Building OpenDNS Stats

Richard Crowley
richard@opendns.com

Hi. I'm Richard Crowley and I work for OpenDNS, which is…

Then: 8 billion DNS queries per day

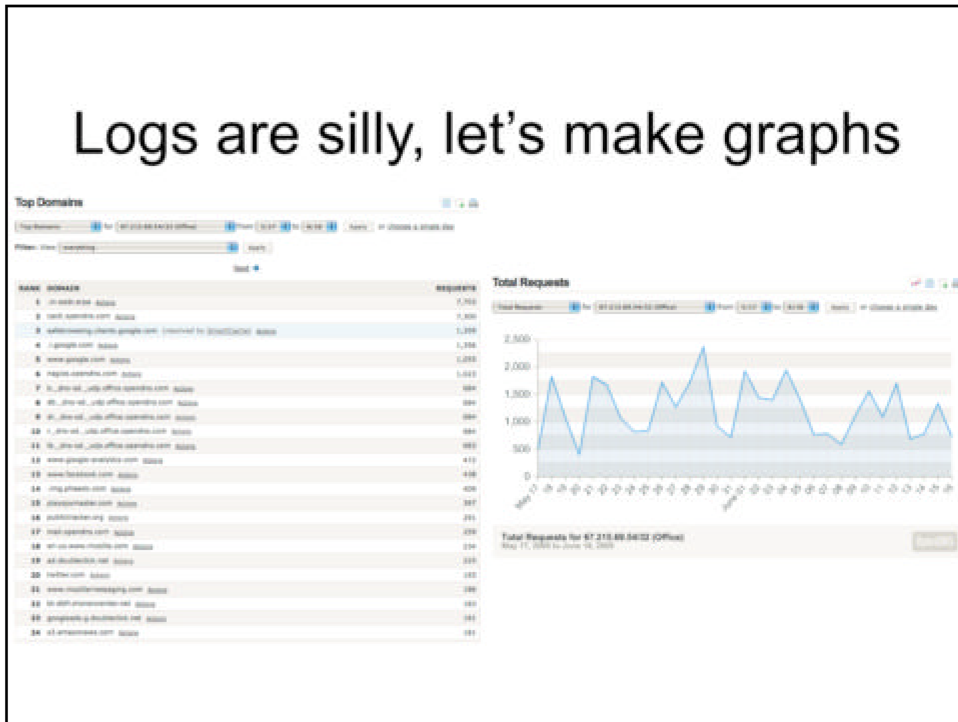
```
in.1.google.com. 1 0
i.co.jp. 1 0
. 1 0
@400000004a381ba80dd3ae34 q1 24.155.125.240 normal 1045953 my-iqquiz.com. 1 0
@400000004a381ba80dd3b604 q1 64.253.103.18 normal 788290 6.164.133.166.in-addr.arpa. 12 2
@400000004a381ba80dd3bdd4 q1 70.246.80.10 normal 0 googleads.g.doubleclick.net. 1 0
@400000004a381ba80dd3c5a4 q1 98.108.66.45 normal 0 _ldap._tcp.nj-bloomfield._sites.dc._msdcs.mrii.c
@400000004a381ba80dd41b94 q1 98.144.16.195 normal 0 js.casalemedia.com. 1 0
@400000004a381ba80dd42364 q1 68.165.29.60 normal 0 img-cdn.mediaplex.com. 1 0
@400000004a381ba80dd42b34 q1 12.233.75.219 normal 0 zsmseno.clnet.cz. 1 0
@400000004a381ba80dd43304 q1 174.37.58.88 normal 0 70.96.118.85.bl.spamcop.net. 16 0
@400000004a381ba80dd43ad4 q1 208.76.86.13 normal 519070 252.76.75.208.bl.spamcop.net. 1 3
@400000004a381ba80dd442a4 q1 201.138.19.196 normal 0 isatap.domain.local. 1 3
@400000004a381ba80dd465cc q1 24.192.98.53 normal 0 208.85.224.82.in-addr.arpa. 12 0
@400000004a381ba80dd46d9c q1 64.91.71.57 normal 0 liveupdate.symantecliveupdate.com. 1 0
@400000004a381ba80dd4756c q1 69.64.43.245 normal 558867 alt4.gmail-smtp-in.l.google.com. 1 0
@400000004a381ba80dd47d3c q1 69.64.43.245 normal 558867 alt4.gmail-smtp-in.l.google.com. 1 0
@400000004a381ba80dd4850c q1 72.10.191.11 normal 812477 iprepl.t.ctmail.com. 1 0
@400000004a381ba80dd49c7c q1 12.233.75.219 normal 0 zsmseno.clnet.cz. 1 0
@400000004a381ba80dd4a44c q1 69.157.60.79 normal 0 img-cdn.mediaplex.com. 1 0
@400000004a381ba80dd4ac1c q1 208.43.52.205 nxdomain 0 haghway.com.br. 1 0
@400000004a381ba80dd4b3ec q1 204.145.0.242 normal 488877 105.12.90.201.asetnhap5duax9a26124rda5g3gv
@400000004a381ba80dd4bbbc q1 206.246.157.1 normal 0 penninegas.co.uk. 15 2
@400000004a381ba80dd4c38c q1 69.21.243.131 normal 0 svn.atomicobject.com. 28 0
@400000004a381ba80dd4dafc q1 163.192.13.65 normal 894966 dns.hitachi-koki.co.jp. 1 0
@400000004a381ba80dd4e2cc q1 76.65.199.42 nxdomain 0 cs16.msg.dcn.yahoo.com. 1 0
@400000004a381ba80dd4ea9c q1 189.169.97.227 normal 0 impaktosoo.gateway.2wire.net. 1 3
@400000004a381ba80dd4f26c q1 69.64.43.245 normal 558867 gmail.com. 15 0
@400000004a381ba80dd4f654 q1 189.168.174.182 normal 0 wpad.2wire.net. 1 3
@400000004a381ba80dd4fe24 q1 69.64.43.245 normal 558867 alt3.gmail-smtp-in.l.google.com. 1 0
@400000004a381ba80dd51594 q1 189.133.170.67 normal 0 v13.lscache5.googlevideo.com. 1 0
@400000004a381ba80dd538bc q1 12.186.60.189 nxdomain 0 carolyn5.ktemca.com. 1 0
@400000004a381ba80dd5408c q1 72.249.148.132 normal 384918 mailin-04.mx.aol.com. 1 0
@400000004a381ba80dd5485c q1 76.65.199.42 nxdomain 0 csa.yahoo.com. 1 0
@400000004a381ba80dd5502c
@400000004a381ba80dd5541
@400000004a381ba80dd55be
```

