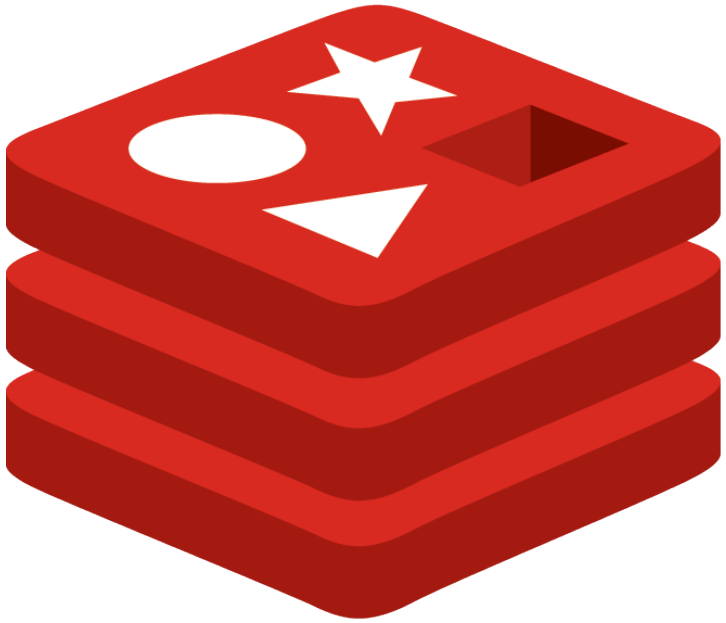

redis

"Little Server of Awesome"

2011 Dvir Volk

Software Architect, Do@
dvir@doat.com http://doat.com

What is redis

- Memcache-ish in-memory key/value store
- But it's also persistent!
- And it also has very cool value types:
  - lists
  - sets
  - sorted sets
  - hash tables
  - append-able buffers
- Open source; very helpful and friendly community. Development is very active and responsive to requests.
- Sponsored by VMWare
- Used in the real world: github, craigslist, engineyard, ...
- Used heavily in do@ as a front-end database, search, geo resolving

Key Features and Cool Stuff

- All data is in memory (almost)
- All data is eventually persistent (But can be immediately)
- Handles huge workloads easily
- Mostly O(1) behavior
- Ideal for write-heavy workloads
- Support for atomic operations
- Supports for transactions
- Has pub/sub functionality
- Tons of client libraries for all major languages
- Single threaded, uses aync. IO
- Internal scripting with LUA coming soon

A little benchmark

This is on my laptop (core i7 @2.2Ghz)

- SET: 187265.92 requests per second
- GET: 185185.17 requests per second
- INCR: 190114.06 requests per second

- LPUSH: 190114.06 requests per second
- LPOP: 187090.73 requests per second

-
- SADD: 186567.16 requests per second
- SPOP: 185873.61 requests per second

Scaling it up

- Master-slave replication out of the box
- Slaves can be made masters on the fly
- Currently does not support "real" clustered mode....
- ... But Redis-Cluster to be released soon
- You can manually shard it client side
- Single threaded - run num_cores/2 instances on the same machine

Persistence

- All data is synchronized to disk - eventually or immediately
- Pick your risk level Vs. performance
- Data is either dumped in a forked process, or written as a append-only change-log (AOF)
- Append-only mode supports transactional disk writes so you can lose no data (cost: 99% speed loss :))
- AOF files get huge, but redis can minimize them on the fly.
- You can save the state explicitly, background or blocking

- Default configuration:
  - Save after 900 sec (15 min) if at least 1 key changed
  - Save after 300 sec (5 min) if at least 10 keys changed
  - Save after 60 sec if at least 10000 keys changed

Virtual Memory

- If your database is too big - redis can handle swapping on its own.
- Keys remain in memory and least used values are swapped to disk.
- Swapping IO happens in separate threads
- **But if you need this - don't use redis, or get a bigger machine ;)**

Show me the features!

Now let's see the key featurs:

- Get/Set/Incr - strings/numbers
- Lists
- Sets
- Sorted Sets
- Hash Tables
- PubSub
- SORT
- Transactions

We'll use redis-cli for the examples.
Some of the output has been modified for readability.

The basics...

