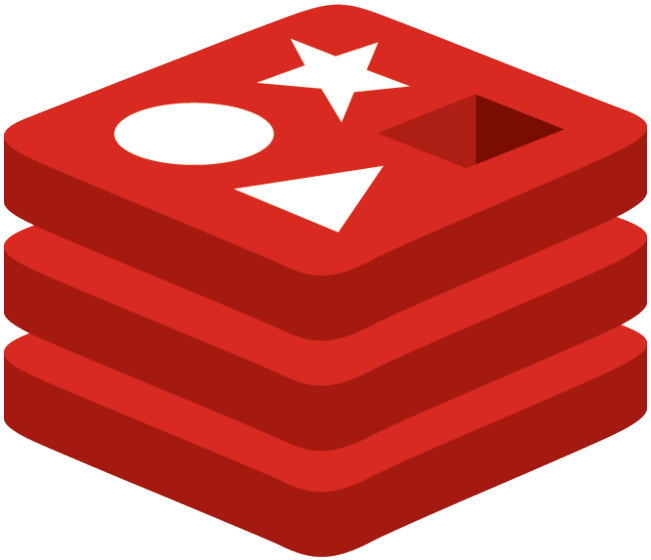
Intro to Redis

Matthew Frazier – NC State FOSS Fair 2012



redis

What is Redis?

- A high-performance server for networked data structures (and some other stuff)
- Non-relational database ("NoSQL")
- Open source (BSD license)
- Created by Salvatore Sanfilippo (@antirez)
  - Really smart and nice guy
  - Kind of a perfectionist ("Quality, or Death :)")
  - Works for VMWare

Features of Redis

- Useful data structures:
  strings, sets, lists, hashes, and sorted sets
- Keeps dataset in RAM, but persists to disk
- Publish/subscribe messaging
- Simple network protocol and API
- Easy to build and deploy
- Well-documented and tested
- Faster than Sonic the Hedgehog on espresso

Live Demo

1: Building and Benchmarking Redis

Good Charlotte, that's fast

- Written in ANSI C
- Custom evented I/O library
- Keeps entire dataset in RAM
  - Still persists to disk 100% reliably with AOF
  - Unfortunately this means your entire dataset has to *fit* in RAM

Data Model

Overview

- Key-value database
- Keys are binary-safe strings
  - (though generally people avoid whitespace and funky binary data)
- Values can be strings, or an assortment of data structures

Data Types

- Strings – binary-safe strings up to 1 GB
- Sets – hash table-backed sets
- Lists – doubly linked lists
- Hashes – hash tables mapping keys to values
- Sorted sets (zsets) – values stored with associated floating-point scores
  - hash table + skip list

API

- Redis API is based on commands
  - e.g. SET key value; SINTER key [key…]
- Issue commands with a simple text-based protocol
- Most commands operate on one data type
  - S for set, L/R for list, H for hash, Z for sorted set
  - raise errors for mismatched types
  - treat nonexistent keys like empty containers
- Each command is guaranteed to be atomic
- Also commands for pub/sub and server management

Implications

- Use the best data structure for each piece of data
- Assemble them into more complex data structures
- Very low-level – you have to glue things together on the client side

What You Can Do With It

Operations on Anything

- TYPE key – return the key's data type
- DEL key [key…] – delete the key
- EXISTS key – check whether the key exists
- EXPIRE key seconds, EXPIREAT key timestamp – mark keys to be deleted later
- RENAME key newkey – atomically rename the key
- RENAMENX key newkey – rename if the destination key does not exist

Use Case: Caching

- Use Redis for caching stuff instead of memcached
  - SET cache:articles:32 “<!doctype html>…”
  - EXPIRE cache:articles:32 60
- 60 seconds later (or when Redis reaches the memory limit), cache:articles:32 is automatically eliminated
- Not just strings – you can also cache all sorts of other data

Strings

- GET key – return the value of key (if it's a string)
- SET key value – set the value of key
- GETSET key value – set the value of key and return the old value
- SETNX key value – set the value of key if it doesn't exist
- MGET key [key…], MSET key value [key value…], MSETNX key value [key value…]

