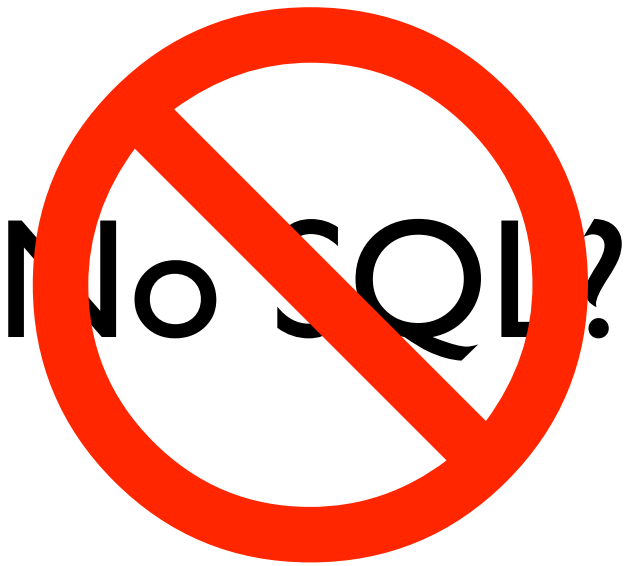
CouchDB

Can we be comfortable without SQL?

ACCU • 14th April 2010

Guy Bolton King • guy@waftex.com

No SQL?


No SQL?

NOSQL

- **N**ot **O**nly **SQL**
- A broad church!

Definition by fiat

NoSQL DEFINITION: Next Generation Databases mostly address some of the points: being **non-relational, distributed, open-source** and **horizontal scalable**. The original intention has been **modern web-scale databases**. The movement began early 2009 and is growing rapidly. Often more characteristics apply as: **schema-free, replication support, easy API, eventually consistency** / **BASE** (not ACID), and more.

http://nosql-database.org/

Definition by example

Definition by example

- Key-Value Stores
  - Amazon SimpleDB, Redis, Scalaris, Tokyo Cabinet, Berkeley DB

Definition by example

- Key-Value Stores
  - Amazon SimpleDB, Redis, Scalaris, Tokyo Cabinet, Berkeley DB
- Distributed Key-Value Stores
  - Amazon Dynamo, Project Voldemort

Definition by example

- Key-Value Stores
  - Amazon SimpleDB, Redis, Scalaris, Tokyo Cabinet, Berkeley DB
- Distributed Key-Value Stores
  - Amazon Dynamo, Project Voldemort
- BigTable
  - Google BigTable, HBase/Hadoop, Cassandra, Hypertable

Definition by example

- Key-Value Stores
  - Amazon SimpleDB, Redis, Scalaris, Tokyo Cabinet, Berkeley DB
- Distributed Key-Value Stores
  - Amazon Dynamo, Project Voldemort
- BigTable
  - Google BigTable, HBase/Hadoop, Cassandra, Hypertable
- Graph
  - Neo4J, VertexDB

Definition by example

- Key-Value Stores
  - Amazon SimpleDB, Redis, Scalaris, Tokyo Cabinet, Berkeley DB
- Distributed Key-Value Stores
  - Amazon Dynamo, Project Voldemort
- BigTable
  - Google BigTable, HBase/Hadoop, Cassandra, Hypertable
- Graph
  - Neo4J, VertexDB
- Document Store
  - CouchDB, MongoDB, Riak

What's wrong with RDBMSs?

What's wrong with RDBMSs?

- Relationships don't survive partitioning:

What's wrong with RDBMSs?

- Relationships don't survive partitioning:
  - Integrity rules and Joins

What's wrong with RDBMSs?

- Relationships don't survive partitioning:
  - Integrity rules and Joins
- Schema rigidity

What's wrong with RDBMSs?

- Relationships don't survive partitioning:
  - Integrity rules and Joins
- Schema rigidity
  - Decomposing data too early

What's wrong with RDBMSs?

- Relationships don't survive partitioning:
  - Integrity rules and Joins
- Schema rigidity
  - Decomposing data too early
  - Hard to change

What's wrong with RDBMSs?

- Relationships don't survive partitioning:
  - Integrity rules and Joins
- Schema rigidity
  - Decomposing data too early
  - Hard to change
- Integrity rules often surplus to requirements and defeat…

What's wrong with RDBMSs?

- Relationships don't survive partitioning:
  - Integrity rules and Joins
- Schema rigidity
  - Decomposing data too early
  - Hard to change
- Integrity rules often surplus to requirements and defeat…
- ORMs

What's wrong with RDBMSs?

- Relationships don't survive partitioning:
  - Integrity rules and Joins
- Schema rigidity
  - Decomposing data too early
  - Hard to change
- Integrity rules often surplus to requirements and defeat…
- ORMs
  - Brrr.

