### MongoDB Dashboard

<table>
<thead>
<tr>
<th>Databases</th>
<th>Total data size</th>
<th>Total index size</th>
<th>Total inserts</th>
<th>Queries</th>
<th>Updates</th>
<th>Deletes</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>74.96 GB</td>
<td>25.51 GB</td>
<td>876.70</td>
<td>729.27</td>
<td>3,515.47</td>
<td>0.00</td>
</tr>
</tbody>
</table>

**Alerts**

- **rs2a** - System: Disk usage (/mongodbdata): 80%

####pb1 - pb1a

<table>
<thead>
<tr>
<th>Node</th>
<th>Data Size</th>
<th>Index Size</th>
<th>Inserts</th>
<th>Updates</th>
<th>Deletes</th>
<th>Queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>pb1a</td>
<td>42%</td>
<td>0.31 GB</td>
<td>1.42 GB</td>
<td>1.66 GB</td>
<td>7,794</td>
<td>0 / 0</td>
</tr>
<tr>
<td>pb1b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

####set1 - rs1b

<table>
<thead>
<tr>
<th>Node</th>
<th>Data Size</th>
<th>Index Size</th>
<th>Inserts</th>
<th>Updates</th>
<th>Deletes</th>
<th>Queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>rs1a</td>
<td>40%</td>
<td>13.00 MB</td>
<td>40.00 MB</td>
<td>7.55 GB</td>
<td>7,168</td>
<td>0 / 0</td>
</tr>
<tr>
<td>rs1b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

####set2 - rs2a

<table>
<thead>
<tr>
<th>Node</th>
<th>Data Size</th>
<th>Index Size</th>
<th>Inserts</th>
<th>Updates</th>
<th>Deletes</th>
<th>Queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>rs2a</td>
<td>95%</td>
<td>24.42 GB</td>
<td>72.10 GB</td>
<td>-8.98 GB</td>
<td>7,165</td>
<td>0 / 0</td>
</tr>
<tr>
<td>rs2b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

####set3 - rs3a

<table>
<thead>
<tr>
<th>Node</th>
<th>Data Size</th>
<th>Index Size</th>
<th>Inserts</th>
<th>Updates</th>
<th>Deletes</th>
<th>Queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>rs3a</td>
<td>19%</td>
<td>0.77 GB</td>
<td>1.40 GB</td>
<td>6.79 GB</td>
<td>7,064</td>
<td>0 / 0</td>
</tr>
<tr>
<td>rs3b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
- Examining each database in turn to look at 3 important factors for production – scaling reads and writes, where bottlenecks can occur and how to deal with redundancy and failover.
- This isn’t a beginner introduction to each database and some knowledge is assumed, although I will go through a basic introduction for each.
Case studies
• Document stores

Designed to store a structured document with fields and values of different types. Documents can have subdocuments and internal structures like arrays. This is in comparison to k/v stores.
• Document stores
• Flexible schema

Unlike traditional RDBMs where you have to define the table structure upfront and changes can require long processes, these databases don’t have a defined structure and every document within a table can have different fields. Schemaless? Semantic.
• Document stores
• Flexible schema
• Purpose

Not general purpose databases – they were all designed with specific purposes. MongoDB for large data and fast inserts, Cassandra for big data and CouchDB for the map reduce query model and how it does replication.
• Document stores

• Flexible schema

• Purpose

• NoSQL

None of them use SQL for querying.
It’s a little different.
ビュービュー
ほかの部屋で火事です
• **Implementation**

MongoDB uses C++, CouchDB uses Erlang and Cassandra uses Java. MongoDB the only one that is truly native with the other 2 having various dependencies.
MongoDB was originally not built for single server durability and although they do now have journaling which provides much more data security, it’s still recommended to run in replication for proper durability. CouchDB is ACID compliant so guarantees data is written, and with Cassandra this is configurable through replication.
Cassandra and MongoDB are similar in that they allow ad-hoc queries against any collection, which are significantly sped up by using indexes. In contrast CouchDB requires you understand your queries in advance by executing map/reduce queries against them, then querying the results of those.
• Global lock

Scaling writes

mongoDB
• Global lock
• Concurrency

Scaling writes

Less of an actual problem
Yielding in 2.0
Separate mongod
Scaling writes

- Global lock
- Concurrency
- Sharding
Scaling writes

MongoDB diagram showing a replica set and shard setup.

