Scaling Viki

Karl Seguin

License

Latest Version

The latest source of this book is available at: http://github.com/karlseguin/scaling-viki

Acknowledgment

Thanks to Cristobal Viedma. Great colleagues make all the difference. I think we built something really neat here; you're a good friend a good coder. Thanks for reviewing this and providing feedback.

Important Notice

The views presented in this book are my own and not those of my employer.

Scaling Viki

It had been almost a year since I quit working for a large international bank. That job had been a good-paying disaster; explaining the lengthy rest period I took thereafter. Working for the bank had been like traveling back to the start of my career at the end of the dot-com bubble: money, tools, and techniques used and wasted with little results.

The team I was part of found a way to apply Agile programming while eliminating all forms of agility. Ship often, even if there's nothing to ship. Write tests, no matter how trivial the code under test is, or how brittle the test was. Why are you alone at your desk mundanely inputing test data? Someone should be pairing with you! Within days, I suspected a bad fit. Within months, I'd resigned myself to stay the year and contribute as best I could.

This experience, while extreme, hasn't been unique for me. In fact, it's been the norm. There's something systematically wrong with our industry. An absolute state of mediocrity. Only my lack of exposure to other industries prevents me from claiming that this is a universal problem; although I suspect that it is. Visiting a doctor is an anxious affair; if the average doctor is anything like the average programmer, I'd rather stay at home, drink lots of water, rest and hope for the best.

My time at the bank wasn't a complete professional write-off. There's value in every experience. So great is the quantity of what I learned that I have no hope of recalling it all. Some of that is lost forever, some my mind will recall - seemingly out of nowhere - when appropriate, and some has changed me profoundly. It's disappointing that most was learned within a negative context. This, I think, is the main cause of burnout and depression in our industry. Personally, I take it better than some and worse than others. For the curious, my gap year was neither driven by depression nor burnout.

Examples of things I learned? CLOBs in a database are rarely ever a good thing. Pride is a horrible and disgusting human trait. Having fewer or even no tests is probably better than having really slow tests and definitely better than having flaky tests. I'm too prideful. Inspecting code which makes use of an ORM is likely to reveal a SELECT N+1. A friend is someone who deserves your absolute honesty; a lover should, occasionally, receive the mercy of a white lie.

Lessons learned and a year later, I found myself reading a job posting on Github. Both the technology stack and location were appealing. I applied. Got the job. Started a new journey.

Chapter 1 - Introduction

This book tells the story of how Cristobal Viedma and I created the platform that powers Viki: a video site focused on international content and community-driven subtitle translations. Our push to production proved to be an enlightening experience. I constantly found myself looking forward to the day's challenges while reflecting on the experiences and lessons I've accumulated in an otherwise unremarkable career. It was a strange but pleasant mix. A journey, I hope, which you'll find worth reading about.

Technically, there's nothing remarkable about Viki, Cris or myself. This book, while detailed with respect to our technical implementation, doesn't provide any silver, or even lead, bullets. Maybe our approach is ridiculously flawed and this book will serve no purpose other than embarrassing us. It'll be another lesson I'll grudgingly take.

The number of people doing remarkable work in the fields of computer science and computer engineering is limited. So too are the truly innovative companies. The rest of us face less difficult challenges. Nonetheless, there's something peculiar that we all face when writing code - namely, the code itself. Almost every method is unique. Every solution is different. Furthermore, we're all coding systems of ever-increasing levels of complexity. The number of people with experience in creating distributed high-throughput systems today is small compared to how many there'll be tomorrow.

This is exactly where we found ourselves. Tasked, more by ourselves than anyone else, to build a fast, distributed and reliable platform while lacking the experience to do so. Even today, months after we started, I'm overwhelmed with thoughts and emotions by this premise. Consider how different people might react to such a challenge. Some would shy away, being naturally risk-averse or lacking confidence. They might also be lazy or bad programmers afraid to be found out. Others would cherish the opportunity, as we did, seeing it as a rare chance to grow. Driven perhaps by arrogance, ambition, the challenge or the experience.

Also, consider what it means to have experience doing something. Surely, someone with direct experience building a similar system would be better suited than someone with no such experience. However, such people are rare, especially when you're a relatively unknown company headquartered in Singapore. Instead, perhaps we should consider indirect experience. One of the things that seems to separate a good programmer from her counterparts is the ability to take experiences doing one thing and applying it to something else. Extrapolating potential solutions or even just a starting point from an incomplete picture, or a gut feeling, is how you grow. This applies at both the macro and micro level. We'll look at more concrete examples throughout the book. For now, I suspect that anyone with experience interviewing applicants knows what I'm talking about. Some candidates don't know the answer, some make poor guesses or babble. The ones worth hiring though, make, or at least try to make, the leap from what they know to what makes sense. (Not that there's anything wrong with saying *I don't know* in an interview every now and again.)

I can't talk about experience without sharing something I learned from the bank. It's resoundingly true that there's a world of difference between twenty years of experience and one year of experience repeated twenty times. In some cases, indirect or no experience is better than too much.

How It All Started

Before being able to explore the path we took, you must first have some understanding of where we began.

I joined Viki at an awkward time. A recent change not only left a team of roughly 18 engineers leaderless, but also left a non-technical product team disconnected with engineers. The system, a monolithic Ruby on Rails and Postgresql application, was starting to burst at the seams. Outages were all too common and page load time was going from bad to worse. Seeing signs of slowed growth, product managers pushed for new features, only to exasperate existing stability and performance issues. Furthermore, a recent experiment of breaking engineers into small teams made it difficult for anyone to address system-wide issues (like stability and performance).

In a lot of ways, the code reminded me of the bank. Large and surprisingly complex, with frustratingly slow and randomly failing tests. Rather than leaky abstraction via Hibernate, it was leaky abstraction via ActiveRecord. Months later, I'd discover that deleting certain objects resulted in over 26 000 cascading changes to the database. Furthermore, the culture felt odd. The office was like a library. There was little to no discussion between engineers, and never any arguments. Dogmatic Agile was, once again, in full effect.

Thankfully, what did differ from the bank was management's awareness of these issues and their willingness to try and address them. While the day-to-day culture might have compared poorly to other startups, the underlying support for taking initiative was intact.

Roughly a month later, a new VP of Engineering was hired. At this point, Cris and I, the platform team, had a few modest successes behind us as well as a clear vision of where we wanted to go and how we wanted to get there. Additionally, we had gelled. When we looked or talked about code, we, more often than not, knew what the other was thinking. We argued, often intensely (especially on Friday for some reason). From this point on, we were given a certain degree of freedom to execute our vision.

The Platform

Our purpose on the platform team is to support Viki's API and other shared services. Previously, this had meant a fairly thin layer, largely composed of controllers, within the Rails application. Our mobile applications, along with a few partner sites, were our clients. The main website did not use the API, but rather had its own path to our shared models and database. This, we all agreed, had to change. While I think it's possible to prematurely adopt an API-based architecture, the ever growing number of clients in Viki's ecosystem called for a shared API.

