Drop ACID and Think About Data

Tuesday, May 5, 2009 at 5:08AM

Todd Hoff in key-value store

The abstract for the talk given by Bob Ippolito, co-founder and CTO of Mochi Media, Inc:

> *Building large systems on top of a traditional single-master RDBMS data storage layer is no longer good enough. This talk explores the landscape of new technologies available today to augment your data layer to improve performance and reliability. Is your application a good fit for caches, bloom filters, bitmap indexes, column stores, distributed key/value stores, or document databases? Learn how they work (in theory and practice) and decide for yourself.*

Bob does an excellent job highlighting different products and the key concepts to understand when pondering the wide variety of new database offerings. It's unlikely you'll be able to say oh, this is the database for me after watching the presentation, but you will be much better informed on your options. And I imagine slightly confused as to what to do :-)

An interesting observation in the talk is that the more robust products are internal to large companies like Amazon and Google or are commercial. A lot of the open source products aren't yet considered ready for prime-time and Bob encourages developers to join a project and make patches rather than start yet another half finished key-value store clone. From my monitoring of the interwebs this does seem to be happening and existing products are starting to mature.

From all the choices discussed the column database Vertica seems closest to Bob's heart and it's the product they use. It supports clustering, column storage, compression, bitmapped indexes, bloom filters, grids, and lots of other useful features. And most importantly: it works, which is always a plus :-)

Here's a summary of some of the points talked about in the presentation: Video Presentation of Drop ACID and Think About Data.

ACID

- The claim to fame for relational databases is they make the ACID promise:
  - Atomicity - a transaction is all or nothing
  - Consistency - only valid data is written to the database
  - Isolation - pretend all transactions are happening serially and the data is correct
  - Durability - what you write is what you get
- The problem with ACID is that it gives you too much, it trips you up when you are trying to scale a system across multiple nodes.
- Down time is unacceptable. So your system needs to be reliable. Reliability requires multiple nodes to handle machine failures.
- To make a scalable systems that can handle lots and lots of reads and writes you need many more nodes.
- Once you try to scale ACID across many machines you hit problems with network failures and delays. The algorithms don't work in a distributed environment at any acceptable speed.

CAP

If you can't have all of the ACID guarantees it turns out you can have two of the following three characteristics:

- Consistency - your data is correct all the time. What you write is what you read.
- Availability - you can read and write and write your data all the time
- Partition Tolerance - if one or more nodes fails the system still works and becomes consistent when the system comes on-line.

BASE

The types of large systems based on CAP aren't ACID they are BASE (har har):

- Basically Available - system seems to work all the time
- Soft State - it doesn't have to be consistent all the time
- Eventually Consistent - becomes consistent at some later time

Everyone who builds big applications builds them on CAP and BASE: Google, Yahoo, Facebook, Amazon, eBay, etc.

Google's BigTable

- Google BigTable - manages data across many nodes.
- Paxos (Chubby) - distributed transaction algorithm that manages locks across systems.
- BigTable Characteristics:
  - stores data in tablets using GFS, a distributed file system.
  - compression - great gains in throughput, can store more, reduces IO bottleneck because you have to store less so you have to talk to the disks less so performance improves.
  - single master - one node knows everything about all the other node (backed up and cached).
  - hybrid between row and column database

    - row database - store objects together
    - column database - store attributes of objects together. Makes sequential retrieval very fast, allows very efficient compression, reduces disks seeks and random IO.
    - versioning
    - bloom filters - allows data to be distributed across a bunch of nodes. It's a calculation on data that probabilistically maps the data to the nodes it can be found on.
    - eventually consistent - append only system using a row time stamp. When a client queries they get several versions and the client is in charge of picking the most recent.
- Pros:
    - Compression is awesome. He really thinks compression is an important attribute of system.
    - Clients are probably simple.
    - Integrates with map-reduce.
- Cons:
    - Proprietary to Google - You can't use it on your system.
    - Single-master - could be a downside but not sure.

Amazon's Dynamo

- A giant distributed hash table, called a key-value store.
- Uses consistent hashing to distribute data to one or more nodes for redundancy and performance.
- Consistent hashing - a ring of nodes and hash function picks which node(s) to store data
- Consistency between nodes is based on vector clocks and read repair.
- Vector clocks - time stamp on every row for every node that has written to it.
- Read repair - When a client does a read and the nodes disagree on

the data it's up to the client to select the correct data and tell the nodes the new correct state.

- Pros:
  - No Master - eliminates single point of failure.
  - Highly Available for Write - This is the partition failure aspect of CAP. You can write to many nodes at once so depending on the number of replicas (which is configurable) maintained you should always be able to write somewhere. So users will never see a write failure.
  - Relatively simple which is why we see so many clones.
- Cons:
  - Proprietary. Clients have to be smart to handle read-repair, rebalancing a cluster, hashing, etc. Client proxies can handle these responsibilities but that adds another hop.
  - No compression which doesn't reduce IO.
  - Not suitable for column-like workloads, it's just a key-value store, so it's not optimized for analytics. Aggregate queries, for example, aren't in it's wheel house.

Facebook's Cassandra

- Peer-to-peer so no master like Dynamo
- Storage model more like BigTable
- Pros:
  - Open source. You can use it.
  - Incremental scalable - as data grows you can add more nodes to storage mesh.
  - Minimal administration - because it's incremental you don't have to do a lot of up front planning for migration.
- Cons:
  - Not polished yet. It was built for in-box searching so may not be work well for other use cases.

    - No compression yet.

