redis

Fundamentals

Ali KATKAR

Contents

Why Redis?

- *Redis can perform*
  - *>100k+ SETs per second,*
  - *>80k+ GETs per second.*
- (You're lucky to get over 6kt/s from MySQL.)

redis

Overview of Caching

Ali KATKAR

What is Cache?

A cache (/ˈkæʃ/ kash) is a component that transparently stores data so that future requests for that data can be served faster.

Types of caches

- Disk drive cache
  - *Mechanical / component*
- Browser cache
  - *client-side / local*
- Server cache
  - *Server-side / remote*

Caching Methods

- SQL Cache
  - *Run queries less often*
- Cached files on disk
  - *Process code less frequently*
- In-Memory Cache
  - *Store frequently-used data*

PHP Caching Software

- Smarty Templates
  - *Creates parsed files on disk*
- WordPress Caching Plugins
  - *W3 Total Cache, SuperCache*
- MVC Frameworks
  - *Symfony, Kohana, CodeIgniter, Zend*

PHP Caching Extensions

- **Memcache**
  - *Key-value store, in memory (volatile)*
- **APC:** The Alternative PHP Cache
  - *Key-value store (memory) & opcode cache (disk)*
- **Redis: Re**mote **Di**ctionary **S**erver

  Data store, in memory (disk-backed)

Introduction to Redis

Ali KATKAR

What is Redis?

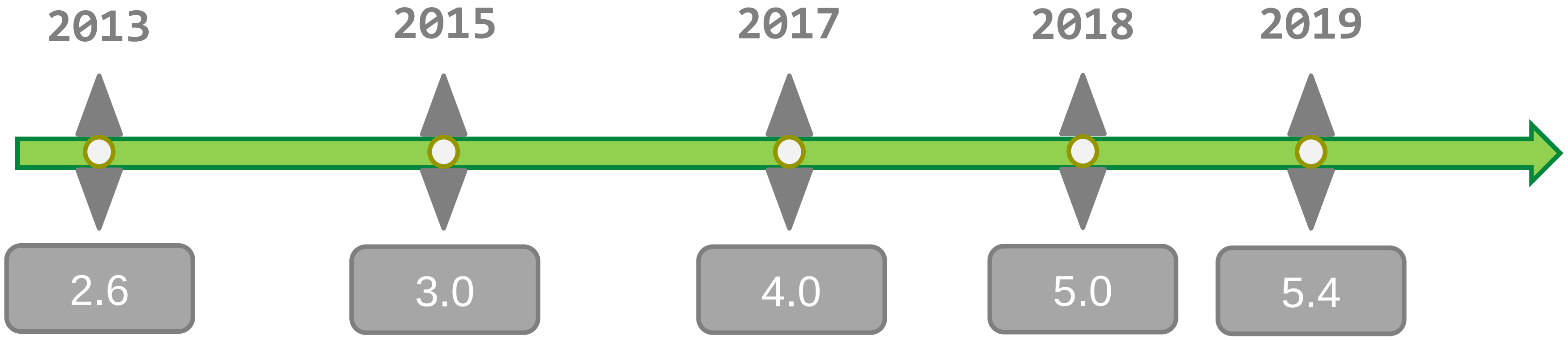
- Redis is an open source, **advanced key-value store**.
- It is often referred to as a data structure server
  - because keys can contain strings, hashes, lists, sets and sorted sets.

What is Redis?

- Originally written by Salvatore Sanfilippo,
- He now works full-time on Redis,
- sponsored by VMWare.

- @antirez on Twitter.

Redis Version History



Ali Katkar

What is Redis?

- Redis is an open source (BSD licensed), in-memory **data structure store**
- Used as a database, cache and message broker.
- It supports data structures such as
  - strings
  - hashes
  - lists
  - sets
  - sorted sets with range queries,
  - bitmaps
  - hyperloglogs
  - streams

Redis Properties

- Redis has built-in
  - Replication
  - Lua scripting
  - LRU eviction
  - Transactions
  - different levels of on-disk persistence
- Redis provides high availability via
  - Redis Sentinel
  - Automatic partitioning with Redis Cluster.

Atomic Operations

- You can run **atomic operations** on these types
  - appending to a string
  - incrementing the value in a hash
  - pushing an element to a list
  - computing set intersection
  - union and difference
  - getting the member with highest ranking in a sorted set.

Performance

- Redis works with an **in-memory dataset**.
  - So it achieves outstanding performance
- Depending on your use case, you can persist it either by
  - dumping the dataset to disk every once in a while,
  - appending each command to a log.
- Persistence can be optionally disabled,
  - if you just need a feature-rich, networked, in-memory cache.

Replication

- Redis supports
  - master-slave asynchronous replication
    - with very fast non-blocking first synchronization,
  - auto-reconnection with partial re-synchronization on net split.

Other Features

- Transactions
- Pub/Sub
- Lua scripting
- Keys with a limited time-to-live
- LRU eviction of keys
- Automatic failover

Redis Clients

- You can use Redis from most programming languages out there.
- Redis is written in **ANSI C** and works in most POSIX systems like
  - Linux, *BSD, OS X without external dependencies.
  - Linux and OS X are the two operating systems where Redis is developed and more tested,
    - **Using Linux for deploying is recommended**.
  - Redis may work in Solaris-derived systems like SmartOS, but the support is *best effort*.
  - There is no official support for Windows builds, but Microsoft develops and maintains a Win-64 port of Redis.

Common Use Cases

- **Caching**
  - Due to its high performance, developers have turned to Redis when the volume of read and write operations exceed the capabilities of traditional databases.
  - With Redis's capability to easily persist the data to disk, it is a superior alternative to the traditional memcached solution for caching.
- **Publish and Subscribe**
  - Since version 2.0, Redis provides the capability to distribute data utilizing the Publish /Subscribe messaging paradigm.
  - Some organizations have moved to Redis and away from other message queuing systems (i.e., RabbitMQ, zeromq) due to Redis's simplicity and performance.
- **Queues**
  - Projects such as Resque use Redis as the backend for queueing background jobs.
- **Counters**
  - Atomic commands such as HINCRBY, allow for a simple and thread-save implementation of counters.
  - Creating a counter is as simple as determining a name for a key and issuing the HINCRBY command.
  - There is no need to read the data before incrementing, and there are no database schemas to update.
  - Since these are atomic operations, the counters will maintain consistency when accessed from multiple application servers.