Now: 14 billion DNS queries per day

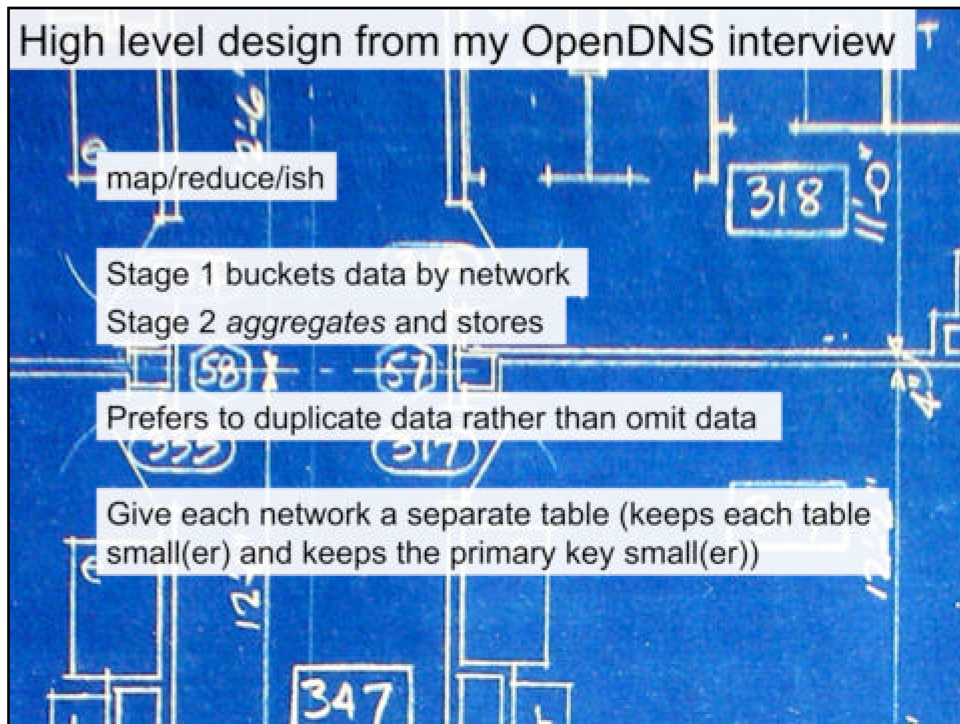
...a recursive DNS service that consumers choose to use over DNS provided by their ISP. We perform over 14 billion DNS queries on behalf of our users each day and aggregate most of them to give our users a better picture of their DNS use (and by proxy, Internet use).

When I started building the stats system, we were doing about 8 billion queries per day. When it soft launched, we were doing almost 10 billion queries per day. Just last week we crossed 14 billion in one day for the first time. That's 162,000 queries per second on average.

Our DNS servers all over the world produce log files that look like this: they're timestamped using DJB's tai64n format, which is a 64-bit timestamp plus a nanosecond component. This is free to us because we use multilog on our DNS servers. They contain a version, the client's IP address and network_id (the unique identifier we use to apply preferences), the QTYPE and RCODE of the query and a note about how our DNS server handled it.



But log files are too verbose. You can't see the forest for the trees. So we aggregate. We list your top domains with counters, graph requests per day, request types (A, MX, etc.) and unique IPs seen on your network, all for the last 30 days.



So with the input and output covered, let's talk about the architecture by way of talking about my interview at OpenDNS. I went in prepared to answer questions about BGP and DNS and was asked only one thing: how would I build the stats system?

Being a hardware designer by education, I like pipelines. This problem lends itself well to map/reduce because the data is by definition partitionable. The two combined and a pipeline that sort of performed map/reduce was born.