Get/Sets - nothing fancy. Keys are strings, anything goes - just quote spaces.

redis> SET foo "bar"
OK
redis> GET foo
"bar"

You can atomically increment numbers

redis> SET bar 337
OK
redis> INCRBY bar 1000
(integer) 1337

Getting multiple values at once

redis> MGET foo bar
1. "bar"
2. "1337"

Keys are lazily expired

redis> EXPIRE foo 1
(integer) 1
redis> GET foo
(nil)

Be careful with EXPIRE - re-setting a value without re-expiring it will remove the expiration!

Atomic Operations

GETSET puts a different value inside a key, retriving the old one

redis> SET foo bar
OK
redis> GETSET foo baz
"bar"
redis> GET foo
"baz"

SETNX sets a value only if it does not exist

redis> SETNX foo bar
OK
redis> SETNX foo baz
FAILS

SETNX + Timestamp => Named Locks! w00t!

redis> SETNX myLock <current_time>
OK
redis> SETNX myLock <new_time>
FAILS

Note that If the locking client crashes that might cause some problems, but it can be solved easily.

List operations

- Lists are your ordinary linked lists.
- You can push and pop at both sides, extract range, resize, etc.
- Random access and ranges at O(N)! :-(

```
redis> LPUSH foo bar
(integer) 1

redis> LPUSH foo baz
(integer) 2

redis> LRANGE foo 0 2
1. "baz"
2. "bar"

redis> LPOP foo
"baz"
```

- **BLPOP: Blocking POP - wait until a list has elements and pop them. Useful for realtime stuff.**

```
redis> BLPOP baz 10 [seconds]
..... We wait!
```

Set operations

- Sets are... well, sets of unique values w/ push, pop, etc.
- Sets can be intersected/diffed /union'ed server side.
- Can be useful as keys when building complex schemata.

```
redis> SADD foo bar
(integer) 1
redis> SADD foo baz
(integer) 1
redis> SMEMBERS foo
["baz", "bar"]

redis> SADD foo2 baz // << another set
(integer) 1
redis> SADD foo2 raz
(integer) 1

redis> SINTER foo foo2 // << only one common element
1. "baz"
redis> SUNION foo foo2 // << UNION
["raz", "bar", "baz"]
```

Sorted Sets

- Same as sets, but with score per element
- Ranked ranges, aggregation of scores on INTERSECT
- Can be used as ordered keys in complex schemata
- Think timestamps, inverted index, geohashing, ip ranges

```
redis> ZADD foo 1337 hax0r
(integer) 1
redis> ZADD foo 100 n00b
(integer) 1
redis> ZADD foo 500 luser
(integer) 1

redis> ZSCORE foo n00b
"100"

redis> ZINCRBY foo 2000 n00b
"2100"

redis> ZRANK foo n00b
(integer) 2
```

```
redis> ZRANGE foo 0 10
1. "luser"
2. "hax0r"
3. "n00b"

redis> ZREVRANGE foo 0 10
1. "n00b"
2. "hax0r"
3. "luser"
```

Hashes

- Hash tables as values
- Think of an object store with atomic access to object members

```
redis> HSET foo bar 1
(integer) 1
redis> HSET foo baz 2
(integer) 1
redis> HSET foo foo foo
(integer) 1

redis> HGETALL foo
{
  "bar": "1",
  "baz": "2",
  "foo": "foo"
}
```

```
redis> HINCRBY foo bar 1
(integer) 2

redis> HGET foo bar
"2"

redis> HKEYS foo
1. "bar"
2. "baz"
3. "foo"
```

PubSub - Publish/Subscribe

- Clients can subscribe to channels or patterns and receive notifications when messages are sent to channels.
- Subscribing is O(1), posting messages is O(n)
- Think chats, Comet applications: real-time analytics, twitter

```
redis> subscribe feed:joe feed:moe feed:
boe

//now we wait
....
                                    <<<<<----------

1. "message"
2. "feed:joe"
3. "all your base are belong to me"
```

```
redis> publish feed:joe "all your base are
belong to me"
(integer) 1 //received by 1
```

SORT FTW!

