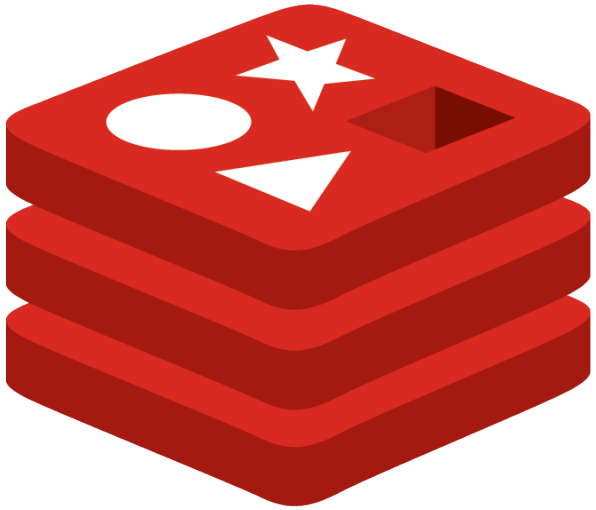
Kicking Ass With



redis

Redis for real world problems

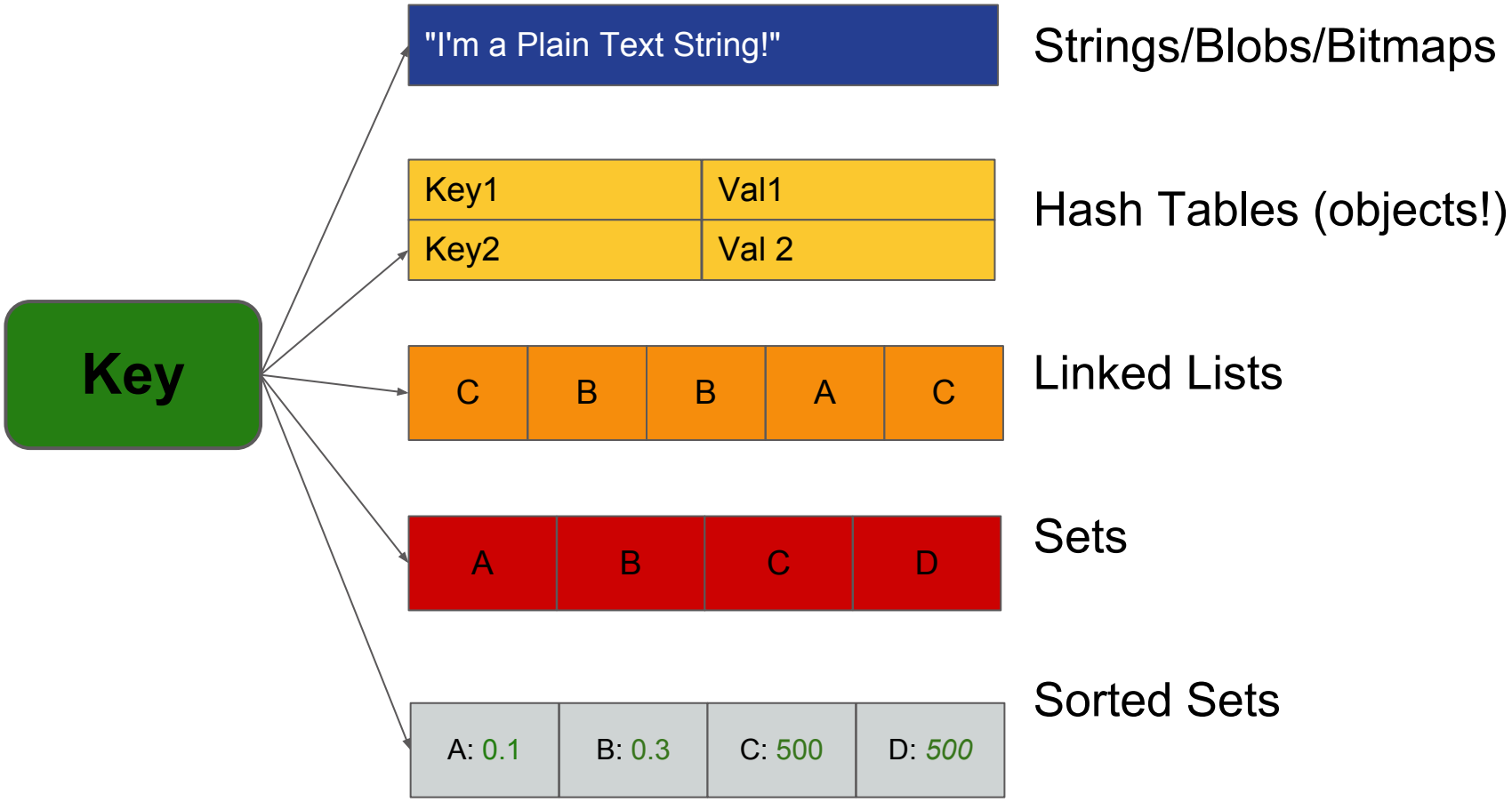
Dvir Volk, Chief Architect, Everything.me (@dvirsky)

