What Should I Do?

Choosing SQL, NoSQL or Both for Scalable Web Applications

Todd Hoff
http://highscalability.com

It's All About Solving Problems and Building Solutions

- Let's set the stage. Imagine a programming team sitting in a conference room, a gray glow of laptop light illuminating their faces. On the white board is a problem to be solved. That's where it all starts. Figuring out how to solve a problem.
- We all are just trying figure things out and make things work.
- Then why is there is so much SQL on NoSQL hate?
- Everything is wonderful and nobody is happy.
- Systems evolve by solving problems. Some of those solutions require discontinuous jumps into the unknown territory and some require small continuous steps on a well known path.
- Organizing the world's information, being the heartbeat of the Internet, or being the world's social network all drive you to different places. Your application has its unique place too.

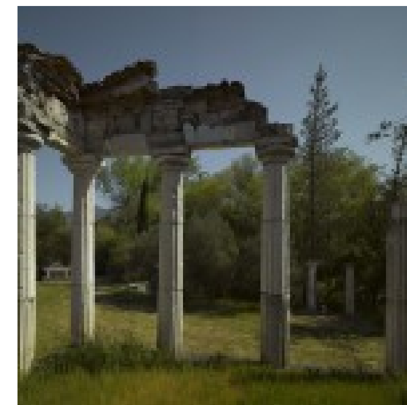
How We Solve Problems is Changing

- It's not just about scaling (distributed systems). Though that's part of it. We are talking architecture. It's about how we build things now. It's not SQL vs NoSQL. This is the main point.
- In the same way ChromeOS is rethinking the personal computer post browser, we have been rethinking architectures post cloud + NoSQL + social, etc.
- Choose simplicity, rapid development, consistency, availability, ACID, latency, scale-out, distribution, cost, operations, elasticity, queryability, manageability, navigability, data model fit, low cost, and so on. Options we never had or even knew we had before.
- Forget SQL vs NoSQL, there's a spectrum of different options and as developers we need to figure which ones to use in which combinations to solve our problems.

James Burke: Connections

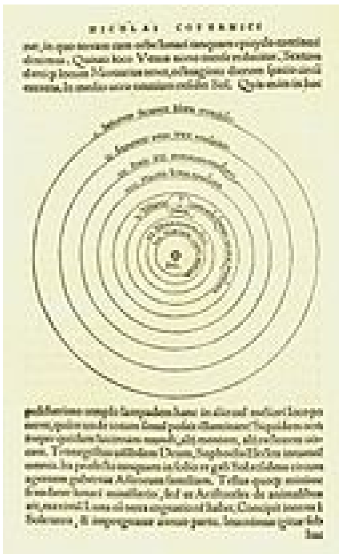
Connections explores an "Alternative View of Change" (the subtitle of the series) that rejects the conventional linear and teleological view of historical progress. Burke contends that one cannot consider the development of any particular piece of the modern world in isolation. Rather, the entire gestalt of the modern world is the result of a web of interconnected events, each one consisting of a person or group acting for reasons of their own (e.g., profit, curiosity, religious) motivations with no concept of the final, modern result of what either their or their contemporaries' actions finally led to.

Architectures of the Past



- Any system is like an archeological dig.
  - Mainframe, Minicomputers, Workstations, PCs
  - Integrated Data Store, relational, embedded, client-server
  - Canonical architecture: CDN, load balancers, web tier, application tier, database tier, storage tier.
  - Mostly fixed, static, monolothic.
- When the web was largely read-only we could scale-up or replicate, pool identical databases, keep caches.
- When the web went real-time and interactive this broke down. Too large for one machine & distributed transactions.
- We made the relational database tier do everything.
- When it couldn't, we extended it Borg-like. Adding memcached, sharding, key-value, custom consistency logic, moving all logic into apps, avoiding joins, etc..

The Idea of the Big Idea



Copernicus's discovery of the heliocentric model of the solar system was published in 1543, and it started a fire storm of scientific invention (distinction between electricity and magnetism; law of free fall; Galilean inertia; theory of lenses; laws of planetary motion, etc), not because of the discovery itself, but because it spread the idea that we puny humans could think and make big discoveries about the universe using nothing but our tiny brains.

