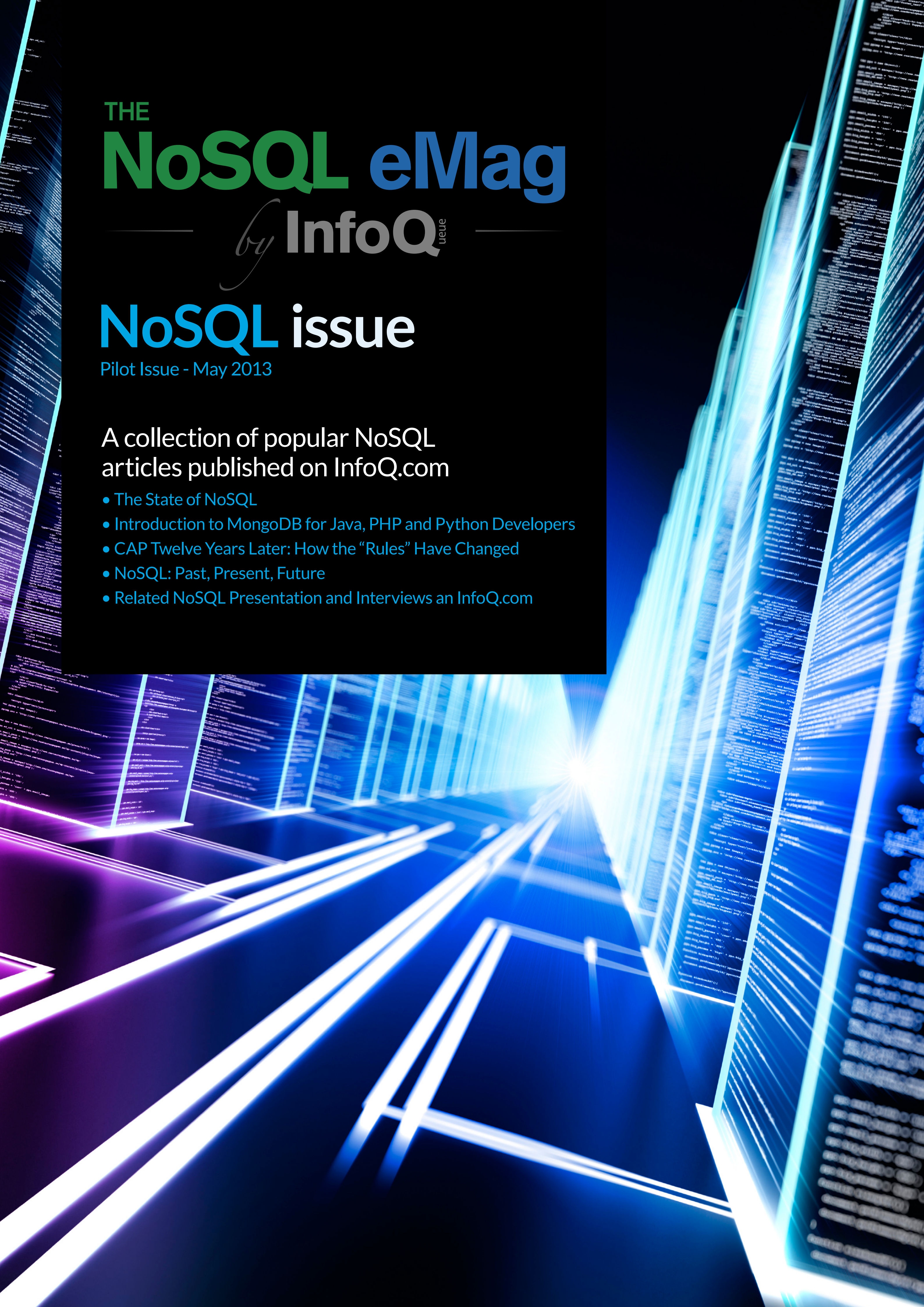
THE NoSQL eMag

by InfoQueue

NoSQL issue

Pilot Issue - May 2013

A collection of popular NoSQL articles published on InfoQ.com



Contents

The State of NoSQL

By Stefan Edlich

After at least four years of tough criticism, it's time to come to an intermediate conclusion about the state of NoSQL. So many things have happened around NoSQL that it is hard to get an overview and value what goals have been achieved and spot where NoSQL failed to deliver.

NoSQL has succeeded in fields throughout industry and academics. Universities are starting to understand that NoSQL must to be covered by the curriculum. It is simply not enough to teach database normalization up and down. This, of course, does not mean that a profound relational foundation is wrong. To the contrary, NoSQL is certainly a perfect and important addition.

"NoSQL solutions will be here to stay"

What happened?

The NoSQL Space has exploded in just five years. The nosql-database.org list numbers about 150 such databases, including some old but still strong dinosaurs like Object Databases. Of course, some interesting mergers have happened, such as the CouchDB and Membase deal that led to Couchbase.

Many people expected the NoSQL space to consolidate but this has not happened. The NoSQL space simply exploded and is still exploding. As with all areas in computer science - like for example with program- ming languages - more and more gaps open to allow for a huge number of databases.

This is all in line with the explosion of the Internet, Big Data, sensors and many more technologies in the future, leading to more data and different requirements for their treatments. In the past four years, we saw only one significant system leaving the stage: the German graph database Sones. The vast majority of NoSQL databases continue to live happily either in the open-source space, without any considerable money turnaround, or in the commercial space.

Visibility and Money

Another important point is visibility and industry adoption. We can see a difference between the old industry protecting its investment, and a new industry of mostly startups. While nearly all of the hot web startups such as Pinterest or Instagram do have a hybrid (SQL + NoSQL) architecture, old industry is still struggling with NoSQL adoption. But we do see that more of these older companies are trying to cut a part of their data streams to be processed and later analyzed with NoSQL solutions like Hadoop, MongoDB, Cassandra, etc.

And this leads to increased demand for developers and architects with NoSQL knowledge. A recent survey showed the following developer skills requested by the industry:

1. HTML5
2. MongoDB
3. iOS
4. Android
5. Mobile Apps
6. Puppet
7. Hadoop
8. jQuery
9. PaaS
10. Social Media

Two of the top ten technology requirements demand experience with NoSQL. And even one before iOS. If this isn't praise, what is?!
NoSQL adoption is going fast and deep.
In a well-known whitepaper in the summer of 2011, Oracle stated that NoSQL feels like an ice-cream flavor that you should not get too attached because it may not be around for too long. Only a few months later, Oracle showed its Hadoop integration into a Big Data Appliance. And even more, we saw the company launch its own NoSQL database, which was a revised BerkeleyDB. Since then, many vendors have been racing to integrate Hadoop. Microsoft, Sybase, IBM and many more already have tight integration. The pattern that can be seen eve-

rywhere: can't fight it, embrace it. One of the strongest signs of broad NoSQL adoption is that NoSQL databases are getting a PaaS standard. Thanks to the easy setup and management of many NoSQL databases, databases like Redis or MongoDB can be seen in dozens of PaaS services like Cloud-Foundry, OpenShift, dotCloud, etc.

As everything moves into the cloud this NoSQL further pressures classic relational databases. Having the choice to select either MySQL/PostGres or MongoDB/Redis, for example, will force companies to think twice about their model and requirements, and will raise other important questions.

An interesting indicator for technologies is the ThoughtWorks radar, which always contains interesting stuff even if you do not fully agree with everything contained within it. Let's have a look at their radar from October 2012 in figure 1:

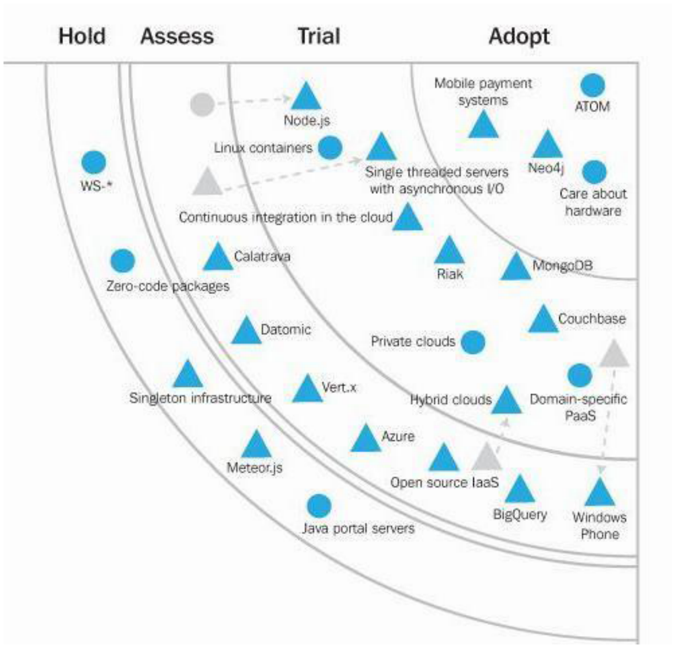


Figure 1: ThoughtWorks Radar, October, 2012

In their platform quadrant they list five databases:

1. Neo4j (adopt)
2. MongoDB (trial but close to adopt)
3. Riak (trial)
4. Couchbase (trial)
5. Datomic (assess)

At least four of these have received a lot of venture capital. If you add up all the venture capital in the entire NoSQL space you will surely count to something in between $100 million and $1 billion! Neo4j received $11 million in a series-B funding. Other companies that received $10-30 million in funding include Aerospike, MongoDB and Couchbase. But let's have a look at the list again: Neo4j, MongoDB, Riak and Couchbase have been in this space for four years and have proven to be market leaders for specific requirements. The fifth database, Datomic, is astonishingly a completely new database, with a new paradigm written by a small team. We will dig into it a bit later when discussing all databases briefly.

Standards

Many people have asked for NoSQL standards, failing to see that NoSQL covers a wide range of models and requirements. Unified languages for major areas such as Wide Column, Key/Value, Document and Graph Databases will surely not be available for a long time because it's impossible to cover all areas. Several approaches, such as Spring Data, try to add a unified layer but it's up to the reader to test whether or not this layer is a leap forward in building a polyglot persistence environment . The graph and the document databases have come up with standards in their own domain. The graph world is more successful with its tinkerpop blueprints, Gremlin, Sparql, and Cypher. In the document space, UnQL and jaql fill some niches, although the first lacks real-world support by a NoSQL database. But with the force of Hadoop, many projects are working to bridge famous ETL languages such as Pig or Hive to other NoSQL databases. The standards world is highly fragmented, but only because NoSQL covers such a wide area.

Landscape

One of the best overviews of the database landscape comes from Matt Aslett in a report of the 451 Group. He recently updated his picture to give more insight. As you can see in the following picture, the landscape is highly fragmented and overlapping. There are several dimensions: Relational vs. Non-relational; Analytic vs. Operational; NoSQL vs. NewSQL. The last two categories have the well known sub-categories Key-Value, Document, Graph and Big Tables for NoSQL and Storage-Engines, Clustering-Sharding, New Databases and Cloud Service Solutions. The interesting part of this picture is that it is increasingly difficult to pin a database to an exact location. Everyone is now trying fiercely to integrate features from databases found in other spaces. NewSQL Systems implement core NoSQL features. NoSQL Systems try more and more to implement "classic" features such as SQL support or ACID or at least often configurable persistence.

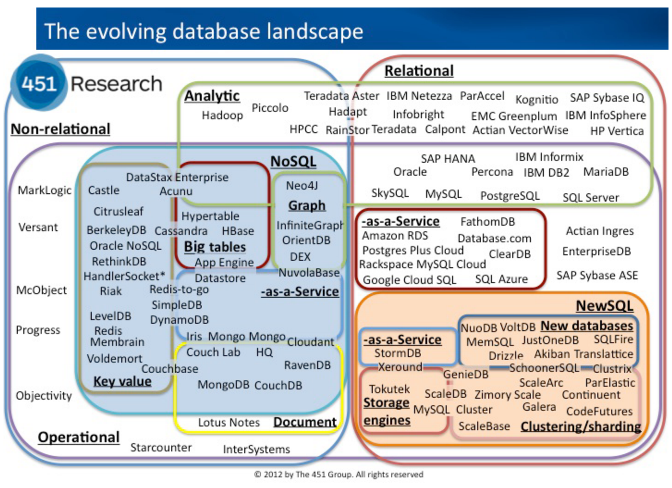


Figure 2: The database landscape by Matt Aslett (451 group)

It all started with the integration of Hadoop that tons of relational databases now offer. But there are many other examples. For example, MarkLogic is now starting to ride the JSON wave and thus also is hard to position. Furthermore, more multi-model databases appear, such as ArangoDB, OrientDB or AlechemyDB (which is now a part of the promising Aerospike DB). They allow users to start with one database model (e.g. document / JSON model) and add other models (graph or key-value) as new requirements pop up.

Books

Another wonderful sign of maturity is the book market. After two German books were published in 2010 and 2011, Wiley published Shashank Tiwari's "Professional NoSQL," a book structured like a hurricane and full of great deep insights. The race continued with two nice books in 2012. Eric Redmond and Jim Wilson's "Seven Databases in Seven Weeks" is surely a masterpiece. Freshly written and full of practical hands-on insights, it takes six famous NoSQL databases and adds PostgreSQL to the mix, making it a top recommendation.

P.J. Sandalage and Martin Fowler take a more holistic approach to cover all the characteristics and help evaluating your path and decisions with NoSQL in their "NoSQL Distilled". But there is more to come. It is just a matter of time till a Manning book appears on the scene: Dan McCreary and Ann Kelly are writing a book called "Making Sense of NoSQL" and the first MEAP chapters are already available. After starting with concepts and patterns, their Chapter 3 will surely look attractive:

- Building NoSQL Big Data solutions
- Building NoSQL search solutions
- Building NoSQL high availability solutions
- Using NoSQL to increase agility

This is a fresh approach and will surely be worth reading. Let's give each NoSQL leader a quick consideration. As one of the clear market leaders, Hadoop is a strange animal. On one hand it has enormous momentum. As mentioned before, each classic database vendor is in a hurry to announce Hadoop support. Companies such as Cloudera and MapR continue to grow and new Hadoop extensions and successors are announced every week. Even Hive and Pig continue to earn acceptance. Nevertheless, there is a fly in the ointment. Companies still complain about an unstructured mess (reading and parsing files could be even faster). MapReduce is "too batch" (even Google goes away from it), management is still hard, there are stability issues, and local training/consultants are difficult to find.

It's still a hot question whether Hadoop will grow as it is or will change dramatically. The second leader, MongoDB, also suffers from flame wars, and it might be the nature of things that leading databases get the most criticism. Nevertheless, MongoDB goes at a fast pace and criticism mostly is:

- Concerning old versions or
- Due to lack of knowledge.This recently culminated in absurd complaints that the 32-bit version can only handle 2GB, although MongoDB states this clearly in the download section and recommends the 64-bit version.

Anyway, MongoDB's partnerships and funding rounds push ambitious roadmaps:

- The industry called for security / LDAP features which are currently being developed
- Full text search will be in soon
- V8 for MapReduce is coming
- Even a finer level then collection level locking will come
- A Hash Shard Key is on the way

Especially this last point catches the interest of many architects. MongoDB was often blamed (also by competitors) for not implementing a concise consistent hashing, which is not entirely correct because such a key can be easily defined. But in the future there will be a config for a hash shard key. This means the user must decide if a hash key for sharding is useful or if he needs some of the (perhaps even rare) advantages of selecting his own sharding key. Surely this increases the pressure on other vendors and will lead to fruitful discussion on when to use a sharding key.

