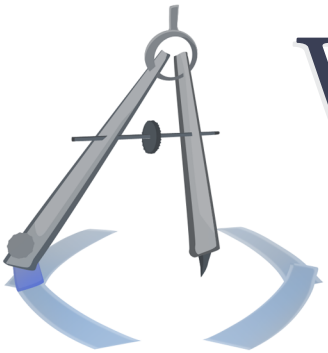


Website Architecture

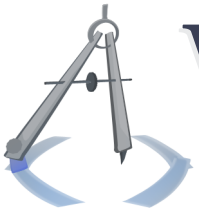
Lezione 16

Data-Driven Architectures

An introduction into data-driven websites, including characterizing them and optimizing them for their particular usage patterns.

Michael Serritella

Summer 2010



Intro to data-driven websites

Data-driven websites are websites whose primary content comes from a data set, which basically always means that there is a database involved, and which usually means that the authors of the site do not primarily *author* the content. Perhaps it is just information about outside events, like a list of movies and their actors and directors. Perhaps it comes from the site's users.

Data-driven sites can be done naïvely, and some significant percent of the time, they would work fine. But to optimize them requires a culmination of the knowledge we have seen in this course. We will see ways to optimize the front end of a data-driven site, which includes the PHP code and its SQL queries, and we will see ways to otimize the back end, which includes its database design and its choice of DBMS(es). To achieve this, we will also characterize data-driven sites by their producer/consumer relationships.

Ongoing example: A forum page

Throughout most of this lesson, we will use a forum page as an example:

- Its database is a typical database server, like PostgreSQL. It uses the purely relational model - e.g. a Threads table, a Posts table, a Users table, etc.
- Its front end/back end relationship is uncomplicated. It selects directly from tables.
- At the top of the forum page, a `SELECT` is done to retrieve the posts of the given thread.
- As each post is written, more `SELECT`s are done to retrieve information about each member. This might be a typical object-oriented design, where the outer `SELECT` gets only user IDs and the loop over each forum post will generate some kind of ForumUser object via the ID; internal to the object, antoher SQL query is made.

Many forums have a fairly naïve design like this, and they usually run into trouble where the server crashes after the forum becomes popular, which is only followed by a series of painful and/or expensive redesigns and hacks. Let's see how to prolong such a problem until as late as possible.

Front-end design

In this section, we talk about the PHP code, its SQL queries, and the resulting markup and file structure.

Minimize the number of queries

First: If we're using a DBMS server, like most heavy data-driven sites, then we should minimize the number of queries we have to make. Pack as much as you can into one query unless the query is ridiculously expensive. Like we saw in the SQL/DBMS+API lesson, there is inter-process-communication overhead and PHP+API calling & language overhead for every query.

Instead of fetching the posts and then fetching each member's information within a loop, fetch everything at the top of the page using join queries. This breaks - or maybe just bends - the object-oriented model, but we may see a way in which we can return to the OO model. You could even compromise by creating null- or blank-initialized Member objects, for instance, and then calling an `Initialize()` method within the loop that

iterates over forum posts. Then, once the Member object is built, use it as normal.

Something very vaguely like this:

PHP

```php
<?php
$threadResult = Database::Query("SELECT * FROM Thread, Users...");
while ($threadRow = $threadResult->fetch(PDO::FETCH_ASSOC))
{
    $memberForThisPost = new Member();
    $memberForThisPost->Initialize($threadRow["MemberID"],
                                   $threadRow["MemberName"],
                                   $threadRow["AvatarFilesystemPath"]);
    $memberForThisPost->DisplayMemberAreaOfPost();
}
?>
```

If there are twenty forum posts per page, you saved yourself 20 queries. If you have an online store and you're showing 120 products per page of search results - you get the picture.