This was a brave new thought. A big idea. Copernicus gave people permission to tackle big challenges and the confidence that they could expect to meet them.

From *It Started with Copernicus* by Howard Margolis

Architectures of the Near Now

- **Big Idea**: Some brave souls started writing radical, specialized solutions to solve specialized problems, like dealing with massive scale, massive distribution, massive concurrency, massive users. Google, Inktomi, Amazon, etc.
- Thoughts from the deep bubbled up. CAP. Sharding. Scale-out. Commodity hardware. Partitioning. ACID is bad for availability. Building highly available systems is different. We can play with consistency, availabillty, and partition tolerances.
- Developers took these ideas and built scalable architectures on top of existing software. Brutal. Facebook, FriendFeed, Flickr, Salesforce, eBay, etc all did this.
- These were simply too hard. It's still easier to scale-up and stick with a relation database core.

Architectures of the Now

- Architectures were still either/or. You built for scale or you didn't.
- That's changing now with a pleasing number of new products that want to help bridge the chasm.
- It's a time of transition and we still have everything all jumbled together at once. That's why it's so dang hard to make a decision.
- It used to be scaling took so much more work it wasn't worth it. Now with the new tool chains it's becoming equal. Not quite, but vendors are speeding in that direction, but it's still not a no-brainer, so the confusion remains.

Architectures of the Future

- William Gibson: *The future is here -- it's just not evenly distributed yet*. True of today. Were not there yet.
- The **Next Big Idea**: all these options are on a sliding scale and you can choose what you want based on the qualities and features you need in your system. SQL vs NoSQL is an illusion. Think of the space as spectrum, where you make choices based on requirements.
- Some of your world may be as Eric Brewer says: *partitioned* + *asynchronous* which implies an architecture that's *weakly consistent* + *delayed exceptions* + *compensation*. Get charged twice and your account is credited. Overbook an airplane and compensate the passengers. Duplicate detection is the delayed exception.
- Some of your world may be ACID where transactions matter. All options are valid based on your application.

The Future is...Polyglot

When asked if Facebook intended to standardize on a single database platform Facebook's Director of Engineering Andrew Bosworth responded...

For the time being, the company intends to use separate platforms for separate tasks. With Facebook's technology stack in general, we've really tried to use the right technology for the problem we're solving. You can get into trouble over-standardizing the technology.

The Future is...Cloud

- Let's assume for a moment you can't build and run your own datacenter...
- The place you can plug your polyglot systems into and expect low operations cost, elastic resource use, advanced multi-datacenter and security infrastructure, and advanced scalable services, no-brainer ease of use is...the cloud.
- Quality & power/bandwidth/cage costs better in colo.
- Look for products that can dynamically scale up and down automatically. Traditional databases do not work this way and is a major leverage point for small teams developing big systems.
- Allows deferring capacity planning by using a scalable architecture and scalable software from the start.
- Clouds can scale-up and scale-out.
- Should work across multiple availability zones.

The Future is...Service Oriented

- It's not all wonderful. Integrating systems across transaction boundaries is a problem. Manually queue, retry, eventually consistent, but not great.
- For protection many sites use a service oriented architecture: Amazon, Playfish, Twitter, The Case Against ORM, Google App Engine
- After HTTP terminates, all applications tend to look a like. A big change from the two and three tier days.
- Loose coupling technology dependencies don't leak through and services can be developed, managed, scaled, deployed, and tuned independently. Creates separate failure domains.
- Organize your internal systems to be automated, service-driven and API-driven.

Services Have Taken Over the World

- Google services: OAuth, User Service, Calendar, Map, Contacts, Document Handling, Videos, Photo, Spreadsheets, Mail, Data Mining, etc.
- Google App Engine services: Memcache, URL Fetch, Mail, XMPP, Images, Google Accounts, Task Queues, Blobstore, Channel
- Amazon Web Services: EC2, EMR, Auto Scaling, Cloud Front, SimpleDB, RDS, FWS, SWS, SNS, CloudWatch, DNS, VPM, ELB, DevPay, S3, EBS, Mechanical Turk
- ProgrammableWeb has a big list of APIs. Twitter, Facebook, Queuing, Lucene, Solr, SimpleGeo, Twilio, Flickr, Foursqure
- Use EC2 for videos instead of wedging it into GAE.
- Building scalability by composing your application from other scalable services. This is how it works now.

