UNIVERSITÉ LIBRE DE BRUXELLES

INFO-H-415 Advanced Databases

Key-value stores and Redis

17.12.2018

Amritansh Sharma - 000473628

CONTENTS

1 Introduction and Background **3**
- 1.1 Relational Databases 3
- 1.2 NoSQL Databases 4
- 1.3 Key Value Stores 5

2 Redis **7**
- 2.1 Redis vs Other Key-Value Stores 7
- 2.2 Redis Data structures 7
  - 2.2.1 Strings 8
  - 2.2.2 Hashes 9
  - 2.2.3 Lists 10
  - 2.2.4 Sets 13
  - 2.2.5 Sorted Sets 13
  - 2.2.6 HyperLogLog 14
  - 2.2.7 Geo-spatial Index 14
- 2.3 Redis Features 15
  - 2.3.1 Pipelining 15
  - 2.3.2 Redis Persistence 16
  - 2.3.3 Replication in Redis 17
  - 2.3.4 Partitioning in Redis 18
- 2.4 Redis Applications 20
  - 2.4.1 Leaderboards 20
  - 2.4.2 Implement expiry on items 21
  - 2.4.3 Unique N items in a given amount of time 21
- 2.5 Redis User Survey 22

3 From RDBMS to Redis: A simple database application **24**

4 Conclusion **26**

References **26**

1 Introduction and Background

1.1 Relational Databases

The traditional Relational Database Management System (RDBMS) is based on the relational model which was introduced by Edgar F. Codd. To model a real world scenario, the Relational model uses three fundamental components :- Data structures, Operators and Integrity rules. In a relational database, information is stored in tables which help organize and structure data in terms of related rows and columns. RDBMS transaction is the important concept of relational databases. A transaction is a single logical unit of work which accesses and possibly modifies the contents of a database. Transactions access data using read and write operations. In order to ensure accuracy, completeness, and data integrity, a transaction in a database system must maintain **A**tomicity, **C**onsistency, **I**solation, and **D**urability − commonly abbreviated as **ACID** properties.

These databases offer powerful information management tools to businesses at various levels, but the technology behind a relational database also imposes limitations as the organizations data grows quickly from a few MB to Terabytes, speed of data generation also becomes fast, and organization generates data in various formats. These drawbacks become apparent not only within the database itself, but also in the mechanisms by which business applications access the data.

The traditional Relational Database Management Systems model an existing problem using relational tables, which are an input for relational databases. Relational database entirely rely on tables, columns, and rows to manage structured and semi structured data. They use a language called Structured Query Language(SQL) for defining the schemas, manipulating and retrieving the data, creating user credentials, and creating constraints and indexes. To do these tasks SQL is enriched with Data Definition Language, Data Manipulation Language and Data Control Language. Relational databases were very popular over the years. But when the requirement and demand of business organizations rapidly change RDBMS lack the ability to satisfy business needs as they have increasingly failed to meet the performance, scalability, and flexibility needs that next-generation, data-intensive applications require, NoSQL databases have been adopted by enterprises to meet these demands.

1.2 NoSQL Databases

Data of business organizations is composed of structured, unstructured and semi structured forms of data. NoSQL, which stand for "non-SQL," is an alternative to traditional relational databases in which data is placed in tables and data schema is carefully designed before the database is built. NoSQL databases are especially useful for working with large sets of distributed data. NoSQL encompasses a wide variety of different database technologies that were developed in response to the demands presented in building modern applications.

With SQL databases, all data has an inherent structure. A conventional database like Microsoft SQL Server, MySQL, or Oracle Database uses a *schema*—a formal definition of how data inserted into the database will be composed. With NoSQL, data can be stored in a schema-less or free-form fashion. NoSQL is particularly useful for storing unstructured data, which is growing far more rapidly than structured data and does not fit the relational schemas of RDBMS.

NOSQL Database Types

To support specific needs and use cases different type of NoSQL database are developed and in use currently. Among these, the following four are the mostly used ones applicable for different scenarios:

1. Key-Value data stores - Key-value NoSQL databases emphasize simplicity and are very useful in accelerating an application to support high-speed read and write processing of non-transactional data. The stored values can be are any accepted type such as integers, strings,video, JSON document,sets and etc. For example - Redis and BerkeleyDB
2. Document Stores - Document databases typically stores self-describing documents. The data is a collection of key value pairs which is quite similar to key-value stores, but the only difference is that the values stored(literary documents) provide some structure and encoding of the managed data. XML,JSON and BSON are the common document encoding techniques
3. Wide-column stores - These store columns of data together instead of rows, and are optimised for queries over large datasets. For example - HBase and Cassandra
4. Graph stores - Graph databases uses edges and nodes to represent, store data and design a relationship among the objects. Graph structures are used with

edges, nodes and properties which provides index-free adjacency. Data can be easily transformed from one model to the other using a Graph database NoSQL database. For example - Neo4j and Graph

Advantages of NOSQL Databases

NoSQL databases offer enterprises important advantages over traditional RDBMS.

