Harnessing the Full power of Redis

appear [here]

Daniel Magliola
daniel@danielmagliola.com
http://danielmagliola.com

What is Redis?

Redis is essentially like Memcached, but better

I mean, it's an in-memory key-value store, right?

Redis is essentially like Memcached, but better

I mean, it's an in-memory key-value store, right?

Only that, actually,
no, it's totally not
just that.

The “better than memcached” situation

- So, Redis is like Memcached but much better.
- Oh, better how?
- I don’t know really, persistence, something something, data structures…
- Cool, let’s cache some stuff in there.

Filter by group: All or search for: e.g. SUNION

ITCOUNT BITOP BITPOS BLPOP BRPOP BRPOPLPUSH DBSIZE DECR DECRBY DEL DUMP EVAL EXEC

XPIRE FLUSHDB GET GETBIT GETRANGE GETSET HDEL HEXISTS HGET HGETALL HINCRBY HINCRBY

KEYS HLEN HMGET HMSET HSET HSETNX HSTRLEN HVALS INCR INCRBY INCRBYFLOAT KEYS LIND

INSERT LLEN LPOP LPUSH LPUSHX LRANGE LREM LSET LTRIM MGET MSET MSETNX PEXPIRE PE

FADD PFCOUNT PFMERGE PSETEX PSUBSCRIBE PUBSUB PTTL PUBLISH PUNSUBSCRIBE RPOP RPOPLI

PUSH RPUSHX SADD SCARD SDIFF SDIFFSTORE SET SETBIT SETEX SETNX SETRANGE SINTER SI

ISMEMBER SMEMBERS SMOVE SPOP SRANDMEMBER SREM STRLEN SUNION SUNIONSTORE ZADD ZCARD

INCRBY ZINTERSTORE ZLEXCOUNT ZRANGE ZRANGEBYLEX ZREVRANGEBYLEX ZRANGEBYSCORE ZRANK Z

REMRANGEBYLEX ZREMRANGEBYRANK ZREMRANGEBYSCORE ZREVRANGE ZREVRANGEBYSCORE ZREVRANK ZSC

UNIONSTORE SCAN SSCAN HSCAN ZSCAN

redis Commands Clients Documentation Community Download Support Licens

Filter by group: All or search for: e.g. SUNION

ITCOUNT BITOP BITPOS BLPOP BRPOP BRPOPLPUSH DBSIZE DECR DECRBY DEL DUMP EVAL EXEC

XPIRE FLUSHDB GET GETBIT GETRANGE GETSET HDEL HEXISTS HGET HGETALL HINCRBY HINCRBY

KEYS HLEN HMGET HMSET HSET HSETNX HSTRLEN HVALS INCR INCRBY INCRBYFLOAT KEYS LIND

INSERT LLEN LPOP LPUSH LPUSHX LRANGE LREM LSET LTRIM MGET MSET MSETNX PEXPIRE PE

FADD PFCOUNT PFMERGE PSETEX PSUBSCRIBE PUBSUB PTTL PUBLISH PUNSUBSCRIBE RPOP RPOPLP

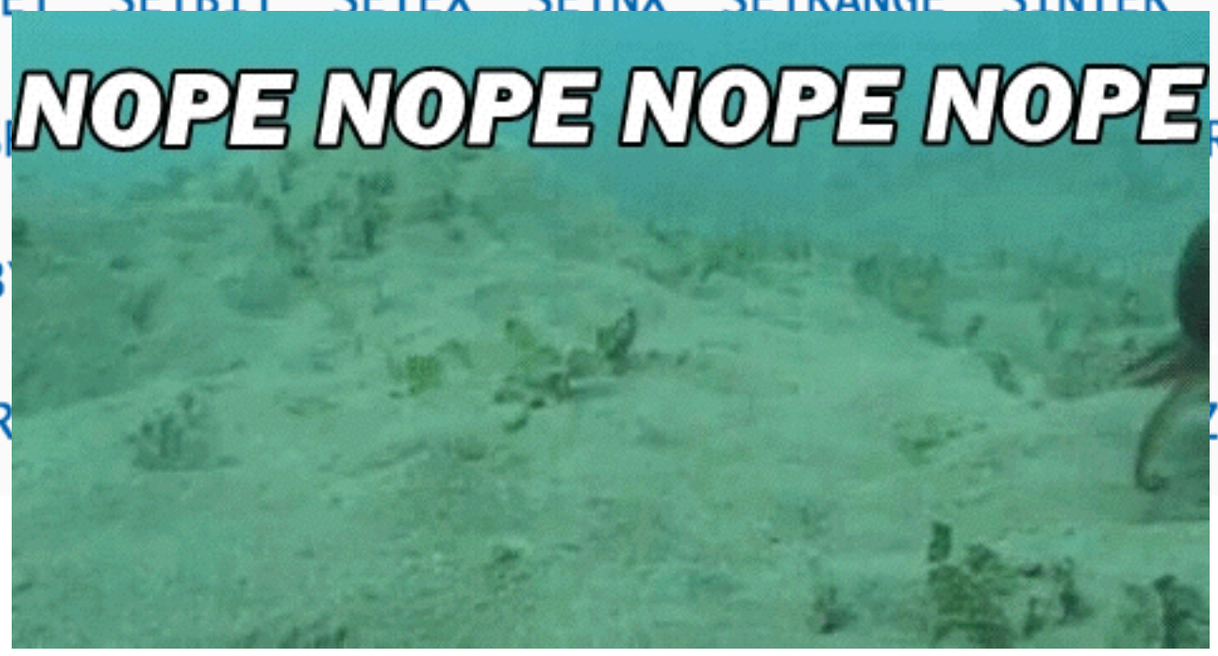
PUSH RPUSHX SADD SCARD SDIFF SDIFFSTORE SET SETBIT SETEX SETNX SETRANGE SINTER SI