Companies Using Redis

Ali Katkar

How to install on ubuntu

Step 1: Update

- Update the apt-get packages index files and also update existing packages to the newest versions by using the following commands:

```
$ sudo add-apt-repository ppa:chris-lea/redis-server
$ sudo apt-get update
```

Step 2: Installing Redis

- The Redis packages are available under the default apt repository.
  - For the installation of Redis on an Ubuntu VPS.
- Run below command from the terminal to install Redis on your machine:

```
$ sudo apt-get install redis-server
```

- Next is to enable Redis to start on system boot. Also restart Redis service once.

```
$ sudo systemctl status redis-server.service
$ sudo systemctl enable redis-server.service
```

Step 3: Configure Redis

- Redis can be started without a configuration file using a built-in default configuration.
- But to make any extra parameter changes you can use its configuration file that is:
  - **`/etc/redis/redis.conf`**
- Edit the Redis configuration file in a text editor to make changes

```
$ sudo gedit /etc/redis/redis.conf
Change as → bind 0.0.0.0
```

Step 4: Install PHP Extension

- Now, if you need to use Redis from PHP application, you also need to install Redis PHP extension on your Ubuntu system.
- Run below command to install:

```
$ sudo apt-get install php-redis
```

Step 5: Running Redis

- You can start and stop Redis server with following commands

```
$ sudo systemctl status  redis-server.service

$ sudo systemctl start   redis-server.service

$ sudo systemctl restart redis-server.service

$ sudo systemctl stop    redis-server.service
```

Ali Katkar

Step 6: Test Connection

- Use redis-cli tool to verify the connection between Redis server and redis-cli.

```
$ redis-cli


127.0.0.1:6379> ping
PONG
127.0.0.1:6379>
```

```
$ redis-server --version
```

Redis Data Types

Redis Key Value

- Redis is <u>not a *plain* key-value store</u>, it is actually a *data structures server*, supporting different kinds of values.
- What this means is that,
  - while in traditional key-value stores you associated **string keys** to **string values**
  - in Redis <u>**the value is not limited to a simple string**</u>, but can also hold more complex data structures.

Data Types - I

- **Binary-safe strings**.
- **Lists:** collections of string elements sorted according to the order of insertion. They are basically *linked lists*.
- **Sets:** collections of unique, unsorted string elements.
- **Sorted sets:** similar to Sets but where every string element is associated to a floating number value, called *score*.
  - The elements are always taken sorted by their score,
  - so unlike Sets it is possible to retrieve a range of elements
  - for example you may ask: give me the top 10, or the bottom 10
- **Hashes**: which are maps composed of fields associated with values.
  - Both the field and the value are strings.
  - This is very similar to Ruby or Python hashes.

Data Types - II

- **Bit arrays** (or simply bitmaps): it is possible, using special commands, to handle String values like an array of bits:
  - you can set and clear individual bits, count all the bits set to 1, find the first set or unset bit, and so forth.
- **HyperLogLogs:** this is a probabilistic data structure which is used in order to estimate the cardinality of a set.
  - Don't be scared, it is simpler than it seems
- **Streams:** append-only collections of map-like entries that provide an abstract log data type.
  - They are covered in depth in the Introduction to Redis Streams.

Redis Keys

- Redis keys are binary safe, this means that you can use any binary sequence as a key,
  - from a string like **"foo"** to the content of a JPEG file.
  - The empty string is also a valid key.

A few other rules about keys:

- Very long keys are not a good idea.
  - For instance a key of 1024 bytes is a bad idea not only memory-wise, but also because the lookup of the key in the dataset may require several costly key-comparisons.
  - Even when the task at hand is to match the existence of a large value, hashing it (for example with SHA1) is a better idea, especially from the perspective of memory and bandwidth.
- Very short keys are often not a good idea.
  - There is little point in writing **"u1000flw"** as a key if you can instead write **"user:1000:followers"**
    - The latter is more readable and the added space is minor compared to the space used by the key object itself and the value object
  - While short keys will obviously consume a bit less memory, your job is to find the right balance.
- Try to stick with a schema.
  - For instance **"object-type:id"** is a good idea, as in **"user:1000"**
  - Dots or dashes are often used for multi-word fields, as in "comment:1234:reply.to" or "comment:1234:reply-to".
- The maximum allowed key size is 512 MB.

Ali Katkar

Strings

- The Redis String type is **the simplest type** of value you can associate with a Redis key.
- It is the only data type in Memcached
  - it is also very natural for newcomers to use it in Redis.
- Since Redis keys are strings, when we use the string type as a value too, we are mapping a **string** to **another string**.
- The string data type is useful for a number of use cases
  - like caching HTML fragments or pages.

SET and GET

- Using the SET and the GET commands are the way we set and retrieve a string value.
- Note that SET **will replace** any existing value already stored into the key
  - in the case that the key already exists, even if the key is associated with a non-string value.
- So SET performs an assignment.

