redisconf19

Work stealing for fun & profit: Making Redis do things you thought it couldn't do

Jim Nelson
Internet Archive, *Cluster Operations Engineer*

presented by redislabs HOME OF REDIS

Internet Archive / archive.org

Universal Access to All Knowledge

- Nonprofit, founded 1996, based in San Francisco
- Archive of digital and physical media
- Includes Web, books, music, film, software & more
- One million visitors per day
- 22+ years of the Web archived for replay: 300 billion pages
- 30+ million media items

Over 45 petabytes of unique data stored on a bespoke archival-centric computing cluster located in San Francisco & Richmond

Internet Archive ♡ Redis

- IA metadata directory service backed by 10-node sharded Redis cluster
- Sharding performed client-side via consistent hashing (PHP, Predis)
- Each node supported by two replicated mirrors (fail-over)
- Specialized Redis instances also used throughout IA's services, including
  - Wayback Machine
  - Search (item metadata and full-text)
  - Administrative & usage tracking
  - …and more

Before we get started

- I'm assuming you've written Redis client code already
- You don't need to know the exact details of the Redis protocol…
- …but you need some understanding of how Redis "works"
  - single-threaded
  - different data types (strings, hash, sorted sets, etc.)
  - Basic commands (GET, SET, HGET, etc.)
  - transactions (MULTI/EXEC)

Some things I wish Redis could do it cannot do today

Redis wishlist

Expire keys *only* after their time-to-live passes

- Redis may evict volatile keys *at any time* when under memory pressure (MAXMEMORY policy)
- EXPIRE marks the key as “volatile”
- If Redis runs out of memory, random volatile keys are freed

Redis wishlist

Expire keys *only* after their time-to-live passes

There are some situations where it's not merely important,

but *vital* the key remains available until the time-to-live expires

(locks, expensive recomputes, leases, etc.)

Redis wishlist

- Our Redis cluster is **constantly** under memory pressure
- In heavy load, up to **1,000 keys / minute** are evicted
- 30+ million media items means 30+ million content identifiers
- All of their metadata are competing for precious bytes on our Redis cluster

Redis wishlist

Examples of keys with important TTLs:

- **Distributed locks (Redlock, etc.)**
- **Leases / licenses**
- **Expensive recomputes held in cache for extended periods of time**

For these applications we need **deterministic expiration**

For other keys, we're happy to allow Redis to evict them as necessary to make room for more frequently used values

Redis wishlist

Expire individual fields in a hash map

- In Redis, only keys can be set to expire
- The entire hash map is evicted when the key expires
- Would be nice if individual *fields* in the hash map had time-to-live

Redis wishlist

field	value	expiration
`content-id`	`hey-diddle-diddle`	`TTL: 12 hours`
`likes`	`42`	`TTL: 5 minutes`
`related-content`	`cat,fiddle,dish,spoon`	`TTL: 1 hour`

One approach:
cron jobs,
daemons,
schedulers, et al.

cron jobs, daemons, etc.

Use automated background tasks to monitor Redis

However:

- cron jobs may fail to run (or, worse, too many run at same time)
- daemons can be tricky to write
- single points of failure / lack of failover
- dedicated hardware to run background jobs

If you've ever had a machine go down over the weekend and your cleanup tasks stopped running, you know what I'm talking about

Another approach: work stealing

Work stealing

Term & concept borrowed from multiprocessor/multithreaded computing

Basic idea:

- “Steal” spare cycles from worker pools (“mugging” or “recruiting”)
- Have recruited workers perform a small slice of work
- If unable to finish work, don’t sweat it—the next recruit will pick up where the current one left off

That’s it—a very simple idea that turns out to be quite powerful.

What’s more, it works well with Redis.

Finding workers with spare cycles to steal

- Web server pools
- Task / job / priority queue workers
- Cache fetches (cache hits in particular)

Finding workers with spare cycles to steal

In general, look for

- A pool or group of workers constantly servicing requests…
- …where the worker code may have a “fast exit” (such as a cache hit or a no-op or a “little work to do” signal)

Finding workers with spare cycles to steal

In our distributed cluster, we steal workers from a content directory service

The directory service is constantly accessed by our Web servers

If a page paint worker has a cache hit, it's a candidate for recruiting

We only need to recruit ~ 0.3% of the candidate workers
to get all our background work completed
with workers to spare

Example #1: Volatile hash map fields

To go back to my wish for hash map with expiring fields, here's how I might implement it:

To store a value:

- In a Redis hash map, store fields + their values
- In a Redis sorted set, store the field names with an expiration time

Example #1: Volatile hash map fields

Set a value (“HSETEX”):

```
; start transaction
MULTI

; set hash field
HSET hash:user:1 <field-name> <value>

; add field name to sorted set (sorted by expiration)
ZADD zset:user:1 <time_t> <field-name>

; commit transaction
EXEC
```

Example #1: Volatile hash map fields

To read a value:

- Reading value from hash map and read the expiration time from the sorted set
- If the field has expired, return to the caller `NULL`, `Undefined`, etc.

