data type in Redis are Strings. In Redis strings are called Simple Dynamic String (SDS). Despite the name, strings in Redis can be considered as a byte array (with maximum size 512 MB) that can hold not only strings but also integers, bitmap, image files and serializable objects. An in-build mechanism is used by Redis to detect the type of data stored in these byte arrays. Typical uses of Redis strings are: to store the IDs of objects, e.g. session IDs or as atomic counters. Table.4 shows the main commands used in Redis for Strings, categorized on three main groups:

Commands Group	Command Name	Description
Setter and getter commands used to set or get values in Redis	GET	Gets the value for a key
	SET key	Sets a value against a key
	SETNX key	Set a value against a key only if a key does not exist. Otherwise no overwrite is done
	GETSET key	Gets the old value and sets a new value
	MGET key1 key	Gets all the corresponding values of the keys
	MSET key	Sets all the corresponding values of the keys
	MSETNX	Sets all the corresponding values of the keys, if all the keys don‘t exist. If one exists, then no values are set
Data clean commands used for managing the life cycle of a value. (By default, the values do not have an expiry time)	SET PX/EX	Removes the values and the key gets expired after expiry time in milliseconds.
	SETEX	Removes the values and the key gets expired after expiry time in seconds.
Utility commands	APPEND	Appends to the existing value or sets the value if it does not exist
	STRLEN	Returns the length of the value stored as string
	SETRANGE	Overwrites the string at the given offset
	GETRANGE	SGets the substring value from the given offsets

Table 4: Commands for String Data Type

3.3.2 Integers and Floats

For integers and floats the main commands of the category “Setters and getters” and “Data clean” are the same with the above commands for Strings. Whereas, the Utility Commands used to manipulate integers and floats are as follows in Table.5 :

Commands Group	Command Name	Description
Utility commands	APPEND	Concatenates the existing integer with a new integer
	DECR	Decrement the value by one
	DECRBY	Decrement the value by a given value
	INCR	Increment the value by one
	INCRBY	Increment the value by a given value
	INCRBYFLOAT	Increment the value by a given floating value

Table 5: Commands for Integer and Float Data Type

3.3.3 Hashes

Hashes data structure in Redis are used to store a collection of fields associated with their values and map this against a key. Hashes can be used for storing data for user profiles.

Commands Group	Command Name	Description
Setter and getter commands	HGET/HMGETS	Gets the value of a field/fields for a key
	HGETALL	Gets all the values and the fields for a key
	HSET/HMSETS	Sets the value of a field/fields for a key
	HVALS	Gets all the values in the hash for the Key
	HSETSNX	Sets the value of a field for a key, if the field does not exist
	HKEYS	Gets all the fields in the Hash for the Key
Data clean commands	HDEL	Deletes the fields foe a Key
Utility commands	HEXISTS	Checks for the existence of a field for a key
	HINCRBY/ HINCRBYFLOAT	Increments the value(integer/float) of a field for a key

Table 6: Commands for Hash Data Type

3.3.4 Lists

Lists in Redis are implemented as a linked list, as they are designed to have a faster write performance then read performance. Hence, an element can be added on the list form the head or tail, but the performance for accessing an element can degrade if the number of elements in the list is high. One of the cases they are used is for log messages. Table.7 shows the main commands used for lists. Commands that start with L are interpreted to be executed from the Left or Head of the List. Otherwise, they are interpreted to be executed from the Right or the tail of the list (when they start by R).

Commands Group	**Command Name**	**Description**
Setter and getter commands	LPUSH/ RPUSH	Add the values to a list (from the left/right of the list)
	LPUSHX/RPUSH	Add the values to a list, if the key exists
	LINSERT	Inserts a value in the list after the pivot position
	LSET	Sets the value of an element in a list based on the index mentioned
	LRANGE	Gets the sub list of elements based on he start index and the end index
Data clean commands	LTRIM	Deletes the elements outside the range specified
	RPOP	Removes the last element
	LREM	Removes the element at the index point specified
	LPOP	Removes the first element of the list
Utility commands	LINDEX	Gets the element from the list based on index
	LLEN	Gets the length of the list

Table 7: Commands for List Data Type

3.3.5 Sets

Sets in Redis represent an unordered collection of elements. Duplicate values are not allowed, meaning that the values in sets are unique. Unlike Lists, they show constant timing for adding, deleting and checking the existence of an element. Sets are used more for analytical purposes: for example how many people browse a product in an e-commerce website and how many purchase that product. The main commands used for Sets are:

Commands Group	Command Name	Description
Setter and getter commands	SADD	ADD one or more element to the Set
Data clean commands	SPOP	Removes and returns a random element from the set
	SREM	Removes and returns the specified element from the set
Utility commands	SCARD	Gets the number of elements in a Set
	SDIFF	Gets the list of elements from the first set after substracting its elements from the other mentioned sets
	SDIFFSTORE	Similar with SDIFF, but here the result is stored in a mentioned set
	SINTER	Gets the common elements in all the sets mentioned
	SINTERSTORE	Similar with SINTER, but here the result is stored in a mentioned set
	SISMEMBER	Finds if the value is a member of the set
	SMOVE	Moves members from one set to another set
	SRANDMEMBER	Gets one or multiple members form the set
	SUNION	Adds multiple sets
	SUNIONSTORE	Similar with SUNION, but here the result is stored in a set

Table 8: Commands for Set Data Type

3.3.6 Sorted Sets

Unlike the Sets, Redis Sorted Sets have the values sorted on the basics of an integer of float value called element score. Like Sets each element is unique and duplicate values are not allowed and they are used more for analytics purposes. The main commands used for Sets are:

Commands Group	**Command Name**	**Description**
Setter and getter commands	ZADD	Adds or updates one or more members in a Sorted Set
	ZRANGE	Gets the specified range in a Sorted Set
	ZRANGEBYSCORE	Gets elements from the Sorted Sets within the range by score that is given
	ZREVRANGEBYSCORE	Gets the elements from the Sorted Sets within the score given
	ZREVRANK	Returns the rank of the members in a Sorted Set
	ZREVRANGE	Returns the specified range of elements in the Sorted Set
Data clean commands	ZREM	Removes the specified elements in the Sorted Set
	ZREMRANGEBYRANK	Removes the members in a Sorted Set within the given indexes
	ZREMRENGEBYSCORE	Removes the members in a Sorted Set within the given scores
Utility commands	ZCARD	Gets the number of members in a Sorted Set
	ZCOUNT	Gets the number of members in a Sorted Set within the score boundaries
	SZINCRBY	Increases the score of an element in the Sorted Set
	ZINTERSCORE	Calculate the common elements in the Sorted Sets given by the specified keys, and store the results in a destination sorted set
	ZRANK	Gets the index of the element in a Sorted Set
	ZSCORE	Returns the score of the member
	ZUNIONSCORE	Computes the union of keys in a given sorted sets and stores the results in another sorted set

Table 9: Commands for Sorted Set Data Type

3.4 Comparison

In this section, Redis is compared to other database management systems according to DB-Engines Ranking. This website publishes monthly rankings of database management systems according to their popularity.
DB-Engines lists 339 different database management systems, which are classified according to their database model (relational DBMS, key-value stores etc.). In the following pie chart,the number of systems in each category is illustrated. According to this chart, key-value stores are the second popular database management systems (*DB Engine*, 2017).

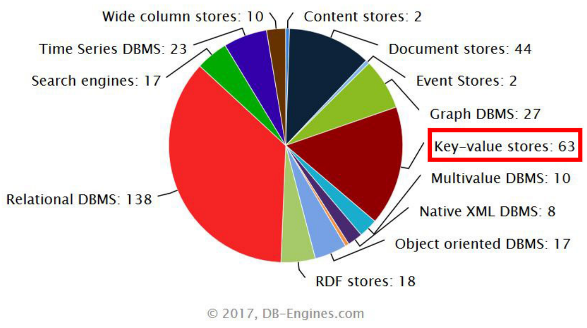


Figure 2: DBMS popularity broken down by database model

According to DB-Engines Ranking, Redis is the most popular key-value stores among all key-value stores. Also it has the ninth rank in comparison to other DBMS technologies which are mostly relational databases.

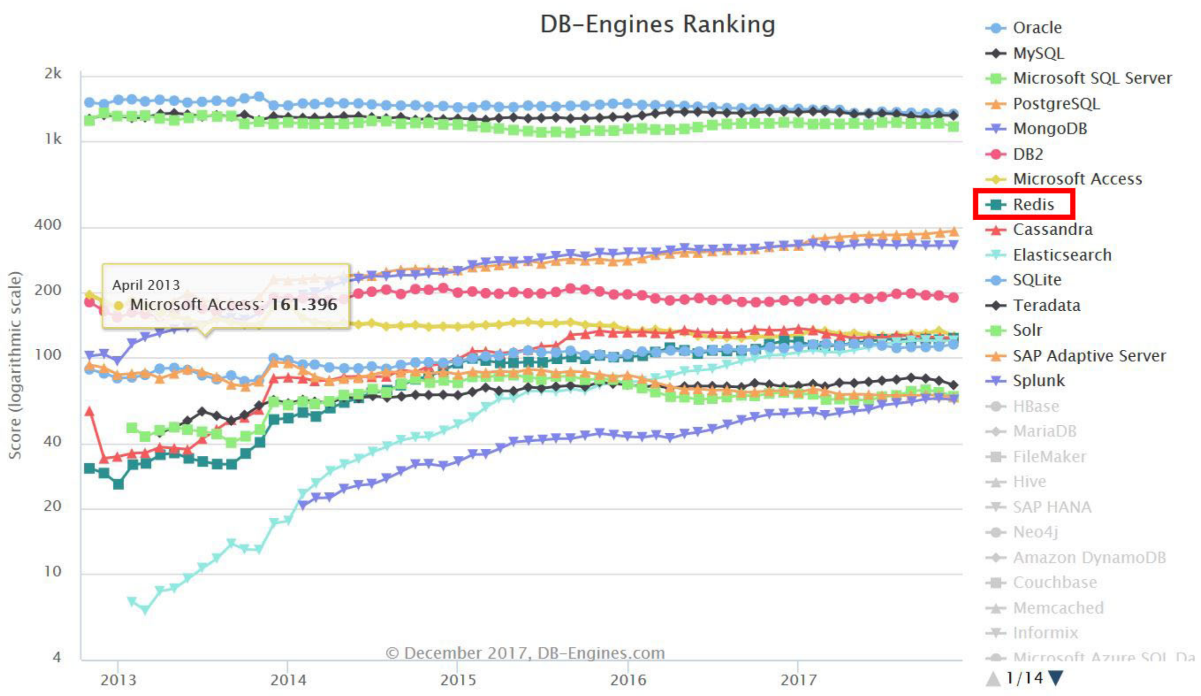
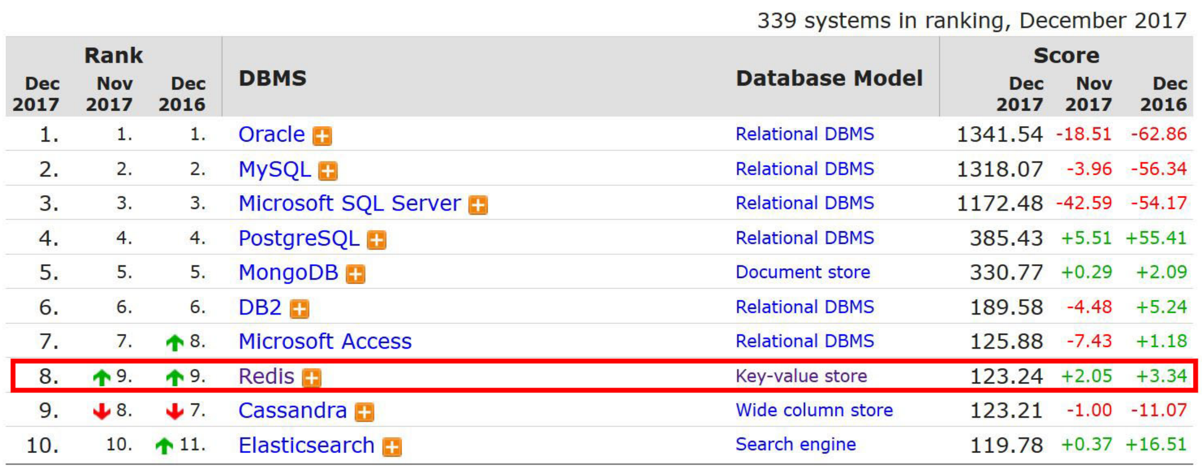