What's wrong with RDBMSs?

- Relationships don't survive partitioning:
  - Integrity rules and Joins
- Schema rigidity
  - Decomposing data too early
  - Hard to change
- Integrity rules often surplus to requirements and defeat…
- ORMs
  - Brrr.
- Mordac

CAP

CAP

- Consistency

CAP

- Consistency
  - All clients see the same data, even with concurrent updates

CAP

- Consistency
  - All clients see the same data, even with concurrent updates
- Availability

CAP

- Consistency
  - All clients see the same data, even with concurrent updates
- Availability
  - All clients can read & write some version of the data

CAP

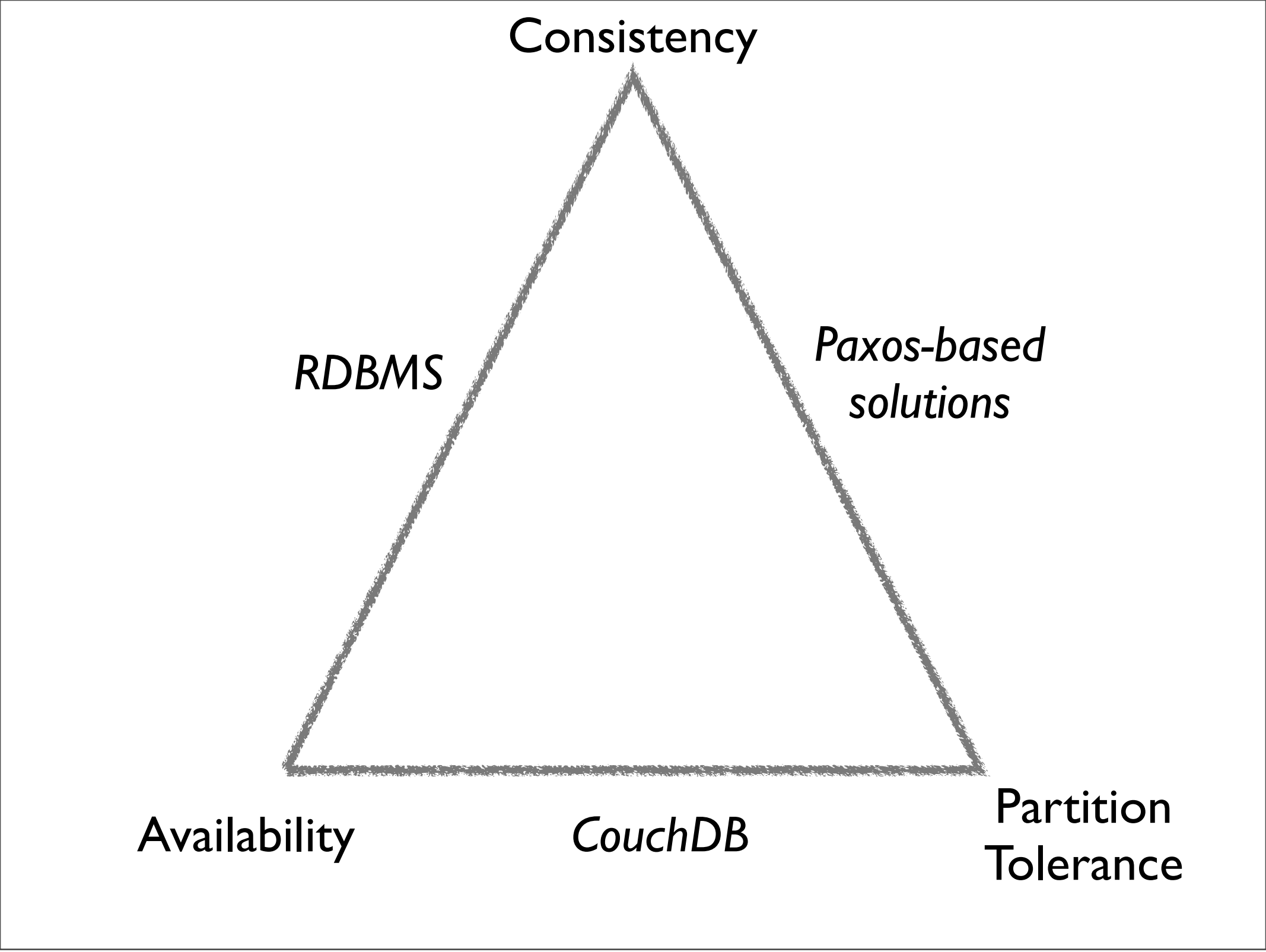
- Consistency
  - All clients see the same data, even with concurrent updates
- Availability
  - All clients can read & write some version of the data
- Partition Tolerance

