Session 302

You Should Be Using Redis

John Hobbs

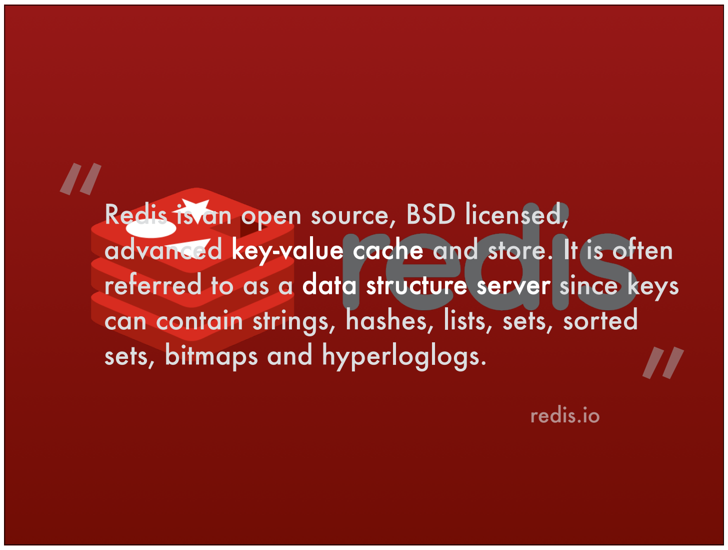
AIM | hdc

I'm John Hobbs and this is "You Should Be Using Redis"

PACK

I'm the developer, ops, etc. at Pack. We use Redis extensively there, and I love it.

If you have questions or comments during this presentation, please ask them.



"Redis is an open source, BSD licensed, advanced **key-value cache** and store. It is often referred to as a **data structure server** since keys can contain strings, hashes, lists, sets, sorted sets, bitmaps and hyperloglogs."

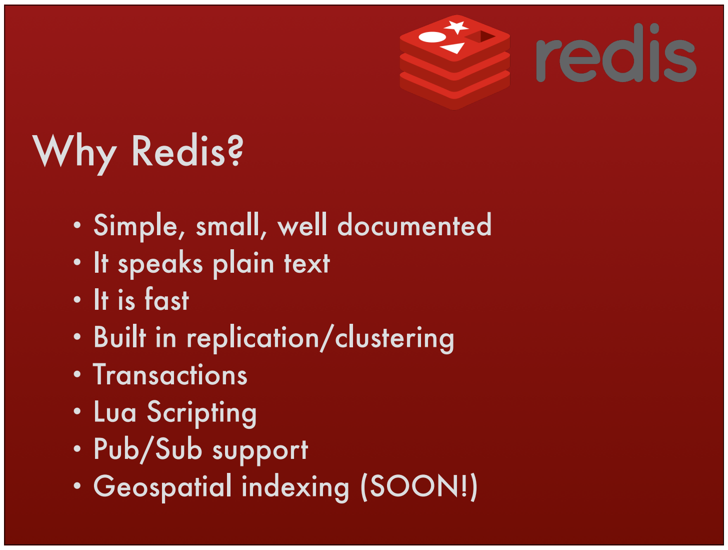
redis.io

As you might imagine, this talk is about redis. But what is redis?

The two key points are that it's a key-value cache, and a data structure server.

Why Redis?

So why redis?



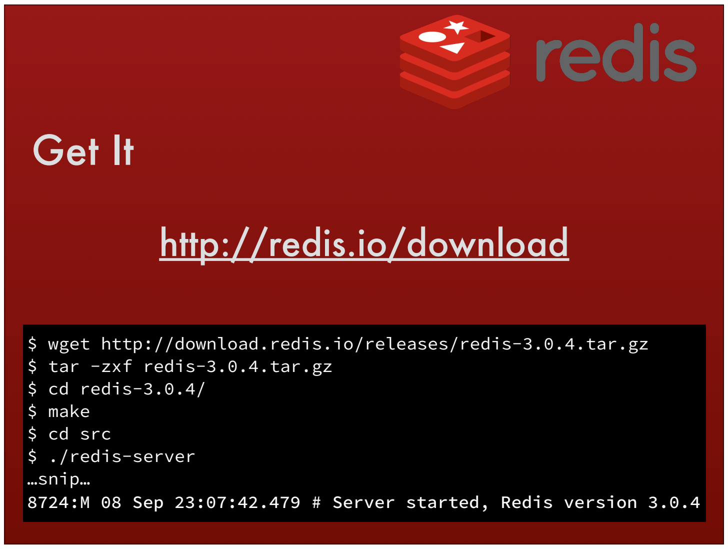
Why Redis?

- Simple, small, well documented
- It speaks plain text
- It is fast
- Built in replication/clustering
- Transactions
- Lua Scripting
- Pub/Sub support
- Geospatial indexing (SOON!)

There are 59 C files in the distribution. No (external) dependencies. 20s to build all server and tools on this machine. Simplicity leads to easy to deploy and configure.
Every command has examples and a big-O notation of complexity.
You can talk to it over telnet if you want.
Redis exists in memory, with disk storage an option. As such, it's fast.
Redis can replicate to disk, has clustering with sharding and automatic failover.
You can do atomic transactions, and pipeline multiple commands for speed.
Lastly, you can extend redis by using the built in Lua scripting.

For this talk I'm going to try to cover realistic use cases first, then edge out into some of the other details, like sentinel. I am NOT advocating you replace your relational database with redis. We want to find ways to leverage redis in your existing stack and let it do what it is best at.

This talk is highly informed by a blog post from antirez which I will link to at the end.



Get It

http://redis.io/download

```
$ wget http://download.redis.io/releases/redis-3.0.4.tar.gz
$ tar -zxf redis-3.0.4.tar.gz
$ cd redis-3.0.4/
$ make
$ cd src
$ ./redis-server
…snip…
8724:M 08 Sep 23:07:42.479 # Server started, Redis version 3.0.4
```

I'd encourage you to grab a copy of redis and follow along if you'd like to.

Redis builds into it's own source directory so it's nice and clean.

If you want to go system wide, it's in most linux repos and available on Homebrew. YMMV. However, 3.0.4 is only a few days old, so I recommend you just grab the source.

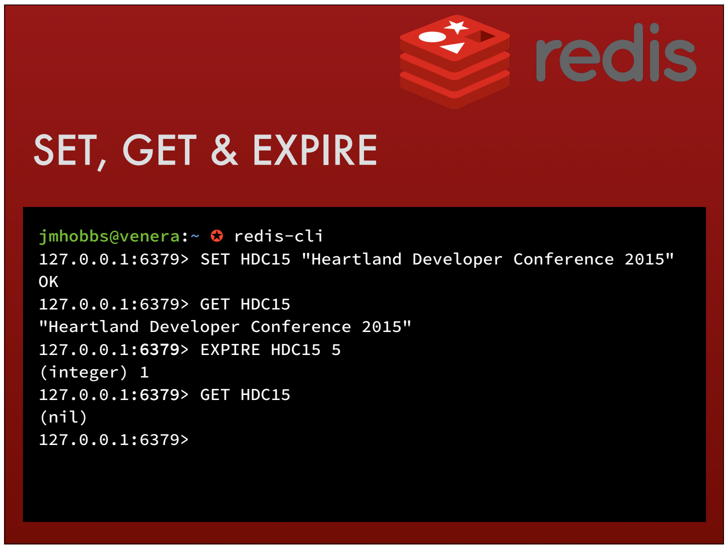
If you are on Windows, there is an unsupported fork by Microsoft, look for MSOpenTech on github. The install from there will be a bit messier of course.

Caching

The first use case is also probably the most obvious. Use it as a cache.

Redis has LRU eviction when memory gets tight, as well as millisecond expiration resolution.