- NoSQL databases are more scalable and tend to provide superior performance.
- High and global availability-data replication across multiple servers and masterclass architecture for automatic resource distribution across nodes
- NOSQL provides flexible data models and dynamic schemas to address to address
    - Object-oriented programming that is easy to use and flexible
    - Large volumes of rapidly changing structured, semi-structured, and unstructured data
- Less need for Extract-Transform-Load
- Scale horizontally on commodity hardware

1.3 Key Value Stores

Key-value stores (or key-value databases) have emerged as an alternative solution to many of the limitations of traditional relational databases, where data is structured in tables and the schema must be predefined. A key-value database is a type of NoSQL database that uses a simple key-value method to store data. It stores data as a collection of key-value pairs in which a key serves as a unique identifier. Both keys and values can be anything, ranging from simple objects to complex compound objects. Each value is identified and accessed via a key, and stored values can be strings, numbers, counters, XML, JSON, HTML, images, short videos, binaries and more. It is the most flexible of the NoSQL models as it gives the application complete control over what is stored in the value.

When to use KVS

For a typical business application where data organisation and management is key, there is no particular reason to prefer Key-value stores over traditional RDBMS. For performance driven applications, however, Key-value stores provide the possibility of

much higher performance. Amazon DynamoDB, Apache Cassandra, Redis and Oracle BerkeleyDB are some popularly used key-value databases.

Example applications:-

- **Session store** - Key-value stores are important to develop session-oriented applications such as a web application that starts a session when a user logs in and is active until the user logs out or the session times out.
- **Shopping cart**- Consider an e-commerce website receiving millions of orders per second during festive shopping season. Key-value databases, through distributed processing and storage, can handle large amounts of data with extremely high number of state changes while servicing millions of users simultaneously. Key-value databases also have built-in redundancy, which can handle the loss of storage nodes.

Why use Key-value stores?

Being schema-less, key-value stores have a flexible data model allowing to accommodate changes easily. Most key-value databases make it easy to scale out on demand without significant redesign of the database. Further, they provide high performance by using simple commands instead of complex database operation in case of traditional relational databases. As the key-value stores are really simple and easy to handle, data can be modeled in a less complicated manner than in RDBMS.
Firstly, compared to other NoSQL databases, key-value stores we use simple, and limited set of operations but powerful to handle many complex tasks. Second , Key-Value Store can handle unstructured data in a fantastic manner. Based on the application needs. With the ability to have complex indices(through keys), associated with data values, applications using key-value databases can manipulate their data in different schemas formats depending on their needs.

Consider, for instance, the example of sensor data which usually comprises of a stream of small chunks of data that needs to be indexed and stored quickly. Document Stores have flexible schemas as well but require the data to be organized in a structured manner, typically XML, which is inefficient considering the amount of data required to store it as well as from the perspective of processing it later. Key-value databases, on the other hand, provide developers the ability to design the most efficient systems for processing such data.

2 Redis

Redis is a NoSQL database which is based on the principle of key-value store. Redis is a flexible, open-source (BSD licensed), in-memory data structure store, used as database, cache, and message broker. Being a NoSQL database, it facilitates users to store huge amounts of data without a size limit.

2.1 Redis vs Other Key-Value Stores

Redis is different compared to other key-value stores because of the following:-

- Redis is a different evolution path in the key-value databases where values can contain more complex data types, with atomic operations defined on those data types.
- Redis data types are closely related to fundamental data structures and are exposed to the programmer as such, without additional abstraction layers.
- Redis is an in-memory but persistent on disk database, so it represents a different trade off where very high write and read speed is achieved with the limitation of data sets that can't be larger than memory.
- Another great advantage of in-memory databases is that the memory representation of complex data structures is much simpler to manipulate compared to the same data structure on disk, so Redis can do a lot, with little internal complexity.

2.2 Redis Data structures

Redis, often described as a Key Value store, is much more than that. Describing redis as a data structure engine supporting different types of values is more accurate and helpful. In contrast to tradition key-value stores that associate String keys to String values, redis values can support not just strings but also much more complex data structures. The list of data structures supported by redis is as follows:-

- Strings
- Lists
- Sets

- Sorted sets
- Hashes
- HyperLogLogs

It can be difficult to understand how these data types work and when to use which data types and functionalities while trying to solve a given problem. Here, we provide a tutorial on Redis data types and their common usage patterns. All the examples in this tutorial will be performed via redis-cli. We start by introducing Redis Keys:-

Redis Keys

Redis keys are binary safe, that is, you can use any binary sequence as a key. The empty string is also a valid key in redis.

A few other rules and suggestions about naming keys:

- Try to stick with a schema. For instance "object-type:id" is a good practice, as in "user:100". For multi-word fields, dots and dashes can be used, for example, "comment:1234:reply-to".
- Avoid using very long keys. For example, using a key of size 1 kB is a bad idea not only memory-wise, but also because the lookup of the key in the dataset may require several costly key-comparisons.
- Avoid using very short keys. Instead of using "u100frnds" as a key, prefer using "user:100:friends". Clearly, the readability of the latter is much more than the former. The additional space for longer key is small compared to the space used by the key object itself and the value object.
- The maximum allowed key size is 512 MB.