Example #1: Volatile hash map fields

Get a value:

```
; start transaction
MULTI

; get hash field
<value> = HGET hash:user:1 <field-name>

; get field's expiration time
<expiration> = ZSCORE zset:user:1 <field-name>

; commit transaction
EXEC
```

Example #1: Volatile hash map fields

When are the expired fields deleted?

Think about those “30+ million content identifiers” mentioned earlier—

If a user accesses a piece of content once and never asks for it again…

… the client code never has a chance to delete expired hash fields.

It will remain stored in Redis permanently,
wasting memory that could be used for “active” records.

Example #1: Volatile hash map fields

field	value	expiration
content-id	hey-diddle-diddle	TTL: 12 hours
likes	42	TTL: 5 minutes
related-content	cat,fiddle,dish,spoon	TTL: 1 hour

Example #1: Volatile hash map fields

To expire those fields in the background:

- A worker is “recruited” from the Web pool when it gets a cache hit
- The recruit randomly selects a single hash map
- It reads the sorted set to determine which fields have expired
- It deletes from the hash map those fields

Example #1: Volatile hash map fields

Work stealing allows for garbage collection

to occur “in the background” without

creating specialized scripts or bots

that may fail or not run often enough

to keep up with demand

Example #1: Volatile hash map fields

hash map (hash:user:1)

content-id	hey-diddle-diddle
likes	42
related	spoon,dish,fork
last-referrer	/details/spoon

sorted set (zset:user:1)

related	9:05am
likes	10:00am
last-referrer	9:00pm

Example #1: Volatile hash map fields

Background expiration (work stealing):

```
WATCH hash:user:1 zset:user:1

; read five oldest hash fields which have expired
<expired-fields> = ZRANGEBYSCORE zset:user:1 -inf time() LIMIT 0 5

; start transaction
MULTI

; delete expired hash fields
HDEL hash:user:1 <expired-fields>

; remove fields from sorted set of expiration times
ZREM zset:user:1 <expired-fields>

; commit transaction or abort if hash:user:1 or zset:user:1 modified
EXEC
```

Things to notice

WATCH is used to prevent modifying the hash map or sorted set if another writer modifies the tables

```
WATCH hash:user:1 zset:user:1
```

WATCH will cause the EXEC command to abort if either of the above keys have been modified

Work stealing: things to notice

Any failure or outlying condition is ignored:

- hash map is empty
- no fields have expired
- failure to connect to server
- network error
- WATCH triggers an abort
- fail to acquire a distributed lock

The whole idea with work stealing is to “steal” slivers of spare cycles from workers

If there’s an error, don’t retry—give up and let the next recruit make an attempt

