Redis Database Implementation

05. 01. 2017

Written by

Florian Sander
florian.sander@student.hs-hannover.de

Nicolai Böker
nikolai.boeker@student.hs-hannover.de

Danar Armin
danaramin5889@gmail.com

Sascha Becker
s.becker@wertarbyte.com

Contents

1 Introduction **1**
1.1 The basics 1

2 Work Process **3**
2.1 Scenario 3
2.1.1 Queries 3
2.2 The Plan 3

3 Solutions **5**
3.1 Store all, process everything later 5
3.2 Store only what was asked for 6
3.2.1 Query 1 6
3.2.2 Query 2 7
3.2.3 Query 3 8
3.2.4 Query 4 8
3.2.5 Query 5 8
3.2.6 Query 6 8
3.3 Store all with partial redundancy and without post-processing 9

4 Conclusion **15**

1 Introduction

The document starts with a basic introduction to Redis and is followed by a description of the scenario which is set. Afterwards a detailed plan is developed. In the end all solutions are presented with a final recommendation.

1.1 The basics

Redis is an aggregated oriented key/value store belonging to the NoSQL databases. Redis stores values with keys, where keys identify the assigned value. For extracting a specific value the key must be known. Redis keys are binary safe, which means any binary sequence can be used as a key. Also an empty string is a valid key. In Redis keys can expire. After a determined time the key is deleted automatically. Redis enables the possibility to store even complex data structures by nesting values into values or mapping objects. Redis supports the following data structures:

- Strings
- Lists
- Hashes
- Sets
- Sorted Sets
- Bitmaps
- Message Queues

Redis provides two different storage mechanisms: Snapshotting and Append Only File Mode (AOF). A “Snapshot” holds all data in the memory. Therefore it is called “in-memory”. The data is stored onto the hard disk at a predetermined interval. These intervals can be configured by defining the number of writing operations and a time limit. Reloading the past operations to retrieve the primary state of the data after a system crash is an advantage of this method. With “AOF” every writing operations is stored onto disk immediately.

When storing strings to a Redis database the command “SET” is used. This way a value is placed to a new key. “SET” overwrites values of existing keys. So “SET” updates the value of existing keys. For retrieving the value “GET” is run. For each data type shown above exists different commands, which operates more or less the same way. Redis does not have a declarative query language. All queries in Redis are based on these commands. They can not be modified, except for the arguments, that can be passed over a command to another. The internal use of these commands is imperative. Having several single commands they can be combined to a single atomic transaction. For this Redis provides pipelining. If a command fails the whole transaction also fails. There are two transaction guarantees:

- All commands in a transaction are serialized, which basically means there is no request by another client served in the middle of the execution of a Redis transactions. The transactions is isolated.
- All commands are processed or none. It guarantees the already mentioned atomic state of a transactions.

When it comes to the more complex data structures as “Hashes” or “Sorted Sets” indexes can be created to pool specific values of these data types. In Redis different kinds of indexes can be created: numerical, lexicographical and composite. In general an indexes in Redis is a score. A numerical index is naturally a numerical score. When by chance two elements of a “Sorted Set” have the same numerical index they are ordered lexicographically, which means the value-strings are compared on binary level and so the elements get sorted by the raw values of their bytes. So internally indexes have a hierarchy.

2 Work Process

Our team consists of four members. We started early with reading documentations about Redis and searched for code snippets to see how the magic is done. After we established a basic understanding we used our newly won knowledge to define a general data model which is detailed described in a later section. Based on this model we followed up different paths of implementation. Each path ended in a successful and usable solution. While working on different ways we shared our experiences, so that everyone is able to understand how things were managed. We learned from each other.

2.1 Scenario

As an example scenario, we will use network data stored in Redis and queried over various use cases. Storing and analyzing network data is important for security. By capturing the network flow we can look into two different granularities, the flow level and the packet level. In this scenario we will look at the packet level.