But I Don't Need to Scale

- Most common reason for standing pat is saying that "I'll never need to scale so why bother? We aren't Twitter or Facebook or Google after all."
- From Tumblr:

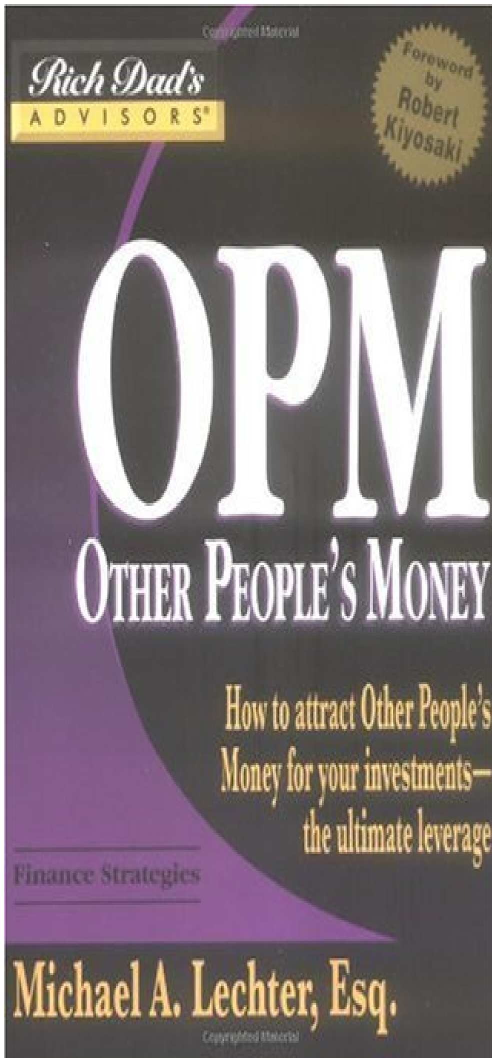
Frankly, keeping up with growth has presented more work than our small team was prepared for — with traffic now climbing more than 500M pageviews each month. But we are determined and focused on bringing our infrastructure well ahead of capacity as quickly as possible. We've nearly quadrupled our engineering team this month alone, and continue to distribute and enhance our architecture to be more resilient to failures like today's.

- What happens when you need to cross the scalability chasm? Do you want to completely change your architecture or evolve from something that was meant to scale?

Leveraging Other People's Scale (LOPS)

- **Social changes everything**. Common business strategy is to leverage off of other people's scale in the form of social networks or data feeds.
- **App stores**. Distribution has totally changed. Your app can be placed immediately in-front of millions of people. I remember shlepping shrink wrapped software around to conferences.
- **Fame**. One of my most read posts had absolutely nothing to do with me. I used "Kevin Rose" in the title. And they came.
- **Population**. There are always more people and things being added to the potential user base.
- Zynga and Playfish needed to scale to hundreds of millions of users quickly because they were LOPSing.

Leveraging Other People's Data (LOPD)



- **Crowdsourcing** as a source of scale. Letting users "help you" by adding their own data to your system can quickly turn into a shockingly massive load.
    - Flickr with 3,000 photos uploaded every minute
    - Facebook adds 12 terabytes per day
    - Twitter adds 7 terabytes a day
- **Freemium business model**. When you give away the milk a lot of people will want milkshakes.

The 3 Big Bucket Model of Systems

- Previously the expensive relational database was tasked with doing everything. We're now seeing people move away from the relational database as the central datastore of record.
- Dwight Merriman, 10gen CEO of MongoDB fame, thinks there will be 3 big buckets of systems:
  1. **Analytics Processing** - complex offline ad-hoc reporting
  2. **OLTP** - complex transactional semantics
  3. **NoSQL** - mostly online processing, agile, high performance, horizontally scalable.
- No one product is best at all three, so systems will tend to divide up this way. Makes sense.

Main Data Models
Adapted from Emil Eifrem. NoSQL databases.

