Rethinking caching in web apps

Having spent a lot of the last few years worrying about the scalability of data-heavy applications like Rapportive, I have started to get the feeling that maybe we have all been "doing it wrong". Maybe what we consider to be "state of the art" application architecture is actually holding us back.

I don't have a definitive answer for how we should be architecting things differently, but in this post I'd like to outline a few ideas that I have been fascinated by recently. My hope is that we can develop ways of better managing scale (in terms of complexity, volume of data and volume of traffic) while keeping our applications nimble, easy and safe to modify, test and iterate.

My biggest problem with web application architecture is how **network communication concerns** are often intermingled with **business logic concerns**. This makes it hard to rearrange the logic into new architectures, such as the precomputed cache architecture described below. In this post I explore why it important to be able to try new architectures for things like caching, and what it would take to achieve that flexibility.

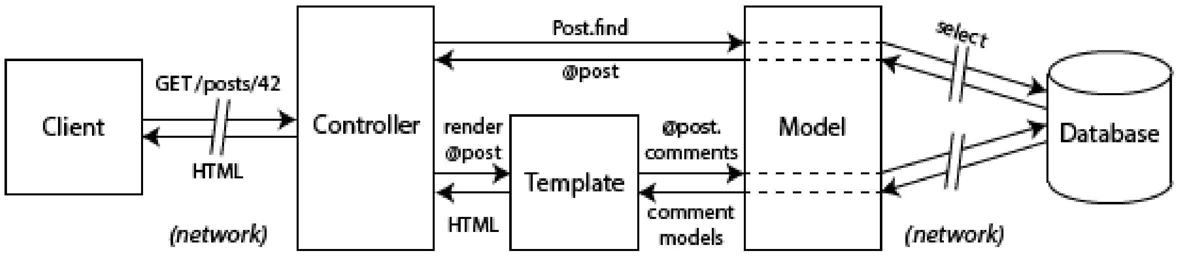
An example

To illustrate, consider the clichéd Rails blogging engine example:

```ruby
class Post < ActiveRecord::Base
  attr_accessible :title, :content, :author
  has_many :comments
end

class Comment < ActiveRecord::Base
  attr_accessible :content, :author
  belongs_to :post
end

class PostsController < ApplicationController
  def show
    @post = Post.find(params[:id])
    respond_to do |format|
      format.html  # show.html.erb
      format.json  { render :json => @post }
    end
  end
end

# posts/show.html.erb:

<h1><%= @post.title %></h1>
<p class="author">By <%= @post.author %></p>
<div class="content">
  <%= simple_format(@post.content) %>
</div>
<h2>Comments</h2>
<ul class="comments">
  <% @post.comments.each do |comment| %>
    <li>
      <blockquote><%= simple_format(comment.content) %></blockquote>
      <p class="author"><%= comment.author %></p>
    </li>
  <% end %>
</ul>
```

Pretty good code by various standards, but it has always irked me a bit that I can't see where the network communication (i.e. making database queries) is happening. When I look at that `Post.find` in the controller, I can guess that probabably translates into a `SELECT * FROM posts WHERE id = ?` internally – unless the same query was already made recently, and ActiveRecord cached the result. And another database query of the form `SELECT * FROM comments WHERE post_id = ?` might be made as a result of the `@post.comments` call in the template. Or maybe the comments were already previously loaded by some model logic, and then cached? Or someone decided to eagerly load comments with the original post? Who knows.

The execution flow for a MVC framework request like `PostsController#show` probably looks something like this:



Of course it is deliberately designed that way. Your template and your controller shouldn't have to worry about database queries — those are encapsulated by the model for many good reasons. I am violating abstraction by even thinking about the database whilst I'm in the template code! I should just think of my models as pure, beautiful pieces of application state. How that state gets loaded from a database is a matter that only the models need to worry about.