CAP

- Consistency
  - All clients see the same data, even with concurrent updates
- Availability
  - All clients can read & write some version of the data
- Partition Tolerance
  - Scaling: we can partition the data across multiple nodes

CAP

- Consistency
  - All clients see the same data, even with concurrent updates
- Availability
  - All clients can read & write some version of the data
- Partition Tolerance
  - Scaling: we can partition the data across multiple nodes
- Pick Two


Consistency
RDBMS
Paxos-based solutions
Availability
CouchDB
Partition Tolerance

CouchDB

CouchDB

Cluster **O**f **U**nreliable **C**ommodity **H**ardware

Features

Concurrency

Concurrency

- Written in Erlang

Concurrency

- Written in Erlang
- Scales to multiple cores smoothly

Concurrency

- Written in Erlang
- Scales to multiple cores smoothly
  - One Erlang process per request

Concurrency

- Written in Erlang
- Scales to multiple cores smoothly
  - One Erlang process per request
- Lock-free design: MVCC

Concurrency

- Written in Erlang
- Scales to multiple cores smoothly
  - One Erlang process per request
- Lock-free design: MVCC
  - Each process sees the version of the database that existed at the beginning of the request

Robustness

Robustness

- Erlang's crash-only design

Robustness

- Erlang's crash-only design
- No DB repair necessary

Robustness

- Erlang's crash-only design
- No DB repair necessary
- Snapshots/hot backups trivial: just copy the files

Fault Tolerance

Fault Tolerance

- Continuous replication
- Load-balancing & sharding via couchdb-lounge

Who's using it?

- BBC
- Ubuntu One
- Opscode Chef
- Mozilla Raindrop
- Couchio
- Cloudant
  - myspace, IBM, Apple, ebay
- …and more

