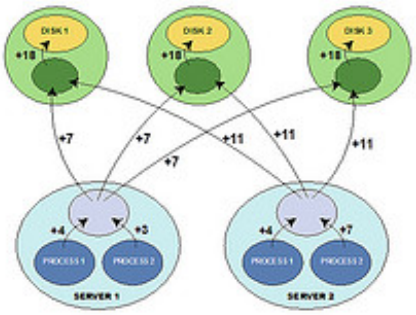
The Anatomy of Search Technology: blekko's NoSQL database

Wednesday, April 25, 2012 at 9:15AM

Todd Hoff in Example

This is a guest post (part 2, part 3) by Greg Lindahl, CTO of blekko, the spam free search engine that had over 3.5 million unique visitors in March. Greg Lindahl was Founder and Distinguished Engineer at PathScale, at which he was the architect of the InfiniPath low-latency InfiniBand HCA, used to build tightly-coupled supercomputing clusters.



Imagine that you're crazy enough to think about building a search engine. It's a huge task: the minimum index size needed to answer most queries is a few billion webpages. Crawling and indexing a few billion webpages requires a cluster with several petabytes of usable disk -- that's several thousand 1 terabyte disks -- and produces an index that's about 100 terabytes in size.

Serving query results quickly involves having most of the index in RAM or on solid state (flash) disk. If you can buy a server with 100 gigabytes of RAM for about $3,000, that's 1,000 servers at a capital cost of $3 million, plus about $1 million per year of server co-location cost (power/cooling/space.) The SSD alternative requires fewer servers, but serves a lot fewer queries per second, because SSDs are much slower than RAM.

You might think that Amazon's AWS cloud would be a great way to reduce the cost of starting a search engine. It isn't, for 4 main reasons:

1. Crawling and indexing requires a lot of resources all the time; you can't save money by only renting most of the servers some of the time.
2. Amazon currently doesn't rent servers with SSDs. Putting the index into RAM on Amazon is very expensive, and only makes sense for a search engine with several % market share.
3. Amazon only rents a limited number of ratios of disk i/o to ram size to core count. It turns out that we need a lot of disk i/o relative to everything else, which makes Amazon less cost effective.
4. At some cluster size, a startup has enough economy of scale to beat Amazon's cost+profit margin. At launch (November, 2010) blekko had 700 servers, and we currently have 1,500. That's well beyond the break-even point.

Software

That's just the hardware: we also needed a software system to manage storage and access to web crawl and index data across our cluster. One of our goals was to also design a storage architecture

that would give our programmers a productivity advantage when working with web-sized datasets in a search engine context.

Our goals were:

- A system that stores data in something that looks like database tables
- Real-time (query serving) and batch (crawling and indexing) processing in the same cluster
- The ability to add direct support for inverted indexes that nicely fit in with the rest of the system
- Great programmer productivity, including:
  - Seamless integration of the datastore and data structures of the main implementation language
  - Resistance to common database and cluster database issues, such as hot spots and dead/live locks
  - A datastore that implements the usual database operations, like BASE and CRUD, and also efficiently implements and assists parallel programming constructs, such as work queues.
  - Support for quick turn-around time for initial batch job results, allowing programmers to quickly find out if their batch job is doing the wrong thing
- Scalability to thousands of disk devices:
  - Disk failures are the most common and annoying problem with scaling clusters to large sizes
  - Gradual degradation in the face of failures
  - Ideally, a failure of 1% of the cluster should only reduce the amount of processing power and storage by 1%. As an example, poor use of RAID on servers can reduce throughput by much more than 1% during the rebuild of a RAID volume on a node. Also, disk failures are so common in a large cluster that we wished to not have any human intervention needed until we were ready to repair many nodes at once, a week or a month later.
- For availability reasons, we wanted to implement as many subsystems as possible using swarm algorithms, instead of electing masters using Paxos. https://en.wikipedia.org/wiki/Paxos_algorithm

Combinators

In a conventional database system, updates to the database are done in transactions, in which the program locks one or more rows of the database, makes some changes, and then commits or aborts the entire transaction. Most NoSQL databases limit transactions to locking & modifying a single row, and limit the types of computation that can be done within a transaction.

In blekko's datastore, we heavily rely on a construct called combinators to do processing at the database cell level. A combinator is an atomic operation on a cell of a database that is associative and preferably commutative. "Add(n)" is an example of a simple combinator; it adds n to whatever

number is in the cell. Crucially, it does not return the sum to the caller; it merely initiates the addition.