The goal of the pipeline is to create two different planes of horizontal scalability. Stage 1 would be communicating with our resolvers so this will need to scale horizontally with DNS queries. Stage 2 must scale horizontally with the number and size of our users. John Allspaw talks about Flickr's databases scaling with photos per user and we're in a similar situation. In the extreme case, a single massive user could have an entire Stage 2 node to himself, I just hope he's paying us for it.

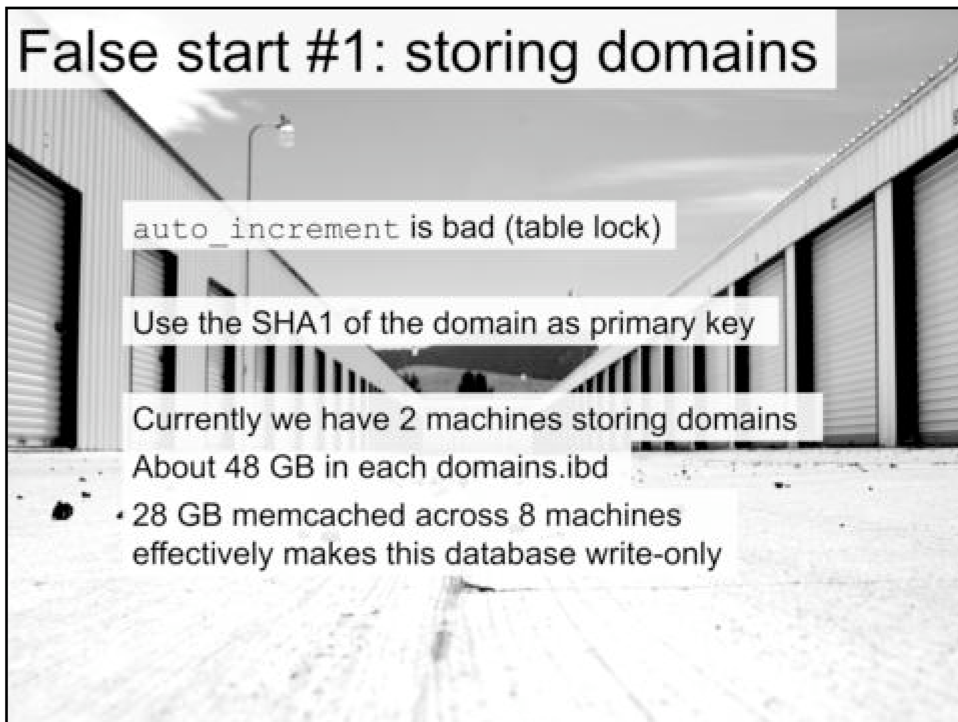
Because DNS already has a fuzzy mapping to actual web use, the counters don't have to be exactly correct. What's another 3 queries to Google? Where it does matter is at the bottom but even there we have some breathing room. When you're dealing with a single request to playboy.com, it is better to report two than zero, so I wanted to design a system that was robust against omission of data by allowing occasional duplication of data.

The final resting place for this data needed to scale horizontally along the same axis as Stage 2. MySQL is certainly the default hammer so we started with it. Giving each network its own table keeps table size and primary key length lower, makes migration between nodes easier and makes it possible to keep networks belonging to stats-hungry users in memory more of the time.

So I took the job. As with any project developed by children (that'd be me), there were false starts. I spent the first two months of my time at OpenDNS band-aiding our old stats system, learning the bottlenecks and evaluating technologies that might be a part of the new system.

The obvious choice is Hadoop, which is quite nice but is inherently a batch system that (at the time) did not meet the low-latency requirements for serving a website. More "scalable" key-value type databases lacked the ability to simulate GROUP BY, COUNT and SUM easily (though now there are compelling options available like Tokyo Cabinet's B+Tree database). I also evaluated using just Hbase on HDFS and unsurprisingly saw the same very high latency. We have a PostgreSQL fan in the office so I looked at that. I revisited BDB and the MemcacheDB network interface and probably some others. MySQL isn't necessarily the best solution but it's a known-known that I can build on with confidence.

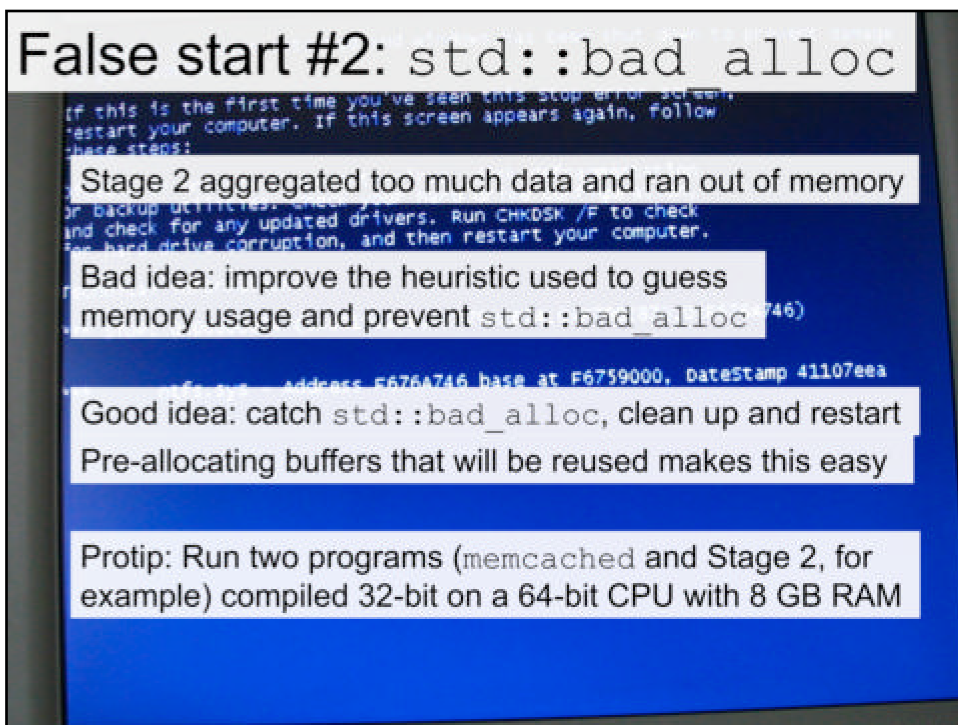
There were still some gotchas, though.



