Are Cloud Based Memory Architectures the Next Big Thing?

Monday, March 16, 2009 at 10:54AM

Todd Hoff in memory-grid

We are on the edge of two potent technological changes: Clouds and Memory Based Architectures. This evolution will rip open a chasm where new players can enter and prosper. Google is the master of disk. You can't beat them at a game they perfected. Disk based databases like SimpleDB and BigTable are complicated beasts, typical last gasp products of any aging technology before a change. The next era is the age of Memory and Cloud which will allow for new players to succeed. The tipping point will be soon.

Let's take a short trip down web architecture lane:

- It's 1993: Yahoo runs on FreeBSD, Apache, Perl scripts and a SQL database
- It's 1995: Scale-up the database.
- It's 1998: LAMP
- It's 1999: Stateless + Load Balanced + Database + SAN
- It's 2001: In-memory data-grid.
- It's 2003: Add a caching layer.
- It's 2004: Add scale-out and partitioning.
- It's 2005: Add asynchronous job scheduling and maybe a distributed file system.
- It's 2007: Move it all into the cloud.
- It's 2008: Cloud + web scalable database.
- It's 20??: Cloud + Memory Based Architectures

You may disagree with the timing of various innovations and you would be correct. I couldn't find a history of the evolution of website architectures, so I just made stuff up. If you have any better information please let me know.

Why might cloud based memory architectures be the next big thing? For now we'll just address the memory based architecture part of the question, the cloud component is covered a little later.

Behold the power of keeping data in memory:

> *Google query results are now served in under an astonishingly fast 200ms, down from 1000ms in the olden days. The vast majority of this great performance improvement is due to holding indexes completely in memory. Thousands of machines process each query in order to make search results appear nearly instantaneously.*

This text was adapted from notes on Google Fellow Jeff Dean keynote speech at WSDM 2009.

Google isn't the only one getting a performance bang from moving data into memory. Both LinkedIn and Digg keep the graph of their network social network in memory. Facebook has northwards of 800 memcached servers creating a reservoir of 28 terabytes of memory enabling a 99% cache hit rate. Even little guys can handle 100s of millions of events per day by using memory instead of disk.

With their new Unified Computing strategy Cisco is also entering the memory game. Their new machines "will be focusing on networking and memory" with servers crammed with 384 GB of RAM, fast processors, and blazingly fast processor interconnects. Just what you need when creating memory based systems.

Memory is the System of Record

What makes Memory Based Architectures different from traditional architectures is that **memory is the system of record**. Typically disk based databases have been the system of record. Disk has been King, safely storing data away within its castle walls. Disk being slow we've ended up wrapping disks in complicated caching and distributed file systems to make them perform.

Sure, memory is used as all over the place as cache, but we're always supposed to pretend that cache can be invalidated at any time and old Mr. Reliable, the database, will step in and provide the correct values. In Memory Based Architectures memory is where the "official" data values are stored.

Caching also serves a different purpose. The purpose behind cache based architectures is to minimize the data bottleneck through to disk. Memory based architectures can address the entire end-to-end application stack. Data in memory can be of higher reliability and availability than traditional architectures.

Memory Based Architectures initially developed out of the need in some applications spaces for very low latencies. The dramatic drop of RAM prices along with the ability of servers to handle larger and larger amounts of RAM has caused memory architectures to verge on going mainstream. For example, someone recently calculated that 1TB of RAM across 40 servers at 24 GB per server would cost an additional $40,000. Which is really quite affordable given the cost of the servers. Projecting out, 1U and 2U rack-mounted servers will soon support a terabyte or more or memory.

RAM = High Bandwidth and Low Latency

Why are Memory Based Architectures so attractive? Compared to disk RAM is a high bandwidth and low latency storage medium. Depending on who you ask the bandwidth of RAM is 5 GB/s. The bandwidth of disk is about 100 MB/s. RAM bandwidth is many hundreds of times faster. RAM wins. Modern hard drives have latencies under 13 milliseconds. When many applications are queued for disk reads latencies can easily be in the many second range. Memory latency is in the 5 nanosecond range. Memory latency is 2,000 times faster. RAM wins again.