Figure 3: DBMS ranking according to their popularity in December 2017



339 systems in ranking, December 2017

Rank			DBMS	Database Model	Score		
Dec 2017	Nov 2017	Dec 2016			Dec 2017	Nov 2017	Dec 2016
1.	1.	1.	Oracle	Relational DBMS	1341.54	-18.51	-62.86
2.	2.	2.	MySQL	Relational DBMS	1318.07	-3.96	-56.34
3.	3.	3.	Microsoft SQL Server	Relational DBMS	1172.48	-42.59	-54.17
4.	4.	4.	PostgreSQL	Relational DBMS	385.43	+5.51	+55.41
5.	5.	5.	MongoDB	Document store	330.77	+0.29	+2.09
6.	6.	6.	DB2	Relational DBMS	189.58	-4.48	+5.24
7.	7.	↑ 8.	Microsoft Access	Relational DBMS	125.88	-7.43	+1.18
8.	↑ 9.	↑ 9.	Redis	Key-value store	123.24	+2.05	+3.34
9.	↓ 8.	↓ 7.	Cassandra	Wide column store	123.21	-1.00	-11.07
10.	10.	↑ 11.	Elasticsearch	Search engine	119.78	+0.37	+16.51

Figure 4: DBMS ranking according to their popularity

3.5 Popular Redis uses

Redis is a well-established open source project and has been used in production for years by big companies, including Twitter, GitHub, Tumblr, Pinterest, Instagram, Hulu, Flickr, and The New York Times.

Also, Redis is used often for: User Session Data Management, Real time analytics (such as counters, leader boards, most viewed, highest–lowest ranks), Recommendations (such as purchase or article recommendations based on common profile characteristics), Message queues for workflow and other jobs and Caching both static and interactive data (Nielsen, 2016).

According to (*Redis Labs*, 2017) there are several use cases proposed by applying Redis to various types of data including "Very Large Datasets", "Geospatial data or location data" and "Time series data". Redis can be utilized in : industries (Financial Services, Media and Entertainment, Retail/E-Commerce, Mobile), Fraud Detection, Personalization (Dynamic Pricing,Custom Advertising, Catalog Recommendation, Credit Risk Analysis, etc.) and Social Apps (Chat, Follow Tracking, Comments, Multi-player Games, Notifications, Ratings Tracking).

3.6 Advantages and Disadvantages of Redis

Redis is a brilliant solutions for specific application scenarios. When the provided data structures are used in the appropriate way, Redis will drive to outstanding performance. Mostly, it is used to analyze and process high-velocity datastore, competing in this way the DBMS that store everything in second storage and have slow read and write operations.As Redis stores everything in primary memory, it will offer speed in read and write of data. But it is important that while it is loading the dataset, this one has to fit comfortably in memory.
Redis has only commands and no support for a query language. Therefore, for coding a stored procedure, for instance, learning a programming language such as Lua is needed (Seguin, n.d.).Furthermore, Redis is a single-threaded server which means it is not designed to benefit from multiple CPU cores but most RDBMS are multi-processed (Oracle, PostgreSQL) or multi-threaded (MySQL, Microsoft SQL Server).In addition, as mentioned above Redis does not provide any mechanism for backup and recovery, but replication is used to solve this.

On the other hand, there are several advantages in using Redis. Redis is a data structure server and supports a wide variety of data types. It is open source and has an active community. Also, Redis is simple to install and has no dependencies and stores generic data types for any purpose. Moreover, Redis is easy to get started on a single cheap and free server. It supports good concurrency, low latency, protocol pipelining, implementing optimistic concurrent patterns, good usability and complexity ratio.

4 Practical Application

“Scaling Redis at Twitter” is the name of a great talk that covers the reasons why Twitter specialized Redis with two new data types that fit their use cases perfectly in order to get the performance they needed. On the other hand, One of the first examples displayed on Redis’ website is a Twitter-clone which is called *Retwis*.

Therefore, we have implemented a Twitter-clone application which mimics the basic design and functionality of the official Twitter application such as registration, follow or unfollow other users, post tweets, review other users’ tweets and search for a user or tweets.

4.1 Development Environment

Redis can be installed in multiple environments. For development purposes Redis has also support for Windows Systems, maintained by Microsoft Open Tech group. For developing our application on Redis, we have installed it in Windows and we have used the open-source Redis DB management tool called: *Redis Desktop Manager*. It can be downloaded from: `https://redisdesktop.com`.

There are Redis clients available for over 30 programming language. We have used a PHP client API to connect to the Redis server and to develop our application, called *PHP-Redis*. This API is also can be downloaded from: `https://github.com/phpredis/phpredis`.

The machine configuration for running the databases and the application, the version of programming language, Redis and MySQL are shown in the following:

Operating System: Microsoft Windows 10 / x64-based PC
Memory: 8GB
Cache: 4096MB L3 and 512MB L2
Processor: Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz, 2904 Mhz, 2 Core(s), 4 Logical Processor(s)

Redis version: 3.2.100
To get the redis version and all other related information, type “INFO” in “redis-cli” command prompt.

PHP Version: 7.1.11
MySQL Version: 10.1.28-MariaDB - mariadb.org binary distribution

4.2 Data Set

Since there are not any suitable dataset for our application and analysis section to compare Redis and MySQL, we created a dataset with the following properties:

Users: 1000000 users with distinct personal information (first name, last name, gender, email, password, sign up date, country and number of following, followers and tweets).

Country: A list of all countries.

Tweets: 928222 distinct tweet text which are preprocessed by NLP steps including “Tokenization” and “Stop words Removal”.

Words: This list contains all important words gained from all tweets after preprocessing steps.

Follow: We assign random number of followers and followings to each user. This list contains 2225000 relationships between the users.

4.3 Data structure in Redis

Users and tweets are the two types of objects that hold the most important information in Twitter. A User object contains basic personal information and the number of followers, following and posted tweets. In this section, we will explain how to design data structures in Redis, to store and retrieve users and tweets. In the following, the process of creating the data structures for users are defined.

4.3.1 Users

Users and their information are the main parts of a social networking website such as Twitter. Therefore, how to design the data structures for these objects are important. We have stored a user's information in Redis as a HASH. This information includes basic personal information such as "user id", "first name", "last name", "email address", "password", "gender" and "country" where the user lives. The number people who are following a user and the user is following them are also stored as "follower" and "following", respectively. The number of posted tweets and the date that a user has signed up, are stored as "tweets" and "sign up Date". When a new user signs up, a new HASH is created with the following, follower and tweets count set to zero, a new timestamp for the sign up time, and the required personal information. A sample HASH that includes this information for an example user in Redis Desktop Manager is shown in Figure.6.

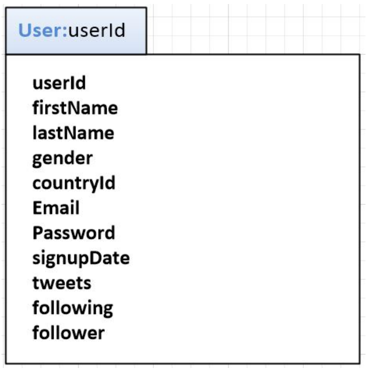
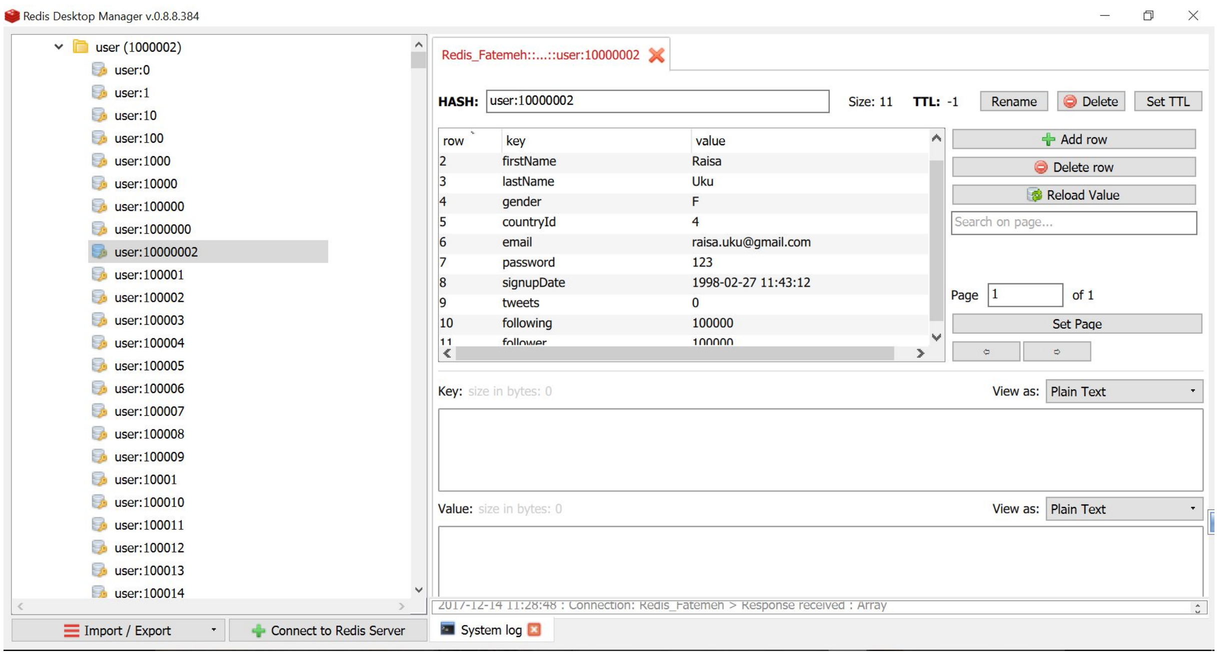


Figure 5: A user HASH structure



Redis Desktop Manager v.0.8.8.384

user (1000002)
- user:0
- user:1
- user:10
- user:100
- user:1000
- user:10000
- user:100000
- user:1000000
- user:10000002
- user:100001
- user:100002
- user:100003
- user:100004
- user:100005
- user:100006
- user:100007
- user:100008
- user:100009
- user:10001
- user:100010
- user:100011
- user:100012
- user:100013
- user:100014

Redis_Fatemeh::...::user:10000002

HASH: user:10000002 Size: 11 TTL: -1

row	key	value
2	firstName	Raisa
3	lastName	Uku
4	gender	F
5	countryId	4
6	email	raisa.uku@gmail.com
7	password	123
8	signupDate	1998-02-27 11:43:12
9	tweets	0
10	following	100000
11	follower	100000

Page 1 of 1

Key: size in bytes: 0 View as: Plain Text

Value: size in bytes: 0 View as: Plain Text

```
2017-12-14 11:28:48 : Connection: Redis_Fatemeh > Response received : Array
```

Figure 6: User HASH in Redis Desktop Manager

4.3.2 Tweets

Tweets are stored inside a HASH. When a tweet is posted, the ID of the user who posted it, text of the tweet and the date of post are stored. Figure.7 and Figure.8 show the structure of HASH and an example tweet, respectively.