Our ambition was greater than just having the web become a consumer of our API. First and foremost, we wanted to address the performance and reliability issues that were plaguing us. Based on some early prototypes, we felt we could deliver any request within 25 milliseconds uncached and within a local area network (10-100x faster than the existing system). We also wanted to build a distributed system which would help address both reliability issues as well as reduce network latency. Viki has a global audience with a significant number of users behind slow connections (either at the last mile, or via international pipes). Our top five cities are distributed across four continents. We already leveraged CDNs for our videos, but what good is that if it takes users six, ten or even twenty seconds to get all the data they need to start playing (with much of that time spent at the network layer)?

We also wanted to improve our development and operational environment. We felt that by moving away from one large codebase, we'd be better able to deliver new features as well as enable new clients. Furthermore, we wanted to improve our logging and general insight into our system's health. For us, that meant less but more meaningful logs which would essentially improve the signal-to-noise ratio.

Our approach was fueled by understanding two fundamental things about Viki. First, we're read-heavy. Second, what writes we have need not be replicated in real-time, or even close to it. It cannot be stressed enough how important it is to understand the actual operational requirements of a system. In all honesty, the only write in Viki that I'd hate to lose is user registration. Anything else I'm more than happy to batch up in an fsync every second (or minute!)

This is a good time to mention Viki's community team. Viki was, and continues to be, built on a community of volunteer translators. The community aspect of Viki has different operational requirements than the main website and mobile clients. It also has its own dedicated team. Where platform is read-heavy, community is more evenly distributed. Where most of our replication can happen in minutes, they are dealing with real-time collaboration. This presented an interesting challenge: community defines Viki but is dwarfed in terms of traffic and complexity by clients and platform. Which do you favor when designing your system, especially your data model?

We opted to lean on our yet-to-be-built platform's ability to work with stale data (slow replication). The community team would continue as-is, working on their own priorities, while we'd replicate data from the existing database into a platform-friendly model. If it took a minute or two for a new subtitle to propagate, so be it. This had the added benefit of letting us try and clean up our existing data. We could dump our failed polymorphic model and correct impossible data, like created_at values set in the future, orphaned children or duplicates that should never have been allowed.

It was to be a rewrite. It had to be fast. Codename: Hyperion.

At the time, none of this was flushed out. The core of the idea was really just to have an optimized data model fronted by a robust API application. Distributed, of course. Somehow, we'd keep it all in sync. In some places we had prototype code, but for very specific and low-level parts. Despite the vagueness, it never felt impossible or even particularly difficult.

Ultimately, two factors defined almost every decision we made. First, our own attitude towards risk and technology. It isn't a job, nor a profession. It's a passion. Secondly, a common sense understanding of the system's non-functional requirements. Almost every technical problem we faced was solved by taking a step back and remembering some basic truths about our system. Not only are we read-heavy and can accommodate slow replication, but in a lot of cases, we can apply fuzzy logic and return an approximation.

With a great deal of enthusiasm, we started to code.

Chapter 2 - The Core

Although there's nothing spectacularly difficult about the new platform, the core part of the system is built differently than most systems. The core is the part of our system that's responsible for finding lists of videos or retrieving details of a video. This includes getting the most popular, trending or other sorting options and getting videos by genre, available subtitles or other filters. It also involves other lists, like a user's subscription and curated content like featured videos.

A Redis-Based Query Engine

One thing that hasn't been mentioned yet is that our data easily fits in memory (at least, as far as platform is concerned). Part of the reason we felt we could deliver all requests within 25 milliseconds was precisely because we figured we'd be able to keep everything in memory. But memory isn't the final solution to all performance problems. Redis isn't fast only because data is held in memory; it's fast because, in many cases, fetching data happens in constant time regardless of how many items you are storing. Algorithms and data structures play as big a part in performance as the storage medium.

The origins of Hyperion can be traced back to a handful of concepts used to explore such data structures and algorithms for searching and sorting data. This was weekend-based experimentation at its finest. At best, I thought it might lead to an interesting blog post or two. Rather than storing data in a table and indexing it, what would happen if we stored the data directly in the application's memory? What if we used bitmap indexes? Sharded using hashes? Tried putting everything in sets and doing intersections and unions? What if we used Ruby, or Go, or Node? *What if*, *what if*, *what if*.

All of this happened during my first couple weekends at Viki, before anyone was thinking about building a new platform. Mostly, I was just messing around, exploring our data and features.

Bitmaps, A First Attempt

Initially, the most promising approach was using bitwise logical operations against bitmap indexes. Bitmap indexes are great when dealing with data with few unique values, like gender on a user's table. A lot of our searches are done against such values. For hundreds of thousands of videos, there are only a few genres and types. Modern databases will make use of bitmap indexes, but my initial focus was in storing these directly in the app's memory.

The way a bitmap index works is by filling an array of 1s and 0s to indicate whether an item has or doesn't have a specific property. For example, given the following documents:

```
{id: 1, genre: [1, 2], type: 1}
{id: 2, genre: [1, 3], type: 1}
{id: 3, genre: [2, 3], type: 2}
{id: 4, genre: [1, 2], type: 2}
{id: 5, genre: [2, 3], type: 1}
```

We could create indexes that look like:

```
  video | genre:1 | genre:2 | genre:3 | type:1 | type:2
   1        1         1         0        1        0
   2       1         0         1         1        0
   3       0         1         1         0        1
   4       1         1         0         0        1
   5       0         1         1         1        0
```

If we wanted to get all videos of `genre:2` and `type:1`, we could apply a bitwise and to those two indexes and end up with with a third bitmap:

```
video | genre:2 & type:1
  1             1
  2             0
  3             0
  4             0
  5             1
```

This means items 0 and 4 (corresponding to ids 1 and 5) match our query. A bitmap index is associated with a value based on the index position, like an array. Nowhere in our indexes or result do you see document ids. When your ids are auto-incrementing, the mapping can be implicit. Here, we know that it's id 1 and 5 by taking the positions and adding 1 (0+1 and 4+1). With GUIDs, you'd need to map the implicit order to the GUID, which is a small pain.

Bitmap indexes are efficient, specifically in terms of memory. You could have a hundred such indexes for a million videos and it would take less than 16 megabytes.

Unfortunately, we eventually gave up on this approach. First, the implementation was never quite as simple or clean as we liked. Secondly, although it was memory-efficient and fast, it wasn't fast enough. The bitwise operation was fast, but taking that result, extracting the matches, sorting and paging weren't. If you had one million videos, you essentially had to loop over the entire result (O(n)) to get all the indexes that were equal to 1.

Given what we've learned since, I think the problem was squarely with our implementation. I'd love to try it again. However, at the time, we decided to move to a different solution.