2.2.1 Strings

Strings are the simplest of the Redis data structures. The basic string commands are `GET` and `SET`:

```
> SET user:email "user1@example.com"
> GET user:email
"user1@example.com"
```

We can define a string with a specific timeout using the `SETEX` command. This is useful to free up storage when the key is no longer required.

```
> SETEX mykey 10 myvalue
```

The key `mykey` will expire 10 seconds after its creation.

The `INCR` command(increments the integer value of a key by 1) is also very useful as it is useful to maintain counts. Some other useful string commands are `DECR`, `GETRANGE` and `SETRANGE`. These are used to decrease the integer value of the string by 1, `GET` a specific substring of a string and `SET` a specific substring of a string to a particular value.

2.2.2 Hashes

The hash data structure is exactly what one might expect it to be: it maintains field-value pairs. Most commonly used commands with this data structure are demonstrated below. `HSET` and `HGET` are used to set the value of a particular field of a key and to obtain the corresponding value respectively. `HMSET` is used to set multiple field-value pairs in one go.

```
> HMSET user:100 username raul birthyear 1987 sex male
OK
> HGET user:100 username
"raul"
> HGET user:100 birthyear
"1987"
> HGETALL user:100
1) "username"
2) "raul"
3) "birthyear"
4) "1987"
5) "sex"
6) "male"
```

`HINCRBY` key field increment : this command increments the value of a particular field of a hash by a given value. The `HDEL` command can be used to delete one or more hash fields.

A key defined as a JSON string can also be used as an alternative to a hash data structure. The ubiquitous support for JSON makes it a useful way to store a hash.

```
> SET user:10 "{username: 'joze', birthyear: 1987}"
```

The choice between using a string or a hash depends on how you need to manipulate and query the object. If control over individual fields is required, and you don't want to put the entire object into the app, a hash should be used. Otherwise, using a String would be better.

2.2.3 Lists

Lists allow you to associate a key with a list of values. Lists in Redis are implemented using Linked Lists. This implies that adding an element to a huge list (say millions of elements) at the start or its end takes `O(1)` time. Redis uses Linked Lists instead of an Array implementation because in database systems, it is important to have the functionality to efficiently add elements to very large lists.

Access to elements near the middle of the list, however, takes `O(N)` time where `N` is the size of the list. When accessing such elements needs to be efficient, Sorted Sets can be used instead of lists. We discuss the Sorted Set data structure later in this tutorial.

Most common types of list commands in Redis are `PUSH`, `POP` and `RANGE`. `LPUSH` and `RPUSH` add a new element to the left (or head) and the right (tail) of a list respectively. Redis allows the addition of multiple elements to the list using a single `LPUSH` (or `RPUSH`) command.

```
> RPUSH mylist alpha beta
(integer) 2
> LPUSH mylist gamma
(integer) 3
> LRANGE mylist 0 -1
1) "gamma"
2) "alpha"
```

```
3) "beta”
```

Note that `LRANGE` takes two indices as arguments, these are the indices of the first and the last index(both included) respectively to return in the range. An index can be negative and indicates that the counting should begin from the end of the list, -1 representing the index of the first element from the right and so on.

Redis lists support the pop element operation. This operation deletes an element from the list and retrieves its value at the same time. Just like `PUSH` operations, elements can be popped from either side of lists using the `LPOP` and `RPOP` operations.

```
> RPUSH mylist alpha beta
(integer) 2
> RPOP mylist
"gamma"
> RPOP mylist
"Beta"
> RPOP mylist
(nil)
```

A `null` value is returned to indicate that there are no elements in the list.

Capped Lists

Many of the use cases involving redis lists involve storing the `latest items` in the respective redis list. For example, they can be used to remember the latest updates posted by users in a social network. Redis lists can also be used to facilitate the communication between processes, using a consumer-producer framework where the producer pushes items into a list and a consumer (often a worker) consumes those items.

The `LTRIM` command in redis allows us to use lists as a capped collection, that is, storing only the most recent `N` items and discarding the rest. It slices the list to contain values specified in some range. The combination for the following commands can be used to maintain the list of most recently added 100 items while adding a new item.

```
> LPUSH mylist item
> LTRIM mylist 0 99
```

Note that though `LRANGE` is theoretically an `O(N)` operation, it is practically a constant time operation for accessing a small range of items near the head or the tail of the list.

