VoltDB Decapitates Six SQL Urban Myths and Delivers Internet Scale OLTP in the Process

Monday, June 28, 2010 at 7:36AM

Todd Hoff in Product, databases, nosql

What do you get when you take a SQL database and start a new implementation from scratch, taking advantage of the latest research and modern hardware? Mike Stonebraker, the sword wielding Johnny Appleseed of the database world, hopes you get something like his new database, VoltDB: a pure SQL, pure ACID, pure OLTP, shared nothing, sharded, scalable, lockless, open source, in-memory DBMS, purpose-built for running hundreds of thousands of transactions a second. VoltDB claims to be 100 times faster than MySQL, up to 13 times faster than Cassandra, and 45 times faster than Oracle, with near-linear scaling.

Will VoltDB kill off the new NoSQL upstarts? Will VoltDB cause a mass extinction of ancient databases? Probably no and no to both questions, but it's a product with a definite point-of-view and is worth a look as the transaction component in your system. But will it be right for you? Let's see...

I first heard the details about VoltDB at Gluecon, where Mr. Stonebraker presented his highly entertaining Urban Myths About SQL (slides) talk. The hallways were buzzing long afterwards debating his in-your-face challenge of the existing order. With a refreshing take no prisoners style he lambastes existing SQL products as not good enough, saying they should either get better or die. NoSQL products fair no better as

they simply aren't necessary, their whole existence based on a perception of the weakness of current SQL implementations rather than what SQL could be, if properly implemented.

As an argument against NoSQL, VoltDB simply asks: if you can get a relational database with all the scalable ACIDy goodness, why would you ever stoop to using a NoSQL database that might only ever be eventually reliable?

The attack against existing relational databases systems is to position them as legacy systems that suffer from archaic architectures that are slow by design. They have to deal with disks, threads, and other performance killing constructs that must not be so much evolved as transcended. You see, VoltDB isn't just competing against NoSQL, it's aiming squarely at existing relational database vendors by using the patented technological leap play.

It's a bold two prong strategy, but will the compromises that are part of the reconceptualization of SQL engine architectures prove too limiting for the majority?

VoltDB's Architecture

The bulk of the rest of this article is about the SQL Myths, but I think touching a bit on Volt's architecture before we address the myths will help frame the discussion a little better.

John Hugg, from VoltDB Engineering, says:

> *VoltDB is designed to solve OLTP at internet scale. If you don't need the scale of VoltDB, then of course you're going to be much happier with a general system like Postgres that offers so many features and is compatible with so much.*

Some other quotes indicating the spirit behind VoltDB:

> *VoltDB is designed to make difficult or impossible problems manageable.*

And:

> *When we set out to build VoltDB, we figured it wasn't worth the tradeoffs unless it's MUCH faster. So it is.*

John is not kidding. What matters to VoltDB is: *speed at scale, speed at scale, speed at scale, SQL, and ACID*. If that matches your priorities then you'll probably be happy. Otherwise, as you'll see, everything is sacrificed for speed at scale and what is sacrificed is often ease of use, generality, and error checking. It's likely we'll see ease of use improve over time, but for now it looks like rough going, unless of course, you are a going for speed at scale.

Some of the more interesting features are:

- **Main-memory storage**. VoltDB stores all its data in RAM. This means there are no buffer pools to manage, so that source of overhead is removed, and there are no blocking disk stalls.
- **Run transactions to completion –single threaded –in timestamp order**. Based on the model that 200 record updates is a hefty transaction, you might as well run them to completion. By single threading all locking overhead is removed. On multi-core systems they allocate chunks of memory to each CPU and run each CPU single threaded.
- **Replicas**. Persistence is guaranteed by having the data reside in multiple main memories. There are no logs, so there are no disks, which removes the disk overhead. For high availability an active-active architecture is used. Each transaction is run twice, once on each node in the same time-stamp order, which means the replicas are ACID consistent. Data can be asynchronously shipped to disk for a 5% performance hit. *VoltDB*

replication is not a master/slave or primary/backup replication system. Each replica is a first class, fully capable instance of a partition.