ISMEMBER SMEMBERS SMOVE SPOP SRANDMEMBER SR RD

INCRBY ZINTERSTORE ZLEXCOUNT ZRANGE ZRANGEB Z

REMRANGEBYLEX ZREMRANGEBYRANK ZREMRANGEBYSCOR ZSC

UNIONSTORE SCAN SSCAN HSCAN ZSCAN



Pretending it's a cache

- GET
- SET
- Maybe an expiry

How to approach this

- Features
- Data Structures
- Use Cases

Why is Redis Better?
(the short version)

1. Persistence

Redis is a DATABASE. It's not a cache.
It actually stores your data and doesn’t forget it.

Why is Redis Better?
(the short version)

2. PUBSUB

As made famous by ActionCable. You won't use it that much, but if you need it, it's amazingly handy

Why is Redis Better?
(the short version)

3. Blocking Reads

You can read a value and if it's not there, wait for it to show up.

Why is Redis Better?
(the short version)

4. Atomic changes, transactions, pipelines

You get atomic changes, transactions that are not transactions, and pipelines that'll make everything faster.

Why is Redis Better?
(the short version)

5. Cheapest message queue EVER

Persistence + Blocking reads + Atomic fancy stuff makes this the cheapest, easiest message queue ever.

Why is Redis Better?
(the short version)

6. Data structures

This is the actual interesting bit.

Data structures make Redis behave unlike any other database i've seen, and lets you do things I wish others did, in a ridiculously more efficient way.

Strings

- This is what you probably know. You set a string, you read it back. You probably want JSON in there
- Commands: GET / SET / MGET / MSET
- INCR / DECR if you’re storing numbers
- SETNX / SETEX / friends for fancy locking stuff

Hashes

- “Keys with subfields”
- Can read / set one field at a time, or all at once
  - HGET / HGETALL / HSET / HMSET
- HINCRBY, ideal for keeping a bunch of daily / hourly counters all in one key

Lists

- Doubly linked lists of strings, allowing you to push and pop at both ends.
- This gives you **queues** (RPUSH / LPOP)
- and **stacks** (RPUSH / RPOP)
- Blocking reads (BLPOP / BRPOP) for producer / consumer patterns (aka Sidekiq / Resque).
- RPOPLPUSH for fun scenarios.

Sets

- Set of unique strings, with no ordering.
- Adding a value twice is a NOP, not an error.
- Useful for list of other keys.
- SADD / SREM / SPOP
- SMEMBERS / SISMEMBER
- Set arithmetic (SUNION, SDIFF, SINTER)

Sorted Sets (ZSET)

- Set of unique strings, with a float value (score) associated with them.
- Designed to quickly index by the score.
- Values are unique, adding the same value twice just updates score.

Sorted Sets (ZSET)

- Uses:
  - Get all the values within a range of scores (**ZRANGEBYSCORE**)
  - Get the first / last N values / slice the list by position (**ZRANGE**)
  - Get the position of a value (**ZRANK**)
  - Remove by value, score or position
  - Prefix any of these with **REV** to invert order

Pipelining

- Because Redis is ludicrously fast, almost always, 100% of the execution time is the roundtrip.
- So, you can send several commands at once:
    - With PgSQL, if you need to UPDATE 3 records, or INSERT to 3 different tables, that's 3 roundtrips.
- Allows you to do multiple reads or multiple writes with a single roundtrip
    - Example, you're tracking music downloads per country, per genre and per track. 3 **INCRBY** / **HINCRBY** in a single roundtrip
- No updating with data you got from a READ, though, that needs scripting or 2 roundtrips.