To show users every domain they visit, we have to store every domain they visit. I didn't want a big varchar in my primary key so the Domains Database was born to store a lookup table for domains. I do quite a bit of sanitization to avoid storing reverse DNS lookups for 4 billion IPv4 addresses or the hashes of every spam email sent to DNS-based spam blacklists.

So, whenever you're in a write-heavy situation, remember that auto_increment is always a table lock, even on an InnoDB table. This limits the concurrency of any application but can be solved. If you define your own primary key (say, a SHA1) and use INSERT IGNORE to ignore errors about inserting a duplicate primary key, you're golden. The domains database stores every domain we've counted, pointed to by its SHA1. Because the data determines the primary key, INSERT IGNORE is safe.

Domains on the Internet pretty well follow an 80/20 rule only it's closer to 90/10. The 878 million domains we have stored so far take up a total of 96 GB on disk. With 28 GB available to memcached we're able to cache about 1/3 of the domains. We see a very low (and nearly constant) eviction rate and a 98% hit rate.

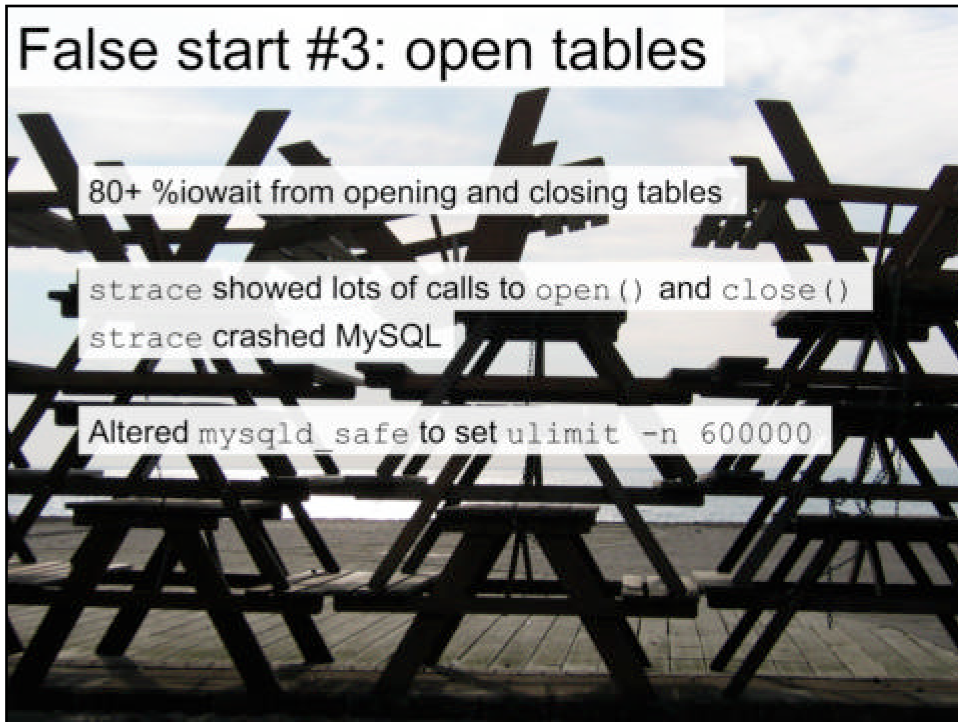


Stage 2 is all about aggregating data so that the flow of INSERTs is gentle enough for MySQL to handle without crying.

Whenever you aggregate things in memory, you're going to run out. My first feeble attempt at avoiding this fate was to track how much memory I was using and free more than I allocated. Not surprisingly, it's very difficult to know exactly how much memory you're using. getrusage() and mallinfo() do an OK job but it's hard to walk the thin line between crashing and not, without precise measurements.

