Focused Topics in Redis

March 9th, 10th, & 11th 2016

Jeremy Nelson
Course Instructor
Colorado Springs, CO 80904
Email: jermnelson@gmail.com

Day 1

- Overview of Data Structures
- Key Schemas & ORMs
- Master-Slave Replication
- Introduction to Redis Cluster

Day 2

- Pub/Sub Messaging
- High Availability w/Redis Sentinel
- Security
- Lua Scripting

Day 3

- Caching and Keyspace Notifications
- Trouble-shooting Redis Latency
- Related Redis Technologies

Focused Topics in Redis Day 1 - #1

Redis data types

A real strength of Redis is its powerful data structures that can be used to model and solve complex problems even with such primitive structures as Strings, Lists, Hashes, Sets, and Sorted Sets. Other data structures are stored as strings but have Redis commands for special uses. These are the Bitmap data type and the HyperLogLogs data type.

In this topic ...

---

it all begins with the **Key**

Redis keys are binary safe -any binary stream can be used as a key- although the most common (and recommended) stream to use is a string key, like "Person", other file formats and binary streams like images, mp3, or other file formats, can be used.

The flexibility of Redis allows for a wide diversity in how keys are structured and stored. Performance and maintainability of the Redis instance can be either positively or negatively impacted by the choices made in designing and constructing the Redis keys using in your database.

A good general practice when designing your Redis keys is to construct at least a rough outline of what information you are trying to store in Redis, which Redis data structure to use, and finally how your data structures relates to other information stored at different keys in your Redis database.

Critical Commands for *Keys*

EXISTS ***key***
TYPE ***key***
DEL ***key***
KEYS ***pattern***
SCAN ***pattern cursor***

Exercise: Key Commands

```
127.0.0.1:6379> SET testkey hello
OK
127.0.0.1:6379> EXISTS testkey
(integer) 1
127.0.0.1:6379> TYPE testkey
string
127.0.0.1:6379> KEYS *
1) "testkey"
127.0.0.1:6379> DEL testkey
(integer) 1
127.0.0.1:6379> EXISTS testkey
(integer) 0
127.0.0.1:6379> TYPE testkey
none
127.0.0.1:6379> SCAN 0
1) "0"
2) 1) "testkey"
127.0.0.1:6379> SCAN 0 MATCH "hell*"
1) "0"
2) (empty list or set)
```

String the most basic data-type

In Redis, a string is not merely alphanumeric characters as strings are normally understood to be in higher-level programming languages, but are serialized characters in C. Integers are stored in Redis as a string.

Critical Commands for *String*

SET ***key string optional nx|xx***
GET ***key***
INCR ***key***
INCRBY ***key integer***
DECR ***key***

DECRBY *key integer*

MSET *key1 string key2 string*

MGET *key1 key2 key3*

Exercise: Using *STRING* data type

Setting a string value with a characters and an integer using the `SET` and `GET` commands.

```
127.0.0.1:6379> SET Book:1 "Infinite Jest"
OK
127.0.0.1:6379> GET Book:1
"Infinite Jest"
127.0.0.1:6379> SET Book:1:ReadAction 1
OK
127.0.0.1:6379> GET Book:1:ReadAction
"1"
```

The `INCR` command will increase an string "integer" value by 1, the `INCRBY` command is similar but increase the value by a integer. The `DECR` and `DECRBY` commands decrease an string "integer" value by 1 or more.

> **NOTE:** the `INCR`, `INCRBY`, `DECR`, and `DECRBY` are atomic so other clients cannot change the value when the command is run.

```
127.0.0.1:6379> INCR Book:1:ReadAction
(integer) 2
127.0.0.1:6379> GET Book:1:ReadAction
"2"
127.0.0.1:6379> INCRBY Book:1:ReadAction 20
(integer) 22
127.0.0.1:6379> GET Book:1:ReadAction
"22"
127.0.0.1:6379> DECR Book:1:ReadAction
(integer) 21
127.0.0.1:6379> GET Book:1:ReadAction
"21"
127.0.0.1:6379> DECRBY Book:1:ReadAction 5
(integer) 16
127.0.0.1:6379> GET Book:1:ReadAction
"16"
```

What happens when `INCR` is used as follows?

```
127.0.0.1:6379> INCR Book:1 "Infinite Jest"
```

You can set multiple key-string values using `MSET` command. The `MGET` command retrieves multiple values from one or more keys.

```
127.0.0.1:6379> MSET Person:2 "Kurt Vonnegut" Person:3 "Jane Austen" Person:4 '
OK
127.0.0.1:6379> MGET Person:2 Person:3 Person:4
1) "Kurt Vonnegut"
2) "Jane Austen"
3) "Mark Twain"
```

A collection of one or more values is a List

Lists in Redis are ordered collections of Redis strings that allows for duplicates values. Because Redis lists are implemented as linked-lists, adding an item to the front of the list with `LPUSH` or to the end of the list with `RPUSH` are relatively inexpensive operations that are performed in constant time of O(1).

Exercise: Using *LIST* data type

The `LPUSH` command adds one or more elements to the front of a list, the `RPUSH` command adds one or more elements to the end of a list.

```
127.0.0.1:6379> LPUSH Book:1:comment "This was a fun read"
(integer) 1
127.0.0.1:6379> LRANGE Book:1:comment 0 -1
1) "This was a fun read"
127.0.0.1:6379> LPUSH Book:1:comment "Way too long!"
(integer) 2
127.0.0.1:6379> LRANGE Book:1:comment 0 -1
1) "Way too long!"
2) "This was a fun read"
127.0.0.1:6379> RPUSH Book:1:comment "Tennis anyone?"
(integer) 3
127.0.0.1:6379> LRANGE Book:1:comment 0 -1
1) "Way too long!"
2) "This was a fun read"
3) "Tennis anyone?"
```

The `LPOP` command remove the first element from the list and returns it the calling client while the `RPOP` command removes and returns the last element of the list.

```
127.0.0.1:6379> LPOP Book:1:comment
"Way too long!"
127.0.0.1:6379> LRANGE Book:1:comment 0 -1
1) "This was a fun read"
2) "Tennis anyone?"
127.0.0.1:6379> RPOP Book:1:comment
"Tennis anyone?"
127.0.0.1:6379> LRANGE Book:1:comment 0 -1
1) "This was a fun read"
```

The `LTRIM` command replaces a list with a range from an existing list.

```
127.0.0.1:6379> RPUSH Organization:1:members Person:1 Person:2 Person:3 Person:
(integer) 4
127.0.0.1:6379> LRANGE Organization:1:members 0 -1
1) "Person:1"
2) "Person:2"
3) "Person:3"
4) "Person:4"
127.0.0.1:6379> LTRIM Organization:1:members 0 2
OK
127.0.0.1:6379> LRANGE Organization:1:members 0 -1
1) "Person:1"
2) "Person:2"
3) "Person:3"
```

To add in implementing simple queues in Redis using the List data-type, the `BLPOP` and `BRPOP` commands are similar to `LPOP` and `RPOP` commands only they will block sending a return back to client if the list is empty. These blocking commands return two values, the key of the list and an element.

```
127.0.0.1:6379> BRPOP Organization:1:members 5
1) "Organization:1:members"
2) "Person:3"
127.0.0.1:6379> LRANGE Organization:1:members 0 -1
1) "Person:1"
2) "Person:2"
```

Critical Commands for *List*

LPUSH *key value*
RPUSH *key value*
LRANGE *key start end*
LPOP *key*
RPOP *key*

LINDEX *key index*

LINSERT *key BEFORE|AFTER pivot value*

BLPOP *key second delay*

BRPOP *key second delay*

Hash is a "dictionary" of fields and values

Hash data structure maps one or more fields to corresponding value pairs. In Redis, all hash values must be Redis strings with unique field names.

Critical Commands for *Hash*

HSET *key field value*

HGET *key field*

HMSET *key field1 value1* [*field2 value2* …]

HMGET *key field* [*field2* …]

HGETALL *key*

HEXISTS *key field*

HLEN *key*

HKEYS *key*

HVALS *key*

HDEL *key field*

HINCRBY *key field increment*

Exercise: Using *Hash* data type

To set the value of a single field, use the `HSET`, to retrieve a single field for a Hash key, use the `HGET` command.

```
127.0.0.1:6379> HSET Book:3 name "Cats Cradle"
(integer) 1
127.0.0.1:6379> HGET Book:3 name
"Cats Cradle"
```

You can set multiple field-values using the `HMSET` command and retrieve multiple field-values with the `HMGET` command. To retrieve all of the values of a Redis Hash, use the `HGETALL` command.

```
127.0.0.1:6379> HMSET Book:4 name "Slaughterhouse-Five" author "Kurt Vonnegut"
OK
127.0.0.1:6379> HMGET Book:4 author ISBN
1) "Kurt Vonnegut"
2) "29960763"
127.0.0.1:6379> HGETALL Book:4
1) "name"
2) "Slaughterhouse-Five"
3) "author"
4) "Kurt Vonnegut"
5) "copyrightYear"
6) "1969"
7) "ISBN"
8) "29960763"
```

The `HEXISTS` determine if a field exists in a Redis Hash. The `HLEN` returns the number fields in a Redis Hash.

```
127.0.0.1:6379> HEXISTS Book:4 copyrightYear
(integer) 1
127.0.0.1:6379> HEXISTS Book:4 barcode
(integer) 0
127.0.0.1:6379> HLEN Book:4
(integer) 4
```

The `HKEYS` returns all of the field names for a Redis Hash and the `HVALS` returns all of the values in the Hash.

```
127.0.0.1:6379> HKEYS Book:4
1) "name"
2) "author"
3) "copyrightYear"
4) "ISBN"
127.0.0.1:6379> HVALS Book:4
1) "Slaughterhouse-Five"
2) "Kurt Vonnegut"
3) "1969"
4) "29960763"
```

`HDEL` deletes a field from a hash and the `HINCRBY` increases the integer value of a hash field by a given number.

```
127.0.0.1:6379> HDEL Book:4 copyrightYear
(integer) 1
127.0.0.1:6379> HGETALL Book:4
1) "name"
2) "Slaughterhouse-Five"
3) "author"
4) "Kurt Vonnegut"
5) "ISBN"
6) "29960763"
127.0.0.1:6379> HSET Book:4 copyrightYear 1968
(integer) 1
127.0.0.1:6379> HGET Book:4 copyrightYear
"1968"
127.0.0.1:6379> HINCRBY Book:4 copyrightYear 1
(integer) 1969
127.0.0.1:6379> HGET Book:4 copyrightYear
"1969"
```

An unique and unordered collection of values is a **Set**

A set is a collection of string values where uniqueness of each member is guaranteed but does not offer ordering of members. Redis sets also implement union, intersection, and difference set semantics along with the ability to store the results of those set operations as a new Redis.