If a bunch of database nodes would like to add 1 to the same cell in the database, without learning what the end value of the cell is, it is possible to combine these operations in a hierarchy within the cluster, such that the final disk operation is a single operation that adds the sum of the initial increments.

The fact that addition is associative and commutative means that we will (eventually) get the same answer in all 3 replicas of this cell. The hierarchy of combinations means that the total number of transactions is dramatically reduced compared to a naive implementation, where every process talks directly to the 3 replicas of the cell, and every addition operation results in 3 immediate transactions.

The Logcount Combinator

Search engines frequently need to count unique items in a set. Examples include counting the number of inlinks to a website, the number of unique geographic areas linking to a website, and the number of unique Class-C IP networks linking to a website, and so on. While we could always run a MapReduce job to get an exact count of these items, we would like to know these counts at any point in time. Keeping a perfect count would require keeping a lot of data, so we invented an **approximate method**, which can count a up to a billion things with accuracy of +- 50% in only 16 bytes.

This combinator, which we call logcount, is implemented in a way which is commutative and associative -- you can AND the bits of two of these together to combine their counts. It also has the property that if you count any string multiple times, it only changes the answer at most once. This last property means that it is re-runnable, i.e. if the same transaction is made twice in the database, the answer doesn't change.

The TopN Combinator

Another common search engine operation is remembering the most important N items in a set. This is used in situations where we don't wish to remember all the items in a set, to keep the size down. For example, we might wish to frequently access the anchor text of the most important 2,500 incoming URLs to every URL we've ever crawled.

The TopN combinator can represent the top N URLs in a finite-sized array that fits into a single cell of the database:

Rank	Key	Ride-along data
URL Rank	URL	anchortext
157	http://sitea.com/	Great site
102	http://siteb.com/	Click here
98	http://sitec.com/mmf	Make money fast

The TopN combinator can be updated incrementally, as we crawl new webpages, and these updates are inexpensive. It can be read back in a single disk operation, without needing indexing or sorting or reading of any data about URLs not in the top N. Thanks to the ranks used to decide which of the urls fit into the N most important, it is commutative and associative. And it's re-runnable.

Additional tricks are possible if we use time as the rank. That allows us to efficiently remember the most recent N things, or the earliest N things.

So far we've used combinators as single cells in our database tables. It's also possible to **use them in variables** within our programs; their implementation stores them as strings, and reading the value out requires calling a "finalize" function. This function, for example, turns the 16 bytes of bitfields in a logcount into an integer approximate number of items.

Meta-Combinators: The Hash Combinator

If a cell in the database contains a hash of (key,value) pairs, the hash meta-combinator can be used to atomically update only some of the (key,value) pairs, leaving the rest unchanged. This gives us considerable freedom to make the columns in the database the ones that make sense to the programmer, instead of having to promote extra things to be columns in order to be able to change them atomically.

Taking The Reduce Out Of Map/Reduce

Since we represent our database tables with combinators, why not use the combinators to shuffle and reduce the output of our MapReduce jobs? Then we can write MapJobs that iterate over a table in the database, and write their output back into the database using combinators. Let's wordcount the web:

```
foreach row in "/crawl/url"
  foreach word in html column
    comb_add( "/wordcount", word, 1 )
```

In this pseudo-code, we iterate over all of the html documents in the table /crawl/url, and for each word found, we increment the appropriate row in the table "/wordcount". The programmer no longer needs to think about what **intermediate form** the shuffle/reduce step should take as input, or specify

a reduce function.

A second feature of this method of wordcount is that the above function can **also be used in a streaming** context, to add the wordcounts of newly crawled documents to the existing counts in the table "/wordcount". The usual way of representing MapReduce computations can't use the same code in a streaming computation.

A third feature is that the first output from this MapJob appears in the /wordcount table a few minutes after the MapJob starts. Most MapReduce systems do not begin the shuffle and reduction until after each shard finishes. Having a **quick, partial answer** allows programmers to find bugs in their MapJobs more quickly.

Did We Implement All Of Our Requirements?

Now that we have nearly 3 years of operational experience with this home-grown datastore, it's nice to look back and see that we mostly met all of the requirements we set out in advance. We've only discussed part of the datastore's features in this initial article, but the ones we've discussed here are well-covered. Our datastore:

- Looks like a tabular database
- Supports real-time and batch processing in the same cluster
- Supports high programmer productivity

In the next installment of this series, we'll take a more detailed look at web crawling, and using combinators to implement a crawler.
