What the heck are you actually using NoSQL for?

Monday, December 6, 2010 at 9:34AM

Todd Hoff in nosql

It's a truism that we should choose the *right tool for the job*. Everyone says that. And who can disagree? The problem is this is not helpful advice without being able to answer more specific questions like: What jobs are the tools good at? Will they work on jobs like mine? Is it worth the risk to try something new when all my people know something else and we have a deadline to meet? How can I make all the tools work together?

In the NoSQL space this kind of real-world data is still a bit vague. When asked, vendors tend to give very general answers like NoSQL is good for BigData or key-value access. What does that mean for for the developer in the trenches faced with the task of solving a specific problem and there are a dozen confusing choices and no obvious winner? Not a lot. It's often hard to take that next step and imagine how their specific problems could be solved in a way that's worth taking the trouble and risk.

Let's change that. What problems are you using NoSQL to solve? Which product are you using? How is it helping you? Yes, this is part the research for my webinar on December 14th, but I'm a huge believer that people learn best by example, so if we can come up with real specific examples I think that will really help people visualize how they can make the best use of all these new product choices in their own systems.

Here's a list of uses cases I came up with after some trolling of the interwebs. The sources are so varied I can't attribute every one, I'll put a

list at the end of the post. Please feel free to add your own. I separated the use cases out for a few specific products simply because I had a lot of uses cases for them they were clearer out on their own. This is not meant as an endorsement of any sort. Here's a master list of all the NoSQL products. If you would like to provide a specific set of use cases for a product I'd be more than happy to add that in.

General Use Cases

These are the general kinds of reasons people throw around for using NoSQL. Probably nothing all that surprising here.

- **Bigness.** NoSQL is seen as a key part of a new data stack supporting: big data, big numbers of users, big numbers of computers, big supply chains, big science, and so on. When something becomes so massive that it must become massively distributed, NoSQL is there, though not all NoSQL systems are targeting big. Bigness can be across many different dimensions, not just using a lot of disk space.
- **Massive write performance**. This is probably the canonical usage based on Google's influence. High volume. Facebook needs to store 135 billion messages a month. Twitter, for example, has the problem of storing 7 TB/data per day with the prospect of this requirement doubling multiple times per year. This is the data is too big to fit on one node problem. At 80 MB/s it takes a day to store 7TB so writes need to be distributed over a cluster, which implies key-value access, MapReduce, replication, fault tolerance, consistency issues, and all the rest. For faster writes in-memory systems can be used.
- **Fast key-value access**. This is probably the second most cited virtue of NoSQL in the general mind set. When latency is important it's hard to beat hashing on a key and reading the value directly from

memory or in as little as one disk seek. Not every NoSQL product is about fast access, some are more about reliability, for example. but what people have wanted for a long time was a better memcached and many NoSQL systems offer that.

- **Flexible schema** and **flexible datatypes**. NoSQL products support a whole range of new data types, and this is a major area of innovation in NoSQL. We have: column-oriented, graph, advanced data structures, document-oriented, and key-value. Complex objects can be easily stored without a lot of mapping. Developers love avoiding complex schemas and ORM frameworks. Lack of structure allows for much more flexibility. We also have program and programmer friendly compatible datatypes likes JSON.
- **Schema migration**. Schemalessness makes it easier to deal with schema migrations without so much worrying. Schemas are in a sense dynamic, because they are imposed by the application at run-time, so different parts of an application can have a different view of the schema.
- **Write availability**. Do your writes need to succeed no mater what? Then we can get into partitioning, CAP, eventual consistency and all that jazz.
- **Easier maintainability, administration and operations**. This is very product specific, but many NoSQL vendors are trying to gain adoption by making it easy for developers to adopt them. They are spending a lot of effort on ease of use, minimal administration, and automated operations. This can lead to lower operations costs as special code doesn't have to be written to scale a system that was never intended to be used that way.
- **No single point of failure**. Not every product is delivering on this, but we are seeing a definite convergence on relatively easy to configure and manage high availability with automatic load balancing and cluster sizing. A perfect cloud partner.
- **Generally available parallel computing.** We are seeing

MapReduce baked into products, which makes parallel computing something that will be a normal part of development in the future.