Use Case: Sessions and Things

- Session data is frequently stored in a relational database
  - Usually in serialized form in a BLOB column
  - Also OAuth tokens, CSRF tokens, etc. etc.
- Storing transient data in Redis is more efficient
  - SET sessions:6a87ac3c <serialized_session_data>
  - GET sessions:6a87ac3c
- SETEX lets you SET and EXPIRE simultaneously
  - SETEX sessions:6a87ac3c 604800 <serialized_session_data>
  - Automatically cleans up the session in a week(ish)

Use Case: Locks

- Complicated but absolutely reliable distributed lock algorithm:

```
while True:
    if SETNX(key, expire_time):
        return True
    else:
        timestamp = int(GET(key))
        if timestamp > time.time():
            timestamp = int(GETSET(key, expire_time))
            if timestamp > time.time():
                return True
        else:
            time.sleep(5)
```

Strings as Buffers

- Of bytes:
  - STRLEN key – get the length of the string at key
  - GETRANGE key start end – get part of a key
  - SETRANGE key offset value – replace part of a key
- Of bits:
  - GETBIT key offset – return the value of the bit at the given offset
  - SETBIT key offset value – sets a specific bit in the string

Strings as Counters

- Treat the key as a signed 64-bit integer
  - INCR key
  - INCRBY key increment
  - DECR key
  - DECRBY key increment
- Redis can actually store the string as an integer internally

Use Case: Stat Counting

- Using counters to track multiple statistics
  - INCR hits:url:{SHA1 of URL}
  - INCR hits:day:2012-03-17
  - INCR hits:urlday:{SHA1 of URL}:2012-03-17
  - INCR hits:country:us
  - And so on…
- Each counter is cheap since it's stored as a machine int
  - Also easily shardable

Use Case: Unique ID's

- When using Redis as a primary datastore, use INCR to get the next available ID
  - INCR articles:maxid
  - For the first one, this will return 1
  - For all subsequent, it will return one not already in use

Sets

- SADD key member [member…] – add members to a set [O(1)*]
- SREM key member [member…] – remove members from a set [O(1)*]
- SMEMBERS key – return every member of a set [O(N)]
- SISMEMBER key member – check whether an item is in the set [O(1)]
- SCARD key – return the cardinality of a set [O(1)]
- SPOP key – delete and return a random member [O(1)]
- SRANDMEMBER key – just return a random member [O(1)]

Use Case: Collections of Stuff

- All items:
  - SADD articles:all 45
- Tagging:
  - SADD article:45:tags redis
  - SADD articles:tagged:redis 45
- Redis optimizes sets consisting entirely of integers to reduce memory usage

Use Case: Random Stuff

- Get a random article ID:
  - SRANDMEMBER articles:all
- Far more efficient than ORDER BY RAND() LIMIT 1
  - Redis: O(1)
  - MySQL: O(VER NINE THOUSAAAAAND!)
- You can even use this if your data is primarily in another datastore

Multiple Sets Simultaneously

- SMOVE source destination member – move member from source to destination atomically
- SUNION key [key…] – return all the items that are in any set [O(N)]
- SINTER key [key…] – return all the items that are in each specified set [O(N*M) worst case]
- SDIFF key [key…] – return the set difference of the first set with the rest [O(N)]
- Also STORE versions of these three:
  - SUNIONSTORE destination key [key…]
  - SINTERSTORE destination key [key…]
  - SDIFFSTORE destination key [key…]

Lists – Unpaired Operations

- LLEN key – return the length of the list [O(1)]
- LRANGE key start stop – slice the list [O(S+N)]
- LINDEX key index – get the value at a certain index [O(N)]
- LSET key index value – replace the value at a specific index [O(N)]
- LINSERT key BEFORE|AFTER pivot value – insert a value somewhere in the list [O(N)]
- LREM key count value – delete values from the list [O(N)]
- LTRIM key start stop – trims a list to a specific range [O(N)]

Lists – L/R Paired Operations

- L is head (index 0), R is tail (index 1)
- LPUSH/RPUSH key value [value…] – add values at the head/tail of the list [O(1)*]
- LPUSHX/RPUSHX key value – append/prepend a value if the list exists [O(1)]
- LPOP/RPOP key – remove and return the value at the head/tail of the list [O(1)]
- RPOPLPUSH source destination – move a value from the tail of one list to the head of another [O(1)]
  - …why no LPOPRPUSH?