Blocking operations on lists

Redis lists have a special feature called blocking operations that make them suitable for implementing queues and in general facilitating message passing and inter-process communication systems. Consider the producer consumer setup where you want to push items into a list with one process, and use a different process in order to actually do some operations on these items. A simple implementation of this setup is as follows:-

- Producers call `LPUSH` to push items to list
- Consumers call `RPOP` to obtain and operate on these items

However, if the list is empty and there is nothing to process `RPOP` in the consumer returns `NULL`. This forces the consumer to wait for some time and retry `RPOP`. This *polling* for a result forces Redis and clients to process useless commands (all the requests when the list is empty do no actual work, they just return `NULL`). It also results in an additional delay to the processing of items, since the worker waits for some time for the next request after it receives a NULL. Redis provides blocking versions of the `LPOP` and the `RPOP` command, namely the `BLPOP` and `BRPOP` commands that block if the list is empty, that is, they return to the caller only after a new element is added to the list or after a user-specified timeout.

The following `BRPOP` command can be called in the worker:

```
> BRPOP task_items 10
```

The above command waits for elements in the list task_items, and if after after 10 seconds no item is available returns `NULL`. If a timeout of 0 is given, it waits for items forever. Also, multiple lists can be specified in order to wait on multiple lists at the same time and get notified when the first list receives an item.

2.2.4 Sets

Sets represent an unordered collection of strings. Redis provides support for basic set operations like adding an element to a set, checking for the existence of an element in a set, set-operations like union, intersection and difference between multiple sets, etc. Some commands are listed below.

- `SADD` - add new elements to a set
- `SMEMBERS` - output all elements in a set
- `SISMEMBER` - check if an element exists in a set
- `SUNION` - union of multiple sets
- `SINTER` - intersection of multiple sets
- `SPOP` - remove and return a random element from a set
- `SCARD` - obtain the cardinality of a set

2.2.5 Sorted Sets

Sorted sets are sets which are ordered. The ordering is implemented by a floating point *score* field associated with each element in the Sorted Set. The elements in the Set are sorted in the order of the score of the elements. The score of an element needs to be provided when adding the element to the Sorted Set. For distinct elements with the same score, the ordering is defined by the lexicographical ordering of these strings.

The `ZADD` commands adds one or more elements to a Sorted Set. Note that it is similar to `SADD` except that it takes an additional argument which is the score corresponding to each element.

```
> ZADD scientists 1879 "Albert Einstein"
(integer) 1
```

Obtaining a list of scientists sorted by their year of birth is trivial with sorted sets since they are already sorted. Sorted sets are implemented using a dual-ported data structure containing both a skip list and a hash table, Redis performs an `O(log(N))` operation while adding an element to a sorted set. Additionally, sorted sets support

range operations on the scores of the elements. The ZRANGEBYSCORE command returns all elements of a sorted set with score in a specified range. Further, for sorted sets with all elements with the same score, the ZRANGEBYLEX command returns the elements in the specified lexicographic range.

2.2.6 HyperLogLog

Redis HyperLogLog is a data structure used to approximately count the number of unique elements in a set using a small constant amount of memory. The cardinality of the set if approximate with a standard error of 0.81% and using memory around 12kB per key.

The PFADD command adds the specified elements to a given HyperLogLog. The PFCOUNT operation returns the approximate cardinality of a given set/sets.

```
> PFADD tutorials "redis"
(integer) 1
> PFADD tutorials "mongodb"
(integer) 1
> PFADD tutorials "mysql"
(integer) 1
> PFCOUNT tutorials
(integer) 3
```

2.2.7 Geo-spatial Index

Redis has several commands for geospatial indexing but these commands lack their own data structure unlike other commands. These commands actually implemented using the sorted set datatype. A geospatial location in the form of latitude and longitude is encoded into the score of a sorted set using the geohash algorithm. A location is added to the index using GEOADD command. The GEODIST command can be used to obtain the distance between two locations. The GEORADIUS command returns the locations that are within a given radius of a specific point. These operations are logarithmic in the size of the index(that is, the sorted set).

2.3 Redis Features

2.3.1 Pipelining

Redis is a TCP server using the client-server model. The process of a client request followed by the server response is accomplished using the following steps:

- The client sends a query to the server, and reads from the socket, usually in a blocking way, for the server response.
- The server processes the command and sends the response back to the client.

The networking link connecting the client and the server can be slow, for instance, an Internet connection with several hops connecting server and client. Given the network latency, the time taken for the request to reach the server from the client and for the response the reach back to the client from the server is the round trip time(RTT). If many requests need to be served in a row, even for a server that can process, say 100k requests per second over a network with RTT of 100ms will be able to serve only 10 requests per second. There is fortunately a way to deal with this issue.

It is possible to implement a request-response server such that it is able to process new requests even before the client has read the old responses. Such an implementation allows the client to send multiple commands to the server without waiting for the replies and read all the replies in a single step instead. This technique is called pipelining and has been used widely since decades. For example, many POP3 protocol implementations used pipelining to speed up the process of downloading new emails from the server. Redis supports pipelining since its very early days.

When the client uses pipelining to send multiple requests, the server queues the replies using memory. It is hence better for the client to send requests in batches of a reasonable size to limit the memory footprint in the server while maintaining nearly the same RTT per request on average.