Document Databases
Lineage: Inspired by Lotus Notes.
Data model: Collections of documents, which contain key-value collections.
Example: CouchDB, MongoDB

Graph Databases
Lineage: Euler and graph theory.
Data model: Nodes & relationships, both which can hold key-value pairs
Example: AllegroGraph, InfoGrid, Neo4j

Relational Databases
Lineage: E. F. Codd in A Relational Model of Data for Large Shared Data Banks
Data Model: a set of relations
Example: VoltDB, Clustrix, MySQL

Object Oriented Databases
Lineage: Graph Database Research
Data Model: Objects
Example: Objectivity, Gemstone

Key-Value Stores
Lineage: Amazon's Dynamo paper and Distributed HashTables.
Data model: A global collection of KV pairs.
Example: Membase, Riak

BigTable Clones
Lineage: Google's BigTable paper.
Data model: Column family, i.e. a tabular model where each row at least in theory can have an individual configuration of columns.
Example: HBase, Hypertable, Cassandra

Data Structure Servers
Lineage: ?
Example: Redis
Data model: Operations over dictionaries, lists, sets and string values.

Grid Databases
Lineage: Data Grid and Tuple Space research.
Data Model: Space Based Architecture
Example: GigaSpaces, Coherence

Data Models are Good at?

- Document Databases: Natural data modeling. Programmer friendly. Rapid development. Web friendly, CRUD.
- Key-value Stores: Handles size well. Processing a constant stream of small reads and writes. Fast. Programmer friendly.
- BigTable Clones: Handles size well. Stream massive write loads. High availability. Multiple-data centers. MapReduce.
- Relational Databases: High performing, scalable OLTP. SQL access. Materialized views. Transactions matter. Programmer friendly transactions.
- Data Structure Servers: Quirky stuff you never thought of using a database for before.
- Graph Databases: Rock complicated graph problems. Fast.
- Grid Databases: High performance and scalable transaction processing.

Use Cases Drive Decisions

If your application needs...

- complex transactions because you can't afford to lose data or if you would like a simple transaction programming model then look at a Relational or Grid database.
  - Example: an inventory system that might want full ACID. I was very unhappy when I bought a product and they said later they were out of stock. I did not want a compensated transaction. I wanted my item!
- to scale then NoSQL or SQL can work. Look for systems that support scale-out, partitioning, live addition and removal of machines, load balancing, automatic sharding and reblancing, and fault tolerance.
- to always be able to write to a database because you need high availability then look at Bigtable Clones which feature eventual consistency.
- to handle lots of small continuous reads and writes, that may be volatile, then look at Document or Key-value or databases offering fast in-memory access.
- to implement social network operations then you first may want a Graph database or second, a database like Riak that supports relationships. An in-memory relational database with simple SQL joins might suffice for small data sets. Redis's set and list operations could work too.

Use Cases...2

If your application needs...

- to operate over a wide variety of access patterns and data types then look at a Document database, they generally are flexible and perform well.
- powerful offline reporting with large datasets then look at Hadoop first and second, products that support MapReduce. Supporting MapReduce isn't the same as being good at it.
- to span multiple data-centers then look at Bigtable Clones and other products that offer a distributed option that can handle the long latencies and are partition tolerant.
- to build CRUD apps then look at a Document database, they make it easy to access complex data without joins.
- built-in search then look at Riak.
- to operate on data structures like lists, sets, queues, publish-subscribe then look at Redis. Useful for distributed locking, capped logs, and a lot more.
- programmer friendliness in the form of programmer friendly data types like JSON, HTTP, REST, Javascript then first look at Document databases and then Key-value Databases.

Use Cases...3

If your application needs...

- transactions combined with materialized views for real-time data feeds then look at VoltDB. Great for data-rollups and time windowing.
- enterprise level support and SLAs then look for a product that makes a point of catering to that market. Membase is an example.
- to log continuous streams of data that may have no consistency guarantees necessary at all then look at Bigtable Clones because they generally work on distributed file systems that can handle a lot of writes.
- to be as simple as possible to operate then look for a hosted or PaaS solution because they will do all the work for you.
- to be sold to enterprise customers then consider a Relational Database because they are used to relational technology.
- to dynamically build relationships between objects that have dynamic properties then consider a Graph Database because often they will not require a schema and models can be built incrementally through programming.
- to support large media then look storage services like S3. NoSQL systems tend not to handle large BLOBS, though MongoDB has a file service.