Exercise: Using *Set* data type

The `SADD` command adds one or members to a set, to display all of the members of a set use the `SMEMBERS` command.

```
127.0.0.1:6379> SET Organization:5 "Beatles"
127.0.0.1:6379> SADD Organization:5:member Paul John George Ringo
(integer) 4
127.0.0.1:6379> SMEMBERS Organization:5:member
1) "Ringo"
2) "John"
3) "Paul"
4) "George"
```

The `SISMEMBER` command determines if a value is a member of a set, the `SCARD` command returns the number of members in a set.

```
127.0.0.1:6379> SISMEMBER Organization:5:member "John"
(integer) 1
127.0.0.1:6379> SISMEMBER Organization:5:member "Ralph"
(integer) 0
127.0.0.1:6379> SCARD Organization:5:member
(integer) 4
```

Set operations, union and intersection, are available with the `SUNION` and `SINTER` commands respectively. The `SDIFF` command subtracts multiple sets.

```
127.0.0.1:6379> SET Organization:6 "Wings"
OK
127.0.0.1:6379> SET Organization:7 "Traveling Wilburys"
OK
127.0.0.1:6379> SADD Organization:6:member Paul Linda Denny
(integer) 3
127.0.0.1:6379> SADD Organization:7:member Bob George Jeff Roy Tom
(integer) 5
127.0.0.1:6379> SUNION Organization:5:member Organization:6:member
1) "Ringo"
2) "John"
3) "Paul"
4) "George"
5) "Denny"
6) "Linda"
127.0.0.1:6379> SUNION Organization:6:member Organization:7:member
1) "Paul"
2) "George"
3) "Roy"
4) "Bob"
5) "Denny"
6) "Tom"
7) "Linda"
8) "Jeff"
127.0.0.1:6379> SUNION Organization:5:member Organization:6:member Organizatio
 1) "Roy"
 2) "George"
 3) "Bob"
 4) "Denny"
 5) "Linda"
 6) "Ringo"
 7) "Paul"
 8) "John"
 9) "Tom"
10) "Jeff"
127.0.0.1:6379> SINTER Organization:5:member Organization:6:member
1) "Paul"
127.0.0.1:6379> SINTER Organization:5:member Organization:7:member
1) "George"
127.0.0.1:6379> SINTER Organization:6:member Organization:7:member
(empty list or set)
127.0.0.1:6379> SDIFF Organization:5:member Organization:6:member
1) "John"
2) "Ringo"
3) "George"
127.0.0.1:6379> SDIFF Organization:6:member Organization:5:member
1) "Linda"
2) "Denny"
```

Critical Commands for *Set*

SADD *key member* [*member* …]

SMEMBERS *key*

SISMEMBER *key member*

SCARD *key*

SUNION *key1 key2* [*key3* …]

SINTER *key1 key2* [*key3* …]

SDIFF *key1 key2* [*key3* …]

Sorted Set an ordered collection of unique values

A sorted set combines characteristics of both Redis lists and sets. Like a Redis list, a sorted set's values are ordered and like a set, each value is assured to be unique. The flexibility of a sorted set allows for multiple types of access patterns depending on the needs of the application.

Critical Commands for *Sorted Set*

ZADD *key score member* [*score member* …]

ZRANGE *key start stop* [*WITHSCORES*]

ZREVRANGE *key start stop* [*WITHSCORES*]

ZRANK *key start stop* [*WITHSCORES*]

ZSCORE *key member*

ZREM *key member*

ZCARD *key*

ZCOUNT *key min max*

ZRANGEBYSCORE *key min max* [*WITHSCORES*] [*LIMIT offset count*]

Exercise: Using *Sorted Set* data type

The `ZADD` command adds a member with a score to a sorted set. To retrieve a range of members by index, use the `ZRANGE` command. To retrieve a range of members from high score to low score, use the `ZREVRANGE` command.

```
127.0.0.1:6379> ZADD copyrightYear 1996 Book:1 2014 Book:2 1963 Book:3
(integer) 3
127.0.0.1:6379> ZADD copyrightYear 1969 Book:4
(integer) 1
127.0.0.1:6379> ZRANGE copyrightYear 0 -1
1) "Book:3"
2) "Book:4"
3) "Book:1"
4) "Book:2"
127.0.0.1:6379> ZRANGE copyrightYear 0 -1 WITHSCORES
1) "Book:3"
2) "1963"
3) "Book:4"
4) "1969"
5) "Book:1"
6) "1996"
7) "Book:2"
8) "2014"
127.0.0.1:6379> ZREVRANGE copyrightYear 0 -1
1) "Book:2"
2) "Book:1"
3) "Book:4"
4) "Book:3"
127.0.0.1:6379> ZREVRANGE copyrightYear 0 -1 WITHSCORES
1) "Book:2"
2) "2014"
3) "Book:1"
4) "1996"
5) "Book:4"
6) "1969"
7) "Book:3"
8) "1963"
```

In Redis sorted sets, the order of members is first determined by the score. If two members have identical scores, the members are then sorted lexicographically by value.

```
127.0.0.1:6379> ZADD Book:names 0 "Into the Wild" 0 "Cat's Cradle"
(integer) 2
127.0.0.1:6379> ZADD Book:names0 "Time Machine, The" 0 "Gravity's Rainbow"
(integer) 2
127.0.0.1:6379> ZRANGE Book:names 0 -1 WITHSCORES
1) "Cat's Cradle"
2) "0"
3) "Gravity's Rainbow"
4) "0"
5) "Into the Wild"
6) "0"
7) "Time Machine, The"
8) "0"
```

The `ZRANK` and `ZSCORE` commands retrieves the index (rank) and score of a member in a sorted set. The `ZREM` command removes the member from the sorted set.

```
127.0.0.1:6379> ZRANK copyrightYear Book:3
(integer) 0
127.0.0.1:6379> ZRANK copyrightYear Book:1
(integer) 2
127.0.0.1:6379> ZRANK copyrightYear Book:56
(nil)
127.0.0.1:6379> ZSCORE copyrightYear Book:3
"1963"
127.0.0.1:6379> ZSCORE copyrightYear Book:1
"1996"
127.0.0.1:6379> ZRANGE copyrightYear 0 -1 WITHSCORES
 1) "Book:3"
 2) "1963"
 3) "Book:1"
 4) "1996"
 5) "Book:5"
 6) "1996"
 7) "Book:6"
 8) "1996"
 9) "Book:2"
10) "2014"
```

Other useful sorted set commands include: `ZCARD` which returns the total number of members in the sorted set. `ZCOUNT` that returns the number of members that are in a range of scores. `ZRANGEBYSCORE` returns all members in a sorted set that have score between a minimum and maximum score. Both `ZCOUNT` and `ZRANGEBYSCORE` use `-inf` for all scores from negative infinity and `+inf` for all score to infinity.

```
127.0.0.1:6379> ZCOUNT copyrightYear 1900 1970
(integer) 2
127.0.0.1:6379> ZRANGEBYSCORE copyrightYear 1900 1970 WITHSCORES
1) "Book:3"
2) "1963"
3) "Book:4"
4) "1969"
127.0.0.1:6379> ZRANGEBYSCORE copyrightYear -inf 2000 WITHSCORES
 1) "Book:3"
 2) "1963"
 3) "Book:4"
 4) "1969"
 5) "Book:1"
 6) "1996"
 7) "Book:5"
 8) "1996"
 9) "Book:6"
10) "1996"
127.0.0.1:6379> ZRANGEBYSCORE copyrightYear 1998 +inf WITHSCORES
1) "Book:2"
2) "2014"
```

---

treating a string as sequence of bits is a **Bit array** or **Bitmap**

Redis bitstrings is a very memory efficient data structures in Redis. In a bitstring, 8 bits are stored per byte, with the first bit at position 0 being the significant one that is either set to either 0 or 1. The maximum size for Redis bitstrings is 512 MB.

Exercise: *Bitmap* **Commands**

`SETBIT` and `GETBIT` commands operate on a single bit offset in a Redis string. Bitmaps are highly efficient in storing boolean information on a range of related

```
127.0.0.1:6379> SET Movie:1 "Blade Runner"
OK
127.0.0.1:6379> HSET Movie:2 name "Star Wars"
OK
127.0.0.1:6379> SADD Movie:3 "2001 a Space Odyssey"
(integer) 1
127.0.0.1:6379> SETBIT Movie:UserPlays:2014-12-11 2 1
(integer) 0
127.0.0.1:6379> GETBIT Movie:UserPlays:2014-12-11 2
(integer) 1
```

The `BITCOUNT` command returns the total number of bits set to 1 in Redis bitmap string. The `BITPOS` command returns the first offset that matches either 1 or 0.

```
127.0.0.1:6379> BITCOUNT Movie:UserPlays:2014-12-11
(integer) 1
127.0.0.1:6379> BITPOS Movie:UserPlays:2014-12-11 1
(integer) 2
127.0.0.1:6379> BITPOS Movie:UserPlays:2014-12-11 0
(integer) 0
```