Use Case: Capped Collections

- Maintain a list of the 100 most recent comments
  - RPUSH comments:latest 838
  - LTRIM 0 99
- Also good for things like social media streams

Lists – Blocking Operations

- BLPOP/BRPOP key [key…] timeout
  - If there's an element at the head/tail of any of the provided lists, returns it (and the list it came from)
  - If there isn't, the server will block the client up to timeout seconds until there is one
  - For multiple clients blocking, it's first come, first served
- BRPOPLPUSH source destination timeout
  - Behaves like BRPOP, but also prepends the returned value to destination once it's returned
  - …why no BLPOPRPUSH?

Use Case: Task Queues

- Add jobs to the queue with:
  - RPUSH queue:mail <ID or serialized job data>
- Then have a bunch of workers running:
  - BLPOP 0 queue:mail
- All jobs posted will get sent to a waiting client ASAP
- Since BLPOP returns both the key and the value, you can wait on multiple jobs with:
  - BLPOP 0 queue:mail queue:trackback queue:archive

Hashes

- HGET key field – get a field from a hash [O(1)]
- HSET key field value – set a hash field [O(1)]
- HDEL key field – delete a field from the hash [O(1)]
- HEXISTS key field – check whether the hash field exists [O(1)]
- HLEN key – return the number of fields in the hash [O(1)]

Hashes continued

- HGETALL key – return all the fields and values of the hash [O(N)]
- HMGET key field [field…], HMSET key field value [field value…] [O(1)*]
- HKEYS/HVALUES key – return the field names or values for the hash, in no particular order [O(N)]
- HINCRBY key field increment – treat the hash field as an integer and increment or decrement it [O(1)]
- HSETNX key field value – set the hash field if it is not already set [O(1)]

Use Case: Object Records

- For object records with scalar fields, hashes are more memory-efficient than separate keys
  - Create: HMSET users:1000 name matthew home /home/matthew shell /bin/bash
  - Retrieve: HGETALL users:1000
  - Update: HSET users:1000 shell /usr/bin/fish
  - Delete: DEL users:1000
- Use “subkeys” to store collection values
  - SADD users:1000:groups 32

Sorted sets

- ZADD key score member [score member…] – add members to the sorted set (or update their scores) [O(log(N))*]
- ZCARD key – return the number of members in the sorted set [O(1)]
- ZSCORE key member – return the score of the member [O(1)]
- ZINCRBY key member increment – increment a member's score [O(log(N))]
- ZREM key member [member…] – remove the members from the sorted set [O(log(N))*]
- ZRANK/ZREVRANK key member – return the rank of the member within the sorted set, with scores in ascending/descending order [O(log(N))]

Sorted sets – query operations

- ZCOUNT key min max – count the number of elements within a certain range [O(log(N)+M)]
- ZRANGE/ZREVRANGE key start stop [WITHSCORES] – return elements of the sorted set by rank
- ZRANGEBYSCORE/ZREVRANGEBYSCORE key min/max max/min [WITHSCORES] [LIMIT offset count] – return elements of the sorted set by score [O(log(N)+M)]

Sorted sets – more operations

- ZREMRANGEBYRANK key start stop – delete all elements of the sorted set within the given indices [O(log(N)+M)]
- ZREMRANGEBYSCORE key min max – delete all elements from the sorted set within a certain range [O(log(N)+M)]
- ZUNIONSTORE [complicated] – take a union of multiple [sorted] sets and stash it in a key [O(N)+O(M log(M))]
- ZINTERSTORE [complicated] – take an intersection of multiple [sorted] sets and stash it in a key [O(N*K)+O(M log(M)) worst case]