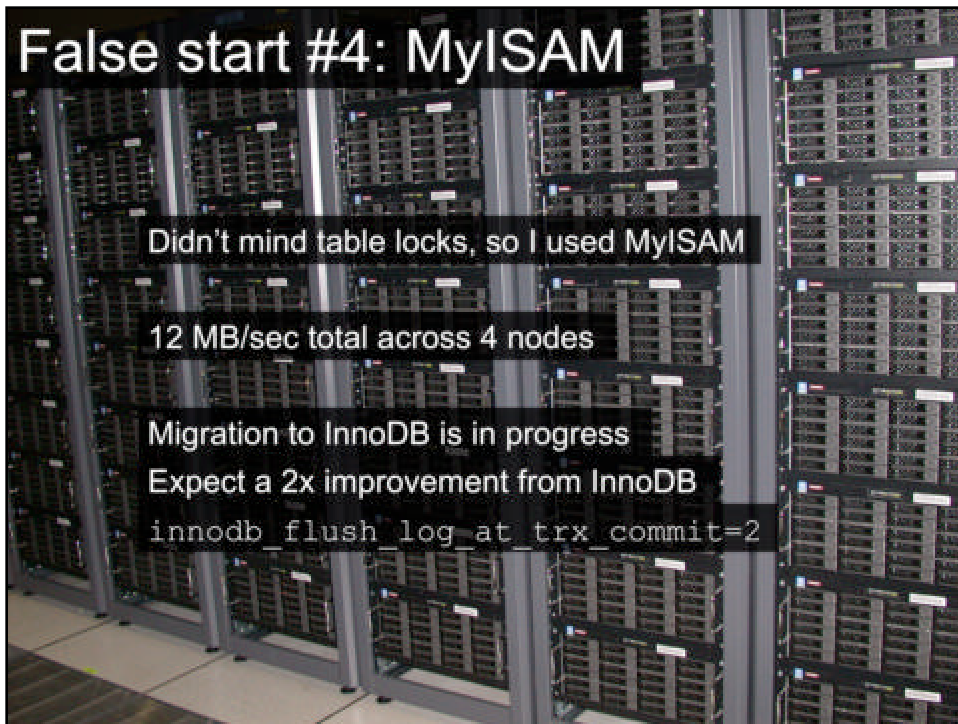
A much better idea is to react sanely when we do run out of memory. The C++ STL throws std::bad_alloc when it can't allocate more memory; malloc and friends return null pointers. In either case, I start shutting down carefully. I use supervise to manage these long running processes and when supervise sees the process end, a new one will be started immediately. The path from in-memory aggregation to disk does not involve allocating memory. Each thread has a set of buffers it uses to write SQL statements to disk in files that fit under max_packet_size. These buffers are recycled instead of freed, allowing shutdown to continue even when std::bad_alloc is being thrown.

In OpenDNS' setup, we have several machines with 64-bit CPUs and 8 GB RAM. Our ops guy likes running 32-bit Debian with a 64-bit kernel on these boxes and from this I discovered that you can avoid the OOM killer and instead get back std::bad_alloc by running 32-bit processes since these processes will run out of addressable space before the machine can ever run out of physical memory. I can give most of the other 4 GB to memcached and use basically every scrap of memory on these boxes.



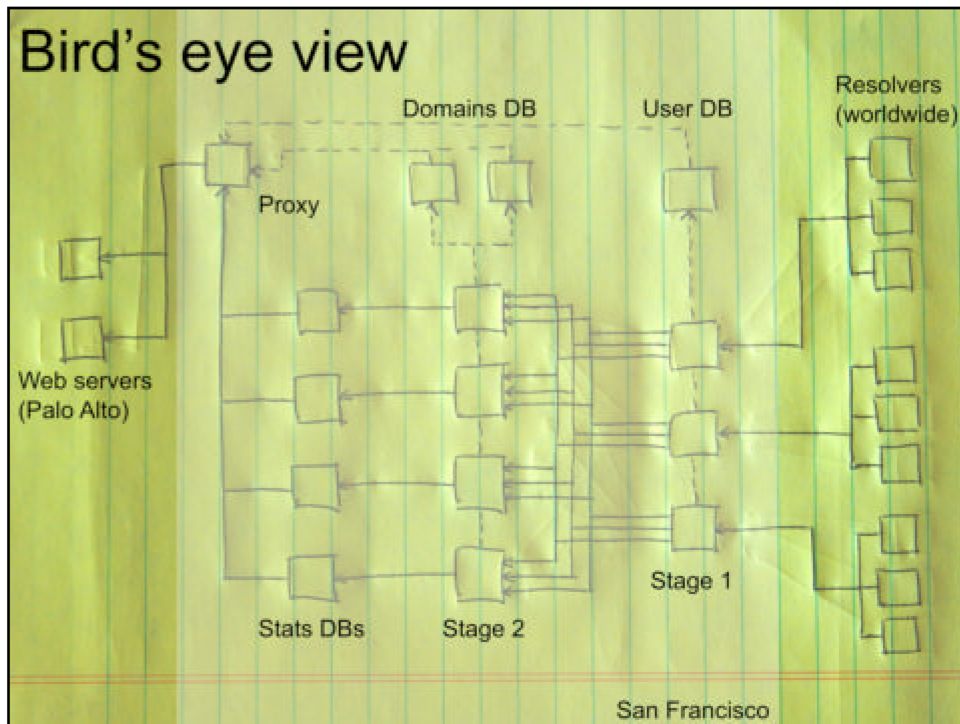
I mentioned all of the good parts of making a table for each network earlier. It makes migrations easier, keeps each table and primary key smaller and let's the guy always hitting refresh keep his stats in memory most of the time. There's a dark side, though, and it is the table cache.