Multiple shared servers for a basic CDN

In our basic forum, we only have one domain. There are forum posts at URIs like `www.tehForum.com/post.php?ID=32326`, and there are users' avatars and other miscellaneous images

at `www.tehForum.com/images/avatars/etc`. This is way suboptimal for both server-side and client-side reasons. For the client side, you may remember that the browser will only make around four simultaneous requests per each domain, so the browser is a bottleneck. On the server side, you're sending out one response with the HTML body and then you get 20 requests back. It's like giving someone a gift and getting back a 20-fingered slap in the face.

Your server does not want to deal with all of these requests building up so unsustainably. You should probably buy a few more hosting accounts and distribute your images on different servers. A shared hosting account right now is around $6/month if you buy a long enough contract. The next step up - virtual private servers - costs around $27/month, regardless of contract length. You could have all shared servers, or you could have your main forum server on a virtual private server and the rest of them shared. Shared servers typically promise you unlimited storage and bandwidth, so you could take advantage of that offer by storing all of your external resources there. You could even have four subdomains per each of these new domains, in order to further optimize browser behavior.

This is a basic Content Distribution Network. Even if you think buying more VPSes is better than buying more shared hosting accounts, just think that you could buy four shared accounts for less than one VPS account. All you need is a little assistance in concurrency, so you might as well buy 1 to 4 shared accounts and a VPS account for your forum. This is a prime setup, even, and is very resilient, all for around $50 a month. If your site is popular enough to warrant this optimization, then you're probably already making more money than this.

Inline images or other resources if appropriate

There is a bit of a downside to the CDN method, and that is that the external resources may load more slowly than optimal. It will probably be quite good, but if you demand that things like images load very quickly or exactly when you page loads, then you may want to inline your images instead of using a CDN for this purpose. You will at least save on the number of requests.

Efficient markup!

Now comes the bottom line. After all, in a data-driven site, you are often sending them a significant amount of markup. If you economize your markup, you see a nearly linear benefit in your total bandwidth costs. Most sites on the Internet are indescribably bloated - anywhere from 5% (and that would be an Olympic feat for a site nowadays) to 75%. Saving even 5% should be fantastic for you; think if you could save 50% of your bandwidth and even alleviate your concurrency problems. Be dilligent in writing clean markup, or rely on a program which takes care of it for you.

Back-end design

Now, how would you go about designing the forum's database? Which type of DBMS should you use? Should you use multiple DBMSes? And how do you divide their responsibilities?

Optimize for `SELECT`

We will see producer/consumer relationships later which may recommend otherwise, but generally, you should optimize your database structure such that `SELECT` executes as quickly as possible. Designing a database is more an art than a science; there are plenty of options and tradeoffs, and you don't always want to optimzie for constraints and correctness or economy of storage. So, at the very least, don't maintain those assumptions when you proceed in designing your database. You should generally assume that you will be doing `SELECT` a vast majority of the time.

DBMS organization

More than the database itself - which is simply the schema - what about the choice of DBMS? Let's assume for now that you have one DBMS. Here are the basic options and tradeoffs between a DBMS server and an embedded DBMS:

Size of data A DBMS server will be more apt to handle very large data sets, especially since the working environment (indices, etc.) will be more complex and may exist persistently; see the SQLite lesson for SQLite's disadvantages in this regard.

Frequency of updates Embedded DBMSes are not as well-suited to frequent updates w/r.t. the frequency of read operations (see SQLite). We will see a more in-depth analysis in the producer/consumer section, but for a basic level of analysis, you can say that frequent updates are appropriate for a DBMS server.

Complexity of queries DBMS servers (good ones (not MySQL)) are good at anaylzing complex queries and finding opportunities for optimization. These are usually not `SELECT` queries which are so complex in data-driven websites (or at least they shouldn't be), and maybe you don't care if the occasional `INSERT` or `UPDATE` is complicated. But if it *is* a complex `SELECT` that you're doing very often, you should probably be using a DBMS server.

