AARHUS UNIVERSITET

Microservices and DevOps

Scalable Microservices

Redis

Henrik Bærbak Christensen

AARHUS UNIVERSITET

Motivation

- RDB's datamodel provide a rich modeling of data and queries
  - Tables, keys, foreign keys
  - Query language, sorting, filtering, ...
  - But lack a notion of objects (nested lists, nested objects, etc.)

- MongoDB's datamodel actually is even richer
  - JSON documents – can have nested documents and lists !
  - Query langage, sorting, filtering is well supported
  - Map Reduce

AARHUS UNIVERSITET

Motivation

- Redis is a key-value store and perhaps closer to the ‘heart’ of NoSQL
  - In memory, extremely fast, simple API
- And…
  - I have been using MongoDB since 2012 so about time to try something new…
- Disclaimer
  - I am on thin ice here, learning as you do!
  - **Feedback – publish good hints on forum ☺**

AARHUS UNIVERSITET

What to do with just the core?

- Key value stores require you to think in other ways
  - How do you create relations? Nested objects?
  - How do you query/search when you only have keys???
- The answer seems to be
  - **Create computed/foreign keys**
    - Key is not ‘8375a66b’ but ‘Room(0,0,0)’ – computed from domain
    - {course: ‘msdo’, participants: ‘plist_64ab2’}; pointing to a list of part.
  - **Create secondary indices**
    - Special lookup data structures that index your core data
    - Any CRUD is then updates of *two*+ datastructures
      - Add both the core data + add meta data to secondary index

AARHUS UNIVERSITET

Redis

An *Advanced* key-value store

Features

Features of Redis

← prev next →

Following is the list of main features of Redis:

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

Persistence: While all the data lives in memory, changes are asynchronously saved on disk using flexible policies based on elapsed time and/or number of updates since last save. Redis supports an append-only file persistence mode. Check more on Persistence, or read the AppendOnlyFileHowto for more information.

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

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

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

Master/Slave Replication: Redis follows a very simple and fast Master/Slave replication. It takes only one line in the configuration file to set it up, and 21 seconds for a Slave to complete the initial sync of 10 MM key set on an Amazon EC2 instance.

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

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

https://www.javatpoint.com/features-of-redis

AARHUS UNIVERSITET

Fundamental Data Structures

Key-value store is a special type of database storage system where data is stored in form of key and value pairs.

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.

- So – besides just (key, value) you also get things like lists (arrays/ordered in insertion sequence) and sets (unordered, no duplicates) and others

Examples

- The core HashMap thingy

Syntax:

```
redis 127.0.0.1:6379> COMMAND KEY_NAME
```

Example

```
redis 127.0.0.1:6379> SET javatpoint redis
OK
redis 127.0.0.1:6379> GET javatpoint
"redis"
```

Examples

- Sets

```
redis 127.0.0.1:6379> SADD javatpoint db2
(integer) 1
redis 127.0.0.1:6379> SADD javatpoint mongodb
(integer) 1
redis 127.0.0.1:6379> SADD javatpoint db2
(integer) 0
redis 127.0.0.1:6379> SADD javatpoint cassandra
(integer) 1
redis 127.0.0.1:6379> SMEMBERS javatpoint
1) "cassandra"
2) "db2"
3) "mongodb"
```

Examples

- Lists can be stored/read from head or tail

Example

```
redis 127.0.0.1:6379> LPUSH javatpoint sql
(integer) 1
redis 127.0.0.1:6379> LPUSH javatpoint mysql
(integer) 2
redis 127.0.0.1:6379> LPUSH javatpoint cassandra
(integer) 3
redis 127.0.0.1:6379> LRANGE javatpoint 0 10
1) "cassandra"
2) "mysql"
3) "sql"
redis 127.0.0.1:6379>
```

Examples

Redis Sorted Sets

← Prev Next →

Redis Sorted Sets are similar to Redis Sets but the first one has the unique feature of values stored. It means every member of a Sorted Set is associated with a score which can be used to take the sorted set ordered, from the smallest to the greatest score.