- **Tables are partitioned across multiple servers**. Partitions are defined in a project XML configuration file that defines the partition keys. Clients can connect through any node. Partitioning is automatic, but you have to decide on the sharding key. They are working on an automated system to give advice on which key to use. If new nodes are added then the data must reloaded to cause the new nodes to be used, the data is not automatically distributed to the new nodes. Not all tables need to be partitioned. *Small, mostly read-only tables can be replicated across all of the partitions of a VoltDB database*.
- **Stored procedures, written in Java, are the unit of transaction**. Only data changed within a single invocation of a stored procedure is in a transaction, transactions can't span multiple rounds of communication with a client. Also, from the same source as the previous link, *The vast majority (almost 100%) of your stored procedure invocations must be single partition for VoltDB to be useful to you. If, for example, you partitioned by graph node, updating multiple nodes in a single transaction/procedure will not be a single partition transaction.* These and other restrictions show the tradeoff between speed and generality.
- **A limited subset of SQL '99 is supported**. DDL operations like ALTER and DROP aren't supported. Operations having to do with users, groups and security have been moved into XML configuration files. Updating table structure on the fly is convenient, but it's not fast, so it's out. You are also discouraged from doing SUM operations because it would take a long time and block other transactions. Single threading means you must quantize your work into small enough chunks that don't stall the work pipeline. The goal is to have transactions run in under 50 milliseconds. This is all done for speed.
- Design a schema and workflow to use single-sited procedures. Data for a table is stored in a partition that is split onto different nodes. Each set of data on a node is called a *slice*. When a query can run on a

single node it it is said to be *single-sited*. Performance is clearly best when a query can run on just one node against a limited set of data.

- **Challenging operations model**. Changing the database schema or reconfiguring the cluster hardware requires first saving and shutting down the database. An exception are stored procedures which can be updated on the fly. In general, choosing speed as the primary design point has made the development
  and deployment process complicated and limiting. VoltDB, for example, does not support bringing a node back into the cluster while the database is running. All clients must be stopped, the database must be snapshotted, the database must be restarted in a special mode, the data is reloaded, and then clients can be restarted. See Using VoltDB for more details.
- **No WAN support**. In the case of a network partition VoltDB chooses consistency over availability, so you will see a hiccup until connectivity can be restored. Out of all the possible failures, Mr. Stonebraker argues, network partitioning is one of the least likely failures, especially compared to programmer error, so choosing strong consistency over availability is the right engineering call. Future versions of VoltDB will do more to address this single-data-center catastrophe scenario.
- **OLAP is purposefully kept separate**. VoltDB is only for OLTP. It's not for reporting or OLAP because those uses require locks be taken which destroys performance. Every update is spooled to a (optional) companion warehouse system that is a second or two behind the main transaction system (yet is perfectly consistent). The companion system could be something like Vertica, an analytics RDBMS also built by Mr. Stonebraker. The justification for this split of responsibilities is that one size does not fit all. You should run transactions on something that is good at transactions and run reporting on something that's good at reporting. An specialized transaction architecture will run circles around (50 times to 100 times faster) a one size fits all solution.

VoltDB is different because it has consciously been architected to remove

what research shows are the four common sources of overhead in database management systems: *logging (19%),* latching *(19%), locking (17%), B-tree, and buffer management operations* (35%). By removing all removing all four sources overhead VoltDB can be really fast while still being ACID. How fast?

VoltDB claims to be 100 times faster than MySQL, up to 13 times faster than Cassandra, and 45 times faster than Oracle, with near-linear scaling. Though I think linear scaling only applies when you are not using distributed transactions. Two-phase transactions with an in-memory database will be relatively fast, but they will still be slow given the protocol overhead.

Keep in mind VoltDB is in-memory, so it should be fast, that should be a given. But VoltDB says they have gone beyond what other in-memory databases have done, they haven't just improved buffer management. By removing locks, latching, and threading overhead it's that much faster than other in-memory databases. You could argue that it's a waste of RAM, that only hot data should be kept in RAM, but the contention is that RAM can hold the entire data set, so there's no reason to compromise anymore.

The performance comparison against databases like Cassandra is somewhat of a strawman as they are designed for a much different purpose. Cassandra can store petabytes of data, across hundreds of nodes, across multiple data centers, and new nodes can be added at will. Operationally there's no comparison. Though I realize the purpose of the benchmarks is to show SQL is not natively slow, can work well for key-value usage patterns, and compares favorably with other industry leaders.