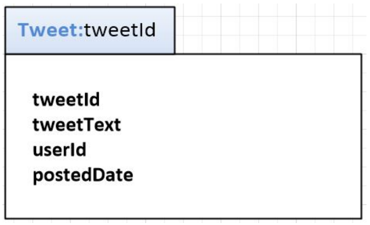
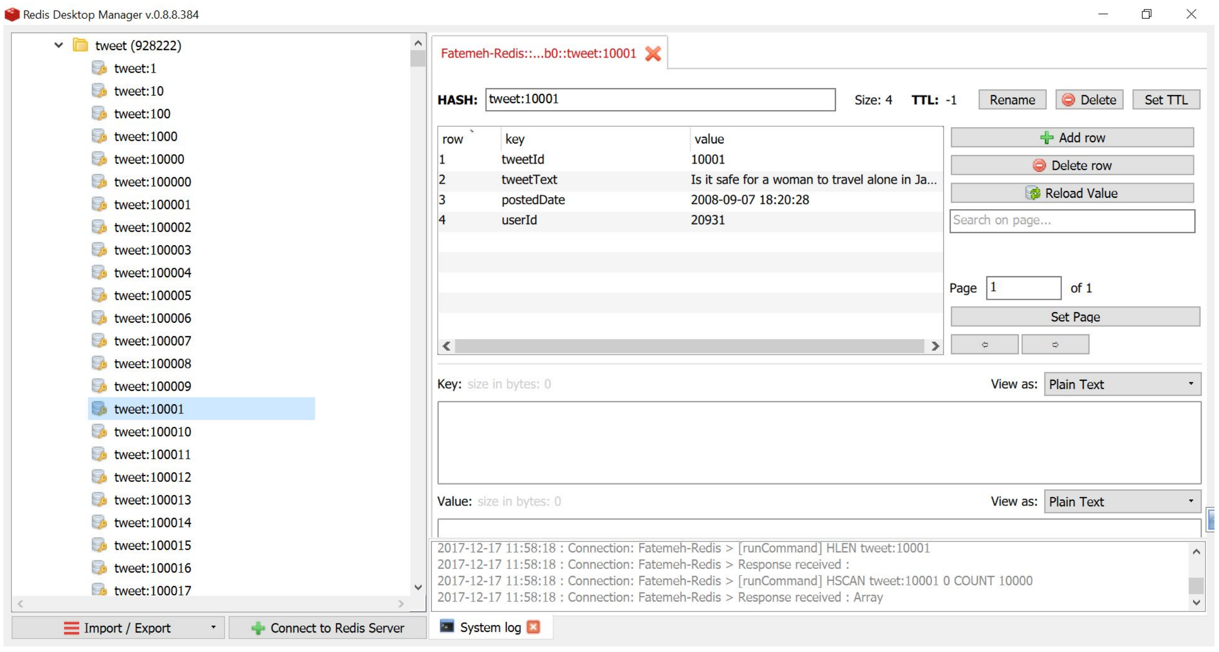


Figure 7: A tweet HASH structure



HASH: tweet:10001 Size: 4 TTL: -1

row	key	value
1	tweetId	10001
2	tweetText	Is it safe for a woman to travel alone in Ja...
3	postedDate	2008-09-07 18:20:28
4	userId	20931

Page 1 of 1

Key: size in bytes: 0 View as: Plain Text

Value: size in bytes: 0 View as: Plain Text

```
2017-12-17 11:58:18 : Connection: Fatemeh-Redis > [runCommand] HLEN tweet:10001
2017-12-17 11:58:18 : Connection: Fatemeh-Redis > Response received :
2017-12-17 11:58:18 : Connection: Fatemeh-Redis > [runCommand] HSCAN tweet:10001 0 COUNT 10000
2017-12-17 11:58:18 : Connection: Fatemeh-Redis > Response received : Array
```

Figure 8: Tweet HASH in Redis Desktop Manager

4.3.3 Followers and Following

In order to manage the lists of followers (users who follow a user) and followings (users whom a user follows), user IDs and following dates are stored in SORTED SET (ZSETs).
When user1 starts following user2, the ID of the user1 is the key, the ID of the user2 is the value and the timestamp is considered as the score in the Followings SORTED SET. On the other hand, the ID of the user2 is the key, the ID of the user1 is the value and the timestamp is the score in the Followers SORTED SET.
In the following figures, the structure of Following and Follower SORTED SETs are shown.

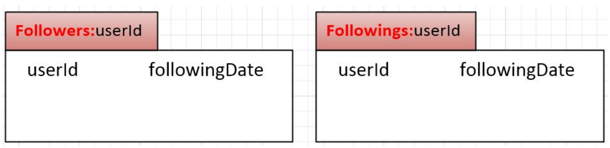
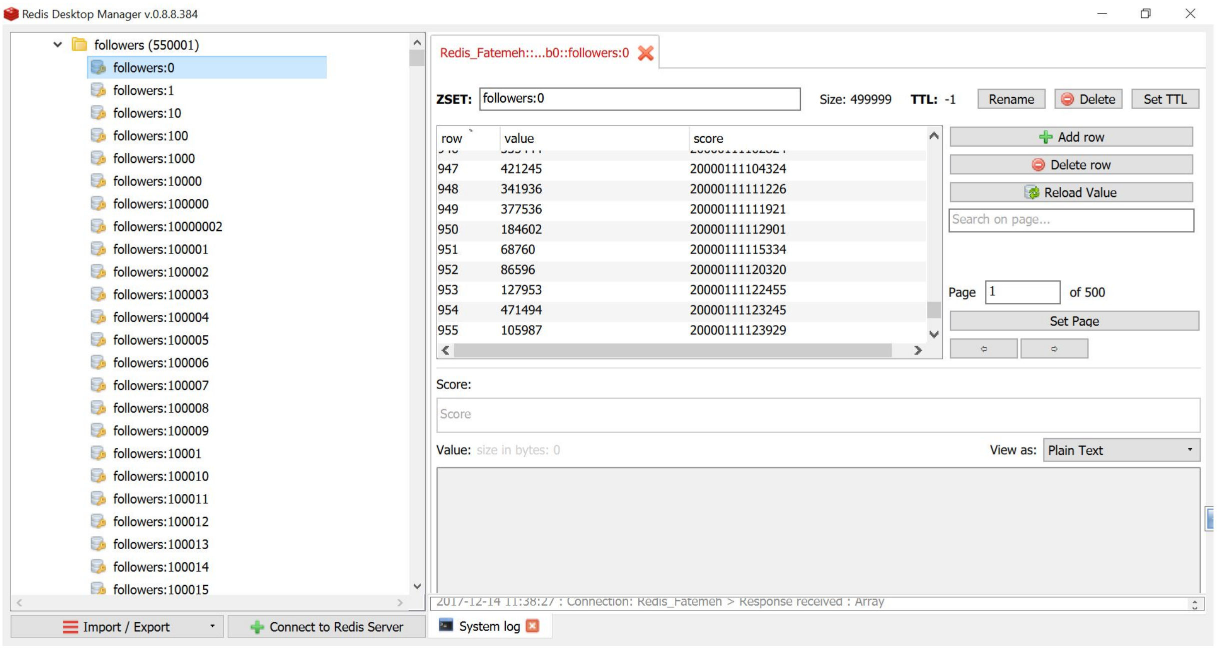


Figure 9: Following and Follower SORTED SETs structure



```
followers (550001)
  followers:0
  followers:1
  followers:10
  followers:100
  followers:1000
  followers:10000
  followers:100000
  followers:10000002
  followers:100001
  followers:100002
  followers:100003
  followers:100004
  followers:100005
  followers:100006
  followers:100007
  followers:100008
  followers:100009
  followers:10001
  followers:100010
  followers:100011
  followers:100012
  followers:100013
  followers:100014
  followers:100015
```

Redis_Fatemeh::...b0::followers:0

ZSET: followers:0   Size: 499999   TTL: -1

row	value	score
947	421245	20000111104324
948	341936	20000111111226
949	377536	20000111111921
950	184602	20000111112901
951	68760	20000111115334
952	86596	20000111120320
953	127953	20000111122455
954	471494	20000111123245
955	105987	20000111123929

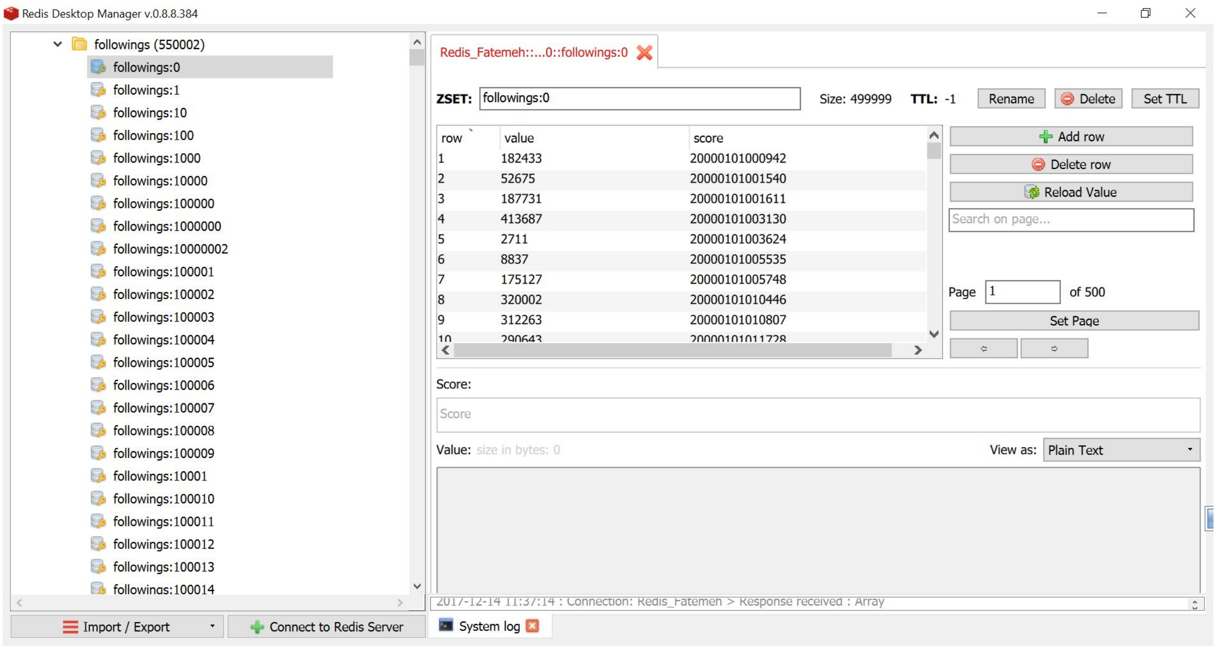
Page 1 of 500

Score:

Value: size in bytes: 0

```
2017-12-14 11:38:27 : Connection: Redis_Fatemeh > Response received : Array
```

Figure 10: Followers SORTED SET in Redis Desktop Manager



```
followings (550002)
  followings:0
  followings:1
  followings:10
  followings:100
  followings:1000
  followings:10000
  followings:100000
  followings:1000000
  followings:10000002
  followings:100001
  followings:100002
  followings:100003
  followings:100004
  followings:100005
  followings:100006
  followings:100007
  followings:100008
  followings:100009
  followings:10001
  followings:100010
  followings:100011
  followings:100012
  followings:100013
  followings:100014
```

Redis_Fatemeh::...0::followings:0

ZSET: followings:0   Size: 499999   TTL: -1

row	value	score
1	182433	20000101000942
2	52675	20000101001540
3	187731	20000101001611
4	413687	20000101003130
5	2711	20000101003624
6	8837	20000101005535
7	175127	20000101005748
8	320002	20000101010446
9	312263	20000101010807
10	290643	20000101011728

Page 1 of 500

Score:

Value: size in bytes: 0

```
2017-12-14 11:37:14 : Connection: Redis_Fatemeh > Response received : Array
```

Figure 11: Followings SORTED SET in Redis Desktop Manager

4.3.4 Timeline (Home)

When a user logs in, he/she sees all his/her tweets and followings' tweets on a page that is called home. For this part, the list of all tweets for a user's home page is stored in a SORTED SET. In this structure, the tweet IDs are the SORTED SET members and the timestamps, when the tweets were posted, are the scores. Figure.12 and Figure.15 show the structure of SORTED SET and an example home, respectively.

