# How I Learned to Stop Worrying and Love Using a Lot of Disk Space to Scale

Tuesday, May 27, 2008 at 10:04AM

Todd Hoff in BigTable, Database, GAE, Shard, Strategy, cloud

**Update 3**: ReadWriteWeb says Google App Engine Announces New Pricing Plans, APIs, Open Access. Pricing is specified but I'm not sure what to make of it yet. An image manipulation library is added (thus the need to pay for more CPU :-) and memcached support has been added. Memcached will help resolve the can't write for every read problem that pops up when keeping counters.
**Update 2**: onGWT.com threw a GAE load party and a lot of people came. The results at Load test : Google App Engine = 1, Community = 0. GAE handled a peak of 35 requests/second and a sustained 10 requests/second. Some think performance was good, others not so good. My GMT

watch broke and I was late to arrive. Maybe next time. Also added a few new design rules from the post.
**Update**: Added a few new rules gleaned from the GAE Meetup: Design By Explicit Cost Model and Puts are Precious.

How do you structure your database using a distributed hash table like BigTable? The answer isn't what you might expect. If you were thinking of translating relational models directly to BigTable then think again. The best way to implement joins with BigTable is: **don't**. You--pause for dramatic effect--**duplicate data instead of normalize it**. **shudder**

Flickr anticipated this design in their architecture when they chose to duplicate comments in both the commentor and the commentee user shards rather than create a separate comment relation. I don't know how that decision was made, but it must have gone against every fiber in their relational bones...

But Flickr's reasoning was genius. To scale you need to partition. User data must spread across the shards. So where do comments belong in a scalable architecture?

From one world view comments logically belong to a relation binding comments and users together. But if your unit of scalability is the user shard there is no separate relation space. So you go against all your training and decide to duplicate the comments. Nerd heroism at its best. Let inductive rules derived from observation guide you rather than deductions from arbitrarily chosen first principles. Very Enlightenment era thinking. Voltaire would be proud.

In a relational world duplication is removed in order to prevent update anomalies. Error prevention is the driving force in relational modeling. Normalization is a kind of ethical system for data. What happens, for

example, if a comment changes? Both copies of the comment must be updated. That leads to errors because who can remember where all the data is stored? A severe ethical violation may happen. Go directly to relational jail :-)

BigTable data ethics are more Mardi Gras than dinner with the in-laws. Data just wants to have fun. BigTable won't stop you from hurting yourself. And to get the best results you may have to engage in some conventionally risky behaviors. But if those are the glass bead necklaces you have to give for a peak at scalability, why not take a walk on the wild side?

For a more modern post-relational discussion of data ethics I'm using as my primary source a thread of conversations from JA Robson, Ben the Indefatigable, Michael Brunton-Spall, and especially Brett Morgan. According to our new Voltaire, Locke, Bacon, and Newton, here's what it takes to act ethically in a BigTable world:

- **Don't bother with BigTable unless your goal is to create a web site that scales to millions of users.** The techniques for building scalable read-mostly web applications are difficult and require a radical mindset change. Standard relational techniques work very well until you scale to huge numbers of users. It is at that point you need to break the rules and do something counter-intuitively different. More of the same will not work. If you don't plan to get to that point it may not be worth the effort to change. BigTable is targeted at building web applications, It's nature makes it a poor match for OLAP, data warehousing, data mining, and other applications performing complex data manipulations.

- **Assume slower random data access rather than fast sequential access.** Every get of an entity could be from a different disk block on a different machine in a cluster. Calculating, for example, the average over

a column in SQL can be efficient because data is stored together on disk. In BigTable data can be anywhere so iterating over every value in a column is expensive. Each read is potentially a random block from anywhere which means the average retrieval time can be relatively high. The implication is to use BigTable you must adopt some unfamiliar and unintuitive strategies in order to deal with such a very different performance profile. Using relational database we are used to writing applications against fast highly performant databases. With BigTable you have to become familiar with the rules for developing against a slower but more scalable database. Neither approach is better for all purposes, but BigTable has the edge for high scalability.

- **Group data for concurrent reads.** Given the high cost of reading data from BigTable your application will not scale if every page requires a large number of reads. The solution: **denormalize**. Store data in the same entity based on what data needs to be read concurrently. Relational modeling groups data together based on the “minimize problems” rule. BigTable’s new rule is “maximize concurrent reads” which implies denormalization. Store entities so they can be read in one access rather than performing a join requiring multiple reads. Instead of storing attributes in separate entities in order to remove duplication, duplicate the attributes and store them where they need to be used. Following this rule minimizes the number of reads required to return an entity.

- **Disk and CPU are cheap so stop worrying about them and scale.** A criticism of denormalization is storing duplicate data wastes disk space. Google’s architecture trades disk space for better performance. Disk is (relatively) cheap, so don’t fight it. On the CPU front a data center’s worth of CPU is at your service. As long as you structure your application in the way GAE forces you to, your application can scale as large as it needs to simply by running on more machines. All scalability bottlenecks have been removed.