Architecture

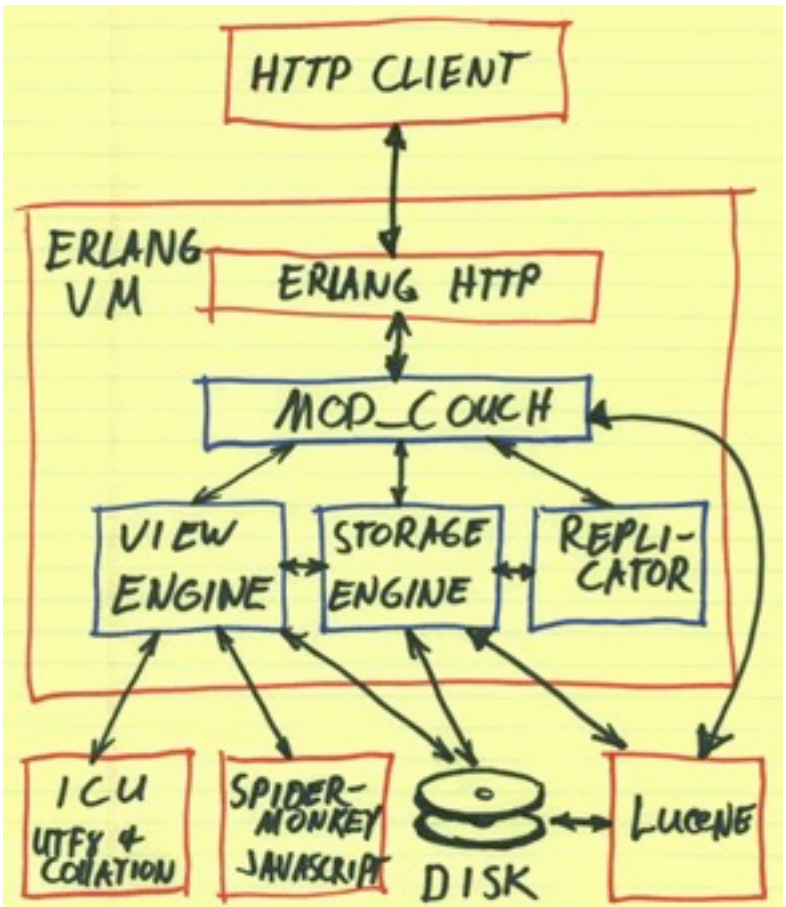
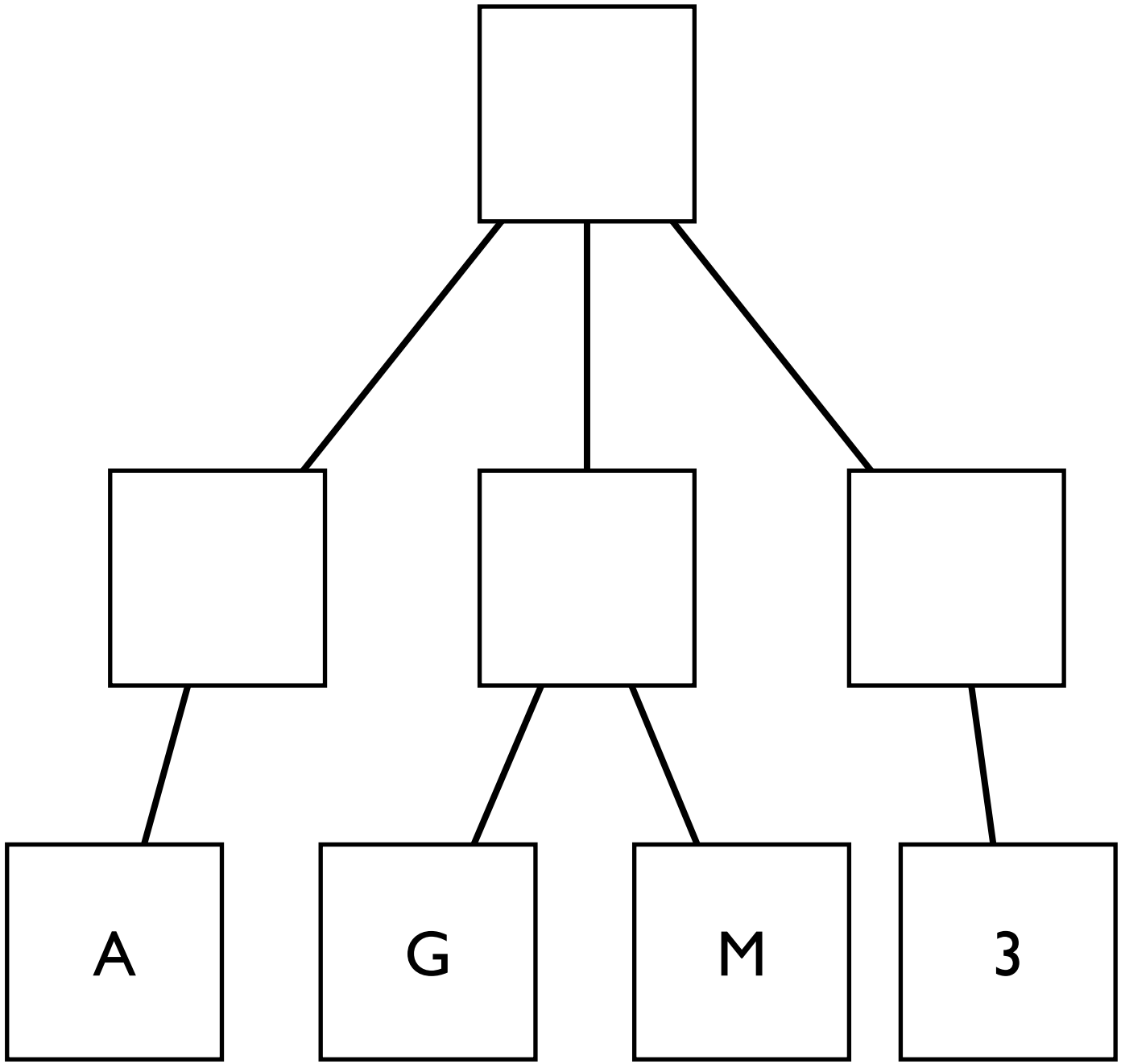