Adding complexity

In the example above, the amount of logic in the model is minimal, but it typically doesn't stay that way for long. As the application becomes popular (say, the blogging engine morphs to become Twitter, Tumblr, Reddit or Pinterest), all sorts of stuff gets added: memcache to stop the database from falling over, spam filtering, analytics features, email sending, notifications, A/B testing, more memcache, premium features, ads, upsells for viral loops, more analytics, even more memcache. As the application inevitably grows in complexity, the big monolithic beast is split into several smaller services, and different services end up being maintained by different teams.

As all of this is happening, the programming model typically stays the same: each service in the architecture (which may be a user-facing web server, or an internal service e.g. for user authentication) communicates over the network with a bunch of other nodes (memcached instances, database servers, other application services), processes and combines the data in some way, and then serves it out to a client.

That processing and combining of data we can abstractly call "business logic". It might be trivially simple, or it might involve half a million lines of parsing, rendering or machine learning code. It might behave differently depending on which A/B test bucket the user is in. It might deal with hundreds of hairy edge cases. Whatever.

At the root of the matter, business logic should be a pure function. It takes a bunch of inputs (request parameters from the client, data stored in various databases and caches, responses from various other services) and produces a bunch of outputs (data to return to the client, data to write back to various databases and caches). It is usually deterministic: given the same inputs, the business logic should produce exactly the same output again. It is also stateless: any data that is required to produce the output or to make a decision has to be provided as an input.

By contrast, the network communication logic is all about 'wiring'. It may end up having a lot of complexity in its own right: sending requests to the right node of a sharded database, retrying failed requests with exponential back-off, making requests to different services in parallel, cross-datacenter failover, service authentication, etc. But the network communication logic ought to be general-purpose and completely independent of your application's business logic.

Both business logic and network communication logic are needed to build a service. But how do you combine the two into a single process? Most commonly, we build abstractions for each type of logic, hiding the gory implementation details. Much like in the blog example above, you end up calling a method somewhere inside the business logic, not really knowing or caring whether it will immediately return a value that the object has already computed, or whether it will talk to another process on the same machine, or load the value from some remote cache, or make a query on a database cluster somewhere.

It's good that the business logic doesn't need to worry about how and when the communication happens. And it's good that the communication logic is general-purpose and not polluted with application-specific concerns. But I think it's problematic that network communication may happen somewhere deeply inside a business logic call stack. Let me try to explain why.

Precomputed caches

As your volume of data and your number of users grow, database access often becomes a bottleneck (there are more queries competing for I/O, and each query takes longer when there's more data). The standard answer to the problem is of course caching. You can cache at many different levels: an individual database row, or a model object generated by combining several sources, or even an entire HTML page ready to serve to a client. I will focus on the mid-to-high-level caches, where the raw data has gone through some sort of business logic before it ends up in the cache.

Most commonly, caches are set up in read-through style: on every query, you first check the cache, and return the value from the cache if it's a hit; otherwise it's a miss, so you do whatever is required to generate the value (query databases, apply business logic, perform voodoo), and return it to the client whilst also storing it in the cache for next time. As long as you can generate the value on the fly in a reasonable time, this works pretty well.

I will gloss over cache invalidation and expiry for now, and return to it below.

The most apparent problem with a read-through cache is that the first time a value is requested, it's always slow. (And if your cache is too small to hold the entire dataset, rarely accessed values will get evicted and thus be slow every time.) That may or may not be a problem for you. One reason why it may be a problem is that on many sites, the first client to request a given page is typically the Googlebot, and Google penalises slow sites in rankings. So if you have the kind of site where Google juice is lifeblood, then your SEO guys may tell you that a read-through cache is not good enough.

So, can you make sure that the data is in the cache even before it is requested for the first time? Well, if your dataset isn't too huge, you can actually **precompute every possible cache entry**, put them in a big distributed key-value store and serve them with minimal latency. That has a great advantage: cache misses no longer exist. If you've precomputed every possible cache entry, and a key isn't in the cache, you can be sure that there's no data for that key.