Sets

Our next attempt was to store our indexes as sets. Initially, this seemed both slower and less memory-efficient than bitmap indexes. However, once we stepped beyond artificial benchmarks, things started to look much more promising. Given the same documents as above:

```
{id: 1, genre: [1, 2], type: 1}
{id: 2, genre: [1, 3], type: 1}
{id: 3, genre: [2, 3], type: 2}
{id: 4, genre: [1, 2], type: 2}
{id: 5, genre: [2, 3], type: 1}
```

We'd create the following indexes:

```
genre:1 = [1, 2, 4]
genre:2 = [1, 3, 4, 5]
```

```
genre:3 = [2, 3, 5]
type:1 = [1, 2, 5]
type:2 = [3, 4]
```

To get all videos of `genre:2` and `type:1`, we could intersect the appropriate sets, and get the following result:

```
[1, 5]
```

As expected, both bitmap indexes and sets return the same result, but the implication of one solution versus the other are significant. Sets, unlike bitmap indexes, can be sparse. For example, imagine a new genre which only applies to video 1 and video 1000000. With a set, this is efficiently represented as:

```
genre:9001 = [1, 1000000]
```

While a bitmap index looks like:

```
genre:9001
1
0
0
0
0
0
... # 999993 more 0s
1
... # more 0s
```

In terms of processing, sets can be intersected in O(N * M) time, where N is the number of elements in the smallest set and M is the total number of sets. In other words, if a small set is involved, performance is great. Given three sets, one with ten items, another with ten thousand items and the last with a million items, an intersection will take be O(30). Given our data and how it was accessed, sets looked like the perfect solution. To seal the deal, sets and set operations are first-class citizens of Redis.

It felt like we had a winning solution, and over the following weeks, we'd improve and tweak it to fit our exact needs.

Managing Our Data and Indexes

One of the nice things about relational databases is that once you've created an index, you don't have to think much about it. The database engine takes care of keeping your indexes up to date. If we were serious about storing everything in Redis and relying on set operations for filtering, we'd need to manage indexes ourselves.

The easy part was storing our data. For this, we used Redis hashes. They key would be `v:VIDEO_ID`. A JSON representation of the object would be held in the `details` field, and other fields would hold other values we were interested in (without having to load the entire details). Creating a video looked something like (as a Lua script):

```
local data = cjson.decode(ARGV[1])
local id = data.id
```

```
redis.call('hmset', 'v:' .. id, 'details', ARGV[1], 'type', data.type, 'created_at', data.
    created_at, ...)
```

When creating a video, we'd add its id to the appropriate index. All our indexes would be stored in sets following a simple naming convention `r:x:NAME:VALUE`:

```
-- r:x:type:movie, r:x:type:episode ....
redis.call('sadd', 'r:x:type:' .. data.type, id)

--  r:x:genre:1, r:x:genre:2
for i=1,#data.genres do
  redis.call('sadd', 'r:x:genre:' .. data.genres[i], id)
end
```

Creating indexes was easy and cleanly isolated. Updating and deleting though didn't initially feel quite so right. Before deleting, do you load the `details` and remove the video's id from the appropriate indexes? Doable, but not clean. The solution ended up being much simpler. When creating a video, we'd use another set to track all of its indexes.

```
local index
local index_tracker = 'r:i:' .. id
-- r:x:type:movie, r:x:type:episode ....
index = 'r:x:type:' .. data.type
redis.call('sadd', index, id)
redis.call('sadd', index_tracker, index)

--  r:x:genre:1, r:x:genre:2
for for i=1,#data.genres do
  index = 'r:x:genre:' .. data.genres[i]
  redis.call('sadd', index, id)
  redis.call('sadd', index_tracker, index)
end
```

Given a video with an id of 9, a type of `movie` and two genres, 2 and 5, we'd end up with the following:

```
r:x:type:movie = [9]
r:x:genre:2 = [9]
r:x:genre:5 = [9]
r:i:9 = [r:x:type:movie, r:x:genre:2, r:x:genre:5]
```

To remove all the indexes when deleting this video, all we had to do was loop through the values at `r:i:9` and issue an `srem`:

```
local indexes = redis.call('smembers', 'r:i:' .. id)
for i=1,#indexes do
  redis.call('srem', indexes[i], id)
end
redis.call('del', 'r:i:' .. id)
```

With this approach, the entire process of managing indexes was isolated in the `create_video` script. Delete didn't know anything about indexes. Adding or removing new types of indexes became a non-issue. All delete cared about was looping through a set - you could put whatever you wanted in that set.

Finally, we implemented updates by doing a delete followed by a create. We had doubts about how well this would work and ultimately had to add the concept of a deep delete versus a shallow delete (the first used for a real delete, and the second used when updating). Even so, we were relatively happy with our data management strategy. It avoids having to load the object, find the difference and figure out which, if any, indexes need to be updated.

Find

Now that we had a way to manage our data, and a general idea of how to query it, we were able to start moving forward at a faster pace. Much of our energy was spent on writing the `find` path - a surprising amount of code would flow through it. Our initial implementation, worthy of taking us to the next phase of development, looked something like:

```
local intersect
local data
local result = {}

-- if only 1 set is provided, no intersect needed
if #KEYS == 1 then
  intersect = KEYS[1]
else
  redis.call('sinterstore', intersect, unpack(KEYS))
end

data = redis.call('sort', intersect, 'BY', 'v:*->created_at', 'desc', 'LIMIT', ARGV[1], ARGV[2],
    'GET', 'v:*->details')

table.insert(result, redis.call('scard', intersect))
table.insert(result, data)
return result
```

We'd be able to call this method with one or more filters and provide paging information:

```
find ['r:x:type:movie', 'r:x:genre:2'], [0, 10]
```

Redis' `sort` command was responsible for much of the heavy lifting. Many developers don't realize just how powerful this command can be. Understanding the *-> syntax is key to understanding the command. The asterisk (*) will be replaced by the value currently being examined, while the arrow (->) tells Redis that the left part is the key and the right part is the field of a hash.

For example, giving an `intersect` with the following values:

```
intersect = [50, 125, 359]
```

The above `sort` command would first sort these by the values at the following hashes:

```
key     | field
v:50    | created_at
v:125   | created_at
v:359   | created_at
```

And then retrieve (and store in the `data` variable) the values at:

```
key     | field
v:50    | details
v:125   | details
v:359   | details
```

This array of sorted details, along with a total count of matching results (for paging), would be returned to our code.

Over the course of development, `find` would get revisited over and over again. The following chapters will explore its evolution. But everything started with this simple script.

Foolishness?

I've often wondered if we were fools for not adopting a more traditional model. I have no doubt that it would have worked. Yet, at some early point, our set approach really started to resonate with both of us. Everything, including features and details we hadn't considered at the time, just fit in our model. Thinking in terms of set operations and having a concrete handle on our indexes became natural, even comfortable. (I'd argue that a downside of having indexes completely managed by relational databases is that they remain somewhat abstract).