RAM is the New Disk

The superiority of RAM is at the heart of the RAM is the New Disk paradigm. As an architecture it combines the holy quadrinity of computing:

- Performance is better because data is accessed from memory instead of through a database to a disk.
- Scalability is linear because as more servers are added data is transparently load balanced across the servers so there is an automated in-memory sharding.
- Availability is higher because multiple copies of data are kept in memory and the entire system reroutes on failure.
- Application development is faster because there’s only one layer of software to deal with, the cache, and its API is simple. All the complexity is hidden from the programmer which means all a developer has to do is get and put data.

Access disk on the critical path of any transaction limits both throughput and latency. Committing a transaction over the network in-memory is faster than writing through to disk. Reading data from memory is also faster than reading data from disk. So the idea is to skip disk, except perhaps as an asynchronous write-behind option, archival storage, and for large files.

Or is Disk is the the new RAM

To be fair there is also a Disk is the the new RAM, RAM is the New Cache paradigm too. This somewhat counter intuitive notion is that a cluster of about 50 disks has the same bandwidth of RAM, so the bandwidth problem is taken care of by adding more disks.

The latency problem is handled by reorganizing data structures and low level algorithms. It's as simple as avoiding piecemeal reads and organizing algorithms around moving data to and from memory in very large batches and writing highly parallelized programs. While I have no doubt this approach can be made to work by very clever people in many domains, a large chunk of applications are more time in the random access domain space for which RAM based architectures are a better fit.

Grids and a Few Other Definitions

There's a constellation of different concepts centered around Memory Based Architectures that we'll need to understand before we can understand the different products in this space. They include:

- **Compute Grid** - parallel execution. A Compute Grid is a set of CPUs on which calculations/ jobs/work is run. Problems are broken up into smaller tasks and spread across nodes in the grid. The result is calculated faster because it is happening in parallel.
- **Data Grid** - a system that deals with data — the controlled sharing and management of large amounts of distributed data.
- **In-Memory Data Grid (IMDG)** - parallel in-memory data storage. Data Grids are scaled horizontally, that is by adding more nodes. Data contention is removed removed by partitioning data across nodes.
- **Colocation** - Business logic and object state are colocated within the same process. Methods are invoked by routing to the object and having the object execute the method on the node it was mapped to. Latency is low because object state is not sent across the wire.
- **Grid Computing** - Compute Grids + Data Grids
- **Cloud Computing** - datacenter + API. The API allows the set of CPUs in the grid to be dynamically allocated and deallocated.

Who are the Major Players in this Space?

With that bit of background behind us, there are several major players in this space (in alphabetical order):