2.1.1 Queries

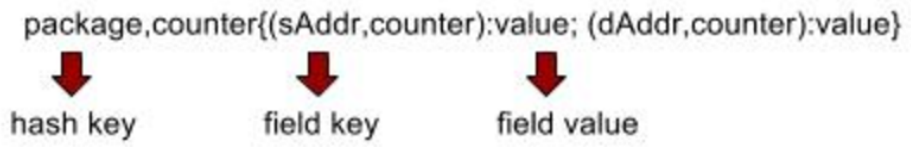
In order to analyze the network data we had been given various queries which should be implemented with redis. These are the queries we had to solve:

- Query 1: Retrieve all active connections at a certain point of time. Whereas active means that a package existed within the last second
- Query 2: Retrieve the overall data volume per minute for all connections between IP a.b.c.d and IP w.x.y.z
- Query 3: Retrieve all hosts that had connections to IP a.b.c.d on the HTTP port.
- Query 4: Retrieve all hosts that had incoming connections on well-known ports
- Query 5: Retrieve all packages that contain the byte sequence 0x35 0xAF 0xF8
- Query 6: Retrieve all hosts that had connections to outside hosts

2.2 The Plan

With the given scenario it was planned to have two different entities. One called “package” holding all for the queries needed attributes like “timestamp” or “source address” and the second called “data” for storing the raw data of the ethernet packages. Having two entities requires an association connecting these two objects. But first the objects must be created. Redis provides a suitable data type for doing so: Hashes. A hash consists of a several key-value fields while a prefixed key identifies a single hash.

Addressing this key means getting the whole hash with all its fields. To search for specific hashes incremental numerical indexes should be generated. The same goes for each field of the hash. Additional to a field key a numerical index should be defined. This way those fields' values of several hashes can be aggregated in a range.



Converting the pseudo code into a valid Redis command it would look like this:
redis.hmset((‘package’,counter),‘sAddr’,counter: sAddr)
Each field inside a single hash is an attribute of an ethernet packet. These attributes are labeled the following:

- timestamp as Integer
- sourceAdress as Long
- sourceAdressPrivate as Boolean
- sourcePort as Integer
- sourcePortWellKnown as Boolean
- destinationAdress as Long
- destinationAdressPrivate as Boolean
- destinationPort as Integer
- destinationPortWellKnown as Boolean

For the “data” entity in contrast simple Strings are applied. These strings should store the attributes “data” and “data.length”: redis.set(payload, data) and redis.set(payloadLength, data.length) Because Redis provides commands to get any key and its value, needing an association between the “data” and “package” is left out. Instead each entity could be accessed directly. When the need to address both entities comes up, multiple commands can be combined to do so.

All data should be stored persistent and therefore written onto disk immediately. In order to achieve this the AOF mechanism should be implemented.

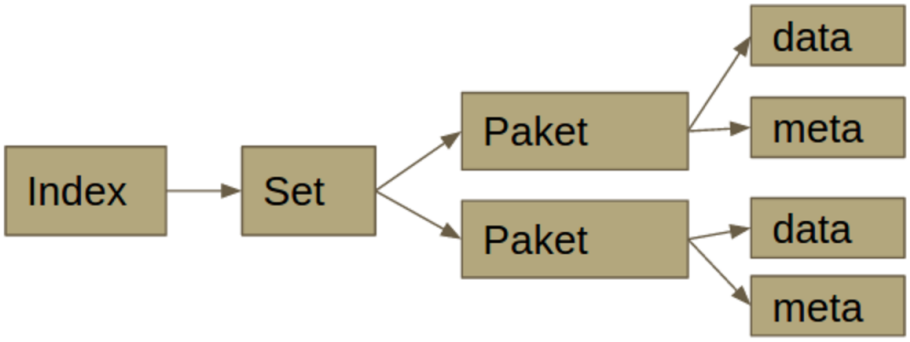
The appealed indexes are kind of pre-processing. Further pre-processing is necessary regarding to check whether a source or a destination IP is private or not. The same is valid if a port is well-known. A port is well-known when its port number is lower than 1024. And a port is a Http-port when its number is 80. Private IPs vary from 10.*.*.* over 172.[16-32].*.* to 192.168.*.*. The only planned post-processing should contain the summing of the data volume per minute.

3 Solutions

This part of the documentation is split up into three sections due to the presence of three various solutions based on different technologies. Each section describes the methodical process of developing the final program and explains the changes from what was originally planned. All solutions are compared with each other. At last there is a recommendation which solution would fit best to the given scenario.