I really like the parallelism of the origin of relational theory with the origin of VoltDB's architecture. Relational theory was invented to remove update anomalies that occur when storing duplicate data. When data is duplicated, either within a record or in different tables, then it's easy to cause inconsistencies when performing updates, deletes, and adds. The normalization process makes it much harder to have inconsistencies because facts are stored once and only once. It may be a stretch, but I think of the

process of creating a SQL engine architecture based on removing performance anomalies as fascinatingly analogous.

The Six SQL Urban Myths

Here are the six myths that Mr. Stonebraker says NoSQL advocates incorrectly perpetuate:

- Myth #1: SQL is too slow, so use a lower level interface
- Myth #2: I like a K-V interface, so SQL is a non-starter
- Myth #3: SQL systems don’t scale
- Myth #4: There are no open source, scalable SQL engines
- Myth #5: ACID is too slow, so avoid using it
- Myth #6: in CAP, choose AP over CA

Myth #1A: SQL is too slow because of heavy interfaces like ODBC/JDBC

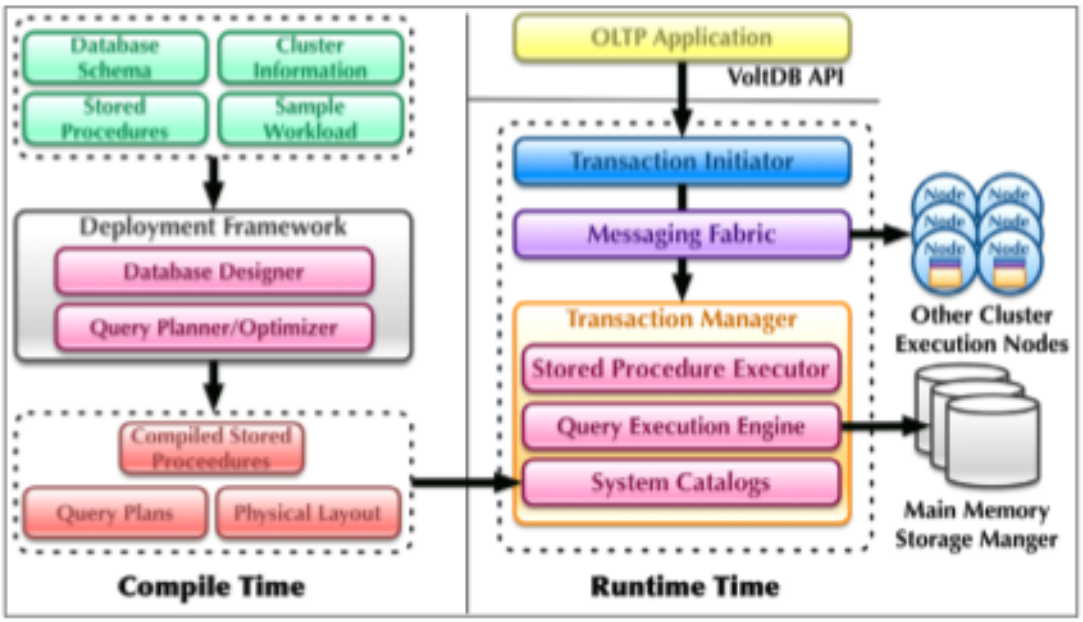
Problem:

SQL is compiled into an optimized intermediate format in a way similar to how languages like C are compiled into assembly. SQL is not slow. What is slow are heavy interfaces like ODBC/JDBC that cause too many round trips to the database. Performance is determined by this interface.

VoltDB's Solution:

Only stored procedures are supported. A main advantage stored procedures have over chatty ODBC/JDBC protocols is one message is sent to the database and one reply is sent back. All the computation is in the database. Much more efficient than ODBC/JDBC. To go even faster you can batch stored procedure calls together. Stored procedures are wildly faster than using ODBC/JDBC.

The execution flow is something like:



Discussion:

This is a bit of a strawman argument in that I've never heard anyone seriously suggest SQL itself as a language is slow.