Pipelining not only reduces the latency cost due to RTT but actually also improves the total operations that can be performed per second in the server. This is because serving commands without pipelining is expensive in terms of doing socket I/O. This involves making the `read()` and `write()` syscall which incurs a speed penalty. Several pipelined requests can the served using a single `read()` and/or `write()` call thereby enhancing performance. We provide comparison of performance with and without

pipelining for some basic data operations using `redis-benchmark`. We observe a performance speedup of nearly `10x` with pipelining.

Intel Core i5 @ 2.5GHz (with pipelining)

```
$ redis-benchmark -r 1000000 -n 2000000 -t get,set,lpush,lpop -P
16 -q
SET: 248570.73 requests per second
GET: 344115.62 requests per second
LPUSH: 285591.91 requests per second
LPOP: 298018.19 requests per second
```

Intel Core i5 @ 2.5GHz (without pipelining)

```
$ redis-benchmark -r 1000000 -n 2000000 -t get,set,lpush,lpop -q
SET: 32437.52 requests per second
GET: 23858.66 requests per second
LPUSH: 30598.82 requests per second
LPOP: 32413.34 requests per second
```

2.3.2 Redis Persistence

Redis provides four possibilities for persistence:

- The RDB(Redis Database Backup) persistence performs point-in-time snapshots of the dataset at specific time intervals.
- The AOF(Append Only File) persistence logs every write operation received by the server, which will be performed again at server startup, to reconstruct the original dataset. Commands are logged using the same format as the Redis protocol itself, in an append-only fashion. Redis is able to rewrite the log on background when it gets too big.
- Persistence can be disabled if the data is needed exist just as long as the server is running, for example, for a temporary cache application.

- A combination of both AOF and RDB can be used in the same instance. Notice that, in this case, when Redis restarts the AOF file will be used to reconstruct the original dataset since it is guaranteed to be the most complete.

RDB is not good if you need to reduce the chances of data loss in case Redis stops working, for instance, after a power outage. Redis allows you to configure the points at which you wish to create snapshots (for example a few minutes or say a 1000 writes against the data). In case Redis stops working without a proper shutdown, the last few minutes of data would be lost. You can use RDB alone for your application if you can manage with a few minutes of data loss in the event of a disaster. However, having an RDB snapshot from time to time is a great idea for database backups, for faster restarts and in case of bugs in the AOF engine. Using AOF alone for persistence is hence not recommended.

2.3.3 Replication in Redis

Redis replication is based on master-slave replication. It allows slave Redis instances to be copies of master instances.

This system works using three main mechanisms:

1. When a master and a slave instances are well-connected, the master keeps the slave updated by sending a stream of commands to the slave, in order to replicate the effects on the dataset happening in the master side due to: client writes, key expiry/eviction or any other actions changing the master data.
2. When the link between the master and the slave breaks, for network issues or because a timeout is sensed in the master or the slave, the slave reconnects and tries to obtain the part of the command stream that it missed during the disconnection, that is, do a partial resynchronization.
3. The slave requests for a full resynchronization in case a partial resynchronization is not possible. Now the master needs to create and send to the slave a snapshot of all its data and then continue to send the stream of commands as its dataset changes.

For a majority of Redis use cases, asynchronous replication, being low latency and high performance, is the natural replication mode. Hence, the default replication mode in redis is asynchronous. However, Redis slaves asynchronously acknowledge with master the amount of data they received. So the master need not wait every time

for a command to be processed by the slaves. Since the master knows which slave already processed which commands, it allows to have synchronous replication if needed.

Some important facts about Redis replication are listed below:

- A master can have multiple slaves.
- Redis uses asynchronous replication. Slaves send asynchronous acknowledgements to the master about the amount of data processed.
- Slaves can have connections with other slaves. Besides connecting a several slaves to the same master, slaves can also be connected to other slaves in a cascading-like structure.
- Redis replication is non-blocking on the master side. When one or more slaves perform the initial synchronization or a partial resynchronization, the master will continue to be able to handle queries.
- Replication can be used not only for scalability, in order to have multiple slaves for read-only queries (for example, slow offloading slow read operations to slaves), but also to simply improving data safety and provide high availability.
- Replication can be used to avoid the cost of having the master writing the dataset to disk through persistence. Redis can typically be configured to avoid persisting to disk at all, and to connect a slave configured to save from time to time instead, or to have AOF enabled. This setup is required be handled carefully, since a master upon restart will start with an empty dataset, so if a slave tries to get synchronized with it, the slave will become empty as well.

2.3.4 Partitioning in Redis

Partitioning is the process of splitting data into multiple redis instances, such that every instance contains only a subset of the keys. Partitioning in redis serves the following two purposes:

- By using the total memory of many computers, it allows for much bigger database sizes instead of being limited to memory of a single computer.
- It enables scaling the computational power to multiple cores and across multiple computers as well as the network bandwidth to multiple computers and network adapters.

