SaleRadar

MONDAY, SEPTEMBER 8, 2008

Searching the Grid: BigTable and Geo Searches

Google's BigTable datastore looks like a relational database. It smells like a relational database. It even tastes just a little like a relational database. But it's no relational database. Google's not shy about this, either. In their words: "It's not a relational database!" Well, OK, I'm paraphrasing, but it really isn't.

Once upon a time, I wrote a multidimensional game of chance where, based on the statuses of adjacent sectors of a playing field, different options of attack and defense were possible. The framework of the game (who is playing, who's turn is next, what does the display look like) was all written in PERL, but once a player's turn started, all of the logic--ALL of it--was in MySQL. It was beautifully convoluted, and terribly processor and RAM-intensive, but man, did it keep my PERL code clean.

BigTable's query language ("gql") may look like SQL, but it's severely limited, and ever day there's someone new on the board saying, "This is so easy in MySQL, why can't I do X?", and the reason is that BT/GQL comes from a whole different design philosophy. Not better, not worse, but certainly different, and it takes a bit of pondering to get used to some of the ideas.

Before I get to my point, I'll go back a bit in history. I don't know how they came up with this plan, but if I had designed BigTable, this is what would have inspired me.

Most of you are likely familiar with RAID for storage and data security. The idea is that 10 cheap disks, set up correctly, can have better performance and reliability than one expensive, fast, disk that's, got the same storage capacity as the 10-disk array is configured to support. The 10 disks can have layers of redundancy spread across them, they can have the data distributed in such a way that effective latency is reduced, and the bus connecting them to the computer can be easily saturated for the greatest efficiency. And the best thing is that they're cheap. One disk fails, you pop a new one in, the data rebuilds, and you're safe until the next one breaks. With some RAID configurations, you might be able to lose several disks and still be able to save your data.

Google (and many other companies) took that same philosophy and applied it to their hardware as well. While they certainly may have some specialized systems in a lab somewhere, the hard work is done by lots and lots of cheap, easily replaced Linux systems in a big cluster. A node goes out, just pop a new one in. They go through a lot of hardware, I'm sure, but everything is fast, fast, fast, and--at least for search--they essentially never have downtime.

So here's the exciting part. BigTable extends that philosophy to data. Instead of making the datastore smart and letting you do all sorts of processor-intensive queries, they'd rather have you build in all sorts of redundant data when you save your objects so that they can index your information for easy searching.

A good example is geographic searches. A lot of people are searching for the solution to this question:

"How do I search for all the records in my database within Q degrees of my point, (long1, lat1)?"

In MySQL, it'd be easy: when you save, save the lat and long, and when you search, do something like:

It's not totally pretty, but depending where on the earth you are, you'll get something up to about a 20 mile by 20 mile square (less area and less square as you get closer to the poles, since the earth is round).

But this query isn't possible in GQL, as only one dimension of inequality is allowed. Basically, this is because, when they're done indexing your data, they want all the possible requests to be able to be represented as a simple flat-file list, so based on the bounds, they just pick and choose the right section of that list. Multiple inequalities means multiple dimensions, and that suddenly gets expensive in compute cycles.

Someone came up with a nice solution called geohashing, which is pretty cool. As you move north and east, the geohash that comes out of your lat/long combination gets higher in value; as you move south and west, it gets lower. All you have to do is do a geohash of your object when you store it, and then when you do your query, it looks something like this:

```
   upperBound = geohash(mypoint.lat + 0.15, mypoint.long +
0.15)
   lowerBound = geohash(mypoint.lat - 0.15, mypoint.long -
0.15)
   surroundings = db.GqlQuery("SELECT * FROM Points
WHERE geohash > :1 AND geohash < :2", lowerBound,
upperBound)
```

Very cute, very useful, but it has problems. First, there are accuracy issues that might force you to add extraneous digits in cases where the lattitude and longitude values you're searching on have few digits after the decimal. A hash of 39,-122 might look like "ABCDE", and a hash of 39.123,-121.98765 would look like "ABCDEFGHIJK". The degree of accuracy alone might throw off a comparison. As such, it's important to make sure that all your numbers have the same degree of accuracy, so tricks like turning 39 into 39.0000000001 and 39.123 into 39.1230000001 are important for making it work.

And again, all of this defeats the purpose of BigTable, it's a bit compute-intensive, and it's using inequalities where you don't actually need it, meaning that if you've got another dimension that you want to exclude on, you can't without a redesign.

So we come back to the redundant data option, and my solution to the problem, and that is to define the world as a distorted grid, save each point with a list of its neighboring grid squares, and then search on that.

The way I do it is this: In addition to the lat and long for each garage sale I save to the datastore (and some other metadata), I save a list I call "surroundings", and that list has nine ASCII values which are related to low-degree-of-accuracy long/lat points.