If that sounds crazy to you, consider these points:

- A database index is a special case of a precomputed cache. For every value you might want to search for, the index tells you where to find occurrences of that value. If it's not in the index, it's not in the database. The initial index creation is a one-off batch job, and thereafter the database automatically keeps it in sync with the raw data. Yes, databases have been doing this for a long time.
- With Hadoop you can process terabytes of data without breaking a sweat. That is truly awesome power.
- There are several datastores that allow you to precompute their files in Hadoop, which makes them very well suited for serving the cache that you precomputed. We are currently using Voldemort in read-only mode (research paper), but HBase and ElephantDB can do this too.
- If you're currently storing data in denormalized form (to avoid joins on read queries), you can stop doing

that. You can keep your primary database in a very clean, normalized schema, and any caches you derive from it can denormalize the data to your heart's content. This gives you the best of both worlds.

Separating communication from business logic

Ok, say you've decided that you want to precompute a cache in Hadoop. As we've not yet addressed cache invalidation (see below), let's just say you're going to rebuild the entire cache once a day. That means the data you serve out of the cache will be stale, out of date by up to a day, but that's still acceptable for some applications.

The first step is to get your raw data into HDFS. That's not hard, assuming you have daily database backups: you can take your existing backup, transform it into a more MapReduce-friendly format such as Avro, and write it straight to HDFS. Do that with all your production databases and you've got a fantastic resource to work with in Hadoop.

Now, to build your precomputed cache, you need to apply the same business logic to the same data as you would in an uncached service that does it on the fly. As described above, your business logic takes as input the request parameters from the user and any data that is loaded from databases or services in order to serve that request. If you have all that data in HDFS, and you can work out all possible request parameters, then in theory, you should be able to take your existing business logic implementation and run it in Hadoop.

Business logic can be very complex, so you should probably aim to reuse the existing implementation rather than rewriting it. But doing so requires untangling the real business logic from all the network communication logic.

When your business logic is running as a service processing individual requests, you're used to making several small requests to databases, caches or other services as part of generating a response (see the blog example above). Those small requests constitute gathering all the inputs needed by the business logic in order to produce its output (e.g. a rendered HTML page).

But when you're running in Hadoop, this is all turned on its head. You don't want to be making individual random-access requests to data, because that would be an order of magnitude too slow. Instead you need to use MapReduce to gather all the inputs for one particular evaluation of the business logic into one place, and then run the business logic given those inputs without any network communication. Rather than the business logic *pulling* together all the bits of data it needs in order to produce a response, the MapReduce job has already gathered all the data it knows the business logic is going to need, and *pushes* it into the business logic function.

Let's use the blog example to make this more concrete. The data dependency is fairly simple: when the blog post `params[:id]` is requested, we require the row in the `posts` table whose `id` column matches the requested post, and we require all the rows in the `comments` table whose `post_id` column matches the requested post. If the `posts` and `comments` tables are in HDFS, it's a very simple MapReduce job to group together the post with `id = x` and all the comments with `post_id = x`.

We can then use a stub database implementation to feed those database rows into the existing `Post` and `Comment` model objects. That way we can make the models think that they loaded the data from a database, even though actually we had already gathered all the data we knew it was going to need. The model objects can keep doing their job as normally, and the output they produce can be written straight to the cache.

By this point, two problems should be painfully clear:

- How does the MapReduce job know what inputs the business logic is going to need in order to work?
- OMG, implementing stub database drivers, isn't that a bit too much pain for limited gain? (Note that in testing frameworks it's not unusual to stub out your database, so that you can run your unit tests without a real database. Still, it's non-trivial and annoying.)

Both problems have the same cause, namely that the network communication logic is triggered from deep inside the business logic.

Data dependencies

When you look at the business logic in the light of precomputing a cache, it seems like the following pattern would make more sense:

1. Declare your data dependencies: "if you want me to render the blog post with ID `x`, I'm going to need the row in the `posts` table with `id = x`, and also all the rows in the `comments` table with `post_id = x`".
2. Let the communication logic deal with resolving those dependencies. If you're running as a normal web app, that means making database (or memcache) queries to one or more databases, and maybe talking to other services. If you're running in Hadoop, it means configuring the MapReduce job to group together all the pieces of data on which the business logic depends.
3. Once all the dependencies have been loaded, the business logic is now a pure function, deterministic and side-effect-free, that produces our desired output. It can perform whatever complicated computation it needs to, but it's not allowed access to the network or data stores that weren't declared as dependencies up front.