Using stored procedures was at one time the canonical relational database architecture. For every operation a stored procedure was created that executed all the required data manipulation and a result was returned. It's perfectly right to say this is the most efficient path, given certain conditions, but there are many problems:

1. Stored procedure languages suck. They are difficult to program, ugly to use, and nearly impossible to debug. So programmers escape to the tool chains they know and love, leaving the database to deal with data, not logic. I understand Java is the stored procedure language for VoltDB. Depending on your allegiances that may be the worst or best thing in the world. The more overarching point is that you have no choice, but that may be the price of performance at scale.
2. Putting logic into the database makes the database an application server,

a function for which they are ill equipped. Let's say during a stored procedure you need to make a REST call to get a discount rate, for example, this involves blocking, IO, threading and all the usual backend server issues that databases don't know how to do well. VoltDB gets around this issue by simply not allowing you to do this.

3. Once it can't scale you are dead, dead, dead. On a few projects I've worked on we've used the stored procedure approach. It works fine until it doesn't. At a certain load the database just dies and as a system you are stuck. You can't make it perform better so you are forced to hack away until all the benefits of using the database are gone and you are left with an expensive and brittle albatross at your system's core. So that's why projects have learned not to trust trust the success of their project to how good of an application server your database can be. Instead, they separate out logic from data and let each scale independently. This is a more risk reduced approach. VoltDB counters this objection by allowing new nodes to be added into the system, by limiting the work you can do in the server, by using RAM, and by sharding so you can control how much work is done on each node. The problem is operationally, the process is so onerous.

An interesting fact about stored procedures is that they can take anywhere from half a second to several seconds to prepare a statement in VoltDB 1.0.01, so ad-hoc queries are not practical. VoltDB also does not allow SQL to *support getting the schema of tables, listing tables or altering tables*. DDL operations aren't considered part of a core OLTP database.

Myth #1B: SQL is too slow because SQL engine implementations have too much overhead

Problem:

Traditional relational databases are using 30 year old architectures that make them slow by design. Ninety percent of all CPU cycles in your typical OLTP

database go into non-productive work like: managing disk buffer pools, locking, crash recovery, multi-threading. To go faster you need to get rid of the overhead. Traditional relational databases do not do this, they are overhead rich and slow, but this is not the fault of SQL, it's the implementations that are faulty.

VoltDB's Solution:

Remove the overhead. VoltDB has made rather radical architectural choices to remove the bottlenecks. The results according to their performance test for single node performance:
• MySQL: X
• A very popular RDBMS elephant: 2.5 * X
• VoltDB: 100 X

VoltDB is 100 times faster than MySQL in their tests.

Discussion:

Seems spot on to me.

Myth #2: I like a K-V interface, so SQL is a non-starter

Problem:

Programmers like the convenience that key-value interfaces provide. Put a value by key and get a value by key. Simple. SQL doesn't provide a key-value interface so programmers won't use SQL.

VoltDB's Solution:

Create a thin get/put layer ontop of SQL using stored procedures. They are easy to support on top of a SQL engine. Using gets and puts their tests show that VoltDB is 15 times faster than MySQL and memcached.

- MySQL/Memcached: X
- Cassandra: 7*X
- VoltDB: 15 * X

Discussion:

Creating a get/put set of stored procedure was at one time standard operating procedure. Any time a table is added so are new stored procedures. What this means is every time any attribute is changed or any table is changed, the effects ripple up and down the entire stack. This is a maintenance nightmare, which reveals one of the major strengths of NoSQL: schemaless design.

I'm sure developers value the ease of putting a value, but what really matters is what that implies, you aren't tied to a rigid schema that cause nightmares every time a data model changes. Should adding a parameter really cause every interface to break? A simple stored procedure facade completely ignores this aspect of the key-value approach.

Myth #3: SQL systems don’t scale

Problem:

Some SQL engines don't support multiple nodes so they don't scale: MySQL, Postgres, Oracle, SQLServer. Some modern SQL engines do scale linearly: DB2,Vertica, Asterdata, Greenplum, DB2, Vertica, Asterdata, Greenplum, and EnterpriseDB.

VoltDB's Solution:

Their architecture scales linearly. So far they've test on 100 cores on 12 nodes and have scaled linearly.

Discussion:

As discussed previously, you must follow very precise rules about how to partition your system, limit how long queries take, accept a less than agile operations model, and accept that replication equals durability. Speed and scale has its costs, if you are willing to pay VoltDB will probably deliver.