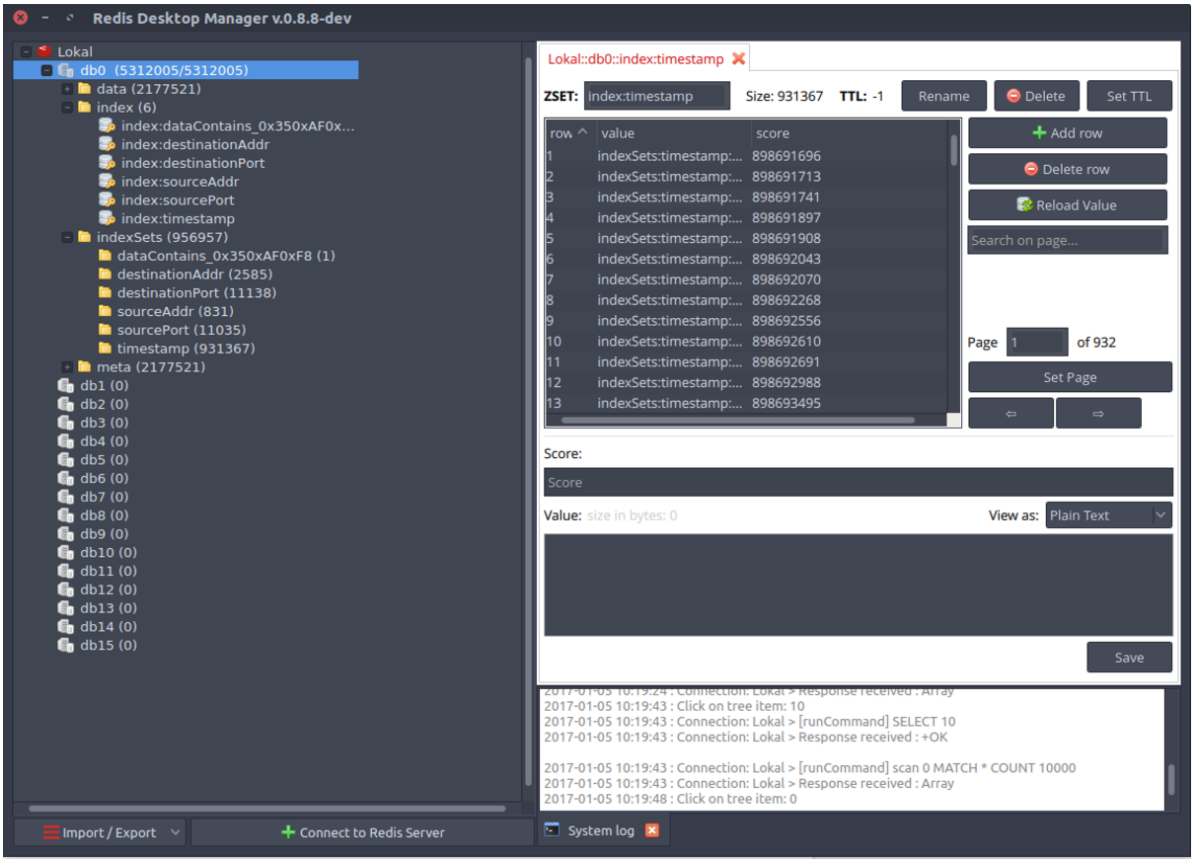
3.1 Store all, process everything later

This solution was realized with the underlying programming language Java. The used framework for Redis is called Jedis. This solution was the first approach, the idea was to just simply store all the references for the specific paket, very similar to foreign Keys in relational Databases. On the other hand this means postprocessing with Java. The repository can be cloned at this address: `https://github.com/SirSandmann/redis_dataimport` The structure of the data in redis for this approach looks like this:



All packets are seperated into the actual data and the meta data of the package, like in the original plan. The packets are in this approach two separated entities, realized with hashmaps in redis. Each packet has a unique number (e.g. 3230:meta for the meta data entity) which can be referenced. The references to the number of the packets are stores in a set. Here it gets a bit complicated. These sets are references by indexes holding the value e.g. indexSets:sourcePort:79 for the key of the set holding all the unique packet numbers. With this chaining post processing is necessary. Reviewing this solution the step with the indexes referencing a set is unnecessary, because the set could have been directly put into the index. The following screenshot shows the data structure in this approach within the Redis Desktop manager.