I started with a tip from Automattic to manually call FLUSH TABLES to keep MySQL happy. This seemed promising at first and I can see how it works wonders for them but when writes dominate reads, this doesn't work so well. After observing the problem with strace, we recovered from observing the probem with strace and altered mysqld_safe to set a high ulimit on the number of open file descriptors, which lets us set a high table_cache, which in turn causes open_files_limit to set itself to twice the table_cache. With the high table_cache and open_files_limit, we can avoid most calls to open() and close(). In the event of a crash, many tables will be marked as crashed because they were marked as open but very few were actually mid-write at the time of the crash, which makes recovery tolerable. Thus far I've chosen explicitly not to do a recovery and instead fix tables that are actually crashed as the system finds them.



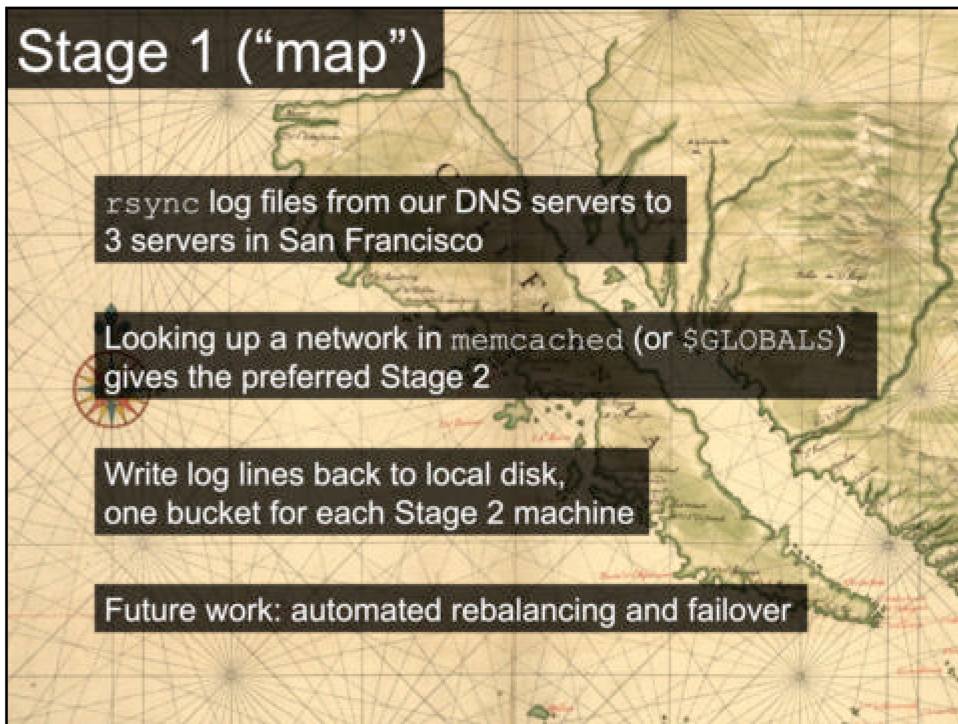
Even with the open tables issue mitigated, MyISAM is still bursting at the seams. This one is still being resolved so it's largely speculative. The MySQL schema is designed to balance row size, table length, and frequency of UPDATEs (as opposed to INSERTs). I've diverted a copy of production data to a dev server with similar specs running InnoDB and have seen much higher write throughput, in the neighborhood of a 2x improvement. I've heard warnings about InnoDB's performance breaking down with high numbers of tables so I'm treading lightly. Using innodb_flush_log_at_trx_commit=2 reduces the frequency of fsync() calls to once per second from once per transaction, so it's possible that we lose a little bit in the case of a full crash. However, the SQL statements are played through in chunks that take longer than 1 second to finish so in the event of a crash, the entire chunk will be replayed since, as you recall, we prefer duplicating data to omitting data.

After all of that, here is what has been running in production for the last 8 months.



At a very high level, here's the setup. Log lines are pulled from DNS servers around the world to Stage 1, running on 3 nodes in San Francisco. Stage 1, with the help of the User Database partitions data for Stage 2, which saves new domain names in the Domains Database and sends log lines as SQL to Stats Databases, which are accessed by a proxy before display on opendns.com, which runs in Palo Alto.

The website, proxy, Stage 1 and most of Stage 2 are written in PHP. The complicated part of Stage 2 is written in C++. The databases are all MySQL.



Each Stage 1 node is responsible for a subset of our DNS servers and use rsync with --remove-source-files to pull logs to local disk. As each of these files is processed, Stage 1 checks $GLOBALS, memcached and finally the User Database to know where to send the user's log data.

Right now, if one of these machines dies, I have to manually change the configuration and redeploy. Not optimal for sure but it's not difficult to add automatic rebalancing later. It's really logically a separate process that would check every so often for failures and react if one is found by splitting the dead node's workload amongst the living. Since log lines are queued on disk at every step, there's no urgency to recover instantly.

Stage 1 also does global request counting for system.opendns.com and multicasts each line into a network that can drink from the firehose for debugging or other real-time analysis.