- Config servers: C1, C2, C3
- Shard 1, 2, 3, 4, 5
- MongoDB processes:
  - Shard 1: mongod, mongod, mongod
  - Shard 2: mongod, mongod, mongod
  - Shard 3: mongod, mongod, mongod
  - Shard 4: mongod, mongod
- Client connection to mongos
Scaling reads
• Replica slaves
• Replica slaves
• Consistency

Scaling reads

Replication delay. Internal network, WAN
- Replica slaves
- Consistency
- `w` flag / tags

Scaling reads

mongoDB
Bottlenecks

www.flickr.com/photos/comedynose/4388430444/
Bottlenecks

• RAM

www.flickr.com/photos/comedynose/4388430444/
### Bottlenecks

**RAM**

<table>
<thead>
<tr>
<th>What?</th>
<th>Should it be in memory?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indexes</td>
<td>Always</td>
</tr>
<tr>
<td>Data</td>
<td>If you can</td>
</tr>
</tbody>
</table>
Bottlenecks

• Disk i/o

www.flickr.com/photos/daddo83/3406962115/
Bottlenecks

• Disk i/o

When not in RAM
Disk seeks = slow
Bottlenecks

- Disk i/o
- Mount points

Separate databases onto different disks or arrays of disks
Separate the journal
Bottlenecks

- Disk i/o

SSDs significantly faster than spinning disks
Could use SSDs for journal
• EC2

Bottlenecks

mongoDB
Local storage is faster as it’s not over the network but it’s not ephemeral.
Bottlenecks

- EC2
- Local storage
- EBS: RAID10 4-8 volumes

EBS works best in RAID10 with 4–8 volumes. However, EBS isn’t consistent on the performance, so it’s really important to use RAM where possible.
Bottlenecks

- EC2
- Local storage
- EBS: RAID10 4-8 volumes
- i/o: rand but not necessarily sequential

More volumes = better random i/o performance but not necessarily sequential performance.
### Amazon EC2 Instance Types

<table>
<thead>
<tr>
<th>Instance Type</th>
<th>Linux/UNIX Usage</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Standard On-Demand Instances</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Small (Default)</td>
<td>$0.085 per hour</td>
<td>32-bit = Don’t Use</td>
</tr>
<tr>
<td>Large</td>
<td>$0.34 per hour</td>
<td>Typical MongoDB</td>
</tr>
<tr>
<td>Extra Large</td>
<td>$0.68 per hour</td>
<td></td>
</tr>
<tr>
<td><strong>Micro On-Demand Instances</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Micro</td>
<td>$0.02 per hour</td>
<td>ConfigD / Arbiter</td>
</tr>
<tr>
<td><strong>High-Memory On-Demand Instances</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Extra Large</td>
<td>$0.50 per hour</td>
<td>Big MongoDB</td>
</tr>
<tr>
<td>Double Extra Large</td>
<td>$1.00 per hour</td>
<td></td>
</tr>
<tr>
<td>Quadruple Extra Large</td>
<td>$2.00 per hour</td>
<td></td>
</tr>
<tr>
<td><strong>High-CPU On-Demand Instances</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Medium</td>
<td>$0.17 per hour</td>
<td>32-bit = Don’t Use</td>
</tr>
<tr>
<td>Extra Large</td>
<td>$0.68 per hour</td>
<td>High CPU not necessary</td>
</tr>
<tr>
<td><strong>Cluster Compute Instances</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Quadruple Extra Large</td>
<td>$1.60 per hour</td>
<td></td>
</tr>
<tr>
<td><strong>Cluster GPU Instances</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Quadruple Extra Large</td>
<td>$2.10 per hour</td>
<td></td>
</tr>
</tbody>
</table>

No 32 bit
No High CPU
RAM RAM RAM.
• Replica sets
Failover

• Replica sets
  - One master accepts all writes
  - Many slaves staying up to date with master
  - Can read from slaves

• Master/slave
Failover

• Replica sets
• Master/slave
• Min 3 nodes

Minimum of 3 nodes to form a majority in case one goes down.
All store data.
Odd number otherwise != majority
Arbiter
• Failover

• Replica sets

• Master/slave

• Min 3 nodes

• Automatic failover

Drivers handle automatic failover. First query after a failure will fail which will trigger a reconnect. Need to handle retries.
Redundancy

- Replica sets
- Safe inserts

Can be sure data gets written locally and to replica sets. Delays over networks.
Redundancy

• Replica sets
• Safe inserts
• w flag

Can specify the number of replica slaves the data must be written to before it returns success.
Redundancy