The index(6) contains all characteristics of the indexed data. The indexSets(956957) contains sets of all packageIDs with the specific characteristics the Set name specifies. These sets reference the data within the meta(2177521) and the data(2177521) space.



3.2 Store only what was asked for

This approach follows the concept of pure application, or in this case query based database. It was implemented with nodejs as the ground framework, pcap for network sniffing, hapi for the REST api and swagger for the REST gui visualisation. You can clone this repo here: `https://github.com/saschb2b/redis-pcap-monitor`

The idea is to write specific data into different redis data types as the data is being sniffed. So every write operation is handled in the "on packet" callback (3.1). As there is no general structure more a per query based implementation we will look at each query individually. Every operation focuses on what is being asked for. Splitted in two factors, what wants the user as an output and what do he needs to put into the query. The key is generally a combination of the query name and the parameters put in. We also convert ips to integers as planned to use range functions provided by redis.

3.2.1 Query 1

The input is a simple timestamp. The user wants all connections around that timeframe. So we created a sorted list called "timestamp" with a timestamp key (3.2). The value is a combination of source and destination address with the used ports. The output uses a redis query "between" to achieve the wanted one second timeframe (3.3)..

```
server.route({
  method: 'GET',
  path: '/service/pcap/start',
  handler: (request, reply) => {
    pcap_session = pcap.createSession("", "tcp")

    pcap_session.on('packet', function (raw_packet) {
      let packet = pcap.decode.packet(raw_packet)
      let ipPackage = packet.payload.payload
      let tcpPackage = ipPackage.payload

      if (ipPackage && ipPackage.version === 4 && tcpPackage) {
        let saddr = ipPackage.saddr.addr[0] * 1000000000 + ipPackage.saddr.addr[1] * 1000000 + ipPackage.saddr.addr[2] * 1000 + ipPackage.saddr.addr[3]
        let daddr = ipPackage.daddr.addr[0] * 1000000000 + ipPackage.daddr.addr[1] * 1000000 + ipPackage.daddr.addr[2] * 1000 + ipPackage.daddr.addr[3]

        let result = {
          saddr,
          daddr,
          sport: tcpPackage.sport,
          dport: tcpPackage.dport,
          timestamp: tcpPackage.options.timestamp | Date.now(),
          length: tcpPackage.dataLength,
          data: tcpPackage.data ? JSON.stringify(JSON.parse(JSON.stringify(tcpPackage.data)).data) : null
        }

        console.log(result)
      }
    })
    reply(`Listening on ${pcap_session.device_name}`)
  },
  config: {
    tags: ['api'],
    description: 'Starts pcap'
  }
})
```

Figure 3.1: Network Sniffer Loop

```
redisClient.zadd('timestamp',
  result.timestamp, `${result.saddr}:${result.sport}-${result.daddr}:${result.dport}`
)
```

Figure 3.2: Fill redis with timestamps

```
server.route({
  method: 'GET',
  path: '/query/connections',
  handler: (request, reply) => {
    redisClient.ZRANGEBYSCORE("timestamp", request.query.start, request.query.end, "WITHSCORES", function (err, obj) {
      reply(obj)
    })
  },
  config: {
    tags: ['api'],
    description: 'Retrieve all connections at a certain point in time',
    validate: {
      query: {
        start: Joi.number().required(),
        end: Joi.number().required()
      }
    }
  }
})
```

Figure 3.3: Query specific timeframes

3.2.2 Query 2

The inputs are two different ips. The user wants the overall data volume per minute between those ips. We created a hash storing three values. The start time which is only set once and never changes. A

stop time which is set whenever a new fitting connection was found. And a dataSum property which increments with every further fitting connection. Some post processing has to be done here. We need to check both connection directions and a conversion to data volume per minute.

```
if (result.data && result.length >= 0 && typeof result.length === 'number') {
  redisClient.hincrby(`${result.saddr}:${result.daddr}`,
    'dataSum', parseInt(result.length, 10)
  )
  redisClient.hsetnx(`${result.saddr}:${result.daddr}`,
    'start', result.timestamp
  )
  redisClient.hset(`${result.saddr}:${result.daddr}`,
    'stop', result.timestamp
  )
}
```