OK - so this is if you have one database. Might you want more than one database, and then might you have a heterogeneous mixture of DBMSes? Here are some of the relevant tradeoffs:

Query result size If the query result size is large, then you should remember that it has to be communicated across the link between Web server and DBMS server. Perhaps the query is simple but has a large size; this is an independent variable. If the query result size is large, you may consider an embedded DBMS.

Communcation and congestion This has more to do with the frequency of queries - perhaps small ones; again, this is an independent variable. If your DBMS server and/or your Web server are mysteriously slow, it may be fielding too many queries simultaneously. An embedded DBMS may be more appropriate.

Multiplicity of databases Imagine you have millions of users, each of whom has a "virtual folder" or some collection of data. Or perhaps you have a catalogue of movies, each of which has an island database (i.e. doesn't relate to any other movie in the sense that one query in one DBMS can join from both).

In a DBMS server, each database may have its own set of credentials, and the DBMS program has to manage all of them in a catalogue of databases. If there are very many databases - like millions of them - then it will be slower and slower to look up any

particular database. With an embedded DBMS, you can make a file for each mini-database.

Might this be equally slow when you look them up in the filesystem? Not really. In the filesystem, you can have a `Big` folder, which has your top-priority databases your, and maybe you can have a sibling directory for `Small`, which, in some tree structure, collects millions of databases. The folder with only `{Big, Small}` in it will only try to find 1 item of a grouping of 2, and then the lookup within `Big` is economical.

Semantic priority of data Now, what about a heterogeneous mixture of databases? When might such a structure be a good idea? If you have millions of members, each with their own toy database, then we can classify that data as *their data*. Separately, there is *your data*, which is perhaps *about* these members. The backbone of your site runs on these queries upon *your* data. Lookups into the Users table should be fast, for instance, because you use it pervasively. Perhaps your own data is actually quite complex, as well. The performance of these mini-databases is far less critical.

So it looks like we have two kinds of problems here. In this case, you should probably use a DBMS server for your own data and a bunch of embedded-DBMS databases for the little guys.

Plan for growth and reconfiguration

At some point, you will probably have to grow your site and change your database structure. Perhaps you're taking that next step and implementing that next optimization. If your site is at all successful, this will happen at some point. Data-driven sites should be flexibly designed from the start, so that even structural changes have minimal impact.

Practically, this means that your PHP's SQL queries should change as little as possible. If you `SELECT` directly from tables, you are somewhat locking yourself in. Not forever, but it means that if the tables change, your application's queries all have to change. If you write software of large-enough mass, this is completely unacceptable.

Use views at the very least. Really, you should use SQL functions - even if they're bound to user-defined functions (or maybe you want to do this later!). If your database changes, just redefine the functions from within your DBMS. Functions offer the maximum in software-engineering benefits, including separation of interface from implementation and thus extensibility.

Your database structure *will* change, and you will be grateful if you used functions from the start.

Warning: The non-schema

Sometimes, a client will come to you with a structure that looks almost like a schema but isn't. It's not well-enough defined. For instance, they'll have a store with product categories and subcategories - sort of. There will be inconsistencies and unaccounted-for cases everywhere. This may be called the **non-schema**, and you should be frightened.

The vast majority non-computer scientists are godawful at organizing information, and even most computer scientists are not so hot. If you can talk with a client in the early stages of developing the *business* (or whatever other subject of this model), then perhaps you can elicit answers that will help you build a sensible schema from scratch.

If you have to build a schema from a non-schema, then **be extremely conservative from the start**. When you model this information, be as generic as you can imagine. Maybe just

one notch below the limits of your imagination. Otherwise, as the nature of the actual non-schema unfolds, you will have to change your schema drastically, and you will hit crushing setbacks. You have been warned.

Back-end maintenance

Here are a few tips for managing relational databases. Relational databases are actually quite difficult to debug and manage, because any one table doesn't have the whole picture - it usually has a bunch of ID numbers which don't immediately make sense to a person. If you miss a quote mark or a dollar sign in your PHP - or whatever - and one column with an ID value gets updated while the others don't, you will spend a long time figuring this out.