The `BITOP` command performs the following bit-wise operations on different strings: **AND**, **OR**, **XOR**, and **NOT**.

```
127.0.0.1:6379> SETBIT Movie:UserPlays:2014-12-12 2 1
(integer) 0
127.0.0.1:6379> SETBIT Movie:UserPlays:2014-12-12 1 1
(integer) 0
127.0.0.1:6379> BITOP AND and_result Movie:UserPlays:2014-12-11 Movie:UserPlays
(integer) 1
127.0.0.1:6379> GETBIT and_result 1
(integer) 0
127.0.0.1:6379> GETBIT and_result 2
(integer) 1
127.0.0.1:6379> GETBIT and_result 3
(integer) 0
127.0.0.1:6379> BITOP OR or_result Movie:UserPlays:2014-12-11 Movie:UserPlays:2
(integer) 1
127.0.0.1:6379> GETBIT or_result 1
(integer) 1
127.0.0.1:6379> GETBIT or_result 2
(integer) 1
127.0.0.1:6379> BITOP OR or_result Movie:UserPlays:2014-12-11 Movie:UserPlays:2
(integer) 1
127.0.0.1:6379> GETBIT or_result 1
(integer) 1
127.0.0.1:6379> GETBIT or_result 2
(integer) 1
127.0.0.1:6379> GETBIT or_result 3
(integer) 0
127.0.0.1:6379> BITOP XOR xor_result Movie:UserPlays:2014-12-11 Movie:UserPlays
(integer) 1
127.0.0.1:6379> GETBIT or_result 1
(integer) 1
127.0.0.1:6379> GETBIT or_result 2
(integer) 1
127.0.0.1:6379> GETBIT or_result 3
(integer) 0
127.0.0.1:6379> BITOP NOT not_result Movie:UserPlays:2014-12-11
(integer) 1
127.0.0.1:6379> GETBIT not_result 1
(integer) 1
127.0.0.1:6379> GETBIT not_result 2
(integer) 0
127.0.0.1:6379> GETBIT not_result 3
(integer) 1
```

Critical Commands for *Bitmap*

SETBIT *key offset value*
GETBIT *key offset*
BITCOUNT *key offset value*
BITPOS *key offset*
BITOP *operation destination_key key1* [*key2* …]

HyperLogLogs special probabilistic data type similar to a Sorted Set

Redis data type is a probabilistic data structure that provides an estimated count of unique items in a collection.

HyperLogLogs offer a reduced

Critical Commands for *HyperLogLogs*

PFADD *key element* [*element* …]
PFCOUNT *key* [*key* …]

PFMERGE *destination_key key1 [key2 ...]*

Exercise: *Bitmap* Commands

`PFADD` adds one or more elements to a HyperLogLogs. The `PFCOUNT` returns an approximation of the count in the HyperLogLogs with an standard error of .81%. With `PFMERGE` multiple HyperLogLogs can be merged into a single HyperLogLogs. The advantage of the HyperLogLogs is that is very efficient in memory (maximum size is 12k bytes) and does not require an proportional amount of memory (memory to items in the set) to perform a population count.

```
127.0.0.1:6379> PFADD EducationEvent:1:attendee Person:1 Person:2 Person:3 Pers
(integer) 1
127.0.0.1:6379> PFCOUNT EducationEvent:1:attendee
(integer) 4
127.0.0.1:6379> PFADD LiteraryEvent:1:attendee Person:4 Person:1
(integer) 1
127.0.0.1:6379> PFMERGE Event:attendee EducationEvent:1:attendee LiteraryEvent:
OK
127.0.0.1:6379> PFCOUNT Event:attendee
(integer) 5
```

Downloading and Using Geo Commands

With a running 3.2 Redis instance, we'll open a second terminal window and launch Redis-cli. We'll then add a couple of data points to the BayArea key with the `GEOADD` and then calculate the distance between San Francisco and San Jose with the `GEODIST`

```
127.0.0.1:6379> GEOADD BayArea 121.8863 37.7833 "San Jose" 122.4167 37.7833 "Sa
127.0.0.1:6379> GEODIST BayArea "San Francisco" "San Jose"
"46624.876174299716"
127.0.0.1:6379> GEODIST BayArea "San Francisco" "San Jose" km
"46.624876174299715"
127.0.0.1:6379> GEODIST BayArea "San Francisco" "San Jose" mi
"28.971426904382987"
```

With the `GEORADIUS` and `GEORADIUSBYMEMBER` returns the geospatial information that are within the borders of an area specified with a central location and a maximum distance from the center.

```
127.0.0.1:6379> GEORADIUS BayArea 121.9692 37.3544 100 mi
1) "San Jose"
2) "Oakland"
3) "San Francisco"
127.0.0.1:6379> GEORADIUS BayArea 121.9692 37.3544 30 mi
1) "San Jose"
```

Use Case: *Linked Data Fragments Server*

Overview of Linked Data

Based upon Sir Tim Berners-Lee idea of the Semantic Web, libraries are adopting various RDF graph-based vocabularies to describe their collections while linking with other institutions and projects to extend and enrich the discovery of their materials by patrons.

RDF Graphs

RDF Graphs are made up of triples, statement made up one each of `Subjects`, `Predicates`, and `Objects`.

Subjects are made up either IRIs (most commonly an URL), or Blank Nodes, unique identifiers within a single RDF graph.

Predicates must be IRIs and is the graph edge that connects the subject and object entities with a relationship.

Objects are made up of either URLs, Literal values, or Blank Nodes.

Data structure evolution

During the development of the Linked Data Fragments Server, the variety and types of Redis data-structures that were used changed as further design, development, and testing continued in the project.

Redis Strings

The initial schema design for the Linked Data Fragments Server was to generate a SHA1 hash digest for each subject, predicate, and object and then storing a simple composite key made up of the triple.

Subject

URL: http://www.denverbroncos.com/

SHA1 Hash digest:
e61da38f50b68b932ddcde7d3ec6de5baddba0e9

Predicate

URL: https://schema.org/name

SHA1 Hash digest:
4dab97eaa98fcba6bd7a204da2fea672e51f6a84

Object

Literal Value: Denver

SHA1 Hash digest:
0a84cadbc87cbca9

```
127.0.0.1:6379> SETNX e61da38f50b68b932ddcde7d3ec6de5baddba0e9 http://www.denve
(integer) 1
127.0.0.1:6379> SETNX 4dab97eaa98fcba6bd7a204da2fea672e51f6a84 https://schema.
(integer) 1
127.0.0.1:6379> SETNX 0a84cadbc87cbca90ddf0f90e4af57d6f0753335 "Denver Broncos"
(integer) 1
127.0.0.1:6379> SETNX e61da38f50b68b932ddcde7d3ec6de5baddba0e9:4dab97eaa98fcba
(integer) 1
```

References and Resources

1. From the redis.io website section on commands http://redis.io/commands

Key Schemas and Object Relational Managers

In this topic …

Based on a brief mention in the official Redis documentation on suggested rules about Redis keys, a consistent key naming schema improves comprehensibility and maintainability of your Redis solution while positioning the datastore for future growth. Redis itself does not have any schema checking or validation functions although some basic validation can be done through the use of the `EXISTS` and `TYPE` Redis commands.

Relationship Modeling

A key schema should provide guidance for adding new Redis keys to an existing Redis-based application. An effective Redis Key schema establishes a naming conventions relating keys together. These relationships loosely couple data structures together onto which application and business logic can apply through client code.

Delimiters

The most commonly used delimiters between words used in Redis Keys and one seen through-out Redis documentation and examples is a colon `:` but any unicode character can be used with other choices for delimiters being a forward slash `/` and period `.`.

Book-Film Key Schema

A simple Book-Film Catalog Schema

Name	Redis Data Type	Description	Relationships
`book:{counter}`	Hash	Stores title, author, ISBN, format, copyright date, page number, and price metadata for a book	Key is stored in different Genre sets and Sales ranking sorted sets
`books:{genre}`	Set	A set of Redis keys for books classified as a genre, such as popular fiction, mysteries, science fiction, and technical books	Stores all book keys that have been classified as a single genre. Used with other genre sets for calculating books in multiple genres with SINTERSTORE and books that are only in a single genre with SDIFFSTORE

`books:sales-rank`	sorted set	Stores sales ranking of each book with the total number of titles sold as the score in the sorted set	Stores ranking of all Redis book keys

Example Book & Film Key Schema

```
all:sales-rank
global:book
book:1
book:2
book:3
books:genre:popular-fiction
books:genre:sci-fiction
books:format:ebook
books:format:paperback
books:sales-rank
global:film
film:1
film:2
films:genre:comedy
films:genre:drama
films:format:bluray
films:format:dvd
films:sales-rank
```

Node.js ORMs

Nohm

A node.js implementation of a Redis Object Relations mapper. Project is available at http://maritz.github.com/nohm/

```
var nohm = require('nohm').Nohm;
var redis = require('redis').createClient();

nohm.setClient(redis);
nohm.setPrefix('catalog');

nohm.model('Book', {
  properties: {
    title: {
      type: 'string',
      unique: false,
      validations: [
        'notEmpty'
      ]
    },
    author: {
      type: 'string',
      unique: false
    },
    price: {
      type: 'integer',
      defaultValue: 20
    }
 }
});

nohm.model('Sales', {
  properties: {
    salesDate: {
      type: 'datetime'
    }
  }
});

var infiniteJest = nohm.factory('Book');
var bookSales = nohm.factory('Sales');


infiniteJest.p({
  title: 'Infinite Jest',
  author: 'David Foster Wallace',
  price: 20
});
```

```
bookSales.p({
  salesDate: '2016-03-09'
});

bookSales.link(infiniteJest, 'item');

infiniteJest.save();
bookSales.save();
```

Redis Key Structure

```
127.0.0.1:6379> KEYS *
 1) "catalog:hash:Sales:ilkx1bwy9813lk9wqtrb"
 2) "catalog:meta:idGenerator:Book"
 3) "catalog:meta:version:Sales"
 4) "catalog:meta:version:Book"
 5) "catalog:meta:properties:Sales"
 6) "catalog:relations:Book:itemForeign:Sales:ilkx1bwxls4rf8loq
 7) "catalog:meta:idGenerator:Sales"
 8) "catalog:relationKeys:Book:ilkx1bwxls4rf8loq10"
 9) "catalog:relationKeys:Sales:ilkx1bwy9813lk9wqtrb"
10) "catalog:idsets:Book"
11) "catalog:relations:Sales:item:Book:ilkx1bwy9813lk9wqtrb"
12) "catalog:meta:properties:Book"
13) "catalog:idsets:Sales"
14) "catalog:hash:Book:ilkx1bwxls4rf8loq10"
```