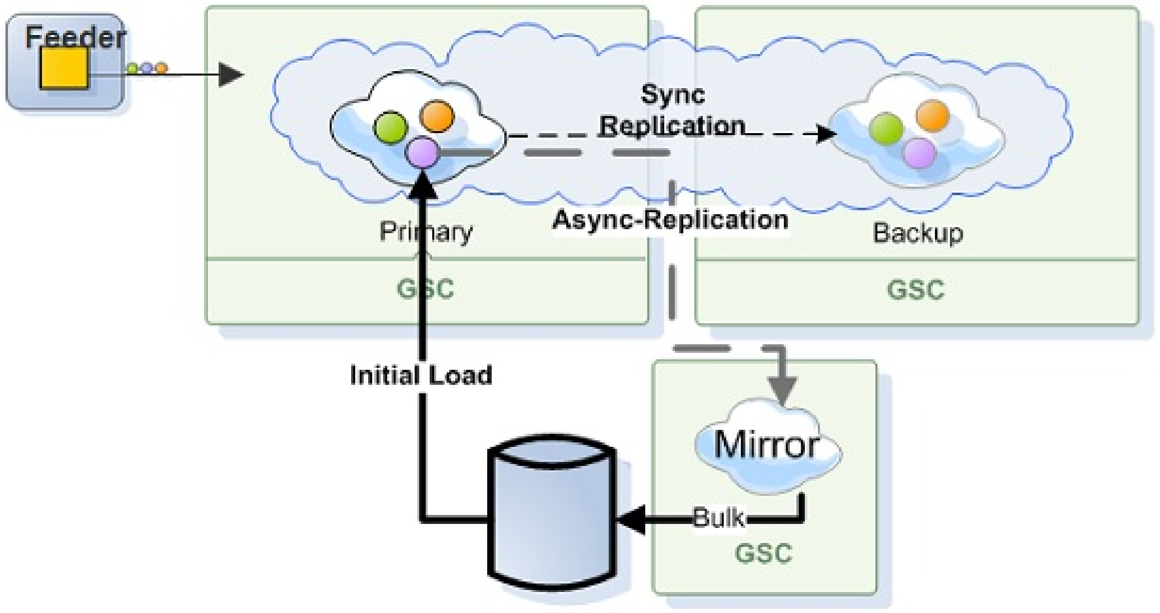
- Coherence - is a peer-to-peer, clustered, in-memory data management system. Coherence is a good match for applications that need write-behind functionality when working with a database and you require multiple applications have ACID transactions on the database. Java, JavaEE, C++, and .NET.
- GemFire - an in-memory data caching solution that provides low-latency and near-zero downtime along with horizontal & global scalability. C++, Java and .NET.
- GigaSpaces - GigaSpaces attacks the whole stack: Compute Grid, Data Grid, Message, Colocation, and Application Server capabilities. This makes for greater complexity, but it means there's less plumbing that needs to be written and developers can concentrate on writing business logic. Java, C, or .Net.
- GridGain - A compute grid that can operate over many data grids. It specializes in the transparent and low configuration implementation of features. Java only.

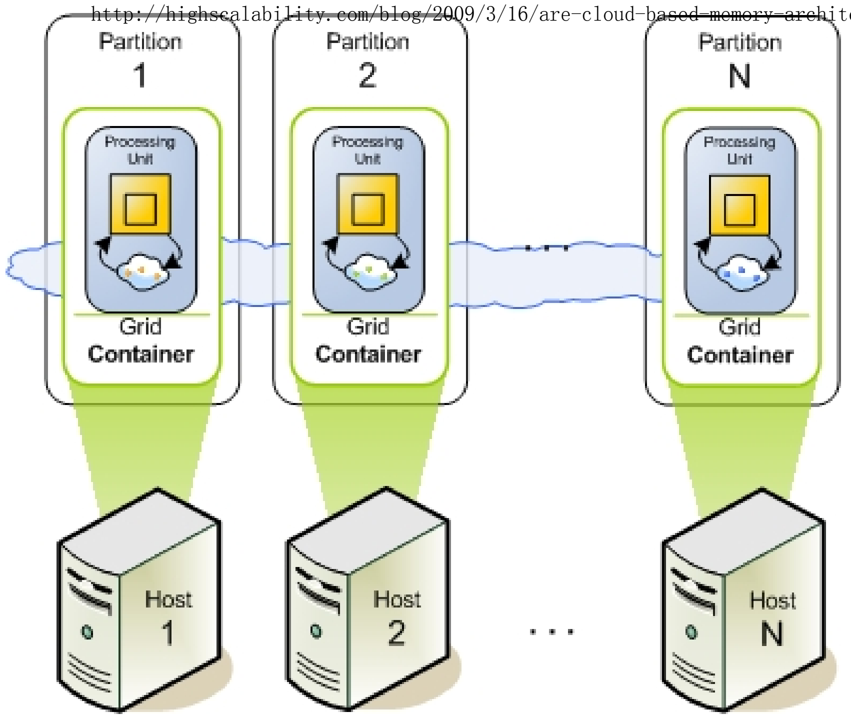
- Terracotta - Terracotta is network-attached memory that allows you share memory and do anything across a cluster. Terracotta works its magic at the JVM level and provides: high availability, an end of messaging, distributed caching, a single JVM image. Java only.
- WebSphere eXtreme Scale. *Operates as an in-memory data grid that dynamically caches, partitions, replicates, and manages application data and business logic across multiple servers.*

This class of products has generally been called In-Memory Data Grids (IDMG), though not all the products fit snugly in this category. There's quite a range of different features amongst the different products.