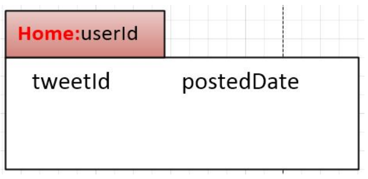
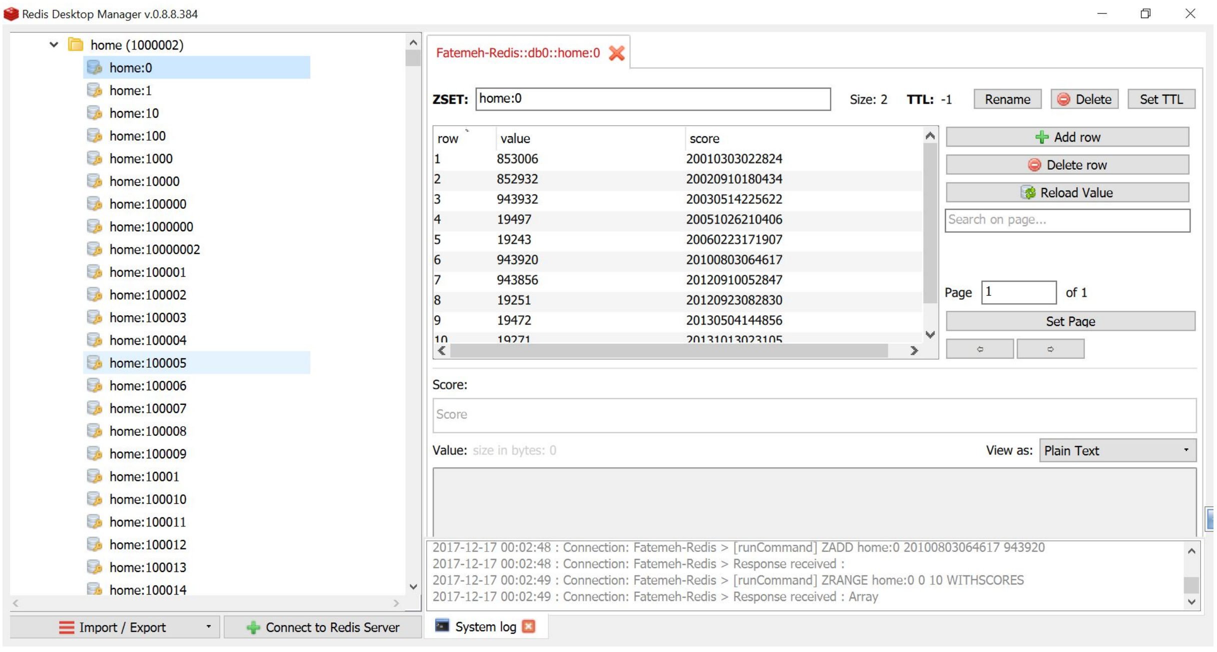


Figure 12: Home SORTED SET structure



ZSET: home:0 Size: 2 TTL: -1

row	value	score
1	853006	20010303022824
2	852932	20020910180434
3	943932	20030514225622
4	19497	20051026210406
5	19243	20060223171907
6	943920	20100803064617
7	943856	20120910052847
8	19251	20120923082830
9	19472	20130504144856
10	19271	20131013023105

Page 1 of 1

Score: Score

Value: size in bytes: 0 View as: Plain Text

```
2017-12-17 00:02:48 : Connection: Fatemeh-Redis > [runCommand] ZADD home:0 20100803064617 943920
2017-12-17 00:02:48 : Connection: Fatemeh-Redis > Response received :
2017-12-17 00:02:49 : Connection: Fatemeh-Redis > [runCommand] ZRANGE home:0 0 10 WITHSCORES
2017-12-17 00:02:49 : Connection: Fatemeh-Redis > Response received : Array
```

Figure 13: Home SORTED SET in Redis Desktop Manager

4.3.5 Tweet Words

When a user posts a tweet, some NLP preprocessing steps are applied to the tweet in order to extract important words. Each word and the tweet ID are stored in "word Set". This is necessary while a user is searching for tweets containing specific words.

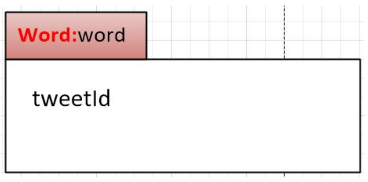
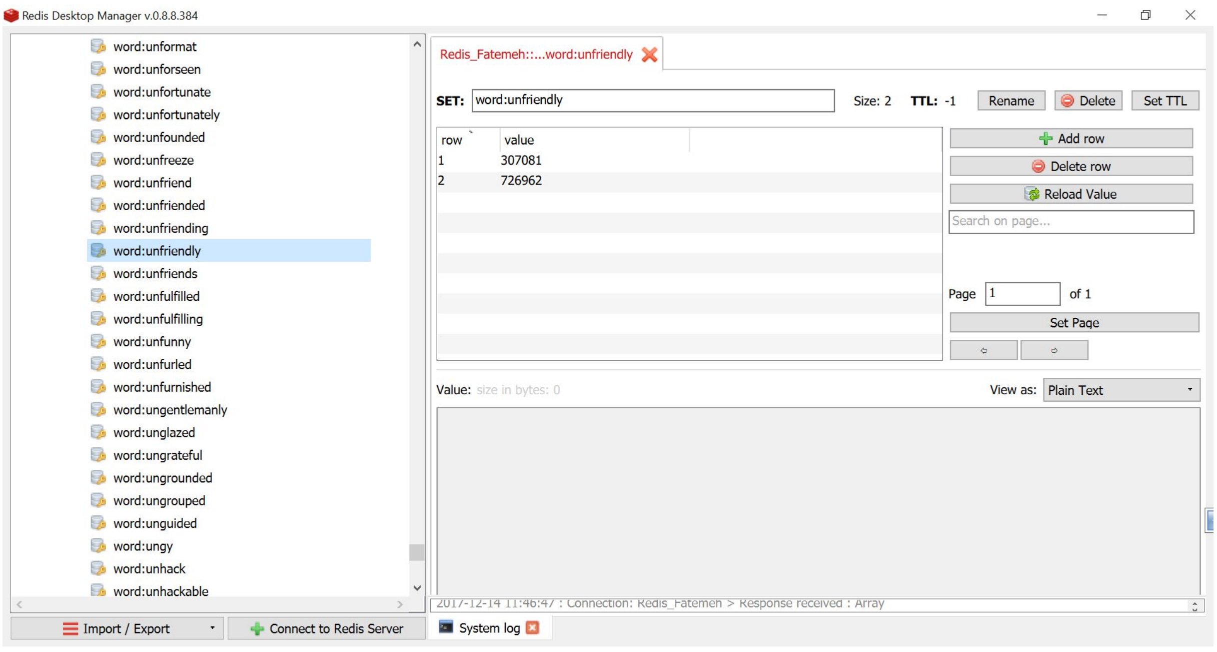


Figure 14: Tweet Word SET structure



Redis Desktop Manager v.0.8.8.384

word:unformat
word:unforseen
word:unfortunate
word:unfortunately
word:unfounded
word:unfreeze
word:unfriend
word:unfriended
word:unfriending
word:unfriendly
word:unfriends
word:unfulfilled
word:unfulfilling
word:unfunny
word:unfurled
word:unfurnished
word:ungentlemanly
word:unglazed
word:ungrateful
word:ungrounded
word:ungrouped
word:unguided
word:ungy
word:unhack
word:unhackable

Redis_Fatemeh::...word:unfriendly

SET: word:unfriendly Size: 2 TTL: -1

row	value
1	307081
2	726962

Page 1 of 1

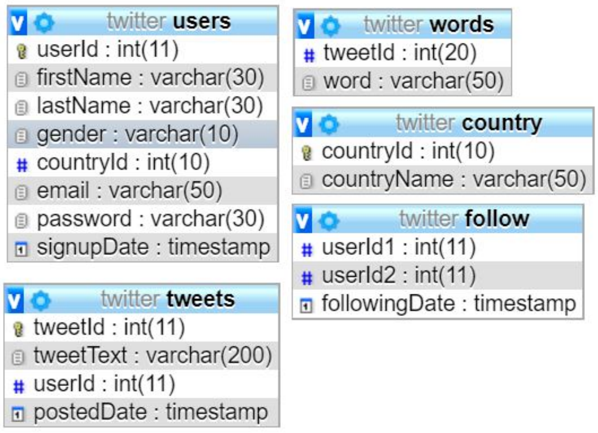
Value: size in bytes: 0 View as: Plain Text

```
2017-12-14 11:46:47 : Connection: Redis_Fatemeh > Response received : Array
```

Figure 15: Tweet Word SET in Redis Desktop Manager

4.4 Tables in MySQL

For the proposed application, five tables have been designed.
users: This table holds the same data as user HASH in Redis, including the personal information of a user.
tweets: This table contains data related to tweets such as tweet text, posted date and the user ID who posted it. It is the same as tweet HASH in Redis.
follow: This table holds the users' IDs as "userId1" and "userId2" which means user1 follows user2. Therefore, all userId2s are the people whom user1 follows them and all userid1s are those who follows user2. In addition, the date of following is stores as "followingDate".
country: In order to run some complex queries, the country of each user is added to the application. This table holds the name of 210 countries.
words: When a tweet is posted, some NLP (Natural Language Processing) preprocessing steps are applied to the tweet text such as "Tokenization" and "Stop Words Removal". The remaining words which contains important meanings and their associated tweet ID are stored in this table. When a user starts searching for a tweet, the application uses this table as an index to retrieve related tweets' IDs and in the next phase, the text of tweet is found in "tweets" table.



twitter users
userId : int(11)
firstName : varchar(30)
lastName : varchar(30)
gender : varchar(10)
countryId : int(10)
email : varchar(50)
password : varchar(30)
signupDate : timestamp

twitter tweets
tweetId : int(11)
tweetText : varchar(200)
userId : int(11)
postedDate : timestamp

twitter words
tweetId : int(20)
word : varchar(50)

twitter country
countryId : int(10)
countryName : varchar(50)

twitter follow
userId1 : int(11)
userId2 : int(11)
followingDate : timestamp

Figure 16: Tables in MySQL

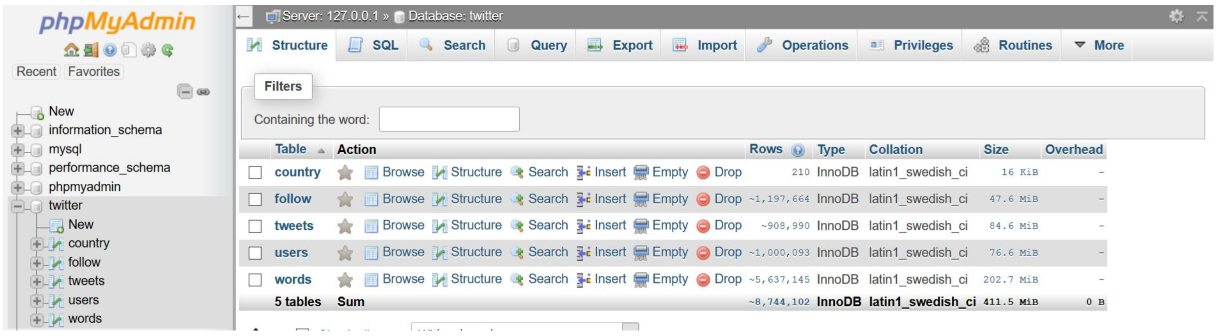


Table	Action	Rows	Type	Collation	Size	Overhead
country	Browse Structure Search Insert Empty Drop	210	InnoDB	latin1_swedish_ci	16 KiB	-
follow	Browse Structure Search Insert Empty Drop	~1,197,664	InnoDB	latin1_swedish_ci	47.6 MiB	-
tweets	Browse Structure Search Insert Empty Drop	~908,990	InnoDB	latin1_swedish_ci	84.6 MiB	-
users	Browse Structure Search Insert Empty Drop	~1,000,093	InnoDB	latin1_swedish_ci	76.6 MiB	-
words	Browse Structure Search Insert Empty Drop	~5,637,145	InnoDB	latin1_swedish_ci	202.7 MiB	-
5 tables	Sum	~8,744,102	InnoDB	latin1_swedish_ci	411.5 MiB	0 B

Figure 17: MySQL Database

4.5 Experiment and Queries

In this section, we will review all the queries defined in the Twitter-clone application in addition to some complex queries to compare the performance of Redis and MySQL. We ran each query for **FIVE TIMES** to get the average time (the first time is omitted from the calculations). According to the requirements of each query, We ran all the queries for **1**, **1000**, **10000**, **100000** or **1000000 users**.