To wit: if I have a point at long -123.11231212312 and lat 33.4456789, I shift the digit over one, floor it, drop the decimal section, and save it as an ASCII coordinate representation. Those points become -1231,334. But that's just the grid square that the point is in (about 7 miles wide at the equator), and searching just for points within that grid won't give you great values if the originating point is on the west edge of that grid square. There might be a garage sale 0.1 miles to the west that will never be found.

To resolve this, I don't just save the grid square that contains the original point, but also all the grid squares around it. Once I've gotten the modified lat and long, I just take all the combinations of adding and subtracting one to each of them, and push them all into a list It looks something like this

```
grid = [str(mLon-1)+','+str(mLat+1),
        str(mLon)+','+str(mLat+1),
        str(mLon+1)+','+str(mLat+1),
        str(mLon-1)+','+str(mLat),
        str(mLon)+','+str(mLat),
        str(mLon+1)+','+str(mLat),
        str(mLon-1)+','+str(mLat-1),
        str(mLon)+','+str(mLat-1),
        str(mLon+1)+','+str(mLat-1)]
```

With this done, I save my point, including the metadata, the actual lat/long, and the 9 grid squares. It's pretty simple:

```
sale = SaleLocation(longitude = float(longitude),
                    lattitude = float(lattitude),
                    address = address,
                    surroundings = grid).put()
```

When I do a search on a new long/lat, I figure out the grid square that contains that point, and then I can simply search like so:

```
   points = db.GqlQuery("SELECT * FROM SaleLocation
WHERE surroundings=:1", myPoint)
```

Because of the way lists are compared, if your search point is in any of the nine grid squares associated with a stored point, that stored point will return.

I chose 0.1 degrees because it's mathematically convenient, and nine grid points because it's easy, but this could easily be modified or extended to any shape or size. Once you've got your results back, then you can cut with a finer blade still, only displaying points, say, that are in a specific circular radius, or some other permutation. But this resolves the problem.

This solution can be applied to other range problems as well, to avoid using inequalities. If you have items in a database that have creation and expiration dates, stop thinking about bounds, and instead create a list of all of the dates that they are valid, to whatever degree of accuracy that is required.   That way instead of searching for items where creation
It's sort of a brute force option, but I get the feeling that this is how the Google search engine works. Lots and lots of cheap data, divided up across lots of infrastructure; all the work is done in creating the entity so that whatever way you're likely to search for it, it'll come back super fast, with little expense on the back end.

It makes sense, too. In many applications, the information is posted once with the intention of being read many (tens, hundreds, thousands of) times. Spending a little extra time on the back end to figure out how it'll be read the fastest, and spending a few extra bits to make it easy to find pays off big time on the front end.

Posted by Benno at 10:36 PM 0 comments
Labels: bigtable, geocoding, geohashing, redundant array of inexpensive data

What is SaleRadar?

So what is SaleRadar? The idea came from driving around and seeing sparsely-attended garage sales throughout my home town, and I thought I might be able to build an easy-to-use system for advertising these sales, that might encourage a sort of grassroots economy to help people in the current downturn in the economy.

The design philosophy I wanted to follow is what I call the three S's: Simple, Simple, Simple. I am planning to add more features later, but at the core of it, I wanted people to be able to choose if they wanted to post or search, and to enter and address where they were either hosting or searching for garage sales.

I got down and dirty with the Google Maps API, and after a little wrangling, I got some basic functionality. Building on one of their samples, I was able to do some searching and posting. One word to the wise: pay attention to that clap-trap about not "unrolling" the marker object. If you want to show multiple markers, you have to take an extra step that's a little counter-intuitive, but that's well documented (although I ignored it at first), so I won't cover it here for now.

Within short order, I was proud to find that I had a system which could take requests, do geocoding lookups, save new listings, and display existing listings when a search was made. But then I read some of the fine print--well, it wasn't *that* fine, but still--and found that Google Maps geocoding engine only accepts 15,000 requests per day from a given IP address, and while I don't know too much about the ins and outs of Google's infrastructure, I ran a test where my app would query my personal server every time I hit a certain URL, and regardless of where the request to my Google AppEngine app originated from--several servers I have access to around the country and the world--the same Google IP address was hitting my servers.

My understanding is that when hitting GAE apps, the user will hit infrastructure close to him or herself, but outgoing HTTP requests all seem to be coming from one proxy, and since I was getting those geocoding requests via HTTP requests, it seemed possible that Google might see them all coming from the same location. I hope not, but I figured it was worth the challenge to move those requests out to the user layer, in the browser.