- Allows you to associate a *score* with each value in the set, and then retrieve only values within a given score range

key score value

```
redis 127.0.0.1:6379> ZADD javatpoint 1 redis
(integer) 0
redis 127.0.0.1:6379> ZADD javatpoint 3 cassandra
(integer) 0
```

```
redis 127.0.0.1:6379> ZRANGE javatpoint 0 10 WITHSCORES
1) "redis"
2) "1"
3) "cassandra"
4) "3"
```

AARHUS UNIVERSITET

Examples

- ‘Sorted Set’ seem like a good candidate for search indices

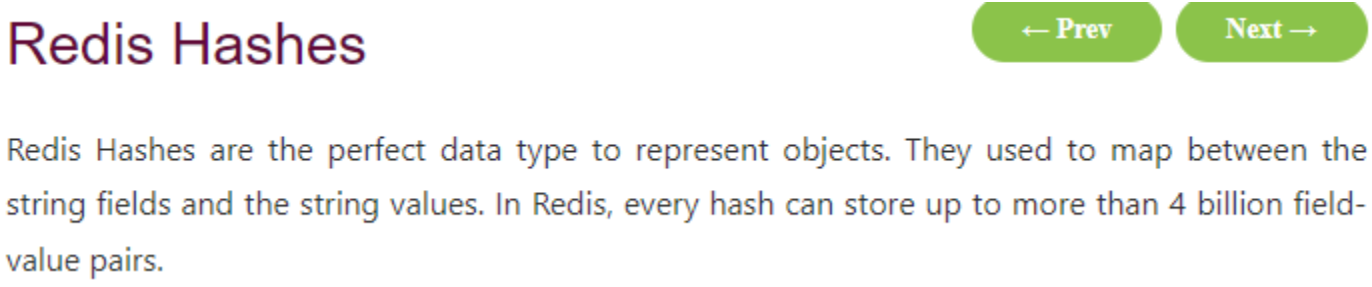
```
csdev@m31:~/proj/broker-internal/demo$ docker exec -ti redis-db redis-cli
127.0.0.1:6379> zadd grade 12 Bjarne
(integer) 1
127.0.0.1:6379> zadd grade 12 Ib
(integer) 1
127.0.0.1:6379> zadd grade 10 Nina
(integer) 1
127.0.0.1:6379> zadd grade 4 Birger
(integer) 1
127.0.0.1:6379> zadd grade 2 Carl
(integer) 1
127.0.0.1:6379> zadd grade 4 Kurt
(integer) 1
```

```
127.0.0.1:6379> zrangebyscore grade 2 2
1) "Carl"
127.0.0.1:6379> zrangebyscore grade 2 4
1) "Carl"
2) "Birger"
3) "Kurt"
127.0.0.1:6379> zrangebyscore grade 10 12
1) "Nina"
2) "Bjarne"
3) "Ib"
127.0.0.1:6379> zrangebyscore grade 12 12
1) "Bjarne"
2) "Ib"
```

‘Query’ = get all values whose score is in a given range.

Examples

- Hashes = the value is a HashMap itself



- I.e something like
  - (‘msdo’, { teacher: hbc, students: 17, website: “baerbak.cs….”})

Examples

- Structured objects/records
  - A set of (key,value) under one single key

```
127.0.0.1:6379> hmset c01 name swea year 2021 teacher hbc
OK
127.0.0.1:6379> hmset c02 name msdo year 2021 teacher hbc
OK
127.0.0.1:6379> hgetall c02
1) "name"
2) "msdo"
3) "year"
4) "2021"
5) "teacher"
6) "hbc"
```

```
127.0.0.1:6379> hget c02 name
"msdo"
127.0.0.1:6379> hget c02 teacher
"hbc"
```

AARHUS UNIVERSITET

Transactions

- Bob bob
  - Insert may mean two things
    - A) update main structure
    - B) update secondary index
  - Design for Failure
    - What happens if Redis dies between A) and B)?
- Atomicity is possible
  - Transactions: multi + exec

Example

```
redis 127.0.0.1:6379> MULTI
OK
redis 127.0.0.1:6379> EXEC
(empty list or set)
redis 127.0.0.1:6379> MULTI
OK
redis 127.0.0.1:6379> SET javatpoint redis
QUEUED
redis 127.0.0.1:6379> GET javatpoint
QUEUED
redis 127.0.0.1:6379> INCR visitors
QUEUED
redis 127.0.0.1:6379> EXEC
1) OK
2) "redis"
3) (integer) 1
```