Also, we weren't trying to develop a one-size-fits-all approach for all of our data. We were focusing on one aspect of the system. If different parts required a different approach, so be it. Maybe that sounds like a lot of work and complexity. Intellectually, I agree. Nevertheless, my gut feeling says that, at some scale, using the right tool for the right job is just as true for data modeling as it is for everything else.

Finally, I'm not suggesting that a set approach is right for everyone. I'm saying that it was right for us. Not just because it was a good fit for our data either. It was also a good fit for us. It was different and exciting and thus, it fueled our enthusiasm. I'm not sure that we'd have spent evenings and weekends refactoring, experimenting and tweaking our response times down to single digit milliseconds had it been just another CRUD-based app. Programming is like running - you won't get better until you go beyond what you can already do.

Chapter 3 - Holdbacks

The most complicated part of our system is the logic that decides whether a user can or cannot watch a video. We call this *holdbacks* and it integrates with almost every single possible query. We hate the idea of users following a link that we provided, say from the homepage or search results, and getting a *not available in your region* message.

The number of conversations we have which end with *"Ya, but what about holdbacks?"* followed by sad faces all around was considerable. Which isn't to say that we dislike holdbacks; it's a fun technical challenge and our content providers are as important as our users. Nor am I trying to imply that it limits what we can do. But it can, without question, make a simple feature much more complicated.

The Legacy System

In the existing system, to get a list of videos, the system would first query the holdback API for a list of restricted ids based on the user's location, platform and various other properties. These ids would then be passed into a `not in` statement to Postgres. I still remember the first time I saw Rails logging a `select` with thousands of values being specified in a `not in`. I'm still amazed at how well Postgres handles that. Nonetheless, it wasn't ideal. Furthermore, getting that list of ids wasn't a straightforward query - it was as much an SQL `select` as it was a mini-rules engine. Without Memcached, I don't think the system could have handled the load.

There was another problem that we were anxious to fix. Holdbacks had been implemented as a standalone system exposed as an HTTP API. APIs are great, but some things just need to be tightly coupled. It wasn't just about the performance, though that was certainly a major factor. Holdbacks is a first-class citizen of our entire data universe - as important as the title or mp4 location of a video. It didn't belong out there, abstracted by a `HoldbackManager`. It belonged right next to our indexes and video data.

Finally, the existing holdback system identified seven or eight dimensions which could be applied to either deny or allow a video from being watched. It also had rules that didn't make sense (they were always invalidated by rules with a higher priority). Finally, it had time sensitive rules that were long expired.

It might help to see a simplified example of holdback rules:

```
video | country | platform |    from    |    to      | allow
  1       SG        TV                                  F
  1       SG        Web      Jan 1 2013   Dec 31 2013   T
  1       SG        Web                                 F
  1       US        Mobile                              T
  1       US        *                                   F
  1       *         *                                   T
```

The rules are processed from top to bottom, stopping once a match is found. This video is never watchable from Singapore on a TV and can only be watched from the website during 2013. Furthermore, US users can only watch it from a mobile device. Anyone else, in any country and from any platform, can watch the video.

A Failed First Attempt

The story of our holdback rewrite is a lot like the story of our bitmaps and sets: our first iterations of both were failures. The idea was simple: we'd take our holdback rules and store them in our application's memory. The structure would be a hash containing linked lists. They hash key would be the video id, and each linked list item would be a set of conditions with either a deny or allow flag. To know if you could watch a video, you'd do something like:

```
function allowed(context, video_id) {
  var ruleSet = rules[video_id];
  for(var i = 0; i < ruleSet.length; ++i) {
    if (isMatch(context, ruleSet[i])) {
      return ruleSet[i].allowed;
    }
  }
  return false;
}
```

`isMatch` would take the context and compare them to the 8 dimensions of the rule to see if they all match. Something like:

```
function isMatch(context, rule) {
  return context.country == rule.country && context.platform == rule.platform && ...;
}
```

For an individual video, this worked well. For a list, not so much. There were two problems. First, much like the legacy system, we were still moving large quantities of video ids back and forth between holdbacks and our `find` system (in Redis). Sure, it was a lot more efficient (not over HTTP, possibly even on the same machine), but it was still an issue. Worse though was getting our page count. Finding 10 or 20 videos was plenty fast, but also getting a count of all matches (for proper paging) was well over 100ms.

It still wasn't as tightly coupled as it should have been, and the iteration over N rules for M videos was just bad.

Permutations

The idea of pre-calculating every possible combination of rules for given conditions had been floated around from the very beginning. With such a pre-calculated list, a check could be done in O(1) using a set. We'd get pen and paper and make some estimates. It never worked out; there are too many countries on this planet!

Seconds after we declared our first attempt a failure, we went back to the whiteboard talking about permutations. We talked to our business experts, our CEO, looked at licensing deals, and then went back to our existing data. We decided that, in reality, we only needed three dimensions: Country, Application and Platform (CAP). Some of the other dimensions were never used, two were mutually exclusive (and merged into Application), and the rest made more sense as filters or some other type of post-processing step. With 200 countries, 4 platforms and 100 applications, things looked more promising.

We built a prototype and started to load the data. It was larger than we were comfortable with - maybe 10GB. Doesn't sound like much, but we had to consider future growth: more videos, more applications and possibly new dimensions. We also had to think about our infrastructure. Given Redis' single-threaded nature, our plan had been to run multiple slaves per Hyperion cluster (a cluster being everything Hyperion needs to fulfill a request from a particular geographic region). That 10GB could easily turn into 100GB (new content * slaves), which, again, made us uncomfortable.

I can never remember who came up with the idea, but one of us suggested that we find matching permutations and alias them. The other one, undoubtedly driven by excitement (and possibly a bit of jealousy for not having thought of it first), quickly coded it. The result was great, holdbacks was less than 800MB. This was further reduced by switching to a 32bit build of Redis.

Integration

We had an efficient representation of holdbacks, but we still had to integrate it with our core system. This was probably around the time that our set-based filters started to resonate with us. By storing holdbacks as sets in Redis, it became nothing more than another intersection of sets. We debated whether holdbacks should be stored as a whitelist or a blacklist. A whitelist would take more space, but might be more efficient. It would also more neatly fit into our existing code as it would just be another set intersection. A blacklist would be more memory efficient, probably a bit slower, require some small tweaks to our code, but would be a more accurate representation of what holdbacks were all about.

Ultimately, we settled for a blacklist - it felt more natural. Besides, our everything-in-memory attitude made us frugal. We've always seen memory as an important but limited resource.