Use Cases...4

If your application needs...

- to bulk upload lots of data quickly and efficiently then look for a product supports that scenario. Most will not because they don't support bulk operations.
- an easier upgrade path then use a fluid schema system like a Document Database or a Key-value Database because it supports optional fields, adding fields, and field deletions without the need to build an entire schema migration framework.
- to implement integrity constraints then pick a database that support SQL DDL, implement them in stored procedures, or implement them in application code.
- a very deep join depth the use a Graph Database because they support blisteringly fast navigation between entities.
- to move behavior close to the data so the data doesn't have to be moved over the network then look at stored procedures of one kind or another. These can be found in Relational, Grid, Document, and even Key-value databases.

Use Cases...5

If your application needs...

- to cache or store BLOB data then look at a Key-value store. Caching can for bits of web pages, or to save complex objects that were expensive to join in a relational database, to reduce latency, and so on.
- a proven track record like not corrupting data and just generally working then pick an established product and when you hit scaling (or other issues) use on of the common workarounds (scale-up, tuning, memcached, sharding, denormalization, etc).
- fluid data types because your data isn't tabular in nature, or requires a flexible number of columns, or has a complex structure, or varies by user (or whatever), then look at Document, Key-value, and Bigtable Clone databases. Each has a lot of flexibility in their data types.
- other business units to run quick relational queries so you don't have to reimplement everything then use a database that supports SQL.
- to operate in the cloud and automatically take full advantage of cloud features then we may not be there yet.

Use Cases...6

If your application needs...

- support for secondary indexes so you can look up data by different keys then look at relational databases and Cassandra's new secondary index support.
- creates an ever-growing set of data that rarely gets accessed then look at Bigtable Clone which will spread the data over a distributed file system.
- to integrate with other services then check if the database provides some sort of write-behind syncing feature so you can capture database changes and feed them into other systems to ensure consistency.
- fault tolerance check how durable writes are in the face power failures, partitions, and other failure scenarios.
- to push the technological envelope in a direction nobody seems to be going then build it yourself because that's what it takes to be great sometimes.

What should your application use?

- Key point is to rethink how your application could work differently in terms of the different data models and the different products. Right data model for the right problem.
- To see what models might help your application take a look at What The Heck Are You Actually Using NoSQL For? In this article I tried to pull together a lot of use cases of the different qualities and features developers have used in building systems.
- Match what you need to do with these use cases. From there you can backtrack to the products you may want to include in your architecture. NoSQL, SQL, it doesn't matter.
- Look at Data Model + Product Features + Your Situation. Products have such different feature sets it's almost impossible to recommend by pure data model alone.
- Which option is best is determined by your priorities.

Experiment Like MythBusters

- Every feature and product is like a myth.
- It must be proven through experiment, thought, and data.
- Don't scramble to implement something in a production environment. Figure out what you need to do. Decide by doing some prototypes. Test. Evaluate. See which solutions fit your architecture.
- MythBusters either plans elaborately or just does it. They'll build scale models, mockups, elaborate props, talk to experts, research, run through a progression.
    - In the end every myth is: busted, plausible, confirmed
    - It's one thing to do it small-scale, it's another to do it large-scale.
    - It wouldn't be MythBusters if it worked the first time.
    - When we experiment and things fail we start to ask why and that's when we learn.

Some Experiments to Try

- Slice off part of a service that may need better performance or scalability onto its own system. For example, the user login subsystem may need to be high performance and this feature could use a dedicated service to meet those goals.
- Turn one of your features into a service. Proceed one-by-one until done.
- Take a feature on your schedule and implement it with a different stack.
- Think about how your code might be refactored if it was rewritten using features from various products.
- Project out your next bottleneck or pain point and think how it might be solved differently.
- Will a two tier approach work? Low latency data is served through a fast interface, but the data itself can be calculated and updated by high latency apps.

Make the Choice Faster

If the previous slides send you into analysis paralysis, I really like this as an antidote, from a Andrei, a commentor on one of my posts:

If you keep going back and forth between upsides and downsides of each choice you'll waste a lot of time for nothing. Start working with one solution (SQL, NoSql, or both) faster rather than later and you'll move towards the best alternative in time as you solve your problems. It's a process!