Exercise 1 - MONITOR mode

A useful troubleshooting technique is to connect to your running Redis instance with `redis-cli` and issue the `MONITOR` command.

```
127.0.0.1:6379> MONITOR
OK
1457532195.717891 [0 127.0.0.1:50575] "info"
1457532195.722390 [0 127.0.0.1:50575] "get" "catalog:meta:version:Book
1457532195.722526 [0 127.0.0.1:50575] "get" "catalog:meta:version:Sale
1457532195.722613 [0 127.0.0.1:50575] "sismember" "catalog:idsets:Book
1457532195.722795 [0 127.0.0.1:50575] "sismember" "catalog:idsets:Sale
1457532195.726865 [0 127.0.0.1:50575] "set" "catalog:meta:version:Book
1457532195.726920 [0 127.0.0.1:50575] "set" "catalog:meta:idGenerator:
1457532195.726952 [0 127.0.0.1:50575] "set" "catalog:meta:properties:B
1457532195.727017 [0 127.0.0.1:50575] "set" "catalog:meta:version:Sale
1457532195.727043 [0 127.0.0.1:50575] "set" "catalog:meta:idGenerator:
1457532195.727060 [0 127.0.0.1:50575] "set" "catalog:meta:properties:S
1457532195.730246 [0 127.0.0.1:50575] "sadd" "catalog:idsets:Book" "il
1457532195.730651 [0 127.0.0.1:50575] "sadd" "catalog:idsets:Sales" "i
1457532195.733823 [0 127.0.0.1:50575] "MULTI"
1457532195.733858 [0 127.0.0.1:50575] "hmset" "catalog:hash:Book:ilkx1
1457532195.733911 [0 127.0.0.1:50575] "EXEC"
1457532195.733917 [0 127.0.0.1:50575] "MULTI"
1457532195.734083 [0 127.0.0.1:50575] "hmset" "catalog:hash:Sales:ilkx
1457532195.734122 [0 127.0.0.1:50575] "EXEC"
1457532195.736822 [0 127.0.0.1:50575] "MULTI"
1457532195.736939 [0 127.0.0.1:50575] "sadd" "catalog:relationKeys:Sal
1457532195.736972 [0 127.0.0.1:50575] "sadd" "catalog:relations:Sales:
1457532195.736996 [0 127.0.0.1:50575] "EXEC"
1457532195.737128 [0 127.0.0.1:50575] "MULTI"
1457532195.737464 [0 127.0.0.1:50575] "sadd" "catalog:relationKeys:Boo
1457532195.737495 [0 127.0.0.1:50575] "sadd" "catalog:relations:Book:i
1457532195.737518 [0 127.0.0.1:50575] "EXEC"
```

Master-Slave Replication

Redis replication is based on master Redis instances that have their contents mirrored with one or more Redis slave instances.

Facts about Master-Slave Replication

- Master and Slaves use asynchronous replication
- Masters can have multiple slaves (1-N)
- Slaves can accept connections from other slaves (i.e. a slave could have as it's master a Redis instance that is slave to a master instance)
- Replication is non-blocking on both master and slave instances; a master can continue to respond to queries while slaves are synchronizing, a slave can be configured to respond to queries with old data while synchronizing with the master

In this topic ...

Overview

Basic Replication

To illustrate the basic operations of using Redis replication, we will start-up two running Redis instances and two `redis-cli` sessions connecting to the two instances.

Open a terminal session and for this example we will use the unix `screen` utility to run all of our sessions. If you don't have `screen` installed, you can open four individual terminal sessions.

Redis Master

```
$ ~/redis/src/redis-server --dbfilename master.rdb --port 6379
```

Redis CLI session of **Redis Master**

```
$ ~/redis/src/redis-cli -p 6379
127.0.0.1:6379> DBSIZE
(integer) 0
```

Next, we will add some field-values to a Redis hash key in our Master datastore

```
127.0.0.1:6379> HMSET Book:1 name "Infinite Jest" author "David
OK
```

Redis Slave

```
$ ~/redis/src/redis-server --dbfilename slave.rdb --port 6380
```

Redis CLI session of **Redis Slave**

```
$ ~/redis/src/redis-cli -p 6380
127.0.0.1:6380> DBSIZE
(integer) 0
```

Now, we'll issue the `SLAVEOF` command to make this instance replicate the master's content:

```
127.0.0.1:6380> SLAVEOF localhost 6379
OK
```

After adding the `Book:1` Redis hash to our Master instance, we'll check to see if the key has been replicated to our slave instance:

```
127.0.0.1:6380> KEYS *
1) "Book:1"
127.0.0.1:6380> HGETALL Book:1
1) "name"
2) "Infinite Jest"
3) "author"
4) "David Foster Wallace"
```

The default mode for Redis slaves is readonly, meaning that if we issue any Redis commands that change the value stored at a key, we'll receive the following:

```
127.0.0.1:6380> HSET Book:1 copyrightYear 1996
(error) READONLY You can't write against a read only slave.
```

SYNC, Partial Resynchronization & Diskless Replication

Redis Replication

Steps:

1. The master then starts background saving, and starts to buffer all new commands received that will modify the dataset.
2. When the background saving is complete, the master transfers the database file to the slave, which saves it on disk, and then loads it into memory.
3. The master will then send to the slave all buffered commands. This is done as a stream of commands and is structured in the Redis.

Partial Resynchronization

Replication is non-blocking on both master and slave instances; a master can continue to respond to queries while slaves are synchronizing, a slave can be configured to respond to queries with old data while synchronizing with the master

Introduction to Redis Cluster

In this topic ...

Redis Cluster provides one method of running a Redis installation where data is sharded across multiple Redis instances while providing a degree of availability for continuing operations in the event of partial node failure or communication failure.

This topics starts with an overview of a minimal Redis cluster with an demonstration of the composite hashing approach used by Redis

Partitioning

Composite Partitioning

Redis cluster uses a form of composite partitioning called consistent hashing that calculates what Redis instance any particular key is assigned called a **hash slot**.

The hash slot is the CRC16 hash algorithm applied to the key and then the computation of a modulo using 16384. The specific algorithm used by Redis cluster to calculate the hash slot for a key:

1. The cyclic redundancy check (CRC) using a polynomial length of 17-bits or CRC16 is calculated on the Redis key
2. The modulo 16384 of the value is then calculated to get the **hash slot** for the key.

Each master node is assigned a sub-range of hash slots and the key and value will reside on that Redis instance.

Cluster Topology & Redis Cluster Protocol

Mesh Network

Nodes in the Redis Cluster use the Redis cluster protocol to connect with every other node in the Redis cluster for a mesh network topology. Nodes communicate using a TCP gossip protocol and along with a configuration update mechanism in order to reduce the number of messages being exchanged between nodes.

When a Redis cluster is running, each node has two TCP sockets open:

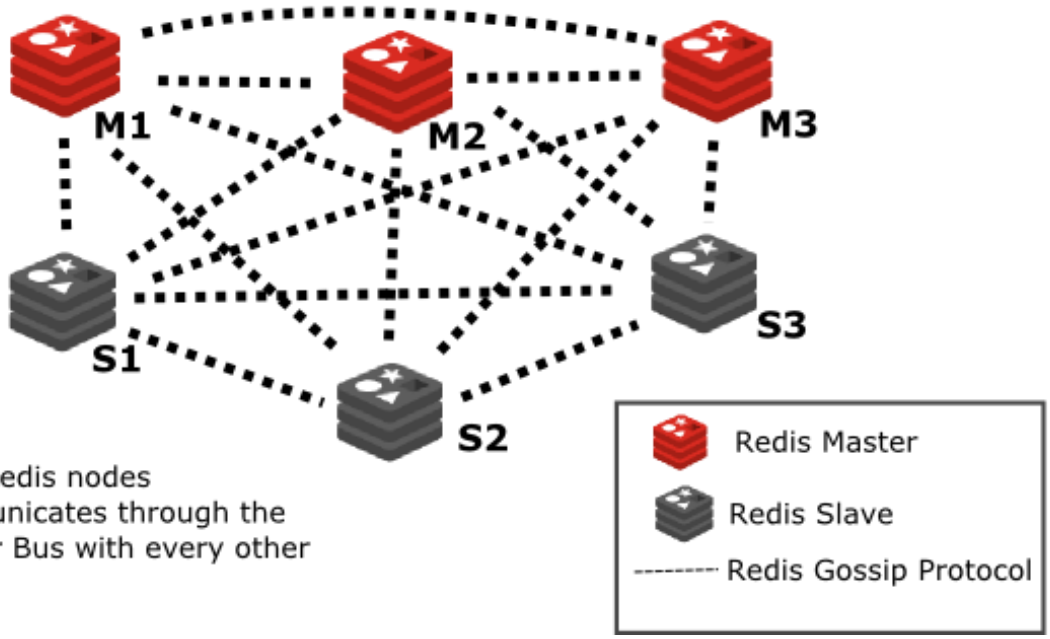
- First is the standard Redis protocol for connecting clients (default port 6379 for the first node)
- Second port is calculated from the sum of the first port plus 10000 (16379 for the default port) and binary protocol for node-to-node communication.

Clients should never need to connect directly with the cluster bus port but with the lower, standard port.

Six-Node Redis Cluster

Running Redis Cluster

In a minimal Redis Cluster made up 3 masters nodes each with a single slave node, each master node is assigned a hash slot range between 0 and 16,384. Both Master and Slaves Run two TCP services, the first is for normal RESP messages and the second is Cluster Bus that communicates with the Redis Cluster Gossip protocol.



Each Redis nodes communicates through the Cluster Bus with every other Node

To illustrate all of the features of Redis Cluster, the minimum recommended Redis cluster setup is to have a 3 Master Redis nodes with each master node replicated with a single Redis slave instance node. Each Master Node hash slots are broken down as

Master One (M1) - allocated hash slots 0-5460

Slave One (S1) replicates Master One, is promoted if a quorum of nodes cannot reach Master One.