O HAI! I CAN HAS REDIS?

Extremely Quick introduction to Redis

- Key => Data Structure server
- In memory, with persistence
- Extremely fast and versatile
- Rapidly growing (Instagr.am, Craigslist, Youporn ....)
- Open Source, awesome community
- Used as the primary data source in **Everything.me**:
  - Relational Data
  - Queueing
  - Caching
  - Machine Learning
  - Text Processing and search
  - Geo Stuff

Key => { Data Structures }



Redis is like Lego for Data

- Yes, It can be used as a simple KV store.
- But to really ***Use it***, you need to think of it as a tool set.
- You have a nail - redis is a hammer building toolkit.
- That can make almost any kind of hammer.
- Learning how to efficiently model your problem is the Zen of Redis.
- Here are a few examples...

Pattern 1: Simple, Fast, Object Store

Our problem:

- Very fast object store that scales up well.
- High write throughput.
- Atomic manipulation of object members.

Possible use cases:

- Online user data (session, game state)
- Social Feed
- Shopping Cart
- Anything, really...

Storing users as HASHes

users:1	**email**	john@domain.com
	name	John
	Password	aebc65feae8b
	id	1
users:2	**email**	Jane@domain.com
	name	Jane
	Password	aebc65ab117b
	id	2

Redis Pattern 1

- Each object is saved as a **HASH**.
- Hash objects are **{ key=> string/number }**
- No JSON & friends serialization overhead.
- Complex members and relations are stored as separate **HASH**es.
- Atomic **set** / **increment** / **getset** members.
- Use **INCR** for centralized incremental ids.
- Load objects with **HGETALL** / **HMGET**