Cassandra is the next in line and doing well, adding more and nicer features such as better querying. However rumors won't stop telling that running a Cassandra cluster is not piece of cake and requires some hard work. Cassandra's most attractive issue is surely DataStax. The new company on top of Cassandra – and its $25 million round-C funding - is mostly addressing analytics and some operational issues. Especially the analytics was a surprise for many because in the early days Cassandra was not known as a powerful query machine. But as this has changed in the latest version, the query capabilities may be sufficient for modern analysis.

The development speed of Redis is also remarkable. Despite developer Salvatore Sanfilippo's assertions that he would have achieved nothing without the community and the help of Pieter Noordhuis, it still looks like a stunning one-man show. The sentinel failover and server-side scripting with the Lua programming language are the latest achievements. The decision for Lua was a bit of a shock for the community because everyone integrates JavaScript as a server-side language. Nevertheless, Lua is a neat language and will help Redis open up a panoply of possibilities. Couchbase also looks like a brilliant solution in terms of scalability and latency despite the strong winds that Facebook and hence Zynga are now facing. It's surely not a hot query machine but if Couchbase could improve querying in the future the portfolio would be comprehensive. The merger with the CouchDB founders was definitely a strong step and it's worthwhile to see the great influences of CouchDB in Couchbase. On every database conference it's also funny to hear the discussions, if CouchDB is doing better or worse after Damien, Chris and Jan have left. One can only hear extreme opinions here. But who cares as long as the database is doing fine. And it looks like it is. The last NoSQL databse to be mentioned here is Riak, which also has improved dramatically in functionality and monitoring. It continues to have a good reputation mostly in terms of stability: "rock solid, invisible and good for your sleep". The Riak CS fork also looks interesting in terms of the modularity of this technology.

Interesting Newcomers

Newcomers are always interesting to evaluate. Let's dig into some of them. Elastic Search surely is one of the hottest new NoSQL products and just got $10 million in series-A funding, and for a good reason. As a scalable search engine on top of Lucene it brings many advantages, primarily a company on top providing services and leveraging all the achievements that Lucene has conceived in the last years. It will surely infiltrate the industry, attacking the big players in the semi-structured information space.

Google has sent its small but fast LevelDB into the field. And it serves as a basis for many uses with specific requirements such as compression integration, even Riak-integrated LevelDB. It remains to be seen when all the new Google internal databases such as Dremel or Spanner will find their way out as open-source projects (like Apache Drill or Cloudera Impala). DynamoDB surely initiated a tectonic shift at the start of 2012. They call it the fastest growing service ever launched at Amazon. It's the ultimate scaling machine. New features are coming slowly but the focus on SSDs and latency is quite amazing.

Multi-model databases are also worth looking at. OrientDB, its famous representative, is not a newcomer but it is improving quickly. Perhaps too quickly, because some customers might now hope that since OrientDB has reached Version 1.0, it will gain a lot more stability. Graph, Document, Key-Value support combined with transactions and SQL are reasons enough to give it second try. Its good SQL support makes it interesting for analytic solutions such as Penthao. Another newcomer in this space is ArangoDB. It is moving fast and it doesn't flinch from comparing itself in benchmarks against the established players.

The native JSON and graph support saves a lot of effort if new requirements have to be implemented and the new data has a different model that must be persisted. By far the biggest surprise this year was Datomic. Written by some rock stars of the Clojure programming language in an incredibly short time, it unveils a whole bunch of new paradigms. It has made its way into the ThoughtWorks radar with the recommendation to have a look at it.

And although it is "just" a layer on top of established databases, it brings advantages such as:

- Transactions
- A time machine
- A fresh and powerful query approach
- A new schema approach
- Caching and scaling features

Currently, Datomic supports DynamoDB, Riak, Couchbase, Infinispan and SQL as the underlying storage engine. It even allows you to mix and query different databases simultaneously. Many veterans have been surprised that such a radical paradigm shift can be possible. Luckily, it is.

Summary

To conclude, let us address three points:

1. New articles by Eric Brewer on the CAP theorem should have come several years earlier. In his article - CAP Twelve Years Later: How the "Rules" Have Changed he states that "2 of 3" is misleading, explaining the reasons, why the world is more complicated than a simple CP/AP i.e. ACID/BASE choice. Nevertheless, thousands of talks and articles kept praising the CAP theorem without critical review for years. Michael Stonebraker was the strongest censor of the NoSQL movement (and the NoSQL space owes him a lot), pointing to these issues some years ago. Unfortunately, not many are listening. But now that Brewer has updated his theorem, the time of simple CAP statements is definitely over. Please be at the very front in pointing out the true and diverse CAP implications.

2. As we all know, the weaknesses of the classical relational databases have lead to the NoSQL field. But it was just a matter of time before the empire would strike back. Under the term "NewSQL" we can see a bunch of new engines (such as database.com, VoltDB, GenieDB, etc. see Figure 2), improving classic solutions, sharding and cloud solutions, thanks to the NoSQL movement.

But as many databases try to implement every feature, clear frontiers vanish. The decision for a database is getting more complicated than ever. You have to know about 50 use cases and 50 databases, and you should answer at least 50 questions. The latter have been gathered by the author over two years of NoSQL consulting and can be found here: Select the Right Database, Choosing between NoSQL and NewSQL.

3. It's common wisdom that every technology shift – since before client-server - is about ten times more costly to switch to. For example, switching from Mainframe to Client-Server, Client-Server to SOA, SOA to WEB, RDBMS to Hybrid Persistence, etc. And as a consequence, many companies hesitate and struggle in adding NoSQL to their portfolio. But it is also known that the early adopters who are trying to get the best of both worlds and thus integrate NoSQL quickly will be better positioned for the future. In this regard, NoSQL solutions will be here to stay and always a gainful area for evaluations.

QCon 2013
New York
SAVE $100 WHEN YOU REGISTER WITH PROMO CODE: "NOSQL"
OVER 100 SPEAKERS | 6 CONCURRENT TRACKS
KEY TOPICS: JAVA, AGILE, CLOUD, MOBILE, NOSQL, BIG DATA

About the Author

Prof. Dr. Stefan Edlich is a senior lecturer at Beuth HS of Technology Berlin (University of App. Sc.). He wrote more than ten IT books for publishers such as Apress, OReilly, Spektrum/Elsevier and others. He runs the NoSQL Archive, did NoSQL consulting, organizes NoSQL conferences, wrote the world's first two NoSQL books and is addicted to the Clojure programming language.

Introduction to MongoDB for Java, PHP and Python Developers

By Rick Hightower

This article covers using MongoDB as a way to get started with NoSQL. It presents an introduction to considering NoSQL, why MongoDB is a good NoSQL implementation to get started with, MongoDB shortcommings and trade-offs, key MongoDB developer concepts, MongoDB architectural concepts (sharding, replica sets), using the console to learn MongoDB, and getting started with MongoDB for Python, Java and PHP developers. The article uses MongoDB, but many concepts introduced are common in other NoSQL solutions. The article should be useful for new developers, ops and DevOps who are new to NoSQL.

From no NoSQL to sure why not

The first time I heard of something that actually could be classified as NoSQL was from Warner Onstine, who is currently working on some CouchDB articles for InfoQ. Warner was going on and on about how great CouchDB was. This was before the term NoSQL was coined. I was skeptical, and had just been on a project that was converted from an XML Document Database back to Oracle due to issues with the XML Database implementation. I did the conversion. I did not pick the XML Database solution, or decide to convert it to Oracle. I was just the consultant guy on the project (circa 2005) who did the work after the guy who picked the XML Database moved on and the production issues started to happen.

This was my first document database. This bred skepticism and distrust of databases that were not established RDBMS (Oracle, MySQL, etc.). This incident did not create the skepticism. Let me explain.

First, all of the Object-Oriented Database (OODB) folks for years preached how it was going to be the next big thing. It has not happened yet. I hear 2013 will be the year of the OODB just like it was going to be in 1997. Then there were the XML Database people preaching something very similar, which did not seem to happen either at least at the pervasive scale that NoSQL is happening. My take was, ignore this document-oriented approach and NoSQL, and see if it goes away. To be successful, it needs some community behind it, some clear use-case wins, and some corporate muscle/marketing, and I will wait until then. Sure the big guys need something like Dynamo and BigTable, but it is a niche, I assumed. Then there was BigTable, MapReduce, Google App Engine, and Dynamo in the news with white papers. Then Hadoop, Cassandra, MongoDB, Membase, HBase, and the constant small but growing drum beat of change and innovation. Even skeptics have limits.

"MongoDB is a good first step for developers dipping their toe into the NoSQL solution space."

Then in 2009, Eric Evans coined the term NoSQL to describe the growing list of open-source distributed databases. Now there is this NoSQL movement, three years in and counting. Like Ajax, giving something a name seems to inspire its growth, or perhaps we don't name movements until there is already a ground swell. Either way, having a name like NoSQL with a common vision is important to changing the world, and you can see the community, use-case wins, and corporate marketing muscle behind NoSQL. It has gone beyond the buzz stage. Also in 2009 was the first project that I worked on that had mass scale-out requirements that was using something that is classified as part of NoSQL.

In 2009, MongoDB was released from 10Gen and the NoSQL movement was in full swing. Somehow MongoDB managed to move to the front of the pack in terms of mindshare followed closely by Cassandra and others (see Figure 1). MongoDB is listed as a top job trend on Indeed.com, in

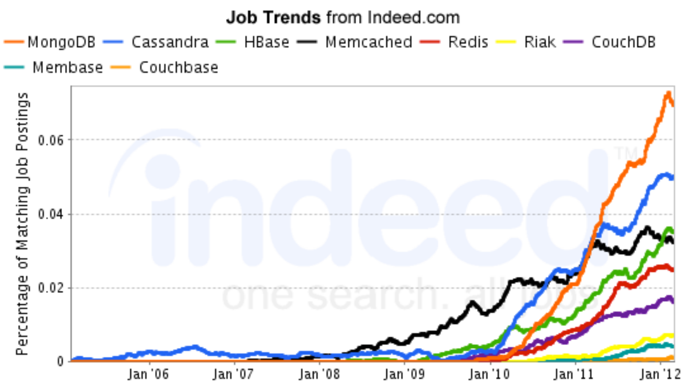


Figure 1: MongoDB leads the NoSQL pack

second place to be exact (behind HTML 5 and before iOS), which is fairly impressive given MongoDB was a relative latecomer to the NoSQL party.

MongoDB is a distributed document-oriented, schema-less storage solution similar to Couchbase and CouchDB. MongoDB uses JSON-style documents to represent, query and modify data. Internally, data is stored in BSON (binary JSON). MongoDB's closest cousins seem to be CouchDB/Couchbase. MongoDB supports many clients/languages, namely, Python, PHP, Java, Ruby, C++, etc. This article is going to introduce key MongoDB concepts and then show basic CRUD/Query examples in JavaScript (part of MongoDB console application), Java, PHP and Python.

Disclaimer: I have no ties with the MongoDB community and no vested interests in MongoDB's success or failure. I am not an advocate. I merely started to write about MongoDB because it seems to be the most successful, seems to have the most momentum for now, and in many ways typifies the diverse NoSQL market. MongoDB success is largely due to having easy-to-use, familiar tools. I'd love to write about CouchDB, Cassandra, Couchbase, Redis, HBase or any number of NoSQL solutions if there was just more hours in the day or stronger coffee or if coffee somehow extended how much time I had. Redis seems truly fascinating.

MongoDB seems to have the right mix of features and ease-of-use, and has become a prototypical example of what a NoSQL solution should look like. MongoDB can be used as a base of knowledge to understand other solutions (compare/contrast). This article is not an endorsement. Other than this, if you want to get started with NoSQL, MongoDB is a great choice.

MongoDB, a gateway drug to NoSQL

MongoDB's combination of features, simplicity, community, and documentation make it successful. The product itself has high availability, journaling (which is not always a given with NoSQL solutions), replication, auto-sharding, map reduce, and an aggregation framework (so you don't have to use map-reduce directly for simple aggregations). MongoDB can scale reads as well as writes.

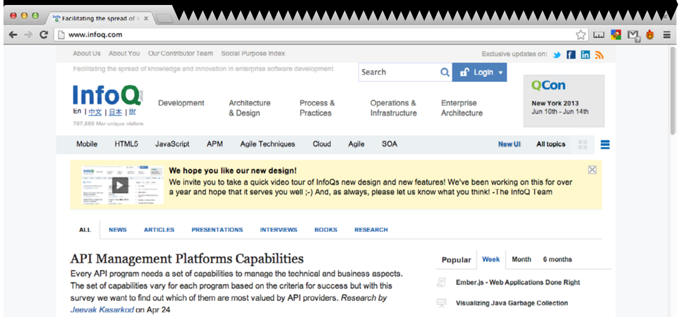
NoSQL, in general, has been reported to be more agile than full RDBMS/SQL due to problems with schema migration of SQL-based systems. Having been on large RDBMS systems and witnessing the trouble and toil of doing SQL schema migrations, I can tell you that this is a real pain to deal with. RDBMS/SQL often require a lot of upfront design or a lot schema migration later. NoSQL is viewed to be more agile in that it allows the applications to worry about differences in versioning instead of forcing schema migration and larger upfront designs. To the MongoDB crowd, it is said that MongoDB has dynamic schema not no schema (sort of like the dynamic language versus untyped language argument from Ruby, Python, etc. developers).

MongoDB does not seem to require a lot of ramp-up time. Its early success may be attributed to the quality and ease-of-use of its client drivers, which was more of an afterthought for other NoSQL solutions ("Hey here is our REST or XYZ wire protocol, deal with it yourself"). Compared to other NoSQL solutions, it has been said that MongoDB is easier to get started. Also, with MongoDB many DevOps things come cheaply or free. This is not that there are never any problems or one should not do capacity planning. MongoDB has become for many an easy onramp for NoSQL, a gateway drug if you will.

MongoDB was built to be fast. Speed is a good reason to pick MongoDB. Raw speed shaped architecture of MongoDB. Data is stored in memory-using memory-mapped files. This means that the virtual memory manager, a

Keep your NoSQL Skills Razor Sharp!

View all NoSQL Content on InfoQ.com



We hope you like our new design!
We invite you to take a quick video tour of InfoQs new design and new features! We've been working on this for over a year and hope that it serves you well ;-) And, as always, please let us know what you think! -The InfoQ Team

API Management Platforms Capabilities
Every API program needs a set of capabilities to manage the technical and business aspects. The set of capabilities vary for each program based on the criteria for success but with this survey we want to find out which of them are most valued by API providers. *Research by* *Jeevak Kasarkod* on Apr 24

Ember.js - Web Applications Done Right
Visualizing Java Garbage Collection

highly optimized system function of modern operating systems, does the paging/caching. MongoDB also pads areas around documents so that they can be modified in place, making updates less expensive. MongoDB uses a binary protocol instead of REST like some other implementations. Also, data is stored in a binary format instead of text (JSON, XML), which could speed writes and reads. Another reason MongoDB may do well is because it is easy to scale out reads and writes with replica sets and autosharding. You might expect if MongoDB is so great that there would be a lot of big names using them, and there are: Craigslist, Disney, The New York Times and many more use it.