- **Programmer ease of use**. Accessing your data should be easy. While the relational model is intuitive for end users, like accountants, it's not very intuitive for developers. Programmers grok keys, values, JSON, Javascript stored procedures, HTTP, and so on. NoSQL is for programmers. This is a developer led coup. The response to a database problem can't always be to hire a really knowledgeable DBA, get your schema right, denormalize a little, etc., programmers would prefer a system that they can make work for themselves. It shouldn't be so hard to make a product perform. Money is part of the issue. If it costs a lot to scale a product then won't you go with the cheaper product, that you control, that's easier to use, and that's easier to scale?
- **Use the right data model for the right problem**. Different data models are used to solve different problems. Much effort has been put into, for example, wedging graph operations into a relational model, but it doesn't work. Isn't it better to solve a graph problem in a graph database? We are now seeing a general strategy of trying find the best fit between a problem and solution.
- Avoid hitting the wall. Many projects hit some type of wall in their project. They've exhausted all options to make their system scale or perform properly and are wondering what next? It's comforting to select a product and an approach that can jump over the wall by linearly scaling using incrementally added resources. At one time this wasn't possible. It took custom built everything, but that's changed. We are now seeing usable out-of-the-box products that a project can readily adopt.
- Distributed systems support. Not everyone is worried about scale or performance over and above that which can be achieved by non-NoSQL systems. What they need is a distributed system that can span datacenters while handling failure scenarios without a hiccup.

NoSQL systems, because they have focussed on scale, tend to exploit partitions, tend not use heavy strict consistency protocols, and so are well positioned to operate in distributed scenarios.

- Tunable CAP tradeoffs. NoSQL systems are generally the only products with a "slider" for choosing where they want to land on the CAP spectrum. Relational databases pick strong consistency which means they can't tolerate a partition failure. In the end this is a business decision and should be decided on a case by case basis. Does your app even care about consistency? Are a few drops OK? Does your app need strong or weak consistency? Is availability more important or is consistency? Will being down be more costly than being wrong? It's nice to have products that give you a choice.

More Specific Use Cases

- Managing large streams of non-transactional data: Apache logs, application logs, MySQL logs, clickstreams, etc.
- Syncing online and offline data. This is a niche CouchDB has targeted.
- Fast response times under all loads.
- Avoiding heavy joins for when the query load for complex joins become too large for a RDBMS.
- Soft real-time systems where low latency is critical. Games are one example.
- Applications where a wide variety of different write, read, query, and consistency patterns need to be supported. There are systems optimized for 50% reads 50% writes, 95% writes, or 95% reads. Read-only applications needing extreme speed and resiliency, simple queries, and can tolerate slightly stale data. Applications requiring moderate performance, read/write access, simple queries, completely authoritative data. Read-only application which complex query requirements.

- Load balance to accommodate data and usage concentrations and to help keep microprocessors busy.
- Real-time inserts, updates, and queries.
- Hierarchical data like threaded discussions and parts explosion.
- Dynamic table creation.
- Two tier applications where low latency data is made available through a fast NoSQL interface, but the data itself can be calculated and updated by high latency Hadoop apps or other low priority apps.
- Sequential data reading. The right underlying data storage model needs to be selected. A B-tree may not be the best model for sequential reads.
- Slicing off part of service that may need better performance/scalability onto it's own system. For example, user logins may need to be high performance and this feature could use a dedicated service to meet those goals.
- Caching. A high performance caching tier for web sites and other applications. Example is a cache for the Data Aggregation System used by the Large Hadron Collider.
- Voting.
- Real-time page view counters.
- User registration, profile, and session data.
- Document, catalog management and content management systems. These are facilitated by the ability to store complex documents has a whole rather than organized as relational tables. Similar logic applies to inventory, shopping carts, and other structured data types.
- Archiving. Storing a large continual stream of data that is still accessible on-line. Document-oriented databases with a flexible schema that can handle schema changes over time.
- Analytics. Use MapReduce, Hive, or Pig to perform analytical queries and scale-out systems that support high write loads.
- Working with heterogenous types of data, for example, different media types at a generic level.
- Embedded systems. They don’t want the overhead of SQL and

servers, so they uses something simpler for storage.