- **Structure data around how it will be used.** Trade SQL sets for application based entities. Queries are slow so the closer data is to the format it is to be used the faster pages will render. It's like the database model becomes the model previously used at the caching layer. Complete entities tend to be cached, not low level detail rows. That's what BigTable models should look like because that's how concurrent reads are maximized. This isn't the same as an object oriented database because the behavior is provided by applications, behavior is not bound to the entity so multiple applications can read the same entities yet implement very different behaviors.

- **Compute attributes at write time.** Since looping over large columns of data is inefficient with BigTable the idea is to calculate values at write time instead of read time. For example, instead of calculating an average by reading an entire column at read time, track the total number and the total value at write time so the average can be calculated with one read on page display. Programmer effort is made up front at write time to minimize the work needed at read time. Preventing applications from iterating over huge data is key for making applications scale. Given the limitations of GAE transactions and quotas, GAE may not be appropriate for business applications that need exact summary statistics. Warning: if the summary stat is written on every read request then this approach will not scale as writes don't scale.

- **Create large entities with optional fields.** Normalization creates lots of small entities. Instead, create larger entities with optional parts so you can do one read and then determine what's present at run time. This shifts work from the database to the CPU while minimizing the number joins.

- **Define schemas in models.** Denormalization requires user developed code to properly keep data consistent across multiple entities. The database won't do it for you anymore. Schemas are really defined in code

because it’s only code that can track all the relationships and maintain correctness. All database access must go through the models or otherwise the much feared inconsistency problems will result.

- **Hide updates using Ajax.** Updates are slow so big bang updates of many entities will appear slow to users . Instead, use Ajax to update the database in little increments. As a user enters form data update the database so the update cost is amortized over many calls rather than one big call at the end. The result is a good user experience and a more scalable app.

- **Puts are Precious.** Updating entities in large batches, say even 200 at a time, isn't
part of the BigTable model. Entity attributes are automatically and synchronously indexed on writes. Indexing is an expensive operation that accumulates a lot of CPU time so the number updates that can be performed in one query is quite limited. The work around is to perform updates in smaller batches driven by an external CPU. Even when GAE provides the ability run batches within GAE the programming model for writes needs to be accounted for in a design.

- **Design By Explicit Cost Model.** If you are going to be charged for an operation GAE wants you to explicitly ask for it. This is why some automatic navigation between objects isn't provided because that will force an explicit query to be written. Writing an explicit query is a sort of EULA for being charged. Click OK in the form of a query and you've indicated that you are prepared to pay for a database operation.

- **Place a many-to-many relation in the entity with the fewest number of elements.** One way to create a many-to-many relationship is to have a list property that contains keys to the other related entities. A Company entity, for example, could contain a list of keys to Contact entities or a Contact entity could contain a list of keys to Company

entities. Since it's likely a Contact is associated with fewer Companies the list should be contained in the Contact. The reasoning is maintaining large lists is relatively inefficient so you want to minimize the number of items in a list as much as possible.

- **Avoid unbounded queries.** Large queries don't scale. Consider showing only the most recent 10 or so values from an attribute.

- **Avoid contention on datastore entities.** If every request to your app reads or writes a particular entity, latency will increase as your traffic goes up because reads and writes on a given entity are sequential. One example construct you should avoid at all costs is the global counter, i.e. an entity that keeps track of a count and is updated or read on every request.

- **Avoid large entity groups.** Any two entities that share a common ancestor belong to the same entity group. All writes to an entity group are sequential, so large entity groups can bog down popular apps quickly if there are a lot of writes to that group. Instead, use small, localized groups in your design.

- **Shard counters.** Increment one of N counters and sum those N counters on the read side. This avoids the dreaded write bottleneck. See Efficient Global Counters by App Engine Fan for more details.

An excellent example showing some of these principles in action can be found in this GQL thread.

Take this nicely normalized schema:

Customer:
 - Name
 - Country
Product:

- Code
- Name
- Description

Purchases:
- Reference to Product Entity
- Reference to Customer Entity
- Date of order

Anyone from a relational background would look at this schema and give it a big thumbs up. With a little effort we can imagine the original physical purchase order that has now been normalized into three different tables.

To recreate the original purchase order a join on purchases, produce and customer is needed. Read speed is not optimized, safety is optimized.

Here’s what the same schema looks like optimized for reading:

Purchase:
- Customer Name
- Customer Country
- Product Code
- Product Name
- Purchase Order Number
- Date Of Order

The three original tables have been folded into one entity. Now a purchase order can be read in one get operation. No join necessary. Notice how the entity looks more like an original purchase order. It is also what would probably be cached and is what our model would probably look like.

But what if you want to update a product name or a customer name? Those attributes are duplicated in all entities. Here's where the protection offered by the relational model comes in. Only one entity needs updating in a normalized model.

In BigTable you have to remember everywhere a customer name and product name and change every instance to new values. It's not a simple, safe, or reliable approach. But it does optimize for read speed and scalability.

For an application with a high proportion of updates to reads this approach wouldn't make sense. But on the web reads usually dominate. How often do you really change a customer name or a product name? Seldom. How often do you read them? All the time.

Designing to scale for reads and taking the pain on writes takes some getting used to. It's a massive change to standard relational tactics. But this is what it takes to scale web applications, even if it feels a little strange at first.

## Related Articles

- ER-Modeling with Google App Engine (updated)
- Tips on writing scalable apps

---