Master Two (M2) - allocated hash slots 5461-10922

Pub/Sub

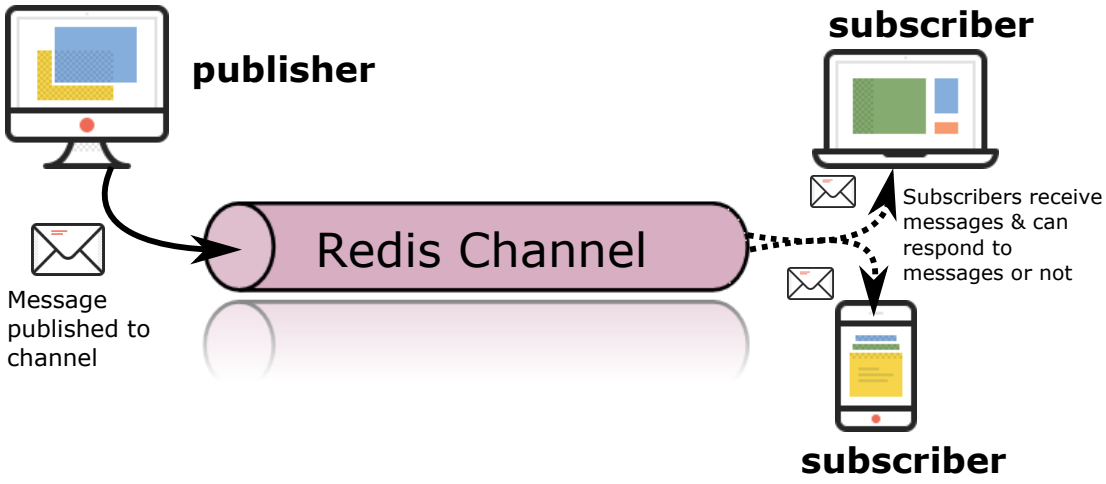
Redis is well suited to supporting the Publish/Subscribe (Pub/Sub) model where publisher applications sends messages to one or more channels that subscriber applications monitor and then react to the messages that posted to the channel.

An important limitation of Pub/Sub that does NOT necessary guarantee the delivery of messages i.e. Redis Pub/Sub is fire and forget. Messages published to a channel are not assured to be delivered to the client monitoring the channel by subscription.

In this topic ...

Pub/Sub Messaging Pattern

Redis Pub/Sub uses a message passing system that message senders - called **publishers** - post a message to a channel that the message receivers - called **subscribers** - can respond to messages without either the publishers or subscribers knowing any details about each other.



Commands for Pub/Sub

SUBSCRIBE *channel* [*channel* ...]
PUBLISH *channel message*
UNSUBSCRIBE [*channel* [*channel* ...]]
PSUBSCRIBE *pattern* [*pattern* ...]
PUNSUBSCRIBE[*channel* [*channel* ...]]

Basic Pub/Sub Operation

Running a single Redis instance, we launch two `redis-cli` sessions, the first client will have the publisher role and the second client will have the consumer role.

Publisher `redis-cli`

```
$ ../redis/src/redis-cli
127.0.0.1:6379> PUBLISH user:1:msgs "How is everything?"
(integer) 1
```

Consumer `redis-cli`

```
$ ../redis/src/redis-cli
127.0.0.1:6379> SUBSCRIBE user:1:msgs user:2:msgs
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "user:1:msgs"
3) (integer) 1
1) "subscribe"
2) "user:2:msgs"
3) (integer) 2
1) "message"
2) "user:1:msgs"
3) "How is everything?"
```

Pub/Sub Message Syntax

Redis Pub/Sub messages are defined as 3 or 4 part RESP (REdis Serialization Protocol) array. RESP is the communication protocol that Redis client communicate with Redis server and is used in other projects beside Redis. We see the raw RESP protocol if we use telnet and connect directly to Redis.

First, we will connect to a single Redis instance running at port 6379 with telnet.

```
$ telnet localhost 6379
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
```

Now, we will subscribe to a **chat_room** channel

and from a `redis-cli` session, we will post a message to the channel.

```
SUBSCRIBE chat_room
*3
$9
subscribe
$9
chat_room
:1
*3
$7
message
$9
chat_room
$18
Anybody out there?
```

Pattern Matching with **PSUBSCRIBE & PUNSUBSCRIBE**

To subscribe to more than one channel, you can use the **PSUBSCRIBE** that take a pattern to match to existing channels.

```
PSUBSCRIBE chat_*
*3
$10
psubscribe
$6
chat_*
:1
*4
$8
pmessage
$6
chat_*
$9
chat_room
$18
What is your name?
```

To unsubscribe to multiple channels at once, you can use the **PUNSUBSCRIBE**

```
PUNSUBSCRIBE chat*
*3
$12
punsubscribe
$5
chat*
:1
```

Pub/Sub with Redis Cluster

To experiment with Pub/Sub and Redis Cluster, we will use a Python Redis Cluster for a simple client consumer.

```
>>> from rediscluster import StrictRedisCluster
>>> startup_nodes = [{"host": "127.0.0.1", "port": "7001"}]
>>> messenger_cluster = StrictRedisCluster(startup_nodes=startu
                                           decode_response=True)
>>> person_consumer = messenger_cluster.pubsub()
>>> person_consumer.subscribe("chat_room")
>>> person_consumer.get_message()
{'channel': 'chat_room', 'type': 'message', 'data': "What's hap
```

Slave Two (S2) replicates Master Two, is promoted if a quorum of nodes cannot reach Master Two.

Master Three (M3) - allocated hash slots 10923-16383

Slave Three (S3) replicates Master Three, is promoted if a quorum of nodes cannot reach Master Three.

Manually Running a Redis Cluster

We'll start by manually create a minimal cluster with three master nodes and three slaves.

```
$ mkdir cluster-test
$ cd cluster-test
$ mkdir 7000 7001 7002 7003 7004 7005
$ touch 7000/redis.conf
$ vi 7000/redis.conf
$ cp 7000/redis.conf 7001/.
$ vi 7001/redis.conf
...
$ cp ~/redis/src/redis-server .
$ cd 7000
$ ~/redis/src/redis-server redis.conf
# New terminal tab
$ cd ../7001
$ ~/redis/src/redis-server redis.conf
...
$ ~/redis/src/redis-server redis.conf
```

Install the Ruby Redis gem and run the `redis-trib.rb`

```
$ sudo gem install redis
$ cd ../../
$ .~/redis/src/redis-trib.rb create --replicas 1 127.0.0.1:7000
127.0.0.1:7002 127.0.0.1:7003 127.0.0.1:7004 127.0.0.1:7005
```

Sample **redis.conf**

Here is the example `redis.conf` we are using for each of Redis cluster nodes

```
port 7000
cluster-enabled yes
cluster-config-file nodes.conf
cluster-node-timeout 5000
appendonly yes
```

We need to change the port number to the correct value for the node, i.e. for node 7002, we need specify the correct port number for that node's `redis.conf` configuration file.

Easier Redis Cluster with *create-cluster* script

Instead of manually setting up and running a Redis cluster, you can instead use the *create-cluster* located in the `redis/utils/create-cluster`.

Starting in the *create-cluster* directory, follow these steps to get a 6-node Redis Cluster with 3 masters and 3 slaves.

1. Create a `config.sh` script:

```
#!/bin/bash
PORT=7000
TIMEOUT=2000
NODES=6
REPLICAS=1
```

2. Start Redis Cluster:

```
~/redis/utils/create-cluster$ ./create-cluster
Starting 7001
Starting 7002
Starting 7003
Starting 7004
Starting 7005
Starting 7006
```

3. Link and create the Redis Cluster

```
>>> Creating cluster
>>> Performing hash slots allocation on 6 nodes
Using 3 masters:
127.0.0.1:7001
127.0.0.1:7002
127.0.0.1:7003
Adding replica 127.0.0.1:7004 to 127.0.0.1:7001
Adding replica 127.0.0.1:7005 to 127.0.0.1:7002
Adding replica 127.0.0.1:7006 to 127.0.0.1:7003
M: 1379949e7d8eaa27a0634285e521079eccc0cc1f 127
   slots:0-5460 (5461 slots) master
M: 470bf4397e0002f211df09dadcd5ec12b458e9c3 127
   slots:5461-10922 (5462 slots) master
M: 7e343391d165ccee34e0f1cf43590270130a9d5b 127
   slots:10923-16383 (5461 slots) master
S: 3d8c532367f0846f292b538d09b7cafdc6b3c6b9 127
   replicates 1379949e7d8eaa27a0634285e521079ec
S: 491abeb14973c0c9495f1b045b4e5d3f0729bcc8 127
   replicates 470bf4397e0002f211df09dadcd5ec12b
S: 933258e1d5ed8752c7e4ff7ce377dfd63543977f 127
   replicates 7e343391d165ccee34e0f1cf435902701
Can I set the above configuration? (type 'yes'
>>> Nodes configuration updated
>>> Assign a different config epoch to each nod
>>> Sending CLUSTER MEET messages to join the c
Waiting for the cluster to join..
>>> Performing Cluster Check (using node 127.0.
M: 1379949e7d8eaa27a0634285e521079eccc0cc1f 127
   slots:0-5460 (5461 slots) master
M: 470bf4397e0002f211df09dadcd5ec12b458e9c3 127
   slots:5461-10922 (5462 slots) master
M: 7e343391d165ccee34e0f1cf43590270130a9d5b 127
   slots:10923-16383 (5461 slots) master
M: 3d8c532367f0846f292b538d09b7cafdc6b3c6b9 127
```

```
   slots: (0 slots) master
   replicates 1379949e7d8eaa27a0634285e521079ec
M: 491abeb14973c0c9495f1b045b4e5d3f0729bcc8 127
   slots: (0 slots) master
   replicates 470bf4397e0002f211df09dadcd5ec12b
M: 933258e1d5ed8752c7e4ff7ce377dfd63543977f 127
   slots: (0 slots) master
   replicates 7e343391d165ccee34e0f1cf435902701
[OK] All nodes agree about slots configuration.
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered.
```

4. Stopping Redis Cluster

```
$ ./create-cluster stop
Stopping 7001
Stopping 7002
Stopping 7003
Stopping 7004
Stopping 7005
Stopping 7006
```

Using Redis Cluster with `redis-cli`

After successfully launching your Redis Cluster, an easy way to interact with it is to use the default Redis command-line client, `redis-cli` in cluster mode.

If the link between the master and slave goes down, the old behavior of Redis is to do a full synchronization with the slave when the connection is re-established. As of Redis 2.8 experimental support was added with **PSYNC** command that will only do a partial synchronization with MASTER based on if a time threshold has been exceeded or not. If time has been exceeded, a full **SYNC** is triggered between the master and slave.