- A "market" game, where you own buildings in a town. You want the building list of someone to pop up quickly, so you partition on the owner column of the building table, so that the select is single-partitioned. But when someone buys the building of someone else you update the owner column along with price.
- JPL is using SimpleDB to store rover plan attributes. References are kept to a full plan blob in S3.
- Federal law enforcement agencies tracking Americans in real-time using credit cards, loyalty cards and travel reservations.
- Fraud detection by comparing transactions to known patterns in real-time.
- Helping diagnose the typology of tumors by integrating the history of every patient.
- In-memory database for high update situations, like a web site that displays everyone's "last active" time (for chat maybe). If users are performing some activity once every 30 sec, then you will be pretty much be at your limit with about 5000 simultaneous users.
- Handling lower-frequency multi-partition queries using materialized views while continuing to process high-frequency streaming data.
- Priority queues.
- Running calculations on cached data, using a program friendly interface, without have to go through an ORM.
- Unique a large dataset using simple key-value columns.
- To keep querying fast, values can be rolled-up into different time slices.
- Computing the intersection of two massive sets, where a join would be too slow.
- A timeline ala Twitter.

Redis Use Cases

Redis is unique in the repertoire as it is a data structure server, with many fascinating use cases that people are excited to share.

- Calculating whose friends are online using sets.
- Memcached on steroids.
- Distributed lock manager for process coordination.
- Full text inverted index lookups.
- Tag clouds.
- Leaderboards. Sorted sets for maintaining high score tables.
- Circular log buffers.
- Database for university course availability information. If the set contains the course ID it has an open seat. Data is scraped and processed continuously and there are ~7200 courses.
- Server for backed sessions. A random cookie value which is then associated with a larger chunk of serialized data on the server) are a very poor fit for relational databases. They are often created for every visitor, even those who stumble in from Google and then leave, never to return again. They then hang around for weeks taking up valuable database space. They are never queried by anything other than their primary key.
- Fast, atomically incremented counters are a great fit for offering real-time statistics.
- Polling the database every few seconds. Cheap in a key-value store. If you're sharding your data you'll need a central lookup service for quickly determining which shard is being used for a specific user's data. A replicated Redis cluster is a great solution here - GitHub use exactly that to manage sharding their many repositories between different backend file servers.
- Transient data. Any transient data used by your application is also a good fit for Redis. CSRF tokens (to prove a POST submission came from a form you served up, and not a form on a malicious third party site, need to be stored for a short while, as does handshake data for various security protocols.
- Incredibly easy to set up and ridiculously fast (30,000 read or writes

a second on a laptop with the default configuration)

- Share state between processes. Run a long running batch job in one Python interpreter (say loading a few million lines of CSV in to a Redis key/value lookup table) and run another interpreter to play with the data that's already been collected, even as the first process is streaming data in. You can quit and restart my interpreters without losing any data.
- Create heat maps of the BNP's membership list for the Guardian
- Redis semantics map closely to Python native data types, you don't have to think for more than a few seconds about how to represent data.
- That's a simple capped log implementation (similar to a MongoDB capped collection)—push items on to the tail of a 'log' key and use ltrim to only retain the last X items. You could use this to keep track of what a system is doing right now without having to worry about storing ever increasing amounts of logging information.
- An interesting example of an application built on Redis is Hurl, a tool for debugging HTTP requests built in 48 hours by Leah Culver and Chris Wanstrath.
- It's common to use MySQL as the backend for storing and retrieving what are essentially key/value pairs. I've seen this over-and-over when someone needs to maintain a bit of state, session data, counters, small lists, and so on. When MySQL isn't able to keep up with the volume, we often turn to memcached as a write-thru cache. But there's a bit of a mis-match at work here.
- With sets, we can also keep track of ALL of the IDs that have been used for records in the system.
- Quickly pick a random item from a set.
- API limiting. This is a great fit for Redis as a rate limiting check needs to be made for every single API hit, which involves both reading and writing short-lived data.
- A/B testing is another perfect task for Redis - it involves tracking

user behaviour in real-time, making writes for every navigation action a user takes, storing short-lived persistent state and picking random items.
- Implementing the inbox method with Redis is simple: each user gets a queue (a capped queue if you're worried about memory running out) to work as their inbox and a set to keep track of the other users who are following them. Ashton Kutcher has over 5,000,000 followers on Twitter - at 100,000 writes a second it would take less than a minute to fan a message out to all of those inboxes.
- Publish/subscribe is perfect for this broadcast updates (such as election results) to hundreds of thousands of simultaneously connected users. Blocking queue primitives mean message queues without polling.
- Have workers periodically report their load average in to a sorted set.
- Redistribute load. When you want to issue a job, grab the three least loaded workers from the sorted set and pick one of them at random (to avoid the thundering herd problem).
- Multiple GIS indexes.
- Recommendation engine based on relationships.
- Web-of-things data flows.
- Social graph representation.
- Dynamic schemas so schemas don't have to be designed up-front. Building the data model in code, on the fly by adding properties and relationships, dramatically simplifies code.
- Reducing the impedance mismatch because the data model in the database can more closely match the data model in the application.