Stage 2 data structures

Stats aggregation (pseudocode)

```
{
  "db1": {
    "123456": {
      "2009-06-17": {
        "last_updated": 1234567890,
        "file_ptrs": [0xDEADBEEF, 0xDECAFBAD],
        "topdomains": {
          "xkcd.com": [12,3,5,47,0,0,6,10,1,9,2,3,0,4,2,0,5,12,19,35,32,2,4,0],
        },
        "requesttypes": { "A": [ /* 24 hours */ ], "MX": [ /* 24 hours */ ] },
        "uniqueips": { "1.2.3.4": [ /* 24 hours */ ] }
      }
    }
  }
}
```

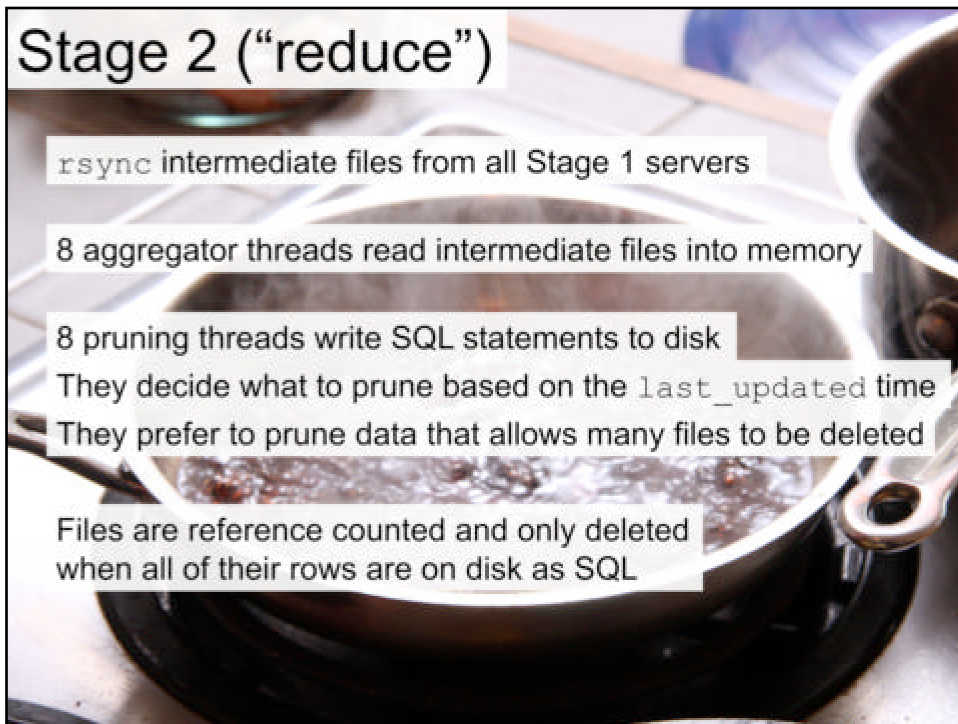
File reference counting (C++)

```
__gnu_cxx::hash_map<
  char *, // Filename
  std::pair<
    unsigned int, // Reference count
    pthread_t // Owning thread or NULL
  >,
  hash_ptr // Hashes a pointer as if it were an integer
>
```

Stage 2, our reduce stage, stores data in a big hash_map of hash_maps that are keyed by database, network and date. Data is pruned from this tree by day so this is where we store the last_updated timestamp and a hash_set of pointers to files. When a day is going to be pruned from the tree, these file pointers are used to decrement the reference count. The data structure at the bottom is how files are reference counted. The filenames are C-style strings pointed to by both the tree and the reference count. When a pruning thread notices a file with zero references and no owning thread, it is deleted.

Within each day in the tree, data for Top Domains, Request Types and Unique IPs is stored in three more hash_maps with each value pointing to a pointer to unsigned int, which itself points to an array of 24 unsigned ints, one for each hour in the day.

The database and network levels of the tree are protected by pessimistic locking using pthread_mutexes. To mitigate lock contention, there are actually 100 locks at within each database node, corresponding to network_id % 100. Once the lock on a set of networks has been acquired, the database is unlocked; once the lock on a network has been acquired, the lock on a set of networks is released. Pessimistic locking is easy to implement and by keeping the locks very fine-grained and very short lived, contention is not an issue.



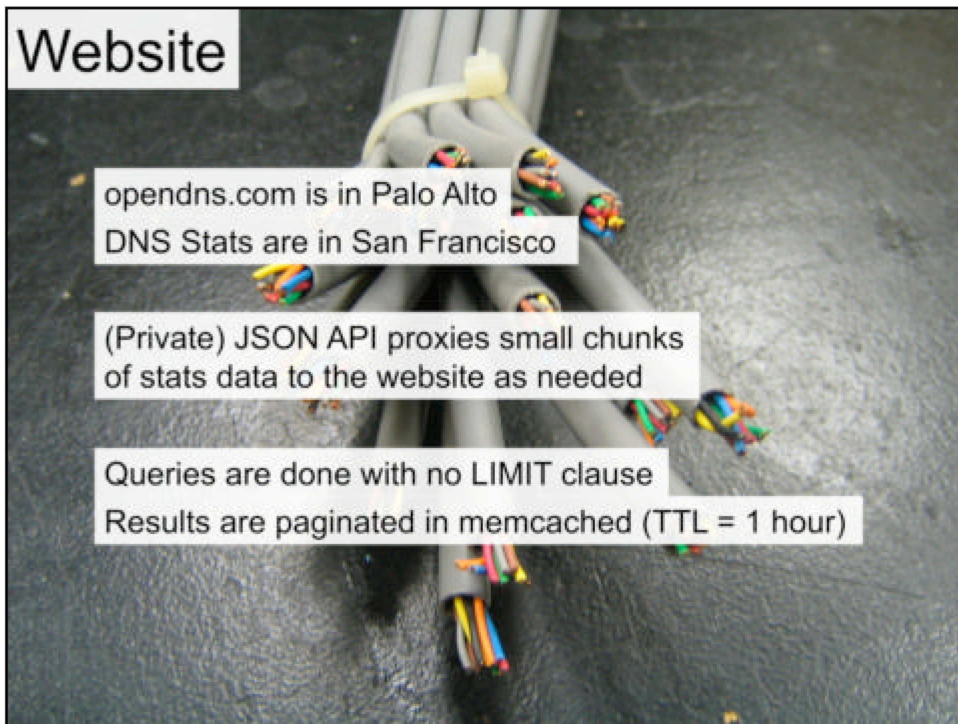
Stage 2 starts out just like Stage 1, by using rsync with --remove-source-files to fetch work from each Stage 1. The main Stage 2 program is written in C++ so it can do real multithreading.