Other benefits over memcached are disk persistence and larger value size (512MB vs 1MB). At least comparable in speed, probably faster http://oldblog.antirez.com/post/update-on-memcached-redis-benchmark.html

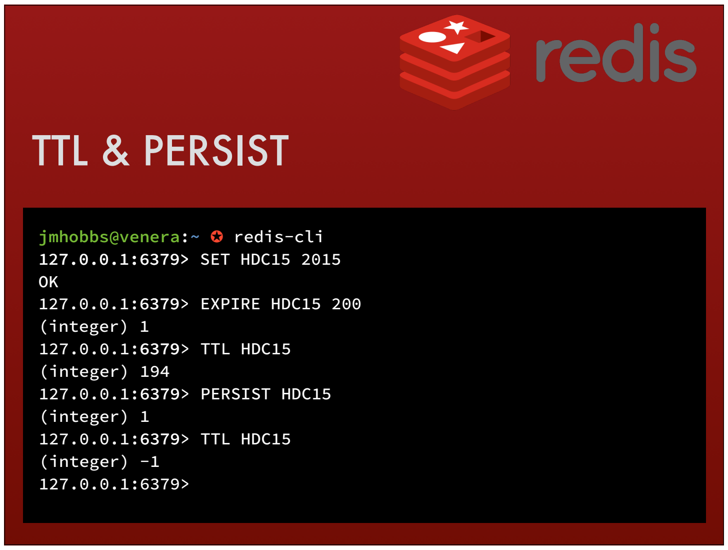


SET, GET & EXPIRE

```
jmhobbs@venera:~ ✪ redis-cli
127.0.0.1:6379> SET HDC15 "Heartland Developer Conference 2015"
OK
127.0.0.1:6379> GET HDC15
"Heartland Developer Conference 2015"
127.0.0.1:6379> EXPIRE HDC15 5
(integer) 1
127.0.0.1:6379> GET HDC15
(nil)
127.0.0.1:6379>
```

The three most basic commands in redis are SET, GET, and EXPIRE.

These do exactly what you think they do. GET a value for a key, and SET one. In the case of GET/SET, the value stored is a string. Almost everything in redis boils down to strings.



TTL & PERSIST

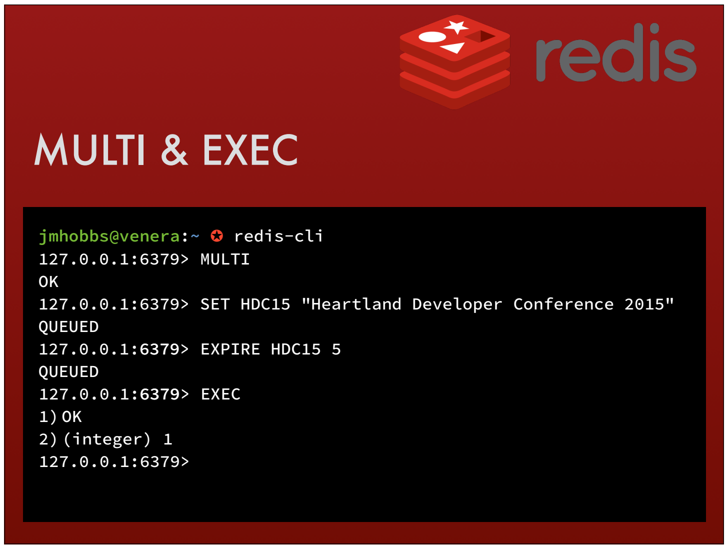
```
jmhobbs@venera:~ ✪ redis-cli
127.0.0.1:6379> SET HDC15 2015
OK
127.0.0.1:6379> EXPIRE HDC15 200
(integer) 1
127.0.0.1:6379> TTL HDC15
(integer) 194
127.0.0.1:6379> PERSIST HDC15
(integer) 1
127.0.0.1:6379> TTL HDC15
(integer) -1
127.0.0.1:6379>
```

Two more commands related to EXPIRE are TTL and PERSIST.

TTL is time to live for a key in seconds. There is a PTTL, which returns milliseconds.

PERSIST will remove the expiration on a volatile key and make it persistent.

Note that the second time I call TTL, I get a negative return value. This is a pattern common to redis. Negative return values are in-band signaling of error conditions. Errors are specific to the command called, in the case of TTL -1 means the key exists, but is not volatile. I find this error handling a bit of a rough edge of redis, but most libraries you use should handle things



MULTI & EXEC

```
jmhobbs@venera:~ ✪ redis-cli
127.0.0.1:6379> MULTI
OK
127.0.0.1:6379> SET HDC15 "Heartland Developer Conference 2015"
QUEUED
127.0.0.1:6379> EXPIRE HDC15 5
QUEUED
127.0.0.1:6379> EXEC
1) OK
2) (integer) 1
127.0.0.1:6379>
```

We can make that a bit better with transactions.

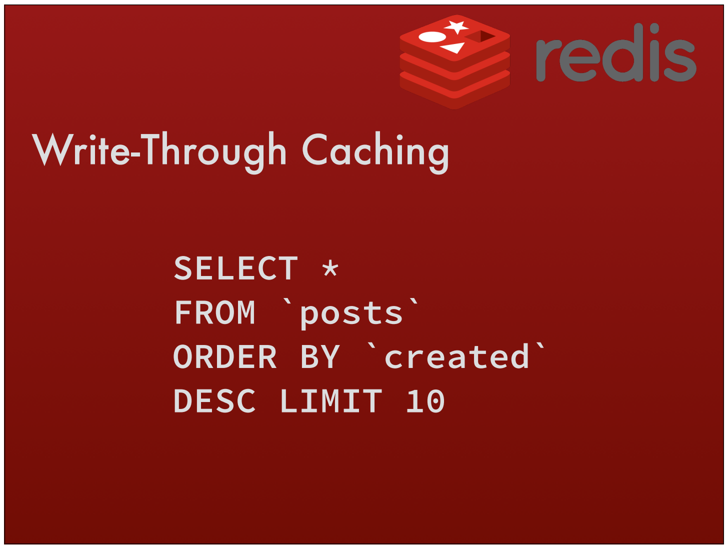
In redis, transactions are isolated and atomic. Commands are processed serially without interruption by other clients, and are either all committed, or none are.

There are more commands for transactions, but these are the core two.

redis

Write-Through Caching

My second use case is more caching, but instead of a typical read-through cache, we will create a write-through cache.



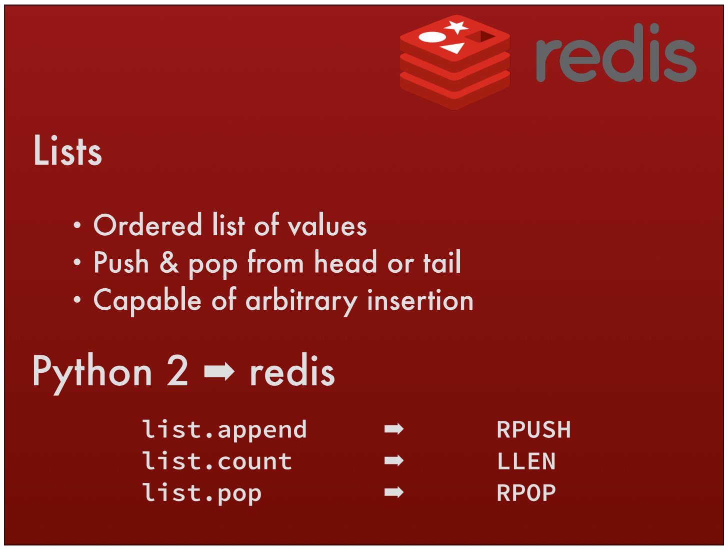
redis

Write-Through Caching

```
SELECT *
FROM `posts`
ORDER BY `created`
DESC LIMIT 10
```

There are many instances in app development when you need to show the N most recent X. 10 most recent posts, for example. The best implementation for timely and accurate data involves cache creation and invalidation on write.

With redis, this is easy to implement through lists.



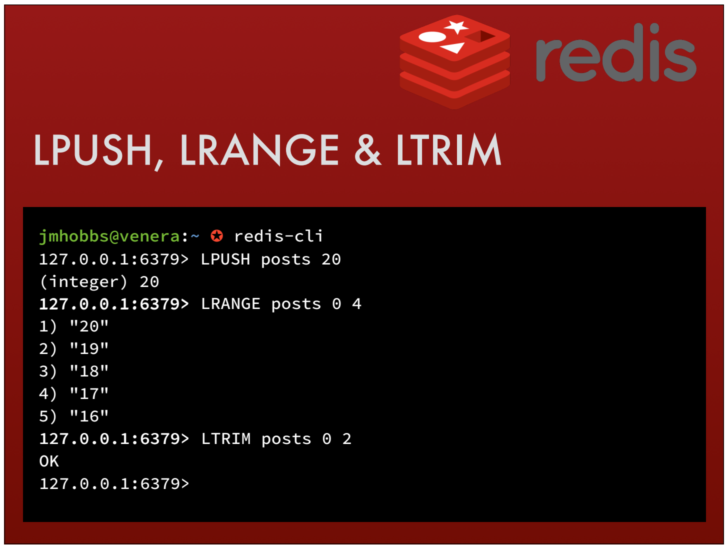
Lists

- Ordered list of values
- Push & pop from head or tail
- Capable of arbitrary insertion

Python 2 ➡ redis

`list.append`	➡	`RPUSH`
`list.count`	➡	`LLEN`
`list.pop`	➡	`RPOP`

The redis list data structure behaves like a list mixed with a queue from most programming languages.



