NoSQL

Thomas Neumann
What are NoSQL databases?

- hard to say
- more a theme than a well defined thing

Usually some or all of the following:
- no SQL interface
- no relational model / no schema
- no joins, emphasize on key/value pairs
- scale out to many machines
- weak or no consistency guarantees
Why not relational databases?

Some commonly stated reasons:

- RDBMS are hard to use
- do not scale to "web-scale"
- relational model is too restrictive
- NoSQL is faster, scales better

Some of this is true (as we will see), but most likely will not affect you!
Illustrational Web Video

MongoDB is web scale
http://www.xtranormal.com/watch/6995033
The Performance Argument

Voter benchmark: People call to vote for American Idol
  • at most 12 votes are counted per caller-id
  • very simple transaction model

On a 8 core Intel Xeon X5570 with 64GB main memory:
  • MongoDB: ca. 10,000 transactions per second
  • relational main-memory database: ca. 1,000,000 transactions per second

Do not blindly follow a hype, do the math!
Sucess stories from the net

- Why we chose MongoDB [...] Very easy to install. [...] Very easy replication
- [...] We cut down the names to 2-3 characters. This is a little more confusing in the code but the disk storage savings are worth it [...] a massive saving.
- [...] Was it the right move? Yes. MongoDB has been an excellent choice [...] MongoDB is going to be very cool!

- MongoDB works fine, but the same query is 25 times faster in PostgreSQL
- [...] MongoDB will win once I have 26 machines
Technical Arguments in favor of NoSQL

CAP-Theorem: In a distributed system you can only have two of the following

- Consistency
  - all nodes see the same data at the same time
- Availability
  - node failures do not prevent survivors from continuing to operate
- Partition Tolerance
  - the system continues to operate despite arbitrary message loss

Basis for the claim the RDBMS are not "web-scale"
Scalability

How to scale to thousands of nodes?

- traditional RDBMS usually scale to less than 100 node
- transaction semantic requires a lot of coordination
- two phase commit is expensive
- $O(n^2)$ network connections
- does not scale to thousands of nodes

Partitioning helps, but usually requires human interaction.
Key/Value - Stores

Life would be much simpler if we only stored key/value pairs

- only (or mostly) point-access
- transactions operate on a single item
- allows for simple partitioning
- by spreading keys over nodes we distribute the data
- usually scales perfectly

Life is much simpler if you only care about individual values...
Distributed Hash Tables

Basis for many distributed storage schemes:

- spread a hash table over a large number of nodes
- nodes can enter and leave (more or less) at will
- nodes know only a few other nodes
- offers scalable distributed storage

Many algorithms exist: Chord, Pastry, P-Grid, etc.

- usual idea: hash nodes into hash domain
- nodes responsible for hash values near to them
Distributed Hash Tables - Chord

Both items and nodes are hashed into ring structure

Finger tables similar to skip lists
Expressiveness

- DHTs are one giant Key→Value table
- only three operations: lookup, insert, delete
- each operations is limited to a single data item
- range queries not supported efficiently

This is a severe limitation. Scalability is obtained by eliminating functionality
What about Consistency?

We want our database to be consistent

- as long as transactions operate on single items (note: **strong restriction**) life is relatively simple
- one node is responsible for the data item
- as long as all changes are atomic or idempotent everything is fine

But: nodes will be replicated for availability

- ensuring consistency adds the same costs as in standard distributed RDBMSs
- most systems aim at "eventual consistency"
- after waiting long enough (without updates in between), all replicas will have the same value

This is usually unacceptable if the data is valuable (e.g., involves money)!
What about Multi-Item Transactions?

Short answer: not supported

Long answer: not supported very well

- one can ignore the issue and run multi single-level transactions
- completely messes up consistency
- some systems offer explicit locking
- some problems as in distributed RDBMSs
How to Query the Data

Data is spread over thousands of nodes
  • point query are supported by DHTs
  • range queries are not
  • aggregation queries are very important

Requires some very heavy machinery inside the NoSQL database
  • query response time usually multiple seconds, even minutes
  • not really suited for interactive queries
  • data will change during query execution
  • usually queries inconsistent data
Map/Reduce

Programming paradigm that allows for easy parallelization. Sequence of two operations:

1. Map: \((k_1, v_1) \rightarrow \text{list}(k_2, v_2)\)
   - constructs key-value pairs from input pair
   - can be computed in parallel, no interaction

2. Reduce: \((k_2, \text{list}(v_2)) \rightarrow \text{list}(k_3, v_3)\)
   - reduces all \(k_2\) pairs into one (or more) value
   - different \(k_2\)s can be parallelized

Simple, scalable scheme, but involves massive movement of data
Word count is the classical example:

1. map\((documentId, document)\)
   for each word \(w\) in \(document\)
   emit \((w,1)\)

2. reduce\((word, counts)\)
   \(count = 0\)
   for each \(c\) in \(counts\)
   \(count + = c\)
   emit\((word, count)\)

Computes the frequency of each word
Map/Reduce - Database Queries

Can also be used to query distributed key/value stores:

1. map($customerId, customerData$)
   emit ($customerId, customerData.amount$)

2. reduce($customerId, revenues$)
   \[
   sum = 0 \\
   \text{for each } r \text{ in } revenues \\
   \quad sum + = r \\
   \text{emit}(customerId, r)
   \]

3. reduce($customerId, revenue$)
   \[
   \text{if } revenue > 10000 \\
   \quad \text{emit } (customerId, revenue)
   \]

- executed across all nodes
- very heavy operation
There is a huge number of NoSQL systems around

- **BigTable**
  - key/value store used inside Google, row/column/time dimensions, slicing
- **Cassandra**
  - key/value store with tunable consistency
- **MongoDB**
  - document centric, JavaScript driven, relatively rich queries
- **CouchDB**
  - document centric, JavaScript driven, MVCC
- **Dynamo, Project Voldemort, Hbase, ...**

Unfortunately all incompatible, all different in some aspects
Who needs this ultra-scalability?

Going fully "web-scale" makes sense in a few cases:

- the data amount is huge
  - petabytes of data
- consistency is not important
  - click streams, not payment data
- access is mostly single-item
  - more complicated queries are expensive

But: very few companies have these characteristics
The real reason why (some) people use NoSQL: Money

A 1TB main-memory machine costs ca. 60K

- most people do not have large amounts of data anyway
- or if they have, the data is not that important
- enterprise database systems are expensive
- NoSQL products tend to be free or cheap
- startups do not have money

But is this really an argument for NoSQL?
Conclusion

NoSQL is a fuzzy term, but usually

- stores non-relational data
- aims at scalability to thousands of nodes
- sacrifices consistency
- support mainly simple queries efficiently

Mainly makes sense if

- data is really huge
- and not very valuable

Otherwise, use a RDBMS!