VoltDB Use Cases

VoltDB as a relational database is not traditionally thought of as in the NoSQL camp, but I feel based on their radical design perspective they are so far away from Oracle type systems that they are much more in the

NoSQL tradition.

- Application: Financial trade monitoring
    1. Data source: Real-time markets
    2. Partition key: Market symbol (ticker, CUSIP, SEDOL, etc.)
    3. High-frequency operations: Write and index all trades, store tick data (bid/ask)
    4. Lower-frequency operations: Find trade order detail based on any of 20+ criteria, show TraderX's positions across all market symbols

- Application: Web bot vulnerability scanning (SaaS application)
    1. Data source: Inbound HTTP requests
    2. Partition key: Customer URL
    3. High-frequency operations: Hit logging, analysis and alerting
    4. Lower-frequency operations: Vulnerability detection, customer reporting

- Application: Online gaming leaderboard
    1. Data source: Online game
    2. Partition key: Game ID
    3. High-frequency operations: Rank scores based on defined intervals and player personal best
    4. Lower-frequency transactions: Leaderboard lookups

- Application: Package tracking (logistics)
    1. Data source: Sensor scan
    2. Partition key: Shipment ID
    3. High-frequency operations: Package location updates
    4. Lower-frequency operations: Package status report (including history), lost package tracking, shipment rerouting

- Application: Ad content serving
    1. Data source: Website or device, user or rule triggered
    2. Partition key: Vendor/ad ID composite
    3. High-frequency operations: Check vendor balance, serve ad (in target device format), update vendor balance
    4. Lower-frequency operations: Report live ad view and click-thru stats by device (vendor-initiated)

- Application: Telephone exchange call detail record (CDR) management
    1. Data source: Call initiation request
    2. Partition key: Caller ID
    3. High-frequency operations: Real-time authorization (based on plan and balance)
    4. Lower-frequency operations: Fraud analysis/detection

- Application: Airline reservation/ticketing
    1. Data source: Customers (web) and airline (web and internal systems)
    2. Partition key: Customer (flight info is replicated)
    3. High-frequency operations: Seat selection (lease system), add/drop seats, baggage check-in
    4. Lower-frequency operations: Seat availability/flight, flight schedule changes, passenger re-bookings on flight cancellations

Analytics Use Cases

Kevin Weil at Twitter is great at providing Hadoop use cases. At Twitter this includes counting big data with standard counts, min, max, std dev; correlating big data with probabilities, covariance, influence; and research on Big data. Hadoop is on the fringe of NoSQL, but it's very useful to see

what kind of problems are being solved with it.

- How many request do we serve each day?
- What is the average latency? 95% latency?
- Grouped by response code: what is the hourly distribution?
- How many searches happen each day at Twitter?
- Where do they come from?
- How many unique queries?
- How many unique users?
- Geographic distribution?
- How does usage differ for mobile users?
- How does usage differ for 3rd party desktop client users?
- Cohort analysis: all users who signed up on the same day—then see how they differ over time.
- Site problems: what goes wrong at the same time?
- Which features get users hooked?
- Which features do successful users use often?
- Search corrections and suggestions (not done now at Twitter, but coming in the feature).
- What can web tell about a user from their tweets?
- What can we tell about you from the tweets of those you follow?
- What can we tell about you from the tweets of your followers?
- What can we tell about you from the ratio of your followers/ following?
- What graph structures lead to successful networks? (Twitter’s graph structure is interesting since it’s not two-way)
- What features get a tweet retweeted?
- When a tweet is retweeted, how deep is the corresponding retweet three?
- Long-term duplicate detection (short term for abuse and stopping spammers)
- Machine learning. About not quite knowing the right questions to

ask at first. How do we cluster users?
- Language detection (contact mobile providers to get SMS deals for users—focusing on the most popular countries at first).
- How can we detect bots and other non-human tweeters?