LPUSH, LRANGE & LTRIM

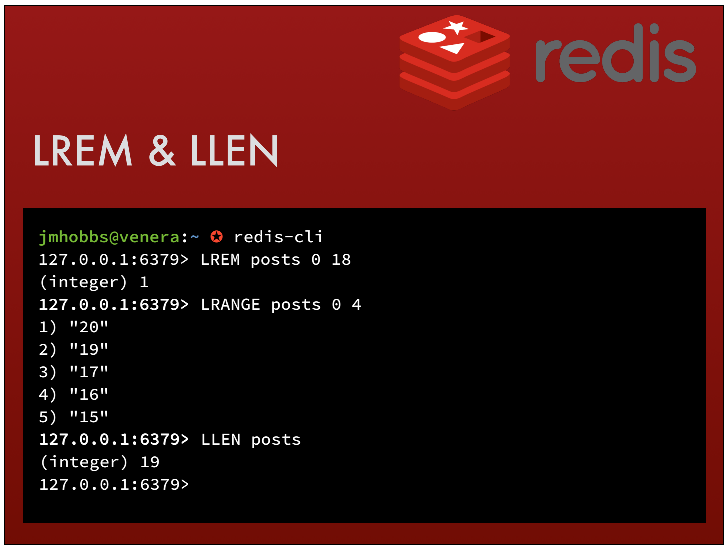
```
jmhobbs@venera:~ ✪ redis-cli
127.0.0.1:6379> LPUSH posts 20
(integer) 20
127.0.0.1:6379> LRANGE posts 0 4
1) "20"
2) "19"
3) "18"
4) "17"
5) "16"
127.0.0.1:6379> LTRIM posts 0 2
OK
127.0.0.1:6379>
```

To implement this cache, we would call LPUSH and LTRIM when inserting a new post into the database.

LPUSH prepends the post ID onto the list, and LTRIM reduces the length of a list.

Note that LRANGE and LTRIM use start and stop indexes, not start and length. Negative indexes are allowed, so -1 is the last item of the list, -2 second to last, etc.

This means that while RPUSH exists for appending to a list, there is no RRANGE or RTRIM, you just have to do the math.

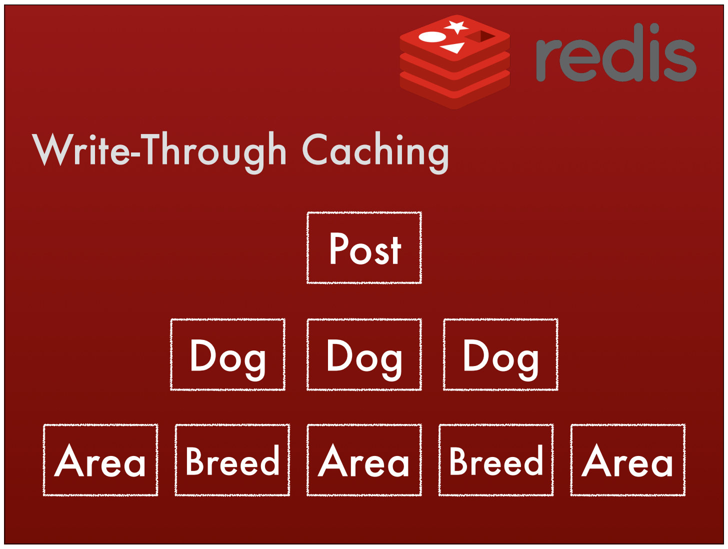


LREM & LLEN

```
jmhobbs@venera:~ ✪ redis-cli
127.0.0.1:6379> LREM posts 0 18
(integer) 1
127.0.0.1:6379> LRANGE posts 0 4
1) "20"
2) "19"
3) "17"
4) "16"
5) "15"
127.0.0.1:6379> LLEN posts
(integer) 19
127.0.0.1:6379>
```

So, what happens if we delete a post? We can either ignore it, if that's a rare event, and just have a short page, or we can use LREM.

Another quick list command is LLEN, which will get you the length of the list. As you see here, it's gotten shorter by 1 because of the LREM.



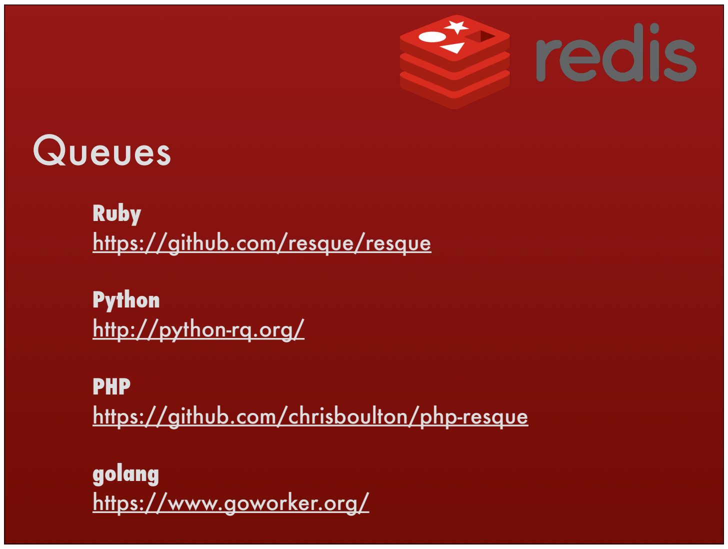
Write-Through Caching

Now, keep in mind that this is a very simple case. Adding an index to the posts table would be far less work and very fast. A more realistic case is where it's computationally expensive.

At Pack we have posts which belong to dogs, which belong to packs. These posts need to show up in the feed of every pack. Looking those up through the join table every page load is more expensive, so the easy solution was a write-through cache.

Queues

I'm going to touch quickly on a third use case, which is queues.



Queues

Ruby
https://github.com/resque/resque

Python
http://python-rq.org/

PHP
https://github.com/chrisboulton/php-resque

golang
https://www.goworker.org/

Redis has inspired a number of simple work queues. These are great for web apps, they let you take expensive processing out of the web app and move it into an isolated service, running on a completely different set of servers if you like.

At pack we use this for sending mail, processing images, updating the search database, all kinds of things that we don't want to slow down the web service with.

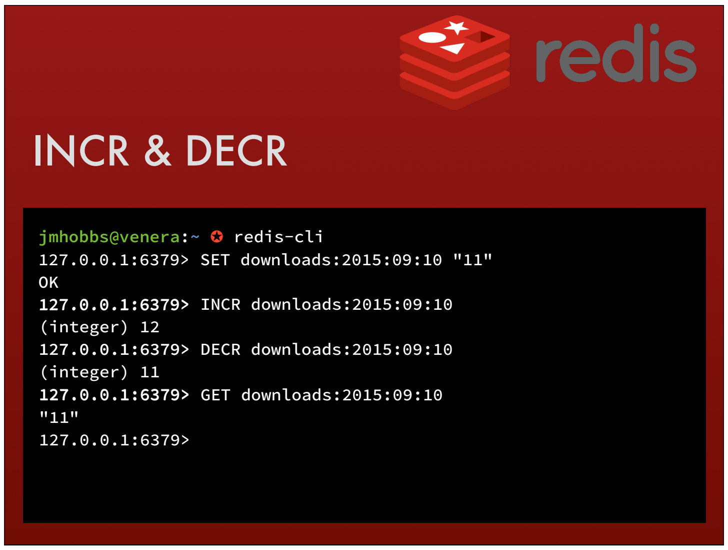
Redis is not a specific queueing backend like RabbitMQ, but the combination of lists, BLPOP, EXPIRE and atomic transactions make it a reasonable replacement, and it's "free" since we are already using redis for other chores like caching.

I should point out that antirez is currently building disque, which I hope will be as easy to use as redis, but provide better durability and message queue features.

Counting Stuff

My fourth case today is counting stuff.

How many times has that file been downloaded today? How about this week?


INCR & DECR

```
jmhobbs@venera:~ ✪ redis-cli
127.0.0.1:6379> SET downloads:2015:09:10 "11"
OK
127.0.0.1:6379> INCR downloads:2015:09:10
(integer) 12
127.0.0.1:6379> DECR downloads:2015:09:10
(integer) 11
127.0.0.1:6379> GET downloads:2015:09:10
"11"
127.0.0.1:6379>
```

Even though the values in redis are strings, redis offers some tools that treat them as numbers.

INCR and DECR are two of those. If INCR by can coerce the string into an integer, it will increment it, if not it will error out. Same thing for DECR.