Perhaps prefer [`DELETE` and] `INSERT` to `UPDATE`

Let's say you have a table like WishList, where Users pick from Products and establish a wish list. The WishList table has mainly IDs, as it is a relational link between Users and Products. The Your Wish List page is a single page which shows all the items and allows the user to make changes. If a user makes only one change, like changing quantity or adding a comment, then you will likely try to track which one changed (by the way: how?) and then affect only the necessary row in WishList.

This turns out to be more work than necessary and is slightly error-prone. In any case, if an error occurs for any reason, you find yourself debugging a relational database, which is never fun. So how do you avoid errors and/or minimize time debugging?

Well, in the form processing page, you have all the data from all wish list items, since they were all on the previous page and they were all submitted with the form. So: How about you delete all the previous items in WishList and re-insert them? Or how about something to that effect.. How about you insert new ones and then leave all the old ones there, and whenever you need the wish list, you retrieve the newest one? You can add a Timestamp column and then sort by Timestamp when you retrieve the wish list. For most sites, this is no big deal.

If you only insert new rows, you minimize the chance that *some* PHP/SQL code wrote one column and *some other* PHP/SQL code wrote another column. In such asynchronicities, it's hard to know where to start looking in your PHP/SQL code when you detect a problem.

Build views for debugging

Early on, you should provisionally make views for any kinds of relations that you will later want to investigate. It doesn't matter if the `SELECT`s are efficient; it's just for your benefit while you debug. Join anything and everything you care about.

Synthesis

Now, let's analyze this with a holistic approach. Let's zoom out. Let's see the Zen of requests.

The Zen of requests

What does a user's HTTP request mean? It means: "Here's what I want. Go get it." Go *get* it. Not necessarily *compute* it. Are you "computing" things that are the same every time? Are you checking if-statements that, by definiton of your application design, are the same every time? A request is simple; obey its simplicity. Just go get it.

In terms of database-driven sites, this means that you should probably already have what the user wants. It's rare that the user actually poses some query to you that you have not comptued before or a request for which that you have to actually generate data. So: You should probably have *exactly* what the user wants, especially since you've produced it before for someone else. We're talking markup and everything. You have computed the markup before; why would you *compute* it

again? Infinitely many times?

You probably want to store the markup persistently. As long as it doesn't change - why not? Well, it's bigger, but whatever. You store the pure data in the database (e.g. the member's name, the forum post's text). The markup surrounding the pure data will always be bigger than the data itself, so now, if you want to store the markup, you are storing some data twice as much as you should, and then some. Isn't this a data management problem?

SQLite to the rescue

Let's assume you have all your primary data in a DBMS-server database. You'd like to not add the long markup strings in there, since there is communication overhead every time you `SELECT` it back - and it's even bigger than your normal `SELECT` would have been.

But wait. This should be an extremely simple, read-only operation. This is perfect for an embedded DBMS! So, let's store the primary data on a server, and whenever we generate new

markup that we expect to read very often, we'll add it to our SQLite database. Perfect.

Static markup, dynamic content

But wait. If we store the markup, it's infeasible to change it if the page changes. Well, perhaps we can only store the middle slice of the page - the largest chunk of the page which will not change. And better yet, perhaps we can only store static code which points to dynamic places.

Let's say a forum member changes his avatar image. He changes it from a toy truck to a bottle of ketchup. Should you have image `src`es that point to ".../toytruck.jpg" and ".../ketchup.jpg"? No; probably not. Point the `src` to ".../Bob89.jpg", which refers to his username. Perhaps hash the username just to keep the filenames a consistent length and to make sure there are no weird characters in the resulting filename. Perhaps hash it for some measure of privacy (by obscurity), especially if you incorporate some other, more private information into the hash.

In any case, point links to these generic places, whose content

you can overwrite later from your PHP pages which process these changes (e.g. avatar change). Similarly, you could have an advertisement whose `src` is "ad?ForumThreadID=235235". And, of course, all of these avatars and ads can be on separate servers in your CDN.