Caveats

MongoDB indexes may not be as flexible as Oracle/MySQL/Postgres or other even other NoSQL solutions.
The order of index matters as it uses B-Trees. Realtime queries might not be as fast as Oracle/MySQL and other NoSQL solutions especially when dealing with array fields, and the query-plan optimization work is not as far as long as more mature solutions. You can make sure MongoDB is using the indexes you set up quite easily with an explain function. Don't let this scare you. MongoDB is good enough if queries are simple and you do a little homework, and it is always improving.

MongoDB does not have or integrate a full-text search engine like many other NoSQL solutions do (many use Lucene under the covers for indexing), although it seems to support basic text search better than most traditional databases. Every version of MongoDB seems to add more features and addresses shortcomings of the previous releases. MongoDB added journaling a while back so they can have single server durability; prior to this you really needed a replica or two to ensure some level of data safety.

10gen improved Mongo's replication and high availability with Replica Sets. Another issue with current versions of MongoDB (2.0.5) is lack of concurrency due to MongoDB having a global read/write lock, which allows reads to happen concurrently while write operations happen one at a time.

There are workarounds for this involving shards, and/or replica sets, but these are not always ideal, and do not fit the "it should just work mantra" of MongoDB. Recently at the MongoSF conference, Dwight Merriman, co-founder and CEO of 10gen, discussed the concurrency internals of MongoDB v2.2 (future release). Dwight noted that MongoDB 2.2 did a major refactor to add database level concurrency, and will soon have collection-level concurrency now that the hard part of the concurrency refactoring is done.

Also keep in mind, writes are in RAM and eventually get synced to disk since MongoDB uses memory-mapped files. Writes are not as expensive as if you were always waiting to sync to disk. Speed can mitigate concurrency issues.

This is not to say that MongoDB will never have some shortcomings and engineering tradeoffs. Also, you can, will and sometimes should combine MongoDB with a relational database or a full text search like Solr/Lucene for some applications. For example, if you run into issue with effectively building indexes to speed some queries you might need to combine MongoDB with Memcached. None of this is completely foreign though, as it is not uncommon to pair RDBMS with Memcached or Lucene/Solr. When to use MongoDB and when to develop a hybrid solution is beyond the scope of this article. In fact, when to use a SQL/RDBMS or another NoSQL solution is beyond the scope of this article, but it would be nice to hear more discussion on this topic.

The price you pay for MongoDB, one of the youngest but perhaps best-managed NoSQL solutions, is lack of maturity. It does not have a code base going back three decades like RDBMS systems. It does not have tons and tons of third party management and development tools. There have been issues, there are issues and there will be issues, but MongoDB seems to work well for a broad class of applications, and is rapidly addressing many issues.

Also finding skilled developers and ops (admins, devops, etc.) who are familiar with MongoDB or other NoSQL solutions might be tough. Somehow MongoDB seems to be the most approachable or perhaps just best marketed. Having worked on projects that used large NoSQL deployments, few people on the team really understand the product (limitations, etc.), which leads to trouble.

In short if you are kicking the tires of NoSQL, starting with MongoDB makes a lot of sense.

MongoDB concepts

MongoDB is document oriented but has many comparable concepts to traditional SQL/RDBMS solutions.

1. **Oracle:** Schema, Tables, Rows, Columns
2. **MySQL:** Database, Tables, Rows, Columns
3. **MongoDB:** Database, Collections, Document, Fields
4. **MySQL/Oracle:** Indexes
5. **MongoDB:** Indexes
6. **MySQL/Oracle:** Stored Procedures
7. **MongoDB:** Stored JavaScript

8. **Oracle/MySQL:** Database Schema
9. **MongoDB:** Schema free!
10. **Oracle/MySQL:** Foreign keys, and joins
11. **MongoDB:** DBRefs, but mostly handled by client code
12. **Oracle/MySQL:** Primary key
13. **MongoDB:** ObjectID

If you have used MySQL or Oracle here is a good guide to similar processes in MongoDB:

Database Process Type	Oracle	MySQL	MongoDB
Daemon/Server	Oracle	mysqld	mongod
Console Client	sqlplus	mysql	mongo
Backup utility	sqlplus	mysqldump	mongodump
Import utility	sqlplus	mysqlimport	mongoimport

You can see a trend here. Where possible, Mongo tries to follow the terminology of MySQL. They do this with console commands as well. If you are used to MySQL, MongoDB tries to make the transition a bit less painful.

SQL operations VS MongoDB operations

MongoDB queries are similar in concept to SQL queries and use a lot of the same terminology. There is no special language or syntax to execute MongoDB queries; you simply assemble a JSON object. The MongoDB site has a complete set of example queries done in both SQL and MongoDB JSON docs to highlight the conceptual similarities. What follows is several small listings to compare MongoDB operations to SQL.

Insert

SQL

```
INSERT INTO CONTACTS (NAME, PHONE_NUMBER)
VALUES('RICK HIGHTOWER','520-555-1212')
```

MongoDB

```
db.contacts.insert({name:'RICK HIGHTOWER',phoneNu
mber:'520-555-1212'})
```

Selects

SQL

```
SELECT name, phone_number FROM contacts WHERE
age=30 ORDER BY name DESC
```

MongoDB

```
db.contacts.find({age:30},
{name:1,phoneNumber:1}).sort({name:-1})
```

SQL

```
SELECT name, phone_number FROM contacts WHERE
age>30 ORDER BY name DESC
```

MongoDB

```
db.contacts.find({age:{$gt:33}},
```

THE ANNUAL INTERNATIONAL SOFTWARE DEVELOPMENT CONFERENCE

SOFTWARE IS CHANGING THE WORLD AND HERE ARE THE DATES:

QCon Chengdu
April 11 - 12, 2013

QCon Tokyo
Apr 23, 2013

QCon Beijing
April 25 - 27, 2013

QCon New York
June 12 - 14, 2013

QCon São Paulo
Aug 29 - 30, 2013

QCon Shanghai
Oct 23 - 25, 2013

QCon San Francisco
Nov 11 - 13, 2013

QCon London
5 - 7 March, 2014

for more info go to
qconferences.com

```
{name:1,phoneNumber:1}).sort({name:-1})
```

Creating indexes

SQL

```
CREATE INDEX contact_name_idx ON contact(name
DESC)
```

MongoDB

```
db.contacts.ensureIndex({name:-1})
```

Updates

SQL

```
UPDATE contacts SET phoneNumber='415-555-1212'
WHERE name='Rick Hightower'
```

MongoDB

```
db.contacts.update({name:'Rick    Hightower'},
{$set:{phoneNumber:1}}, false, true)
```

Additional features of note

MongoDB has many useful features like Geo Indexing (How close am I to X?), distributed file storage, capped collection (older documents auto-deleted), aggregation framework (like SQL projections for distributed nodes without the complexities of MapReduce for basic operations on distributed nodes), load sharing for reads via replication, autosharding for scaling writes, high availability, and your choice of durability (journaling) and/or data safety (make sure a copy exists on other servers).

Architecture replica sets, autosharding

The model of MongoDB is such that you can start basic and use features as your growth/needs change without too much trouble or change in design. MongoDB uses replica sets to provide read scalability, and high availability. Autosharding is used to scale writes (and reads). Replica sets and autosharding go hand in hand if you need mass scale out. With MongoDB scaling out seems easier than traditional approaches as many things seem to come built-in and happen automatically. Less operation/administration and a lower TCO than other solutions seems likely. However you still need capacity planning (good guess), monitoring (test your guess), and the ability to determine your current needs (adjust your guess).

Replica sets

The major advantages of replica sets are business continuity through high availability, data safety through data redundancy, and read scalability through load sharing (reads). Replica sets use a share-nothing architecture. A fair bit of the brains of replica sets is in the client libraries. The client libraries are replica-set aware. With replica sets, MongoDB language drivers know the current primary. Language driver is a library for a particular programming language, think JDBC driver or ODBC driver, but for MongoDB. All write operations go to the primary. If the primary is down, the drivers know how to get to the new primary (an elected new primary); this is auto failover for high availability. The data is replicated after writing. Drivers always write to the replica set's primary (called the master), the master then replicates to slaves. The primary is not fixed. The master/primary is nominated.

Typically you have at least three MongoDB instances in a replica set on different server machines (see figure 2). You can add more replicas of the primary if you like for read scalability, but you only need three for high availability failover. There is a way to sort of get down to two, but let's leave that out for this article. Except for this small tidbit, there are advantages of having three versus two in general. If you have two instances and one goes down, the remaining instance has 200% more load than before. If you have three instances and one goes down, the load for the remaining instances only go up by 50%. If you run your boxes at 50% capacity typically and you have an outage that means your boxes will run at 75% capacity until you get the remaining box repaired or replaced. If business continuity is your thing or important for your application, then having at least three instances in a replica set sounds like a good plan anyway (not all applications need it).

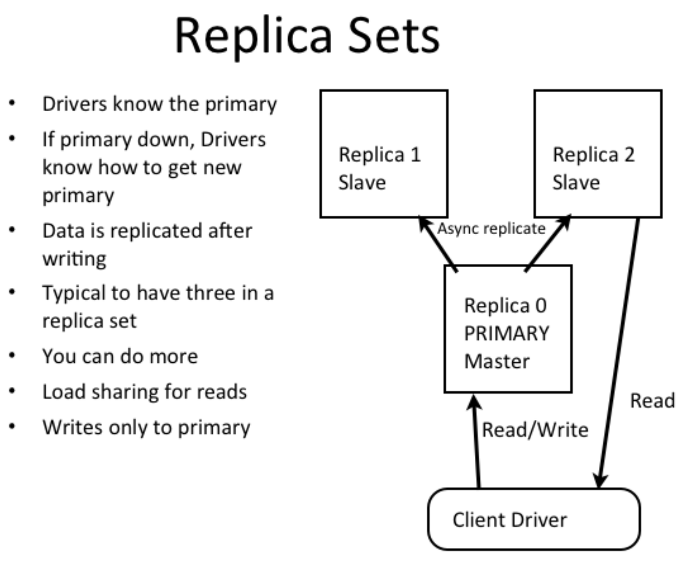


Figure 2: Replica sets

In general, once replication is set up it just works. However, in your monitoring, which is easy to do in MongoDB, you want to see how quickly data is replicated from the primary (master) to replicas (slaves). The slower the replication is, the dirtier your reads are. The replication is by default async (non-blocking). Slave data and primary data can be out of sync for however long it takes to do the replication. There are already whole books written just on making Mongo scalable, if you work at a Foursquare-like company or a company where high availability is very important and use Mongo, I suggest reading such a book.

By default, replication is non-blocking/async. This might be acceptable for some data (category descriptions in an online store), but not other data (a shopping cart's credit-card transaction data). For important data, the client can block until data is replicated on all servers or written to the journal (journaling is optional). The client can force the master to sync to slaves before continuing. This sync blocking is slower. Async/non-blocking is faster and is often described as eventual consistency. Waiting for a master to sync is a form of data safety.

There are several forms of data safety and options available to MongoDB from syncing to at least one other server to waiting for the data to be written to a journal (durability). Here is a list of some data safety options for MongoDB:

1. Wait until write has happened on all replicas
2. Wait until write is on two servers (primary and one other)
3. Wait until write has occurred on majority of replicas
4. Wait until write operation has been written to journal

(The above is not an exhaustive list of options.) The key word of each option above is wait. The more syncing and durability, the more waiting, and the harder it is to scale cost effectively.

Journaling: Is durability overvalued if RAM is the new disk? Data safety versus durability

It may seem strange to some that journaling was added as late as version 1.8 to MongoDB. Journaling is only now the default for 64-bit OS for MongoDB 2.0. Prior to that, you typically used replication to make sure write operations were copied to a replica before proceeding if the data was very important. The thought being that one server might go down, but two servers are very unlikely to go down at the same time. Unless somebody backs a truck over a highvoltage utility pole causing all of your air-conditioning equipment to stop working long enough for all of your servers to overheat at once, but that never happens (it happened to Rackspace and Amazon). And if you were worried about
this, you would have replication across availability zones, but I digress.
At one point MongoDB did not have single-server durability, now it does with addition of journaling. But this is far from a moot point. The general thought from the MongoDB community was and maybe still is that to achieve Web Scale, durability was thing of the past. After all, memory is the new disk. If you could get the data on a second server or two, your risk is reduced because the chances of them all going down at once is very, very low.

How often do servers go down these days? What are the chances of two servers going down at once? The general thought from MongoDB community was (is?) durability is overvalued and was just not Web Scale. Whether this is a valid point or not, there was much fun made about this at MongoDB's expense.

As you recall, MongoDB uses memory-mapped files for its storage engine so it could be a while for the operating system to sync the data in memory to the disk. If you did have several machines go down at once (which should be very rare), complete recoverability would be impossible. There were workaround with tradeoffs; for example to get around or minimize this now non-issue, you could force MongoDB to do an fsync of the data in memory to the file system, but as you guessed even with a RAID level four and a really awesome server that can get slow quick.

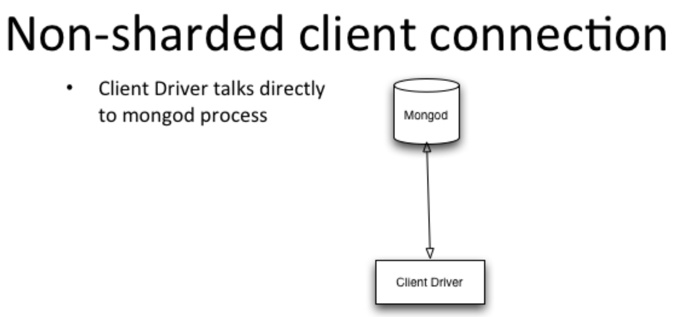
The moral of the story is MongoDB has journaling as well as many other options so you can decide what the best engineering tradeoff in data safety, raw speed and scalability. You get to pick. Choose wisely.

The reality is that no solution offers complete reliability, and if you are willing to allow for some loss (which you can with some data), you can get enormous improvements in speed and scale. Let's face it, your virtual-farm-game data is just not as important as Wells Fargo's bank transactions.

I know your mom will get upset when she loses the virtual tractor she bought for her virtual farm with her virtual money, but unless she pays real money she will likely get over it. I've lost a few posts on Twitter over the years, and I have not sued once. If your servers have an uptime of 99 percent and you block/replicate to three servers then the probability of them all going down at once is 1 in 1,000,000. Of course, problems you could replicate to another availability zone or geographic area connected with a high-speed WAN. How much speed and reliability do you need? How much money do you have?

An article on when to use MongoDB journaling versus older recommendations will be a welcome addition. Generally, it seems journaling is mostly a requirement for very sensitive financial data and single-server solutions. Your results may vary, and don't trust my math, it has been a few years since I got a B+ in statistics, and I am no expert on SLA of modern commodity servers (the above was just spitballing). If you have ever used a single non-clustered RDBMS system for a production system that relied on frequent backups and a transaction log (journaling) for data safety, raise your hand. Okay, if you raised your hand, then you just may not need autosharding or replica sets. To start with MongoDB, just use a single server with journaling turned on.