Objects as Hashes

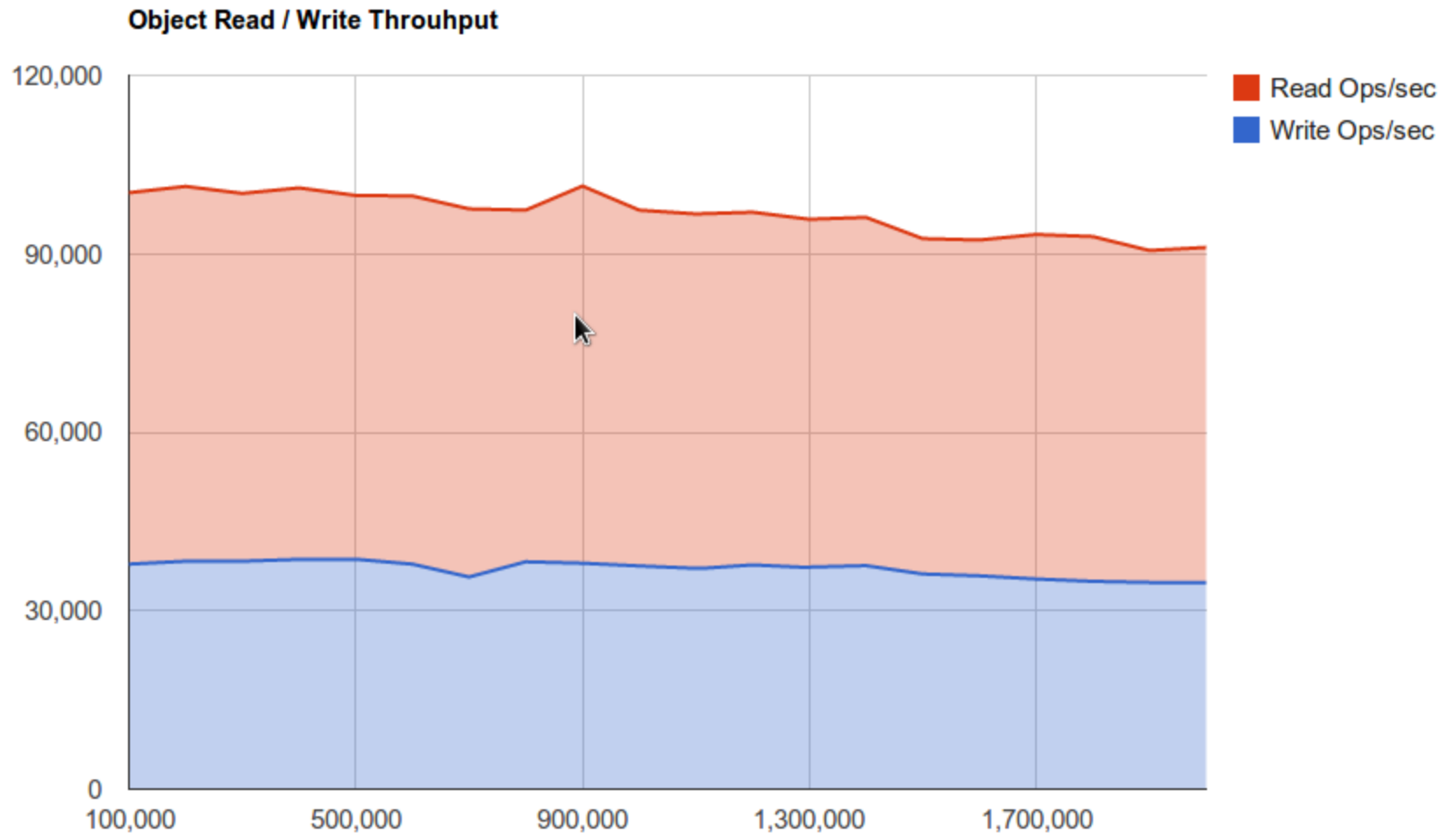
```
class User(RedisObject):
    def __init__(email, name, password):
        self.email = email
        self.name = name
        self.password = password
        self.id = self.createId()


user = User('user@domain.com', 'John',
'1234)

user.save()
```

```
> INCR users:id
(integer) 1
> HMSET "users:1"
   "email" "user@domain.com"
   "name" "John"
   "password" "1234"
OK
> HGETALL "users:1"
{ "email": "user@domain.com", ... }
```