Figure 3.4: Fill redis with meta data concerning the data

3.2.3 Query 3

The inputs are an ip address as a destination and a port. In this case HTTP port which is 80 but we wanted some room. The user wants all hosts that this combination had connections to. We created several lists called “hosts” followed by the input combination. A valid list name could be “hosts:201.2.2.1:80”. The values are source addresses. Addresses that had connections to this specific connection. The output is then a generated query via the input followed by all the values within this specific list.

3.2.4 Query 4

The input is a start port and an end port. We wanted all connections that had incoming connections to well-known ports. Which means ports that are lower or equal to 1024. This was ideal for redis between. So we used a sorted list called “ports”. The key was the port and the value the destination address. The query simple asked for every key that was lower or equal to 1024.

3.2.5 Query 5

The user input is a decimal sequence. Due to technical limitation we can’t scan for a hex sequence. We created list called “data” holding all the packet data. We needed some pre processing in order to make this value searchable. The output starts a redis method called sscan which scans a list with a given regex and returns all packets that meet this criteria.

3.2.6 Query 6

The input is nothing. We wanted all hosts that had connections to outside hosts. We created a sorted list called “connections” where the key is the destination address and the value is the source address. The output queries via zrangebyscore every key that fits between a numeric range representing public ips.

```
server.route({
  method: 'GET',
  path: '/query/datavolume/perminute',
  handler: (request, reply) => {
    let resultSD = {}
    let resultDS = {}
    redisClient.hgetall(`${request.query.source}:${request.query.destination}`, function (err, obj) {
      resultSD.connection = `${request.query.source}->${request.query.destination}`

      if (obj) {
        resultSD.info = obj
        resultSD.duration = 60000 / ((obj.start - obj.stop) * -1)
        resultSD.dataPerMinute = parseInt(obj.dataSum, 10) / resultSD.duration
      }
      redisClient.hgetall(`${request.query.destination}:${request.query.source}`, function (err, obj2) {
        resultDS.connection = `${request.query.destination}->${request.query.source}`

        if (obj2) {
          resultDS.info = obj2
          resultDS.duration = 60000 / ((obj2.start - obj2.stop) * -1)
          resultDS.dataPerMinute = parseInt(obj2.dataSum, 10) / resultDS.duration
        }

        reply({
          resultSD,
          resultDS
        })
      })
    })
  },
  config: {
    tags: ['api'],
    description: 'Retrieve the overall data volume per minute for all connections between source and destination',
    validate: {
      query: {
        source: Joi.number().integer().required(),
        destination: Joi.number().integer().required()
      }
    }
  }
})
```

Figure 3.5: Query data volume over several packets

```
redisClient.sadd(`hosts:${result.daddr}:${result.dport}`, result.saddr)
```

Figure 3.6: Fill redis with hosts

As you can see this approach is very tight binded to its queries. Some allow a little more freedom and some none. If the application should grow the sniff loop needs to grow as well. Old data may not be suitable for further new queries.

3.3 Store all with partial redundancy and without post-processing

This solution was realized with the underlying programming language Python. Redis-py servers as the essential client. The solution does not dissociate from the original plan very much. It is based on the usage of Hashes to store each ethernet packet. At first all attributes on all layers of an ethernet packet are extracted by using a pcap reader in this case pyshark. These extracted values are assigned to variables

```
server.route({
  method: 'GET',
  path: '/query/hosts/ipport',
  handler: (request, reply) => {
    redisClient.smembers(`hosts:${request.query.destination}:${request.query.port}`, function (err, obj) {
      reply(obj)
    })
  },
  config: {
    tags: ['api'],
    description: 'Retrieve all hosts that had connections to ip a.b.c.d on a specific port',
    validate: {
      query: {
        destination: Joi.number().integer().max(255255255255).required().description('Destination ip'),
        port: Joi.number().integer().min(0).required().description('Destionation port')
      }
    }
  }
})
```