Atomicity

- Redis is single-threaded.
- This makes concurrency very, very simple, and because it's so fast, it's not really a blocker.
- If you issue a **transaction**, all commands happen at once. (Wrap in a pipeline for extra speed).
- No rollback, though, that's not what transactions are for.

Use Cases

IP to Location

- Very common problem: turning an IP into a country. I've found 3 approaches:

IP to Location

- Very common problem: turning an IP into a country. I've found 3 approaches:
- Call an external service (**100ms**)

IP to Location

- Very common problem: turning an IP into a country. I've found 3 approaches:
- Call an external service (**100ms**)
- Store a DB table of 300k records of "from / to" IP blocks, and query that (**20ms**)
  - SELECT * FROM ip_locations WHERE 1123089460 BETWEEN ip_from AND ip_to

IP to Location

- Very common problem: turning an IP into a country. I've found 3 approaches:
- Call an external service (**100ms**)
- Store a DB table of 300k records of "from / to" IP blocks, and query that (**20ms**)
    - SELECT * FROM ip_locations WHERE 1123089460 BETWEEN ip_from AND ip_to
- Store that table in a ZSET, scored by ip_to: (**well under 1ms**)
    - **ZRANGEBYSCORE** ip_locations 1123089460 +inf limit 0, 1

Sidekiq

- Sidekiq uses pretty much all of Redis, quite cleverly
- Set **queues** holds the list of existing queues.
- Each queue is a list of jobs (named after the queue).
- Enqueue jobs with **LPUSH**, consume with **BRPOP**.
  - **BRPOP** can listen to multiple keys at once, so if *any* of them get a job, it returns.
  - But it returns from the first one requested, allowing to prioritize.

Sidekiq

- Scheduled jobs are queued in a **ZSET**, their score is the timestamp at which they need to run.
- Failed jobs go to a “retry” **ZSET**, their score is the rety timestamp.
- Every N seconds, Redis polls these for scores on the past and enqueues those jobs:
  - **ZRANGEBYSCORE retry, '-inf', now, :limit => [0, 1]**
    ZREM retry, ‘job’
    LPUSH default ‘job’
- Dead jobs go to a “dead” **ZSET**, scores by time of death, gets periodically garbage collected.

Buffered Updates
INSERT

- PgSQL and other DBs allow multi-INSERT. However, if logging events, you get records one at a time. Inserting one at a time is massively inefficient. If you won't be reading them right away…
- Store these events in a Redis LIST.
- In a cron:
    - **LRANGE** to pop the first 1,000 at once
      LTRIM the list to remove them from Redis (in a Txn)
      multi-INSERT them all at once

Buffered Updates

UPDATE a string field

- For frequent updates (last_action_at), PgSQL forces you to update one at a time. These are infinitely worse than INSERTs, because of row locking.
- Store in Redis in a hash, key = record ID
- Store all the IDs in a ZSET (you cannot get “1000 values” from a hash, you need to know the field names), score is time of last update.

Buffered Updates
UPDATE a string field

- In a cron:
  - **ZRANGEBYSCORE** the zset to get the oldest 1000 IDs.
    ZREM to remove them
    HMGET to get all the values
    HREM to remove them
    - You need to do all this in a LUA script to make it atomic.
- Do some magic with temporary tables and “UPDATE SELECT” to do all updates in one go.

Buffered Updates

UPDATE an increasing number

- If the values you're updating are constantly increasing numbers (floats, timestamps, etc), you can use a single ZSET
  - ZADD
  - ZRANGEBYSCORE
  - PgSQL UPDATE
  - ZREMRANGEBYSCORE
  - Look ma, no LUA!
- I'm planning to extract all this magic into a gem, talk to me if you want it soon.

Round Robin

- Example: You need to fairly distribute requests between a number of servers. Or text messages between a number of phone lines.
- Store all servers in a list.
  - **RPUSH** servers server1 server2 server3
- Pop the list when you need one, re-pushing to the same list, making a cycle.
  - **RPOPLPUSH servers servers**

Phone Line Rotator

- In USA, the FCC mandates that:
  - You need to own your phone lines.
  - Maximum of roughly 500 messages per day and per phone line. If this is exceeded **your number will be blocked and will no longer work**.
  - Not more than one message per second per phone number. If you send messages more quickly, they **will be rejected.**
- No, Twilio doesn’t magically solve this for you. Yeah, I know, it sounds like it should. It don’t.