Performance with growing data



Pattern 2: Object Indexing

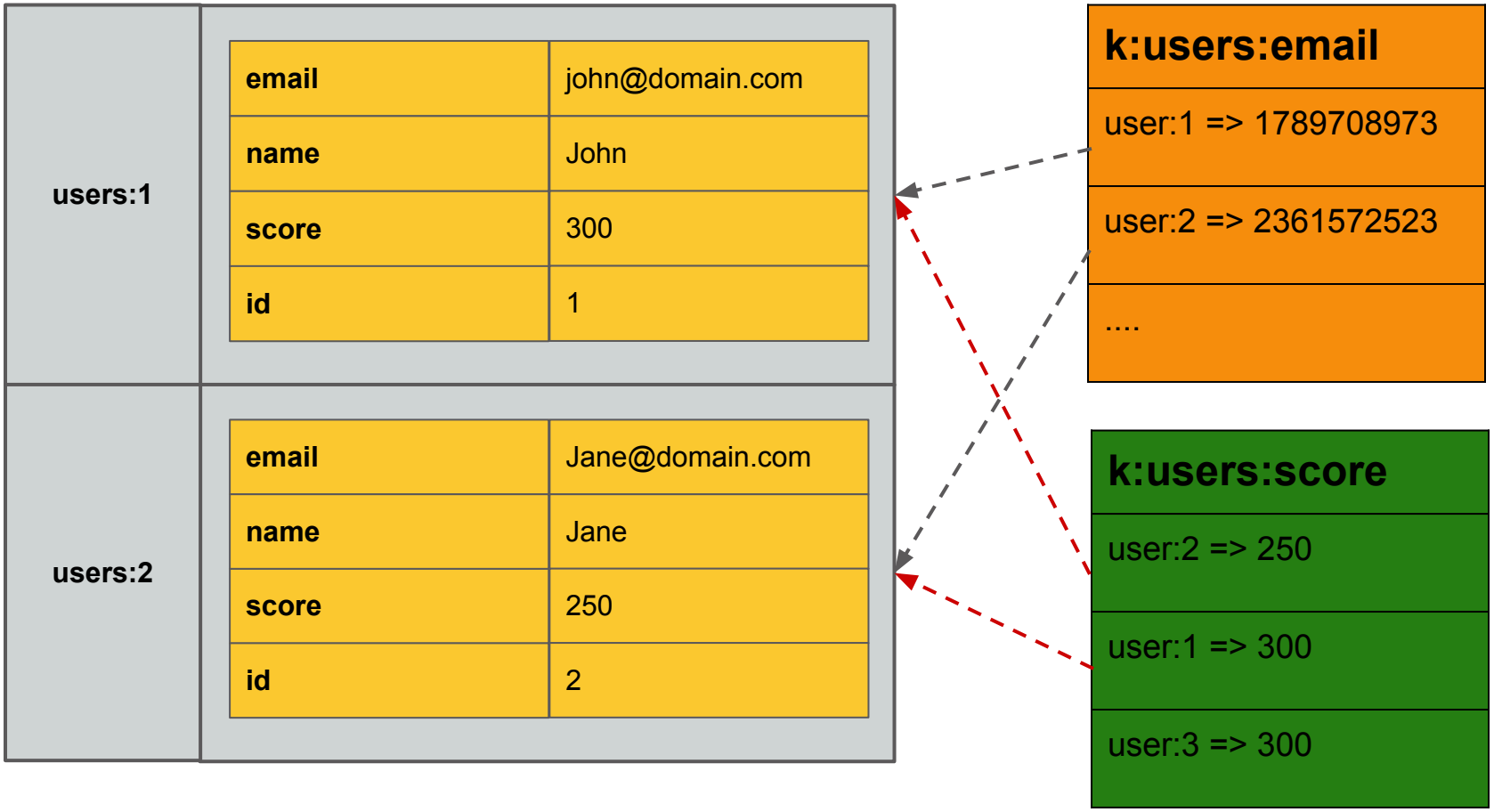
The problem:

- We want to index the objects we saved by various criteria.
- We want to rank and sort them quickly.
- We want to be able to update an index quickly.

Use cases:

- Tagging
- Real-Time score tables
- Social Feed Views

Indexing with Sorted Sets



Redis Pattern

- Indexes are sorted sets (**ZSET**s)
- Access by value O(1), by score O(log(N)). plus ranges.
- Sorted Sets map **{ value => score (double) }**
- So we map **{ objectId => score }**
- For numerical members, the value is the score
- For string members, the score is a hash of the string.

- Fetching is done with **ZRANGEBYSCORE**
- Ranges with **ZRANGE** / **ZRANGEBYSCORE** on numeric values only (or very short strings)
- Deleting is done with **ZREM**
- Intersecting keys is possible with **ZINTERSTORE**
- Each class' objects have a special sorted set for ids.

Automatic Keys for objects

```
class User(RedisObject):

    _keySpec = KeySpec(
        UnorderedKey('email'),
        UnorderedKey('name'),
        OrderedNumericalKey('points')
    )
    ....

#creating the users - now with points
user = User('user@domain.com', 'John',
'1234', points = 300)


#saving auto-indexes
user.save()


#range query on rank
users = User.getByRank(0,20)


#get by name
users = User.get(name = 'John')
```

```
> ZADD k:users:email 238927659283691 "1"
1
> ZADD k:users:name 9283498696113 "1"
1
> ZADD k:users:points 300 "1"
1
> ZREVRANGE k:users:points 0 20 withscores
1) "1"
2) "300"
> ZRANGEBYSCORE k:users:email 238927659283691
238927659283691
1) "1"
redis 127.0.0.1:6379> HGETALL users:1
{ .. }
```