Figure 3.7: Query specific hosts

```
redisClient.zadd('ports',
  result.dport, result.daddr
)
```

Figure 3.8: Fill redis with ports

afterwards. As planned to check whether an IP is private or not is pre-processed likewise if a port is a http or well-known. But instead of using an boolean type here, a simple integer value 1 or 0 is selected. This is for the later explained indexes. Then a hash object will be generated. This hash object stores the following attributes:

- source address
- source port
- destination address
- destination port
- packet length
- timestamp

The amount of stored attributes is less than initially intended to store. Like stated in the original plan each hash gets an index and a name operating as the key identifying a single hash. As name the simple

```
server.route({
  method: 'GET',
  path: '/query/ports',
  handler: (request, reply) => {
    redisClient.ZRANGEBYSCORE('ports', request.query.start, request.query.end, 'WITHSCORES', function (err, obj) {
      let result = {}
      for(let i = 1; i < obj.length; i+=2){
        result[obj[i]] = []
      }
      for(let i = 0; i < obj.length; i+=2){
        result[obj[i+1]].push(obj[i])
      }
      reply(result)
    })
  },
  config: {
    tags: ['api'],
    description: 'Retrieve all hosts for ports',
    validate: {
      query: {
        start: Joi.number().integer().min(0).required().description('Start port'),
        end: Joi.number().integer().min(0).required().description('End port')
      }
    }
  }
})
```

Figure 3.9: Query specific port ranges

```
redisClient.sadd('data', result.data)
```

Figure 3.10: Fill redis with data blobs

```
server.route({
  method: 'GET',
  path: '/query/data/contains',
  handler: (request, reply) => {
    redisClient.sscan('data', 0, 'match', `*${request.query.pattern}*`, function (err, obj) {
      reply(obj)
    })
  },
  config: {
    tags: ['api'],
    description: 'Retrieve all packets that contain a byte sequence',
    validate: {
      query: {
        pattern: Joi.string().required().description('Byte sequence')
      }
    }
  }
})
```

Figure 3.11: Query specific data fragments

string “eth” is chosen. The index is a simple numerical index incremental rising and append to the name of the hash. So each hash object is unique and can be addressed correctly. Diverging from the plan no

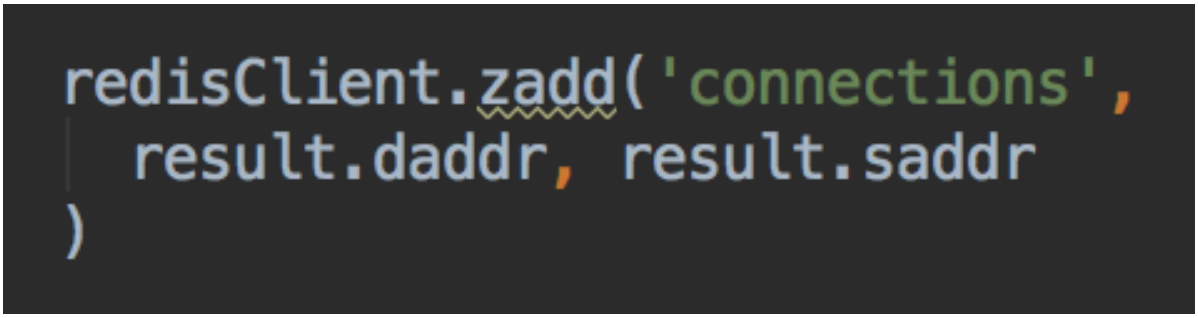


Figure 3.12: Fill redis with connections

```
server.route({
  method: 'GET',
  path: '/query/connections/public',
  handler: (request, reply) => {
    redisClient.ZRANGEBYSCORE("connections", 0, 10000000000, function (err, obj1) {
      redisClient.ZRANGEBYSCORE("connections", 11000000000, 172015255255, function (err, obj2) {
        redisClient.ZRANGEBYSCORE("connections", 172032000000, 192167255255, function (err, obj3) {
          redisClient.ZRANGEBYSCORE("connections", 192169000000, 255255255255, function (err, obj4) {
            reply({
              obj1,
              obj2,
              obj3,
              obj4
            })
          })
        })
      })
    })
  },
  config: {
    tags: ['api'],
    description: 'Retrieve all hosts that have connections to outside hosts'
  }
})
```