If you require speed, you can configure MongoDB journaling to batch writes to the journal (which is the default). This is a good model to start out with and probably very much like quite a few applications you've already worked on (assuming that most application don't need high availability). The difference is, of course, if later your application needs high availability, read scalability, or write scalability, MongoDB has you covered. Also setting up high availability seems easier on MongoDB than other more established solutions.


Non-sharded client connection

- Client Driver talks directly to mongod process

Figure 3: Simple setup with journaling and single server; okay for a lot of applications

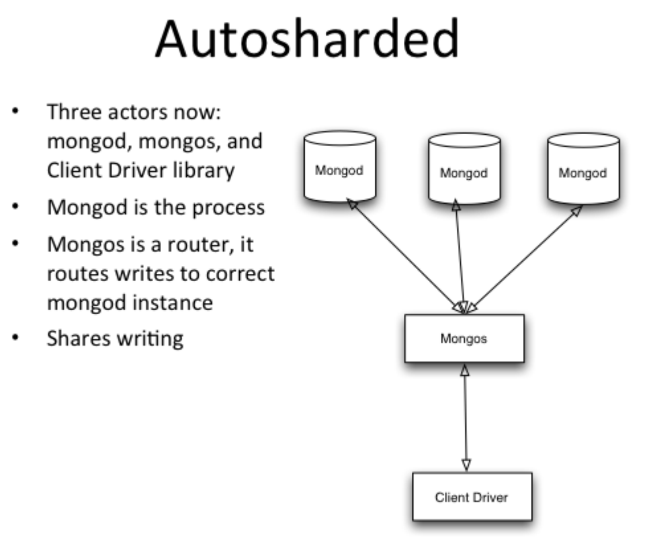
If you can afford two other servers and your app reads more than it writes, you can get improved high availability and increased read scalability with replica sets. If your application is write-intensive then you might need autosharding. The point is you don't have to be Facebook or Twitter to use MongoDB. You can even be working on a one-off dinky application. MongoDB scales down as well as up.

Autosharding

Replica sets are good for failover and speeding up reads, but to speed up writes, you need autosharding. According to a talk by Roger Bodamer on Scaling with MongoDB, 90% of projects do not need autosharding. Conversely, almost all projects will benefit from replication and high availability provided by replica sets. Also once MongoDB improves its concurrency in version 2.2 and beyond, it may be the case that 97% of projects don't need autosharding.

Sharding allows MongoDB to scale horizontally. Sharding is also called partitioning. You partition to each of your servers
a portion of the data to hold or the system does this for you. MongoDB can automatically change partitions for optimal data distribution and load balancing, and it allows you to elastically add new nodes (MongoDB instances). How to setup autosharding is beyond the scope of this introductory article. Autosharding can support automatic failover (along with replica sets). There is no single point of failure. Remember 90% of deployments don't need sharding, but if you do need scalable writes (apps like Foursquare, Twitter, etc.), autosharding will work with minimal impact on your client code.

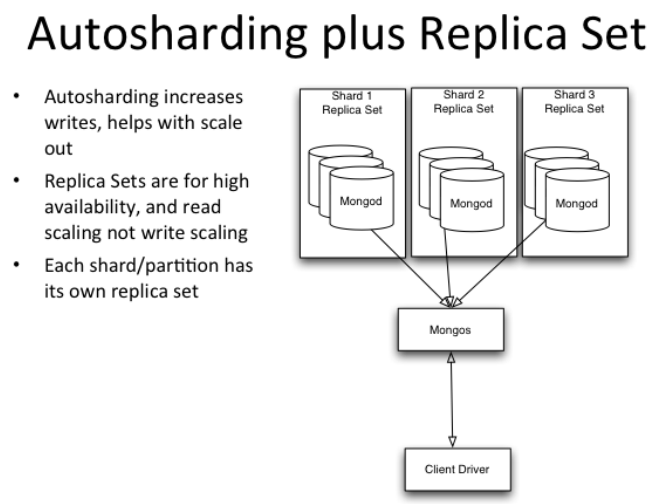
There are three main process actors for autosharding: mongod (database daemon), mongos, and the client-driver library. Each mongod instance gets a shard. Mongod is the process that manages databases, and collections. Mongos is a router, it routes writes to the correct mongod instance for autosharding. Mongos also handles looking for which shards will have data for a query. To the client driver, mongos looks like a mongod process more or less (autosharding is transparent to the client drivers). Autosharding increases


Autosharded

- Three actors now: mongod, mongos, and Client Driver library
- Mongod is the process
- Mongos is a router, it routes writes to correct mongod instance
- Shares writing

Figure 4: MongoDB Autosharding

write and read throughput, and helps with scale out. Replica sets are for high availability and read throughput. You can combine them as shown in figure 5.


Autosharding plus Replica Set

- Autosharding increases writes, helps with scale out
- Replica Sets are for high availability, and read scaling not write scaling
- Each shard/partition has its own replica set

Figure 5: MongoDB Autosharding plus Replica Sets for scalable reads, scalable writes, and high availability

You shard on an indexed field in a document. Mongos collaborates with config servers (mongod instances acting as config servers), which have the shard topology (where do the key ranges live?). Shards are just normal mongod instances. Config servers hold meta-data about the cluster and are also mongodb instances.

Shards are further broken down into 64-MB chunks called chunks. A chunk is 64-MB worth of documents for a collection. Config servers hold which shard the chunks live in. The autosharding happens by moving these chunks around and distributing them into individual shards. The mongos processes have a balancer routine that wakes up so often, it checks to see how many chunks a particular shard has. If a particular shard has too many chunks (nine more chunks than another shard), then mongos starts to move data from one shard to another to balance the data capacity amongst the shards. Once the data is moved then the config servers are updated in a two-phase commit.

The config servers contain a versioned shard topology and are the gatekeeper for autosharding balancing. This topology maps which shard has which keys. The config servers are like a DNS server for shards. The mongos process uses config servers to find where shard keys live. Mongod instances are shards that can be replicated using replica sets for high availability. Mongos and config server processes do not need to be on their own server and can live on a primary box of a replica set for example. For sharding you need at least three config servers, and shard topologies cannot change unless all three are up at the same time. This ensures consistency of the shard topology. The full autosharding topology is show in figure 6.

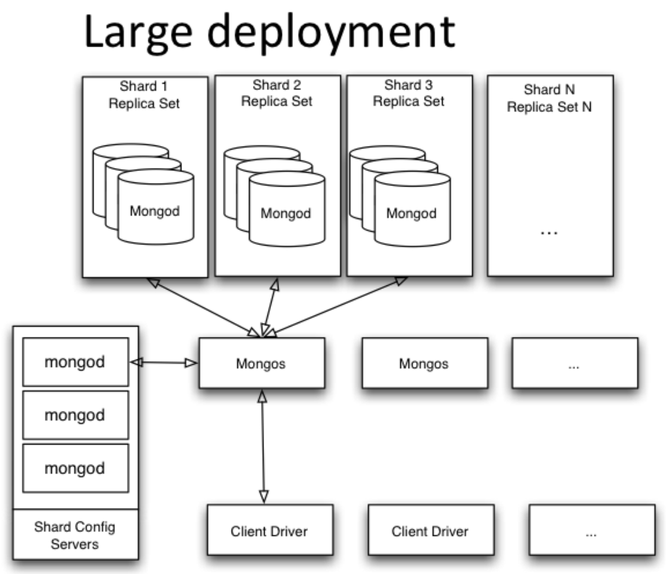


Figure 6: MongoDB Autosharding full topology for large deployment including Replica Sets, Mongos routers, Mongod Instance, and Config Servers

Kristina Chodorow, author of “Scaling MongoDB”, gave an excellent talk on the internals of MongoDB sharding at OSCON 2011 if you would like to know more.

-

MapReduce

MongoDB has MapReduce capabilities for batch processing similar to Hadoop. Massive aggregation is possible through the divide-and-conquer nature of MapReduce. Before the aggregation framework, MongoDB’s MapReduce could be used instead to implement what you might do with SQL projections (Group/By SQL). MongoDB also added the aggregation framework, which negates the need for MapReduce for common aggregation cases. In MongoDB, Map and Reduce functions are written in JavaScript. These functions are executed on servers (mongod), which allows the code to be next to data that it is operating on (think stored procedures, but meant to execute on distributed nodes and then collected and filtered). The results can be copied to a results collection.

MongoDB also provides incremental MapReduce. This allows you to run MapReduce jobs over collections, and then later run a second job but only over new documents in the collection. You can use this to reduce work required by merging new data into existing results collection.

Aggregation framework

The aggregation framework was added in MongoDB 2.1. It is similar to SQL group by. Before the aggregation framework, you had to use MapReduce for things like SQL's group by. Using the aggregation framework capabilities is easier than MapReduce. Let's cover a small set of aggregation functions and their SQL equivalents inspired by the Mongo docs as follows:

Count

SQL

```
SELECT COUNT(*) FROM employees
```

MongoDB

```
db.users.employees([
{ $group: {_id:null, count:{$sum:1}} }
])
```

Sum salary where of each employee who are not retired by department

SQL

```
SELECT dept_name SUM(salary) FROM employees
WHERE retired=false GROUP BY dept_name
```

MongoDB

```
db.orders.aggregate([
 { $match:{retired:false} },
     { $group:{_id:"$dept_name",
total:{$sum:"$salary"}} }
])
```

Installing MongoDB

Let's mix in some code samples to try out along with the concepts.

To install MongoDB go to the download page, download and unrar/unzip the download to ~/mongodb-platform-version/. Next you want to create the directory that will hold the data and create a mongodb.config file (/etc/mongodb/mongodb.config) that points to said directory as follows:

Listing: Installing MongoDB

```
$ sudo mkdir /etc/mongodb/data

$ cat /etc/mongodb/mongodb.config
dbpath=/etc/mongodb/data
```

The /etc/mongodb/mongodb.config has one line dbpath=/etc/mongodb/data that tells mongo where to put the data.

Next, you need to link mongodb to /usr/local/mongodb and then add it to the path environment variable as follows:

Listing: Setting up MongoDB on your path

```
$ sudo ln -s ~/mongodb-platform-version/ /usr/
local/mongodb
$ export PATH=$PATH:/usr/local/mongodb/bin
```

Run the server passing the configuration file that we created earlier.

Listing: Running the MongoDB server

```
$ mongod --config /etc/mongodb/mongodb.config
```

Mongo comes with a nice console application called mongo that lets you execute commands and JavaScript. JavaScript to Mongo is what PL/SQL is to Oracle's database. Let's fire up the console app, and poke around.

Firing up the mongos console application

```
$ mongo
MongoDB shell version: 2.0.4
connecting to: test
…
> db.version()
2.0.4
>
```

One of the nice things about MongoDB is the self-describing console. It is easy to see what commands a MongoDB database supports with the db.help() as follows:

Client: mongo db.help()

```
> db.help()
DB methods:
db.addUser(username, password[, readOnly=false])
db.auth(username, password)
db.cloneDatabase(fromhost)
db.commandHelp(name) returns the help for the
command
db.copyDatabase(fromdb, todb, fromhost)
db.createCollection(name, { size : ..., capped :
..., max : ... } )
db.currentOp() displays the current operation in
the db
```

```
db.dropDatabase()
db.eval(func, args) run code server-side
db.getCollection(cname) same as db['cname'] or
db.cname
db.getCollectionNames()
db.getLastError() - just returns the err msg
string
db.getLastErrorObj() - return full status object
db.getMongo() get the server connection object
db.getMongo().setSlaveOk() allow this connection
to read from the nonmaster member of a replica
pair
db.getName()
db.getPrevError()
db.getProfilingStatus() - returns if profiling is
on and slow threshold
db.getReplicationInfo()
db.getSiblingDB(name) get the db at the same
server as this one
db.isMaster() check replica primary status
db.killOp(opid) kills the current operation in
the db
db.listCommands() lists all the db commands
db.logout()
db.printCollectionStats()
db.printReplicationInfo()
db.printSlaveReplicationInfo()
db.printShardingStatus()
db.removeUser(username)
db.repairDatabase()
db.resetError()
db.runCommand(cmdObj) run a database command. if
cmdObj is a string, turns it into { cmdObj : 1 }
db.serverStatus()
db.setProfilingLevel(level,{slowms}) 0=off 1=slow
2=all
db.shutdownServer()
db.stats()
db.version() current version of the server
db.getMongo().setSlaveOk() allow queries on a
replication slave server
db.fsyncLock() flush data to disk and lock server
for backups
db.fsyncUnock() unlocks server following a
db.fsyncLock()
```

You can see some of the commands refer to concepts we discussed earlier. Now let's create a employee collection, and do some CRUD operations on it.

Create Employee Collection

```
> use tutorial;
switched to db tutorial
> db.getCollectionNames(); [ ]
  > db.employees.insert({name:'Rick Hightow-
er', gender:'m', gender:'m', phone:'520-555-1212',
age:42});
Mon Apr 23 23:50:24 [FileAllocator] allocating
new datafile /etc/mongodb/data/tutorial.ns, ...
```

The use command uses a database. If that database does not exist, it will be lazily created the first time we access it (write to it). The db object refers to the current database. The current database does not have any document collections to start with (this is why db.getCollections() returns an empty list). To create a document collection, just insert a new document. Collections like databases are lazily created when they are actually used. You can see that two collections are created when we inserted our first document into the employees collection as follows:

```
> db.getCollectionNames();
[ "employees", "system.indexes" ]
```

The first collection is our employees collection and the second collection is used to hold onto indexes we create.

To list all employees you just call the find method on the employees collection.

```
> db.employees.find()
{ "_id" : ObjectId("4f964d3000b5874e7a163895"),
"name" : "Rick Hightower",
    "gender" : "m", "phone" : "520-555-1212",
"age" : 42 }
```

The above is the query syntax for MongoDB. There is not a separate SQL like language. You just execute JavaScript code, passing documents, which are just JavaScript associative arrays, err, I mean JavaScript objects. To find a particular employee, you do this:

```
> db.employees.find({name:"Bob"})
```

Bob quit so to find another employee, you would do this:

```
> db.employees.find({name:"Rick Hightower"})
{ "_id" : ObjectId("4f964d3000b5874e7a163895"),
"name" : "Rick Hightower", "gender" : "m",
"phone" : "520-555-1212", "age" : 42 }
```

The console application just prints out the document right to the screen. I don't feel 42. At least I am not 100 as shown by this query:

```
> db.employees.find({age:{$lt:100}})
{ "_id" : ObjectId("4f964d3000b5874e7a163895"),
"name" : "Rick Hightower", "gender" : "m", "phone"
: "520-555-1212", "age" : 42 }
```

Notice to get employees less than a 100, you pass a document with a subdocument, the key is the operator ($lt), and the value is the value (100). Mongo supports all of the operators you would expect like $lt for less than, $gt for greater than, etc. If you know JavaScript, it is easy to inspect fields of a document, as follows:

```
> db.employees.find({age:{$lt:100}})[0].name
Rick Hightower
```

If we were going to query, sort or shard on employee names, then we would need to create an index as follows:

```
db.employees.ensureIndex({name:1}); //ascending
index, descending would be -1
```

Indexing by default is a blocking operation, so if you are indexing a large collection, it could take several minutes and perhaps much longer. This is not something you want to do casually on a production system. There are options to build indexes as a background task, to setup a unique index, and complications around indexing on replica sets, and much more. If you are running queries that rely on certain indexes to be performant, you can check to see if an index exists with db.employees.getIndexes(). You can also see a list of indexes as follows:

```
> db.system.indexes.find()
{ "v" : 1, "key" : { "_id" : 1 }, "ns" : "tuto-
rial.employees", "name" : "_id_" }
```

By default all documents get an object id. If you don't give an object an _id, it will be assigned one by the system (like a criminal suspect gets a lawyer). You can use that _id to look up an object as follows with findOne:

```
> db.employees.findOne({_id : ObjectId("4f964d3000
b5874e7a163895")})
{ "_id" : ObjectId("4f964d3000b5874e7a163895"),
"name" : "Rick Hightower",
  "gender" : "m", "phone" : "520-555-1212", "age"
: 42 }
```

Java and MongoDB

Pssst! Here is a dirty little secret. Don't tell your Node.js friends or Ruby friends this. More Java developers use MongoDB than Ruby and Node.js. They just are not as loud about it. Using MongoDB with Java is very easy.