I tossed IDMG the acronym in favor of Memory Based Architectures because the "in-memory" part seems redundant, the grid part has given way to the cloud, the "data" part really can include both data and code. And there are other architectures that will exploit memory yet won't be classic IDMG. So I just used Memory Based Architecture as that's the part that counts.

Given the wide differences between the products there's no canonical architecture. As an example here's a diagram of how GigaSpaces In-Memory-Data-Grid on the Cloud works.





Some key points to note are:

● A POJO (Plain Old Java Object) is written through a proxy using a hash-based data routing mechanism to be stored in a partition on a Processing Unit. Attributes of the object are used as a key. This is straightforward hash based partitioning like you would use with memcached.

● You are operating through GigaSpace's framework/container so they can automatically handle things like messaging, sending change events, replication, failover, master-worker pattern, map-reduce, transactions, parallel processing, parallel query processing, and write-behind to databases.

● Scaling is accomplished by dividing your objects into more partitions and assigning the partitions to Processing Unit instances which run on nodes-- a scale-out strategy. Objects are kept in RAM and the objects contain both state and behavior. A Service Grid component supports the dynamic creation and termination of Processing Units.

Not conceptually difficult and familiar to anyone who has used caching systems like memcached. Only is this case memory is not just a cache, it's the system of record.

Obviously there are a million more juicy details at play, but that's the gist of it. Admittedly GigaSpaces is on the full featured side of the product equation, but from a memory based architecture perspective the ideas should generalize. When you shard a database, for example, you generally lose the ability to execute queries, you have to do all the assembly yourself. By using GigaSpaces framework you get a lot of very high-end features like parallel query processing for free.

The power of this approach certainly comes in part from familiar concepts like partitioning. But the speed of memory versus disk also allows entire new levels of performance and reliability in a relatively simple and easy to understand and deploy package.

NimbusDB - the Database in the Cloud

Jim Starkey, President of NimbusDB, is not following the IDMG gang's lead. He's taking a completely fresh approach based on thinking of the cloud as a new platform unto itself. Starting from scratch, what would a database for the cloud look like?

Jim is in position to answer this question as he has created a transactional database engine for MySQL named Falcon and added multi-versioning support to InterBase, the first relational database to feature MVCC (Multiversion Concurrency Control).

What defines the cloud as a platform? Here's are some thoughts from Jim I copied out of the Cloud Computing group. You'll notice I've quoted Jim way way too much. I did that because Jim is an insightful guy, he has a lot of interesting things to say, and I think he has a different spin on the future of databases in the cloud than anyone else I've read. He also has the advantage of course of not having a shipping product, but we shall see.

- I've probably said this before, but the cloud is a new computing platform that some have learned to exploit, others are scrambling to master, but most people will see as nothing but a minor variation on what they're already doing. This is not new. When time sharing as invented, the batch guys considered it as remote job entry, just a variation on batch. When departmental computing came along (VAXes, et al), the timesharing guys considered it nothing but timesharing on a smaller scale. When PCs and client/server computing came along, the departmental computing guys (i.e. DEC), considered PCs to be a special case of smart terminals. And when the Internet blew into town, the client server guys considered it as nothing more than a global scale LAN. So the batchguys are dead, the timesharing guys are dead, the departmental computing guys are dead, and the client server guys are dead. Notice a pattern?
- The reason that databases are important to cloud computing is that virtually all applications involve the interaction of client data with a shared, persistent data store. And while application processing can be easily scaled, the limiting factor is the database system. So if you plan to do anything more than play Tetris in the cloud, the issue of database management should be foremost in your mind.
- Disks are the limiting factors in contemporary database systems. Horrible things, disk. But conventional wisdom is that you build a clustered database system by starting with a distributed file system. Wrong. Evolution is faster processors, bigger memory, better tools. Revolution is a different way of thinking, a different topology, a different way of putting the parts together.
- What I'm arguing is that a cloud is a different platform, and what works well for a single computer doesn't work at all well in cloud, and things that work well in a cloud don't work at all on the single computer system. So it behooves us to re-examine a lot an ancient and honorable assumptions to see if they make any sense at all in this brave new world.