So a couple of hours of broken application later, I made it so that all geocoding requests come from the browser, and then the browser queries the SaleRadar system asynchronously, and gets an XML response with either an acknowledgement that the sale was saved, or a list of locations. Now, I realize that I've got nowhere near 15,000 visitors a day (just me, for now!), but this actually solved a few problems:

- If (when?) my app becomes uber-popular, I don't have to worry about running out of free Google geocoder requests.
- If (when?) my app becomes uber-popular, I save a bit on bandwidth and processing, since the browser is doing a pretty good chunk of the work.
- It made the user interface nicer: no page reloads, since everything is AJAX driven.
- It had the side-effect of creating an API; if I want, I can publish it so that people can post or search my database from other apps. I need to add some security if I'm going to go that route, but the option is there.
- It made it easier to separate out the front-end from the back-end, without having to dive into templates. Effectively speaking, all user interactions are with a simple HTML/Javascript page, and it's only the AJAX calls that talk to the Python AppEngine code.

Previously, I'd had the Javascript interspersed with my Python, and it got pretty ugly in short order. I plan to learn Django templating, but suddenly I found I did not need to do so, so I even saved myself another step.

But one thing was troubling me: searching for map coordinates surrounding a given point. The BigTable datastore system was confounding me with its limitations, and based on chatter in the AppEngine group, I wasn't alone...

Posted by Benno at 10:07 PM 0 comments
Labels: craig's list, etc., garage sale, map

Working on Cloud Nine

Well, maybe it's just cloud three, but you know what I mean.

I've been involved in Internet-based businesses since 1994, everything from designing and building datacenters to managing servers, to designing, implementing and supporting large database-driven applications. I've worked in a number of environments and supported clients in a wide range of industries, and every couple of years, I've watched as the barrier to entry for businesses keeps getting lower and lower. While there are certainly some problems associated with this--a web presence can give a fraudster more apparent credibility--the benefits are outstanding. Large corporations will always have their place, but if the leader in a market niche does not address the needs of all of their clients, the environment is getting more and more friendly for small, agile companies to meet those needs, either in complement to or competition with the larger corporations.

The latest such democratizing technology that is in my sights is Google AppEngine (GAE). There are other competing cloud technologies, most notably Amazon Web Services (AWS), which has stand-alone services for bulk data and database storage, process queuing and virtual machine-based processing power. I spent some time playing with AWS, and it's got a really powerful feature set, but GAE takes a tack that speaks more to my desired approach to cloud computing.

The main thing is simplicity. AWS seems to be trying to make the old guard more comfortable by simulating datacenters. If you want to scale your application, you build it to automatically spawn new virtual servers, and build it such that those virtual servers load balance effectively, and drop off appropriately when load requirements are down. Data and process synchronization is up to you, the developer.

With GAE, it's kind of like they took AWS and applied BestBuy's "easy button" philosophy. There are some features that may be missing, and some developers feel very strongly about those features, but for me, the fact is, I can write my application, and if it gets hit once, it's available. If it gets hit a million times, it's available, and (once GAE is out of beta), I just pay for the added volume of transactions and associated processing/storage as needed. I don't need to modify my code. The datastore (BigTable) is part of the same service, so there is no special key management or other hoops to go through in order to get and put your data. While I would not call AWS byzantine by any stretch, I found that what I was able to accomplish within a few hours of discovering AWS, I was able to do within a few minutes of discovering GAE.

Of course, one key word here is "beta". GAE is in beta testing, with no road map concerning feature updates or release dates, and so many people are hesitant to commit to it. I'm keeping my ear to the ground on this one, because without knowing what's coming, it's hard to treat this as more than a hobby.

But that having been said, I'm just thrilled that I don't have to configure Apache, that I don't have to update any kernel. That I don't have to synch any *SQL databases. That I don't have to harden my linux boxes. That I don't have to... you get the picture.

So this blog is about the development of an application I'm working on, currently called SaleRadar, although that name may change. I'm building it from scratch in the GAE environment, though, so much of the content will be focused on the intricacies of solving problems with that system's restrictions. Google has, with GAE, reinvented a few wheels, and it's interesting to pick apart what's gained and what's lost with each of these revisions; it forces the developer to think about their application's architecture differently than they might have.

Finally, there will likely be some discussion if the ins and outs of Python in this blog, because I have typically worked in PERL (with some PHP and Java) on the server side, and Javascript on the client side. Currently, Python is the only language supported by GAE, and while my initial reaction was that they should also support my language of choice (and there were choruses of similar shouts about other laguages), I figured I wouldn't wait around, so I've dived into Python as well, and found myself appreciating the move more than I expected to.

A basic version of SaleRadar is already working. In my next post, I'll talk about some of the steps I took to get there before I discuss some of the changes I have planned for the near future.

Posted by Benno at 12:34 PM 0 comments
Labels: barrier to entry, cloud computing, gae, google app engine

Subscribe to: Posts (Atom)