Scale-Up as Long as You Can

- Best examples are StackOverflow and PlentyOfFish.
- Though there are a lot of reasons to go NoSQL other than scale, it's still easier to use one machine. A relational database can scale amazingly on the mega-hardware we have today.
- When your data needs won't fit on a single machine anymore then you have choices to make about how to span machines.
- At some point the point of staying with what you have is greater than the pain of learning something new.

You Don't Have to Make Excuses for Choosing Something Different

- RDBMSs are the default way to solve database problems. If you do anything different you must pass a quiz of the 99 things you must have tried to get your RDBMS to scale. And you can never pass the quiz.
- Hitting a limit is seen as your failure. You don't have skillz. Is your schema correct? Denormalized, but not too much? Queries optimized? Indexes optimized? Did you hire a DBA? Did you size your hardware properly? Use a better database?
- Most scaling problems can be solved with money that you may not have.
- Do your homework, run tests, pick what you want, and have a plan B.

Free Yourself from Feeling Guilt Over Using What You Know

- Just because Facebook, Google, Twitter, etc do something doesn't mean you need to, too.
- The people in these organization are just people, trying to solve a problem, with certain resources, certain requirements, and certain quirks.
- They may know something you don't, but then again maybe they don't. Your experience, research, and knowledge is just as valid.
- Forget what's cool. Focus on the end product and how to deliver it and keep on delivering it.

Go With the Strengths of Your Team

Michael Westen, Burn Notice:

Special forces squads are built around the skills of the individual members. But no matter how good each member of the squad is, every mission comes down to one thing: how well they work together. Because in the end you don't need a hero to succeed in the field you need a team.

- You have to go with the strengths of your team unless you are prepared for a transition period. If they know a technology it may be safer to go with that. Manage risk. This is a big reason some don't go with NoSQL.
- One team that started with Erlang and moved to Java so they could find programmers. Think about those scenarios.

Scaling Requires Making Tradeoffs

- With scalable systems you may notice that you can't open up a transaction, update 10 different tables, hundreds of records, and expect it all to just work. Not that simple.
- Nope, like a good poem, these systems require some constraints be followed so they can operate at scale.
- In GAE, you have task queues for long running jobs, there are numerous quotas and queries can only be so expressive.
- In KV stores you can only update one K in a transaction.
- Most products don't support secondary indexes.
- Availability may require the programmer to implement consistency with read-repair and compensating transactions.
- In relational systems you'll need to partition correctly.
- These are all part of it. There are tradeoffs.
- You are right, this is still too hard.

If You Love New York Take I-30 East

- From Why We Make Mistakes. This was a bumper sticker seen in Texas.
- The meaning is when people undergo major changes, like moving, one their biggest mistakes is not changing how they use their time.
- In other words, if you move to Texas learn to enjoy the things Texas has to offer. Don't move there expecting to find a great bagel as you would in NY, or great beaches as you would in L.A.
- Learn to love the rodeo or the Dallas Cowboys or the vast open spaces of Texas--or else you will be miserable.
- Same applies when switching database too. Really learn how these things work and change to make the best of them.

Related Articles

Please see the reference list at the end of
What The Heck Are You Actually Using NoSQL For?

Any Questions?

Support Slides

We won't talk about these, but you may find them useful.

Where are you starting from?

- Greenfield application?
- In the middle of a project and worried about hitting bottlenecks?
- Worried about hitting the scaling wall once you deploy?
- Adding a separate loosely coupled service to an existing system?
- What are your resources? expertise? budget?
- What are your pain points? What's so important that if it fails you will fail? What forces are pushing you?
- What are your priorities? Prioritize them. What is really important to you, what must get done?
- What are your risks? Prioritize them. Is the risk of being unavailable more important than being inconsistent?

What are you trying to accomplish?

- What are you trying to accomplish?
- What's the delivery schedule?
- Do the research to be specific, like Facebook did with their messaging system:

Facebook chose HBase because they monitored their usage and figured out what was needed: a system that could handle two types of data patterns.