Phone Line Rotator

- So, **RPOPLPUSH** Round Robin of phone lines, right?
    - No, these phone lines run out, and are rate limited.
- ZSETs to the rescue! (plus SETs and HASHes)

Phone Line Rotator

- Store all phone lines in a ZSET, scored by the last time you sent a message with them.
- Keep a HASH per phone line, HINCRBY for each message, counting how many SMS were sent that day.
- Keep a SET of “depleted” phone lines (hopefully empty)

Phone Line Rotator

Grab a phone line

- Grab the least recently used:
  - **ZRANGEBYSCORE** phone_line_rota -inf **1.second.ago** LIMIT 0 1
    - “from -inf to 1.second.ago” makes sure that we don’t send more than 1 SMS per second for each line. If all lines have messaged in the last second, we don’t get a line back, we need to fail and retry later.
- Re-add to the pool:
  - **ZADD** phone_line_rota **Time.now** phone_number
- Do this in a LUA script, to make it atomic

Phone Line Rotator

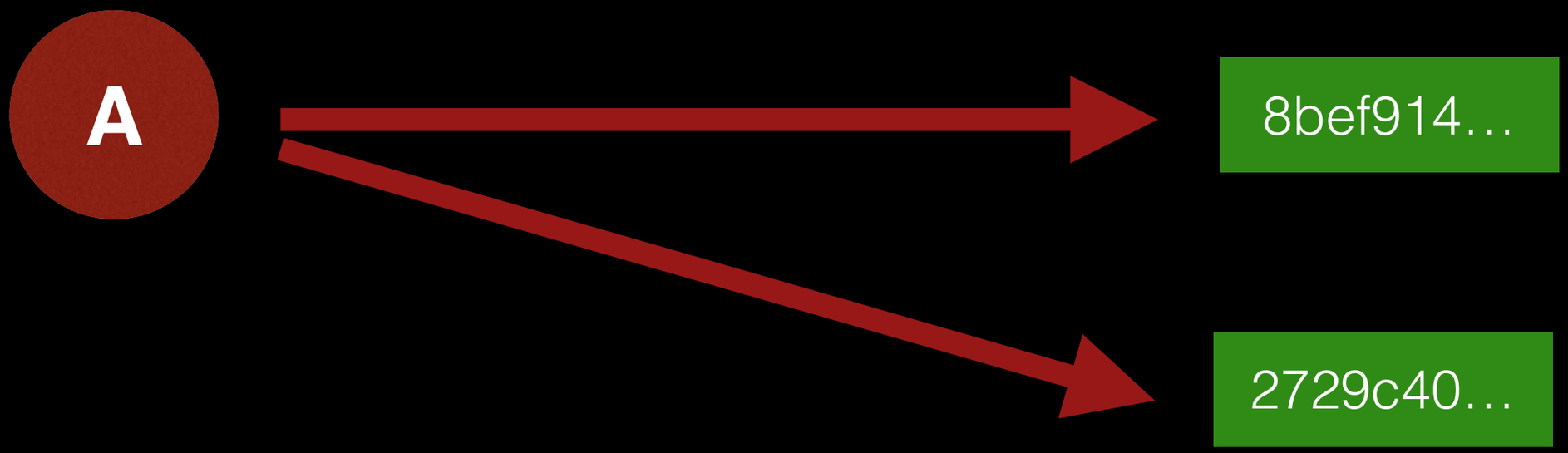
Once a message is sent

- Increase the counters
  - **HINCRBY** phoneline_counter:phone_number Date.today 1
  - **HINCRBY** will return the new value
- If return value >= 500
  - **SADD** depleted_lines phone_number
  - **ZREM** phone_line_rota phone_number