Suppose we have four redis instances `R0, R1, R2, R3,` and keys representing several users like `user:1`, `user:2`, `user:3...` and so on. Now, there are different ways to map a given key to a given Redis server.

One of these ways is to perform range partitioning, that is, mapping ranges of objects into specific redis instances. For instance, we can map users from ID 0 to ID 1000 into instance `R0`, while users form ID 1001 to ID 2000 into instance `R1` and so on. This system works and is actually used in practice, however, it has the disadvantage of requiring a table that maps ranges to instances. This table needs to be managed and a table is needed for every kind of object, so therefore range partitioning in Redis is often undesirable because it is much more inefficient than other alternative partitioning approaches. Alternatively, hash partitioning uses a hash function to map users to redis instances. For instance, the following two step scheme is used partitioning in case of the example above.

- Apply a hash function (say the `crc32` hash function) to a key (user) to convert it into a number.
- Use the modulo operation to transform this number into a number between 0 and 3, in order to map it one of my four redis instances.

Moreover, partitioning can be the implemented in different ways with respect to the software stack.

- Client side partitioning : The client directly selects the right node where to write or read a given key.
- Proxy assisted partitioning : The client sends requests to a proxy. The proxy forwards the request to the right node based on the configured partitioning schema, and sends back the replies to the client.
- Query routing : The client may send a query to a random instance, and the instance will ensure that the query is forwarded to the right node.

Some features of Redis don't work well with partitioning:

- Operations involving multiple keys are usually not supported. For instance, you can't (directly) perform the intersection between two sets if they are stored in keys mapped to different nodes.
- Redis transactions involving multiple keys can not be used.
- It is not possible to shard a dataset with a single huge key like a very big sorted set since partitioning is done at the key level granularity.

- Data handling becomes more complex with partitioning. For instance, multiple RDB / AOF files need to be handled.

2.4 Redis Applications

Redis has a wide range of applications: the most popular ones being as a high-speed cache, for message passing, as a real time data buffer, pub/sub and many more. Instead of using redis as the primary database, it is possible to leverage Redis by just adding it to your existing stack and use its functionalities to solve problems which are otherwise too slow or impossible to solve using the traditional Database. Here are some classic problems where Redis and its features can be used to provide solutions without the need to be the primary database.

2.4.1 Leaderboards

Consider the problem of taking a list of items, sorted by a score which are updated in real time at the frequency if several times per second. This pattern can be applied to various different scenarios. One classic scenario of this problem is the leaderboard in say an online game.

In an online game, several score updates are received from different users and you want to display a leaderboard with, say the top 10 users in terms of scores. Also, you want to show every user his current score. This problem is hard to model for databases that are not in-memory. However, even at the scale of millions of users with millions of score updates every minute, these problems are trivial to solve using sorted sets in Redis as described below.

Every time a new score is received by a user, we add to the sorted set:

```
> ZADD leaderboard <score> <userID>
```

To get the top 10 users by score simply do:

```
> ZREVRANGE leaderboard 0 9
```

To obtain the global rank of a user, do:

```
> ZRANK leaderboard <userID>.
```

Note that you can do more than this, for instance it is trivial to show the user the scores of users "near" his position, that is, to show the portion of the leaderboard that includes the score of our user.

2.4.2 Implement expiry on items

Sorted sets can also be used to index items by time. Using the unix time as score can be used to index items by time, in general. A use-case of this indexing is to expire items in our main database after a given amount of time has elapsed.

The implementation is as follows:

- Whenever a new item is added to our main (non Redis) database we add it into a sorted set. We use the time at which the item should expire as its score, basically *current_time + time_to_live*
- A background worker queries the sorted set using, for example, `ZRANGE` ... `WITHSCORES` to take the latest 10 items. We delete those items from the database having scores representing times already in the past.

2.4.3 Unique N items in a given amount of time

Another interesting application is finding the number of unique users accessing a resource in a specified amount of time (say on a particular day). Suppose, for instance, we wish to find the number of unique IP addresses that accessed the product page of a product on an e-commerce website. This is a problem which is hard to solve using other databases but trivial using redis.

For every new pageview, just add the IP address to a set corresponding to that product page:

```
> SADD pageviews:<date>:<product_page_id> <IP_address>
```

Now we can use `SCARD` and `SMEMBER` to obtain the unique IP addresses and check if an IP address accessed a product page respectively.