String Example

```
$ redis-cli
127.0.0.1:6379>
127.0.0.1:6379> set server:name fido
OK
127.0.0.1:6379> get server:name
"fido"
127.0.0.1:6379>
```

DEL and INCR

- Other common operations provided by key-value stores are:
  - DEL to delete a given key and associated value,
  - INCR to atomically increment a number stored at a given key

DEL and INCR Example

```
127.0.0.1:6379> set connections 10
127.0.0.1:6379> incr connections
(integer) 11
127.0.0.1:6379> get connections
"11"
127.0.0.1:6379> del connections
(integer 1)
127.0.0.1:6379> get connections
nil
127.0.0.1:6379> incr connections
(integer 1)
```

SET If Not Exists (nx|xx)

- SET-if-not-exists (called SETNX on Redis) that sets a key only if it does not already exist

```
127.0.0.1:6379> set mykey value nx
127.0.0.1:6379> get mykey
127.0.0.1:6379> set mykey value xx
127.0.0.1:6379> get mykey
127.0.0.1:6379> set mykey newValue nx
127.0.0.1:6379> get mykey
127.0.0.1:6379> set mykey newValue xx
127.0.0.1:6379> get mykey
```

EXPIRE and TTL

- Redis can be told that a key should only exist for a certain length of time.
- This is accomplished with the EXPIRE and TTL commands.
- You can test how long a key will exist with the TTL command.
    - The *-2* means that the key does not exist (anymore).
    - The *-1* means that it will never expire.
    - Note that if you SET a key, its TTL will be reset.

EXPIRE and TTL Example

```
127.0.0.1:6379> SET mykey value EX 25
127.0.0.1:6379> TTL mykey

127.0.0.1:6379> SET mykey value
127.0.0.1:6379> EXPIRE mykey 25
127.0.0.1:6379> TTL mykey
```

Ali Katkar

String Commands

STRINGS			
APPEND	Append	INCRBYFLOAT	Add to float
BITCOUNT	Count set bits	MGET	Get multiple
BITOP	Bitwise operations	MSET	Set multiple
BITPOS	Find first set bit	MSETNX	Set multiple if don't exist
DECR	Decrement integer	PSETEX	Set with expiry (ms)
DECRBY	Subtract from integer	SET	Set
GET	Get by key	SETBIT	Set bit by index
GETBIT	Get bit by index	SETEX	Set with expiry (seconds)
GETRANGE	Get substring	SETNX	Set if doesn't exist
GETSET	Set, returning old value	SETRANGE	Set substring
INCR	Increment integer	STRLEN	Get length
INCRBY	Add to integer		

Lists

- Redis lists are implemented via Linked Lists.
- This means that even if you have millions of elements inside a list, the operation of adding a new element in the head or in the tail of the list is performed *in constant time*.
  - The speed of adding a new element with the LPUSH command to the head of a list with ten elements is the same as adding an element to the head of list with 10 million elements.
- What's the downside?
  - Accessing an element *by index* is very fast in lists implemented with an Array (constant time indexed access)
  - Not so fast in lists implemented by linked lists (where the operation requires an amount of work proportional to the index of the accessed element).

LPUSH, RPUSH, LRANGE

- RPUSH puts the new value at the end of the list.
- LPUSH puts the new value at the start of the list.
- LRANGE gives a subset of the list.
  - It takes the index of the first element you want to retrieve as its first parameter
  - the index of the last element you want to retrieve as its second parameter.
  - A value of -1 for the second parameter means to retrieve elements until the end of the list.

LPUSH, RPUSH, LRANGE Example

```
127.0.0.1:6379> rpush friends alice
127.0.0.1:6379> rpush friends bob michael

127.0.0.1:6379> lrange friends 0 -1


127.0.0.1:6379> lpush friends sam


127.0.0.1:6379> lrange friends 0 -1
```

LPOP, RPOP, LLEN

- LLEN returns the current length of the list.

```
127.0.0.1:6379> llen friends
```

- LPOP removes the first element from the list and returns it.

```
127.0.0.1:6379> lpop friends
```

- RPOP removes the last element from the list and returns it.

```
127.0.0.1:6379> rpop friends
```

  - Note that the list now has two elements

Use Case for Lists

- Twitter takes the latest tweets posted by users into Redis lists.
- To describe a common use case step by step:
  - imagine your home page shows the latest photos published in a photo sharing social network
  - and you want to speedup access.
  - Every time a user posts a new photo, we add its ID into a list with LPUSH.
  - When users visit the home page, we use LRANGE 0 9 in order to get the latest 10 posted items.

Capped Lists

- In many use cases we just want to use lists to store the *latest items*, whatever they are:
  - social network updates, logs, or anything else.
- Redis allows us to use lists as a capped collection, only remembering the latest N items and discarding all the oldest items using the LTRIM command.
- The LTRIM command is similar to LRANGE, but **instead of displaying the specified range of elements** it sets this range as the new list value.
- All the elements outside the given range are removed.

LTRIM Example

```
127.0.0.1:6379> rpush numbers 1 2 3 4 5 6 7 8

127.0.0.1:6379> lrange numbers 0 -1

127.0.0.1:6379> ltrim numbers 0 5

127.0.0.1:6379> lrange numbers 0 -1

127.0.0.1:6379> ltrim numbers 1 5

127.0.0.1:6379> lrange numbers 0 -1
```

Blocking Operations

- Lists have a special feature that make them suitable to
  - implement queues
  - building block for inter process communication systems
- Blocking operations
- Imagine you want to push items into a list with one process, and use a different process in order to actually do some kind of work with those items.
  - This is the usual producer / consumer setup, and can be implemented in the following simple way:
  - To push items into the list, producers call LPUSH.
  - To extract / process items from the list, consumers call RPOP.
- However it is possible that sometimes the list is empty and there is nothing to process, so RPOP just returns NULL.
- In this case a consumer is forced to wait some time and retry again with RPOP.

Blocking Operations

- This is called *polling*, and is not a good idea in this context because it has several drawbacks:
  - Forces Redis and clients to process useless commands (all the requests when the list is empty will get no actual work done, they'll just return NULL).
  - Adds a delay to the processing of items, since after a worker receives a NULL, it waits some time.
  - To make the delay smaller, we could wait less between calls to RPOP, with the effect of amplifying problem number 1, i.e. more useless calls to Redis.

- So Redis implements commands called BRPOP and BLPOP
  - which are versions of RPOP and LPOP able to block if the list is empty:
  - they'll return to the caller only when a new element is added to the list,
  - or when a user-specified timeout is reached.