AARHUS UNIVERSITET

Playing Around

Docker

AARHUS UNIVERSITET

Playing Around

- Test driving Redis is easy
  - Start a Redis non-persisting DB on standard port 6379

```
Started non-persisting redis using docker

    docker run -d --name redis-db -p 6379:6379 redis:5.0
```

  - Start a CLI
    - docker exec –ti redis-db redis-cli
  - And play around in the shell

- Find the ‘latest’ redis to use on Docker Hub

AARHUS UNIVERSITET

Java Driver

Jedis

AARHUS UNIVERSITET

Examples

- Redis commands are reflected ‘verbatim’ in Jedis

```
redis 127.0.0.1:6379> SADD javatpoint db2
(integer) 1
redis 127.0.0.1:6379> SADD javatpoint mongodb
(integer) 1
redis 127.0.0.1:6379> SADD javatpoint db2
(integer) 0
redis 127.0.0.1:6379> SADD javatpoint cassandra
(integer) 1
redis 127.0.0.1:6379> SMEMBERS javatpoint
1) "cassandra"
2) "db2"
3) "mongodb"
```

```java
public void addSets() {
  //let us first add some data in our redis server using Redis SET.
  String key = "members";
  String member1 = "Sedarius";
  String member2 = "Richard";
  String member3 = "Joe";

  //get a jedis connection jedis connection pool
  Jedis jedis = pool.getResource();
  try {
    //save to redis
    jedis.sadd(key, member1, member2, member3);

    //after saving the data, lets retrieve them to be sure that it has really added in redis
    Set<String> members = jedis.smembers(key);
    for (String member : members) {
      System.out.println(member);
    }
  } catch (JedisException e) {
    //if something wrong happen, return it back to the pool
    if (null != jedis) {
      pool.returnBrokenResource(jedis);
      jedis = null;
    }
  } finally {
    ///it's important to return the Jedis instance to the pool once you've finished using it
    if (null != jedis)
      pool.returnResource(jedis);
  }
}
```

https://javapointers.com/tutorial/use-redis-java-using-jedis/

Jedis Pool

- pool.getResource()
  - Returns a jedis connection; i.e. a new socket connection
- Do not use just a single connection for all transactions
  - Instead retrieve a new one for each ‘operation’ you need
    - Ala ‘cavestorage.addRoom()…’
    - Why not?
- Use the modern Java style ‘try-with-resources’
  - try (Jedis jedis = pool.getResource()) { do stuff; }
  - … to avoid writing a zillion ‘finally { jedis.close(); }

AARHUS UNIVERSITET

Jedis – Java Driver

- Java Driver

```
dependencies {
  compile project(':broker')
  compile group: 'org.mongodb', name: 'mongo-java-driver', version: '3.6.3'

  // Get the Redis java client
  compile group: 'redis.clients', name: 'jedis', version: '2.9.0'

  // Bind SLF4J it to the Log4J logging framework
  compile group: 'org.slf4j', name: 'slf4j-log4j12', version: '1.7.25'

  testCompile 'junit:junit:4.12'
  testCompile 'org.hamcrest:hamcrest-library:1.3'
}
```

https://javapointers.com/tutorial/use-redis-java-using-jedis/

- *Find latest driver at mvnrepository.com*

AARHUS UNIVERSITET

Scaling to MSDO

AARHUS UNIVERSITET

Architecture in MS Context

- Promises ‘availability’ and ‘scalability’

Persistence: While all the data lives in memory, changes are asynchronously saved on disk using flexible policies based on elapsed time and/or number of updates since last save. Redis supports an append-only file persistence mode. Check more on Persistence, or read the AppendOnlyFileHowto for more information.

Master/Slave Replication: Redis follows a very simple and fast Master/Slave replication. It takes only one line in the configuration file to set it up, and 21 seconds for a Slave to complete the initial sync of 10 MM key set on an Amazon EC2 instance.

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

- Looking forward to try it ☺