Therefore, for a given CAP permutation, a holdback key could be one of two things. The first would be a list of denied video ids:

```
hb:us:48:mobile = [38, 480, 19, 200, 3984, 40092]
```

The second would be an alias (implemented as a Redis string) to a real key:

```
hb:ca:48:mobile = hb:us:48:mobile
```

On each request, we'd resolve the real holdback key (if it's an alias, de-reference it) and pass this holdback key to our `find` method:

```
local intersect
local diffed
local data
local result = {}

-- do not intersect if only 1 filter is provided
if #KEYS == 2 then
  intersect = KEYS[2]
else
  redis.call('sinterstore', intersect, unpack(KEYS, 2))
end
```

```
redis.call('sdiffstore', diffed, intersect, KEYS[1])

data = redis.call('sort', diffed, 'BY', 'v:*->created_at', 'desc', 'LIMIT', ARGV[1], ARGV[2], '
    GET', 'v:*->details')

table.insert(result, redis.call('scard', diffed))
table.insert(result, data)
return result
```

The only difference with the previous version is that we `diff` the holdback set (`KEYS[1]`) from the intersected result. In other words, after we've gotten all the action movies, we subtract the ids that the user isn't allowed to watch.

Design Can Be Slow

As we moved forward, holdbacks would see some small changes. There were cases where we'd want to ignore holdbacks (if the user was an administrator, for example). We also spent a fair amount of time improving the performance of our permutation generator, including ways to further reduce the number of permutations.

If there's one lesson we learned, it's that some solutions evolve over a longer period of time than others. Permutations had always been on our minds; we'd explore the possibility at varying levels. Sometimes, it would be little more than *"If only we could store all the permutations"*. We explored various schemes to make it work, talked it over with colleagues, and even introduced it as an interview question (that didn't last long though).

I'm glad we didn't give up on the possibility. However, I'm sure holdbacks will be the first part of the new platform to see a rewrite. It'll take years to really solve this problem, in part because I expect the problem to evolve - more content and new types of restrictions. Some might see having to rewrite so quickly as a failure. I don't.

Chapter 4 - Tweaks

The core part of our system, including holdbacks, represented a disproportionately large amount of design and experimentation compared to the amount of code involved. The bulk of the code was yet to be written, but it would either be built on top of our foundation or use more traditional designs. Data that didn't need to be integrated with video data or holdbacks ended up either on the file system or in a Postgres database.

Furthermore, aside from a few conversations, we hadn't even begun to tackle the distributed portion of the system. Before moving on to that topic though, we made a number of tweaks to `find` which were worth exploring.

One thing to note is that, at this point at least, we were ahead of schedule and ahead of the other teams. We wanted to ship, but to what end? Without our infrastructure or clients, we were nothing. Our business experts would validate our work through a nice client, not raw JSON manipulated via query string parameters. At the rate we were going, there was plenty of time to build out Hyperion's secondary endpoints, figure out the distributed aspect, and have fun shaving milliseconds from our code.

Sorting

While our implementation was fast, it wasn't as fast as we had promised, at least, not under all conditions. The set operations were fast; the sorting and paging were not. Getting a page of action movies sorted by views might take 15ms. Getting a page of all videos sorted by views might take 100ms. The more you filtered, the smaller your final result was, the faster you could sort.

In our `find` function, we use Redis' `sort` command and supply a BY argument:

```
redis.call('sort', intersect, 'BY', 'v:*->created_at', 'desc', 'LIMIT', ARGV[1], ARGV[2], 'GET',
     'v:*->details')
```

We could sort by views or whatever value we had available simply by switching `v:*->created_at` for another field. However, even if you just want the first ten records out of a hundred thousand, you still need to sort the entire list.

We decided to store pre-sorted values in sorted sets. In theory, we'd be able to apply this as just another filter with the distinct advantage that we could stop filtering as soon as we had a page worth of results.

There were a few problems with this plan. First, we were using `sdiffstore` to implement holdbacks. Unfortunately, Redis doesn't have an equivalent `zdiffstore` command. Secondly, we have enough sort options that memory was a concern.

The first problem was solved by writing a specialized Redis command in C: `vfind`. Like our Lua-based `find`, `vfind` did intersections, holdbacks, sorting and detail retrieval. But it did it all much faster and did it against our sorted sets.

The second problem was solved in two parts. First, rather than creating sorted sets that held all videos, we created a sorted set that only stored the top videos for that order. For example, instead of having a sorted set containing all videos ranked by updated date, we'd have a sorted set with the 5000 most recently updated. Essentially, this approach optimized our code for retrieving the first X pages. Requests for pages 1, 2, 3, 4 … could be satisfied by `vfind`. If `vfind` failed to return a full page, we'd fall back to `find`.

That solved part of our memory pressure. The real issue though was that a number of sorts are country-specific. For example, you can list the most popular videos based on US views. Take 4 sorts * 200 countries and you end up needing a lot of memory. We cheated our way out of this problem. We analyzed views across countries and noticed that a lot of countries have nearly identical viewing behaviour. Why keep a list of top Canadian content when it's almost identical to the list for US? This, we convinced ourselves, also had the positive side effect of helping surface content. A kind of naive recommendation engine.

So how fast was `vfind` for our slowest (100ms) query? Around 15ms.

Paging

Even though we could efficiently filter and sort at the same time, we still had to count up results for paging purposes. Even here, `vfind` was a lot faster - it still had to count everything, but once it had filled a page, it really was just a matter of checking holdbacks and increasing a counter. No need to track all those values and then do a NLog(N) sort.

At this point, our website was starting to take shape and we noticed that a lot of queries were being used without paging. The homepage, for example, showed the top 3 trending movies, TV shows and music videos. We changed our code so that, by default, only partial paging information was returned. If you asked for 10 videos, we'd return 10 videos plus a boolean field that indicated whether more results were available or not. Getting a full count was O(N) where N was the total number of matches. However, to only find out if there's at least 1 more video available, it's O(N+1) where N is the number of videos you are requesting. In other words, with full paging details, N was 5000; without, it's 11.

This works well if you just care about top X results, or if you are doing endless scrolling or limiting your paging to *prev* and *next*.

As an aside, if you happen to be doing paging where you can maintain state (mobile or using Ajax) it's also possible to optimize this by only getting the total count for the initial load. There's really no need to get the total count each time the user switches to a different page.

Single digit: 9ms.

Serialization

That last tweak that we did, and most likely the first one we'll roll back, had to do with data serialization. We knew from past experience that rendering a view, whether it's HTML or XML or JSON, can be quite slow. Up until this point, it hadn't been a problem. However, with everything else tweaked, it was now the slowest part of our code.

Since we were dealing exclusively with JSON, there wasn't much to tweak. The only thing that seemed plausible was to directly return pre-formatted JSON strings. This fit quite well with our data model. Inside the `details` field of our video's hash, we already had the JSON body of the video. All we had to do was join the matching video details by a comma, wrap them in square brackets, and write it to the socket.

We had been doing some light processing on the response before sending it out to improve the usability of our API, but decided to give it a try and see how it worked (so far, none of our internal clients have complained!).

The change cut our response time to around 5ms. That, we decided, was good enough…for now.

Wasted Time?

I wouldn't disagree with accusations of having taken it a bit too far. However, the only time-consuming change we made, `vfind`, was also the most important. It was also the one we learned the most from. It wouldn't be a stretch for me to say that `vfind` and the accompanying sorted set generation was some of the funnest, funniest and most educational code I've written in a long time.