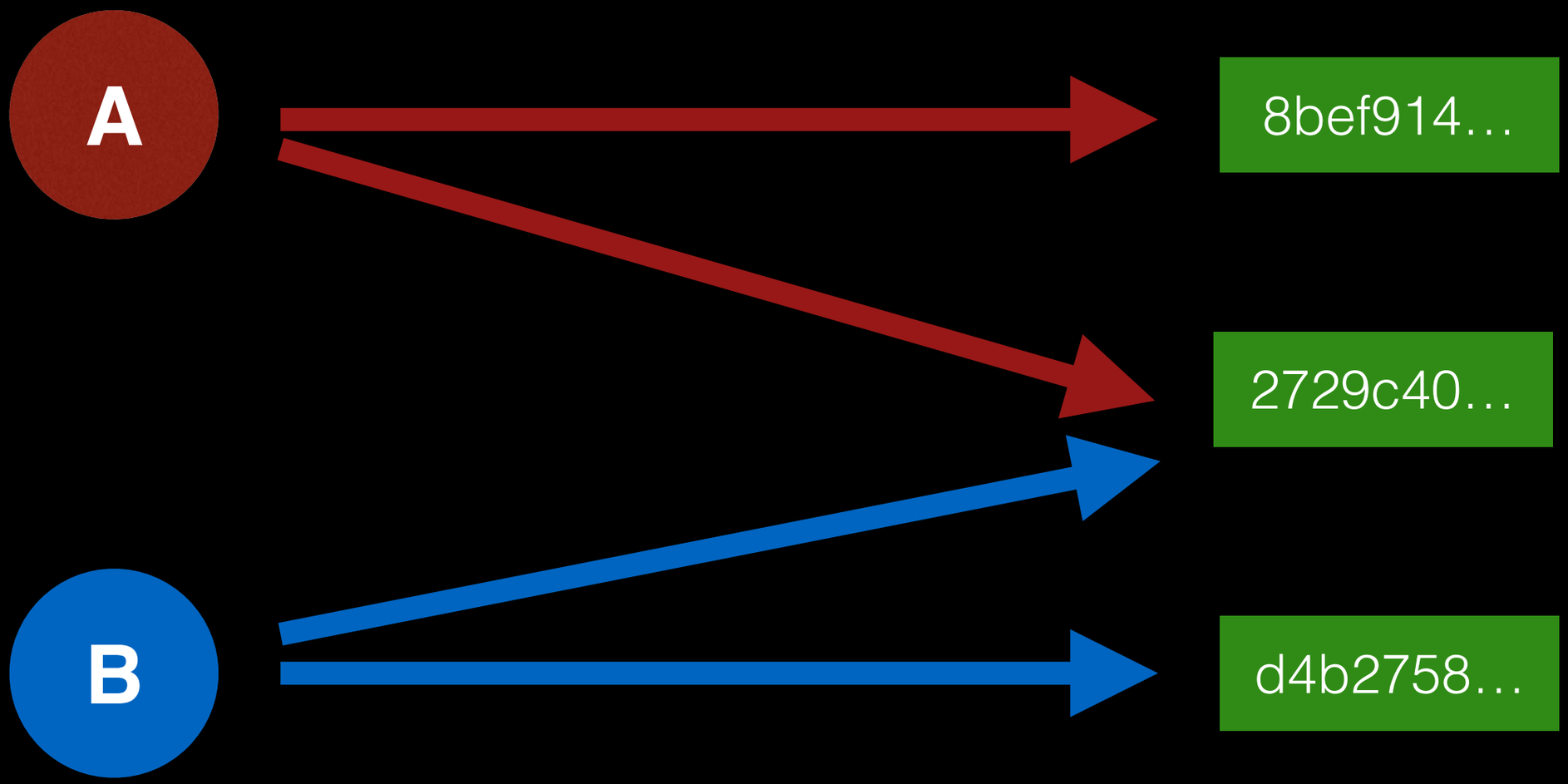
Reverse Phone Book Search

- We have a “Whatsapp style friend discovery”, via phonebook.
- But we want to notify existing users when one of their friends joins the app, so new users start getting messages.
- We need to upload and store everyone’s phone books (hashed, don’t worry!), which is about 1000 records per user.