1. A short set of temporal data that tends to be volatile
2. An ever-growing set of data that rarely gets accessed

Things to Consider...Your Problem

- Do you need to build a custom system?
- Access patterns: 1) A short set of temporal data that tends to be volatile 2) An ever-growing set of data that rarely gets accessed 3) High write loads 4) High throughput, 5) Sequential, 6) Random
- Requires scalability?
- Is availability more important than consistency, or is it latency, transactions, durability, performance, or ease of use?
- Cloud or colo? Hosted services? Resources like disk space?
- Can you find people who know the stack?
- Tired of the data transformation (ORM) treadmill?
- Store data that can be accessed quickly and is used often?
- Would like a high level interface like PaaS?

Things to Consider...Money

- Cost? With money you have different options than if you don't. You can probably make the technologies you know best scale.
- Inexpensive scaling?
- Lower operations cost?
- No sysadmins?
- Type of license?
- Support costs?

Things to Consider...Programming

- Flexible datatypes and schemas?
- Support for which language bindings?
- Web support: JSON, REST, HTTP, JSON-RPC
- Built-in stored procedure support? Javascript?
- Platform support: mobile, workstation, cloud
- Transaction support: key-value, distributed, ACID, BASE, eventual consistency, multi-object ACID transactions.
- Datatype support: graph, key-value, row, column, JSON, document, references, relationships, advanced data structures, large BLOBs.
- Prefer the simplicity of transaction model where you can just update and be done with it? In-memory makes it fast enough and big systems can fit on just a few nodes.

Things to Consider...Performance

- Performance metrics: IOPS/sec, reads, writes, streaming?
- Support for your access pattern: random read/write; sequential read/write; large or small or whatever chunk size you use.
- Are you storing frequently updated bits of data?
- High Concurrency vs High Performance?
- Problems that limit the type of work load you care about?
- Peak QPS on highly-concurrent workloads?
- Test your specific scenarios?

Things to Consider...Features

- Spooky scalability at a distance: support across multiple data-centers?
- Ease of installation, configuration, operations, development, deployment, support, manage, upgrade, etc.
- Data Integrity: In DDL, Stored Procedure, or App
- Persistence design: Memtable/SSTable; Apend-only B-tree; B-tree; On-disk linked lists; In-memory replicated; In-memory snapshots; In-memory only; Hash; Pluggable.
- Schema support: none, rigid, optional, mixed
- Storage model: embedded, client/server, distributed, in-memory
- Support for search, secondary indexes, range queries, ad-hoc queries, MapReduce?
- Hitless upgrades?

Things to Consider...More Features

- Tunability of consistency models?
- Tools availability and product maturity?
- Expand rapidly? Develop rapidly? Change rapidly?
- Durability? On power failure?
- Bulk import? Export?
- Hitless upgrades?
- Materialized views for rollups of attributes?
- Built-in web server support?
- Authentication, authorization, validation?
- Continuous write-behind for system sync?
- What is the story for availability, data-loss prevention, backup and restore?
- Automatic load balancing, partitioning, and repartitioning?
- Live addition and removal of machines?

Things to Consider...The Vendor

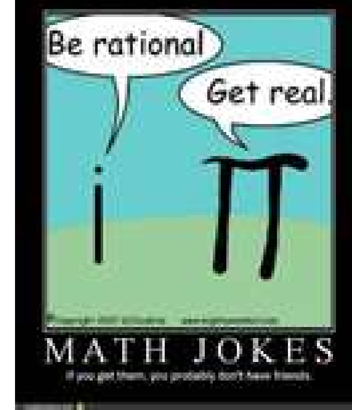
- Viability of the company?
- Future direction?
- Community and support list quality?
- Support responsiveness?
- How do they handle disasters?
- Quality and quantity of partnerships developed?
- Customer support: enterprise-level SLA, paid support, none

Size Matters Not -- Yoda

- Scalability, handling large data volumes may have been the original motivation for NoSQL systems. Like 7TB a day for Twitter. Not the only motivation anymore.

- NoSQL or SQL isn't just about scaling. It's about distributed architectures, reduce complexity via rich data models that more easily represent a domain. Or "it does what you need doing."
- More than one machine means splitting data and worrying about consistency. Leads to 2PC or quorums, or just writing a complex value, which loses support for references and data integrity, causes things like last update wins, vector clocks for read repair, and gossip protocols.