Diskless replication

An experimental feature that sends the full RDB file to the slave without forking the master and using the disk as an intermediate storage.

Critical Commands for Master-Slave Replication

SLAVEOF ***server port***
TYPE ***key***
CLIENT LIST

Caveats when using Replication

If persistence is turned off on the master A and two slaves B,C the following problem can occur:

- We have a setup with node A acting as master, with persistence turned down, and nodes B and C replicating from node A.
- A crashes, however it has some auto-restart system, that restarts the process. However since persistence is turned off, the node restarts with an empty data set.
- Nodes B and C will replicate from A, which is empty, so they'll effectively destroy their copy of the data.

Configuring Master-Slave Replication

Master-Slave replication can be set-up in the **redis.conf** file for each slave.

```
# Master-Slave replication. Use slaveof to make a Redis instanc
# another Redis server.
slaveof
```

Runtime

```
127.0.0.1:6379> SLAVE OF localhost 6379
```

Use Case: *Linked Data Fragments Server*

High Availability with Redis Sentinel

In this topic …

From the Redis Sentinel documentation, Redis Sentinel manages Redis instances through four main ways:

- **Monitors** running Redis masters and slaves
- **Notifies** system administrators or monitoring software of problems with Redis instances
- **Automatic failover** by promoting running slaves to master if a master instance is experiencing problems
- **Configuration provider** for clients to connect and discover the address for a given Redis master.

Configuring `sentinel.conf`

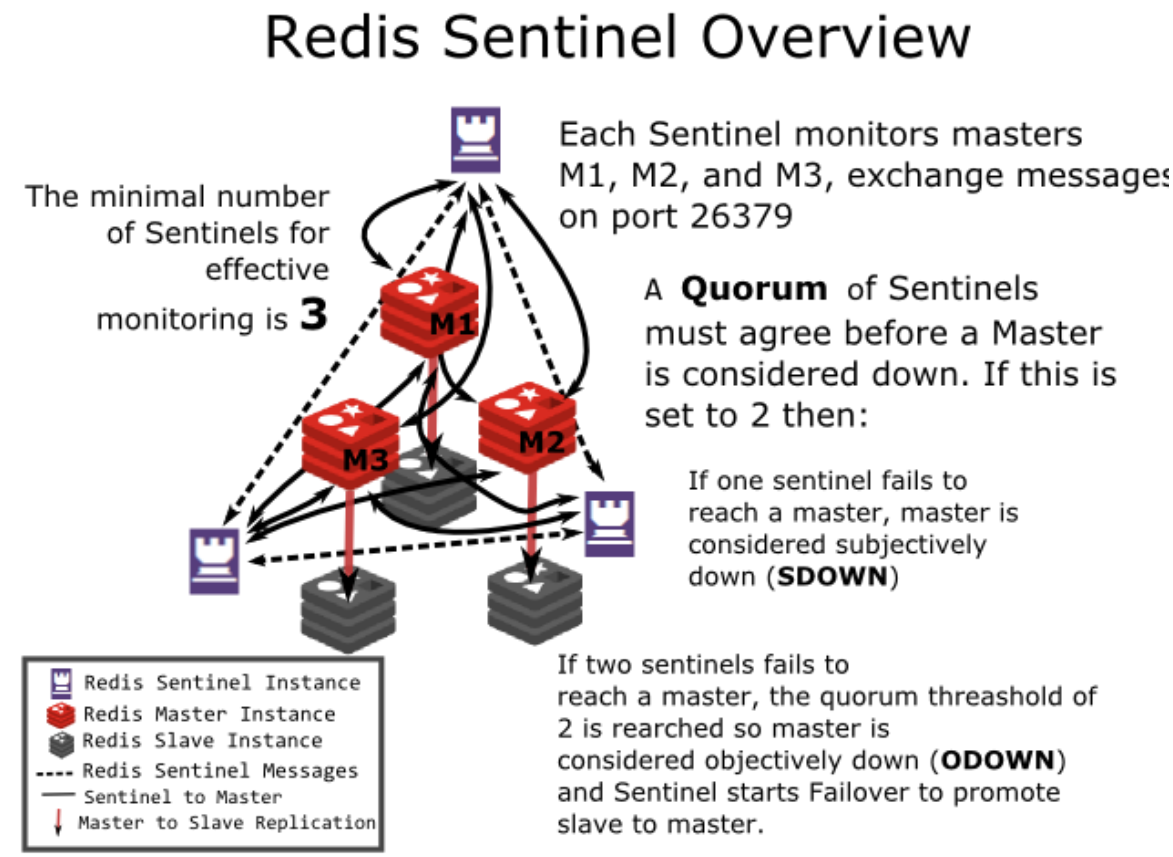
Notes about the **quorum** setting

In the `sentinel.conf` file, the **sentinel monitor** configuration directive has a *quorum* setting.

Redis Sentinel Startup

Running Redis Sentinel can be accomplished either with the `redis-sentinel` binary or by passing the **--sentinel** parameter when running `redis-server`. Both options require a `sentinel.conf` file.

Redis Sentinel requires an open TCP port 26279



Notifications & Redis Sentinel

Here is the relevant section in `sentinel.conf`

Security

Security improvements in the latest versions of Redis offer better ways to secure a Redis instance.

In this topic ...

Overview

Redis Default Security Settings

One of the biggest changes in Redis 3.2 version is how Redis's default security is handled when running Redis server. The new Redis Protected mode is a layer of security protection, to avoid that Redis instances left open on the Internet are accessed and exploited. This mode is active when the following occurs:

1. The server is not binding explicitly to a set of addresses using the "bind" directive.
2. No password is configured.Before, the only way to restrict access to a Redis instance was to set a password

Configuring Redis's Protected Mode

Manually with `redis-cli`

```
127.0.0.1:6379> CONFIG SET protected-mode no OK
```

At startup *redis.conf*

Redis does have a small level of authentication which can be set in the `redis.conf` with the **requirepass** configuration directive. Clients connecting need to first send an **AUTH** command followed by a password.

For slaves to connect to a master instance that has a password, the **masterauth** configuration directive is used to store the master's password for the slave to authenticate.

```
    requirepadd my-great-Password
```

Disabling of Commands

Redis allows you to disable specific commands that are dangerous in the wrong hands i.e. **FLUSHALL** or **CONFIG** that you may want to disable. In the `redis.conf` using the *rename-command*, you can either rename the command to a hard-to-guess string or completely disable completely by setting it to an empty string.

```
rename-command CONFIG thisIsmynewSpecialConfigCommandName
rename-command FLUSHALL ""
```

```
# SCRIPTS EXECUTION
#
# sentinel notification-script and sentinel reconfig-script are
# to configure scripts that are called to notify the system adm
# or to reconfigure clients after a failover. The scripts are e
# with the following rules for error handling:
#
# If script exits with "1" the execution is retried later (up t
# number of times currently set to 10).
#
# If script exits with "2" (or an higher value) the script exec
# not retried.
#
# If script terminates because it receives a signal the behavio
# as exit code 1.
#
# A script has a maximum running time of 60 seconds. After this
# reached the script is terminated with a SIGKILL and the execu

# NOTIFICATION SCRIPT
#
# sentinel notification-script
#
# Call the specified notification script for any sentinel event
# generated in the WARNING level (for instance -sdown, -odown,
# This script should notify the system administrator via email,
# other messaging system, that there is something wrong with th
# Redis systems.
#
# The script is called with just two arguments: the first is th
# and the second the event description.
#
# The script must exist and be executable in order for sentinel
# this option is provided.
#
# Example:
#
# sentinel notification-script mymaster /var/redis/notify.sh
```

Here is an example of a notitification script

Failover with Redis Sentinel

a

Here is the relevant section in `sentinel.conf`

```
# sentinel failover-timeout
#
# Specifies the failover timeout in milliseconds. It is used in
#
# - The time needed to re-start a failover after a previous fai
#   already tried against the same master by a given Sentinel,
#   times the failover timeout.
#
# - The time needed for a slave replicating to a wrong master a
#   to a Sentinel current configuration, to be forced to replic
#   with the right master, is exactly the failover timeout (cou
#   the moment a Sentinel detected the misconfiguration).
#
# - The time needed to cancel a failover that is already in pro
#   did not produced any configuration change (SLAVEOF NO ONE y
#   acknowledged by the promoted slave).
#
# - The maximum time a failover in progress waits for all the s
#   reconfigured as slaves of the new master. However even afte
#   the slaves will be reconfigured by the Sentinels anyway, bu
#   the exact parallel-syncs progression as specified.
#
# Default is 3 minutes.
sentinel failover-timeout mymaster 180000
```

Automatic Configuration with Redis Sentinel

```
# CLIENTS RECONFIGURATION SCRIPT
#
# sentinel client-reconfig-script
#
# When the master changed because of a failover a script can be
# order to perform application-specific tasks to notify the cli
# configuration has changed and the master is at a different ad
#
# The following arguments are passed to the script:
#
#
#
#  is currently always "failover"
#  is either "leader" or "observer"
#
# The arguments from-ip, from-port, to-ip, to-port are used to
# the old address of the master and the new address of the elec
# (now a master).
#
# This script should be resistant to multiple invocations.
#
# Example:
#
# sentinel client-reconfig-script mymaster /var/redis/reconfig.
```

Use Case: *Linked Data Fragments Server*

Lua Scripting

In this topic ...

Since Redis version 2.6, a Lua interpreter has been embedded into the Redis server. With Lua, a high-level programming language, functions and business-logic can be run directly in Redis for improved application performance.