Myth #4: There are no open source, scalable SQL engines

VoltDB is open source. In fact, Mr. Stonebraker believes all future databases will be open source.

Myth #5: ACID is too slow, so avoid using it

Problem:

Transactions are too expensive which is used by NoSQL vendors as an excuse not to provide real ACID transactions. Some applications require ACID semantics. If the database doesn't provide this feature then it's pushed to user code which is the worst of all worlds. Databases live a long time, so if you need ACID later and your database doesn't support it then you are in trouble.

Mr. Stonebraker contends 99% of all OLTP database are 1TB or less, especially if you factor out static content like pictures. This means transactional database can or will fit in main memory. Facebook is not the common case.

Now combine this with the finding in OLTP through the looking glass, and what we found there that current transactional databases do about 12% useful work, which is the actual cost of doing the retrieves and updates, the rest is spent on *logging (19%), latching (19%), locking (17%), B-tree, and buffer management operations* (35%). Multi-threading overhead on shared data structures is surprisingly high.

The abstract from the paper:

> *Online Transaction Processing (OLTP) databases include a suite of features - disk-resident B-trees and heap files, locking-based concurrency control, support for multi-threading - that were optimized for computer technology of the late 1970's. Advances in modern processors, memories, and networks mean that today's computers are vastly different from those of 30 years ago, such that many OLTP databases will now fit in main memory, and most OLTP transactions can be processed in milliseconds or less. Yet database architecture has changed little.*
>
> *Based on this observation, we look at some interesting variants of conventional database systems that one might build that exploit recent hardware trends, and speculate on their performance through a detailed instruction-level breakdown of the major components involved in a transaction processing database system (Shore) running a subset of TPC-C. Rather than simply profiling Shore, we progressively modified it so that after every feature removal or optimization, we had a (faster) working system that fully ran our workload. Overall, we identify overheads and optimizations that explain a total difference of about a factor of 20x in raw performance. We also show that there is no single "high pole in the tent" in modern (memory resident) database systems, but that substantial time is spent in logging, latching, locking, B-tree, and buffer management operations.*

There's not one thing you can do to improve performance as the cost is spread around evenly. Better B-Trees, for example, don't really help. You are only optimizing the 12% part of the overhead which gets you nowhere.

VoltDB's Solution:

VoltDB removes all four sources of overhead. Getting rid of ACID only gets you to two times faster, if you want to go ten times faster you need to get rid of the buffer pool overhead and the multithreading overhead too. This is a far more radical architecture change.

The result of getting rid of all four sources of overhead is that VoltDB supports ACID transaction semantics while retaining performance. Transactions can be used all the time with no penalty. The obvious implication being: so why would you ever not use a fast, scalable relational database that is faster than anything else you have ever used?

Discussion:

I assume, if it were possible, everyone would like ACID transactions. To make it possible VoltDB makes a number of restrictions that make VoltDB less than ideal as general purpose database. If you have lots of data then VoltDB wont work for you because lots of data still won't fit in RAM. If you want easy operations then VoltDB won't work for you. If you want to use your own language then VoltDB won't work for you. If you want to have longer lived transactions that integrate inputs from a series of related inputs then VoltDB will not work for you. If you want a little OLAP with your OLTP then VoltDB will not work for you. If you want cross data center availability then VoltDB will not work for you.

And VoltDB never said it would do all that. VoltDB promises damn fast and scalable ACID transactions at all costs. But given all the limitations needed to get there, saying NoSQL simply thinks ACID is too expensive, seems mighty unfair.

Myth #6: in CAP, choose AP over CA

Problem:

CAP theorem says partitions happen so you have to get rid of consistently or

availability, if you prefer availability then you have to jettison consistency.

Mr. Stonebraker says there a few things to think about when making this tradeoff:

- **Order matters**. Not all transactions are commutative. So if you get rid of ACID and you have operations where orders matters then eventual consistency will give you the wrong result. If you are transactions are a series of additions and subtractions then the order doesn't matter. But as soon as you throw in a non-commutative operation like multiplication then you will get wrong results. If you decide later that you need ACID type transactions then it's a fork-lift upgrade.
- **Semantic constraints**. If you have something like stock for an item and you don't want to send out an item when there is non stock left, when there's a partition in an eventually consistent system you will not be able to do this as each partition has no idea of what the others are doing. So if you have semantic constraints you need ACID.
- **Errors**. Errors are what cause partitions. Humans screwing up are the biggest source of errors. Next are bugs in applications or the database itself. Lastly, network failures cause partitions. Network failures are rare so designing an eventually consistent system for the rarest failure case seems like a bad engineering system. It makes more sense to take the hit for the rare case of a network being down, especially given the number of failures you'll have because of other non-network related problems is so much higher. Sacrificing consistency for availability is a bad engineering tradeoff.