BRPOP, BLPOP Example

```
127.0.0.1:6379> rpush numbers 1


127.0.0.1:6379> rpush numbers 2
```

```
127.0.0.1:6379> brpop numbers 30


127.0.0.1:6379> brpop numbers 30
```

BRPOP, BLPOP

- Note that
  - you can use 0 as timeout to wait for elements forever
  - you can also specify multiple lists and not just one, in order to wait on multiple lists at the same time,
  - get notified when the first list receives an element.

- A few things to note about BRPOP:
  - Clients are served in an ordered way: the first client that blocked waiting for a list, is served first when an element is pushed by some other client, and so forth.
  - The return value is different compared to RPOP:
    - it is a two-element array since it also includes the name of the key, because BRPOP and BLPOP are able to block waiting for elements from multiple lists.
  - If the timeout is reached, NULL is returned.

Automatic Creation

- So far in our examples we never had to
  - **create empty lists** before pushing elements,
  - **remove empty lists** when they no longer have elements inside.
- It is Redis' responsibility
  - to delete keys when lists are left empty, or
  - to create an empty list if the key does not exist when we are trying to add elements to it, for example, with LPUSH.
- This is not specific to lists, it applies to all the Redis data types composed of multiple elements
  - Streams, Sets, Sorted Sets and Hashes.

Automatic Creation

- Basically we can summarize the behavior with three rules:
  1. When we add an element to an aggregate data type, if the target key does not exist, an empty aggregate data type is created before adding the element.
  2. When we remove elements from an aggregate data type, if the value remains empty, the key is automatically destroyed.
     - The **Stream** data type is the only exception to this rule.
  3. Calling a read-only command such as LLEN (which returns the length of the list), or a write command removing elements, with an empty key, always produces the same result as if the key is holding an empty aggregate type of the type the command expects to find.

Sets

- A set is similar to a list, **except** it does not have a specific order and each element may only **appear once**.
- Some of the important commands in working with sets are
  - SADD
  - SREM
  - SISMEMBER
  - SMEMBERS
  - SUNION

Important Commands of Sets

- SADD adds the given value to the set.
- SREM removes the given value from the set.
- SISMEMBER tests if the given value is in the set. It returns 1 if the value is there and 0 if it is not.
- SMEMBERS returns a list of all the members of this set.
- SUNION combines two or more sets and returns the list of all elements.

Commands of Sets Examples

```
> SADD superpowers "flight"
> SADD superpowers "x-ray vision"
> SADD superpowers "reflexes"
> SREM superpowers "reflexes"

> SISMEMBER superpowers "flight"
> SISMEMBER superpowers "reflexes"
> SMEMBERS superpowers

> SADD birdpowers "pecking"
> SADD birdpowers "flight"
> SUNION superpowers birdpowers
```

Sorted Sets

- Sets are a very handy data type
- but as they are unsorted they don't work well for a number of problems.
- This is why Redis 1.2 introduced Sorted Sets.
- A sorted set is similar to a regular set, but now each value has an associated score.
- This score is used to sort the elements in the set.

Sorted Sets

- ZADD is similar to SADD, but takes one additional argument (placed before the element to be added) which is the score.
- ZADD is also variadic, so you are free to specify multiple score-value pairs
- With sorted sets it is **trivial to return a list as sorted** by their score because actually *they are already sorted*.
- **Implementation note:**
  - Sorted sets are implemented via a dual-ported data structure containing both a skip list and a hash table
  - so every time we add an element Redis performs an O(log(N)) operation.
  - That's good, but when we ask for sorted elements Redis does not have to do any work at all, it's already all sorted

Sorted Set Examples

```
> ZADD hackers 1940 "Alan Kay"
> ZADD hackers 1906 "Grace Hopper"
> ZADD hackers 1953 "Richard Stallman"
> ZADD hackers 1965 "Yukihiro Matsumoto"
> ZADD hackers 1916 "Claude Shannon"
> ZADD hackers 1969 "Linus Torvalds"
> ZADD hackers 1957 "Sophie Wilson"
> ZADD hackers 1912 "Alan Turing"

> ZRANGE hackers 2 4
> ZREVRANGE hackers 0 -1
> ZREVRANGE hackers 0 -1 withscores
```

Sorted Sets
Operating on Ranges

- Sorted sets are more powerful than this.
- They can operate on ranges.
  - Let's get all the individuals that were born up to 1950 inclusive.

```
> ZRANGEBYSCORE hackers -inf 1950
```

    - We asked Redis to return all the elements with a score between negative infinity and 1950 (both extremes are included).
  - It's also possible to remove ranges of elements.
  - Let's remove all the hackers born between 1940 and 1960 from the sorted set:

```
> ZREMRANGEBYSCORE hackers 1940 1960
```

Lexicographical Scores

- With recent versions of Redis 2.8, a new feature was introduced that allows getting ranges lexicographically,
- Assuming elements in a sorted set are all inserted with the same identical score
  - Elements are compared with the C memcmp function, so it is guaranteed that there is no collation,
  - Every Redis instance will reply with the same output

- The main commands to operate with lexicographical ranges are ZRANGEBYLEX, ZREVRANGEBYLEX, ZREMRANGEBYLEX and ZLEXCOUNT.

  - For example, let's add again our list of famous hackers, but this time use a score of zero for all the elements:

ZRANGEBYLEX Examples

```
> ZADD hackers 0 "Alan Kay"
> ZADD hackers 0 "Grace Hopper"
> ZADD hackers 0 "Richard Stallman"
> ZADD hackers 0 "Yukihiro Matsumoto"
> ZADD hackers 0 "Claude Shannon"
> ZADD hackers 0 "Linus Torvalds"
> ZADD hackers 0 "Sophie Wilson"
> ZADD hackers 0 "Alan Turing"

> ZRANGE hackers 0 -1
> ZRANGEBYLEX hackers [B [P
```

Updating the score: Leader Boards

- Sorted sets' scores can be updated at any time.
- Just calling ZADD against an element already included in the sorted set will update **its score** (and position) with O(log(N)) time complexity.
- As such, sorted sets are suitable when there are tons of updates.
- A common use case is leader boards.
  - The typical application is a Facebook game where you combine the ability to take users sorted **by their high score**, plus **the get-rank operation**
  - In order to show the top-N users, and the user rank in the leader board
    - (e.g., "you are the #4932 best score here").