- Replica sets
- Safe inserts
- \texttt{w} flag
- Tags

Define groups of servers so you can determine if the data was written to them.
Tags

{  
  _id : "someSet",
  members : [  
    {_id : 0, host : "A", tags : {"dc": "ny"}},
    {_id : 1, host : "B", tags : {"dc": "ny"}},
    {_id : 2, host : "C", tags : {"dc": "sf"}},
    {_id : 3, host : "D", tags : {"dc": "sf"}},
    {_id : 4, host : "E", tags : {"dc": "cloud"}}
  ]
  settings : {  
    getLastErrorModes : {  
      veryImportant : {"dc" : 3},
      sortOfImportant : {"dc" : 2}
    }
  }
}

> db.foo.insert({x:1})
> db.runCommand({getLastError : 1, w : "veryImportant"})
Redundancy

Tags

{  
  _id : "someSet",
  members : [
    {_id : 0, host : "A", tags : {"dc" : "ny"}},
    {_id : 1, host : "B", tags : {"dc" : "ny"}},
    {_id : 2, host : "C", tags : {"dc" : "sf"}},
    {_id : 3, host : "D", tags : {"dc" : "sf"}},
    {_id : 4, host : "E", tags : {"dc" : "cloud"}}
  ]
  settings : {
    getLastErrorModes : {
      veryImportant : {"dc" : 3},
      sortOfImportant : {"dc" : 2}
    }
  }
}

> db.foo.insert({x:1})
> db.runCommand({getLastError : 1, w : "veryImportant"})

(A or B) + (C or D) + E
Redundancy

Tags

```javascript
{
    _id: "someSet",
    members: [
        {_id: 0, host: "A", tags: {"dc": "ny"}},
        {_id: 1, host: "B", tags: {"dc": "ny"}},
        {_id: 2, host: "C", tags: {"dc": "sf"}},
        {_id: 3, host: "D", tags: {"dc": "sf"}},
        {_id: 4, host: "E", tags: {"dc": "cloud"}}
    ],
    settings: {
        getLastErrorModes: {
            veryImportant: {"dc": 3},
            sortOfImportant: {"dc": 2}
        }
    }
}
```

```shell
> db.foo.insert({x:1})
> db.runCommand({getLastError: 1, w: "sortOfImportant"})
```
Case Study

- Server Density
  - 26 nodes
  - 6 replica sets
- Primary datastore = 15 nodes
Case Study

- Server Density
  - +7TB / mth
  - +1bn docs / mth
  - 2-5k inserts/s @ 3ms

mongoDB
Scaling writes

1) Commit log

First go into a commit log for durability
Scaling writes

1) Commit log
2) Memtable

Then go into a mem table within memory. At this stage the write is considered successful.
Scaling writes

1) Commit log
2) Memtable
3) Disk - sequentially

This is batched and then written to disk periodically. The important thing is they are flushed sequentially which gives high i/o performance.
Connect to any node and issue read/write requests
Node co-ordinates where to get the actual data from i.e proxy
In multi-dc environments, only a single node co-ordinates so only 1 connection is needed
Unlike MongoDB when sharding is optional, in Cassandra partitioning is key. Multiple nodes serve the request and it’s combined into a result set by the co-ordinator.
3 billion rows = 400GB data = 26 hours
More inserts = slower because as there is more data on the heap the JVM has to do more garbage collection.
Commercial company – Acunu – has an optimised storage engine which degrades linearly, unlike the core Cassandra storage engine
Problem for users is latency – some inserts take up to 40s to complete, which could just be a timeout to the user.
• Many SSTables

When flushed to disk = stored as SSTables
Scaling reads

• Many SSTables

• Locate the right one(s)

Can be many of these so reads use bloom filters to find the correct SSTable without having to load it from disk. Very efficient in memory storage.
Scaling reads

• Many SSTables
• Locate the right one(s)
• Fragmentation

This causes fragmentation and lot of files. Although Cassandra does do compaction, it’s not immediate. 1 bloom filter per table.
This works well and scales by simply adding nodes = less data per node
But for range queries it requires every SSTable be queried as bloom filters cannot be used. So performance is directly related to how many SSTables there are = reliant on compaction.
Bottlenecks