- Sharing a high performance disk system is fine on a single computer, troublesome in a cluster, and miserable on a cloud.
- I'm a database guy who's had it with disks. Didn't much like the IBM 1301, and disks haven't gotten much better since. Ugly, warty, slow, things that require complex subsystems to hide their miserable characteristics. The alternative is to use the memory in a cloud as a distributed L2 cache. Yes, disks are still there, but they're out of the performance loop except for data so stale that nobody has it memory.
- Another machine or set of machines is just as good as a disk. You can quibble about reliable power, etc, but write queuing disks have the same problem.
- Once you give up the idea of logs and page caches in favor of asynchronous replications, life gets a great deal brighter. It really does make sense to design to the strengths of cloud (redundancy) rather than their weaknesses (shared anything).
- And while one guys is fetching his 100 MB per second, the disk is busy and everyone else is waiting in line contemplating existence. Even the cheapest of servers have two gigabit ethernet channels and switch. The network serves everyone in parallel while the disk is single threaded
- I favor data sharing through a formal abstraction like a relational database. Shared objects are things most programmers are good at handling. The fewer the things that application developers need to manage the more likely it is that the application will work.
- I buy the model of object level replication, but only as a substrate for something with a more civilized API. Or in other words, it's a foundation, not a house.
- I'd much rather have a pair of quad-core processors running as independent servers than contending for memory on a dual socket server. I don't object to more cores per processor chip, but I don't want to pay for die size for cores perpetually stalled for memory.
- The object substrate worries about data distribution and who should see what. It doesn't even know it's a database. SQL semantics are applied by an engine layered on the object substrate. The SQL engine doesn't worry or even know that it's part of a distributed database -- it just executes SQL statements. The black magic is MVCC.
- I'm a database developing building a database system for clouds. Tell me what you need. Here is my first approximation: A database that scales by adding more computers and degrades gracefully when machines are yanked out; A database system that never needs to be shut down; Hardware and software fault tolerance; Multi-site archiving for disaster survival; A facility to reach into the past to recover from human errors (drop table customers; oops;); Automatic load balancing
- MySQL scales with read replication which requires a full database copy to start up. For any cloud relevant application, that's probably hundreds of gigabytes. That makes it a mighty poor candidate for on-demand virtual servers.
- Do remember that the primary function of a database system is to maintain consistency. You don't want a dozen people each draining the last thousand buckets from a bank account or a debit to happen without the corresponding credit.
- Whether the data moves to the work or the work moves to the data isn't that important as long as they both end up a the same place with as few intermediate round trips as possible.