The language driver for Java seems to be a straight port of something written with JavaScript in mind, and the usuability suffers a bit because Java does not have literals for maps/objects like JavaScript does. Thus an API written for a dynamic language does not quite fit Java. There can be a lot of usability improvement in the MongoDB Java language driver (hint, hint). There are alternatives to using just the straight MongoDB language driver, but I have not picked a clear winner (mjorm, morphia, and Spring data MongoDB support). I'd love just some usability improvements in the core driver without the typical Java annotation fetish, perhaps a nice Java DAO DSL (see section on criteria DSL if you follow the link).

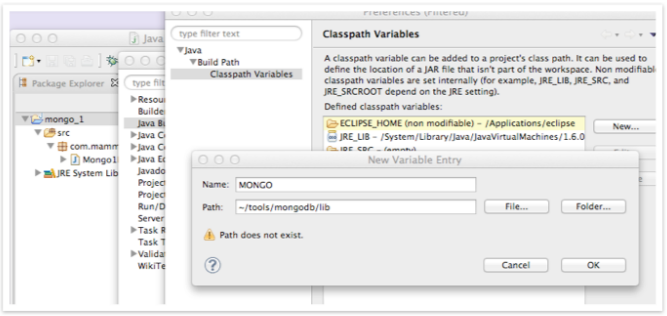
Setting up Java and MongoDB

Let's go ahead and get started then with Java and MongoDB.

Download latest mongo driver from github (https://github.com/mongodb/mongo-java-driver/downloads), then put it somewhere, and then add it to your classpath as follows:

```
$ mkdir tools/mongodb/lib
$ cp mongo-2.7.3.jar tools/mongodb/lib
```

This assumes you are using Eclipse, but if not by now you know how to translate these instructions to your IDE anyway. The short story is put the mongo jar file on your classpath. You can put the jar file anywhere, but I like to keep mine in ~/tools/.
If you are using Eclipse it is best to create a classpath variable so other projects can use the same variable and not go through the trouble. Create a new Eclipse Java project in a new Workspace. Now right-click your new project, open the project properties, go to the Java Build Path->Libraries->Add Variable->Configure Variable shown in Figure 7.



Classpath Variables

A classpath variable can be added to a project's class path. It can be used to define the location of a JAR file that isn't part of the workspace. Non modifiabl classpath variables are set internally (for example, JRE_LIB, JRE_SRC, and JRE_SRCROOT depend on the JRE setting).

Defined classpath variables:

ECLIPSE_HOME (non modifiable) – /Applications/eclipse
JRE_LIB – /System/Library/Java/JavaVirtualMachines/1.6.0

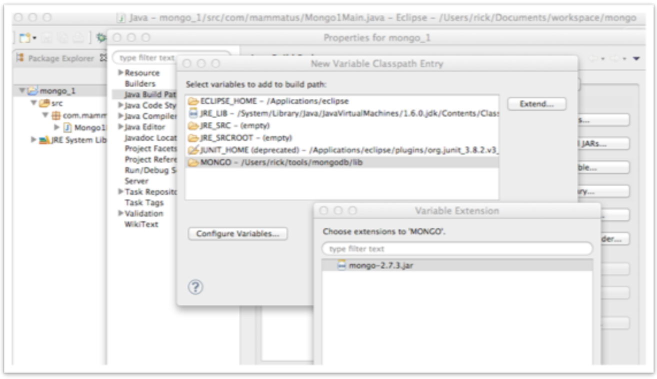
New Variable Entry

Name: MONGO
Path: ~/tools/mongodb/lib

Path does not exist.

Figure 7: Adding Mongo jar file as a classpath variable in Eclipse

For Eclipse from the "Project Properties->Java Build Path->Libraries", click "Add Variable", select "MONGO", click "Extend...", select the jar file you just downloaded.



New Variable Classpath Entry

Select variables to add to build path:

ECLIPSE_HOME – /Applications/eclipse
JRE_LIB – /System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Class
JRE_SRC – (empty)
JRE_SRCROOT – (empty)
JUNIT_HOME (deprecated) – /Applications/eclipse/plugins/org.junit_3.8.2.v3_
MONGO – /Users/rick/tools/mongodb/lib

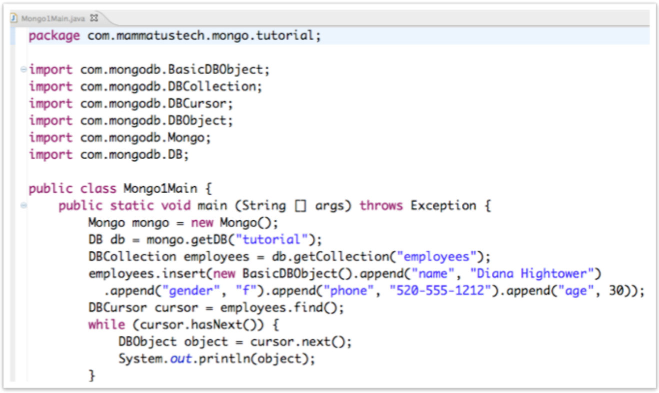
Variable Extension

Choose extensions to 'MONGO'.

mongo-2.7.3.jar

Figure 8: Adding Mongo jar file to your project

Once you have it all setup, working with Java and MongoDB is quite easy as shown in figure 9.



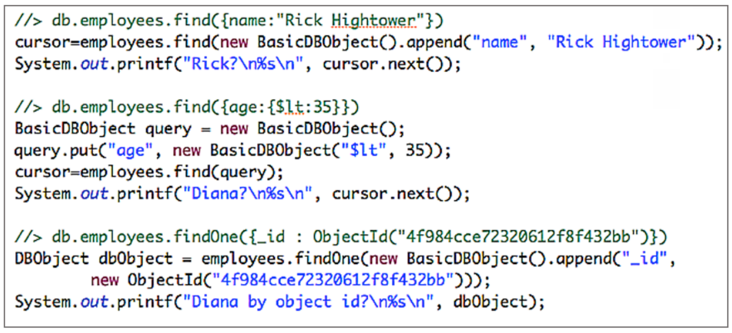
```java
package com.mammatustech.mongo.tutorial;

import com.mongodb.BasicDBObject;
import com.mongodb.DBCollection;
import com.mongodb.DBCursor;
import com.mongodb.DBObject;
import com.mongodb.Mongo;
import com.mongodb.DB;

public class Mongo1Main {
    public static void main (String [] args) throws Exception {
        Mongo mongo = new Mongo();
        DB db = mongo.getDB("tutorial");
        DBCollection employees = db.getCollection("employees");
        employees.insert(new BasicDBObject().append("name", "Diana Hightower")
          .append("gender", "f").append("phone", "520-555-1212").append("age", 30));
        DBCursor cursor = employees.find();
        while (cursor.hasNext()) {
            DBObject object = cursor.next();
            System.out.println(object);
        }
```

Figure 9: Using MongoDB from Eclipse

The above is roughly equivalent to the console/JavaScript code that we were doing earlier. The BasicDBObject is a type of Map with some convenience methods added. The DBCursor is like a JDBC ResultSet. You execute queries with DBColleciton. There is no query syntax, just finder methods on the collection object. The output from the above is:

```
Out:
{ "_id" : { "$oid" : "4f964d3000b5874e7a163895"}
, "name" : "Rick
Hightower" , "gender" : "m" , "phone" : "520-
555-1212" ,
"age" : 42.0}
{ "_id" : { "$oid" : "4f984cce72320612f8f432bb"}
, "name" : "Diana
Hightower" , "gender" : "f" , "phone" : "520-
555-1212" ,
"age" : 30}
```

Once you create some documents, querying for them is quite simple as show in figure 10.



```
//> db.employees.find({name:"Rick Hightower"})
cursor=employees.find(new BasicDBObject().append("name", "Rick Hightower"));
System.out.printf("Rick?\n%s\n", cursor.next());

//> db.employees.find({age:{$lt:35}})
BasicDBObject query = new BasicDBObject();
query.put("age", new BasicDBObject("$lt", 35));
cursor=employees.find(query);
System.out.printf("Diana?\n%s\n", cursor.next());

//> db.employees.findOne({_id : ObjectId("4f984cce72320612f8f432bb")})
DBObject dbObject = employees.findOne(new BasicDBObject().append("_id",
        new ObjectId("4f984cce72320612f8f432bb")));
System.out.printf("Diana by object id?\n%s\n", dbObject);
```

Figure 10: Using Java to query MongoDB

The output from figure 10 is as follows:

```
Rick?
{ "_id" : { "$oid" : "4f964d3000b5874e7a163895"}
, "name" : "Rick
Hightower" , "gender" : "m" , "phone" : "520-
555-1212" , "age" : 42.0}
Diana?
{ "_id" : { "$oid" : "4f984cae72329d0ecd8716c8"}
, "name" : "Diana
Hightower" , "gender" : "f" , "phone" : "520-
555-1212" , "age" : 30}

Diana by object id?
{ "_id" : { "$oid" : "4f984cce72320612f8f432bb"}
, "name" : "Diana
Hightower" , "gender" : "f" , "phone" : "520-
555-1212" , "age" : 30}
```

Just in case anybody wants to cut and paste any of the above, here it is again all in one go in the following listing.

Listing: Complete Java Listing

```
package com.mammatustech.mongo.tutorial;
import org.bson.types.ObjectId;
import com.mongodb.BasicDBObject;
import com.mongodb.DBCollection;
import com.mongodb.DBCursor;
import com.mongodb.DBObject;
import com.mongodb.Mongo;
import com.mongodb.DB;
public class Mongo1Main {
     public static void main (String [] args)
throws Exception {
Mongo mongo = new Mongo();
DB db = mongo.getDB("tutorial");
DBCollection           employees          =
db.getCollection("employees");
employees.insert(new          BasicDBObject().
append("name", "Diana Hightower")
  .append("gender", "f").append("phone", "520-
555-1212").append("age", 30));
DBCursor cursor = employees.find();
while (cursor.hasNext()) {
     DBObject object = cursor.next();
     System.out.println(object);
}

//> db.employees.find({name:"Rick Hightower"})
cursor=employees.find(new       BasicDBObject().
append("name", "Rick Hightower"));
System.out.printf("Rick?\n%s\n", cursor.next());

//> db.employees.find({age:{$lt:35}})
BasicDBObject query = new BasicDBObject();
        query.put("age", new BasicDBObject("$lt",
35));
cursor=employees.find(query);
System.out.printf("Diana?\n%s\n",        cursor.
next());

//> db.employees.findOne({_id : ObjectId("4f984c
ce72320612f8f432bb")})
DBObject dbObject = employees.findOne(new Ba-
sicDBObject().append("_id",
new ObjectId("4f984cce72320612f8f432bb")));
System.out.printf("Diana by object id?\n%s\n",
dbObject);


     }
}
```

Please note that the above is completely missing any error checking, or resource cleanup. You will need to do some of course (try/catch/finally, close connection, you know that sort of thing).

Python MongoDB Setup

Setting up Python and MongoDB are quite easy since Python has its own package manager.

To install mongodb lib for Python MAC OSX, you would do the following:

```
  $ sudo env ARCHFLAGS='-arch i386 -arch x86_64'
  $ python -m easy_install pymongo
```

To install Python MongoDB on Linux or Windows do the following:

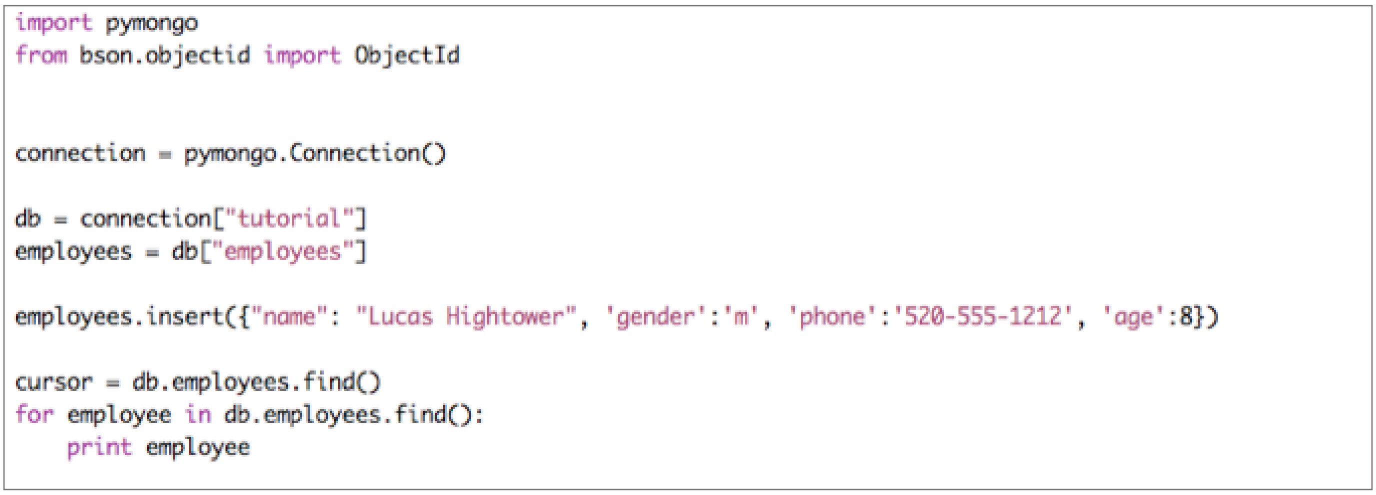
```
  $ easy_install pymongo
```

or

```
  $ pip install pymongo
```

If you don't have easy_install on your Linux box you may have to do some sudo apt-get install python-setuptools or sudo yum install python-setuptools iterations, although it seems to be usually installed with most Linux distributions these days. If easy_install or pip is not installed on Windows, try reformatting your hard disk and installing a real OS, or if that is too inconvient go here.