4.5.1 Query 1: Registration

MySQL: When a new user wants to register, he/she enters personal information. First, the country ID is retrieved from "country" table. Second, a new row in "users" table is created and all data is inserted.
Redis: A unique user ID is assigned to the user. Then, a new "user HASH" is created. The key of this HASH is the user ID. The number of followings, followers and tweets are set to zero.
The average time is shown in the following table.

Number of Users	MySQL	Redis
1	0.0085	0.0015
100	0.2526	0.0489
1000	1.9012	0.5723
10000	3.192	1.4219

Table 10: Registration - Time Results (second)

4.5.2 Query 2: Post a Tweet

When a user logs in, he/she is directed to his/her home page where the user can post tweets.
MySQL: When a tweet is posted, a new row in "tweets" table is created, holding the userId that posted the tweet. In addition the important words of the tweets are extracted and inserted into "words" table.
Redis: A new "tweet HASH" is created in Redis. The number of tweets for the user is updated in "user HASH". The tweet ID is inserted in to "home Sorted Set" of the user and all user's followers' "home SORTED SET". Finally, the important words of the tweets are extracted and inserted into "words Set".

Number of Tweets	MySQL	Redis
1	0.0443	0.0074
100	1.3201	0.0472
1000	13.6749	0.3974
10000	23.0933	3.2217

Table 11: Post a Tweet - Time Results (second)

4.5.3 Query 3: Retrieve the Followers for a user

When a user logs in, a list of his/her followings is retrieved.
MySQL: The followers' IDs and their names are obtained from "follow" and "users" tables.
Redis: Retrieve the required data from "followers Sorted Set" and "user HASH".

Number of Followers	MySQL	Redis
1	0.9455	0.000107
100000	4.0059	0.000302
1000000	4.5887	0.000521

Table 12: Retrieve Followers - Time Results (second)

4.5.4 Query 4: Retrieve the Followings for a user

Similar to previous section, a list of user's followings is retrieved When a user logs in.
MySQL: The followings' IDs and their names are retrieved from "follow" and "users" tables.
Redis: The required information is obtained from "followings Sorted Set" and "user HASH".

Number of Followings	MySQL	Redis
1	0.9487	0.000138
100000	3.9943	0.000561
1000000	4.3693	0.000455

Table 13: Retrieve Followings - Time Results (second)

4.5.5 Query 5: Follow a user

When a user visits other users' pages, he/she can follow or unfollow them. In this query, we consider to follow a user.
MySQL: When user1 starts following user2, a new row is inserted in "follow" table.
Redis: In Redis, "user HASH" for both users (to update the number of followings for user1 and the number of followers for user2), "followers Sorted Set" for user2, "followings Sorted Set" for user1, and "home Sorted Set" for user1 (to add user2's tweets to user1's home page) are updated.

Number of Users to Follow	MySQL	Redis
1	0.0012	0.000413
1000	1.6431	0.1476
10000	15.0036	1.6697

Table 14: Follow a user - Time Results (second)

4.5.6 Query 6: Unfollow a user

MySQL: When user1 decides to unfollow user2, the associated row in “follow” table is deleted.
Redis: Similar to previous query, “followers Sorted Set” for user2, “followings Sorted Set” for user1, “user HASH” for both users and “home Sorted Set” for user1 (to delete user2’s tweets from user1’s home page) are updated.

Number of Users to Unfollow	MySQL	Redis
1	0.0652	0.000679
1000	0.0619	0.1329
10000	0.5631	1.4668

Table 15: Unfollow a user - Time Results (second)

4.5.7 Query 7: Retrieve all Users

MySQL: In this query, we retrieve all user’s first names, last names and user IDs from “users” table.
Redis: To retrieve each user’s personal information, all “user HASH” were scanned.

Number of Users	MySQL	Redis
1000000	0.5964	0.000108

Table 16: Retrieve all Users - Time Results (second)

4.5.8 Query 8: Retrieve all tweets for a user

In order to perform this query, we created a user with 900000 tweets.
MySQL: These tweets are retrieved from “tweets” and “users” tables.
Redis: Similar to previous query, all “tweet HASH” were scanned and the users’ names retrieved from “user HASH”

Number of Tweets	MySQL	Redis
900000	3.8453	0.000126

Table 17: Retrieve all tweets for a user - Time Results (second)

4.5.9 Query 9: Search tweets

In this query, we have considered that a user searches for tweets containing a keyword.
MySQL: We used the “Full Text Search” property in MySQL.
Redis: We have also created an index for the list of words. Therefore, we defined “word SET” that each word is a key and the tweet IDs containing that word are the values. The tweet texts are retrieved from “tweet HASH”.
As it is shown in the table, MySQL can be as fast as Redis when it uses “Full Text Search” property.

Number of Tweets	MySQL (Full Text Search)	Redis
928222	0.000297	0.000275

Table 18: Retrieve all tweets containing a keyword - Time Results (second)

4.5.10 Query 10: Search users

In this query, a user can look for a specific person.
MySQL: We did not use “Text Search” property in MySQL.
Redis: We have used a SET as an index for “user HASH”. This index uses the names of users as key and their IDs as values.

Number of Users	MySQL	Redis
1000000	0.3551	0.000258

Table 19: Retrieve all tweets containing a keyword - Time Results (second)

CONCLUSION for QUERY 9 AND 10 : It can be concluded that MySQL can perform as well as Redis when its“Text Search” is used.

4.5.11 Query 11

Find a user’s followers who started following the user after his/her (user’s) first tweet.

MySQL: For this query three tables,“users”, “follow” and “tweets” are joined to extract the target followers for the defined user.
Redis: In the first step, we examine that the user have posted any tweets. If he/she has posted tweets, the posted date of his/her first tweet is retrieved from “tweet HASH”. Then, the required followers’ IDs are obtained from “followers Sorted Set”. Finally their names are extracted from the associated “user HASH”.

MySQL	Redis
2.7501	0.001175

Table 20: Query 11 - Time Results (second)

4.5.12 Query 12

Find tweets including a word that are posted from 2003 to 2010 by users who joined twitter in 2000.

MySQL: To answer this query three tables,“users”, “tweets” and “words” are joined to find the required tweets.
Redis: First, all the tweet IDs that contains the keyword are retrieved. Then, the related “tweet HASH” for each tweet is examined to check whether it was posted during the specified period. Finally, the user IDs and the names who posted the tweets are found.

MySQL	Redis
3.3909	0.00544

Table 21: Query 12 - Time Results (second)

4.5.13 Query 13

Modify the male users’ countries to Belgium who has more than 10 followers and 10 followings. (While running this query, 500000 records among 1000000 records were updated.)

MySQL: For this query three tables,“users”, “country” and “follow” are joined to update the target users’ countries.
Redis: First, all male users are extracted from “user HASH”. Then, the number of followins and followers for each user is examined. Finally, the country value in “user HASH” is changed to Belgium for the candidate users.

MySQL	Redis
26.7254	3.8437

Table 22: Query 13 - Time Results (second)

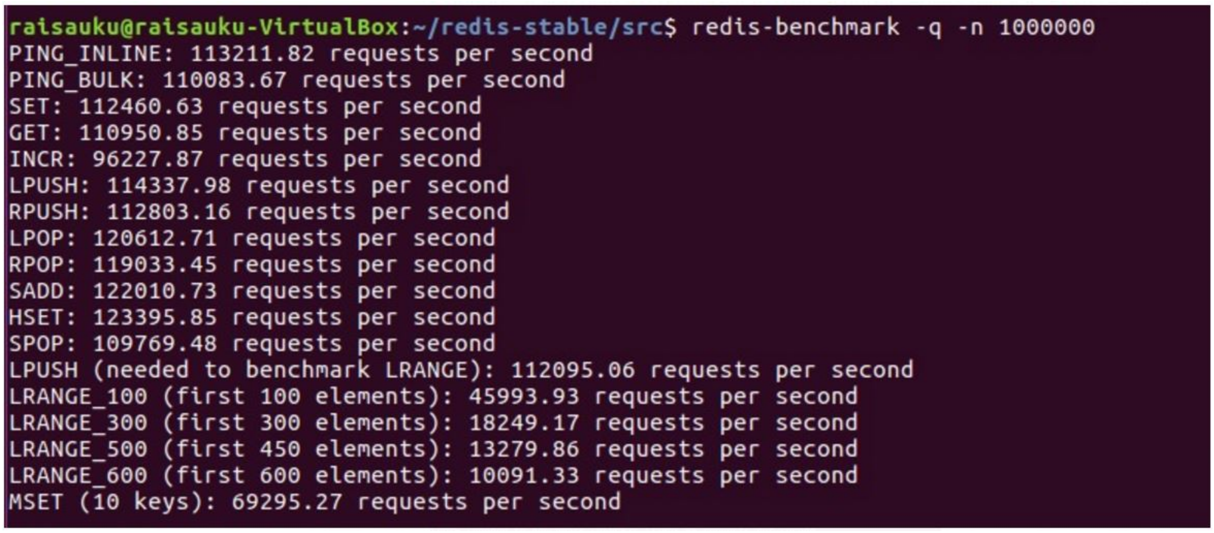
5 Redis-Benchmark Utility

Redis includes the redis-benchmark utility that simulates running commands done by N clients at the same time sending M total queries. Remember that Redis-benchmark utility is a quick and useful way to evaluate the performance of a Redis instance on a given hardware(*How fast is Redis?*, 2017).
The basic syntax of Redis benchmark is as follows:
redis-benchmark [option] [option value]
In table.23 a list of available options in Redis benchmark is provided

Option	Description	Default Value
-h	Specifies server host name	127.0.0.1
-p	Specifies server port	6379
-s	Specifies server socket	
-a	Specifies password for Redis	
-c	Specifies the number of parallel connections	50
-n	Specifies the total number of requests	10000
-d	Specifies data size of SET/GET value in bytes	2
—**dbnum**	SELECT the specified db number	0
-k	1=keep alive, 0=reconnect	1
-r	Use random keys for SET/GET/INCR, random values for SADD	
-p	Pipeline ¡numreq¿ requests	1 (no pipeline)
-q	Forces Quiet to Redis. Just shows query/sec values	
–**csv**	Output in CSV format	
-l	Generates loop, Run the tests forever	
-t	Only runs the comma-separated list of tests	
-I	Idle mode. Just opens N idle connections and wait	

Table 23: Redis benchmark options

In the following figure, we utilized this utility to check our machine configuration by calling 1000000 commands.



```
raisauku@raisauku-VirtualBox:~/redis-stable/src$ redis-benchmark -q -n 1000000
PING_INLINE: 113211.82 requests per second
PING_BULK: 110083.67 requests per second
SET: 112460.63 requests per second
GET: 110950.85 requests per second
INCR: 96227.87 requests per second
LPUSH: 114337.98 requests per second
RPUSH: 112803.16 requests per second
LPOP: 120612.71 requests per second
RPOP: 119033.45 requests per second
SADD: 122010.73 requests per second
HSET: 123395.85 requests per second
SPOP: 109769.48 requests per second
LPUSH (needed to benchmark LRANGE): 112095.06 requests per second
LRANGE_100 (first 100 elements): 45993.93 requests per second
LRANGE_300 (first 300 elements): 18249.17 requests per second
LRANGE_500 (first 450 elements): 13279.86 requests per second
LRANGE_600 (first 600 elements): 10091.33 requests per second
MSET (10 keys): 69295.27 requests per second
```

Figure 18: Redis-Benchmark Utility results for 1000000 commands

6 Conclusion

This report describes in detail Redis NoSQL database, which is considered as in-memory, key-value datastore. By storing the data in the primary memory which makes it fast in both writing and reading, by providing five advanced data structures (accompanied with a set of operations) which differentiate it from other key-value databases and by being embraced by various well known companies, Redis is the most popular key-value database today.

