Google's Colossus Makes Search Real-time by Dumping MapReduce

Saturday, September 11, 2010 at 5:50AM

Todd Hoff in google

As the Kings of scaling, when Google changes its search infrastructure over to do something completely different, it's news. In Google search index splits with MapReduce, an exclusive interview by Cade Metz with Eisar Lipkovitz, a senior director of engineering at Google, we learn a bit more of the secret scaling sauce behind Google Instant, Google's new faster, real-time search system.

The challenge for Google has been how to support a real-time world when the core of their search technology, the famous MapReduce, is batch oriented. Simple, they got rid of MapReduce. At least they got rid of MapReduce as the backbone for calculating search indexes. MapReduce still excels as a general query mechanism against masses of data, but real-time search requires a very specialized tool, and Google built one. Internally the successor to Google's famed Google File System, was code named Colossus.

Details are slowly coming out about their new goals and approach:

- Goal is to update the search index continuously, within seconds of content changing, without rebuilding the entire index from scratch using MapReduce.
- The new system is a "database-driven, Big Table–variety indexing system."

- When a page is crawled, changes are made incrementally to the web map stored in BigTable, using something like database triggers.
- A compute framework executes code on top of BigTable.

The use of triggers is interesting because triggers are largely ignored in production systems. In a relational database triggers are integrity checks that are executed on a record every time a record is written. The idea is that if the data is checked for problems before it is written into the database, then your data will always be correct and the database can be a pure source of validated facts. For example, an account balance could be checked for a negative number. If the balance was negative then the write would fail, the transaction aborted, and the database would maintain a correct state.

In practice there are many problems with triggers:

- **Destroys performance**. Since triggers happen on every write they slow down the write path to the extent that database performance is often killed. Triggers also take record locks which makes contention even worse.
- **Checking is distributed**. Not all integrity checks can be made from data inside the database so checks that are database oriented are put in the triggers and checks, that say access a 3rd party system, are kept in application code. So now we have checks in multiple places, which leads to the update anomalies in code that ironically, relational databases are meant to prevent in data.
- **Checking logic is duplicated**. Checking in the database is often too late. A user needs to know immediately when they are doing something wrong, they can't wait until a transaction is committed to the database. So what ends up happening is checking code is duplicated, which again leads to update anomalies.
- **Not scalable**. The database CPU becomes dedicated to running

triggers instead of "real" database operations, which slows down the entire database.

- **Application logic moves into triggers**. Application code tends to move into triggers over time since triggers happen on writes (changes), in a single common place, they are a very natural place to do all the things that need to be done to data. For example, it's common to emit events about data changes and perform other change sensitive operations. It's easy to imagine building secondary indexes from triggers and synchronizing to backup systems and other 3rd party systems. This makes the performance problems even worse. Instead of application code being executed in parallel on horizontally scalable nodes, it's all centralized on the database server and the database becomes the bottleneck.

So, triggers tend not to be used in OLTP scenarios. Applications contain the same logic and it's up to release policies and testing to make sure nothing falls through the cracks.

This isn't to say you don't want to put logic in triggers, you do. Triggers are ideal in an event oriented world because the database is the one central place where changes are recorded. The key is to make triggers efficient.

It sounds like Google may have made a specialized database where triggers are efficient by design. We know hundreds of thousands of pages are being crawled simultaneously. When the pages are written to the database that's the perfect opportunity perform calculations and updates. The compute framework would make it efficient to perform operations in parallel on pages as they come in so that processing isn't completely centralized on each node.

One can imagine that the in-memory data structures that existed on the MapReduce nodes have been extracted in some form and reified within

BigTable. What we have is a sort of Internet DOM, analogous to the browser DOMs that have made it possible to have such incredibly powerful browser UIs, but for the web. I imagine programming the web for Google has become something like programming a browser DOM. I could be completely wrong of course, but I explored this idea a while ago in All the world is a DOM.

There's an interesting quote at the end of the article: "*We're in business of making searches useful, we're not in the business of selling infrastructure*."

These could actually be the same goal. It's a bit silly to have everyone in the world crawl the same pages and build up yet another representation of the Web. Imagine if the Web was represented by an Internet DOM that you could program, that you could write to, and that you could add code to just like Javascript is added to the browser DOM. Insert a node into the Internet DOM, like a div in an HTML page, and it would just show up on the Web.

This Internet DOM could be shared and all that backbone bandwidth and web site CPU reclaimed for something more useful. Google is pretty open, but they may not be that open.