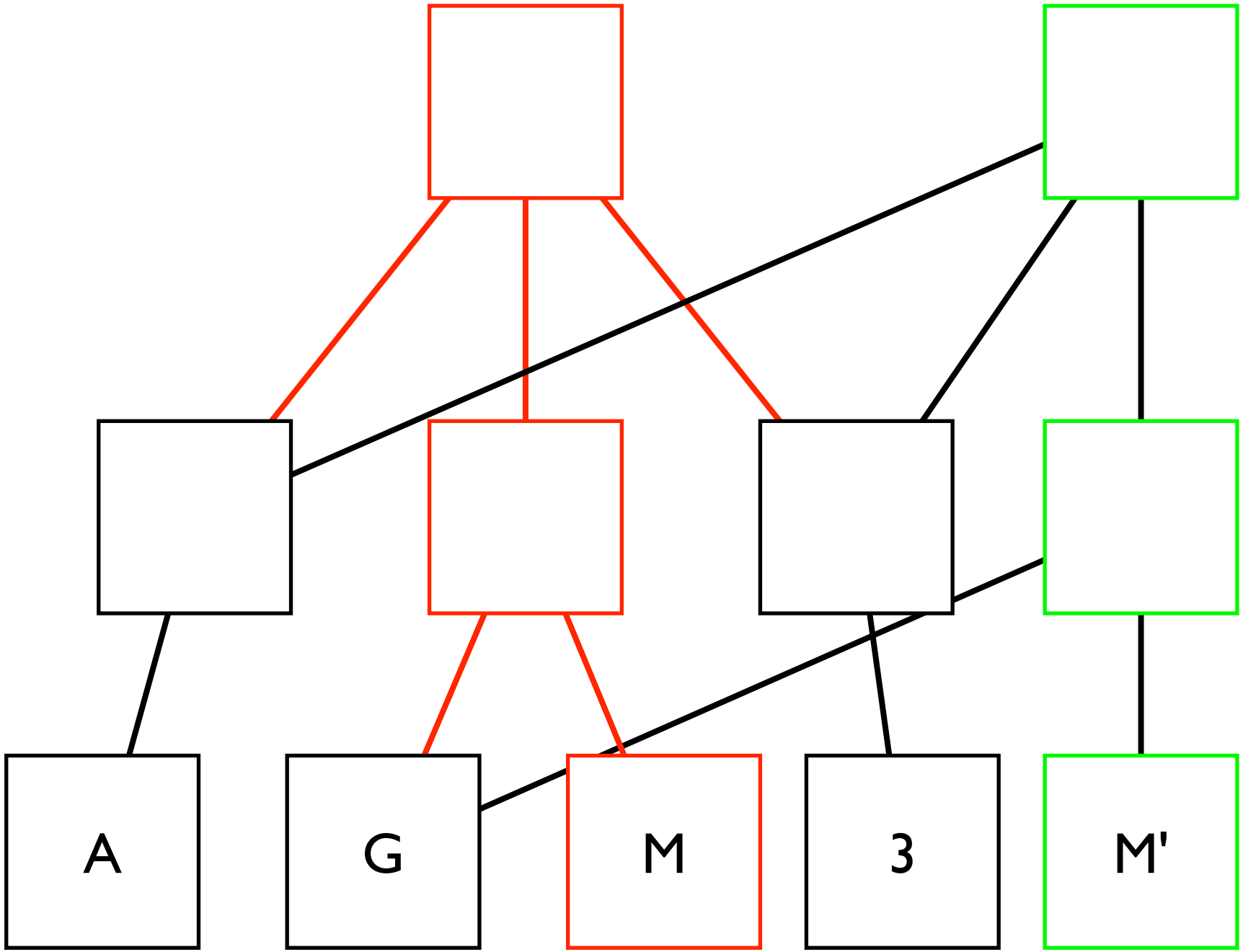
diagram from couchdb.apache.org

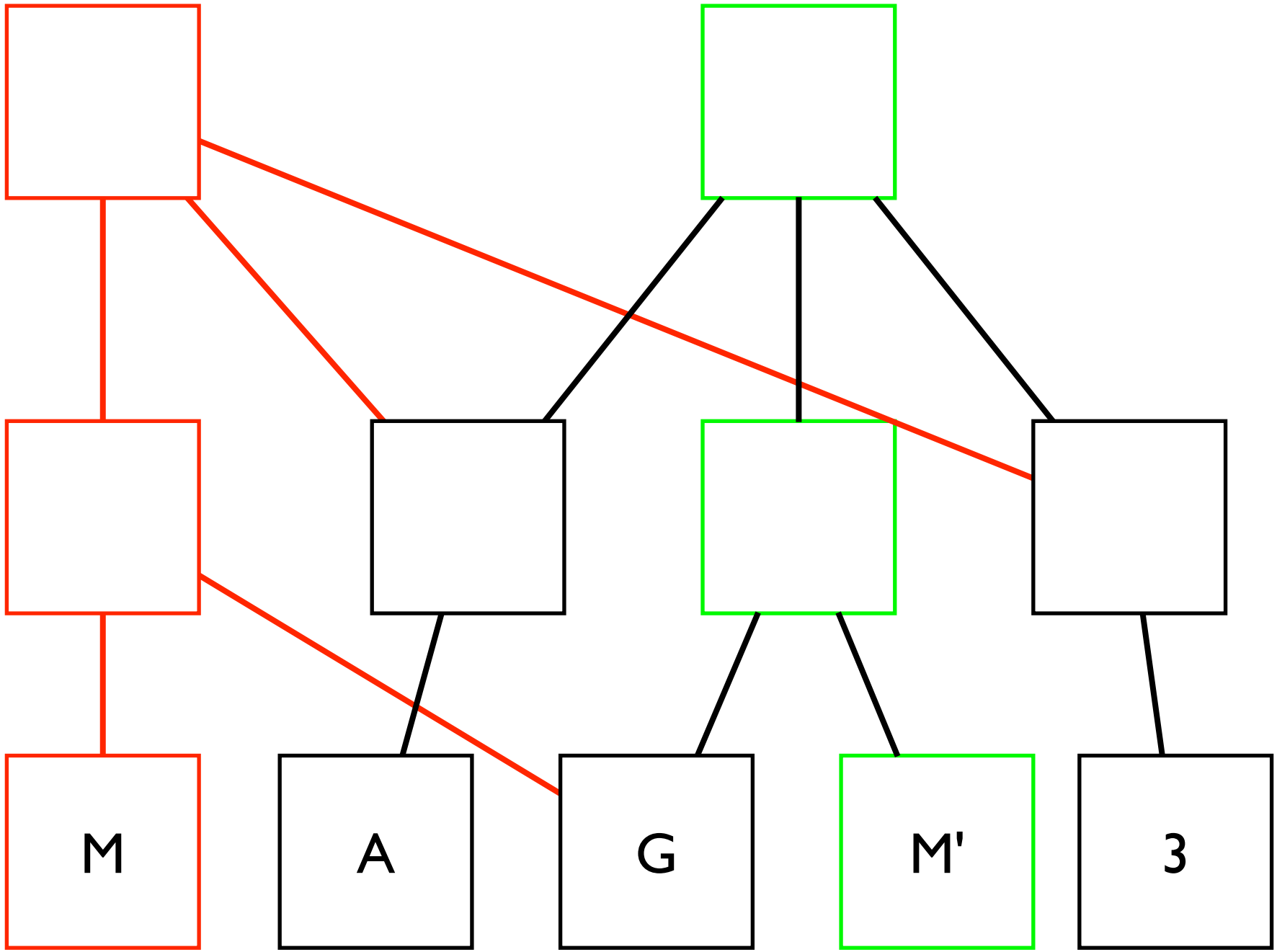
Storage Engine

Storage Engine

- B+Tree
- Append-only design
- Like SVN
- Values are structured JSON documents, with free-form binary attachments if needed
- Each document has a unique _id key


A
G
M
3


A
G
M
3
M'