- In my area, for example, databases are either limited by the biggest, ugliest machine you can afford *or* you have to learn to operation without consistent, atomic transactions. A bad rock / hard place choice that send the cost of scalable application development through the ceiling. Once we solve that, applications that server 20,000,000 users will be simple and cheap to write. Who knows where that will go?
- To paraphrase our new president, we must reject the false choice between data consistency and scalability.
- Cloud computing is about using many computers to scale problems that were once limited by the capabilities of a single computer. That's what makes clouds exciting, at least to me. But most will argue that cloud computing is a better economic model for running many instances of a single computer. Bah, I say, bah!
- Cloud computing is a wonder new platform. Let's not let the dinosaurs waiting for extinction define it as a minor variation of what they've been doing for years. They will, of course, but this (and the dinosaurs) will pass.
- The revolutionary idea is that applications don't run on a single computer but an elastic cloud of computers that grows and contracts by demand. This, in turn, requires an applications infrastructure that can a) run a single application across as many machines as necessary, and b) run many applications on the same machines without any of the cross talk and software maintenance problems of years past. No, the software infrastructure required to enable this is not mature and certainly not off the shelf, but many smart folks are working on it.
- There's nothing limiting in relational except the companies that build them. A relational database can scale as well as BigTable and SimpleDB but still be transactional. And, unlike BigTable and SimpleDB, a relational database can model relationships and do exotic things like transferring money from one account to another without "breaking the bank.". It is true that existing relational database systems are largely constrained to single cpu or cluster with a shared file system, but we'll get over that.
- Personally, I don't like masters any more than I like slaves. I strongly favor peer to peer architectures with no single point of failure. I also believe that database federation is a work-around
rather than a feature. If a database system had sufficient capacity, reliability, and availability, nobody would ever partition or shard data. (If one database instance is a headache, a million tiny ones is a horrible, horrible migraine.)
- Logic does need to be pushed to the data, which is why relational database systems destroyed hierarchical (IMS), network (CODASYL), and OODBMS. But there is a constant need to push semantics higher to further reduce the number of round trips between application semantics and the database systems. As for I/O, a database system that can use the cloud as an L2 cache breaks free from dependencies on file systems. This means that bandwidth and cycles are the limiting factors, not I/O capacity.
- What we should be talking about is trans-server application architecture, trans-server application platforms, both, or whether one will make the other unnecessary.
- If you scale, you don't/can't worry about server reliability. Money spent on (alleged) server

reliability is money wasted.

- If you view the cloud as a new model for scalable applications, it is a radical change in computing platform. Most people see the cloud through the lens of EC2, which is just another way to run a server that you have to manage and control, then the cloud is little more than a rather
boring business model. When clouds evolve to point that applications and databases can utilize whatever resources then need to meet demand without the constraint of single machine limitations, we'll have something really neat.
- On MVCC: Forget about the concept of master. Synchronizing slaves to a master is hopeless. Instead, think of a transaction as a temporal view of database state; different transactions will have different views. Certain critical operations must be serialized, but that still doesn't require that all nodes have identical views of database state.
- Low latency is definitely good, but I'm designing the system to support geographically separated sub-clouds. How well that works under heavy load is probably application specific. If the amount of volatile data common to the sub-clouds is relatively low, it should work just fine provided there is enough bandwidth to handle the replication messages.
- MVCC tracks multiple versions to provide a transaction with a view of the database consistent with the instant it started while preventing a transaction from updating a piece of data that it could not see. MVCC is consistent, but it is not serializable. Opinions vary between academia and the real world, but most database practitioners recognize that the consistency provided by MVCC is sufficient for programmers of modest skills to product robust applications.
- MVCC, heretofore, has been limited to single node databases. Applied to the cloud with suitable bookkeeping to control visibility of updates on individual nodes, MVCC is as close to black magic as you are likely to see in your lifetime, enabling concurrency and consistency with mostly non-blocking, asynchronous messaging. It does, however, dispense with the idea that a cloud has at any given point of time a single definitive state. Serializability implemented with record locking is an attempt to make distributed system march in lock-step so that the result is as if there there no parallelism between nodes. MVCC recognizes that parallelism is the key to scalability. Data that is a few microseconds old is not a problem as long as updates don't collide.

Jim certainly isn't shy with his opinions :-)

My summary of what he wants to do with NimbusDB is:

- Make a scalable relational database in the cloud where you can use normal everyday SQL to perform summary functions, define referential integrity, and all that other good stuff.
- Transactions scale using a distributed version of MVCC, which I do not believe has been done before. This is the key part of the plan and a lot depends on it working.
- The database is stored primarily in RAM which makes cloud level scaling of an RDBMS possible.
- The database will handle all the details of scaling in the cloud. To the developer it will look like just a very large highly available database.

I'm not sure if NimbusDB will support a compute grid and map-reduce type functionality. The low latency argument for data and code collocation is a good one, so I hope it integrates some sort of extension mechanism.

Why might NimbusDB be a good idea?