Distributed Database Musings

- Distributed databases are the new web framework.
- None are awesome yet. No obvious winners.
- There are many clones with partial features implemented.
    - For example Project Voldemort doesn't have rebalancing, no garbage collection.
- Pick one and start submitting patches. Don't start another half-baked clone.

Simple Key-Value Store

- Some people are using simple key-value stores to replace relational database.
- A key (array of bytes) maps using a hash to a value (a BLOB). It's like an associative array.
- They are really fast and simple.

Memcached Key-Value Stores

- Is a key-value store that people use as a cache.
- No persistence
- RAM only
- LRU so it throws data away on purpose when there's too much data
- Lightening fast
- Everyone uses it so well supported.
- A good first strategy in removing load from the database.
- Dealing with mutable data in a cache is really hard. Adding cache to an ACID system is something you'll probably get wrong and is difficult to debug because it does away with several ACID

properties:

- Isolation is gone with multiple writers. Everyone sees the current written value where in a database you see a consistent view of the database.
- On a transaction fail the cache may reflect the new data when it has been rolled back in the database.
- Dependent cache keys are difficult to program correctly because they aren't transactional. Consistency is impossible to keep. Update one key and what happens to the dependent keys?
- It's complicated and you'll get it wrong and lose some of the consistency that the database had given your.

Tokyo Cabinet/Tyrant Key-Value Store

- Similar use cases as for BerkelyDB.
- Disk persistence. Can store data larger than RAM.
- Performs well.
- Actively developed. Lots of developers adding new features (but not bug fixes).
- Similar replication strategy to MySQL. Not useful for scalability as it limits the write throughput to one node.
- Optional compressed pages so has some compression advantages.

Redis Data Structure Store

- Very new.
- It's a data structure store not a key-value store, which means it understands your values so you can operate on them. Typically in a key-value store the values are opaque.
- Can match on key spaces. You can look for all keys that match an expression.
- Understands lists and sets. So you can do list and set operation in

the process of the database server which is much more efficient because all the data doesn't have to be paged to the client. Updates can then b done atomically on the server side which is difficult to do on the client side.
- Big downside is it requires that full data store in RAM. Can't store data on disk.
- It is disk backed so it is reliable over a reboot, but still RAM limited.
- Maybe useful as a cache that supports higher level operations than memcache.

Document Databases

- Schema-free. You don't have to say which attributes are in the values that are stored.
- Makes schema migration easier because it doesn't care what fields you have. Applications must be coded to handle the different versions of documents.
- Great for storing documents.

CouchDB Document Database

- Apache project so you can use it.
- Written Erlang
- Asynchronous replication. Still limited to the write speed of one node.
- JSON based so easy to use on the web.
- Queries are done in a map-reduce style using views.
    - A view is created by writing a Javascript function that is applied to all documents in the document store. This creates a matching list of documents.
    - Views are materialized on demand. Once you hit the view once it saves the list until an update occurs.

- Neat admin UI.

MongoDB Document Database

- Written in C++
- Significantly faster the CouchDB
- JSON and BSON (binary JSON-ish) formats.
- Asynchronous replication with auto-sharding coming soon.
- Supports indexes. Querying a property is quick because an index is automatically kept on updates. Trades off some write speed for more consistent read spead.
- Documents can be nested unlike CouchDB which requires applications keep relationships. Advantage is that the whole object doesn't have to be written and read because the system knows about the relationship. Example is a blog post and comments. In CouchDB the post and comments are stored together and walk through all the comments when creating a view even though you are only interested in the blog post. Better write and query performance.
- More advanced queries than CouchDB.

Column Databases

- Some of this model is implemented by BigTable, Cassandra, and HyperTable.
- Sequential reads are fast because data in a column is stored together.
- Columns compress better than rows because the data is similar.
- Each column is stored separately so IO is efficient as only the columns of interest are scanned. When using column database you are almost always scanning the entire column.
- Bitmap indexes for fast sequential scans.
    - Turning cell values into 1 or more bits.
    - Compression reduces IO even further.

    - Indexes can be logical anded and ored together to know which rows to select.
    - Used for big queries for performing joins of multiple tables. When a row is 2 bits (for example) there's a lot less IO than working on uncompressed unbitmapped values.
- Bloom Filters
    - Used by BigTable, Cassandra and other projects.
    - Probabilistic data structure.
    - Lossy, so you can lose data.
    - In exchange for losing data you can store all information in constant space
    - Gives you false positives at a known error rate.
    - Store bloom filter for a bunch of nodes. Squid uses this for its cache protocol. Knowing a bloom filter you can locally perform a computation to know which nodes the data may be on. If a bit is not set in the filter then the data isn't on the node. If a bit is set it may or may not be on the node.
    - Used for finding stuff and approximating counts in constant space.

MonetDB Column Database

- Research project which crashes a lot and corrupts your data.

LucidDB Column Database

- Java/C++ open source data warehouse
- No clustering so only single node performance, but that can be enough for the applications column stores are good at.
- No experience so can't speak to it.

Vertica Column Database

- The product they use.
- Commercial column store based on C-store.
- Clustered
- Actually works.

Related Articles

- Availability & Consistency by Werner Vogels
- BASE: An Acid Alternative by Dan Pritchett
- MongoDB - a high-performance, open source, schema-free document-oriented data store that's easy to deploy, manage and use.
- Vertica - blazing-fast data warehousing software
- LucidDB - the first and only open-source RDBMS purpose-built entirely for data warehousing and business intelligence.
- CouchDB - a distributed, fault-tolerant and schema-free document-oriented database accessible via a RESTful HTTP/JSON API.
- Memcached Tag at High Scalability
- Key-Value Store Tag at High Scalability
- BigTable Tag at High Scalability
- Dynamo.