A return to form

We saw earlier in our forum-page example that if you do a `SELECT` at the top of the page and then a bunch of `SELECT`s inside your loop - one for each poster - then this can facilitate object-oriented design but also puts a huge drag on performance. Well, that cost analysis made a certain assumption. It assumed that the (`SELECT` + n*`SELECT`) happens every time the page loads, which is essentially an unlimited number of times with a large coefficient (n). What if you only compose the page once and then store the markup, like in our Zen method? Your runaway cost becomes a constant, and you don't so much care how long it takes to compose the page. So, you can fall back on OOP, even if it tends to incur more lookups than necessary. Don't go nuts - if you're composing pages extremely often, then maybe it's a bad idea - but it's a nice opportunity to return to form.

Producer/consumer relationships

Most of these best practices depend on certain assumptions. They assume that you will more often read from the database than write to the database. And that is overwhelmingly true. But what about when it is less true? Let's characterize data-driven sites by the strength of their producer- and consumer-driven behavior.

Consumer-dominant

The vast majority of data-driven sites are consumer-dominant. In this case, you should optimize your `SELECT` queries so they are as fast as possible. You should almost certainly use the triger-relational model, unless your site is so small that it doesn't matter. This does mean that the `INSERT`s and other intermediate/processing queries are going to be slower, but you almost certainly don't care. The input is usually getting typed or submitted by a human in this case, and they're not going to notice a one- or two-second lag when they press "Submit". It's better to impose lag on them than on each of your front-end users.

Examples: Control panels/Content Management Systems; news sites.

Producer-dominant

Somewhat rarely, you will have a producer-dominant site, in which a vast amount of data is inserted, significant portions of which may rarely or never be queried or whose queries are infrequent. In this case, you probably have a lot of data, perhaps from an outside source (e.g. environmental data, movie data, DNA sequencing). If you have a ton of data, and perhaps complex relationships, it's probably best to design your database for relatively efficient storage and natural relational semantics. This is the traditional way of designing databases outside of the website context.

In these types of sites, the consumers are probably searching for things, and the site is probably most obviously a database (e.g. "Internet Movie **Database**"). Thus, the consumer is probably OK with waiting a second or two for these big queries.

Examples: Scientific databases; movie databases; travel sites.

Producer/consumer parity

Often enough, there are equal amounts of producers and consumers. This is the hardest strucutre to design and optimize, because it sits in the middle of your tradeoffs. However, there is perhaps-interesting division in the middle of this category. Are the producers and consumers the same people?

Different parties

If the producers and consumers are different, then more than likely, the producers are members of your business and the consumers are the customers. So the question is: Who is OK with waiting? Any sensible business would probably allow the business side to be slower or less convenient than the customer side. So they will probably allow a lot of the optimizations that you would make for a consumer-dominant site.

Examples: Streaming news and video sites? Hyperconnectivity features of big-business websites (e.g. Where Is Al Roker This Second?)?

Same parties

Well.. what if they're the same people? What if a user of a forum makes a post and then he refreshes the page 10 more times? He's the producer and the consumer. These categories are the absolute hardest to optimize. You will likely have to use a little of both types of optimziations - but probably consumer-dominant, and you may want to see if you can exploit parallelism by having more databases and DBMS servers (i.e. "spend more money"). This is a tough call and is probably the most application-specific of any of the models, but exploiting parallelism is a good route.

Examples: Forums; social-networking sites.

Bonus topic: BLOBs

A BLOB is a column type in a database (sort of; the name will vary). It stands for "binary large object". This means you can actually store the contents of a file - like a JPEG - in a database column. Both SQLite and PostgreSQL have support for BLOBs, which are also called LOs ("large objects"). You can see PostgreSQL's documentation **here**, SQLite's **here**, and PHP's PDO documentation **here** (with PostgreSQL functions **here**). The syntax isn't so important for our lesson; even understanding how the database stores this data isn't so important. What's more important is how this affects your data-driven site architecture.