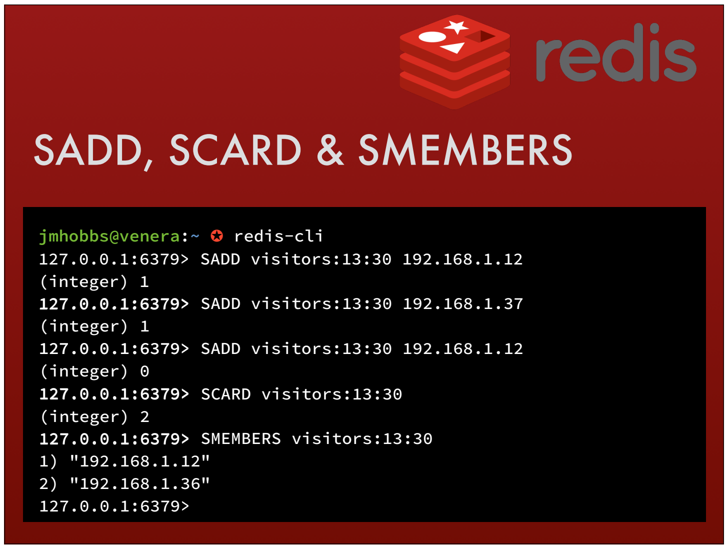
There are also INCRBY and DECRBY for counting by more than one, and a INCRBY for floats.

Notice the use of colon (:) in the key. This is a common pattern in the redis world to create a sense of “namespaces”. There is nothing special about the colon, it’s just a convention. You could as easily use underscores or something else.

Counting Uniques

Counting stuff is cool, but you know what’s cooler? Counting unique stuff.

Say you want to count the number of unique visitors on your website THIS VERY MINUTE.



SADD, SCARD & SMEMBERS

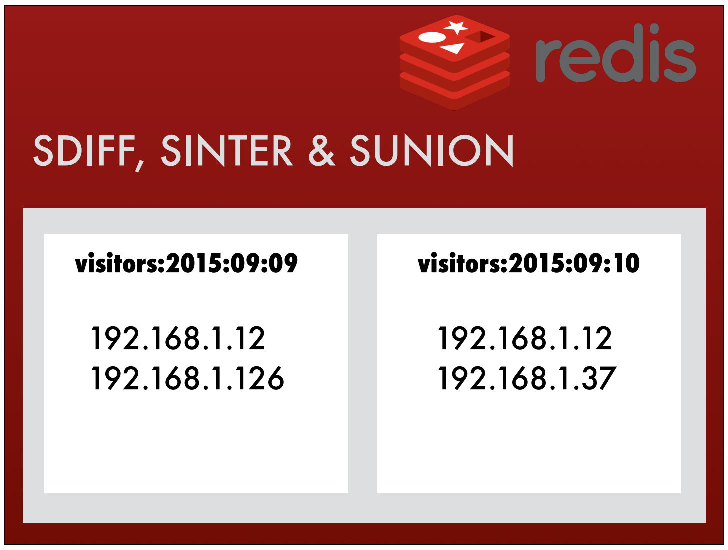
```
jmhobbs@venera:~ ✪ redis-cli
127.0.0.1:6379> SADD visitors:13:30 192.168.1.12
(integer) 1
127.0.0.1:6379> SADD visitors:13:30 192.168.1.37
(integer) 1
127.0.0.1:6379> SADD visitors:13:30 192.168.1.12
(integer) 0
127.0.0.1:6379> SCARD visitors:13:30
(integer) 2
127.0.0.1:6379> SMEMBERS visitors:13:30
1) "192.168.1.12"
2) "192.168.1.36"
127.0.0.1:6379>
```

To accomplish this, we will use the set data type. Sets are sets. They are unordered, but are guaranteed to hold only one of any given value.

SADD will add a value to a set. It won’t complain if the value is already in the set.

SCARD gets the number of users we consider “online”.

We would also want to set an EXPIRE after each SADD so that the sets don’t hang around from day to day.



SDIFF, SINTER & SUNION

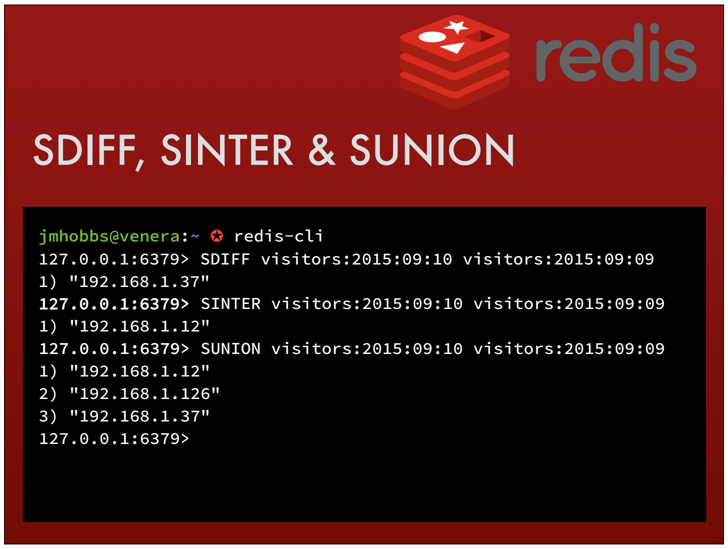
visitors:2015:09:09	visitors:2015:09:10
192.168.1.12	192.168.1.12
192.168.1.126	192.168.1.37

That's pretty useful, but there are other set operations which make it even better.

Want to know who visited today but not yesterday? Count at the day level and use SDIFF. It can work across multiple sets too, so you can see who visited today but no other day this week.

Want to know everyone who visited today or yesterday? Use SUNION.

Want to know who visited yesterday AND today? SINTER can do that.



SDIFF, SINTER & SUNION

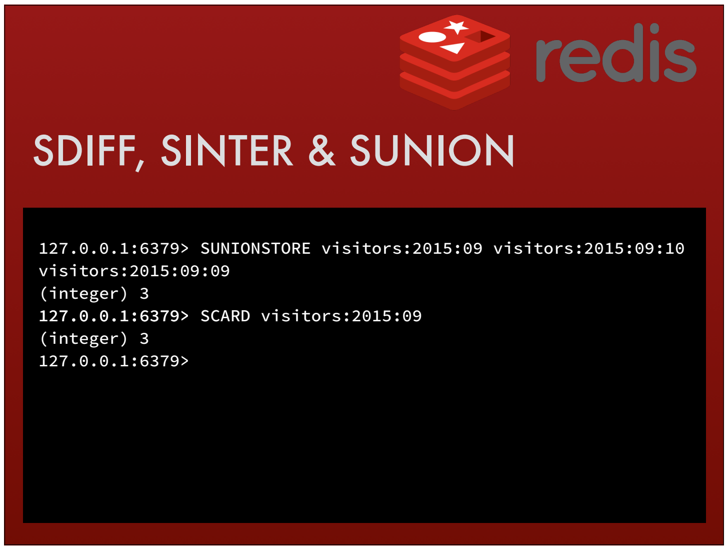
```
jmhobbs@venera:~ ✪ redis-cli
127.0.0.1:6379> SDIFF visitors:2015:09:10 visitors:2015:09:09
1) "192.168.1.37"
127.0.0.1:6379> SINTER visitors:2015:09:10 visitors:2015:09:09
1) "192.168.1.12"
127.0.0.1:6379> SUNION visitors:2015:09:10 visitors:2015:09:09
1) "192.168.1.12"
2) "192.168.1.126"
3) "192.168.1.37"
127.0.0.1:6379>
```

That's pretty useful, but there are other set operations which make it even better.

Want to know who visited today but not yesterday? Count at the day level and use SDIFF. It can work across multiple sets too, so you can see who visited today but no other day this week.

Want to know everyone who visited today or yesterday? Use SUNION.

Want to know who visited yesterday AND today? SINTER can do that.



SDIFF, SINTER & SUNION

```
127.0.0.1:6379> SUNIONSTORE visitors:2015:09 visitors:2015:09:10
visitors:2015:09:09
(integer) 3
127.0.0.1:6379> SCARD visitors:2015:09
(integer) 3
127.0.0.1:6379>
```

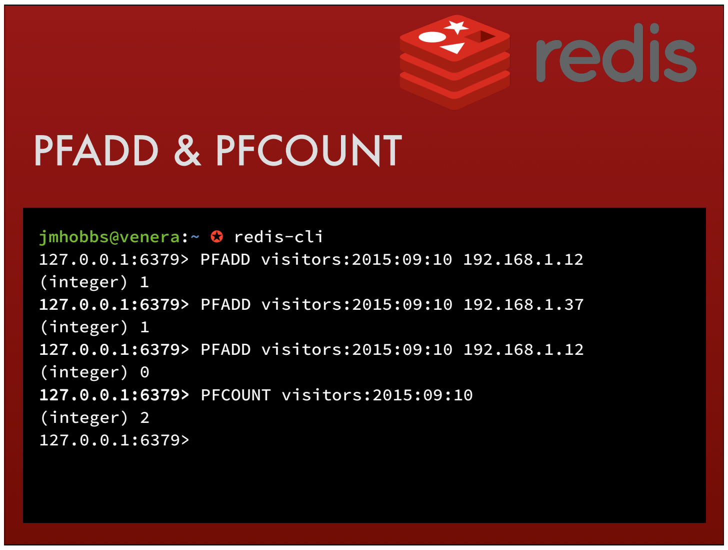
Each of these has a STORE variant, so you can run the union, difference or intersection, store it into another set, and run things like SCARD on it. These are good for rolling up stats and can be executed in a transaction to keep it clean.