VoltDB's Solution:

On network failures VoltDB proudly chooses consistency over availability. Your database is down until the network partition and the database are repaired.

Discussion:

I agree completely that moving the repair logic to the programmer is a recipe for disaster. Having programmers worry about read repair, vector clocks, the commutativity of transactions, how to design compensatory transactions to make up for previous failed transactions, and the other very careful bits of design, is asking for a very fragile system. ACID transactions are clean and understandable and that's why people like them.

Do you buy the argument that network partitions are rare? To the extent you buy this argument determines how you feel about VoltDB's design choices. Cloud architectures now assume failure as a starting point. They generally assume nodes within a data center are unreliable and that the more nodes you have and the more data center boundaries you cross, the more reason there is to expect failure. Amazon, for example, made the decision to go with an eventually consistent architecture by using Dynamo to back their shopping carts. Would they have done that for no reason? With that philosophy in mind, Cassandra has embraced failure at its core. Nodes can be added or taken down at any time, all because failure is assumed from the start. This makes for a very robust system. The price is a lack of distributed ACID transactions.

In contrast, the VoltDB model seems from a different age: a small number of highly tended nodes in a centralized location. In return though you get performance and ACID guarantees. It all depends on where you want to place your partition failure bets.

General Discussion

Our Future is Polyglot

VoltDB is a racehorse. It's a very specialized component that has been bred to do one thing and one thing only: be a very fast and scalable OLTP database. If that's what you need then VoltDB could be your Triple Crown winner.

What VoltDB reinforces for me is that our future is polyglot. Specialized tools for specialized purposes. VoltDB is specialized for OLTP. It's a singletasker that doesn't want to be mediocre at everything, it wants to be great at one thing. A pattern we are seeing more and more. For graph access use a graph database. For search use a search subsystem. And so on.

We Need Cache Consistency Protocols

What we may need is a robust cache coherence protocol between different data silos. In this kind of architecture there's no real master database other databases can sync to. Every silo has their equally valid perspective on the data. As events stream in to each silo we need some way of keeping all the silos consistent. I have not seen this type of component yet.

Open Source Means We Can Build Complex Interacting Systems

What makes the polyglot future possible is open source. With open source it's possible to make a system of many complex parts because open source pricing means it doesn't cost anything until you want extra support. Spending a good percentage of your budget on a mega database means it simply has to do everything because you can't afford anything else. With open source we can create very powerful composite systems that wouldn't otherwise be possible to create with lower end budgets.

Related Articles

- VoltDB FAQ
- VoltDB Technical Overview
- OLTP through the looking glass, and what we found there by Harizopoulos et. al.
- VoltDB finally launches at Monash Research.
- The End of an Architectural Era (It’s Time for a Complete Rewrite)

- VoltDB Community Forum
- NimbusDb is another from the ground rewrite of the relational database that takes a very different approach than VoltDb.
- H-Store - father of VoltDB.
- High Volume Transaction Processing. Without Concurrency Control, Two Phase Commit, SQL or C++. Arthur Whitney, Dennis Shasha, Stevan Apter, KX Systems.
- On the Radar: VoltDB, Just the Latest Database Company from Mike Stonebraker by Scott Kirsner.
- Comparing VoltDB to Postgres by Dave Page.
- VoltDB Don’ts Validating NoSQL Assumptions on myNoNSql.
- What is a latch? From Locking and Latching in a Memory-Resident Database System: A latch, or short-term lock, is a low level primitive that provides a cheap serialization mechanism with shared and exclusive lock modes, but no deadlock detection. In Starburst, things, gain exclusive or shared access to buffer pool pages. Each time the VRM storage component needs access to a page, it contacts the buffer pool which then latches the page in the buffer pool in a shared or exclusive mode.