A lot of our performance testing was focused on single queries running locally. Obviously this is the easiest thing to test and tweak. While not ideal, a 10x performance improvement in such a controlled environment is bound to improve performance in the chaos of a production environment. We were also testing without the caching layers we'd have in production. Finally, our architecture was designed to scale horizontally. Supporting a high level of concurrency would at least partially be resolved by throwing hardware at the problem.

Chapter 5 - Surprisingly Reusable

Up until now we've been looking at our design exclusively from the point of view of videos. In reality, we've always had two first-class citizens: videos and containers. The only difference between the two is that containers can have zero or more child videos. That might sound like a big difference, but it really isn't. For all intents and purposes, the two are similar: both can be filtered, both can have holdbacks applied, and both have similar detailed information.

Initially the two had slightly different code paths; namely, we had holdbacks specifically for videos and holdbacks specifically for containers. At some point, we started to just think about both as resources and our code started to merge to work on *resources* instead of *videos* or *containers*. Even months after this evolution, I've found myself pleasantly surprised at just how generic our model has been.

As an example, let's go back to adding support for containers having zero or more videos. That sounds significant, but it's really just another set. Want to find all of container 1000c's videos? Simply query the `r:1000c:videos` set. You can take this further by applying another set to it, say `r:x:t:clips`, to get all of the container's clips, and then use all of the existing data to sort it however you want. It's all just about intersection of sets. All of this code goes through the same `find` function which does caching, filtering, holdbacks, sorting, paging and data retrieval.

(OK, to be honest, we have a fair amount of **`if`** statements to handle a variety of possibilities. The code is far from a perfectly reusable, unblemished masterpiece. For example, when retrieving a container's videos, we skip `vfind` since it doesn't have the necessary data and our normal `find` works in a few milliseconds for such small sets).

The main difference between our resources is the code to create them. They each have unique indexes (such as a video's container) and unique sort options (in addition to many shared ones). Since the list of a resource's index is tracked during creation, they all share the same delete code which just deletes everything in that list.

We've since introduced other resources and the only requirement for them to work is that they store something in the `details` field under a key named `r:RESOURCE_ID`. That's all `find`/`vfind` needs in order to function.

Unique Ids

The old system didn't use unique ids, a decision we initially decided to stick with, not because we agreed with it. It was simply too much trouble to change. However, to merge *videos* and *containers* into the same holdback sets, into *resources*, this is something we had to address. Given a holdback of resources:

```
hb:us:48:mobile = [38, 480, 19, 200, 3984, 40092]
```

How do you know if that's video 38 or container 38?

We considered GUIDs, but you probably aren't surprised to know it didn't fit very well with our everything-in-memory-with-lots-of-indexes approach. There was also some concern about mapping it to old ids. There aren't many things I dislike more than an `old_id` field.

The approach we settled on was to take the existing ids and append a character at the end. Thus container `50` became `50c` and video `9001` became `9001v`. This had a far smaller memory impact than using GUIDs. It also made it simple to map to the old system, and made debugging much easier: the id tells you what type of object you are looking at.

It’s worked very well for us. Certainly anyone migrating from a system without unique ids to a system with unique ids should consider it. For a new system? My gut says to use GUIDs, but I’m not sure why. I think I’m just afraid to take a stand on something so basic that goes against conventional wisdom. That’s not good.

Lists

In Chapter 4 we saw how sorted sets of pre-calculated sorting options were used to greatly improve the performance of our system. The same technique is used for almost any list of resources. The best two examples are a user’s subscriptions and curated content (like our featured content). Our concept of a list is truly that simple: a sorted set containing resource ids:

```
# sorted sets are inserted as: score1 value1 score2 value2 scoren valuen
r:9002u:subscriptions = [0 1000v 1 384v 2 50c]
```

By its very nature, the list is pre-sorted, so getting your most recently subscribed or least recently subscribed resources is trivial. In the above example, the score of each resource is an incrementing count (we actually use a creation timestamp).

What’s been fun about lists is how often we’ve said *“I think we can just use a list for this”* and how often that ended up being the case. They are fast, memory efficient, and, best of all, go through our same `find` logic. Again, this means that as soon as we populated a sorted set with resources that have a `details` fields, our core logic applies (most importantly holdbacks).

It’d be nice to say that the reusable nature of the system was intentional, but it wasn’t. Even as we implemented it specifically to improve holdbacks, it wasn’t at all obvious just how flexible yet simple we were making things. The same is true for lists. They were built for a very specific purpose, but as soon as we had the concept of resources, it became the right solution for many things. Our system is made up of millions of lists. And we’ve only scratched the surface.

Chapter 6 - Writes, Bloat and Queues

You wouldn't be able to tell from the way I've described the system so far, but we do have writes. From the beginning our vision for managing the distributed nature of Hyperion was vague. We envisioned having a centralized system which used queues to communicate changes in our data. We named this system Oceanus. Although centralized, a failure in Oceanus wouldn't ripple to our Hyperion clusters. At worse, write operations would fail, but users, mobile applications and other partners would still be able to use our system to browse and watch videos.

Garbage

Before we could think about accepting new writes, we had to figure out how to migrate data from our existing system into Oceanus. Worse, it was starting to look like both systems would be run in parallel for some time. Even though we eventually managed to shrink that gap, some tools, like our administrative CMS, would continue to operate against the old database for some time.

From past experience, it was evident that migrating data from new to old, not just once but continually, wasn't going to be pretty. A new project, named *Garbage*, was born. Garbage was a mess to work with, had no test coverage (consequently never a broken build either), and contained some truly bad code. Befitting its name, Garbage relied on triggers to detect inserted/updated/deleted rows in the existing system. These triggers would insert a row in a `changes` table, which contained the table name, row id, and, in some cases, some additional details.

The core of Garbage was a Ruby script which would pick up these changes, transform the data, and write it into Oceanus. Looking back, this simple to describe yet hard to write code might have been our most important project. We cleaned up so much data, so much legacy, so many unknowns, that it's hard to imagine we once saw Garbage as nothing more than an unfortunate side project. Oftentimes we broke things, but that always led to questions for which answers didn't always exist. What does it mean for a video to be *enabled* but *deleted*? What does the *license_exception* column mean? What's up with container id 2767? (I don't remember, but at some point it must have caused us so much grief that it's the sole member of a global blacklist in Garbage).

In the end, it wasn't just Garbage's value that we underestimated. To our surprise, after a couple major refactorings, it actually ran well and was flexible enough to adapt as needed. The real lesson though is the absurd cost of a sloppy data model. As Hyperion shows, I believe in optimizing code/data for reads, but only when paired with a clean and normalized representation. Without Garbage, we'd never have had that clean model to build an optimized representation from.