- Key redis awesomeness
- Sort SETs or LISTS using external values, and join values in one go:

```
SORT key
SORT key BY pattern (e.g. sort userIds BY user:*->age)
SORT key BY pattern GET othervalue

SORT userIds BY user:*->age GET user:*->name
```

- ASC|DESC, LIMIT available, results can be stored, sorting can be numeric or alphabetic

- Keep in mind that it's blocking and redis is single threaded. Maybe put a slave aside if you have big SORTs

Transactions

- MULTI, ..., EXEC: Easy because of the single thread.
- All commands are executed **after** EXEC, block and return values for the commands as a list.
- Example:

```
redis> MULTI
OK
redis> SET "foo" "bar"
QUEUED
redis> INCRBY "num" 1
QUEUED
redis> EXEC
1) OK
2) (integer) 1
```

- Transactions can be discarded with DISCARD.

- WATCH allows you to lock keys while you are queuing your transaction, and avoid race conditions.

Gotchas, Lessons Learned

- Memory fragmentation can be a problem with some usage patterns. Alternative allocators (jemalloc, tcmalloc) ease that.
- Horrible bug with Ubuntu 10.x servers and amazon EC2 machines [resulted in long, long nights at the office...]
- 64 bit instances consume much much more RAM.
- Master/Slave sync far from perfect.
- DO NOT USE THE KEYS COMMAND!!!
- vm.overcommit_memory = 1
- Use MONITOR to see what's going on

Example: *Very* Simple Social Feed

```
#let's add a couple of followers
>>> client.rpush('user:1:followers', 2)
>>> numFollowers = client.rpush('user:1:followers', 3)
>>> msgId = client.incr('messages:id') #ATOMIC OPERATION

#add a message
>>> client.hmset('messages:%s' % msgId, {'text': 'hello world', 'user': 1})

#distribute to followers
>>> followers = client.lrange('user:1:followers', 0, numFollowers)

>>> pipe = client.pipeline()
>>> for f in followers:
    pipe.rpush('user:%s:feed' % f, msgId)
>>> pipe.execute()

>>> msgId = client.incr('messages:id') #increment id
#....repeat...repeat..repeat..repeat..
#now get user 2's feed
>>> client.sort(name = 'user:2:feed', get='messages:*->text')
['hello world', 'foo bar']
```

Other use case ideas

- Geo Resolving with geohashing
  - Implemented and opened by yours truly https://github.com/doat/geodis
- Real time analytics
  - use ZSET, SORT, INCR of values
- API Key and rate management
  - Very fast key lookup, rate control counters using INCR
- Real time game data
  - ZSETs for high scores, HASHES for online users, etc
- Database Shard Index
  - map key => database id. Count size with SETS
- Comet - no polling ajax
  - use BLPOP or pub/sub
- Queue Server
  - resque - a large portion of redis' user base

Melt - My little evil master-plan

- We wanted freakin' fast access to data on the front-end.
- but our ability to cache personalized and query bound data is limited.
- Redis to the rescue!
- But we still want the data to be in an RDBMs.
- So we made a framework to "melt the borders" between them...

Introducing melt

- **ALL** front end data is in RAM, denormalized and optimized for speed. Front end talks only to Redis.
- We use Redis' set features as keys and scoring vectors.
- All back end data is on mysql, with a manageable normalized schema. The admin talks only to MySQL.
- A sync queue in the middle keeps both ends up to date.
- A straightforward ORM is used to manage and sync the data.
- Automates indexing in Redis, generates models from MySQL.
- Use the same model on both ends, or create conversions.
- Central Id generator.

Melt - an example:

```
#syncing objects:
with MySqlStore:
    users = Users.get({Users.id: Int(1,2,3,4)})
    with RedisStore:
        for user in users:
            Users.save(user)


#pushing a new feed item from front to back:
with RedisStore:
    #create an object - any object!
    feedItem = FeedItem(userId, title, time.time())
    #use the model to save it
    Feed.save(feedItem)
    #now just tell the queue to put it on the other side
    SyncQueue.pushItem(action = 'update', model = FeedItem,
      source = 'redis', dest = 'mysql',
       id = feedItem.id)
```

Coming soon to a github near you! :)

More resources

Redis' website:
http://redis.io

Excellent and more detailed presentation by Simon Willison:
http://simonwillison.net/static/2010/redis-tutorial/

Much more complex twitter clone:
http://code.google.com/p/redis/wiki/TwitterAlikeExample

Full command reference:
http://code.google.com/p/redis/wiki/CommandReference