This separation would make application architecture very different from the way it is commonly done today. I think this new style would have several big advantages:

- By removing the assumption that the business logic is handling one request at a time, it becomes much easier to run the business logic in completely different contexts, such as in a batch job to precompute a cache. (No more stubbing out database drivers.)
- Testing becomes much easier. All the tricky business logic for which you want to write unit tests is now just a function with a bunch of inputs and a bunch of outputs. You can easily vary what you put in, and easily check that the right thing comes out. Again, no more stubbing out the database.
- The network communication logic can become a lot more helpful. For example, it can make several queries in parallel without burdening the business logic with a lot of complicated concurrency stuff, and it

can deduplicate similar requests.

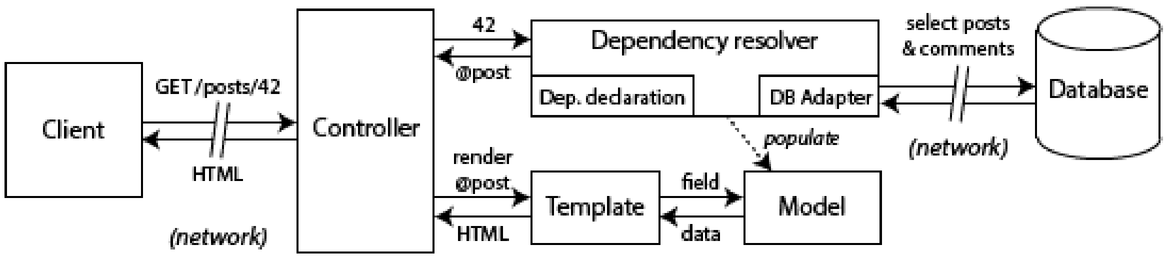
- Because the data dependencies are very clearly and explicitly modelled, the system becomes easier to understand, and it becomes easier to move modules around, split a big monolithic beast into smaller services, or combine smaller services into bigger, logical units.

I hope you agree that this is a very exciting prospect. But is it practical?

In most cases, I think it would not be very hard to make business logic pure (i.e. stop making database queries from deep within) — it's mostly a matter of refactoring. I have done it to substantial chunks of the Rapportive code base, and it was a bit tedious but perfectly doable. And the network communication logic wouldn't have to change much at all.

The problem of making this architecture practical hinges on having a good mechanism for declaring data dependencies. The idea is not new — for instance, LinkedIn have an internal framework for resolving data dependencies that queries several services in parallel — but I've not yet seen a language or framework that really gets to the heart of the problem.

Adapting the blog example above, this is what I imagine such an architecture would look like:



We still have models, and they are still used as encapsulations of state, but they are no longer wrappers around a database connection. Instead, the dependency resolver can take care of the messy business of talking to the database; the models are pure and can focus on the business logic. The models don't care whether they are instantiated in a web app or in a Hadoop cluster, and they don't care whether the data was loaded from a SQL database or from HDFS. That's the way it should be.

In my spare time I have started working on a language called **Flowquery** (don't bother searching, there's nothing online yet) to solve the problem of declaring data dependencies. If I can figure it out, it should make precomputed caches and all the good things above very easy. But it's not there yet, so I don't want to oversell it.

But wait, there is one more thing…

Cache invalidation

> There are only two hard things in Computer Science: cache invalidation and naming things. — Phil Karlton

How important is it that the data in your cache is up-to-date and consistent with your "source of truth" database? The answer depends on the application and the circumstances. For example, if the user edits their own data, you almost certainly want to show them an up-to-date version of their own data post-editing, otherwise they will assume that your app is broken. But you might be able to get away with showing stale data to other users for a while. For data that is not directly edited by users, stale data may always be ok.

If staleness is acceptable, caching is fairly simple: on a read-through cache you set an expiry time on a cache key, and when that time is reached, the entry falls out of the cache. On a precomputed cache you do nothing, and just wait until the next time you recompute the entire thing.