Hashes

- Hashes are maps between string fields and string values
- They are the perfect data type to represent objects
- (eg: A User with a number of fields like name, surname, age, and so forth):

HSET, HMSET, HGETALL

```
> hset user:1000 name "john Smith"
> hset user:1000 email "john@example.com"
> hset user:1000 password "secret"

> hgetall user:1000

> hmset user:1001 name "mary" email "mary@example.com"

> hget user:1001 name
```

Ali Katkar

BitMaps

- Bitmaps are not an actual data type, but a set of bit-oriented operations defined on the String type.
  - Since strings are binary safe blobs and their maximum length is 512 MB, they are suitable to set up to 2^{32} different bits.
- Bit operations are divided into two groups:
  - Constant-time single bit operations, like setting a bit to 1 or 0, or getting its value,
  - Operations on groups of bits,
    - for example counting the number of set bits in a given range of bits (e.g., population counting).
- One of the biggest advantages of bitmaps is that they often provide **extreme space savings** when storing information.
  - For example in a system where different users are represented by incremental user IDs, it is possible to remember a single bit information (for example, knowing whether a user wants to receive a newsletter) of 4 billion of users using just 512 MB of memory.
- Bits are set and retrieved using the SETBIT and GETBIT commands:

```
> SETBIT key 10 1
> GETBIT key 10
```

BitMaps: Group of Bits

- There are three commands operating on group of bits:
  - BITOP performs bit-wise operations between different strings.
    - The provided operations are AND, OR, XOR and NOT.
  - BITCOUNT performs population counting, reporting the number of bits set to 1.
  - BITPOS finds the first bit having the specified value of 0 or 1.
- Both BITPOS and BITCOUNT are able to operate with byte ranges of the string, instead of running for the whole length of the string.
  - The following is a trivial example of BITCOUNT call:

```
> SETBIT key 0 1
> SETBIT key 100 1
> BITCOUNT key
```

BitMaps: Common Use Cases

- Common use cases for bitmaps are:
  - Real time analytics of all kinds.
  - Storing space efficient but high performance boolean information associated with object IDs.
- For example imagine you want to know the longest streak of daily visits of your web site users.
  - You start counting days starting from zero, that is the day you made your web site public,
  - and set a bit with SETBIT every time the user visits the web site.
  - As a bit index you simply take the current unix time, subtract the initial offset, and divide by 3600*24.
  - This way for each user you have a small string containing the visit information for each day.
  - With BITCOUNT it is possible to easily get the number of days a given user visited the web site,
  - while with a few BITPOS calls, or simply fetching and analyzing the bitmap client-side, it is possible to easily compute the longest streak.

HyperLogLogs

- A HyperLogLog is a probabilistic data structure used in order to count unique things
  - technically this is referred to estimating the cardinality of a set
- Usually **counting unique items** requires using an amount of memory proportional to the number of items you want to count,
  - Because you need to remember the elements you have already seen in the past in order to avoid counting them multiple times.
- However there is a set of algorithms that trade memory for precision:
  - You end with an estimated measure with a standard error, which in the case of the Redis implementation is less than 1%.
  - The magic of this algorithm is that you no longer need to use an amount of memory proportional to the number of items counted, and instead can use a constant amount of memory!
  - 12k bytes in the worst case, or a lot less if your HyperLogLog (We'll just call them HLL from now) has seen very few elements.

HyperLogLogs

- HLLs in Redis, while technically a different data structure, are encoded as a Redis string
  - So you can call GET to serialize a HLL,
  - SET to deserialize it back to the server.
- Conceptually the HLL API is like using Sets to do the same task.
  - You would SADD every observed element into a set,
  - and would use SCARD to check the number of elements inside the set,
  - which are unique since SADD will not re-add an existing element.
- While you don't really *add items* into an HLL, because the data structure only contains a state that does not include actual elements, the API is the same:
  - Every time you see a new element, you add it to the count with PFADD.
  - Every time you want to retrieve the current approximation of the unique elements *added* with PFADD so far, you use the PFCOUNT.
- An example of use case for this data structure is counting unique queries performed by users in a search form every day.

```
> PFADD hll a b c d
> PFCOUNT hll
```

Publish/Subscribe
Pub/Sub

Ali KATKAR

Pub/Sub

- SUBSCRIBE, UNSUBSCRIBE and PUBLISH implement the Publish/Subscribe messaging paradigm
  - Where (citing Wikipedia) senders (publishers) are not programmed to send their messages to specific receivers (subscribers).
  - Rather, published messages are characterized into channels, without knowledge of what (if any) subscribers there may be.
  - Subscribers express interest in one or more channels, and only receive messages that are of interest, without knowledge of what (if any) publishers there are.
  - This decoupling of publishers and subscribers can allow for greater scalability and a more dynamic network topology.

Pub/Sub: Subscribe

- For instance in order to subscribe to channels **first** and **second:**
  - The client issues a SUBSCRIBE providing the names of the channels:

```
> SUBSCRIBE first second
```

  - Messages sent by other clients to these channels will be pushed by Redis to all the subscribed clients.

Pub/Sub
Commands for Subscribers

- A client subscribed to one or more channels should not issue commands
- Although it can subscribe and unsubscribe to and from other channels.
- The replies to subscription and unsubscription operations are sent in the form of messages, so that the client can just read a coherent stream of messages where the first element indicates the type of message.
- The commands that are allowed in the context of a subscribed client are SUBSCRIBE, PSUBSCRIBE, UNSUBSCRIBE, PUNSUBSCRIBE, PING and QUIT.

❖ *Please note that redis-cli <u>will not accept</u> any commands once in subscribed mode and can* ***only quit*** *the mode* ***with Ctrl-C****.*

Pub/Sub Format of Pushed Messages

A message is an **Array** reply with **three** elements.