Counting Uniques

(again)

Redis offers another way to count uniques.

Redis supports a probabilistic data structure called HyperLogLog, which estimates the cardinality of a set using a tiny fraction of the memory. In exchange for this tiny memory usage, you get loose some precision. In the case of redis, the standard error is less than one percent.



PFADD & PFCOUNT

```
jmhobbs@venera:~ ✪ redis-cli
127.0.0.1:6379> PFADD visitors:2015:09:10 192.168.1.12
(integer) 1
127.0.0.1:6379> PFADD visitors:2015:09:10 192.168.1.37
(integer) 1
127.0.0.1:6379> PFADD visitors:2015:09:10 192.168.1.12
(integer) 0
127.0.0.1:6379> PFCOUNT visitors:2015:09:10
(integer) 2
127.0.0.1:6379>
```

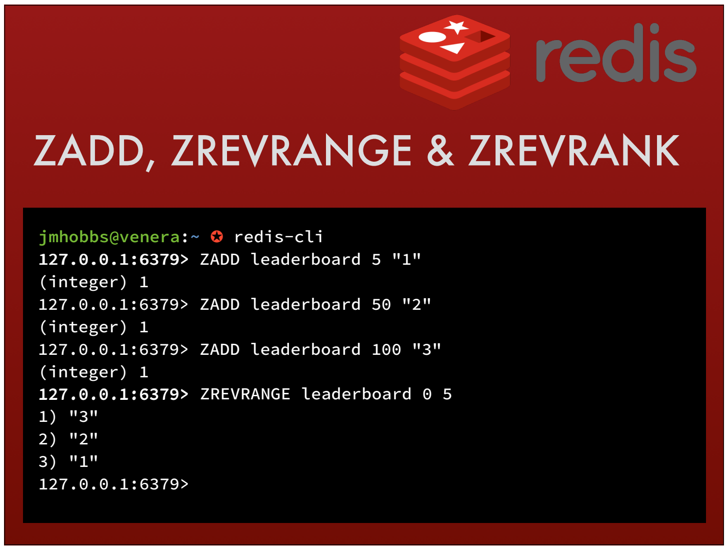
PFADD acts just like SADD, and PFCOUNT acts like SCARD.

There is also a PFMERGE which is like SUNIONSTORE, merging the HyperLogLogs into a new key.

Leaderboards

Leaderboards are something you might think only applies to games, but the same concept can be used for sorting comments, posts, etc. by popularity or age.

Redis has sorted sets. This data structure is like a set in that it can only contain one of each value, but it is ordered. Every value must have an integer score which is used to determine the order of the set. When two values have the same score, ordering is done lexicographically.



ZADD, ZREVRANGE & ZREVRANK

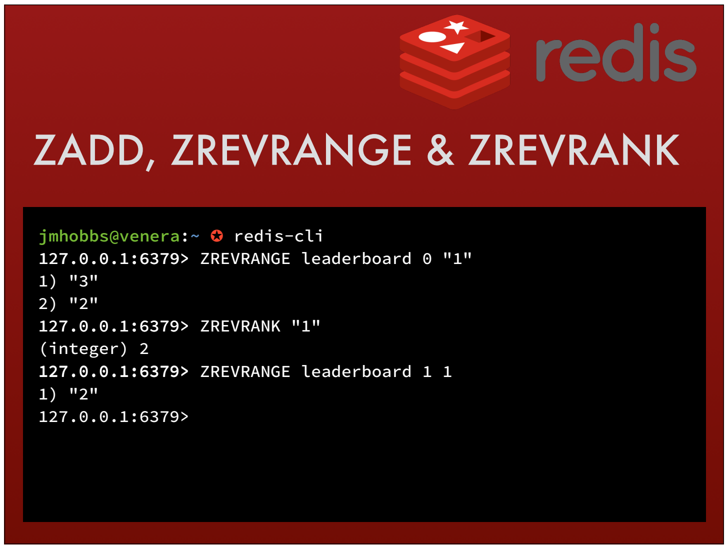
```
jmhobbs@venera:~ ✪ redis-cli
127.0.0.1:6379> ZADD leaderboard 5 "1"
(integer) 1
127.0.0.1:6379> ZADD leaderboard 50 "2"
(integer) 1
127.0.0.1:6379> ZADD leaderboard 100 "3"
(integer) 1
127.0.0.1:6379> ZREVRANGE leaderboard 0 5
1) "3"
2) "2"
3) "1"
127.0.0.1:6379>
```

We use this for our street team leaderboard. The score is a value derived from the interactions our street team members have on the website. For instance commenting on posts, inviting users, etc. Calculating it covers many tables, so it's not great for calculating on read.

The core sorted sets commands ZADD, ZREVRANGE and ZREVRANK.

ZADD key score value

ZREVRANGE returns the requested set members, sorted from high to low.

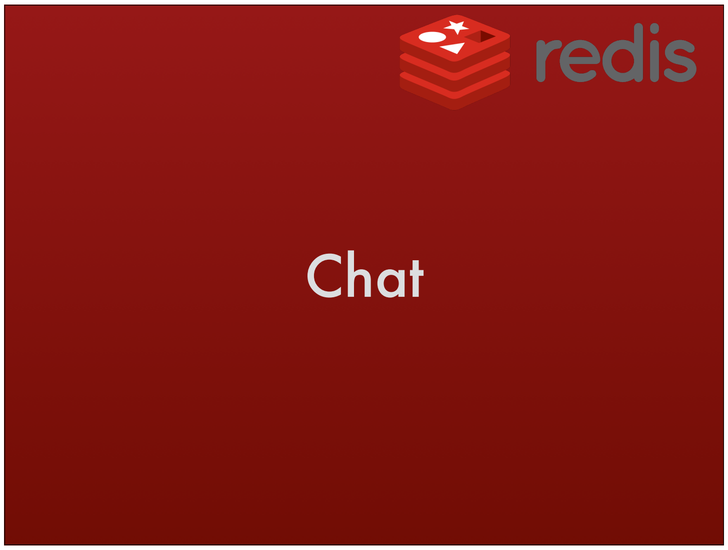


ZADD, ZREVRANGE & ZREVRANK

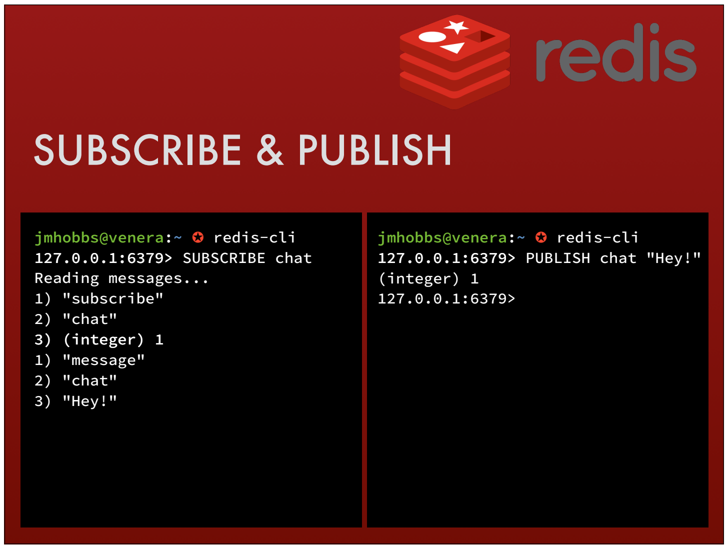
```
jmhobbs@venera:~ ✪ redis-cli
127.0.0.1:6379> ZREVRANGE leaderboard 0 "1"
1) "3"
2) "2"
127.0.0.1:6379> ZREVRANK "1"
(integer) 2
127.0.0.1:6379> ZREVRANGE leaderboard 1 1
1) "2"
127.0.0.1:6379>
```

But what if the active user isn't in the top 10? Where's user 1?

We can get their exact rank with ZREVRANK, then view a "window" of scores above and below the user with ZREVRANGE.