Poor Use Cases

- **OLTP**. Outside VoltDB, complex multi-object transactions are generally not supported. Programmers are supposed to denormalize, use documents, or use other complex strategies like compensating transactions.
- **Data integrity**. Most of the NoSQL systems rely on applications to enforce data integrity where SQL uses a declarative approach. Relational databases are still the winner for data integrity.
- **Data independence**. Data outlasts applications. In NoSQL applications drive everything about the data. One argument for the relational model is as a repository of facts that can last for the entire lifetime of the enterprise, far past the expected life-time of any individual application.
- **SQL**. If you require SQL then very few NoSQL system will provide a SQL interface, but more systems are starting to provide SQLish interfaces.
- **Ad-hoc queries**. If you need to answer real-time questions about your data that you can’t predict in advance, relational databases are generally still the winner.
- **Complex relationships**. Some NoSQL systems support relationships, but a relational database is still the winner at relating.
- **Maturity and stability**. Relational databases still have the edge here. People are familiar with how they work, what they can do, and have confidence in their reliability. There are also more programmers and toolsets available for relational databases. So when in doubt, this is the road that will be traveled.

Related Articles

- Hacker News and Reddit thread on this Post. More use case suggestions there.
- List of NoSQL Systems
- Pig - infrastructure to support ad-hoc analysis of very large data sets.
- Digging Deeper Into Data With Hadoop By Gary Orenstein
- NoSQL East 2009 - Summary of Day 1 by Eivind Uggedal
- The “NoSQL” approach: struggling to see the benefits by Neil Saunders
- Design Patterns for DistributedNon-Relational Databases by Todd Lipcon.
- The Future Is Big Data in the Cloud By Ping Li
- One size does not fit all: “document stores”, “nosql databases” , ODBMSs by Roberto V. Zicari
- NoSQL is for niches By Dana Blankenhorn.
- MongoDB Use Cases
- MongoDB and Ecommerce by Kyle Banker
- Archiving - a good MongoDB use case?
- Five Reasons to Use NoSQL by Jeremiah Peschka
- The Business Case for NoSQL, NoETL and NoProblems by Loraine Lawson
- Is It Time For NoETL? by Seth Grimes
- Holy Large Hadron Collider, Batman!
- I Can't Wait for NoSQL to Die by Ted Dziuba.
- NoSQL Basics, Benefits and Best-Fit Scenarios by Curt Monash
- Redis Tutorial by Simon Willison
- Why I Like Redis by Simon Willison
- Redis: Lightweight key/value Store That Goes the Extra Mile by Jeremy Zawodny

- Remote Dictionary Server
- Is NoSQL for me? I'm just a small fish by Hadi Hariri
- Why Big Enterprises are Interested in NoSQL by Jon Moore
- Visual Guide to NoSQL Systems by Nathan Hurst
- You Can't Sacrifice Partition Tolerance by Coda Hale
- NoSQL Netflix Use Case Comparison by Adrian Cockcroft
- Comparison guide of horizontally scalable datastores by Rick Cattell
- Weak Consistency and CAP Implications by Ilya Grigorik
- Schema-Free MySQL vs NoSQL by Ilya Grigorik
- Neo4j - 5 Cool Graph Examples
- Has anyone used Graph-based Databases
- Neotechnology Uses Cases
- CouchOne Ataxo Case Study
- NOSQL: scaling to size and scaling to complexity by Emil Eifrem
- NoSQL, NoProblem (Not Really… but it's still awesome) by Jeremy Pinkham
- An Expert's Guide to Oracle Technology by Lewis Cunningham
- NoSQL vs SQL, Why Not Both? By Alaric Snell-Pym
- Databases: relational vs object vs graph vs document by On Target
- Use cases are driving the divergence, and the convergence, of NoSQL solutions by James Phillips
- The beginning of the end of NoSQL by Matthew Aslett
- Going NoSQL with MongoDB by Ted Neward
- NoSQL Ecosystem by Jonathan Ellis
- The New Dimension of NoSQL Scalability: Complexity by Alex Popescu
- Quick Reference to Alternative data storages by Alex Popescu
- 6 Reasons Why Relational Database Will Be Superseded by

Robin Bloor

- NoSQL Misconceptions by Ben Scofield
- SQL Databases Don't Scale by Adam Wiggins
- “One Size Fits All”: An Idea Whose Time Has Come and Gone by Michael Stonebraker and U•ur Çetintemel
- Future of RDBMS is RAM Clouds & SSD by Ilya Grigorik
- To scale or not to scale: Key/Value, Document, SQL, JPA by Uri Cohen

---