

NoSQL

Non-relational next generation
operational datastores and databases

Why NoSQL?

- Massive size (see Google BigTable)
- Query complexity (see Digg)
- Data doesn't map to table-based schema

<http://infogrid.org/blog/2009/11/the-nosql-business-and-use-cases/>

Scaling out, not up

- No joins
- Simple transactions
- Allows simple massive horizontal scaling

NoSQL = ?

- non-relational
- distributed
- horizontal scalable
- schema-free
- replication support
- eventual consistency

<http://nosql-database.org/>

New Data Models

- Key/value stores
- Tabular
- Document oriented

Lots of options

- Hadoop/HBase
- Cassandra
- CouchDB
- MongoDB
- Riak
- Terrastore
- Redis
- MemcacheDB
- Tokyo Cabinet/Tyrant
- FleetDB
- Amazon SDB
- Dynamo
- Voldemort
- Neo4J
- InfoGrid
- **many, many more...**

Lots of options

- Hadoop/HBase
- Cassandra
- CouchDB
- MongoDB
- Riak
- Terrastore
- Redis
- MemcacheDB
- Tokyo Cabinet/Tyrant
- FleetDB
- Amazon SDB
- Dynamo
- Voldemort
- Neo4J
- InfoGrid
- **many, many more...**

Apache Cassandra

- Simple fault tolerance, decentralisation and high availability
- Scale read and write throughput by adding more nodes (can be on the fly)
- Eventually consistent
- More than key/value store, less than document database

Cassandra concepts

Keyspace: `'Anothersocnet'`

Column family: `:Users`

Columns:

```
{ name: 'username',  
  value: 'alanb',  
  timestamp: 12345678 }
```

Super columns:

```
{ 'friends': {  
  'bert': 'bert@pigeons.com',  
  'ernie': 'ernie@duckie.com' } }
```

Cassandra

- Insert rows of data into a ColumnFamily with unique key
- Data is sorted on insert by the column/SC name
 - Most often use TimeUUID as unique key
- Find by specifying Keyspace, ColumnFamily, row key (and optionally start key and number of results).

:Users

`123': { `login': `alanb' }

`456': { `login': `freddy' }

:Statuses

`1': { `user_id': `123',
`text': `first post!' }

:UserRelationships

`123': { `user_timeline': { UUID.new: 1 } }

`123': { `user_timeline': { UUID.new: 2 } }

MongoDB

- Document-oriented storage
- Dynamic queries & MapReduce
- Replication and fail-over
- Auto-sharding
- Fast, in-place updates
- Efficient storage of binary large objects

Representing a blog post

```
{title: 'Is NoSQL over-hyped?',  
  author: 'alanb',  
  ts: Date('09-Mar-10 20:15'),  
  tags: ['databases', 'nosql'],  
  comments: [  
    {author: 'FredF',  
      comment: 'Like, totally' }  
    {author: 'BarneyR',  
      comment: 'Hell, no' }  
  ]  
}
```

Simple Queries

Submit a query as a JSON document:

```
db.posts.find({tags: 'nosql'})
```

```
db.posts.find(  
  {author: 'alanb', tags: 'nosql'})
```

```
db.posts.find(  
  {author: 'alanb', tags: 'nosql'},  
  {comments: 0})
```

```
db.posts.find(  
  {'comments.author': 'FredF' } )
```

Using JavaScript in Queries

You can run JavaScript on the server:

```
function erase_comments() {  
  db.posts.find().forEach( function(obj) {  
    delete obj.comments;  
    db.post.save(obj);  
  } );  
}
```

```
db.eval(my_erase);
```

Map/reduce

Allows batch manipulation and aggregation of data using JavaScript:

map function formats data as key-value pairs:

```
var map=function({
  this.comments.forEach(function(z) {
    emit(z.author, {count:1})
  })
});
```

Applied to each document in your collection.

Map/reduce

Reduce function invoked for each key with array of values:

```
var reduce=function(key, values) {  
  var total=0;  
  for (var i=0; i<values.length; i++)  
    total +=values[i].count;  
  return {count:total};  
};
```

Map/reduce

```
var op = db.posts.mapReduce(map,  
reduce);
```

```
> db[res2.result].find();
```

```
{ "_id" : "BarneyR", "value" :  
  { "count" : 6 } }
```

```
{ "_id" : "FredF", "value" :  
  { "count" : 3 } }
```

<http://incubator.apache.org/cassandra/>

<http://www.mongodb.org/>

Essential reading:

<http://nosql-database.org/>

<http://nosql.mypopescu.com/>

<http://twitter.com/nosqlupdate>

<http://pinboard.in/u:alanb/t:nosql/>

@alanb / <http://alanbradburne.com>