- **subscribe**: means that we successfully subscribed to the channel given as the second element in the reply.
  - The third argument represents the number of channels we are currently subscribed to.
- **unsubscribe:** means that we successfully unsubscribed from the channel given as second element in the reply.
  - The third argument represents the number of channels we are currently subscribed to.
  - When the last argument is zero, we are no longer subscribed to any channel, and the client can issue any kind of Redis command as we are outside the Pub/Sub state.
- **message:** it is a message received as result of a PUBLISH command issued by another client.
  - The second element is the name of the originating channel,
  - The third argument is the actual message payload.

Pub/Sub Publish

- At this point, from another client we issue a PUBLISH operation against the channel named second:

```
> PUBLISH second "Hello there"
```

- This is what the first client receives

```
1) "message"
2) "second"
3) "Hello there"
```

Streams

Ali KATKAR

Streams

- The Stream is a new data type introduced with Redis 5.0
  - which models a *log data structure* in a more abstract way,
- Redis Streams is essentially a message queue
  - but it is also unique compared to other message middleware such as Kafka and RocketMQ.
- It is, by nature, a message publishing and subscription component on Redis kernel
  - non Redis Module

Streams

- Although the existing PUB/SUB and BLOCKED LIST can be used as a message queue service in simple scenarios **Redis Streams is more comprehensive.**
- It provides
  - Message persistence and Master/ slave data replication
  - The new RadixTree data structure to support more efficient memory use and message reading.
  - A function similar to Kafka's Consumer Group.

Basic Commands

- Streams are an append only data structure
- **XADD** appends a new entry into the specified stream.
- A stream entry is not just a string, but is instead composed of **one or multiple** field-value pairs.

```
127.0.0.1:6379> XADD mystream * sensor-id 1234 temperature 19.8
1518951480106-0
127.0.0.1:6379> XLEN mystream
(integer) 1
127.0.0.1:6379> XRANGE mystream - +
127.0.0.1:6379> XRANGE mystream - + COUNT 2
127.0.0.1:6379> XRANGE mystream 1518951480106 1518951480107
127.0.0.1:6379> XREVRANGE mystream + - COUNT 1
```

Listening for New Items

- Fundamental differences in the way you consume a stream
  - A stream can have multiple clients (consumers) waiting for data.
    - Every new item, by default, will be delivered to *every consumer* that is waiting for data in a given stream.
    - This behavior is different than blocking lists, where each consumer will get a different element. However, the ability to *fan out* to multiple consumers is similar to Pub/Sub.
  - All the messages are appended in the stream indefinitely unless the user explicitly asks to delete entries
    - in Pub/Sub messages are *fire and forget* and are never stored anyway
    - while when using blocking lists, when a message is received by the client it is *popped* (effectively removed) form the list
  - Streams Consumer Groups provide a level of control with different groups for the same stream
    - explicit acknowledge of processed items,
    - ability to inspect the pending items
    - claiming of unprocessed messages
    - coherent history visibility for each single client, that is only able to see its private past history of messages.

Listening for New Items

- The command that provides the ability to listen for new messages arriving into a stream is called **XREAD**
  - $ means last Id

```
127.0.0.1:6379> XREAD COUNT 2 STREAMS mystream 0
127.0.0.1:6379> XREAD BLOCK 0 STREAMS mystream $
```

Consumer Groups

- Provide a *different subset* of messages from the same stream to many clients.
- An obvious case where this is useful is the case of slow to process messages:
  - The ability to have N different workers that will receive different parts of the stream allow to scale message processing, by routing different messages to different workers that are ready to do more work.
- if we imagine having three consumers C1, C2, C3, and a stream that contains the messages 1, 2, 3, 4, 5, 6, 7
- then what we want is to serve the messages like in the following diagram:

```
1 -> C1     2 -> C2     3 -> C3
4 -> C1     5 -> C2     6 -> C3
7 -> C1
```

- In order to obtain this effect, Redis uses a concept called ***consumer groups***.

Consumer Groups

- Each message is served to a different consumer
  - so that it is not possible that the same message is delivered to multiple consumers.
- Consumers are identified by a name,
  - which is a case-sensitive string that the clients implementing consumers must choose.
  - This means that even after a disconnect, the stream consumer group retains all the state, since the client will claim again to be the same consumer.
- Each consumer group has the concept of the *first ID never consumed*
  - When a consumer asks for new messages, it can provide just messages that were never delivered previously.
- Consuming a message however requires explicit acknowledge using a specific command
  - to say: this message was correctly processed, so can be evicted from the consumer group.
- A consumer group tracks all the messages that are currently pending,
  - Messages that were delivered to some consumer of the consumer group, but are yet to be acknowledged as processed.
  - Thanks to this feature, when accessing the history of messages of a stream, each consumer *will only see messages that were delivered to it*.

Creating Consumer Group

```
127.0.0.1:6379> XGROUP CREATE mystream mygroup $

127.0.0.1:6379> XADD mystream * message apple
127.0.0.1:6379> XADD mystream * message orange
127.0.0.1:6379> XADD mystream * message strawberry
127.0.0.1:6379> XADD mystream * message apricot
127.0.0.1:6379> XADD mystream * message banana

127.0.0.1:6379> XREADGROUP GROUP mygroup Alice COUNT 1 STREAMS mystream >

// History of pending messages
127.0.0.1:6379> XREADGROUP GROUP mygroup Alice STREAMS mystream 0
127.0.0.1:6379> XACK mystream mygroup 1526569495631-0
127.0.0.1:6379> XREADGROUP GROUP mygroup Alice STREAMS mystream 0
127.0.0.1:6379> XREADGROUP GROUP mygroup Bob STREAMS mystream >
```

Ali Katkar

Consumer Group More Commands

- Observer pending entires

```
127.0.0.1:6379> XPENDING mystream mygroup
127.0.0.1:6379> XPENDING mystream mygroup - + 10
```