Introducing Lua

Lua stores data in variables, that are declared and initialized by specifying the variable name and using the assignment operator (**=**) to assign a value. Lua core datatypes are nil (null), number (combines both int and float), boolean, and strings.

```
> x = 5
> x
5
> blank = nil
> blank
nil
> is_true, is_false = true, false
> is_true
true
> is_false
false
> message = "Redis with Lua is cool."
> message
Redis with Lua is cool.
>
```

A table in Lua is similar to the hash type in Redis, where a unique set of keys are mapped to a set of values. To retrieve a value from a table by key, either use square brackets **[]** or use the **.** operator if the key is a valid Lua identifier.

```
> table1 = {}
> table1[1] = "The first value in table1"
> table1["second"] = "The second value in table1"
> table1
table: 0000000000620900
> table1[1]
The first value in table1
> table1.second
The second value in table1
>
```

The standard mathematical operations are `multiplication *`, `division /`, `addition +`, `subtraction -`.

```
> 3*5
15
> 12/2
6.0
> 896-785
111
> 129+745
874
> (2+3) * 5
25
>
```

Lua strings can concatenated using two periods; `..`. To create a Lua multi-line string without any character escaping, use double square brackets at the beginning `[[` and at the end `]]`.

```
> message = "Redis is SOOO fast"
> message
Redis is SOOO fast
> readme = [[The way to a good a project is
>> good documentation and plenty of coffee]]
> readme
The way to a good a project is
good documentation and plenty of coffee
>
```

Lua has the standard if/then conditional branching as well as various looping structures for tables. The `do while` and `repeat until` are similar looping structures with main difference being that the `repeat until` will execute at least once per loop. Lua has two types of `for` loops; the first takes a start value, end value, and increment while the second `for` type takes an iterator over a table and iterates through each key-value pair in the table.

```
> x = 3.14
> if x > 10 then
>>    print("X is greater than 10")
>> elseif x == 2 then
>>    print("X is 2")
>> else
>>    print("X is "..x)
>> end
X is 3.14
>
> table2 = {20, 5, 60}
> sum, counter = 0, 1
> #table2
3
> while counter <= #table2 do
>>     sum = sum + table2[counter]
>>     counter = counter + 1
>> end
> print("The sum is ", sum)
The sum is      85
> sum, counter = 0, 1
> repeat
>>     sum = sum + table2[counter]
>>     counter = counter + 1
>> until counter > #table2
> print("The sum is: ", sum)
The sum is:     85
> sum = 0
> for counter = 1, #table2, 1 do
>>     sum = sum + table2[counter]
>> end
> print("The sum is: ", sum)
The sum is:     85
> for counter, num in ipairs(table2) do
>>     sum = sum + num
>> end
> print("The sum is: ", sum)
The sum is:     170
>
```

To create a function in Lua, use the `function` keyword.

```
> crazy_eight = function(a_table)
>>     sum = 0
>>     for counter =1, #a_table, 1 do
>>         sum = sum + a_table[counter]*8
>>     end
>>     return sum
>> end
> crazy_eight(table2)
680
```

Commands for *Lua* Scripting

EVAL ***numkeys key [key ...] arg [arg ...]***

EVALSHA ***sha1 numkeys key*** **[key ...] arg [arg ...]**

Exercise: First Lua Script

Lua scripts can be run using the EVAL and EVALSHA commands. Within the Lua script, Redis commands can be run using the `redis.call` and `redis.pcall` Lua functions. The difference between the two is that if an error occurs evaluating the `redis.pcall`, Lua will capture and return the error while `redis.call` will raise a Lua error and the EVAL command will also return an error.

```
127.0.0.1:6379> EVAL "return {KEYS[1],KEYS[2],ARGV[1],ARGV[2]}"
1) "Book:1"
2) "Book:2"
3) "War and Peace"
4) "Critique of Pure Reason"
127.0.0.1:6379> EVAL "return redis.call('SET', 'Book:1', 'War a
OK
127.0.0.1:6379> GET Book:1
"War and Peace"
```

The EVALSHA command's first argument it is the SHA1 digest script instead of the script itself. Use the SCRIPT LOAD command to load script first and then use the resulting SHA1 to run script on the server

```
127.0.0.1:6379> SCRIPT LOAD "return {KEYS[1],KEYS[2],ARGV[1],AR
"a42059b356c875f0717db19a51f6aaca9ae659ea"
127.0.0.1:6379> EVALSHA a42059b356c875f0717db19a51f6aaca9ae659e
1) "Book:1"
2) "Book:2"
3) "War and Peace"
4) "Critique of Pure Reason"
```

Use Case **Tracking Participants in a Chat Room**

Examples of a Lua script that takes a Room UUID and emails of users in a chat room and creates a Set following a simple Redis Key Schema.

Download

```
local room_key = "Room:"..KEYS[1]
for pos, email in ipairs(ARGV) do
  redis.pcall("SADD", room_key, email)
end
return True
```

Caching & Keyspace Notifications

Redis provides a number of Less Recently Used (LRU) algorithms in addition to its standard time-based expiration functionality that are set on Redis keys.

Supporting post-processing when a Redis key is ejected from the Redis datastore, you can subscribe to specific events using special PUB/SUB mode in Redis.

In this topic ...

Default Expiration

As an in-memory datastore, Redis behavior when it is running out of memory can be adjusted depending on the needs of the application. Redis's default policy when reaching the maximum

memory available is **noeviction** that raises an Out-of-memory OOM error.

One strategy to avoid OOM, is to set **EXPIRE** on a key that will be evicted from the datastore when it's time is up. Using the **TTL** command gives the remaining time before a key is evicted, while the **PERSIST** command clears an existing expiration on a key.

EXPIRE ***key seconds***
EXPIREAT ***key timestamp***
INFO [***section***]
TTL ***key***
PERSIST ***key***

Setting a memory ceiling for your running Redis instance is accomplished by either setting the maxmemory directive either in `redis.conf` or during runtime.

```
27.0.0.1:6379> CONFIG GET maxmemory
1) "maxmemory"
2) "1048576"
127.0.0.1:6379> CONFIG GET maxmemory-policy
1) "maxmemory-policy"
2) "noeviction"
127.0.0.1:6379> CONFIG SET maxmemory 1mb
OK
```

We'll now create a Python function that adds data until the datastore is full

```
>>> import uuid
>>> def add_id(redis_instance):
        redis_key = "uuid:{}".format(redis_instance.incr("globa
        redis_instance.set(redis_key, uuid.uuid4())
```

Setting an expiration, polling time remaining, and clearing an existing expiration

```
127.0.0.1:6379> SADD authors "David Foster Wallace" "James Clav
(integer) 3
127.0.0.1:6379> EXPIRE authors 180
(integer) 1
127.0.0.1:6379> TTL authors
(integer) 173
127.0.0.1:6379> TTL authors
(integer) 146
127.0.0.1:6379> PERSIST authors
(integer) 1
127.0.0.1:6379> TTL authors
(integer) -1
127.0.0.1:6379> EXPIRE authors 10
(integer) 1
127.0.0.1:6379> TTL authors
(integer) -2
```

Redis LRU Caching Options

Redis offers a number of different LRU (less recently used) caching options to better handle OOM cases in your running Redis instance.

The Redis LRU algorithm is not exact, as Redis does not automatically choose the best candidate key for eviction, the least used key, or the key with the earliest access date. Instead, Redis default behavior is take a sample of five keys and evict the least used of those five keys. If we want to increase the accuracy of our LRU algorithm, we can the change the maxmemory-samples directive in either redis.conf or during runtime with the CONFIG SET maxmemory-samples command. Increasing the sample size to 10 improves the performance of the Redis LRU so that it approaches a true LRU algorithm but with the side-effect of more CPU computation. Decreasing the sample size to 3 reduces the accuracy of Redis LRU but with a corresponding increase in processing speed.

volatile LRU Policy Redis keys are evicted if the keys have **EXPIRE** set, if there not any keys to be evicted when Redis reaches maxmemory an OOM error is returned to the client. Note: under this policy when Redis reached maxmemory, it will start evicting keys that have an expiration set even if the time limit on keys hasn't been reached yet.

Testing volatile LRU

```
127.0.0.1:6379> FLUSHDB
127.0.0.1:6379> CONFIG SET maxmemory-policy volatile-lru
```

Python function to add and set keys

```
>>> def add_id_expire(redis_instance):
        count = redis_instance.incr("global:uuid")
        redis_key = "uuid:{}".format(count)
        redis_instance.set(redis_key, uuid.uuid4())
        if count <= 75:
            redis_instance.expire(redis_key, 300)
```

The **allkeys-lru** evicts keys based on the ttl. The allkeys-lru can delete ANY key in Redis and there is no way to restrict which keys are to be deleted. If you application needs to persist some Redis keys (say for configuration or reference look-up) DON'T use allkeys-lru policy!

Testing allkeys-lru

```
127.0.0.1:6379> FLUSHDB
127.0.0.1:6379> CONFIG SET maxmemory-policy allkeys-lru
```

< Running the add_id function in an infinite while loop and a counter

```
>>> count = 1
>>> while 1:
        add_id(local_redis)
        count += 1
```

Using the **INFO stats** will show us the status of Redis cache.

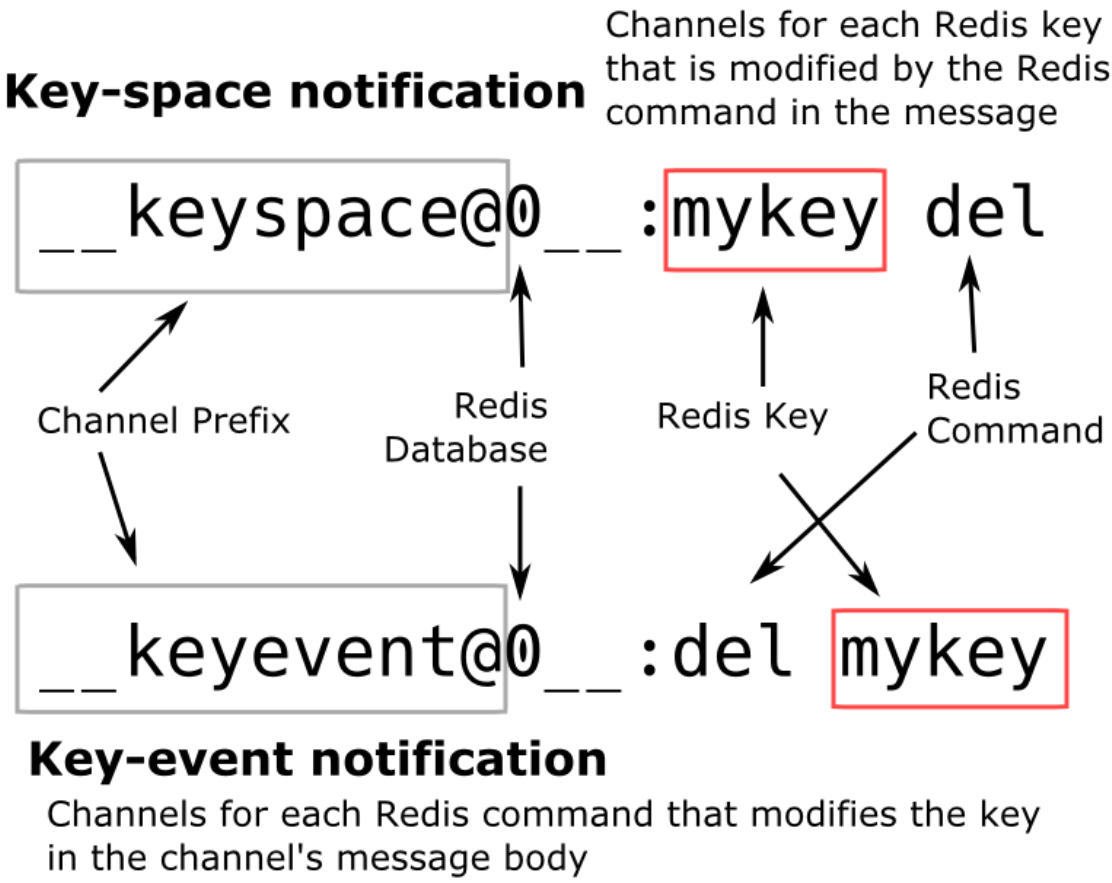
Redis offer two other types of non-LRU maxmemory policies, **volatile-random** and **allkeys-random** mirror the previous polices but instead of calculating LRU of the keys, the keys are either randomly evicted based on the key's TTL in the case of the **volatile-random** or any random keys in the case of **allkeys-random**.