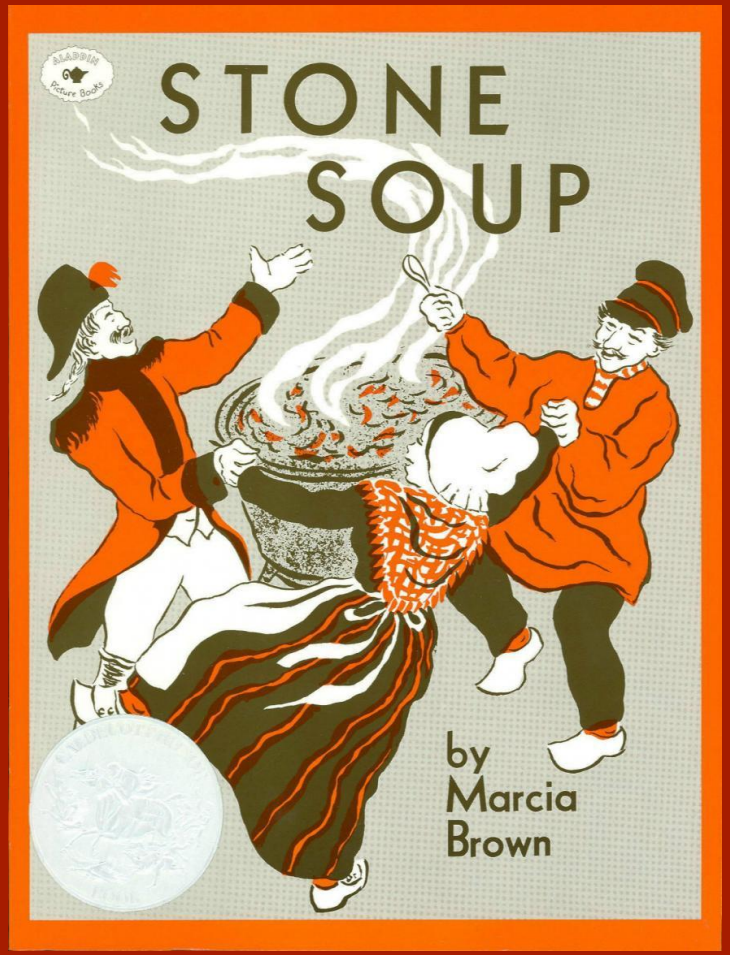
Work stealing: things to notice

Only a small amount of work is done in any stolen cycle:

```
ZRANGEBYSCORE zset:user:1 -inf time() LIMIT 0 5
```

Although this could be coded to fetch *all* expired fields, instead only the five (5) oldest fields are retrieved

A large number of workers,
each performing a small amount of work,
means a lot can be done in aggregate


ALADDIN
Picture Books
STONE
SOUP
by
Marcia
Brown

Example #2: Task queue follower

At Internet Archive, we have a custom task queuing system that performs all content transformations and mutations

Some subsystems need to know **quickly**
if a particular content object has work queued or running

(We have 30+ million content identifiers,
but only a small percentage of them
are being worked on at any one time)

Example #2: Task queue follower

We use Redis to track when a content identifier has queued tasks with **two work stealing jobs**

The first **adds content identifiers** by walking the database table and adding identifiers to a Redis set

(Again, each Redis worker is only responsible for adding a small number of tasks to Redis: 50 rows at most)

Example #2: Task queue follower

The other work stealing job is to **remove content identifiers** when they no longer have work queued

Randomly samples a small number of content identifiers in the Redis set and queries the database to see if the identifier still has any tasks queued

Any content identifiers with no queued work is removed from the Redis set

Example #2: Task queue follower

```
WATCH set:tasks:queued

; sample 5 content identifiers
<ids> = SRANDMEMBER set:tasks:queued 5

<quiesced> = <database query asking which of hose 5 have no pending tasks>

; start transaction
MULTI

; remove all with no pending work
SREM set:tasks:queued <quiesced>

; commit transaction or abort if set:tasks:queued was modified
EXEC
```

Writing work stealing routines for Redis

- Use WATCH and transactions to abort when other writers modify tables
- Random sampling is your friend
- Log failures and outlying conditions…
- …but don’t retry, just exit and let the next worker make an attempt
- Don’t retry locks either, let the next worker make the attempt
- Each recruited worker performs only a small slice of work

“Wait—are you saying *all* my workers should be doing this extra work?”

Recruiting workers (“mugging”) for work stealing

The approach I like (and has worked well for us):

#1: Workers are “enlisted” if they fast-exit in a particular bit of work:

- get a cache hit
- paint a simple Web page
- don’t need to query database
- etc.

Recruiting workers (“mugging”) for work stealing

The approach I like (and has worked well for us):

#2: Each work stealing job declares a rate—a percentage of enlisted workers it needs for its work

- For example, 0.001 to receive 0.1% of the enlisted workers
- If you have more traffic, use a lower job rate

Recruiting workers (“mugging”) for work stealing

The approach I like (and has worked well for us):

#3: Each recruited worker generates a random number

- If that number if below the job’s rate, the enlistee is “recruited” and performs a little bit of work
- Otherwise, the enlistee is dismissed—no extra work needs to be done at this time

Recruiting workers (pseudo-code)

```
method enlist() {
    rand = Random::float(); // from 0.0 to 1.0
    target = 0.0;
    foreach (self.registered_jobs as job) {
        target += job.rate;
        if (rand <= target)
            return job.perform();
    }
}
```

Recruiting workers

It’s deceptively simple

It works surprisingly well

It requires no extra state (tracking the number of workers, distributed locks, job queues, etc.)

We’re doing this in production right now

Recruiting workers

One caveat:

Instrument your code!
(StatsD, Prometheus, etc.)

It's important to know

- How many workers are selected for background work
- How many finish their work, how many stop due to errors/aborts
- Fine-tune your job rates (percentages) so enough workers are being recruited

“Show me the code.”

https://github.com/internetarchive/work-stealing

redisconf19

Questions?

```
TS.RANGE key FROM_TIMESTAMP
TO TIMESTAMP [aggregationType]
[bucketSizeSeconds]
```

presented by

redislabs
HOME OF REDIS

redisconf19

Thank you!

```
TS.RANGE key FROM_TIMESTAMP
TO TIMESTAMP [aggregationType]
[bucketSizeSeconds]
```

presented by

redislabs
HOME OF REDIS

redis