M
A
G
M'
3

View Servers

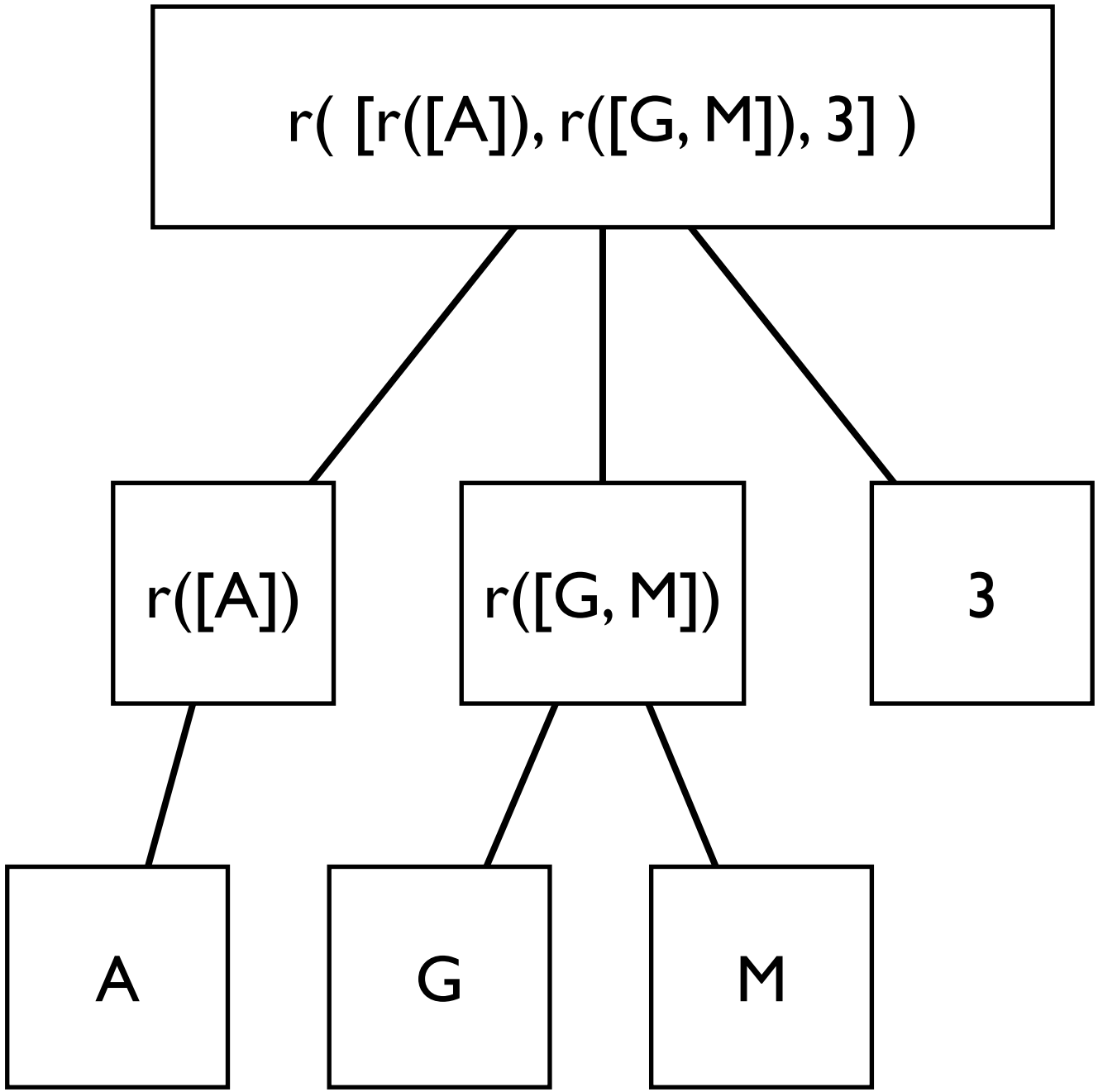
View Servers

- Map/Reduce design
- Map/Reduce functions stored in special "_design" documents
- View server protocol allows any language to be used
  - JavaScript by default
  - Python, Scala, Ruby and others available
- Native Erlang views possible too

View results

View results

- Each view's results stored as a B+Tree
- Map function results as leaf nodes
- Reduce function results in inner nodes
- Results are only updated when view used
- …and only for updated documents


r([r([A]), r([G, M]), 3])
r([A])
r([G, M])
3
A
G
M

JavaScript

- Objects are hashes:

```
var obj = { "k1": "value", "k2": 12 };
```

- Values retrieved via index:

```
var val = obj["value"];
```

- …or dot-notation (iff key is not a keyword):

```
var val = obj.value;
```

- Arrays are objects with integer keys and a length member:

```
var ar = ["foo", 12];
ar[0] === "foo";
ar.length === 2;
```

Functions

- Functions are declared like this:

```
function f(a, b) = { return a + b; }
```

- They're values; this is better:

```
var f = function(a, b) { return a + b; }
```

Iteration

- Iteration using for:

```
for (k in obj) { print(obj[k]); }
```

JSON

- **J**ava**S**cript **O**bject **N**otation
- Doug Crockford
- Serialise data as JavaScript representation
- That's all!

Client API

Client API

- RESTful HTTP transport

Client API

- RESTful HTTP transport
- JSON request and response body

Client API

- RESTful HTTP transport
- JSON request and response body
- CouchDB listens on 127.0.0.1:5984 by default

RESTful HTTP

- PUT path/**new-resource data**
- POST path/resource **data**
- GET path/resource
- DELETE path/resource

Databases

- PUT **db**
- DELETE **db**
- GET _all_dbs

Documents

- Create + Update
  - PUT **db/id data**
- Read
  - GET **db/id**
- Delete
  - DELETE **db/id**

Documents

- Updates for HTML <form> users
  - POST **db/id data**
  - Requires Content-Type: multipart/form-data

Creating Documents

- Create returns ok, id, rev
- Retrieve shows the same id and rev as _id and _rev members

Updating Documents

- Remember MVCC?
- Must specify latest _rev when updating
- If you *don't* have latest _rev, request will fail
  - Need to get latest _rev again
  - Resolve any conflicts in client

Attachments

- Create
  - PUT **db/id/attachment**

    Content-Type: **type**
- Read
  - GET **db/id/attachment**
- Update
  - PUT **db/id/attachment**?rev=**old-rev**
- Delete
  - DELETE **db/id/attachment**
- Attachment meta-data is stored in a document's _attachment key

Tedious

- Specifying _rev gets a bit tedious when using curl
- There must be an easier way to experiment…

Futon

http://localhost:5984/_utils

Document IDs

Document IDs

- Must be unique

Document IDs

- Must be unique
- Ensure uniqueness using your own scheme
  OR

Document IDs

- Must be unique
- Ensure uniqueness using your own scheme
  OR
- Ask CouchDB for some IDs:
  - GET _uuids?count=**n**

Document IDs

- Must be unique
- Ensure uniqueness using your own scheme OR
- Ask CouchDB for some IDs:
  - GET _uuids?count=**n**
- OR use on-the-fly id creation (inefficient):
  - POST **db data**

Retrieving All Documents

- GET **db**/_all_docs?include_docs=true

Limiting

- Use parameters to _all_docs
  - startkey
  - endkey
  - limit
  - skip
- GET **db**/_all_docs?startkey="**s**"&limit=**n**

Querying with Views

- So far, just looked at CouchDB as KV store
- Views are how we do queries
- _all_docs is a built-in view
- We need some data first

Food Example

- Each document represents a person and their fruit & veg preferences

Who likes pears?

Who likes pears?

- people-by-fruit:

Who likes pears?

- people-by-fruit:

```
function(doc) {
```

Who likes pears?