Use Case: High score tables

- Add people to the table:
  - ZADD scores:2012-01-03 <game ID> 8810
  - ZADD scores:2012-01-04 <game ID> 10270
- Show the user their rank:
  - ZREVRANK scores:2012-01-04 <game ID>
- Get the top 10 for a day:
  - ZREVRANGE scores:2012-01-04 0 9 [WITHSCORES]
- Create weekly, monthly, or yearly tables:
  - ZUNIONSTORE scores:2012-W01 7 2012-01-01 […] 2012-01-07 AGGREGATE MAX

Use Case: Date-based indices

- Use UNIX timestamps as scores
  - ZADD articles:bydate 1329257190 44
- Get all the articles for February 2012:
  - ZREVRANGEBYSCORE articles:bydate 1330541999 1328072400
- Even paginate:
  - ZREVRANGEBYSCORE articles:bydate 1330541999 1328072400 LIMIT 0 10
  - ZREVRANGEBYSCORE articles:bydate 1330541999 1328072400 LIMIT 10 10

Composition

MULTI/EXEC

- Execute multiple commands simultaneously
  - Call MULTI to begin queuing the commands
  - Enter all the commands
  - Call EXEC to execute them all in a row
  - Call DISCARD to cancel
- EXEC will return the return values from all the commands
- No other commands will be run until EXEC completes
- Redis checks the syntax when you queue commands, but they can still fail at runtime – Redis will just return the error and keep plowing through

Optimistic Locking with WATCH

- Problem: What if we read a value, start queuing commands based on it, but then the value changes before we EXEC?
- Solution: WATCH!
  - Call WATCH with the keys we plan on using
  - Read values
  - Then do MULTI/EXEC
  - If the keys have changed since we called WATCH, EXEC errors out instead of running our commands
  - In that case, start over!

Lua Scripting

- Run Lua scripts server-side
  - They have access to all the Redis commands, and some Lua libraries
  - No other commands are served while a Lua script is running – use this to perform complex operations
  - Should be completely deterministic in order to replicate and AOF properly
- To be released in Redis 2.6

Lua Scripting – EVAL Command

- EVAL source numkeys [key …] [arg …]
  - Calling conventions are kinda funky, but it lets Redis detect problems when running in a cluster
  - Also EVALSHA – if you have run the script before, Redis will use the compiled bytecode
- Inside the script:
  - Access arguments using KEYS and ARGV
  - Call Redis commands with redis.call and/or redis.pcall
  - Lua standard libraries + Redis tools + CJSON + lstruct

ZPOP using Lua Scripting

- Remove and return the lowest-scored item from a sorted set, atomically

```
local key = KEYS[1]
local element = redis.call(“ZRANGE”, key, 0, 0)[1]
redis.call(“ZREM”, key, element)
return element
```

- Run using EVAL <source> 1 <key>

Other Features

Publish/Subscribe Messaging

- Developed out of BLPOP and BRPOP
- Subscribe to a channel:
  - SUBSCRIBE freenode:#ncsulug freenode:#bottest
- Publish a message on a channel:
  - PUBLISH freenode:#ncsulug "<lessthanthree> leafstorm: UNLOCKED: VOLCANIC LAVA FLOW"
- When you're done listening:
  - UNSUBSCRIBE freenode:#ncsulug freenode:#bottest

Master-Slave Replication

- Add “slaveof <host> <port>” to the slave’s config file
- Slave will sync dataset from master, then master will send commands to slave
- Slave will automatically resync if it loses the connection
- Alter settings at runtime with:
  - SLAVEOF <HOST> <PORT> to set a new master
  - SLAVEOF NO ONE to turn slave into master

Cluster

- Not actually here yet!
  - (antirez keeps changing the spec)
- But eventually it will let you distribute datasets across multiple nodes intelligently
- Until then, use simple sharding or lots of RAM

Questions?