Keyspace Notifications Overview

The ability for an application to respond to changes that may occur to the value stored at a particular key or keys. Redis provides a mechanism for client code to subscribe to a Pub/Sub channel that monitors events related to data. Called **keyspace notification**, functionality for monitoring events like all the commands that change a given key, all keys receiving specific commands like HSET, or all keys that are about deleted because of an EXPIRE command.

Redis Keyspace Notification

Since Redis 2.8, enabling the **`notify-keyspace-events`** configuration directive will publish two types of events that impact the Redis data space to special Pub/Sub channels that formated like this:



Troubleshooting Redis Latency

In this topic …

Latency, as understood in the Redis community, is broken down into three ways:

1. **command latency** , is the amount of time it takes to execute a command. Some commands are fast and operate in O(1) while other commands have O(n) time complexity and are thereby a likely source of this type of latency.
2. **round-trip latency** the time between when a client issues a command and then receives the response from the Redis server that can be caused by network congestion.
3. **client-latency** if multiple clients attempt to connect to Redis at the same time, concurrency latency can be introduced as later clients may be waiting in queue for early client processes to complete.

   To help troubleshoot, Redis has a special mode for monitoring command latency that can be set in either `redis.conf` or from issuing a CONFIG SET for the latency-monitor-threshold directive.

   You can quickly run an latency check with `redis-cli`

```
redis-cli --latency -h {host} -p {port}
```

Latency Monitoring

The Redis **latency-monitor-threshold** directive sets a limit in milliseconds that will log all or some of the commands and activity (called events) of the Redis instance that exceed that limit with a default of 0, meaning Redis does not automatically run latency monitoring but must be actively set.

First, we'll set our threshold to 100 milliseconds

```
127.0.0.1:6379> CONFIG SET latency-monitor-threshold 100
```

We'll run a series of **DEBUG SLEEP** to demonstrate the various subcommands and functionality of Redis's latency monitor.

```
127.0.0.1:6379> DEBUG SLEEP 1
127.0.0.1:6379> DEBUG SLEEP .25
127.0.0.1:6379> LATENCY LATEST
1) 1) "command"
   2) (integer) 1433877394
   3) (integer) 250
   4) (integer) 1000
```

The **LATENCY LATEST** the event name, the UNIX timestamp when the latency event occurred, the latest event latency in milliseconds, and the all-time maximum latency for this event.

The **LATENCY HISTORY** command and subcommand returns the latest 160 latency events that are being tracked

The **LATENCY RESET** either clear all latency events or just selected events by passing in one or more event names.

The **LATENCY GRAPH** command produces an ASCII-style graph of the logged latency events since the last **LATENCY RESET** command.

The **LATENCY DOCTOR** mode provides a rich set of human-readable (with flashes of HAL 9000 from Stanley Kubrick's film 2001!) statistical data such as average time between latency spikes, median deviations of those spikes as well as human understandable analysis of the latency events and suggestions for reducing latency.

```
127.0.0.1:6379> latency doctor
Dave, I have observed latency spikes in this Redis instance. Yo

1. command: 9 latency spikes (average 595ms, mean deviation 261

I have a few advices for you:

- Check your Slow Log to understand what are the commands you a
- Deleting, expiring or evicting (because of maxmemory policy)
127.0.0.1:6379>
```

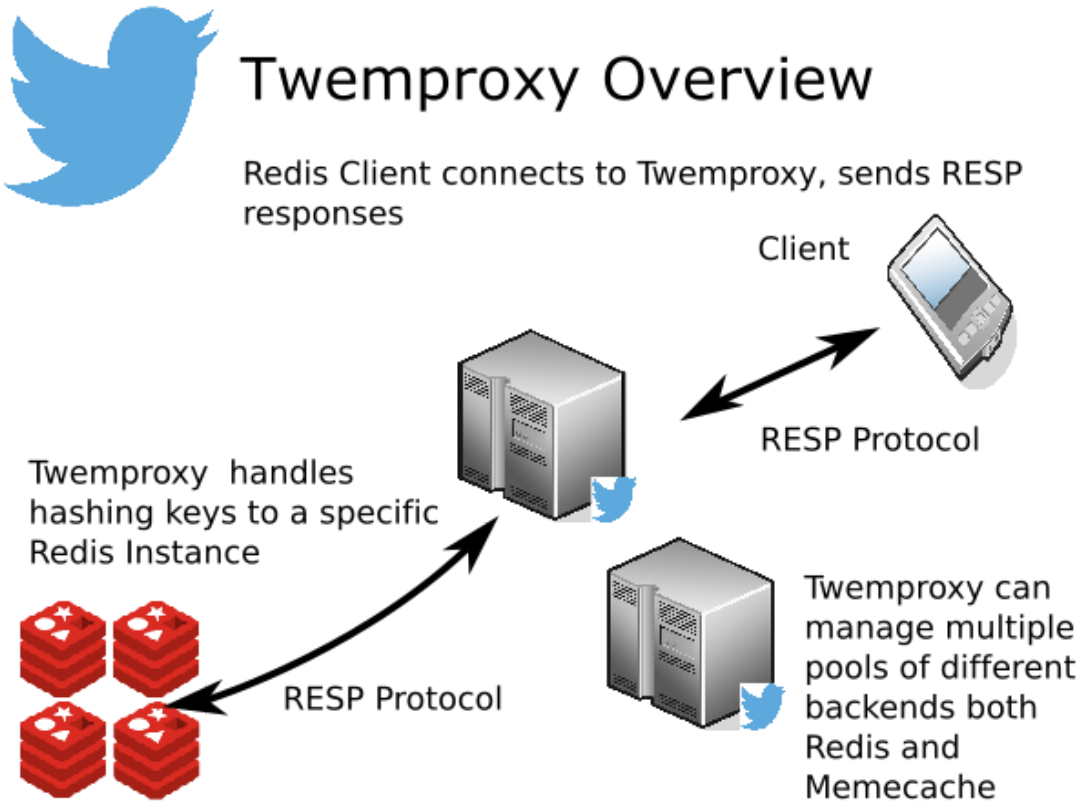
Related Technologies

Redis-related Technologies like Salvidore's own Discue provide alternative ways for implementing such things as cluster and messaging using Redis or Redis-related standards.

In this topic …

Twemproxy
https://github.com/twitter/twemproxy

Twemproxy is an open-source project released by Twitter for creating a caching proxy between a client and backend made up of either Memecache or Redis instances. Twemproxy separates the client calls from the datastore backend through the use of an intermediary middleware that then implements a sharding strategy based on your preferences that are set in a configuration YAML file. Twemproxy supports a twelve different hash functions including md5, crc16, two versions of crc32, four variants of the Fowler-Noll-Vo (fnv), among others with the default being a fnv1a_64 hash functions.



Dynomite
https://github.com/Netflix/dynomite

Netflix sponsors Dynomite, is to be able to implement high availability and cross-datacenter replication on storage engines that do not inherently provide that functionality.

Disque
https://github.com/antirez/disque

Disque is a distributed message broker created by Salvatore Sanfilippo, the creator of Redis. Disque is a message broker with at-least-once and at-most-once delivery semantics.

Help & Downloads

This website's source code is available at https://github.com/jermnelson/focused-redis-topics.

Day 1 Topic 1 - Redis data types

- Download Redis, 3.2 Release Candidate
- Introduction to Redis Datatypes on Redis website.

Day 1 Topic 2 - Key Schemas and Object Relational Managers

- Node.js Object Relationship Mapper: Nohm

Day 1 Topic 3 - Master-Slave Replication

- Redis Official documentation on replication

Day 1 Topic 4 - Introduction to Redis Cluster

- Redis Official documentation on Cluster Tutorial and Redis Cluster Specficiation

Day 2 Topic 5 - Pub/Sub

- Redis Official documentation on Pub/Sub
- Redis Protocol Specification (RESP)

Day 2 Topic 6 - High Availability with Redis Sentinel

- Redis Official documentation on Sentinel

Day 2 Topic 7 - Security

- Redis Official documentation on Master/Slave Replication

Day 2 Topic 8 - Lua Scripting

- From the redis.io website: topic on EVAL
- Lua Language homepage
- Download add-chat-room-users.lua

Day 3 Topic 9 - Caching & Keyspace Notifications

- Redis Official documentation on LRU Cache
- Redis Official documentation on Keyspace Notifications

Day 3 Topic 10 - Troubleshooting Redis Latency

- Redis Official documentation on Lacency Monitoring
- Redis View Dashboard documentation

Day 3 Topic 11 - Related Technologies

- Twemproxy
- Dynomite
- Disque