Pattern 3: Unique Value Counter

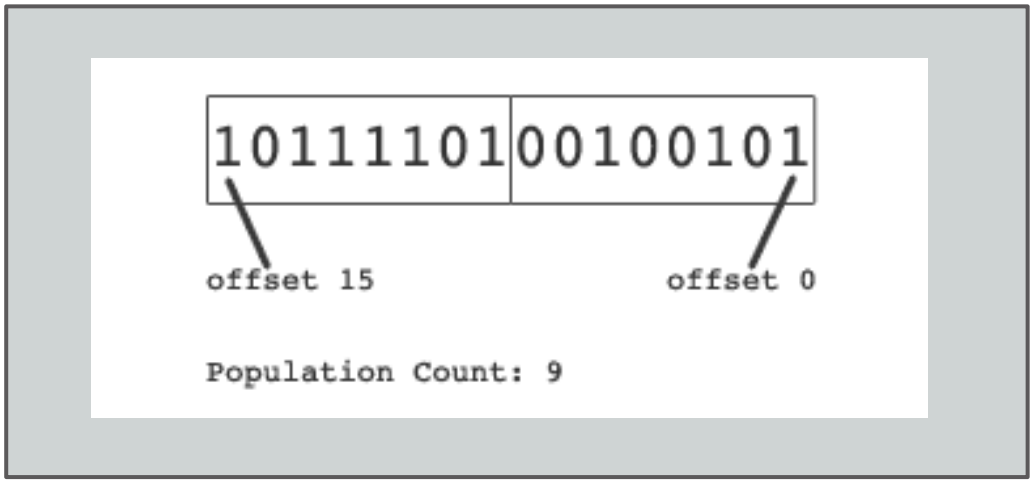
The problem:

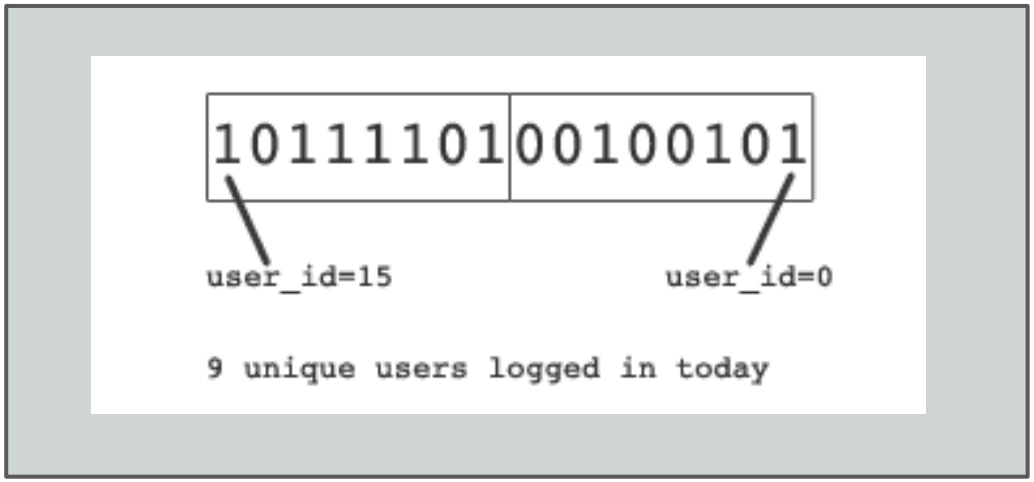
- We want an efficient way to measure cardinality of a set of objects over time.
- We may want it in real time.
- We don't want huge overhead.

Use Cases:

- Daily/Monthly Unique users
- Split by OS / country / whatever
- Real Time online users counter

Bitmaps to the rescue





Redis Pattern

- Redis strings can be treated as bitmaps.
- We keep a bitmap for each time slot.
- We use **BITSET offset=<object id>**
- the size of a bitmap is **max_id/8** bytes
- Cardinality per slot with **BITCOUNT** (2.6)
- Fast bitwise operations - OR / AND / XOR between time slots with **BITOP**
- Aggregate and save results periodically.
- Requires sequential object ids - or mapping of (see incremental ids)

Counter API (with redis internals)

```
counter = BitmapCounter('uniques',  timeResolutions=(RES_DAY,))

#sampling current users
counter.add(userId)
> BITSET uniques:day:1339891200 <userId> 1
#Getting the unique user count for today
counter.getCount(time.time())
> BITCOUNT uniques:day:1339891200

#Getting the the weekly unique users in the past week
timePoints = [now() - 86400*i for i in  xrange(7, 0, -1)]
counter.aggregateCounts(timePoints, counter.OP_TOTAL)
> BITOP OR tmp_key uniques:day:1339891200 uniques:day:1339804800 ....
> BITCOUNT tmp_key
```