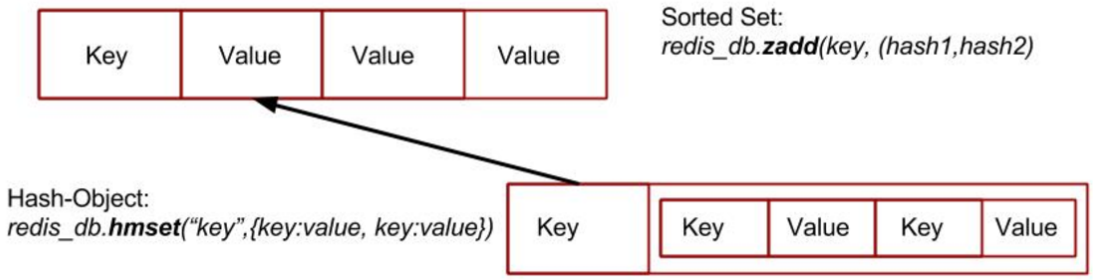
Figure 3.13: Query all outside connections

hash field gets an own index. It is not necessary. Each hash field consists of a name as a key to access this field and the assigned value. An example is shown in the picture:

As seen the hash object is created like it was primarily planned. There are two different field keys defined. The “statical” is a simple string which has always the same name. The “dynamically” ones are combined out of two variables holding the extracted source and destination address. The name as the key is the sum of these addresses as integers. So each hash object has a different field name here. Why it is done this way is explained later in the document. Having all essential attributes stored, it continues with preparing another Redis data type for the queries derived from the scenario. In order to perform these special queries or any query at all sorted sets are needed. This also a new aspect which was not indicated in the original plan. These sorted sets are the main lists which retrieve the stored hash objects as the result of a query. So sorted sets can hold the same values which leads to partial redundancy. An example of such a sorted set is shown in the following:

zadd('DPwK', dstPort$_w$*ellKnown*, (*redis$_d$b.hmget*((*'eth'*, *ct*),*' sIP'*,*' sPort'*)))