Chat

Redis has a simple, stable, fast implementation of pub/sub. One use case is for event coordination. If you have multiple websocket app servers, you'll want to make sure you deliver the event to the client regardless of which websocket backend they are connected to.


SUBSCRIBE & PUBLISH

```
jmhobbs@venera:~ ✪ redis-cli
127.0.0.1:6379> SUBSCRIBE chat
Reading messages...
1) "subscribe"
2) "chat"
3) (integer) 1
1) "message"
2) "chat"
3) "Hey!"
```

```
jmhobbs@venera:~ ✪ redis-cli
127.0.0.1:6379> PUBLISH chat "Hey!"
(integer) 1
127.0.0.1:6379>
```

With redis pub/sub, this is easy.

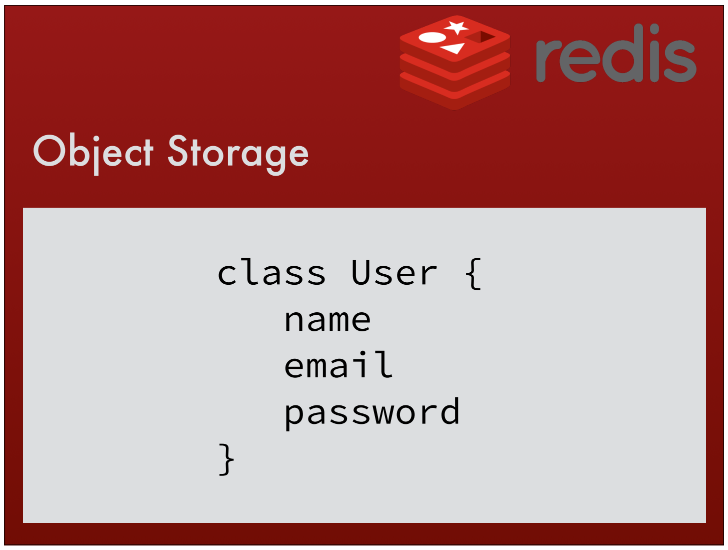
Each websocket server connects to the chat channel with SUBSCRIBE.

Then, when an event happens, the origin uses PUBLISH to send it to all connected servers, which push it down to their clients.

When you issue the SUBSCRIBE command, you enter a streaming mode in which only a subset of redis commands are valid. However, in this mode you can change your channel and pattern subscriptions, so it's easy to change channels without disconnecting and potentially losing messages.

Object Storage

Most of these examples have been about adding redis onto your stack for single attributes of objects, or serialized objects at the most. But you can use redis to store some or all the members of your objects or structs if you want to using hashes.



Object Storage

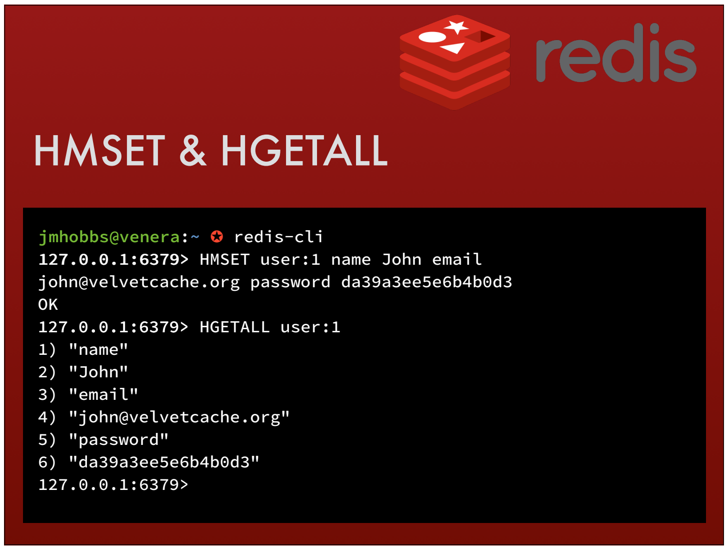
```
class User {
    name
    email
    password
}
```

In redis, hashes are equivalent to what other languages might call a dictionary, map or associative array. It's essentially a nested key-value store on a single key in redis.

Say you have this pseudocode user class, with a name, email and password.

We can store this class with HMSET, an retrieve it with HGETALL.

The downside to this is that you need to maintain your own indexes. If I want to be able to find a user by id, I can by using the id to build the key. If I want to find them by email, I will have to create a sorted set, or just individual keys.



HMSET & HGETALL

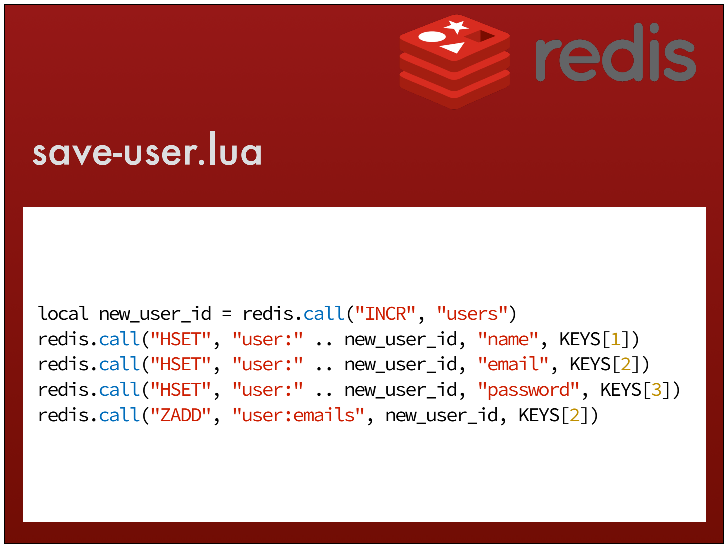
```
jmhobbs@venera:~ ✪ redis-cli
127.0.0.1:6379> HMSET user:1 name John email
john@velvetcache.org password da39a3ee5e6b4b0d3
OK
127.0.0.1:6379> HGETALL user:1
1) "name"
2) "John"
3) "email"
4) "john@velvetcache.org"
5) "password"
6) "da39a3ee5e6b4b0d3"
127.0.0.1:6379>
```

In redis, hashes are equivalent to what other languages might call a dictionary, map or associative array. It's essentially a nested key-value store on a single key in redis.

Say you have this pseudocode user class, with a name, email and password.

We can store this class with HMSET, an retrieve it with HGETALL.

The downside to this is that you need to maintain your own indexes. If I want to be able to find a user by id, I can by using the id to build the key. If I want to find them by email, I will have to create a sorted set, or just individual keys.

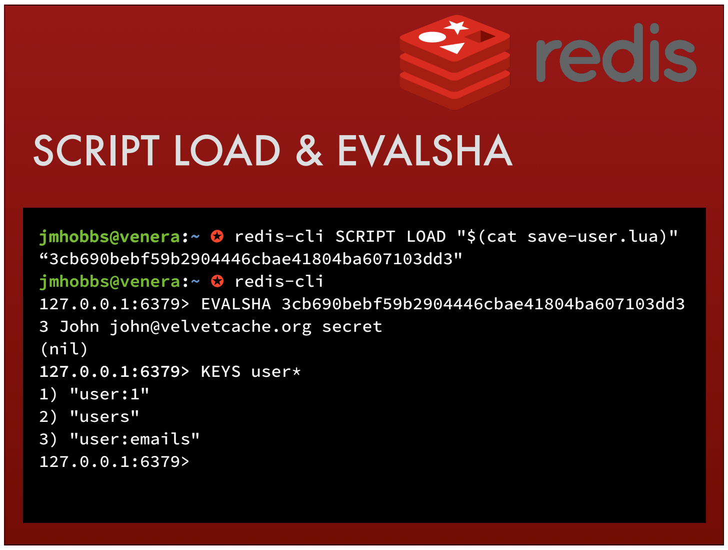


save-user.lua

```
local new_user_id = redis.call("INCR", "users")
redis.call("HSET", "user:" .. new_user_id, "name", KEYS[1])
redis.call("HSET", "user:" .. new_user_id, "email", KEYS[2])
redis.call("HSET", "user:" .. new_user_id, "password", KEYS[3])
redis.call("ZADD", "user:emails", new_user_id, KEYS[2])
```

So let's handle that index work by using lua scripting. Lua is a fast, embeddable scripting language. It shows up a lot in game development, and antirez likes it, so it's embedded in redis. It's a unique language, but it can feel a lot like JavaScript.

I'm not going to try to cover all of Lua, because I don't know a lot of it myself. However, we can at least execute some redis commands.