Pattern 4: Geo resolving

The Problem:

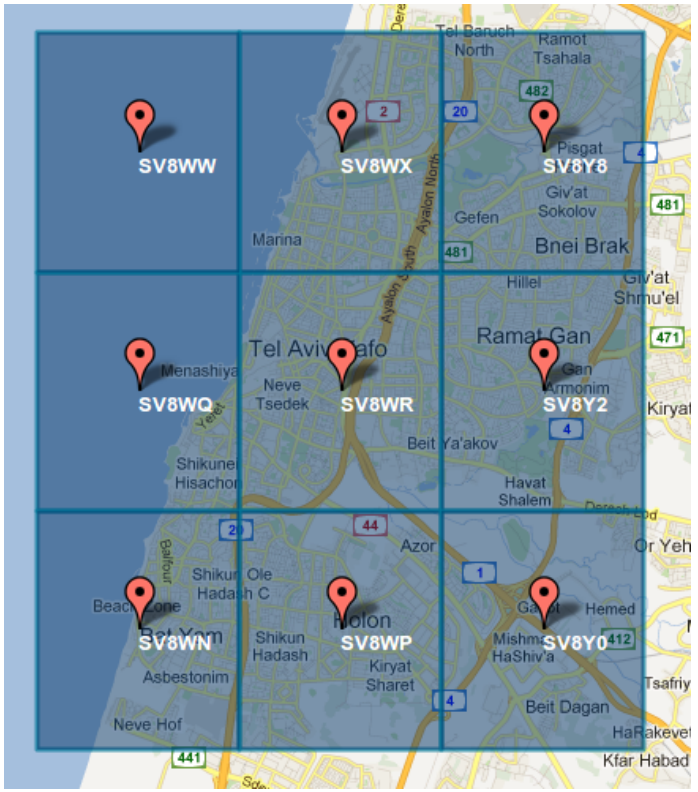
- Resolve lat,lon to real locations
- Find locations of a certain class (restaurants) near me
- IP2Location search

Use Cases:

- Find a user's City, ZIP code, Country, etc.
- Find the user's location by IP

A bit about geohashing

- Converts (lat,lon) into a single 64 bit hash (and back)
- The closer points are, their common prefix is generally bigger.
- Trimming more lower bits describes a larger bounding box.
- **example**:
  - Tel Aviv (32.0667, 34.7667) => **14326**45594530418103 5
  - Netanya (32.3336, 34.8578) => **14326**50217449870938 1
- We can use geohash as scores in sorted sets.
- There are drawbacks such as special cases near lat/lon 0.



Redis Pattern

- Let's index cities in a sorted set:
  - **{ cityId => geohash(lat,lon) }**
- We convert the user's {lat,lon} into a goehash too.
- Using **ZRANGEBYSCORE** we find the N larger and N smaller elements in the set:
  - **ZRANGEBYSCORE** <user_hash> +inf 0 8
  - **ZREVRANGEBYSCORE** <user_hash> -inf 0 8
- We use the scores as lat,lons again to find distance.
- We find the closest city, and load it.
- We can save bounding rects for more precision.
- The same can be done for ZIP codes, venues, etc.
- IP Ranges are indexed on a sorted set, too.

Other interesting use cases

- **Distributed Queue**
  - Workers use blocking pop (**BLPOP**) on a list.
  - Whenever someone pushes a task to the list (**RPUSH**) it will be popped by exactly one worker.

- **Push notifications / IM**
  - Use redis **PubSub** objects as messaging channels between users.
  - Combine with WebSocket to push messages to Web Browsers, a-la googletalk.

- **Machine learning**
  - Use redis sorted sets as a fast storage for feature vectors, frequency counts, probabilities, etc.
  - Intersecting sorted sets can yield SUM(scores) - think log(P(a)) + log (P(b))

Get the sources

Implementations of most of the examples in this slideshow:

https://github.com/EverythingMe/kickass-redis

Geo resolving library:

http://github.com/doat/geodis

Get redis at http://redis.io