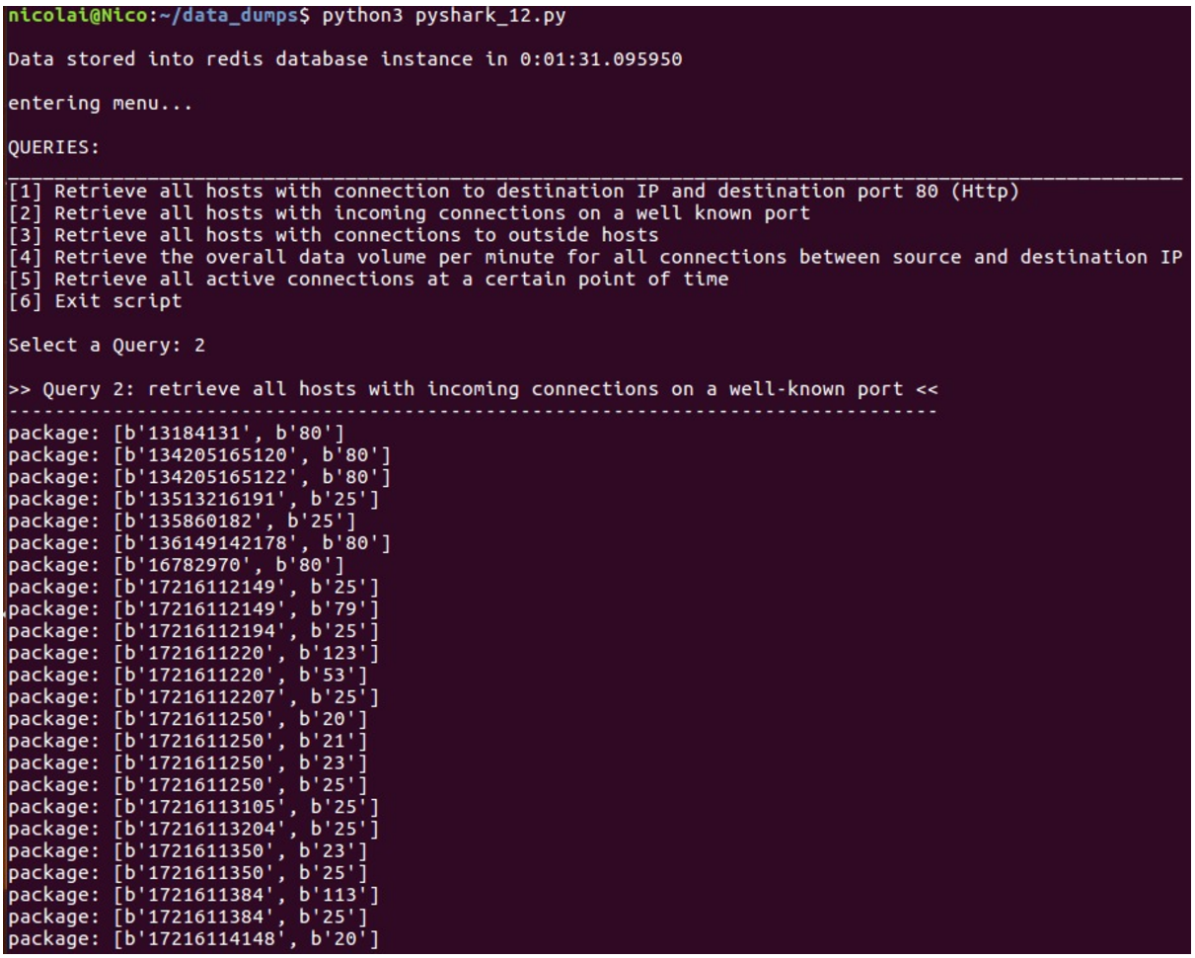
A sorted set gets a unique name "DPwK" as its key. The second parameter is the previously pre-processed value functioning as the index for the queries. The third one is a reference to the hash objects and particular fields of this hash object depending on the query. Using this reference overwrites any already existing value in the sorted set which leads to natural filtering duplicates - duplicates are not given and each value occurs only once. A reference nested into a sorted list is what was described as a complex data structure back in the basics and makes Redis very flexible when it comes to various kinds of queries. The example seen above points out that the sorted set retrieve all hash objects and its hash fields generated where the corresponding ethernet packet has a well-known destination port. There can be numerous sorted sets to cover any query. The general approach is displayed below:



Next follows the queries. As mentioned Redis provides complex data structures which can be accessed by simple commands. A further advantage of Redis. Basing on the example a query to retrieve all hosts that have a connection to a well-known port could be read like:

zrangebyscore('DPwK',1,1)

The command "zrangebyscore" enables to retrieve a bunch of stored hash objects with the same score. Therefore Redis first looks which key to access. This is defined in the first parameter of the command. So Redis searches for the sorted set with the name "DPwK" as its key. The second and third parameter define the limitation of the score range. Both are 1 meaning that all sorted sets with a score of 1 should return their values. "zrangebyscore" needs float or integer values to operate. That's why integers instead of booleans were selected while pre-processing if a port is well-known. The found sorted sets return their values. These values are references to hash objects generated when the pre-processed variable "dstPort$_w$*ellKnown*$\jmath$*was*1.



The screenshots shows the result of the query for all well-known ports. Each package exists only once due to filtering duplicates and consists of the destination address and the destination port belonging to the hash object which fulfills the querying criteria.
Every query in this solution works the same. It shows what Redis is capable of in mind that Redis provides many options and commands. The only post-processing necessary is facing the query to retrieve the overall data volume per minute between two IPs. All in all this solution is rather simple and does not exploit the full capability of Redis. Nevertheless it is sufficient.

4 Conclusion

Every of our solutions works fine and are actually build on one another. Through the working process we came to the final conclusion that we would recommend rather the last solution. This one allows to store everything we want and retrieve any data depending on the desired query. It aims for the native language of Redis and can therefore be implemented with any technology around. Its complexity is low. You only need to read the Redis documentation for understanding what we have done. All in all it mirrors our learning progress. This progress does not end with this final solution knowing there are many possibilities provided by Redis (e.g. Lua Script) we did not use.