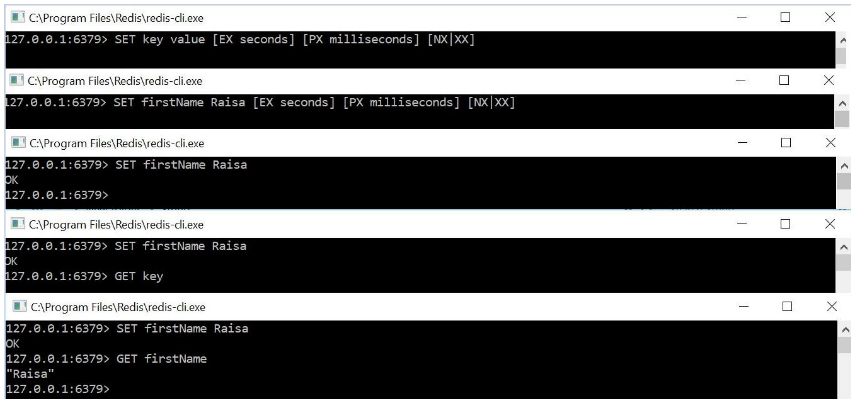
Furthermore, we have developed a simple version of a social media application, to evaluate Redis performance against MySQL relational database, as it is ranked the second popular database by DB Engine. As expected, the results of benchmarking prove that Redis has better performance in use cases where processing and analyzing high-velocity data is very important.

Finally, it is important to emphasis that Redis perform well for specific scenarios. Therefore, it is mostly used alongside a relational database or another NoSQL database to improve the overall performance of the application.

7 Appendix: Redis Commands in Redis-cli and PHP-Redis

Redis-cli is the Redis command line interface, a simple program that allows to send commands to Redis, and read the replies sent by the server, directly from the terminal.
The **PHP-Redis** extension provides an API for communicating with the Redis key-value store in PHP.

String: In the following example, SET and GET are Redis commands, “firstName” is the “key” used in Redis and “Raisa” is the string “value” that is stored in Redis. Remember that a string value can be 512 megabytes long.



```
127.0.0.1:6379> SET key value [EX seconds] [PX milliseconds] [NX|XX]
```

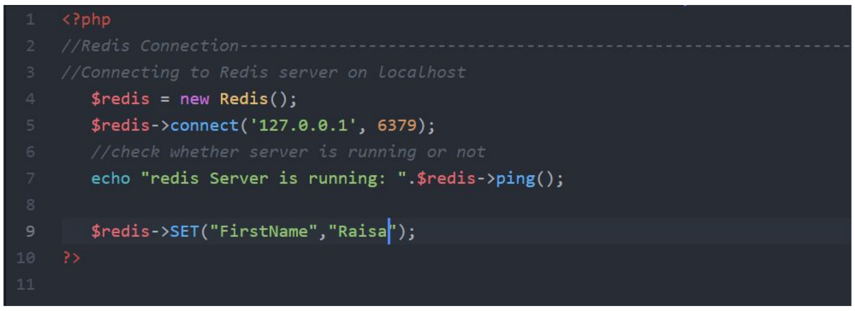
```
127.0.0.1:6379> SET firstName Raisa [EX seconds] [PX milliseconds] [NX|XX]
```

```
127.0.0.1:6379> SET firstName Raisa
OK
127.0.0.1:6379>
```

```
127.0.0.1:6379> SET firstName Raisa
OK
127.0.0.1:6379> GET key
```

```
127.0.0.1:6379> SET firstName Raisa
OK
127.0.0.1:6379> GET firstName
"Raisa"
127.0.0.1:6379>
```

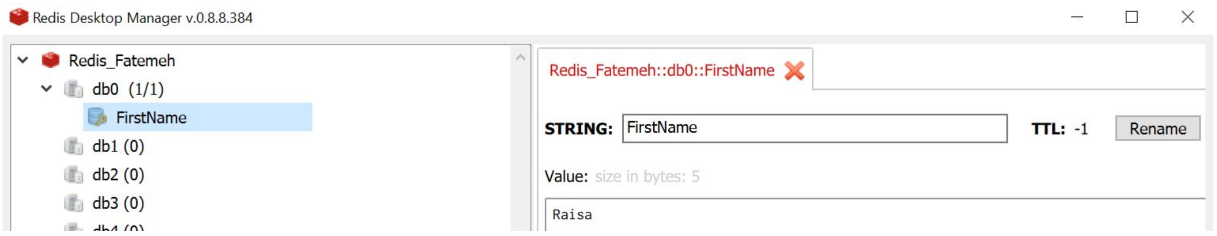
Figure 19: SET and GET commands for String Data type in Redis-cli



```php
<?php
//Redis Connection-----------------------------------------------------------
//Connecting to Redis server on localhost
   $redis = new Redis();
   $redis->connect('127.0.0.1', 6379);
   //check whether server is running or not
   echo "redis Server is running: ".$redis->ping();

   $redis->SET("FirstName","Raisa");
?>
```

Figure 20: SET and GET commands for String Data type in PHP-Redis


Redis Desktop Manager v.0.8.8.384
Redis_Fatemeh
db0 (1/1)
FirstName
db1 (0)
db2 (0)
db3 (0)
Redis_Fatemeh::db0::FirstName
Rename

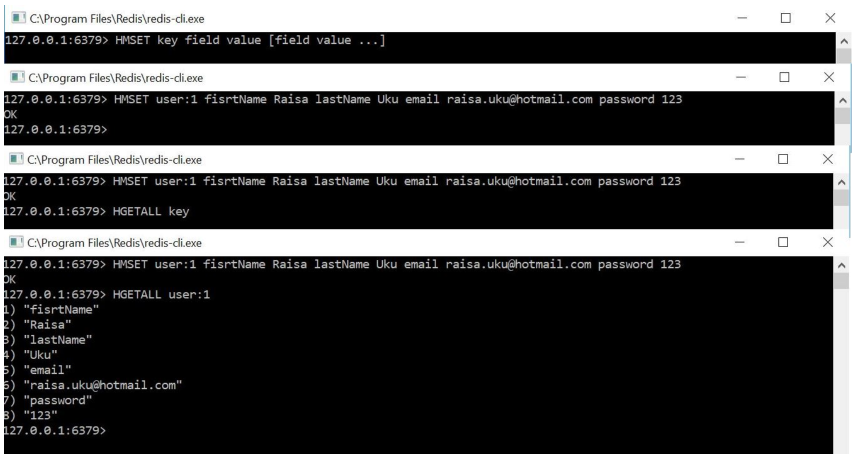
STRING: FirstName TTL: -1

Value: size in bytes: 5

Raisa

Figure 21: String Data type in Redis Desktop Manager

Hash: In this example, hash data type is used to store the basic information of a user. HMSET, HGETALL, HSET, HGET are commands for Redis and “user:1” is the “key”. Every hash can store up to $2^{32} - 1$ field-value pairs (more than 4 billion).



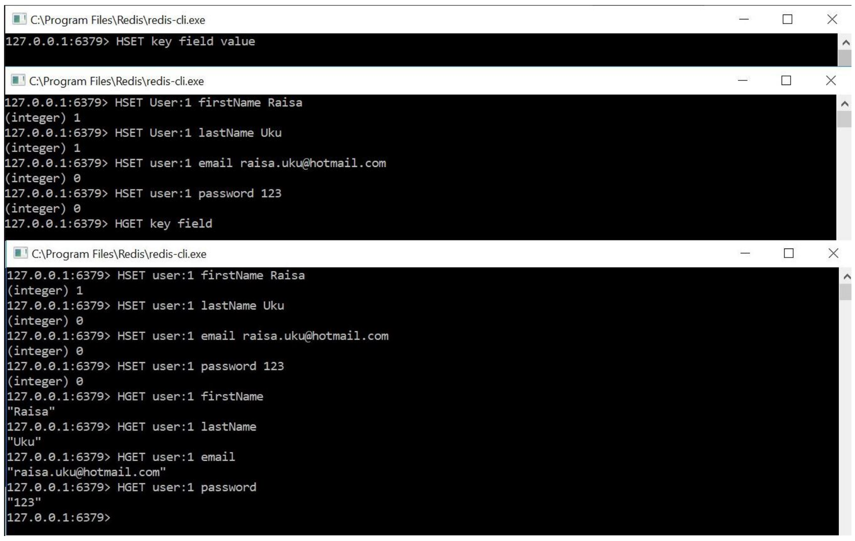
```
C:\Program Files\Redis\redis-cli.exe
127.0.0.1:6379> HMSET key field value [field value ...]

C:\Program Files\Redis\redis-cli.exe
127.0.0.1:6379> HMSET user:1 fisrtName Raisa lastName Uku email raisa.uku@hotmail.com password 123
OK
127.0.0.1:6379>

C:\Program Files\Redis\redis-cli.exe
127.0.0.1:6379> HMSET user:1 fisrtName Raisa lastName Uku email raisa.uku@hotmail.com password 123
DK
127.0.0.1:6379> HGETALL key

C:\Program Files\Redis\redis-cli.exe
127.0.0.1:6379> HMSET user:1 fisrtName Raisa lastName Uku email raisa.uku@hotmail.com password 123
DK
127.0.0.1:6379> HGETALL user:1
1) "fisrtName"
2) "Raisa"
B) "lastName"
4) "Uku"
5) "email"
5) "raisa.uku@hotmail.com"
7) "password"
B) "123"
127.0.0.1:6379>
```

Figure 22: HMSET and HGETALL commands for Hash Data type in Redis-cli

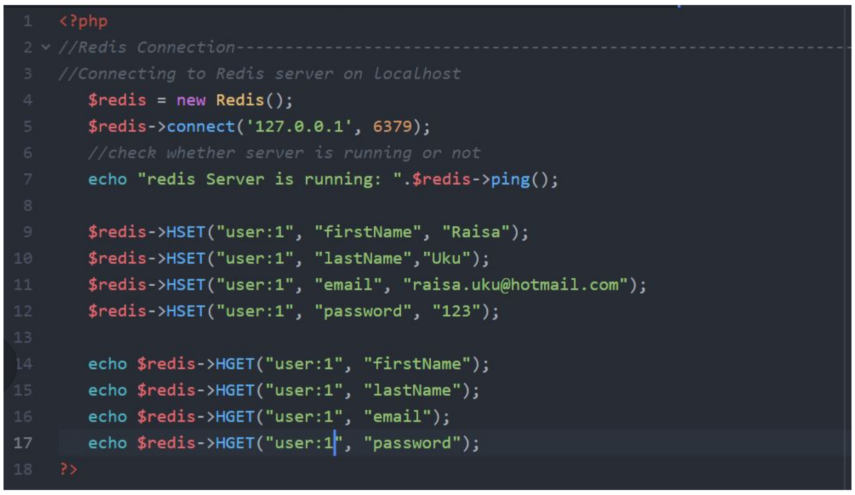


```
C:\Program Files\Redis\redis-cli.exe
127.0.0.1:6379> HSET key field value

C:\Program Files\Redis\redis-cli.exe
127.0.0.1:6379> HSET User:1 firstName Raisa
(integer) 1
127.0.0.1:6379> HSET User:1 lastName Uku
(integer) 1
127.0.0.1:6379> HSET user:1 email raisa.uku@hotmail.com
(integer) 0
127.0.0.1:6379> HSET user:1 password 123
(integer) 0
127.0.0.1:6379> HGET key field

C:\Program Files\Redis\redis-cli.exe
127.0.0.1:6379> HSET user:1 firstName Raisa
(integer) 1
127.0.0.1:6379> HSET user:1 lastName Uku
(integer) 0
127.0.0.1:6379> HSET user:1 email raisa.uku@hotmail.com
(integer) 0
127.0.0.1:6379> HSET user:1 password 123
(integer) 0
127.0.0.1:6379> HGET user:1 firstName
"Raisa"
127.0.0.1:6379> HGET user:1 lastName
"Uku"
127.0.0.1:6379> HGET user:1 email
"raisa.uku@hotmail.com"
127.0.0.1:6379> HGET user:1 password
"123"
127.0.0.1:6379>
```