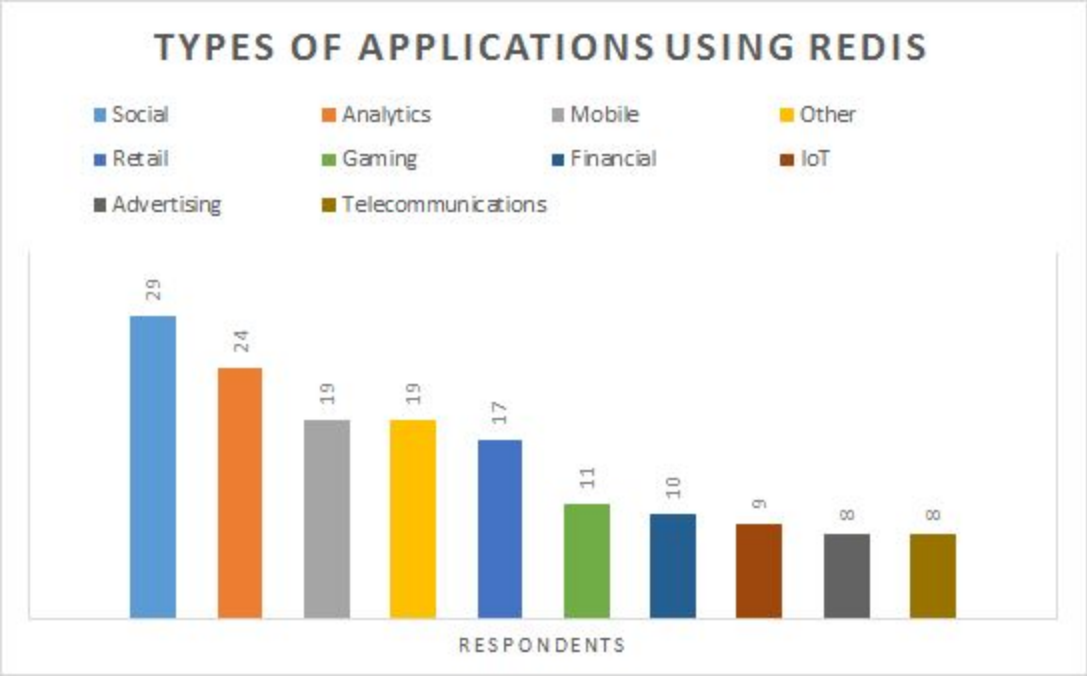
```
> SCARD pageviews:<date>:<product_page_id>

> SMEMBER pageviews:<date>:<product_page_id>
```

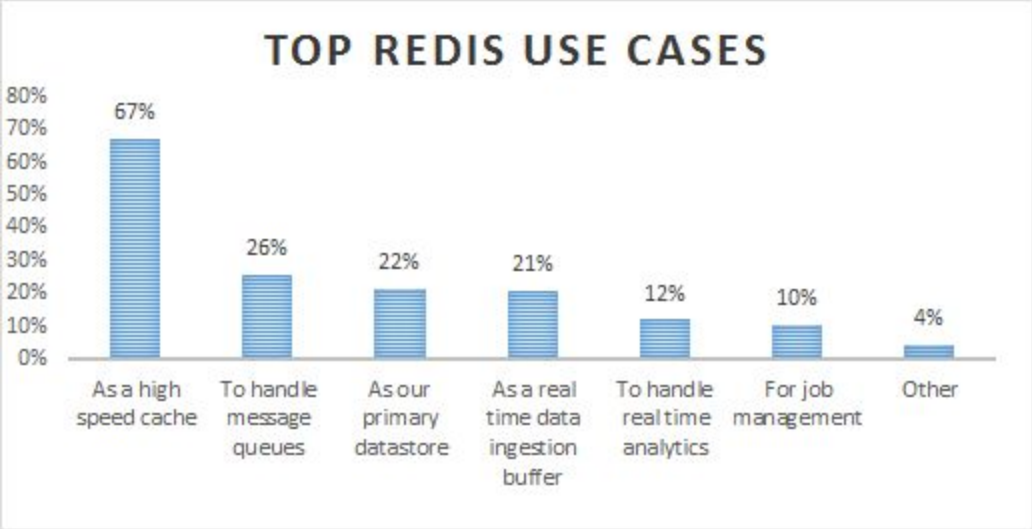
2.5 Redis User Survey

We present results of a recent survey of 116 redis users conducted by Redis Labs. They claim that 71% of respondents were planning to increase their usage of Redis as their data, applications and users scaled up. Following are some of the aspects of Redis usage from the survey:-

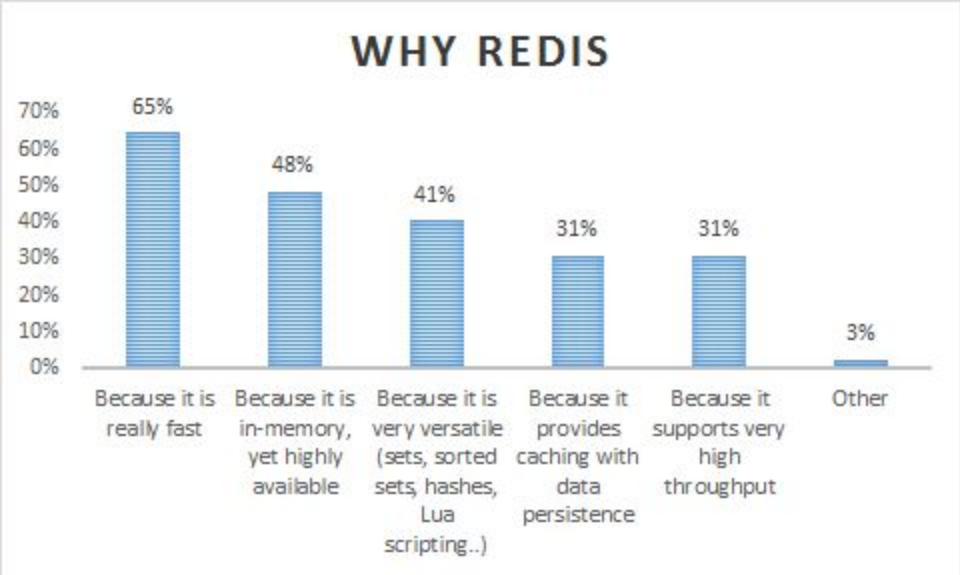
1. Top applications using Redis spanned Analytics, Mobile, Social, Retail, Financial, Gaming etc with a total of around 26 different application fields.