- Alice may decide that after 20 hours of not processing messages,
- Bob will probably not recover in time, and it's time to *claim* such messages and resume the processing in place of Bob.

```
Client 1: XCLAIM mystream mygroup Alice 3600000 1526569498055-0
Clinet 2: XCLAIM mystream mygroup Lora 3600000 1526569498055-0
```

Observer Streams

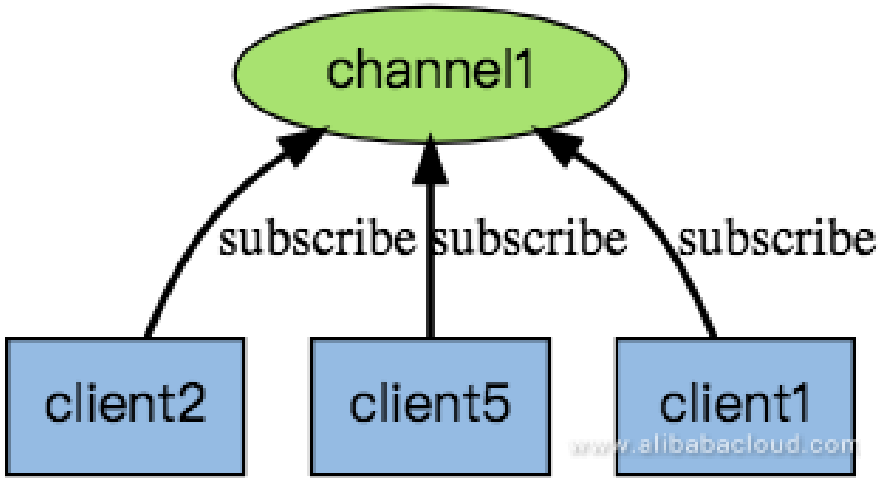
- The **XINFO** command is an observability interface that can be used with sub-commands in order to get information about streams or consumer groups.

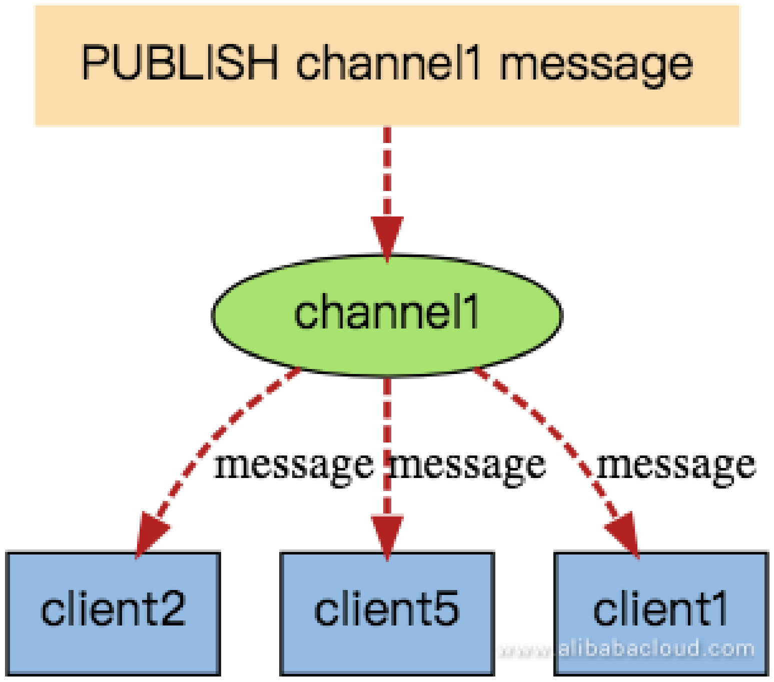
```
127.0.0.1:6379> XINFO STREAM mystream
127.0.0.1:6379> XINFO GROUPS mystream
127.0.0.1:6379> XINFO CONSUMERS mystream mygroup
127.0.0.1:6379> XINFO help
```

Streams & IRC Systems

Ali Katkar

Streams & IRC Systems





Ali Katkar

IRC Pub/Sub Problems

- PUB/SUB’s uses the so-called “**Fire-and-Forget**” message model.
- That is to say Redis does not save any message history.
- If a user in an IRC channel **is disconnected** due to network failure and rejoins the IRC channel later on, he will not be able to see the chat log for the period when he is disconnected.
- New users also cannot see the historical chat log for the latest discussion.
- This causes great inconvenience for users who want to quickly understand the currently discussed topics.
- In addition, if Redis is restarted, then all users need to **resubscribe** to their channels again.

IRC Channels With Streams

- PUB/SUB's uses the so-called "**Fire-and-Forget**" message model.
- That is to say Redis does not save any message history.
- If a user in an IRC channel **is disconnected** due to network failure and rejoins the IRC channel later on, he will not be able to see the chat log for the period when he is disconnected.
- New users also cannot see the historical chat log for the latest discussion.
- This causes great inconvenience for users who want to quickly understand the currently discussed topics.
- In addition, if Redis is restarted, then all users need to **resubscribe** to their channels again.