The Message Flow

With data synchronized to Oceanus, our next step was to push it out to our distributed Hyperion clusters. For this we opted to use a real queue. When a change was made (initially by Garbage, but eventually by our replacement CMS), an event would be written to the queue. The message would be simple, something like:

```
{resource: 'video', id: '598v', change: 'update'}
```

Hyperion Aggregator would be listening for these messages and use them to build up the denormalized message which Hyperion would serve. This step would pull in data from various sources to create near-final messages (Hyperion itself might do some minor tweaks to the document before saving it in Redis). Although final, the message wouldn't go straight from Hyperion Aggregator to the clusters. What if a cluster was down or slow? The generation of the message had to be isolated from the actual synchronization.

To achieve this, Hyperion Aggregator would add its processed message to another queue. Each Hyperion cluster would have a Hyperion Worker subscribed to this queue with the sole purpose of delivering the message. This extra step allowed clusters and workers to fail without affecting other workers or clusters.

Fail Fast and Be Idempotent

In most apps, reads and writes are served from the same place (a database). With our model, reads and writes were disconnected. Why did it work? First, we insisted that all communication be idempotent. Applying the same change from Garbage, or message from Hyperion Aggregator or Hyperion Worker would just work. Hyperion Aggregator kept a fingerprint of each resource and discarded updates which didn't alter the final document (as seen by clients). Hyperion treated all writes as upserts and wouldn't complain when deleting an already deleted resource.

We also decided to fail fast. A write to the queue was deemed as important as a write ot the database. Open a DB transaction, save the row(s) and enqueue a new message. We'd never save a DB record without the corresponding message. Also, our systems demanded that messages be processed in order. Creating then deleting a video can't be processed in any other order. Therefore, failure to process a message would stop the queue, raise an alarm, and require manual intervention. Some work could be parallelized. Our search index could use a different queue and worker from our recommendations.

Hyperion The Bloated

Around the time all these things were falling into place, we started to add more write endpoints to Hyperion. Most of these endpoints would be proxied through Hyperion into Oceanus. From the moment I finished writing user registration, I felt ill. Nothing about it felt right. Node.js suddenly felt like the wrong tool and proxying through Hyperion felt unnecessarily complex. I was sure we were doing something wrong, that we were butchering Hyperion.

A small group of us talked it over and we quickly came to a decision. Hyperion's responsibility was to let people watch videos. Other systems would be developed and given a well-defined responsibility. We'd have a system dedicated to user management (account creation, sessions, profiles), one for community-related jobs, and so on. We also decided that some systems would need to be isolated due to operational, rather than logical reasons. The relatively write-heavy user activity component (think *Last 10 videos that you've watched*) is a good example of a system that we felt would benefit from such isolation.

These systems wouldn't just be web layers; they'd be the authority for whatever data they were responsible for. Any system that wanted information on a user would go to our user management system (named Gaia). We'd also get the benefit of using whatever tools made the most sense for a specific system.

Since we didn't want to expose this implementation detail to clients, all services would use a single domain: `api.viki.io`. This would point to a custom Go cache/proxy which would know where to route each request.

In the course of a twenty-minute conversation, we went from having two systems, Hyperion and Oceanus, to having five. It might sound like we caused ourselves a bunch of extra work, but what we really did was properly organize and isolate the work that we already had. Even so, it became evident that there was more left to do than Cris and I could manage. Thankfully, two colleagues saved our asses: Albert Callarisa and Nuttanart Pornprasitsakul. Despite their own workload and deadlines, they volunteered to take responsibility for some of these.

Queues for Notification, APIs for Data

Having multiple authoritative systems wasn't easy. Who was responsible for what, and how did you present rich and meaningful data without coupling the systems? As an example, Gaia, the user system, had an endpoint to show the community roles a user has. On its own, this data is meaningless. *LetoA is manager of 123234c* isn't acceptable. It really needs to be presented with a summary of the container, which is Oceanus' domain.

Our approach was to lean on our queues. When a container changed, a message was already being written to the queue. So far, only Hyperion Aggregator cared about such messages. However, there was no reason Gaia couldn't also pay attention. When a container changed that Gaia cared about, it could ask Oceanus, via its API, for more details about this newly-updated container - perhaps opting to keep a local, non-authoritative, copy. There was some confusion about when to use queues and when to use APIs. To be pragmatic, we extended the queues' responsability of only notifying to, in some cases, also including a small payload.

In another life, I spent far too much time worrying about inversion of control. To me, the best IoC technique has always been event-based programming. The decoupling is absolute in that, for all you know, no one cares about your event. The queue architecture is exactly the same thing, but on a macro level. Make a change and write a message to the queue. Maybe nobody cares. Maybe someone does and they'll come back to you a second later asking for more details.

We're still finding our footing. Things are far from perfect. But something about it feels good.

An Alternative

We have considered an alternative approach at times. Rather than having Gaia care about container summaries, what if it only knew container ids? A request for a user's role would be intercepted and the container ids could be substituted on the fly (from a cache or by fetching from Oceanus) for container summaries.

The idea is inspired by Edge Side Includes (as found in Varnish, for example). On paper, we're big fans of the idea. We simply thought about it too late. It further decouples the systems while simplifying them. It probably doesn't perform as well, but I doubt that it'd be noticeable. I hope we get to try it soon.

Queues Keep On Giving

Despite this alternative, I wouldn't give up on queues. Having changes sent to a queue, where workers can pick up what they want whenever they want, opens up all types of possibilities.

For example, we recently rolled out workers which listen to various messages and send a purge request to our frontend caches. Rather than caching a subtitle for 5 minutes, why not cache it forever and explicitly update as needed? Users not only get better performance, they also see changes much faster.

I think most people use queues as a way to free foreground threads from blocking on long running jobs. That's a good reason to use queues, but I now feel that, in general, queues are being underused. They're an amazing way to decouple systems, distribute data and communicate. Making a stream of changes available, backed by APIs, is enabling. It also helps isolate failures. As long as messages are saved, a failed system can be restored and brought back to a fresh state.

Chapter 7 - Conclusion

It’s hard to end this story when I feel that there’s so much more to talk about. Initially I had hoped to intermix technical chapters with Peopleware-oriented ones. But, as I wrote, that just didn’t happen. Still, I can’t help but think that our real story, challenges and lessons were more about the people than the code.

Which isn’t to say that the entire technical side of the story was told either. Our distributed approach demanded centralized logging and monitoring. We set up logging systems around Kibana, Logstash, Statsd and Graphite, largely following various guides and blogs. While I generally suffer from a bad mix of NIH syndrome and cheapness, I had no problem with our decision to use ScoutApp for system monitoring. Self-hosted monitoring solutions have fallen far behind, while other hosted ones are unreasonably priced.

The devops effort alone warrants its own book (not to be written by me). Controversial to say, I know, but a programmer that can set up his own server and configure his stack is probably better than one who can’t. It’s worth knowing, especially considering how little time is involved.