SCRIPT LOAD & EVALSHA

```
jmhobbs@venera:~ ✪ redis-cli SCRIPT LOAD "$(cat save-user.lua)"
“3cb690bebf59b2904446cbae41804ba607103dd3"
jmhobbs@venera:~ ✪ redis-cli
127.0.0.1:6379> EVALSHA 3cb690bebf59b2904446cbae41804ba607103dd3
3 John john@velvetcache.org secret
(nil)
127.0.0.1:6379> KEYS user*
1) "user:1"
2) "users"
3) "user:emails"
127.0.0.1:6379>
```

To load a lua script into the script cache, you use the SCRIPT LOAD and pass it the content of the script, which we load here via command substitution.

This returns a SHA1 hash of the script, and the script will remain in the redis script cache unless removed with SCRIPT FLUSH.

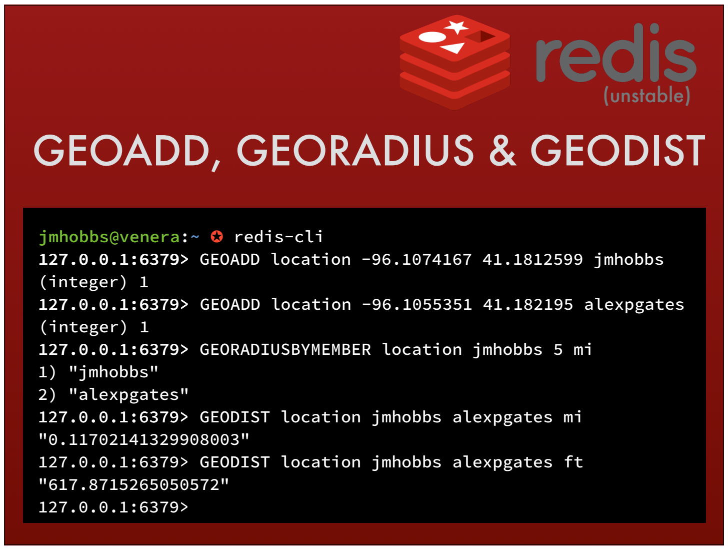
We can then use this hash to execute the script with EVALSHA, and it does it’s magic.

Who’s Nearby?

Something coming soon to Redis is geospatial indexing. Previously, this was an extension you could compile into a special version of redis, now it’s been brought into the unstable and should come out soon.

We can use this to find who is online near us, for a mobile messaging app perhaps.

Redis geo commands are super simple to use compared to many RDMS geo extensions. Definitely better than MySQL.



GEOADD, GEORADIUS & GEODIST

```
jmhobbs@venera:~ ✪ redis-cli
127.0.0.1:6379> GEOADD location -96.1074167 41.1812599 jmhobbs
(integer) 1
127.0.0.1:6379> GEOADD location -96.1055351 41.182195 alexpgates
(integer) 1
127.0.0.1:6379> GEORADIUSBYMEMBER location jmhobbs 5 mi
1) "jmhobbs"
2) "alexpgates"
127.0.0.1:6379> GEODIST location jmhobbs alexpgates mi
"0.11702141329908003"
127.0.0.1:6379> GEODIST location jmhobbs alexpgates ft
"617.8715265050572"
127.0.0.1:6379>
```

So, I'm here at HDC, so I use GEOADD to show that.

My friend Alex is over across the street at the Tropical Island Bar & Grill.

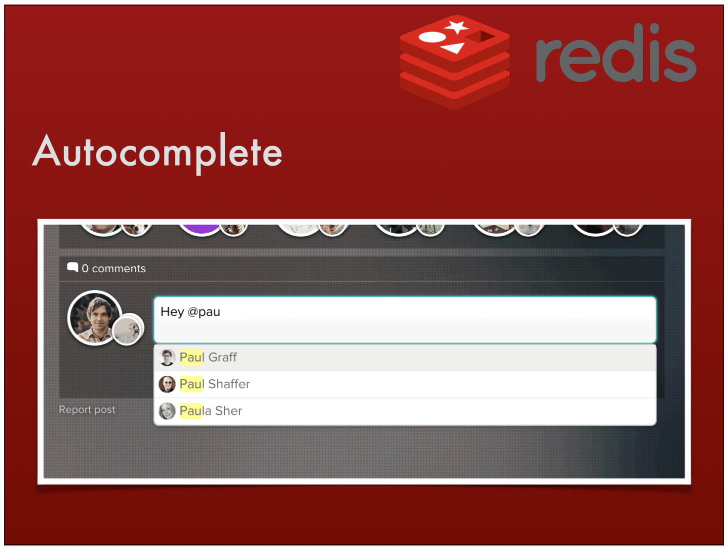
GEORADIUSBYMEMBER answers what users are near me, say within five miles.

There is also a generic GEORADIUS for looking up arbitrary lat/lng.

GEODIST can tell me exactly how far away Alex is, in miles, feet or various metric units

Autocomplete

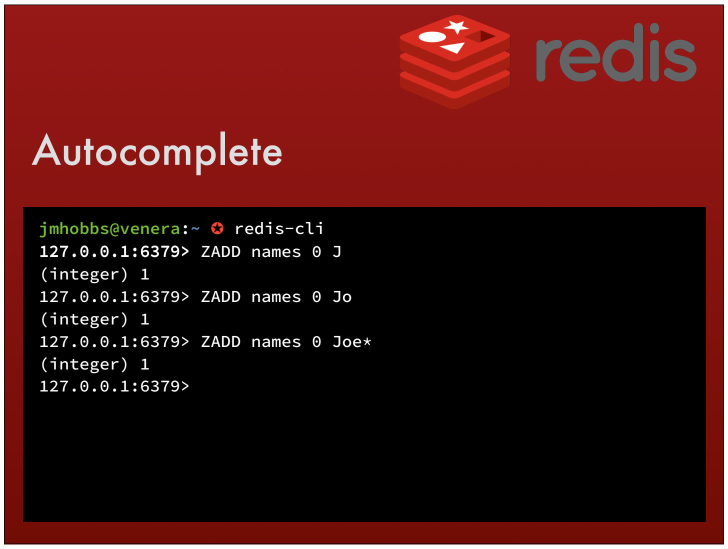
Let's look at one last use case, autocomplete. This is a great application of the speed and strengths of redis, and one we just recently implemented at Pack.



Autocomplete

As part of our comment system, we wanted to add the ability to mentions of your friends. We wanted this to be fast, so we built it using sorted sets and go.

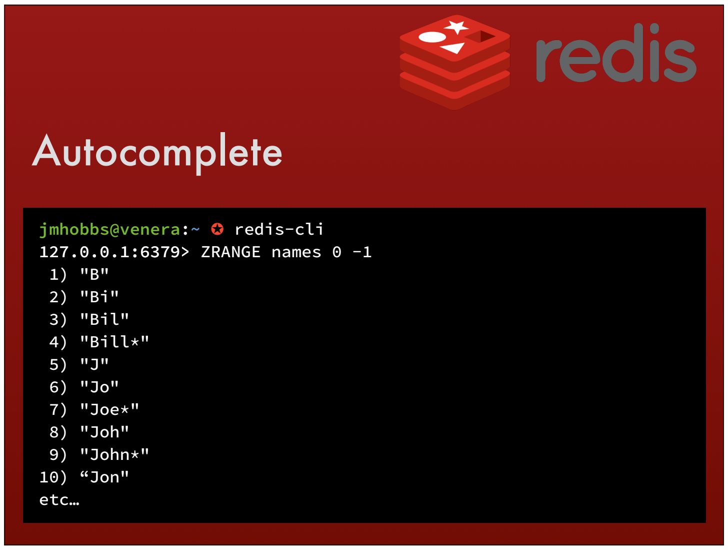
One of the hidden features of sorted sets is that values with equivalent scores are sorted lexicographically. We can leverage that to present autocomplete values quickly and cleanly.



Autocomplete

```
jmhobbs@venera:~ ✪ redis-cli
127.0.0.1:6379> ZADD names 0 J
(integer) 1
127.0.0.1:6379> ZADD names 0 Jo
(integer) 1
127.0.0.1:6379> ZADD names 0 Joe*
(integer) 1
127.0.0.1:6379>
```

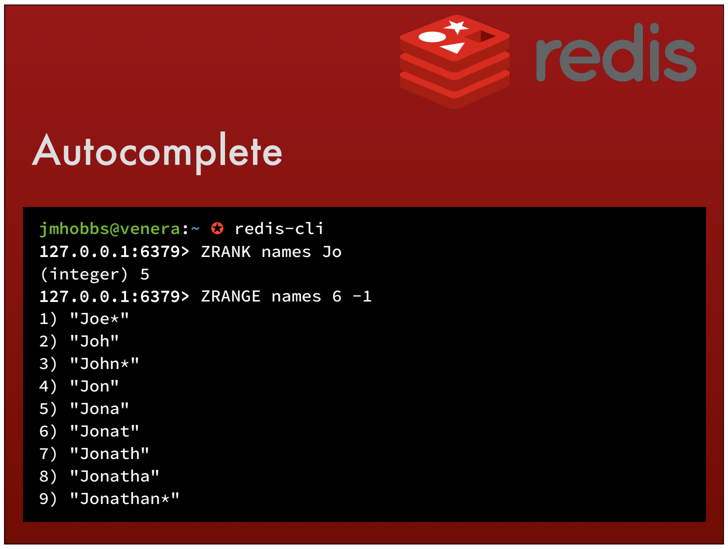
For every word in our autocomplete data set, we add it to the ZSET with a score of 0, building it up one letter at a time. When we have the complete word, we end it with a delimiter, in our case we choose the asterisk.