Figure 23: HSET and HGET commands for Hash Data type in Redis-cli

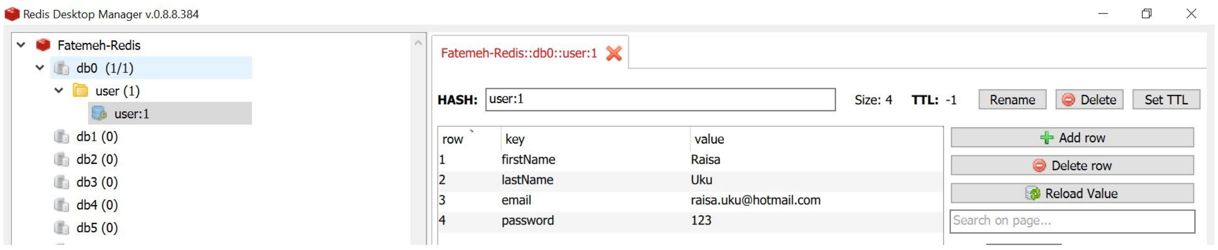


```php
<?php
//Redis Connection------------------------------------------------------------
//Connecting to Redis server on localhost
   $redis = new Redis();
   $redis->connect('127.0.0.1', 6379);
   //check whether server is running or not
   echo "redis Server is running: ".$redis->ping();

   $redis->HSET("user:1", "firstName", "Raisa");
   $redis->HSET("user:1", "lastName","Uku");
   $redis->HSET("user:1", "email", "raisa.uku@hotmail.com");
   $redis->HSET("user:1", "password", "123");

   echo $redis->HGET("user:1", "firstName");
   echo $redis->HGET("user:1", "lastName");
   echo $redis->HGET("user:1", "email");
   echo $redis->HGET("user:1", "password");
?>
```

Figure 24: HSET and HGET commands for Hash Data type in PHP-Redis

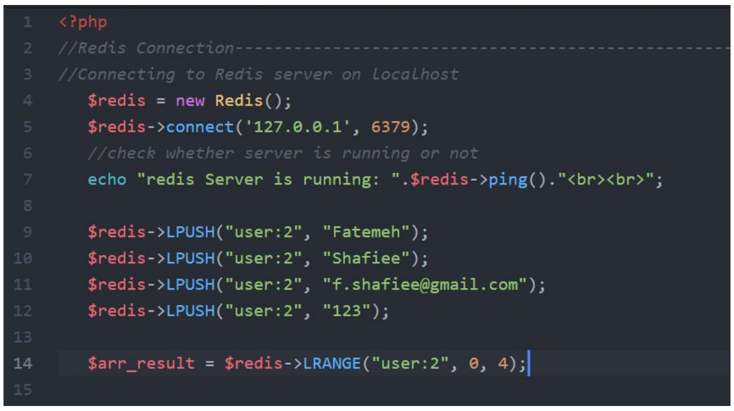


HASH: user:1 Size: 4 TTL: -1

row	key	value
1	firstName	Raisa
2	lastName	Uku
3	email	raisa.uku@hotmail.com
4	password	123

Figure 25: Hash Data type in Redis Desktop Manager

List: In Redis, Lists are lists of strings which are sorted by insertion order. Elements can be added to a List from left or right.The max length of a list is $2^{32} - 1$ elements (more than 4 billion of elements in a list).

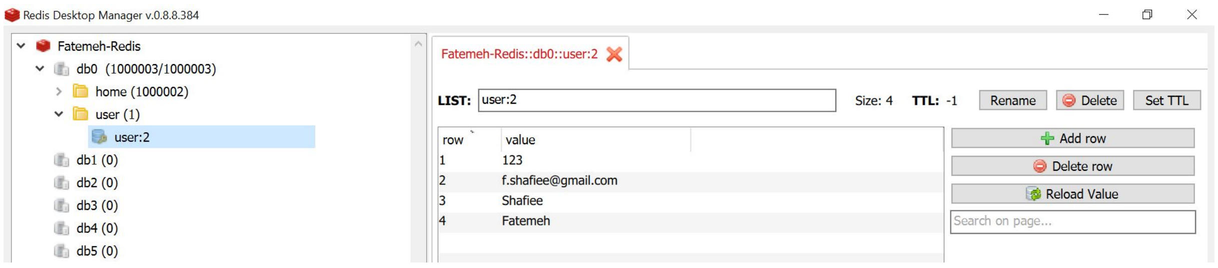


```php
<?php
//Redis Connection--------------------------------------------------------
//Connecting to Redis server on localhost
   $redis = new Redis();
   $redis->connect('127.0.0.1', 6379);
   //check whether server is running or not
   echo "redis Server is running: ".$redis->ping()."<br><br>";

   $redis->LPUSH("user:2", "Fatemeh");
   $redis->LPUSH("user:2", "Shafiee");
   $redis->LPUSH("user:2", "f.shafiee@gmail.com");
   $redis->LPUSH("user:2", "123");

   $arr_result = $redis->LRANGE("user:2", 0, 4);
```

Figure 26: LPUSH and LRANGE commands for List Data type in PHP-Redis


Redis Desktop Manager v.0.8.8.384

- Fatemeh-Redis
  - db0 (1000003/1000003)
    - home (1000002)
    - user (1)
      - user:2
  - db1 (0)
  - db2 (0)
  - db3 (0)
  - db4 (0)
  - db5 (0)

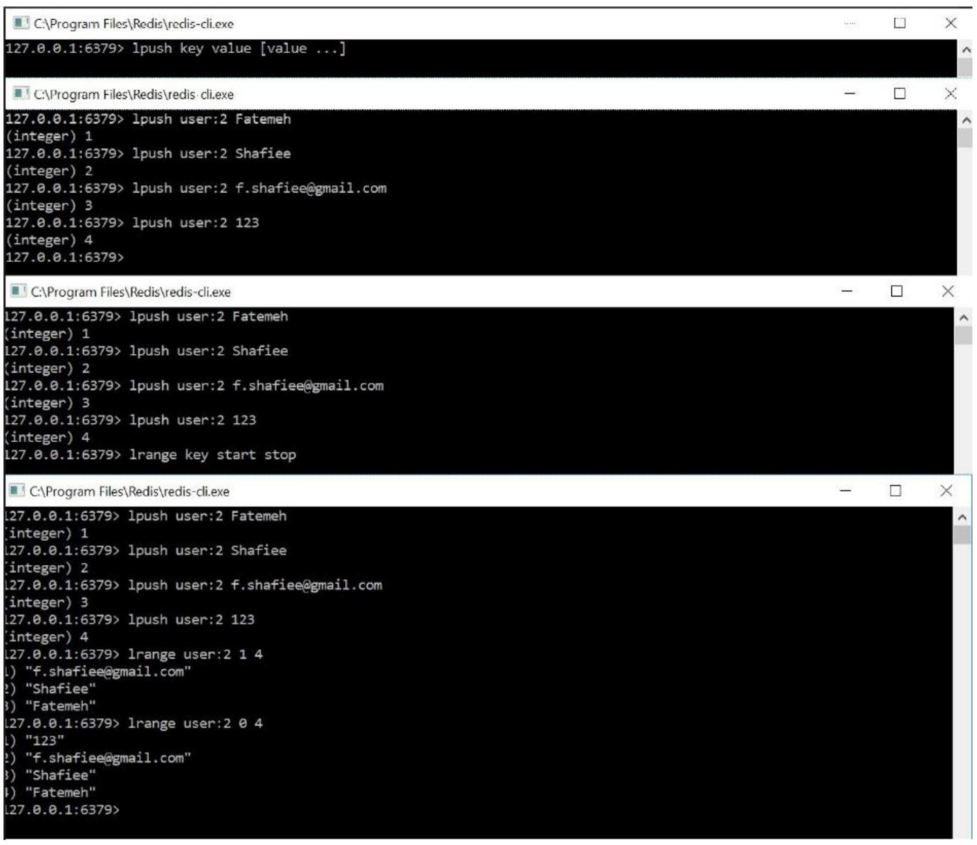
Fatemeh-Redis::db0::user:2

LIST: user:2 Size: 4 **TTL:** -1 Rename Delete Set TTL

row	value
1	123
2	f.shafiee@gmail.com
3	Shafiee
4	Fatemeh

Add row
Delete row
Reload Value
Search on page...

Figure 27: List Data type in Redis Desktop Manager


C:\Program Files\Redis\redis-cli.exe

```
127.0.0.1:6379> lpush key value [value ...]
```

C:\Program Files\Redis\redis-cli.exe

```
127.0.0.1:6379> lpush user:2 Fatemeh
(integer) 1
127.0.0.1:6379> lpush user:2 Shafiee
(integer) 2
127.0.0.1:6379> lpush user:2 f.shafiee@gmail.com
(integer) 3
127.0.0.1:6379> lpush user:2 123
(integer) 4
127.0.0.1:6379>
```

C:\Program Files\Redis\redis-cli.exe

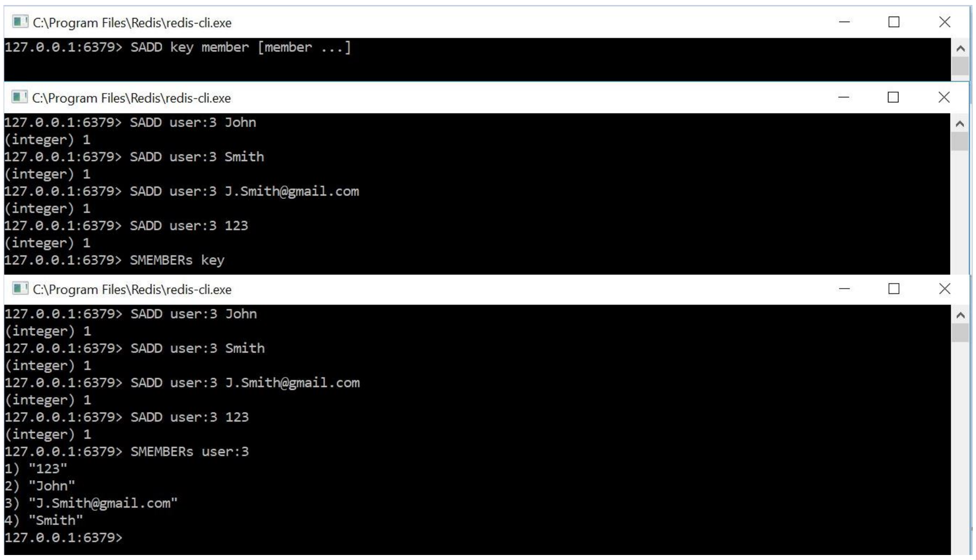
```
127.0.0.1:6379> lpush user:2 Fatemeh
(integer) 1
127.0.0.1:6379> lpush user:2 Shafiee
(integer) 2
127.0.0.1:6379> lpush user:2 f.shafiee@gmail.com
(integer) 3
127.0.0.1:6379> lpush user:2 123
(integer) 4
127.0.0.1:6379> lrange key start stop
```

C:\Program Files\Redis\redis-cli.exe

```
127.0.0.1:6379> lpush user:2 Fatemeh
(integer) 1
127.0.0.1:6379> lpush user:2 Shafiee
(integer) 2
127.0.0.1:6379> lpush user:2 f.shafiee@gmail.com
(integer) 3
127.0.0.1:6379> lpush user:2 123
(integer) 4
127.0.0.1:6379> lrange user:2 1 4
1) "f.shafiee@gmail.com"
2) "Shafiee"
3) "Fatemeh"
127.0.0.1:6379> lrange user:2 0 4
1) "123"
2) "f.shafiee@gmail.com"
3) "Shafiee"
4) "Fatemeh"
127.0.0.1:6379>
```

Figure 28: LPUSH and LRANGE commands for List Data type in Redis-cli

Set: Sets are a collection of unordered strings. Therefore adding, removing, and checking for the existence of a string in the set can be done in O(1) time complexity. When a string is inserted in to set twice, it is added only once. The max number of members in a set is 2̂32 - 1 element (more than 4 billion of members in each set).