Yet I come back to the people aspect of it. We had fun, we learned, we’re proud, and we added value for our employer. Throw everything else away because it barely registers against those four criteria. Be passionate. Put your money where your mouth is, especially when it comes to the virtues of risk. Learn.

Appendix A - Why Node?

The core of Hyperion is driven by Node and Redis. We do use other technologies, both within Hyperion and other components. None though, as extensively as Node and Redis. Interestingly, no one ever asks *"Why C?"* or *Why Go?"*, but they always ask *"Why Node?"*

The answer is simple: **To Learn It.**

This isn't the answer most people seem to expect. Surely there are better criteria for picking a language?

Consider the primary reason people normally pick a language: familiarity. It's compelling, no doubt about it. However, time and again, I've seen what happens to developers and companies driven by the comfort of familiarity. .NET developers who only know .NET, Java developers who only know Java, RoR developers who only know RoR, etc. tend to be significantly worse than those who explore, probably for two reasons. First, learning new things makes you better. Secondly, a programmer interested in learning is more than likely a programmer passionate about what he or she is doing.

It isn't that we didn't consider other factors. I had done enough Node to know it would be OK (and we always could have changed, if need be). Hiring was already difficult (being based outside of any hot market), and there was no reason to think this would significantly hurt or help us.

Curiosity should always be encouraged; comfort, extinguished.

Appendix B - I Love Premature Optimization

I hate programming adages that give developers an excuse to avoid work. Developers shouldn't test their own code. Laziness is one of our great virtues. Premature optimization is the root of all evil. I understand the spirit of these (well, except the testing one, that's just silly), but too often I see them wielded as an excuse for mediocrity.

I'm particularly weary of any such performance-related battle cry. The bar is now set so low that what some people call *premature optimization*, I call *making it work*.

There are a lot of reasons why optimizing code is important. Performance *is* a feature. Some even say performance is **the** feature. We're also seeing all types of shift in computing hardware: the move to portable devices and the move to slower multi-core machines. Also, performance can be something that's hard to fix after the fact - it normally has a major impact on the design decisions you'll make. If you don't stay on top of it and work diligently at it, it will get out of control.

The real reason why you should consider spending time optimizing your code, even for something trivial, is because it's a great way to learn. You'll discover all types of crazy things you didn't know about your language or a library you are using. Continuously benchmark your code, try variations, and learn from your observations. Whatever time you spend is a small price to pay for gaining fundamental knowledge about compilers, memory and processors.

I can't pass up an opportunity to talk about third-party libraries with respect to performance. We ran into a number of performance-related issues that were a result of some third-party library or tool. The clearest example was in Redis' `sdiff` command. When we first considered using it for holdbacks, I told Cris that I was apprehensive given its O(N) characteristics (where N is the total number of elements in all sets). Cris looked at me suspiciously and said: *"No it isn't"*. So I opened up Redis' website and showed him that it was - because that's exactly what it says. Cris pulled out a piece of paper and proved to me that both Redis and I were wrong.

We looked at the implementation and emailed Salvatore to propose an alternative implementation. Before we even had our editors open, he'd already committed a patch (`sdiff` will now use its inputs to determine whether it should pick the initial implementation or a new one, which is O(N * M) where N is the the size of the smallest set and M is the total number of sets).

There are two lessons here. First, don't blindly accept the work of another programmer just because he or she is better than you. I have a lot of respect for Salvatore and the work he's done, but we all make mistakes and we all tend to get a narrow perspective around our own code. Always be a critical thinker.

More importantly, third-party libraries can be a huge source of performance problem. Bugs tend to get fixed as they have an immediate and obvious impact. But poor performance can go uncorrected for a long time.

Appendix C - Our Simple AutoComplete

Search was the first feature that we planned to tackle after our launch. Until then, we'd simply make do with our existing third-party-hosted provider. But it was the end of the year and I had spent the last few work days in a funk. I needed a challenge to fuel me up and the four-day long weekend provided the perfect opportunity to put headphones on and code. I knew I wouldn't be able to tackle search, but autocomplete seemed doable.

I considered a few possibilities, read some blogs, looked at some open source code, and settled on storing the data directly in our application using a trie (which I pronounce as *try* because *tree* is absurdly ambiguous).

A trie is an ordered tree where all child nodes have a prefix associated with their parent. It's pretty easy to implement a trie using a hash. Here's what such a structure might look like if it contained the words *uni*, *unicorn* and *unicron*:

```
u: {
  n: {
    i: {
      $: 1
      c: {
        o: {
          r: {
            n: {$: 2}
          }
        },
        r: {
          o: {
            n: {$: 3}
          }
        }
      }
    }
  }
}
```

We use the $ symbol to represent a special key that holds the values id. It's possible to be more space-efficient by collapsing prefixes without ids, but let's keep things simple. Given the above structure, finding all the ids that have a certain prefix is just a matter of walking through a hash. First, we find the node that represents the user's input. For example, if the user entered `unic`, we could quickly walk down the hash to the 4th level (`c`):

```
# first we find the node that represents the ids
node = trie_root
for c in input
  node = node[c]
  return null unless node? # there are no matches
return load_ids(node)
```

Next we load every id from this point down through all child nodes (in our case, that would mean visiting both branches and getting 2 and 3). Since we are more concerned with performance than memory usage, we store the matching ids at every level. Therefore, our structure actually looks like:

```
u: {
  $: [1, 2, 3]
  n: {
    $: [1, 2, 3]
    i: {
      $: [1, 2, 3]
      c: {
        $: [2, 3]
        o: {
          $: [2]
          r: {
            $: [2, 3]
            n: {$: 2}
          }
        },
        r: {
          $: [3]
          o: {
            $: [3]
            n: {$: 3}
          }
        }
      }
    }
  }
}
```

The result is that we don't need a recursive or iterative method to load all child ids. We just need to access them directly from `node.$`. Regardless of how many values we store, retrieving all matching ids is O(N) where N is the length of the input.

This is only part of our implementation. We still need to apply holdbacks, sort them (based on whatever dimension we want, say, popularity) and get the details for the top matches. For this, we take the found ids and pass them to Redis. At this point, the code is similar to the rest of our video listing code. Unfortunately, the ids can't be definitively pre-sorted because many of our sorts are context-sensitive. However, we do store them using a best-guess effort, which hopefully results in less secondary sorting (I've never actually measured it, but I can't see how it could be worse than a random order).

Server processing time is around 2 to 4 milliseconds and it takes single-digit megabytes to store (more since we store a copy per process, have multiple versions, plus have it stored permanently in Redis; but the point is, it isn't much).

With the basic code in place, there are a number of options available to improve the quality of the results. You can transform the words in the trie and the inputed word to help deal with common issues. For example, we strip out many characters which helps avoid misses due apostrophe, dashes or spaces (we hated that in the old system *astro boy* worked but *astroboy* didn't). You can apply stemming and soundexes. You can even manually edit the trie based on analysis of actual usage.

I'm sure one day, maybe soon, we'll outgrow it, but it was fun to code.