Transactions

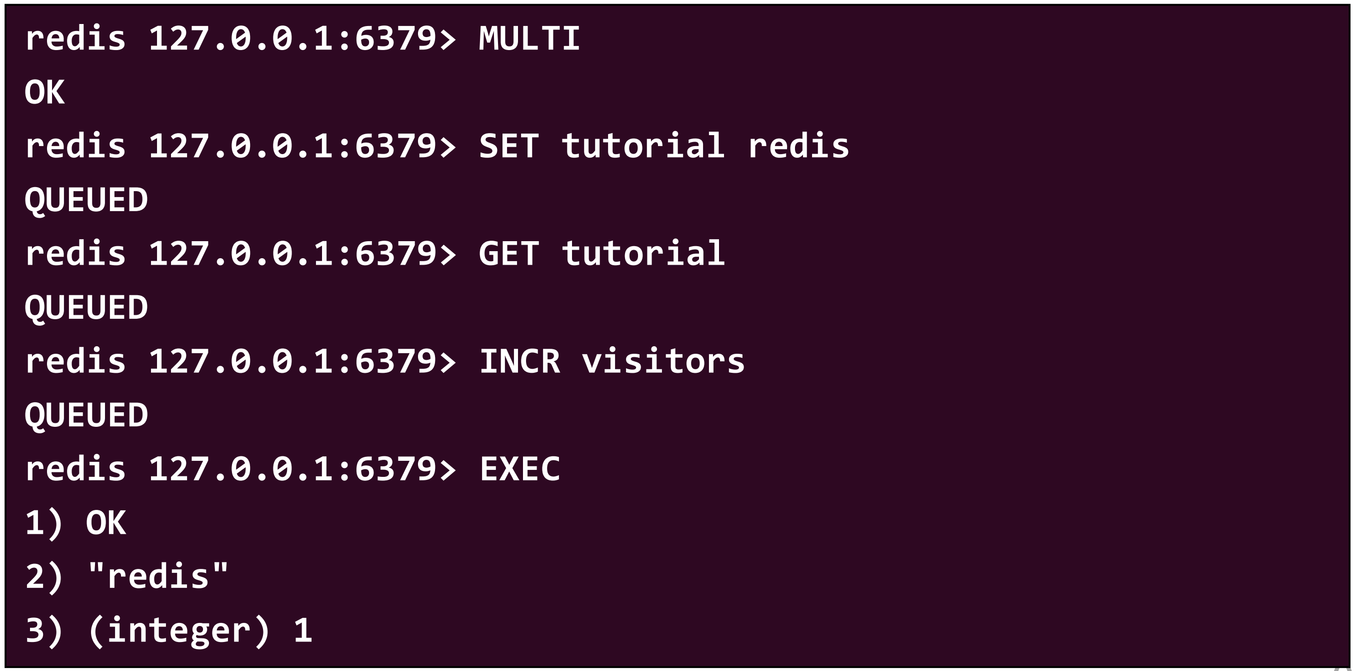
Ali KATKAR

Transactions

- Redis transactions allow the execution of a group of commands in a single step.
- Following are the two properties of Transactions.
  - All commands in a transaction are sequentially executed as a single isolated operation.
    - It is not possible that a request issued by another client is served in the middle of the execution of a Redis transaction.
  - Redis transaction is also atomic.
    - Atomic means either all of the commands or none are processed.

Transactions

- Redis transaction is initiated by command **MULTI**
- Then you need to pass a list of commands that should be executed in the transaction
- All commands are executed by **EXEC** command



```
redis 127.0.0.1:6379> MULTI
OK
redis 127.0.0.1:6379> SET tutorial redis
QUEUED
redis 127.0.0.1:6379> GET tutorial
QUEUED
redis 127.0.0.1:6379> INCR visitors
QUEUED
redis 127.0.0.1:6379> EXEC
1) OK
2) "redis"
3) (integer) 1
```

Transactions Commands

Command	Description
DISCARD	Discards all commands issued after MULTI
EXEC	Executes all commands issued after MULTI
MULTI	Marks the start of a transaction block
UNWATCH	Forgets about all watched keys
WATCH key [key ...]	Watches the given keys to determine the execution of the MULTI/EXEC block

Software Clients Python & Java

Ali KATKAR

Software Clients for Redis

ActionScript	D	GNU Prolog	Lua	Perl	Ruby	VCL
Bash	Dart	Go	Matlab	PHP	Rust	Xojo
C	Delphi	Haskell	mruby	PL/SQL	Scala	
C#	Elixir	Haxe	Nim	Pure Data	Scheme	
C++	emacs lisp	Io	Node.js	Python	Smalltalk	
Clojure	Erlang	Java	Objective-C	R	Swift	
Common Lisp	Fancy	Julia	OCaml	Racket	Tcl	
Crystal	gawk	Lasso	Pascal	Rebol	VB	

Java Client for Redis

- We will use **Jedis**
  - There are multiple alternatives to Jedis, but only two more are currently worthy of their recommendation star, lettuce, and Redisson.
- Why **Jedis**
  - There are multiple alternatives to Jedis, but only two more are currently worthy of their recommendation star, lettuce, and Redisson.
  - These two clients do have some unique features like thread safety, transparent reconnection handling and an asynchronous API, all features of which Jedis lacks.
  - However, Jedis is small and considerably faster than the other two.
  - Besides, it is the client library of choice of the *Spring Framework* developers, and it has the biggest community of all three.

Java Client for Redis

- Maven Dependencies
  - We need to declare only dependency we will need in the ***pom.xml***:

```
<dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
    <version>2.8.1</version>
</dependency>
```

  - If you're looking for the latest version of the library, check out this page.

Python Client for Redis

- Installing redis-py

```
> pip install redis
```

- Now you can use redis from python shell

```
C:\Users\Ali Katkar>python
Python 3.6.0 (v3.6.0:41df79263a11, Dec 23 2016, 07:18:10) ...
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> import redis
>>>
>>> cache = redis.StrictRedis(host='192.168.1.123', port=6379, db=0)
>>> cache.set("mykey", "deneme")
True
>>> cache.get("mykey")
b'deneme'
>>>
```

Scripting

Ali KATKAR

Scripting

- Redis scripting is used to evaluate scripts using the **Lua** interpreter.
- It is built into Redis starting from version 2.6.0.
- The command used for scripting is **EVAL** command

```
redis 127.0.0.1:6379> EVAL "return {KEYS[1],KEYS[2],ARGV[1],ARGV[2]}" 2 key1
key2 first second
1) "key1"
2) "key2"
3) "first"
4) "second"
```

Redis Scripting Commands

Command	Description
EVAL	Executes a Lua script.
EVALSHA	Executes a Lua script. Cached
SCRIPT EXISTS	Check in the script cache by hash
SCRIPT FLUSH	Clear cache
SCRIPT KILL	Kill running script
SCRIPT LOAD	Loads the specified Lua script into the script cache.