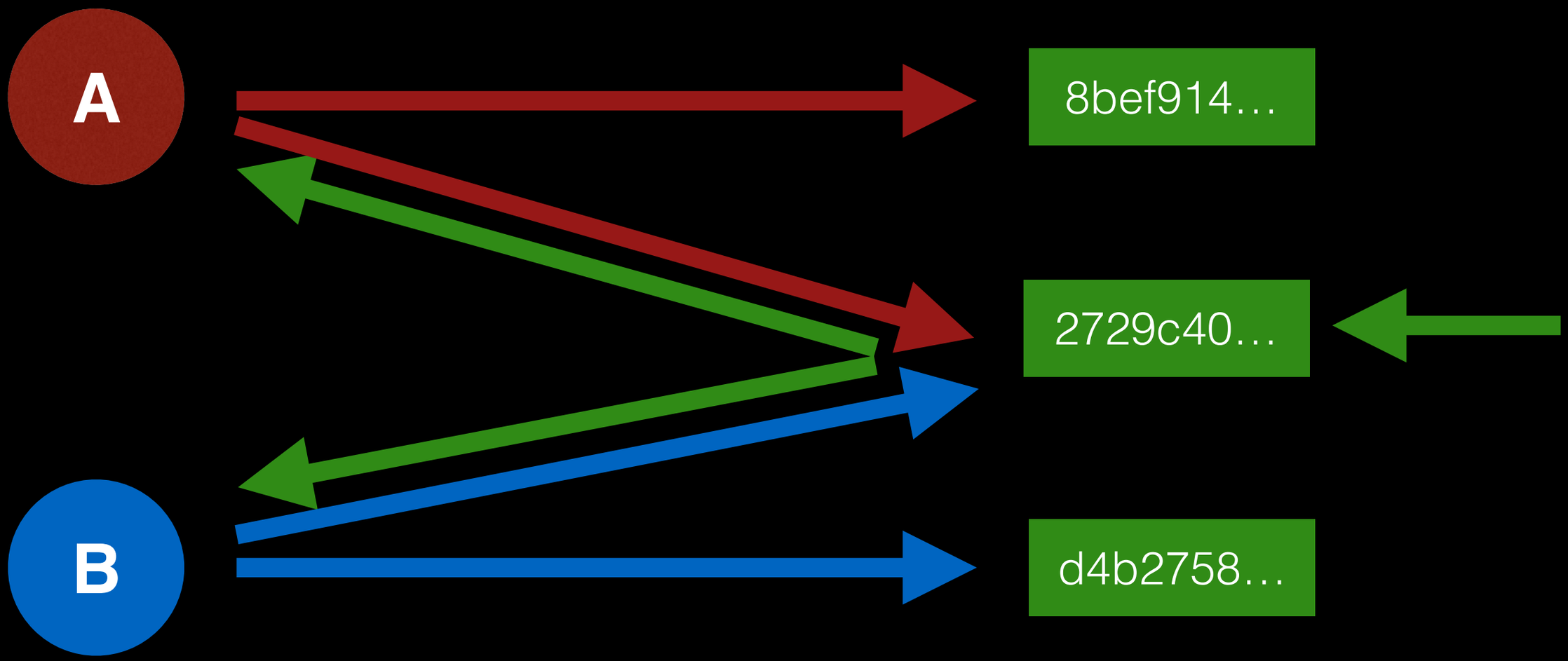
Reverse Phone Book Search



Reverse Phone Book Search



Reverse Phone Book Search



Reverse Phone Book Search

- 2 problems: reading and writing
- We need to store 1,000 records per signup. Fast.
- We need to be able to find who to notify back. This doesn’t need to be super fast, but it needs to not destroy the server.

Reverse Phone Book Search

PgSQL

	id [PK] serial	user_id integer	friend_hashed_phone_number character varying(255)
1	1	2	f06bd5764c3251b3f245daf3f7ebe0dcd3dbcf278a6733b01e2a55cd547a5e68
2	2	2	c5a1e5500cb484b56346638f4c3afada109faa1e850a1ef6b886f31d812880ae
3	3	2	7a1c78743fe149bd8c17dfd2771828934fafcfb3198c1917a71cd240dda96101
4	4	2	a2315ad24c673f55f3c2b9a5e3aa9c30fc7d7b6f0a89bb6066c4d76df0e71002
5	5	2	9442c301d61787ee7e8c8baba0c5afd282d7319c2c273c7f930b3d22fbb5f5cb
6	6	2	4acbeb0dc97b0be47c73285cee90e2af3751aaa162c6c52606d83f4527690f11
7	7	2	d5065e1c340c8a004850bb08dabec079767dd2a1e2a7b3170c2c8bdbfaf4ed92
8	8	2	1121b93817d4dd1cfc9cd14ca8512ea8c29aed40282d17d305f9fdfd7166fce2

Index

Reverse Phone Book Search

PgSQL

	id [PK] serial	user_id integer	friend_hashed_phc character varying(
1	1	2	f06bd5764c3251b
2	2	2	c5a1e5500cb484b
3	3	2	7a1c78743fe149b
4	4	2	a2315ad24c673f5
5	5	2	9442c301d61787e
6	6	2	4acbeb0dc97b0be
7	7	2	d5065e1c340c8a004850bb08dabec079767dd2a1e2a7b3170c2c8bdbfaf4ed92
8	8	2	1121b93817d4dd1cfc9cd14ca8512ea8c29aed40282d17d305f9fdfd7166fce2

- 1000 writes with 1 roundtrip: ✓
- Concurrency: ✓
- Updating that mega-index: ☹
- Read: Very intensive for poor server.