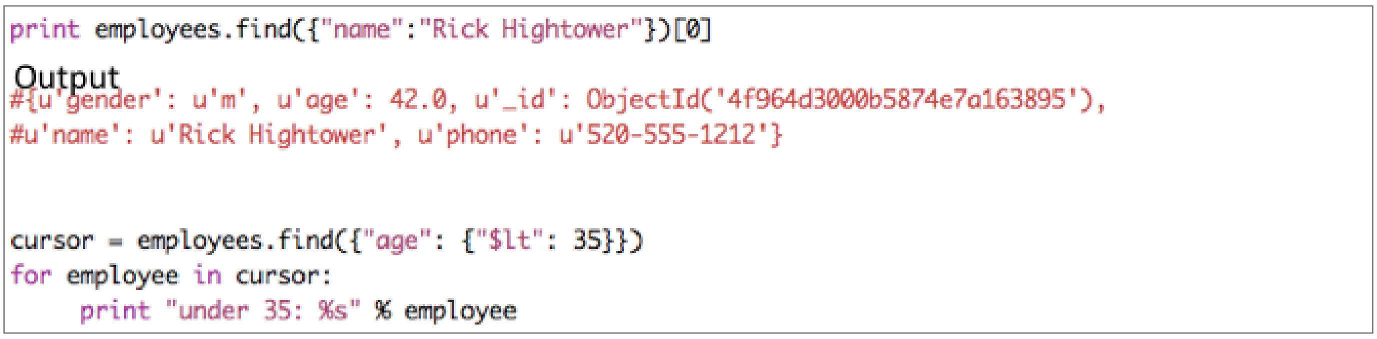
Once you have it all setup, you will can create some code that is equivalent to the first console examples as shown in Figure 11.



```
import pymongo
from bson.objectid import ObjectId


connection = pymongo.Connection()

db = connection["tutorial"]
employees = db["employees"]

employees.insert({"name": "Lucas Hightower", 'gender':'m', 'phone':'520-555-1212', 'age':8})

cursor = db.employees.find()
for employee in db.employees.find():
    print employee
```

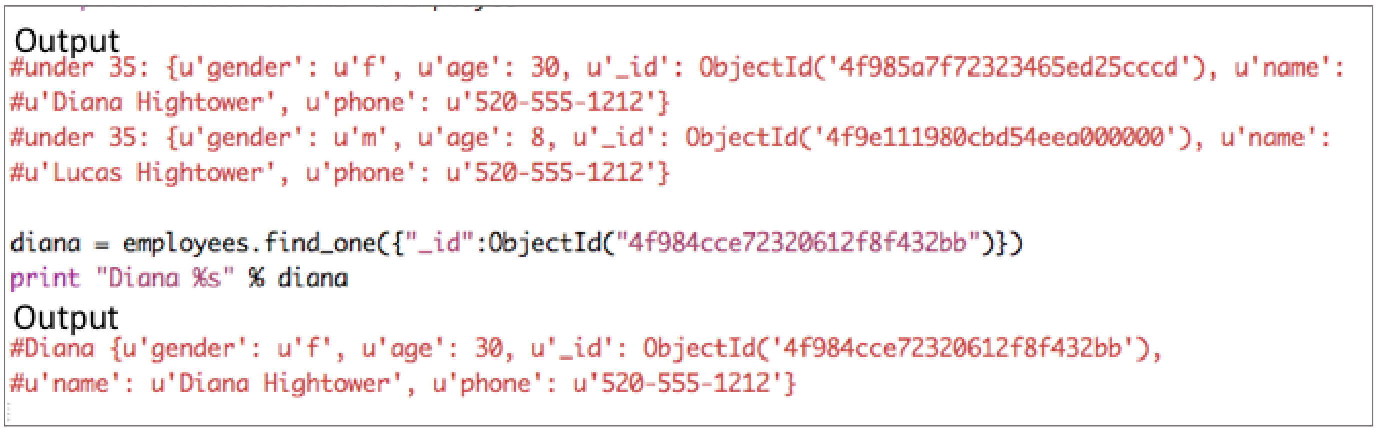
Figure 11: Python code listing part 1

Python does have literals for maps so working with Python is much closer to the JavaScript/Console from earlier than Java is. Like Java there are libraries for Python that work with MongoDB (MongoEngine, MongoKit, and more). Even executing queries is very close to the JavaScript experience as shown in figure 12.



```
print employees.find({"name":"Rick Hightower"})[0]

Output
#{u'gender': u'm', u'age': 42.0, u'_id': ObjectId('4f964d3000b5874e7a163895'),
#u'name': u'Rick Hightower', u'phone': u'520-555-1212'}


cursor = employees.find({"age": {"$lt": 35}})
for employee in cursor:
     print "under 35: %s" % employee
```

Figure 12: Python code listing part 2



```
Output
#under 35: {u'gender': u'f', u'age': 30, u'_id': ObjectId('4f985a7f72323465ed25cccd'), u'name':
#u'Diana Hightower', u'phone': u'520-555-1212'}
#under 35: {u'gender': u'm', u'age': 8, u'_id': ObjectId('4f9e111980cbd54eea000000'), u'name':
#u'Lucas Hightower', u'phone': u'520-555-1212'}

diana = employees.find_one({"_id":ObjectId("4f984cce72320612f8f432bb")})
print "Diana %s" % diana
Output
#Diana {u'gender': u'f', u'age': 30, u'_id': ObjectId('4f984cce72320612f8f432bb'),
#u'name': u'Diana Hightower', u'phone': u'520-555-1212'}
```

Here is the complete listing to make the cut-and-paste crowd (like me), happy.

Listing: Complete Python listing

```
import pymongo
from bson.objectid import ObjectId
connection = pymongo.Connection()
db = connection[“tutorial”]
employees = db[“employees”]
employees.insert({“name”: “Lucas Hightower”, ‘gender’:’m’, ‘phone’:’520-555-1212’, ‘age’:8})
cursor = db.employees.find()
for employee in db.employees.find():
  print employee
print employees.find({“name”:”Rick Hightower”})[0]
cursor = employees.find({“age”: {“$lt”: 35}})
for employee in cursor:
   print “under 35: %s” % employee
diana = employees.find_one({“_id”:ObjectId(“4f984cce72320612f8f432bb”)})
print “Diana %s” % diana
```

The output for the Python example is as follows:

```
{u’gender’: u’m’, u’age’: 42.0, u’_id’: ObjectId(‘4f964d3000b5874e7a163895’), u’name’: u’Rick Hightower’,
u’phone’:
u’520-555-1212’}
{u’gender’: u’f’, u’age’: 30, u’_id’: ObjectId(‘4f984cae72329d0ecd8716c8’), u’name’: u’Diana Hightower’,
u’phone’:
u’520-555-1212’}
{u’gender’: u’m’, u’age’: 8, u’_id’: ObjectId(‘4f9e111980cbd54eea000000’), u’name’: u’Lucas Hightower’,
u’phone’:
u’520-555-1212’}
```

All of the above but in PHP

Node.js , Ruby, and Python in that order are the trend settting crowd in our industry circa 2012. Java is the corporate crowd, and PHP is the workhorse of the Internet, the “get it done” crowd. You can’t have a decent NoSQL solution without having good PHP support.

To install MongoDB support with PHP use pecl as follows:

```
$ sudo pecl install mongo
```

Add the mongo.so module to php.ini.

```
extension=mongo.so
```

Then assuming you are running it on Apache, restart as follows:

```
$ apachectl stop
$ apachectl start
```

Figure 13 shows our roughly equivalent code listing in PHP.

```
<?php

$m = new Mongo();
$db = $m->selectDB("tutorial");
$employees = $db->selectCollection("employees");
$cursor = $employees->find();

foreach ($cursor as $employee) {
    echo var_export ($employee, true) . "<br/>";
}

?>
```

Figure 13 PHP code listing

The output for figure 13 is as follows:

```
Output:
array ( ‘_id’ => MongoId::__set_state(array(
‘$id’ => ‘4f964d3000b5874e7a163895’, )), ‘name’
=> ‘Rick Hightower’,
‘gender’ => ‘m’, ‘phone’ => ‘520-555-1212’, ‘age’
=> 42, )

array ( ‘_id’ => MongoId::__set_state(array(
‘$id’ => ‘4f984cae72329d0ecd8716c8’, )), ‘name’
=> ‘Diana Hightower’, ‘gender’ => ‘f’,
‘phone’ => ‘520-555-1212’, ‘age’ => 30, )

array ( ‘_id’ => MongoId::__set_state(array(
‘$id’ => ‘4f9e170580cbd54f27000000’, )), ‘gen-
der’ => ‘m’, ‘age’ => 8, ‘name’ => ‘Lucas High-
tower’,
‘phone’ => ‘520-555-1212’, )
```

The other half of the equation is in figure 14. (bottom)
The output for figure 14 is as follows:

```
Output
Rick?
array ( ‘_id’ => MongoId..., ‘name’ => ‘Rick
Hightower’, ‘gender’ => ‘m’,
‘phone’ => ‘520-555-1212’, ‘age’ => 42, )
Diana?
array ( ‘_id’ => MongoId::..., ‘name’ => ‘Diana
Hightower’, ‘gender’ => ‘f’,
‘phone’ => ‘520-555-1212’, ‘age’ => 30, )
Diana by id?
array ( ‘_id’ => MongoId::..., ‘name’ => ‘Diana
Hightower’, ‘gender’ => ‘f’,
‘phone’ => ‘520-555-1212’, ‘age’ => 30, )
```

Here is the complete PHP listing.

```
PHP complete listing

<!--?php

$m = new Mongo();
$db = $m->selectDB(“tutorial”);
$employees=$db->selectCollection(“employees”);
$cursor = $employees->find();
```

```
$cursor=$employees->find( array( "name" => "Rick Hightower"));
echo "Rick? <br /> " . var_export($cursor->getNext(), true);

$cursor=$employees->find(array("age" => array('$lt' => 35)));
echo "Diana? <br /> " . var_export($cursor->getNext(), true);

$cursor=$employees->find(array("_id" => new MongoId("4f984cce72320612f8f432bb")));
echo "Diana by id? <br /> " . var_export($cursor->getNext(), true);
```

Figure 14 PHP code listing

```
foreach ($cursor as $employee) {
 echo var_export ($employee, true) . "< br />";
}

$cursor=$employees->find( array( "name" => "Rick
Hightower"));
echo "Rick? < br /> " . var_export($cursor-
>getNext(), true);

$cursor=$employees->find(array("age"        =>
array('$lt' => 35)));
echo "Diana? < br /> " . var_export($cursor-
>getNext(), true);

$cursor=$employees->find(array("_id" => new Mong
oId("4f984cce72320612f8f432bb")));
echo "Diana by id? < br /> " . var_export($cursor-
>getNext(), true);
?>
```

If you like object-mapping to documents you should try the poorly named MongoLoid for PHP.

Additional shell commands, learning about MongoDB via the console

One of the nice things that I appreciate about MongoDB, that is missing from some other NoSQL implementation, is getting around and finding information with the console application. They seem to closely mimic what can be done in MySQL so if you are familiar with MySQL, you will feel fairly at home in the mongo console.

To list all of the databases being managed by the mongod instance, you would do the following

```
> show dbs
local      (empty)
tutorial   0.203125GB
```

To list the collections being managed by a database you could do this:

```
> show collections
employees
system.indexes
```

```
To show a list of users, you do the following:

> show users
```

To look at profiling and logging, you do this:

```
> show profile
db.system.profile is empty
Use db.setProfilingLevel(2) will enable profiling
..

> show logs
global

> show log global
Mon Apr 23 23:33:14 [initandlisten] MongoDB
starting :
  pid=11773 port=27017 dbpath=/etc/mongodb/data
64-bit…
…
Mon Apr 23 23:33:14 [initandlisten] options:
  { config: "/etc/mongodb/mongodb.config", dbpath:
"/etc/mongodb/data" }
```

Also the commands and JavaScript functions themselves have help associated with them. To see all of the operations that DBCollection supports you could do this:

```
> db.employees.help()
DBCollection help

db.employees.find().help() - show DBCursor help
db.employees.count()
db.employees.dataSize()
db.employees.distinct( key ) - eg. db.employees.
distinct( 'x' )
db.employees.drop() drop the collection
db.employees.dropIndex(name)
db.employees.dropIndexes()
db.employees.ensureIndex(keypattern[,options]) -
options is an object with these possible fields:
name, unique, dropDups
db.employees.reIndex()
db.employees.find([query],[fields]) - query is an
optional query filter. fields is optional set of
fields to return.
  e.g. db.employees.find( {x:77} , {name:1, x:1} )
db.employees.find(...).count()
db.employees.find(...).limit(n)
db.employees.find(...).skip(n)
db.employees.find(...).sort(...)
db.employees.findOne([query])
db.employees.findAndModify( { update : ... , re-
move : bool [, query: {}, sort: {}, 'new': false]
} )
db.employees.getDB() get DB object associated
with collection
db.employees.getIndexes()
db.employees.group( { key : ..., initial: ...,
reduce : ...[, cond: ...] } )
db.employees.mapReduce( mapFunction , reduce-
Function , {optional params=""} )
db.employees.remove(query)
db.employees.renameCollection( newName ,
{droptarget} ) renames the collection.
db.employees.runCommand( name , {options} )
runs a db command with the given name where
the first param is the collection name
db.employees.save(obj)
db.employees.stats()
db.employees.storageSize() - includes free
space allocated to this collection
db.employees.totalIndexSize() - size in bytes
of all the indexes
db.employees.totalSize() - storage allocated
for all data and indexes
db.employees.update(query, object[, upsert_
bool, multi_bool])
db.employees.validate( {full} ) - SLOW
db.employees.getShardVersion() - only for use
with sharding
db.employees.getShardDistribution() - prints
statistics about data distribution in the
cluster ...
```

Just to show a snake eating its own tail, you can even get help about help as follows:

```
> help
     db.help()             help on db methods
     db.mycoll.help()         help on collection
methods
     rs.help()                help on replica set
methods
     help admin           administrative help
     help connect         connecting to a db help
     help keys             key shortcuts
     help misc             misc things to know
     help mr               mapreduce

     show dbs             show database names
     show collections      show collections in cur-
rent database
     show users            show users in current da-
tabase
     show profile          show most recent system.
profile entries time>= 1ms

     show logs             show the accessible logger
names
     show log [name]     prints out the last segment
of log in memory,
> help
…
```

The MongoDB console reminds me of a cross between the MySQL console and Python's console. Once you use the console, it is hard to imagine using a NoSQL solution that does not have a decent console (hint, hint).

Conclusion

The poorly named NoSQL movement seems to be here to stay. The need to have reliable storage that can be easily queried without the schema migration pain and scalability woes of traditional RDBMS is increasing. Developers want more agile systems without the pain of schema migration. MongoDB is a good first step for developers dipping their toe into the NoSQL solution space. It provides easy-to-use tools like its console and provides many language drivers. This article covered the basics of MongoDB architecture, caveats and how to program in MongoDB for Java, PHP, JavaScript and Python developers.

I hope you enjoyed reading this article half as much as I enjoyed writing it.

Resources

Click here to view the complete list of references on InfoQ.