C:\Program Files\Redis\redis-cli.exe

```
127.0.0.1:6379> SADD key member [member ...]
```

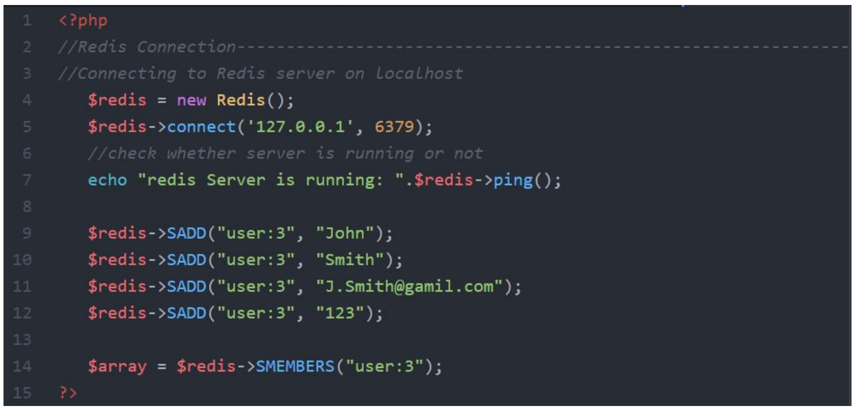
C:\Program Files\Redis\redis-cli.exe

```
127.0.0.1:6379> SADD user:3 John
(integer) 1
127.0.0.1:6379> SADD user:3 Smith
(integer) 1
127.0.0.1:6379> SADD user:3 J.Smith@gmail.com
(integer) 1
127.0.0.1:6379> SADD user:3 123
(integer) 1
127.0.0.1:6379> SMEMBERs key
```

C:\Program Files\Redis\redis-cli.exe

```
127.0.0.1:6379> SADD user:3 John
(integer) 1
127.0.0.1:6379> SADD user:3 Smith
(integer) 1
127.0.0.1:6379> SADD user:3 J.Smith@gmail.com
(integer) 1
127.0.0.1:6379> SADD user:3 123
(integer) 1
127.0.0.1:6379> SMEMBERs user:3
1) "123"
2) "John"
3) "J.Smith@gmail.com"
4) "Smith"
127.0.0.1:6379>
```

Figure 29: SADD and SMEMBERS commands for Set Data type in Redis-cli

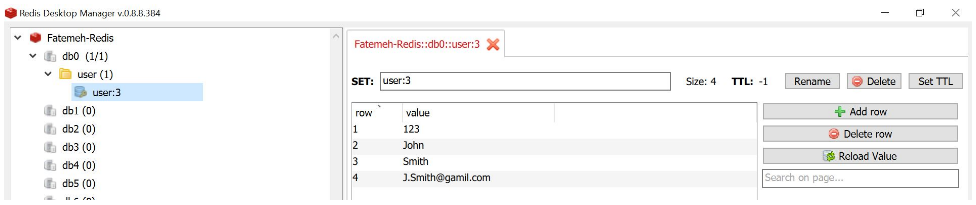


```php
<?php
//Redis Connection--------------------------------------------------------------
//Connecting to Redis server on localhost
   $redis = new Redis();
   $redis->connect('127.0.0.1', 6379);
   //check whether server is running or not
   echo "redis Server is running: ".$redis->ping();

   $redis->SADD("user:3", "John");
   $redis->SADD("user:3", "Smith");
   $redis->SADD("user:3", "J.Smith@gamil.com");
   $redis->SADD("user:3", "123");

   $array = $redis->SMEMBERS("user:3");
?>
```

Figure 30: SADD and SMEMBERS commands for Set Data type in PHP-Redis

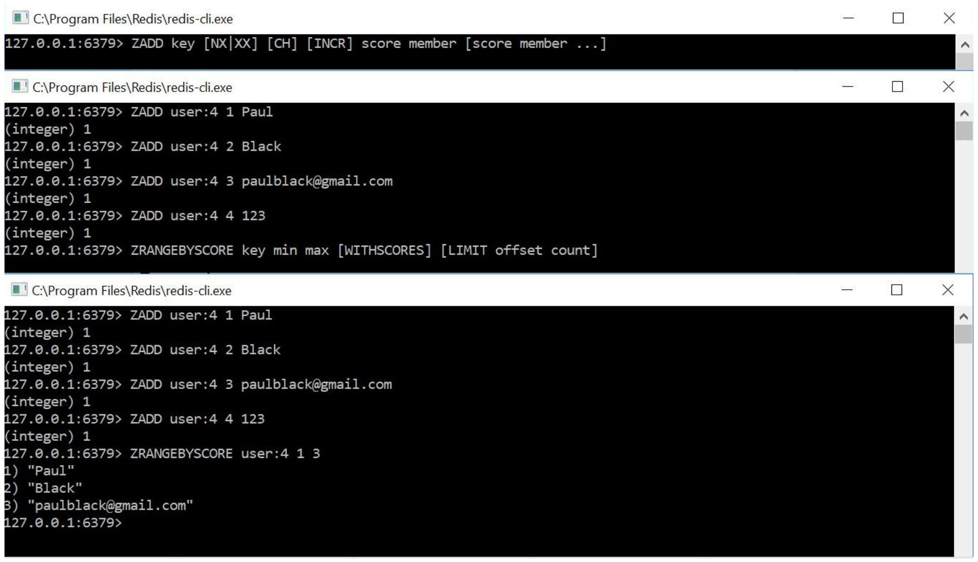


SET: user:3   Size: 4   TTL: -1

row	value
1	123
2	John
3	Smith
4	J.Smith@gamil.com

Figure 31: Set Data type in Redis Desktop Manager

Sorted Sets: Sorted Sets are similar to Sets, but every member of a Sorted Set is associated with a score. These scores are used to order the elements from the smallest to the greatest score. The scores can be repeated.

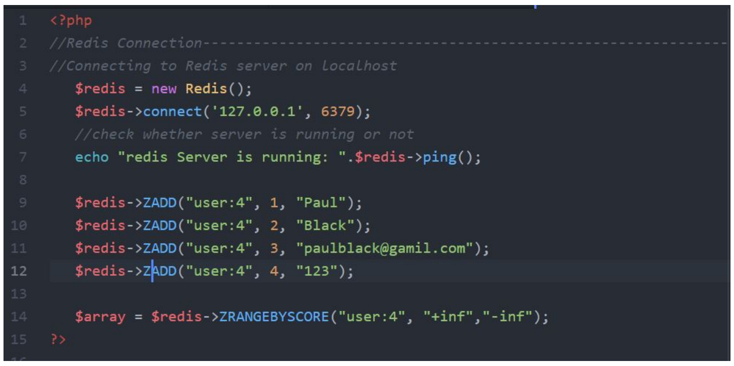


```
127.0.0.1:6379> ZADD key [NX|XX] [CH] [INCR] score member [score member ...]
```

```
127.0.0.1:6379> ZADD user:4 1 Paul
(integer) 1
127.0.0.1:6379> ZADD user:4 2 Black
(integer) 1
127.0.0.1:6379> ZADD user:4 3 paulblack@gmail.com
(integer) 1
127.0.0.1:6379> ZADD user:4 4 123
(integer) 1
127.0.0.1:6379> ZRANGEBYSCORE key min max [WITHSCORES] [LIMIT offset count]
```

```
127.0.0.1:6379> ZADD user:4 1 Paul
(integer) 1
127.0.0.1:6379> ZADD user:4 2 Black
(integer) 1
127.0.0.1:6379> ZADD user:4 3 paulblack@gmail.com
(integer) 1
127.0.0.1:6379> ZADD user:4 4 123
(integer) 1
127.0.0.1:6379> ZRANGEBYSCORE user:4 1 3
1) "Paul"
2) "Black"
3) "paulblack@gmail.com"
127.0.0.1:6379>
```

Figure 32: ZADD and ZRANGEBYSCORE commands for Sorted Set Data type in Redis-cli

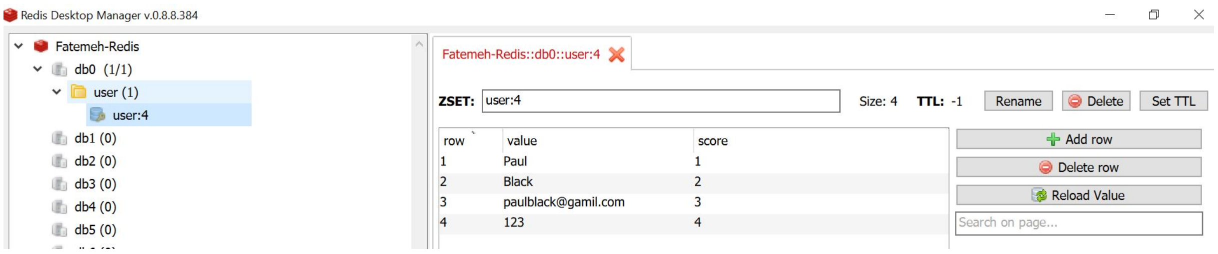


```php
<?php
//Redis Connection------------------------------------------------------------
//Connecting to Redis server on localhost
   $redis = new Redis();
   $redis->connect('127.0.0.1', 6379);
   //check whether server is running or not
   echo "redis Server is running: ".$redis->ping();

   $redis->ZADD("user:4", 1, "Paul");
   $redis->ZADD("user:4", 2, "Black");
   $redis->ZADD("user:4", 3, "paulblack@gamil.com");
   $redis->ZADD("user:4", 4, "123");

   $array = $redis->ZRANGEBYSCORE("user:4", "+inf","-inf");
?>
```

Figure 33: ZADD and ZRANGEBYSCORE commands for Sorted Set Data type in PHP-Redis



ZSET: user:4 Size: 4 TTL: -1

row	value	score
1	Paul	1
2	Black	2
3	paulblack@gamil.com	3
4	123	4

Figure 34: Sorted Set Data type in Redis Desktop Manager

References

Carlson, J. L. (2013). *Redis in action.* Manning. Retrieved from `https://redislabs.com/resources/ebook/`

Cummings, A. B., Eftekhary, D., & House, F. G. (2015). Caching a mongodb database with redis. Retrieved from `https://www.sitepoint.com/caching-a-mongodb-database-with-redis/`

Das, V. (2015). *Learning redis.* Birmingham ,UK: PACKT Publishing. Retrieved from `https://redislabs.com/resources/ebook/`

Db engine. (2017). Retrieved from `https://db-engines.com/en/`

How fast is redis? (2017). Retrieved from `https://redis.io/topics/benchmarks`

Nielsen, D. (2016). Popular redis uses for beginners. Retrieved from `https://redislabs.com/docs/popular-uses-redis-beginners-guide/`

Redis documentation. (2017). Retrieved from `https://redis.io/documentation`

Redis labs. (2017). Retrieved from `https://redislabs.com/`

Seguin, K. (n.d.). *The little redis book.* Retrieved from `http://openmymind.net/redis.pdf`

Silva, M. D. D., & Tavares, H. L. (2015). *Redis essentials.* Birmingham ,UK: Packt Publishing. Retrieved from `http://shop.oreilly.com/product/9781784392451.do`

What is a key-value database? (2016). Retrieved from `http://database.guide/what-is-a-key-value-database`

Woudehouse, C. (2016). Sql vs. nosql databases: What's the difference? Retrieved from `https://www.upwork.com/hiring/data/sql-vs-nosql-databases-whats-the-difference/`