Autocomplete

```
jmhobbs@venera:~ ✪ redis-cli
127.0.0.1:6379> ZRANGE names 0 -1
 1) "B"
 2) "Bi"
 3) "Bil"
 4) "Bill*"
 5) "J"
 6) "Jo"
 7) "Joe*"
 8) "Joh"
 9) "John*"
10) “Jon"
etc…
```

We do this for every word in our data set.



Autocomplete

```
jmhobbs@venera:~ ✪ redis-cli
127.0.0.1:6379> ZRANK names Jo
(integer) 5
127.0.0.1:6379> ZRANGE names 6 -1
1) "Joe*"
2) "Joh"
3) "John*"
4) "Jon"
5) "Jona"
6) "Jonat"
7) "Jonath"
8) "Jonatha"
9) "Jonathan*"
```

Now, we can use ZRANK to find the index of whatever the user has typed, Jo for example.

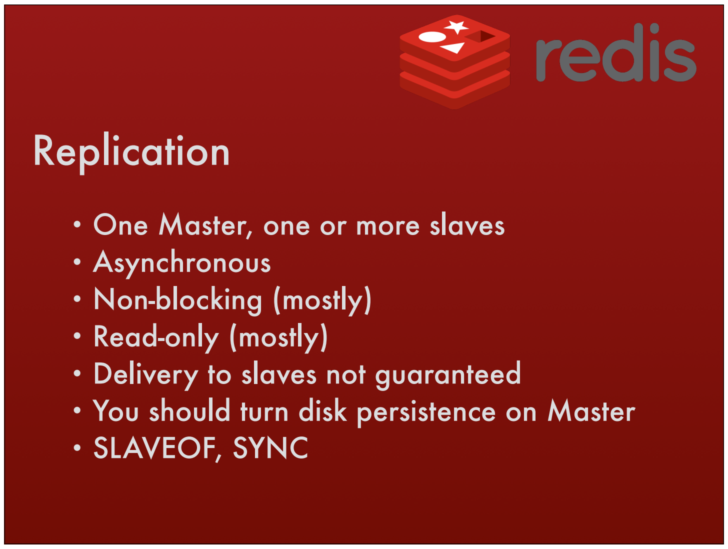
We then use ZRANGE to get all of the entries from our item to the end, and filter for entries that match our prefix, and end in an asterisk.

If we implement this in code, we would loop over ZRANGE reading blocks of 50 until our prefix stopped matching.

In production, we created a serialized format for our result rows which included the user id and full name so we could save the round trip to the MySQL database and show results immediately.

Replication

Ok, that's all the use cases. Let's look briefly at replication in redis.



Replication

- One Master, one or more slaves
- Asynchronous
- Non-blocking (mostly)
- Read-only (mostly)
- Delivery to slaves not guaranteed
- You should turn disk persistence on Master
- SLAVEOF, SYNC

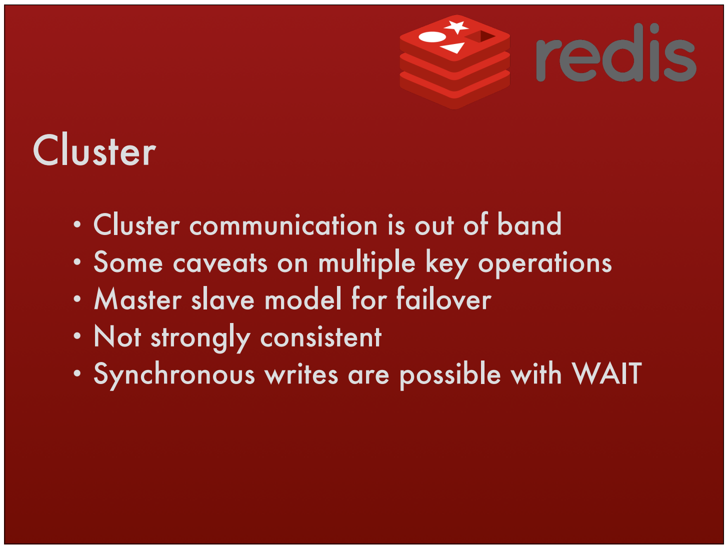
Non-blocking except slaves during initial sync. You can configure to use an old dataset during that time.

Read only by default, but it's currently possible to have writable slaves that desync.

Writes on the master are accepted regardless of slave status. Starting with Redis 2.8 you can have the master reject writes if slaves are not connected or are lagging to far behind.

Cluster

Redis cluster is a way to run redis and shard the data across multiple nodes, and keep data available even when nodes in the cluster are down.



Cluster

- Cluster communication is out of band
- Some caveats on multiple key operations
- Master slave model for failover
- Not strongly consistent
- Synchronous writes are possible with WAIT

For multiple key operations, keys have to belong to the same "hash slot", which is how redis divides keys between nodes. Keys can be coerced into the same hash slot with hash tags.

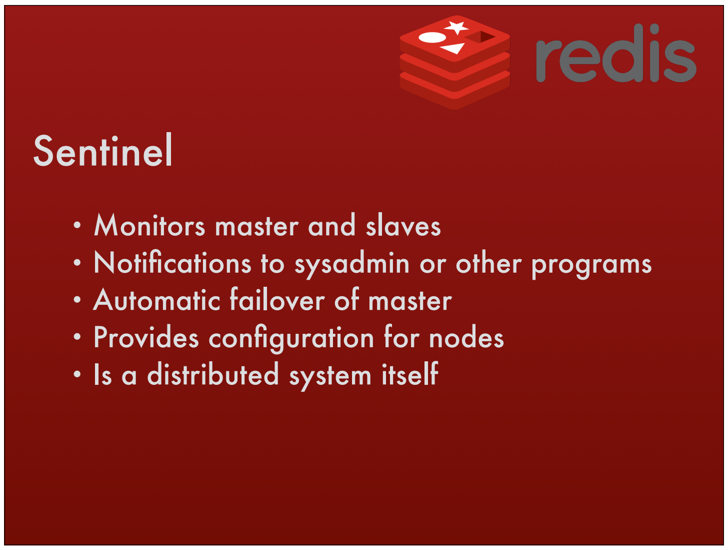
Every hash slot needs a master and at least one slave for failover, otherwise the cluster will halt.

Clustering is not, by default, strongly consistent. You have the same issue as replication with lost writes.

The WAIT command can give you synchronous writes, but it's not guaranteed that all slaves have gotten the write.

Sentinel

Redis sentinel is a second application which provides high availability for redis.



Sentinel

- Monitors master and slaves
- Notifications to sysadmin or other programs
- Automatic failover of master
- Provides configuration for nodes
- Is a distributed system itself

Sentinel constantly checks on the health and synchronization status of your master and slaves.
It can notify your ops team if things aren't healthy.
It can promote a slave to master, and reconfigure the other slaves.
It is the single source of truth for replication.
Sentinel itself is distributed, and you should spread out your instances.
Sentinel instances require a quorum to decide on master health. You need more than two instances to prevent immediate split-brain. Sentinels can run on client boxes too.

Who uses redis?

Social proof is a driving force in tech adoption, so who has been using redis in the last six and a half years of its life?

Who uses redis?

- Instagram
- Weibo
- Twitter
- Tumblr

Instagram has been a user of redis for a long time, and they have a number of blog posts about their big deployments of redis.

Weibo is a massively popular Chinese microblogging platform, like Twitter. They have a caching deployment of redis they like to brag about online, with over 200 nodes in it.

In 2014, Twitter reported they were using 105TB of RAM and achieving 39MM QPS.

In 2012, Tumblr reported using redis for cache and for ephemeral notifications, with plans to expand usage. Hundreds of redis instances on tall servers.

Thank You

http://velvetcache.org
http://twitter.com/jmhobbs

john@velvetcache.org

I'll be posting the slides online, you should be able to find them on twitter or my website. I'll also post links to resources I used for this presentation.

If anyone has any final questions, please ask.