About the Author

Rick Hightower, CTO of Mammatus, has worked as a CTO, Director of Development and a Developer for the last 20 years. He has been involved with J2EE since its inception. He worked at an EJB container company in 1999, and did large scale Java web development since 2000 (JSP, Servlets, etc.). He has been working with Java since 1996 (Python since 1997), and writing code professionally since 1990. Rick was an early Spring enthusiast. Rick enjoys bouncing back and forth between C, PHP, Python, Groovy and Java development. He has written several programming and software development books as well as many articles and editorials for journals and magazines over the years. Lately he has been focusing on NoSQL, and Cloud Computing development.

BIG DATA AND APPLIED DATA SCIENCE QCon NEW YORK JUNE 12-14, 2013

BIG DATA AND APPLIED DATA SCIENCE
@ QCON NEW YORK: JUNE 10-14, 2013
With case studies from LinkedIn, NASA, eHarmony, and Twitter, the "Big Data" and "Applied Data Science" tracks at QCon New York bring you up to speed on the latest tools, tricks, and technologies for handling the biggest of datasets.
Save $100 when you register with promo code: "NOSQL"

"Lessons Learned Building Storm"
by Nathan Marz, Lead Engineer Backtype @Twitter

"LinkedIn Endorsements: Bootstrapping a Data Product"
by Sam Shah, Principal Engineer in LinkedIn Data Team

"eHarmony Case Study: Data Science of Love"
by Vaclav Petricek, Principal Data Scientist at eHarmony

"A Little Graph Theory for the Busy Developer"
by Jim Webber, Chief Scientist at Neo Technology and Co-Author of "Rest in Practice"

"Evolving Panorama of Data"
by Rebecca Parsons, CTO Thoughtworks

"The Art of Information Refinery at Cue"
by Daniel Gross, CEO of Cue

"Building Applications using Apache Hadoop"
by Eli Collins, Cloudera Platform Lead

CAP Twelve Years Later: How the "Rules" Have Changed

By Eric Brewer

The CAP theorem asserts that any networked shared-data system can have only two of three desirable properties. However, by explicitly handling partitions, designers can optimize consistency and availability, thereby achieving some trade-off of all three.

In the decade since its introduction, designers and researchers have used (and sometimes abused) the CAP theorem as a reason to explore a wide variety of novel distributed systems. The NoSQL movement also has applied it as an argument against traditional databases.

"CAP prohibits... perfect availability and consistency in the presence of partitions."

The CAP theorem states that any networked shared-data system can have at most two of three desirable properties:

- Consistency (C) equivalent to having a single up-to-date copy of the data;
- High availability (A) of that data (for updates); and
- Tolerance to network partitions (P).

This expression of CAP served its purpose, which was to open the minds of designers to a wider range of systems and tradeoffs; indeed, in the past decade, a vast range of new systems has emerged, as well as much debate on the relative merits of consistency and availability. The "2 of 3" formulation was always misleading because it tended to oversimplify the tensions among properties. Now such nuances matter. CAP prohibits only a tiny part of the design space: perfect availability and consistency in the presence of partitions, which are rare.

Although designers still need to choose between consistency and availability when partitions are present, there is an incredible range of flexibility for handling partitions and recovering from them. The modern CAP goal should be to maximize combinations of consistency and availability that make sense for the specific application. Such an approach incorporates plans for operation during a partition and for recovery afterward, thus helping designers think about CAP beyond its historically perceived limitations.

Why "2 of 3" is missleading

The easiest way to understand CAP is to think of two nodes on opposite sides of a partition. Allowing at least one node to update state will cause the nodes to become inconsistent, thus forfeiting C. Likewise, if the choice is to preserve consistency, one side of the partition must act as if it is unavailable, thus forfeiting A. Only when nodes communicate is it possible to preserve both consistency and availability, thereby forfeiting P. The general belief is that for wide-area systems, designers cannot forfeit P and therefore have a difficult choice between C and A. In some sense, the NoSQL movement is about creating choices that focus on availability first and consistency second; databases that adhere to ACID properties (atomicity, consistency, isolation, and durability) do the opposite. The "ACID, BASE, and CAP" sidebar explains this difference in more detail.

In fact, this exact discussion led to the CAP theorem. In the mid-1990s, my colleagues and I were building a variety of cluster-based wide-area systems (essentially early cloud computing), including search engines, proxy caches, and content distribution systems[1]. Because of both revenue goals and contract specifications, system availability was at a premium, so we found ourselves regularly choosing to optimize availability through strategies such as employing caches or logging updates for later reconciliation. Although these strategies did increase availability, the gain came at the cost of decreased consistency.

The first version of this consistency-versus-availability argument appeared as ACID versus BASE[2], which was not well received at the time, primarily because people love the ACID properties and are hesitant to give them up. The CAP theorem's aim was to justify the need to explore a wider design space-hence the "2 of 3" formulation. The theorem first appeared in fall 1998. It was published in 1999[3] and in the keynote address at the 2000 Symposium on Principles of Distributed Computing[4], which led to its proof.

As the "CAP Confusion" sidebar explains, the "2 of 3" view is misleading on several fronts. First, because partitions are rare, there is little reason to forfeit C or A when the system is not partitioned. Second, the choice between C and A can occur many times within the same system at very fine granularity; not only can subsystems make different choices, but the choice can change according to the operation or even the specific data or user involved. Finally, all three properties are more continuous than binary. Availability is obviously continuous from 0 to 100 percent, but there are also many levels of consistency, and even partitions have nuances, including disagreement within the system about whether a partition exists.

Exploring these nuances requires pushing the traditional way of dealing with partitions, which is the fundamental challenge. Because partitions are rare, CAP should allow perfect C and A most of the time, but when partitions are present or perceived, a strategy that detects partitions and explicitly accounts for them is in order. This strategy should have three steps: detect partitions; enter an explicit partition mode that can limit some operations; and initiate a recovery process to restore consistency and compensate for mistakes made during a partition.

ACID, BASE and CAP

ACID and BASE represent two design philosophies at opposite ends of the consistency-availability spectrum. The ACID properties focus on consistency and are the traditional approach of databases. My colleagues and I created BASE in the late 1990s to capture the emerging design approaches for high availability and to make explicit both the choice and the spectrum. Modern large-scale wide-area systems, including the cloud, use a mix of both approaches.

Although both terms are more mnemonic than precise, the BASE acronym (being second) is a bit more awkward: Basically Available, Soft state, Eventually consistent. Soft state and eventual consistency are techniques that work well in the presence of partitions and thus promote availability.

The relationship between CAP and ACID is more complex and often misunderstood, in part because the C and A in ACID represent different concepts than the same letters in CAP and in part because choosing availability affects only some of the ACID guarantees. The four ACID properties are: Atomicity (A). All systems benefit from atomic operations. When the focus is availability, both sides of a partition should still use atomic operations. Moreover, higher-level atomic operations (the kind that ACID implies) actually simplify recovery.

Consistency (C). In ACID, the C means that a transaction preserves all the database rules, such as unique keys. In contrast, the C in CAP refers only to single-copy consistency, a strict subset of ACID consistency. ACID consistency also cannot be maintained across partitions. Partition recovery will need to restore ACID consistency. More generally, maintaining invariants during partitions might be impossible, thus the need for careful thought about which operations to disallow and how to restore invariants during recovery.

Isolation (I). Isolation is at the core of the CAP theorem: if the system requires ACID isolation, it can operate on at most one side during a partition. Serializability requires communication in general and thus fails across partitions. Weaker definitions of correctness are viable across partitions via compensation during partition recovery.

Durability (D). As with atomicity, there is no reason to forfeit durability, although the developer might choose to avoid needing it via soft state (in the style of BASE) due to its expense. A subtle point is that, during partition recovery, it is possible to reverse durable operations that unknowingly violated an invariant during the operation.

However, at the time of recovery, given a durable history from both sides, such operations can be detected and orrected. In general, running ACID transactions on each side of a partition makes recovery easier and enables a framework for compensating transactions that can be used for recovery from a partition.

Cap-latency connection

In its classic interpretation, the CAP theorem ignores latency, although in practice, latency and partitions are deeply related. Operationally, the essence of CAP takes place during a timeout, a period when the program must make a fundamental decision: the partition decision:

- cancel the operation and thus decrease availability, or
- proceed with the operation and thus risk inconsistency.

Retrying communication to achieve consistency, for example via Paxos or a two-phase commit, just delays the decision. At some point the program must make the decision; retrying communication indefinitely is in essence choosing C over A.

Thus, pragmatically, a partition is a time bound on communication. Failing to achieve consistency within the time bound implies a partition and thus a choice between C and A for this operation. These concepts capture the core design issue with regard to latency: are two sides moving forward without communication?

This pragmatic view gives rise to several important consequences. The first is that there is no global notion of a partition, since some nodes might detect a partition, and others might not. The second consequence is that nodes can detect a partition and enter a partition mode-a central part of optimizing C and A.

Finally, this view means that designers can set time bounds intentionally according to target response times; systems with tighter bounds will likely enter partition mode more often and at times when the network is merely slow and not actually partitioned.

Sometimes it makes sense to forfeit strong C to avoid the high latency of maintaining consistency over a wide area. Yahoo's PNUTS system incurs inconsistency by maintaining remote copies asynchronously[5]. However, it makes the master copy local, which decreases latency. This strategy works well in practice because single user data is naturally partitioned according to the user's (normal) location. Ideally, each user's data master is nearby.
Facebook uses the opposite strategy[6]: the master copy is always in one location, so a remote user typically has a closer but potentially stale copy. However, when users update their pages, the update goes to the master copy directly as do all the user's reads for a short time, despite higher latency. After 20 seconds, the user's traffic reverts to the closer copy, which by that time should reflect the update.

Cap confusion

Aspects of the CAP theorem are often misunderstood, particularly the scope of availability and consistency, which can lead to undesirable results. If users cannot reach the service at all, there is no choice between C and A except when part of the service runs on the client. This exception, commonly known as disconnected operation or offline mode, is becoming increasingly important. Some HTML5 features - in particular, on-client persistent storage - make disconnected operation easier going forward. These systems normally choose A over C and thus must recover from long partitions.

Scope of consistency reflects the idea that, within some boundary, state is consistent, but outside that boundary all bets are off. For example, within a primary partition, it is possible to ensure complete consistency and availability, while outside the partition, service is not available.

Paxos and atomic multicast systems typically match this scenario. In Google, the primary partition usually resides within one datacenter; however, Paxos is used on the wide area to ensure global consensus, as in Chubby, and highly available durable storage, as in Megastore.

Independent, self-consistent subsets can make forward progress while partitioned, although it is not possible to ensure global invariants. For example, with sharding, in which designers prepartition data across nodes, it is highly likely that each shard can make some progress during a partition. Conversely, if the relevant state is split across a partition or global invariants are necessary, then at best only one side can make progress and at worst no progress is possible.

Does choosing consistency and availability (CA) as the "2 of 3" make sense? As some researchers correctly point out, exactly what it means to forfeit P is unclear. Can a designer choose not to have partitions? If the choice is CA, and then there is a partition, the choice must revert to C or A. It is best to think about this probabilistically: choosing CA should mean that the probability of a partition is far less than that of other systemic failures, such as disasters or multiple simultaneous faults.

Such a view makes sense because real systems lose both C and A under some sets of faults, so all three properties are a matter of degree. In practice, most groups assume that a datacenter (single site) has no partitions within, and thus design for CA within a single site; such designs, including traditional databases, are the pre-CAP default. However, although partitions are less likely within a datacenter, they are indeed possible, which makes a CA goal problematic. Finally, given the high latency across the wide area, it is relatively common to forfeit perfect consistency across the wide area for better performance.

Another aspect of CAP confusion is the hidden cost of forfeiting consistency, which is the need to know the system's invariants. The subtle beauty of a consistent system is that the invariants tend to hold even when the designer does not know what they are. Consequently, a wide range of reasonable invariants will work just fine. Conversely, when designers choose A, which requires restoring invariants after a partition, they must be explicit about all the invariants, which is both challenging and prone to error. At the core, this is the same concurrent updates problem that makes multithreading harder than sequential programming.

Managing partitions

The challenging case for designers is to mitigate a partition's effects on consistency and availability. The key idea is to manage partitions very explicitly, including not only detection, but also a specific recovery process and a plan for all of the invariants that might be violated during a partition. This management approach has three steps:

- Detect the start of a partition,
- Enter an explicit partition mode that may limit some operations, and
- Initiate partition recovery when communication is restored.

The last step aims to restore consistency and compensate for mistakes the program made while the system was partitioned.

Figure 1 shows a partition's evolution. Normal operation is a sequence of atomic operations, and thus partitions always start between operations. Once the system times out, it detects a partition, and the detecting side enters partition mode. If a partition does indeed exist, both sides enter this mode, but one-sided partitions are possible. In such cases, the other side communicates as needed and either this side responds correctly or no communication was required; either way, operations remain consistent. However, because the detecting side could

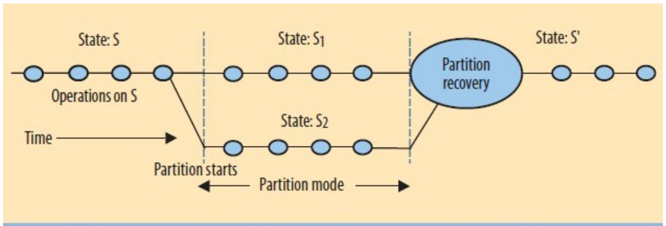


Figure 1. The state starts out consistent and remains so until a partition starts. To stay available, both sides enter partition mode and continue to execute operations, creating concurrent states S_1 and S_2, which are inconsistent. When the partition ends, the truth becomes clear and partition recovery starts. During recovery, the system merges S_1 and S_2 into a consistent state S′ and also compensates for any mistakes made during the partition.

have inconsistent operations, it must enter partition mode. Systems that use a quorum are an example of this one-sided partitioning. One side will have a quorum and can proceed, but the other cannot. Systems that support disconnected operation clearly have a notion of partition mode, as do some atomic multicast systems, such as Java's JGroups.

Once the system enters partition mode, two strategies are possible. The first is to limit some operations, thereby reducing availability. The second is to record extra information about the operations that will be helpful during partition recovery. Continuing to attempt communication will enable the system to discern when the partition ends.

Which operations should proceed?

Deciding which operations to limit depends primarily on the invariants that the system must maintain. Given a set of invariants, the designer must decide whether or not to maintain a particular invariant during partition mode or risk violating it with the intent of restoring it during recovery. For example, for the invariant that keys in a table are unique, designers typically decide to risk that invariant and allow duplicate keys during a partition. Duplicate keys are easy to detect during recovery, and, assuming that they can be merged, the designer can easily restore the invariant.

For an invariant that must be maintained during a partition, however, the designer must prohibit or modify operations that might violate it. (In general, there is no way to tell if the operation will actually violate the invariant, since the state of the other side is not knowable.) Externalized events, such as charging a credit card, often work this way. In this case, the strategy is to record the intent and execute it after the recovery. Such transactions are typically part of a larger workflow that has an explicit order-processing state, and there is little downside to delaying the operation until the partition ends. The designer forfeits A in a way that users do not see. The users know only that they placed an order and that the system will execute it later.