- **Keeps simple things simple**. Web scale databases like BigTable and SimpleDB make simple things difficult. They are full of quotas, limits, and restrictions because by their very nature they are just a key-value layer on top of a distributed file system. The database knows as little about the data as possible. If you want to build a sequence number for a comment system, for example, it takes complicated sharding logic to remove write contention. Developers are used to SQL and are comfortable working within the transaction model, so the transition to cloud computing would be that much easier. Now, to be fair, who knows if NimbusDB will be able to scale under high load either, but we need to make simple things simple again.
- **Language independence**. Notice the that IDMG products are all language specific. They support some combination of .Net/Java/C/C++. This is because they need low level object knowledge to transparently implement their magic. This isn't bad, but it does mean if you use Python, Erlang, Ruby, or any other unsupported language then you are out of luck. As many problems as SQL has, one of its great gifts is programmatic universal access.
- **Separates data from code**. Data is forever, code changes all the time. That's one of the common reasons for preferring a database instead of an objectbase. This also dovetails with the language independence issue. Any application can access data from any language and any platform from now and into the future. That's a good quality to have.

The smart money has been that cloud level scaling requires abandoning relational databases and distributed transactions. That's why we've seen an epidemic of key-value databases and eventually consistent semantics. It will be fascinating to see if Jim's combination of Cloud + Memory + MVCC can prove the insiders wrong.

Are Cloud Based Memory Architectures the Next Big Thing?

We've gone through a couple of different approaches to deploying Memory Based Architectures. So are they the next big thing?

Adoption has been slow because it's new and different and that inertia takes a while to overcome. Historically tools haven't made it easy for early adopters to make the big switch, but that is changing with easier to deploy cloud based systems. And current architectures, with a lot of elbow grease, have generally been good enough.

But we are seeing a wide convergence on caching as way to make slow disks perform. Truly enormous amounts of effort are going into adding cache and then trying to keep the database and applications all in-sync with cache as bottom up and top down driven changes flow through the system.

After all that work it's a simple step to wonder why that extra layer is needed when the data could have just as well be kept in memory from the start. Now add the ease of cloud deployments and the ease of creating scalable, low latency applications that are still easy to program, manage, and deploy. Building multiple complicated layers of application code just to make the disk happy will make less and less sense over time.

We are on the edge of two potent technological changes: Clouds and Memory Based Architectures. This evolution will rip open a chasm where new players can enter and prosper. Google is the master of disk. You can't beat them at a game they perfected. Disk based databases like SimpleDB and BigTable are complicated beasts, typical last gasp products of any aging technology before a change. The next era is the age of Memory and Cloud which will allow for new players to succeed. The tipping point is soon.

Related Articles

- GridGain: One Compute Grid, Many Data Grids
- GridGain vs Hadoop
- Cameron Purdy: Defining a Data Grid
- Compute Grids vs. Data Grids
- Performance killer: Disk I/O by Nathanael Jones
- RAM is the new disk... by Steven Robbins
- Talk on disk as the new RAM by Greg Linden
- Disk-Based Parallel Computation, Rubik's Cube, and Checkpointing by Gene Cooperman, Northeastern Professor, High Performance Computing Lab - Disk is the the new RAM and RAM is the new cache
- Disk is the new disk by David Hilley.
- Latency lags bandwidth by David A. Patterson
- InfoQ Article - RAM is the new disk... by Nati Shalom
- Tape is Dead Disk is Tape Flash is Disk RAM Locality is King by Jim Gray
- Product: ScaleOut StateServer is Memcached on Steroids
- Cameron Purdy: Defining a Data Grid
- Compute Grids vs. Data Grids
- Latency is Everywhere and it Costs You Sales - How to Crush it
- Virtualization for High Performance Computing by Shai Fultheim
- Multi-Multicore Single System Image / Cloud Computing. A Good Idea? (part 1) by Greg Pfister
- How do you design and handle peak load on the Cloud ? by Cloudiquity.

- Defining a Data Grid by Cameron Purdy
- The Share-Nothing Architecture by Zef Hemel.
- Scaling memcached at Facebook
- Cache-aside, write-behind, magic and why it sucks being an Oracle customer by Stefan Norberg.
- Introduction to Terracotta by Mike
- The five-minute rule twenty years later, and how flash memory changes the rules by Goetz Graefe

---