In cases where greater consistency is required, you have to explicitly invalidate cache entries when the original data changes. If just one cache key is affected by a change, you can write-through to that cache key when the "source of truth" database is updated. If many keys may be affected, you can use generational caching and clever generalisations thereof. Whatever technique you use, it usually ends up being a lot of manually written, fiddly and error-prone code. Not a great joy to work with, hence the terribly clichéd quote above.

But… observe the following: in our efforts to separate pure business logic from network communication logic, we decided that we needed to explicitly model the data dependencies, and only data sources declared there are permitted as inputs to the business logic. In other words, the data dependency framework knows exactly which pieces of data are required in order to generate a particular piece of output — and conversely, when a piece of (input) data changes, it can know exactly which outputs (cache entries) may be affected by the change!

This means that if we have a real-time feed of changes to the underlying databases, we can feed it into a stream processing framework like Storm, run the data dependency analysis in reverse on every change, recompute the business logic for each output affected by the change in input, and write the results to another datastore. This store sits alongside the precomputed cache we generated in a batch process in Hadoop. When you want to query the cache, check both the output of the batch process and the output of the stream process. If the stream process has generated more recent data, use that, otherwise use the batch process output.

If you've been following recent news in Big Data, you may recognise this as an application of Nathan Marz' lambda architecture (described in detail in his upcoming book). I cannot thank Nathan enough for his amazing work in this area.

In this architecture, you get the benefits of a precomputed cache (every request is fast, including the first one), it keeps itself up-to-date with the underlying data, and because you have already declared your data dependencies, you don't need to manually write cache invalidation code! The same dependency declaration can be used in three different ways:

1. In 'online' mode in a service or web app, for driving the network communication logic in order to make all the required queries and requests in order to serve an incoming request, and to help with read-through

caching.

2. In 'offline' mode in Hadoop, to configure a MapReduce pipeline that brings together all the required data in order to run it through the business logic and generate a precomputed cache of all possible queries.
3. In 'nearline' mode in Storm, to configure a stream processing topology that tracks changes to the underlying data, determines which cache keys need to be invalidated, and recomputes the cache values for those keys using the business logic.

I am designing Flowquery so that it can be used in all three modes — you should be able to write your data dependencies just once, and let the framework take care of bringing all the necessary data together so that the business logic can act on it.

My hope is to make caching and cache invalidation as simple as database indexes. You declare an index once, the database runs a one-off batch job to build the index, and thereafter automatically keeps it up-to-date as the table contents change. It's so simple to use that we don't even think about it, and that's what we should be aiming for in the realm of caching.

The project is still at a very early stage, but hopefully I'll be posting more about it as it progresses. If you'd like to hear more, please leave your email address and I'll send you a brief note when I post more. Or you can follow me on Twitter or App.net.

Thanks to Nathan Marz, Pete Warden, Conrad Irwin, Rahul Vohra and Sam Stokes for feedback on drafts of this post.

Join the discussion about this article on Hacker News.

Ghostery blocked comments powered by Disqus.

Subscribe

Site RSS feed

Enjoyed this? To get notified when I write something new, follow me on Twitter, subscribe to the RSS feed, or type in your email address:

Email: Subscribe

I won't give your address to anyone else, won't send you any spam, and you can unsubscribe at any time.

About

Hello! I'm Martin Kleppmann, entrepreneur and software craftsman. I co-founded Rapportive (acquired by LinkedIn in 2012) and Go Test It (acquired by Red Gate Software in 2009).

I care about making stuff that people want, great people and culture, the web and its future, marvellous user experiences, maintainable code and scalable architectures.

I'd love to hear from you, so please leave comments, or feel free to contact me directly.

Recent posts

- 12 Aug 2013: System operations over seven centuries
- 24 May 2013: Improving the security of your SSH private key files
- 05 Dec 2012: Schema evolution in Avro, Protocol Buffers and Thrift
- 08 Oct 2012: The complexity of user experience
- 01 Oct 2012: Rethinking caching in web apps
- Full archive