There's Truth in Humor



Let's start with something a little fun, yet educational...

- Hilarious Relational Database Vs NoSQL Fanbois by Garrett Smith (NSFW)
  - Oh so funny...classic worse is better argument. Peter Gabriel: *It will take much less time and effort to implement initially and it will be easier to adapt to new situations. Porting becomes far easier. Thus its use will spread rapidly. Once spread, there's pressure to improve its functionality, but users have already been conditioned to accept "worse" rather than the "right thing". Therefore, the worse-is-better software first will gain acceptance, second will condition its users to expect less, and third will be improved to a point that is almost the right thing.*
- Hilarious Fault-Tolerance Cartoon by John Muellerleile (NSFW)
  - John is from Riak and this cartoon was based on their actual experience. When standard, well-worn ways change a lot it can be disorienting.
- Flow Chart For Project Decision Making by Anonymous (NSFW)
  - If it's not broke don't fix it. Rewriting rarely goes well. This was Twitter's choice with Cassandra for Tweet storage.
- Everything's Amazing and Nobody's Happy by Louis CK

EVERYTHING
IS AMAZING
AND NOBODY
IS HAPPY

- Love this interview. We live an amazing, amazing world. We are using high speed internet from a plane! While it's flying through the air! You are sitting in a chair in the sky!
- We tend to see all this confusion in the market place and get anxious.
- Things were simpler when there was one way to do things.
- But we can do so much more with less today than ever before. A few people can do now in half a year and $150K what it took a team of 20 people a year and $1.5 million.
- There's so much energy and excitement and learning.
- Thinking back to some of my old projects I can see how they would be totally different today, in a good way.

Why So Much SQL on NoSQL Hate?

- Everything is amazing, then why are people so mean?
- Look at the flame wars of SQL vs NoSQL
- A guy writes about his experiences with GAE and he gets hammered in the comments. Really?
- Comments like:
  - "But your reasoning is just lame. Get better coders."
  - "Is english your 2nd language?"
  - "You obviously are not a very good programmer or craftsman for that matter because both craftsman and programmer know how to use their tools before starting a project. "
- Nothing really serious is at stake. Chill.

The Future is Looking a Little Biological

- Our body has multiple drives. You may think we have a drive to procreate for example, but we don't. What we have:
  - **sex drive** - craving for sexual satisfaction, activity of testosterone, get you out there to find a range of partners, find a mate
  - **romantic love** - being in love, focussed attention, intense energy, highly motivated to win individual, associated with dopamine and norephedrine (stimulant) and low levels of seratonin, lasts about a year, stays focussed enough to get the person, form pair bond
  - **attachment** - companion at love, calm security for long term partner, associated with oxytocin and vasopresin, tolerate each other at least long enough to rear a child, rear children as a team
- And these three brain systems work in a mix and match way to serve many of the demands of reproduction.
- The brain keeps a rich representation of the living organism and through a nervous system manages life and regulates the individual.
- Homeostatic mechanisms keep the systems in balance.
- Looks like adaptive service architectures in the cloud.

Which is Better?

- Moving for a 25% improvement is probably not a reason to go NoSQL.
- Benchmark relevancy depends on the use case. Does it match your situation(s)?
- Are you a startup that needs to release a product as soon as possible and you are playing around with ideas? Both SQL and NoSQL can make an argument.
- Performance may be equal on one box, but what happens when you need N?
- Everything has problems, if you look at Amazon forums it's EBS is slow, or my instances won't reply, etc. GAE it's the datastore is slow or X. Every product which people are using will have problems.

Wanting Victories

- I'm watching a course on the US Civil War and one of the most impressive things I've learned is how Lincoln deftly handled his generals. George McClellan and Joseph Hooker, who clearly¬†disagreed¬†with him and were even against him. Lincoln didn't fire them for that. He fired them when they didn't produce victories. Lincoln wanted¬†victories above all else because that's what it took to win a war. So he managed them anyway and pressed on.
- This is kind of how I think of building systems. Not everything is perfect or clean, but we want victories.