Index

Reverse Phone Book Search

MongoDB and friends

```
{
    id: '8bef914…',
    friend_of: ['A']
},
{
    id: '2729c40…',
    friend_of: ['A', 'B']
},
{
    id: 'd4b2758…',
    friend_of: ['B']
},
```

Reverse Phone Book Search

MongoDB and friends

```
{
    id: '8bef914…',
    friend_of: ['A']
},
{
    id: '2729c40…',
    friend_of: ['A', 'B']
},
{
    id: 'd4b2758…',
    friend_of: ['B']
},
```

- 1000 writes with 1 roundtrip: ☹️
- Concurrency: ☹️
- Read: 👍

Reverse Phone Book Search

Redis

```
PIPELINE do
    SADD 8bef914…, A
    SADD 2729c40…, A
end

PIPELINE do
    SADD 2729c40…, B
    SADD d4b2758…, B
end
```

Reverse Phone Book Search

Redis

```
PIPELINE do
    SADD 8bef914…, A
    SADD 2729c40…, A
end

PIPELINE do
    SADD 2729c40…, B
    SADD d4b2758…, B
end
```

- 1000 writes with 1 roundtrip: 👍
- Instantaneous read: 👍
- Concurrency: 👍

Reverse Phone Book Search

Redis

```
PIPELINE do
    SADD 8bef914…, A
    SADD 2729c40…, A
end

PIPELINE do
    SADD 2729c40…, B
    SADD d4b2758…, B
end
```

- 1000 writes with 1 roundtrip: 👍
- Instantaneous read: 👍
- Concurrency: 👍
- 144 Gb ought to be enough for everybody: ☹️ ☹️ ☹️

Can’t I just use PgSQL?

- Yes! PgSQL is **AWESOME!**
- PgSQL 9.5’s **SKIP LOCKED** actually helps a lot.
- But Redis is just more efficient by using very specific data structures to solve very specific problems, as opposed to the generic being pretty awesome at generic stuff.

Can’t I just use PgSQL?

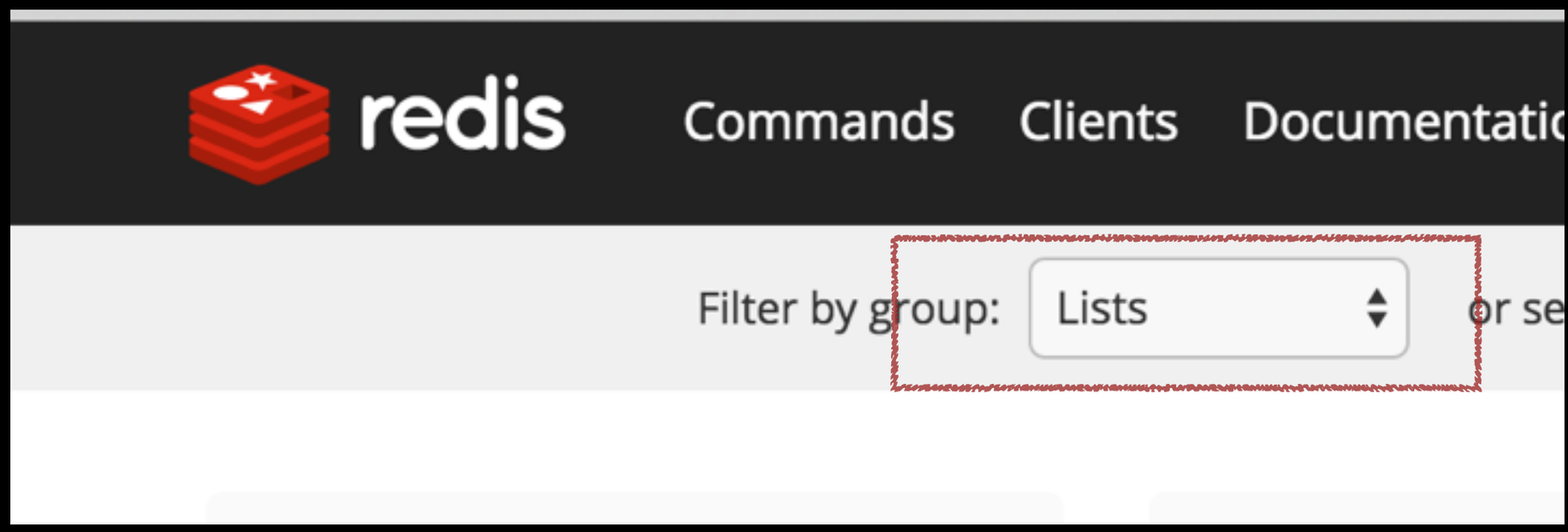
- ZSETs are way faster than indexes
- “Popping” things from DB queues is awkward and leads to lock contention (best case scenario if you do it right)
- Having an intermediate place where you drip-add things sporadically, and then you bulk-feed them to your data store is much more efficient.
- Concurrency is much harder: You need to think more and take more things into account:
  - Atomicity is much harder, locks slow things down, deadlocks abound!