2. Most users respondents had more than one use case for Redis. Use as a high speed cache is most common as expected.



3. The top reason for using Redis by the users can be seen in the chart below.



3 From RDBMS to Redis: A simple database application

In this section, we demonstrate how a redis developer should design a redis database corresponding to a typical relational database with the ability to answer typical queries efficiently. Consider the following SQL command to create a table

```
CREATE TABLE `items` (
 `itemID` varchar(100) UNIQUE,
 `type` varchar(50) NOT NULL,
 `price` decimal(10,2) NOT NULL,
 `name` varchar(100) NOT NULL,
 `description` text NOT NULL,
);
```

In Redis, we do not need to define anything. Instead, we must think about how the data should be laid out. That is, how to name the keys and what data structures to use.

Here, we can use a common key pattern to represent each row in the above relation as a key. Let us define our keyspace as follows:

```
shopping:items:itemID
```

Note that the key pattern is only a convention we follow for our application and hence the key hierarchy is only implied from this convention. Actually, all keys belong to a flat namespace. However, all keys are in a flat namespace.

Let’s add a row to the items table using SQL try to replicate the functionality in Redis:

```
INSERT INTO `items` (`itemID`, ‘type’, `price`, `name`,
`description`) VALUES
```

```
('1', ‘Chocolate’, '3.90', 'Kitkat', 'Wafer chocolate');
```

With Redis, we create a hash to represent every row in the table:

```
> HMSET shopping:items:Kitkat itemID 1 type Chocolate price
10.99
name Kitkat description “Wafer chocolate”
```

At this point, we have a single hash with several field-value pairs. We can add several more hashes representing more items. However, to do anything more complex than just obtaining a single-value, we need to use more data structures. For instance, we can put all the keys representing items in a set.

```
> sadd shopping:all_items shopping:items:Kitkat
```

We can now do more complex queries, for instance, obtain the 3 most expensive items.

```
> SORT shopping:all_items BY shopping:items:*->price GET # GET
redishop:items:*->price LIMIT 0 3 DESC

1) "5.20"
2) "Snickers"
3) "5.15"
4) "Mars"
5) "3.90"
6) "Kitkat"
```

What we see here is that unlike a relational database with a pre-defined schema, the layout of data in redis depends in the kind of queries expected. For instance, if we expect range queries on the price of items, we should probably implement an index on the price of items using a sorted set.

```
 > zadd shopping:price_index 3.90 Kitkat
```

We can now make a range query to this sorted set.

```
> ZRANGEBYSCORE shopping:price_index 3 4 WITHSCORES

1) "Kitkat"

2) "3.90"
```

4 Conclusion

In many ways, Redis presents a simplified way to deal with data. It gets rid of much of the complexity and abstraction available in other systems. In many cases, this may make using Redis the wrong choice. In other applications, it may feel like Redis is custom-built for your data and application. Given the vast number of new technologies, it can be difficult to figure out what's worth investing time into learning. Given how easy it is to learn and considering the potential benefits Redis has to offer with its simplicity, it's one of the best investments, in terms of learning, that an individual or a team can make.

References

[1] Redis Documentation, https://redis.io/documentation

[2] The Results are in - Redis usage Survey 2016, https://redislabs.com/blog/the-results-are-in-redis-usage-survey-2016/

[3] How to take advantage of Redis just adding it to your stack, http://oldblog.antirez.com/post/take-advantage-of-redis-adding-it-to-your-stack.html

[4] The Road to Redis, https://medium.com/@stockholmux/from-sql-to-redis-chapter-1-145c82e4baa0

[5] Improving database performance with Redis, https://medium.com/@amangoeliitb/improving-database-performance-with-redis-dbd38fdf3cb

[6] The Little Redis Book, Karl Seguin, https://www.openmymind.net/redis.pdf

**[7] Redis from the ground up, Michael Russo,
http://blog.mjrusso.com/2010/10/17/redis-from-the-ground-up.html#caching**

[8] How Redis is used in practice, https://redislabs.com/blog/how-redis-is-used-in-practice/