More generally, partition mode gives rise to a fundamental user-interface challenge, which is to communicate that tasks are in progress but not complete. Researchers have explored this problem in some detail for disconnected operation, which is just a long partition. Bayou's calendar application, for example, shows potentially inconsistent (tentative) entries in a different color.[13] Such notifications are regularly visible both in workflow applications, such as commerce with e-mail notifications, and in cloud services with an offline mode, such as Google Docs.

One reason to focus on explicit atomic operations, rather than just reads and writes, is that it is vastly easier to analyze the impact of higher-level operations on invariants. Essentially, the designer must build a table that looks at the cross product of all operations and all invariants and decide for each entry if that operation could violate the invariant. If so, the designer must decide whether to prohibit, delay, or modify the operation. In practice, these decisions can also depend on the known state, on the arguments, or on both. For example, in systems with a home node for certain data, 5 operations can typically proceed on the home node but not on other nodes.

The best way to track the history of operations on both sides is to use version vectors, which capture the causal dependencies among operations. The vector's elements are a pair (node, logical time), with one entry for every node that has updated the object and the time of its last update. Given two versions of an object, A and B, A is newer than B if, for every node in common in their vectors, A's times are greater than or equal to B's and at least one of A's times is greater.

If it is impossible to order the vectors, then the updates were concurrent and possibly inconsistent. Thus, given the version vector history of both sides, the system can easily tell which operations are already in a known order and which executed concurrently. Recent work[14] proved that this kind of causal consistency is the best possible outcome in general if the designer chooses to focus on availability.

Partition recovery

At some point, communication resumes and the partition ends. During the partition, each side was available and thus making forward progress, but partitioning has delayed some operations and violated some invariants. At this point, the system knows the state and history of both sides because it kept a careful log during partition mode. The state is less useful than the history, from which the system can deduce which operations actually violated invariants and what results were externalized, including the responses sent to the user. The designer must solve two hard problems during recovery:

- The state on both sides must become consistent, and
- There must be compensation for the mistakes made during partition mode.

It is generally easier to fix the current state by starting from the state at the time of the partition and rolling forward both sets of operations in some manner, maintaining consistent state along the way. Bayou did this explicitly by rolling back the database to a correct time and replaying the full set of operations in a well-defined, deterministic order so that all nodes reached the same state.[15] Similarly, source-code control systems such as the Concurrent Versioning System (CVS) start from a shared consistent point and roll forward updates to merge branches.

Most systems cannot always merge conflicts. For example, CVS occasionally has conflicts that the user must resolve manually, and wiki systems with offline mode typically leave conflicts in the resulting document that require manual editing.[16]

Conversely, some systems can always merge conflicts by choosing certain operations. A case in point is text editing in Google Docs,[17] which limits operations to applying a style and adding or deleting text. Thus, although the general problem of conflict resolution is not solvable, in practice, designers can choose to constrain the use of certain operations during partitioning so that the system can automatically merge state during recovery. Delaying risky operations is one relatively easy implementation of this strategy.

Using commutative operations is the closest approach to a general framework for automatic state convergence. The system concatenates logs, sorts them into some order, and then executes them. Commutativity implies the ability to rearrange operations into a preferred consistent global order. Unfortunately, using only commutative operations is harder than it appears; for example, addition is commutative, but addition with a bounds check is not (a zero balance, for example).

Recent work by Marc Shapiro and colleagues at INRIA[18,19] has greatly improved the use of commutative operations for state convergence. The team has developed commutative replicated data types (CRDTs), a class of data structures that provably converge after a partition, and describe how to use these structures to ensure that all operations during a partition are commutative, or represent values on a lattice and ensure that all operations during a partition are monotonically increasing with respect to that lattice.
The latter approach converges state by moving to the maximum of each side's values. It is a formalization and improvement of what Amazon does with its shopping cart:[20] after a partition, the converged value is the union of the two carts, with union being a monotonic set operation. The consequence of this choice is that deleted items may reappear.

However, CRDTs can also implement partition-tolerant sets that both add and delete items. The essence of this approach is to maintain two sets: one each for the added

“MongoDB is a good first step for developers dipping their toe into the NoSQL solution space.”

and deleted items, with the difference being the set’s membership. Each simplified set converges, and thus so does the difference. At some point, the system can clean things up simply by removing the deleted items from both sets. However, such cleanup generally is possible only while the system is not partitioned. In other words, the designer must prohibit or postpone some operations during a partition, but these are cleanup operations that do not limit perceived availability. Thus, by implementing state through CRDTs, a designer can choose A and still ensure that state converges automatically after a partition.

Compensation issues in an automated teller machine

In the design of an automated teller machine (ATM), strong consistency would appear to be the logical choice, but in practice, A trumps C. The reason is straightforward enough: higher availability means higher revenue. Regardless, ATM design serves as a good context for reviewing some of the challenges involved in compensating for invariant violations during a partition.

The essential ATM operations are deposit, withdraw, and check balance. The key invariant is that the balance should be zero or higher. Because only withdraw can violate the invariant, it will need special treatment, but the other two operations can always execute.

The ATM system designer could choose to prohibit withdrawals during a partition, since it is impossible to know the true balance at that time, but that would compromise availability. Instead, using stand-in mode (partition mode), modern ATMs limit the net withdrawal to at most k, where k might be $200. Below this limit, withdrawals work completely; when the balance reaches the limit, the system denies withdrawals. Thus, the ATM chooses a sophisticated limit on availability that permits withdrawals but bounds the risk.

When the partition ends, there must be some way to both restore consistency and compensate for mistakes made while the system was partitioned. Restoring state is easy because the operations are commutative, but compensation can take several forms. A final balance below zero violates the invariant. In the normal case, the ATM dispensed the money, which caused the mistake to become external. The bank compensates by charging a fee and expecting repayment. Given that the risk is bounded, the problem is not severe. However, suppose that the balance was below zero at some point during the partition (unknown to the ATM), but that a later deposit brought it back up. In this case, the bank might still charge an overdraft fee retroactively, or it might ignore the violation, since the customer has already made the necessary payment.

In general, because of communication delays, the banking system depends not on consistency for correctness, but rather on auditing and compensation. Another example of this is “check kiting,” in which a customer withdraws money from multiple branches before they can communicate and then flees. The overdraft will be caught later, perhaps leading to compensation in the form of legal action.

Acknowledgments

I thank Mike Dahlin, Hank Korth, Marc Shapiro, Justin Sheehy, Amin Vahdat, Ben Zhao, and the IEEE Computer Society volunteers for their helpful feedback on this work.

References

Click here to view the complete list of references on InfoQ.

About the Author

Eric Brewer is a professor of computer science at the University of California, Berkeley, and vice president of infrastructure at Google. His research interests include cloud computing, scalable servers, sensor networks, and technology for developing regions. He also helped create USA.gov, the official portal of the federal government. Brewer received a PhD in electrical engineering and computer science from MIT. He is a member of the National Academy of Engineering. Contact him at brewer@cs.berkeley.edu

NoSQL: Past, Present, Future

Presentation Summary By Abel Avram

During the session NoSQL: Past, Present, Future held at QCon San Francisco 2012, Eric Brewer, the author of the CAP Theorem, briefly traced the origins of NoSQL databases and their evolution to the present time, then took a peek at their future along with some considerations on partitioning.

The first NoSQL database Brewer is aware of was a Navigational Database (ND) created by Charles Bachman, an ACM Turing Award winner. It was the pre-SQL era so the database was not labeled as 'NoSQL', but it had many things in common with today's key-value stores and graph databases. ND was "a database made of pointers and records: you traverse the pointers to get the records and it's very manual," but it had " tremendously high performance" being used for airline reservations worldwide until 2005 when it was retired.

While quite useful at its time, ND had a major drawback, according to Brewer. "Because everything is pointers and all the pointers are in the code or on the disk, it's very hard to change anything so it really has no encapsulation whatsoever and [is] really hard to evolve." Something had to be done, being clear that the code had to be separated from the data.

Further, the evolution of data storage and data management technologies took two different paths. One was built around a top-down data model that would allow data to be moved off one storage device to another while still performing reasonably well, which eventually happened with some help from "Moore's law and increased computing power." Data was to be accessed through an API, a query language, and the first implementation was IBM System R with its SEQUEL, later to be called SQL.

At about the same time, Unix was created, and its main idea was to provide an efficient file system that would persist data to the disk. Unix was built from the bottom up using layers and one of them was a key-value store called DBM, running on top of the file system. This represented the beginning of the other development path in data storage which later came to be known as NoSQL.

Because of its influence from the Unix tradition, NoSQL takes a different approach to the problem of data persistence, taking a "systems view of how to build data-storage systems versus a database view of the data storage systems. There are many things that fit this model that are directly literally from the Unix tradition," according to Brewer.

What matters in the SQL world is to have a well-defined data model that provides ACID transactions and clean semantics; the implementation is not important as long as the system provides reasonable performance. On the other side, the systems view emphasizes low-level implementation, modularity, having transactions but with the ability to change the implementation. NoSQL is a reaction to the monolithic database and all the problems associated with object-relational mapping.

The ideal database should have a clean model implemented in a well-structured modular system that allows "you to get to the inside pieces or the top-level view. We don't actually have this artifact today" and "it's not clear yet how to mix them very well," according to Brewer. NoSQL systems borrow concepts from SQL databases such as high-level models, operations, multi component transactions, indices, and joins because they are good and we have the raw power necessary to do them.

During the pre-Internet era ACID was "sacred" and BASE was disregarded. It was a time when availability expectations were low, when credit card transactions were done on paper. But later, availability became very important especially for online systems that had to satisfy client requests in real time. It was the time for BASE to be taken more seriously.
The CAP theorem was formulated for that reason, to emphasize the fact that you cannot have it all: you cannot have both availability and instantaneous consistency when it comes to writes in partitioned systems. And some companies internalized the idea and implemented it as it was the case with Dynamo at Amazon.

CAP does not imply you should forfeit consistency, but you should suspend it temporarily while you are partitioned and get it back when you close the partition. Durability is also useful and needed to fix partitions. Durability can be achieved in single-site transactions which are atomic operations taking place without any partitioning within that site. That's how Google BigTable works, with transactions limited to one row or a set of columns within a row. And the

“When a timeout occurs you need to choose between consistency and availability.“

respective columns are co-located within a set of nodes that are not partitioned.

Partitions are temporary, and you need to consider what to do when recovering from that state. It is possible to provide two different types of services, for the normal state and for the partitioned one, each with its own SLA. You can detect when a system is partitioned. When a timeout occurs you need to choose between consistency and availability. You either want to preserve data consistency and you wait until the write is performed or you commit locally and try to reconcile the differences later achieving eventual consistency. Another approach is implementing lazy consistency: write locally and reconcile differences only when a read is requested.

When a system is partitioned, some of the operations may be allowed while some may not, especially those for which consistency is a must. The CAP theorem does not specify whether all operations are forbidden or allowed during partitioning.

When partitioning ends, one should proceed with partition recovery, the main goal being to restore consistency. This can be done through a combination of rollbacks, merging, commutative operations, etc. The recovery is successful if there were no “bad” side effects associated with reconciled operations. A system is well designed when there are no such side effects, such as telling a client he has his purchase secured when in fact the item is out of stock. Or worse, the client’s credit card is charged and the item is not on inventory.

ACID was used instead of BASE for banking operations in the past because such transactions were said to need to be consistent at all times. But there has been the case of ATM operations where transactions may be temporarily inconsistent and the system partitioned. It happens sometimes that ATMs are disconnected from the network and they keep serving clients instead of being out of order. A partition is created during that time, and it is easily reconciled later because operations are commutative - increments or decrements -which can be reordered. This is a case where availability is chosen over consistency.

One of the problems is possible negative side effects: the client wants to withdraw an amount of money larger than what’s actually available in his or her bank account. The system will discover later what happened but for the bank it won’t be good news. Usually, this is not a huge problem because clients tend to add money to their accounts later. The bank could also limit the amount withdrawn when the ATM is disconnected to limit damages. This example shows that consistency is not paramount even for banking, and partitioning may involve assuming a risk.

In conclusion, a realistic database model, which in fact is currently used by some businesses, is to tolerate partitioning but to establish beforehand what the system can and cannot do during that state. And to have a recovery plan, including some actions to be taken when there are negative side effects. A model where the system is always consistent and never makes mistakes is unrealistic.

If CAP told people in the beginning to choose between consistency and availability, now the message is “CAP disallows only a fraction of what’s possible”, that is, you can’t have full consistency with full availability. Other than that, ACID systems are improving their availability while BASE systems their consistency. Both systems are converging, heading in the same direction. The ideal solution may be some kind of combination of the two but this may not always be easy to achieve.

Related NoSQL Presentations and Interviews on InfoQ.com

NoSQL Presentations

Connecting Millions of Mobile Devices to the Cloud

http://www.infoq.com/presentations/Couchbase-Syncpoint
Damien Katz explains how Couchbase Syncpoint provides real time data synchronization capabilities between multiple mobile devices and the cloud.

Erlang, the Language from the Future?

http://www.infoq.com/presentations/Erlang-Pros-Cons
Damien Katz explains the benefits and drawbacks of using Erlang, why this language is from the future and why Couchbase has migrated some of the CouchDB's initial Erlang code to C/C++.

NoSQL Database Technology: A Survey and Comparison of Systems

http://www.infoq.com/presentations/NoSQL-Survey-Comparison
James Phillips presents the origins of NoSQL, followed by a comparison of various NoSQL solutions and ending with an architect's view of Couchbase.

Why CouchDB?

http://www.infoq.com/presentations/Why-CouchDB
Benjamin Young introduces CouchDB, its schema-less data store, REST API, HTTP-based replication, plugins such as R-tree and GeoCouch, ways to scale it out and then scaling down with mobile solutions.

A Little Graph Theory for the Busy Developer

http://www.infoq.com/presentations/neo4j-graph-theory
Jim Webber explores data analytic techniques using social graph properties inspired by anthropology and sociology, extracting online business intelligence from graph matching.

MongoDB Large-Scale Data Centric Architectures

http://www.infoq.com/presentations/MongoDB-Design
Kenny Gorman provides advice on designing systems using MongoDB in order to avoid some of the pitfalls lurking along the way.

MySQL to NoSQL: Data Modeling Challenges in Supporting Scalability

http://www.infoq.com/presentations/MySQL-NoSQL-Data-Modeling
Kenneth M. Anderson shares some of the data-modeling issues encountered while transitioning from a relational database to NoSQL.

Data Modeling with Graphs

http://www.infoq.com/presentations/Data-Modeling-Graphs
Peter Bell presents several patterns for modeling and retrieving data from graph databases using Neo4j in his examples.

NoSQL Interviews

InfoQ: Rich Hickey on Datomic: Datalog, Databases, Persistent Data Structures

http://www.infoq.com/interviews/hickey-datomic

Rich Hickey and Justin Sheehy about Datastores, NoSql and CAP

http://www.infoq.com/interviews/rich-Hickey-and-justin-sheehy-about-datastores,-nosql-and-cap

Debasish Ghosh on Functional Programming, NoSQL

http://www.infoq.com/interviews/ghosh-functional