Proof of concept

First: How would this work? Let's get a mental image of this workflow.

It turns out that since BLOBs could be very L, you don't really access them by grabbing the entire object. You stream it. You ask for the first 1,000 bytes, the next 1,000 bytes, etc.

(or whatever). This is great for Web, which prefers streaming algorithms.

So - conceptually, here's how it goes. You could actually expose a URI which looks like a file - like "mySite/.../anImage.jpg". Using Apache and URI rewriting, you can transparently direct this request to a PHP script. In the PHP script, you probably want to start your HTTP response by saying that the content type is going to be "image/jpeg" or whatever is appropriate. You start a database query for this BLOB and, using some special function calls in the API, retrieve the first 1,000 bytes; you stick them into a PHP string[1]. Print the string, which just sends it out as part of the HTTP response as usual. Keep going until you have printed the whole file. And voila.

There are actually functions which wrap most of this process; see the PHP PDO documentation for more.

[1]Strings can store binary data just as easily as text; it just may look like gibberish if you try to print it out and read it as a human.

Lookup speed and convenience

In general, you should probably trust that a strong DBMS will look up data faster than an operating system looking through its filesystem. So let's just take it on faith for now that a DBMS has the speed advantage; a real comparison of some real products is far out of our scope.

Additionally, you can probably look up the image in a much more convenient way if it's in a database. You can search by whatever fields you want.. author name, album title, tags.. etc. In current filesystems, the only way you can get a hold of a file is by knowing its path. Your application may want to exploit a different lookup mechanism. Normally, you'd have to have these same lookups and then you'd finally get a filesystem path as a string, which you go and fetch as a separate operation. Now you can combine those operations to save on the number of disk accesses, at the very least.

Lookup speed, but not turnaround time?

There's more to this whole request & response than the database lookup time. Apache fields the request, and it probably does

some rewriting. Then it invokes PHP, which is a significant cost. And then the database has to talk to PHP via the API. Even if each of these things is pretty efficiently implemented, it's still more steps than the usual Apache-only request & response for an image. Though your costs might be lower, the user's experience will be marginally slower.

Abstraction of structure

Similar to the aspect of lookup convenience, you can enjoy the fact that files don't have to be placed in a heirarchical structrure, like that of a typical filesystem. If this isn't a very good fit for your application - like maybe you have semantic tags for your images instead of heirarchies - then you would probably like to store them directly in the database and not have some awkward mapping between a natural structure of images and their flat or heirarchical structure in the filesystem.

Apart from that, though, there are some slight security concerns. If a user uploads a file and you store it directly in your filesystem, then you have a file whose name is dictated by user input. Perhaps some filenames are problematic for your type of

filesystem, and a malicious user could exploit this. Perhaps the user could give the file a unique name and then, upon later gaining access to your filesystem, searches for this unique name and deduces something about where you store your users' uploaded files. Most filesystems' permissions are very crude compared to what you can do with a DBMS, so you may want to move all of this classified information into a better-regulated container.

Disk space

When the operating system allocates space on a disk for a particular file, it allocates that space in fixed-size units; these are called **disk blocks**. Typically, a disk block is 4KB. Even if a file only has one letter in it (which is 1B), it takes up 4KB of space on the disk. This waste of space is a type of **fragmentation**. If an operating system is given the job of storing three images, totalling 3KB, it may still take 12KB to do the job. If a DBMS is given this same task, it may actually take 3KB or just a little more, including some metadata. This is because the DBMS doesn't make a new file for each piece of data; it lumps the whole database into a few very large files, and it looks up database data at locations within these files.

If you have a data-driven site where you let users upload millions of photos, you should consider the cumulative fragmentation cost. If a 4KB disk block yields a simple average of 2KB of fragmentation for the last/leftover block of the file, then that's two terabytes per million photos! A vast majority of this wasted space can be recovered by using the DBMS's storage scheme.