8 aggregator threads repeatedly reserve a file on disk by renaming it and read it line-by-line into shared memory tree. The file is owned by this thread throughout the aggregation. These threads are all on the lookout for std::bad_alloc and start the shutdown process if they ever catch it. Before they can actually shutdown, though, they have to finish the file they're working on.

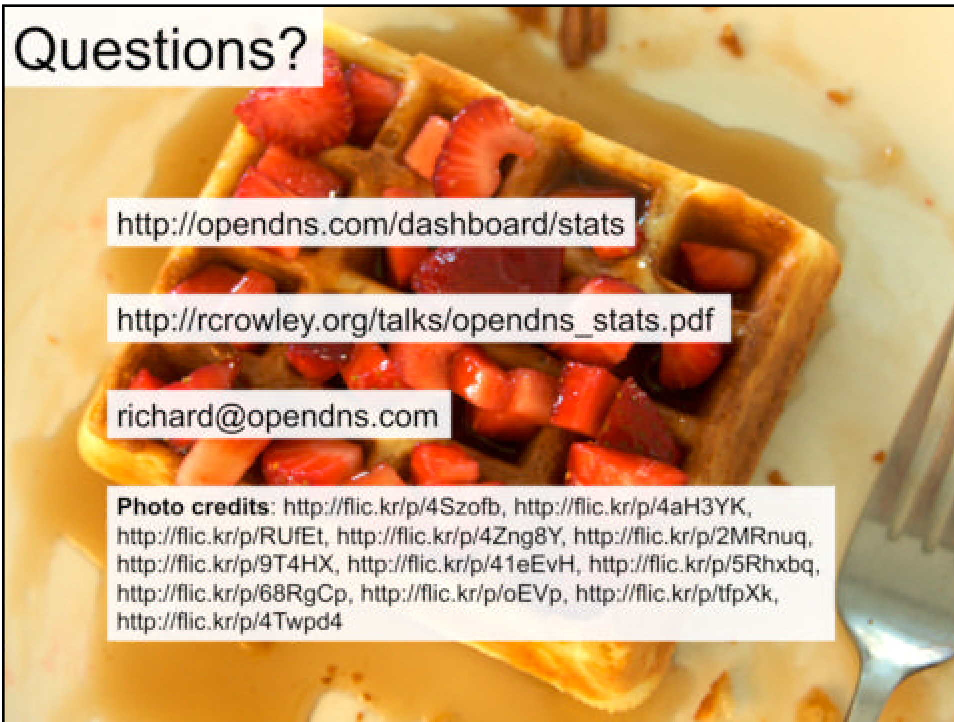
To match the aggregator threads there are 8 pruning threads which are constantly removing data from the tree and writing it to disk as SQL. These threads are not susceptible to std::bad_alloc since all the memory they use is in pre-allocated buffers. Usually, they're selective about what gets pruned but if std::bad_alloc has been caught, they prune everything as quickly as possible. The normal formula makes days that haven't been updated for a while or days with lots of data more likely to be pruned.

The production databases are now running stock MySQL 5.0.77 and all of the tables besides the domains table are MyISAM. I'm in the process of switching to a hybrid setup to begin the long process of moving to InnoDB. This means switching to MySQL 5.0.77-percona, which turns the normally infinite InnoDB data dictionary cache into an LRU, tunable with the innodb_dict_size_limit variable. The data dictionary cache stores field names, types and sizes, and information about indexes for opened tables, and will grow without bound in standard InnoDB.

Beyond the storage engine change, scaling involves adding more spindles, just like any other write-heavy database installation.



opendns.com is served from Palo Alto, so a proxy in San Francisco handles the database queries to reduce network congestion between data centers. That little spinner you see the first time you view your stats is the database being hit. Running a query with LIMIT means I'd probably just have to run it again soon with a different LIMIT, so I chose to pay the price once and paginate into memcached with an hour TTL. The result is that every page beyond the first is fast. Changes to the databases will help make the first page faster, too.



That's it, then. The time I spend these days on DNS Stats is all about gently moving the databases in the right direction, making queries from the website faster and keeping queues short.

Thank you, now let's have some questions.