• RAM

RAM isn’t as directly correlated to performance as it is with MongoDB because bloom filters are memory efficient and fit into RAM easily. This means there is no disk i/o until it’s needed. But as always the more RAM the better = avoids any disk i/o at all.
Compression in Cassandra 1.0 helps with reads and writes – reduces SSTable size so requires less memory. This works well on column families with many rows having the same columns.
Bottlenecks

- RAM
- Compression
- Wide rows

Using bloom filters, Cassandra is able to know which SSTables the row is located in and so reduce disk i/o. However for wide rows or rows written over time, it may be that the row exists across every SSTable. This can be mitigated by compaction but this requires multiple passes eventually degrading to random i/o which defeats the whole point of compacting – sequential i/o.
**Bottlenecks**

- **Node size**

  No larger than a few 100GB, less with many small values

  Disk ops become very slow due to prev mention issue accessing every bloom filter / SSTable

  Locks when changing schemas – time taken related to data size.
Bottlenecks

• Node size

• Startup time

Startup time proportional to data size which could see a restart taking hours as stuff loaded into mem
Bottlenecks

- Node size
- Startup time
- Heap

All the bloom filters and indexes must fit into its heap, which you can't make larger than ~8GB, as then various GC issues start to kill performance (and introduce random, long pauses, up to 35 seconds!).
• Replication

Replication = core. Required.
Data is evenly distributed around all the nodes.
Failover

- Local reads – don’t need to go across data centres
- Redundancy – allow for full failure
- Data centre and rack aware
Queries define the level of consistency so writes go to a minimum number of nodes and reads also do the same. Where the same data exists on multiple nodes the most recent copy gets priority.

Reads – can be direct = not necessarily consistent / read repair = consistent
Case Study

• Britain’s Got Talent

• RDS m1.large = 300/s

• 10k votes/s

• 2 nodes

Originally on RDS
Peak load 10k/s and atomic
Switched to 2 Cassandra nodes
Scaling

3 things

www.ex-astris-scientia.org/inconsistencies/ent_vs_tng.htm (yes it's a replicator from Star Trek)
• Replication

www.ex-astris-scientia.org/inconsistencies/ent_vs_tng.htm (yes it’s a replicator from Star Trek)
• Replication
• Replication

www.ex-astris-scientia.org/inconsistencies/ent_vs_tng.htm (yes it’s a replicator from Star Trek)
Scaling

- Replication
- Replication
- Replication
- Replication

Each node is individual and on its own
Configure replication on a node level
Master / slave configuration up to you
Can be master / master with 2 way replication

[http://www.ex-astris-scientia.org/inconsistencies/ent_vs_tng.htm](http://www.ex-astris-scientia.org/inconsistencies/ent_vs_tng.htm) (yes it’s a replicator from Star Trek)
Picture is unrelated! Mmm, ice cream.
Access is over HTTP / REST so down to you to implement it. Overhead of HTTP vs wire protocol?
Can therefore use load balancing like a normal HTTP service
Disk space quickly inflates. We found CouchDB using hundreds of GB which fit into just a few GB in MongoDB. Compaction doesn’t help much. Option to not store full document when building queries.
Bottlenecks

- Disk space
- No ad-hoc

Have to know all your queries up-front. Very slow to build new queries because requires full m/r job.
Lots of updates can cause merge errors on replication. Namespace also inflates significantly. Compaction is extremely intensive.
Failover

Master / master so up to you to decide which is the slave
• Replication

Master / master so up to you to decide which is the slave
• Failover

• Replication

• Eventual consistency

Unlike MongoDB / Cassandra, no built-in consistency features
Failover

• Replication

• Eventual consistency

• DNS

Failover on a DNS level
• Replication

Replication works very well but it’s up to you to define roles
There is no failover handling
• Replication
• Failover
• Queries

You can’t query anything without defining everything in advance
Case Study

• BBC

• Eventual consistency

• 8 nodes per DC

• DNS failover

Master / master pairing across DCs
Eventual consistency handled by replication
Use DNS level failover
Case Study

- BBC
- Max 1k PUT/s/node
- 24 PUT/s
- 500 GET/s
- Max 1k PUT/s/node

Hardware benchmarked to 1k PUT/s maximum
**Case Study**

- **BBC**

- **k/v store**

- **Caching**

- **No direct access**

Used as a k/v cache
No direct access by applications – layer on top manages access, failover, security and partitioning evenly across nodes
Have 100% headroom
Used for items like BBC Homepage with customisable layouts – millions of docs. 75+ projects using the k/v store e.g. CBBC games profiles
• Tom Wilkie, Acunu
• Simon Lucy, BBC