Can’t I just use PgSQL?

- At low-scale, it doesn’t matter.
- At Facebook scale, you’re screwed anyway.
- In the middle, Redis can help **a lot**.
- All else being equal, anything that can be served by Redis is something that Postgres doesn't have to do, freeing your main bottleneck for the things where it really shines (OLTP)

Redis commands page

- Know really well what data types exist.
- Use the Redis commands page to check all the things you can do with each type, many times there is one command that solves your **entire** problem.



Questions?

- NOTES:
  - Slides at: **http://bit.ly/daniel-redis**
  - All the use cases I've talked about I've put into gems or I'm in the process of doing so. If you have any of these problems, talk to me!
  - If you've downloaded this keynote, keep going, there's lots more stuff that didn't fit in 20 minutes….
  - We're hiring! Come work on fun problems with some very amazing people!

appear [here]

Daniel Magliola
daniel@danielmagliola.com
http://danielmagliola.com

PS: Other Stuff

Persistence and eviction

- Being a database (and not a cache), when Redis runs out of RAM, it doesn't let you write to it anymore. It doesn't evict values.

- You can configure this behaviour, however: http://redis.io/topics/lru-cache

- The best value is **volatile-lru**. It'll evict keys if it runs out of RAM, but **only** if they have an expiration value.

- Store your real, important data without an expiration, and your cached data with an expiration (far into the future if you want it to stay), and your important data will be safe.

PS: Other Stuff
Extending beyond RAM

- In a scenario like the Reverse Address Book Search, you may want to expand beyond RAM. You may not need the speed that much, and disk is much cheaper, obviously.

- RedisLabs offers Enterprise Cluster, which allows extending into Flash.

- Hashtable of keys / key metadata is still in RAM, actual data is in Flash.

- It's a commercial solution, though.

  - https://redislabs.com/rlec-flash

PS: Other Stuff

Performance tips

Hashes

- Very memory efficient: You can have a lot more values per Gb storing in hashes instead of individual strings.

- But do not add too many (> 1,000) fields into a hash. Shard them manually, spreading them into multiple hashes.

    - Instagram has a good article on this: http://instagram-engineering.tumblr.com/post/12202313862/storing-hundreds-of-millions-of-simple-key-value

- Ruby Redis client maps automatically between Ruby hashes and Redis hashes, but sometimes, JSON encoding/decoding and storing as strings is faster. Highly context dependent, measure if you care about sub-milliseconds

PS: Other Stuff

Performance tips

- Do not list keys! Using **KEYS** in production will lead to pain, unless your keyspace is highly under control.

- Keep a SET of the relevant keys at the time of creating them. For example, if you have one hash per country, and you need to keep track of all the hashes you have, when you do **HSET**, also do an **SADD** into a set that holds a list of all the hashes. No extra round-trip, no extra cost.

PS: Other Stuff

- Replication: Single-master multi-slave configuration, with auto-promotion of slave if master dies.
    - Look at Sentinel for promotions.
- Clustering (from v3.0): Multi-master sharding of data. If you're going to do it, make sure you understand which CAP trade-offs apply, Redis is a bit odd.

PS: Other Stuff

- Locks:
  - **SETNX** (or **SET** with the **NX** flag) sets a key if it doesn't exist, and it returns whether it was set.
  - This plus expiry gives you centralized locking. Read more on the **SET** and **SETNX** command pages, and on ***redlock***.
- Distributed locks:
  - If you want more fault tolerance, the *distlock* algorithm helps
  - http://redis.io/topics/distlock
  - But don't use it if you must **absolutely guarantee** exclusivity, there are cases where it can fail, it's not partition-tolerant.

PS: Other Stuff

- Geo math (in Beta):
  - **GEOADD**
  - **GEODIST**
  - **GEORADIUS**

PS: Other Stuff

- Hyperloglogs: Approximately count uniques in humongous groups of things. You probably won't use them, but if you have this specific problem, they are very interesting.

PS: Other Stuff

- BIT Strings: Strings can be treated as bit strings, setting and reading individual bits, and doing logical operations on them. It may come in handy if you want to store millions of flags in a really compact way…