- people-by-fruit:

```
function(doc) {
  if (doc.preferences.fruit) {
```

Who likes pears?

- people-by-fruit:

```
function(doc) {
  if (doc.preferences.fruit) {
    for (i in doc.preferences.fruit) {
```

Who likes pears?

- people-by-fruit:

```
function(doc) {
  if (doc.preferences.fruit) {
    for (i in doc.preferences.fruit) {
      emit(doc.preferences.fruit[i], doc._id);
```

Who likes pears?

- people-by-fruit:

```
function(doc) {
  if (doc.preferences.fruit) {
    for (i in doc.preferences.fruit) {
      emit(doc.preferences.fruit[i], doc._id);
    }
```

Who likes pears?

- people-by-fruit:

```
function(doc) {
  if (doc.preferences.fruit) {
    for (i in doc.preferences.fruit) {
      emit(doc.preferences.fruit[i], doc._id);
    }
  }
```

Who likes pears?

- people-by-fruit:

```
function(doc) {
  if (doc.preferences.fruit) {
    for (i in doc.preferences.fruit) {
      emit(doc.preferences.fruit[i], doc._id);
    }
  }
}
```

Who likes pears?

- people-by-fruit:

```
function(doc) {
  if (doc.preferences.fruit) {
    for (i in doc.preferences.fruit) {
      emit(doc.preferences.fruit[i], doc._id);
    }
  }
}
```

- emit(key, value) adds a row to the result

Who likes pears?

- people-by-fruit:

```
function(doc) {
  if (doc.preferences.fruit) {
    for (i in doc.preferences.fruit) {
      emit(doc.preferences.fruit[i], doc._id);
    }
  }
}
```

- emit(key, value) adds a row to the result
- i.e. it stores value under key in the view's B-tree

Design Documents

- Begin with _design/
- _design/food:

```
{
  "views": {
    "people-by-fruit": {
      "map": "function(doc) { ... }"
    }
  }
}
```

- (curl not much use for this: use Futon)

Running views

- GET **db**/_design/**view**
- Returns list of doc-id, key, value
- Can limit using key, startkey, endkey, limit and skip
- Show full containing doc with include_docs=true

Who likes pears?

```
$ curl -X GET 'http://localhost:5984/accu-food/
_design/food/_view/people-by-fruit?key="pears"'

{"total_rows":10,"offset":6,"rows":[
{"id":"alice","key":"pears","value":"alice"},
{"id":"eve","key":"pears","value":"eve"},
{"id":"harry","key":"pears","value":"harry"},
{"id":"tom","key":"pears","value":"tom"}
]}
```

What's happening?

What's happening?

- The map function is applied to any updated documents, and the view's B-tree updated

What's happening?

- The map function is applied to any updated documents, and the view's B-tree updated
- We then index into the results with key

What's happening?

- The map function is applied to any updated documents, and the view's B-tree updated
- We then index into the results with key
- If no documents are updated before the next use of the view, it will just index into the B-tree of results

Who likes carrots?

- Could do the same for veg as we did for fruit
- Can do better
- Modify people-by-fruit to emit a complex key

Who likes carrots?

```
$ curl -g -X GET 'http://localhost:5984/accu-food/
_design/food/_view/people-by-food?key=
["veg","carrots"]'

{"total_rows":16,"offset":11,"rows":[
{"id":"bob","key":["veg","carrots"],"value":"bob"},
{"id":"dick","key":["veg","carrots"],"value":"dick"}
]}
```

Who likes pears and carrots?

Who likes pears and carrots?

- Can't really be done in CouchDB alone

Who likes pears and carrots?

- Can't really be done in CouchDB alone
- Need to run two queries

Who likes pears and carrots?

- Can't really be done in CouchDB alone
- Need to run two queries
- Make the intersection in client software

Who likes pears and carrots?

- Can't really be done in CouchDB alone
- Need to run two queries
- Make the intersection in client software
- Not every classic RDBM problem can be solved!

Who likes statistics?

Who likes statistics?

- We all do!

Who likes statistics?

- We all do!
- How do we count

Who likes statistics?

- We all do!
- How do we count
  - The number of people who like pears?

Who likes statistics?

- We all do!
- How do we count
  - The number of people who like pears?
  - The number of people who like veg?

Reduce

- Modify people-by-food to return a count instead of doc._id
- Add a reduce function:

```
{
  "views": {
    "people-by-fruit": {
      "map": "function(doc) { ... }"
      "reduce": "function(keys, values, rereduce)
{ ... }"
    }
  }
}
```

How many people like pears?

```
$ curl -g -X GET 'http://localhost:5984/accu-food/
_design/food/_view/food-count?key=["fruit","pears"]'

{"rows":[
{"key":null,"value":4}
]}
```

How many people like veg?

```
$ curl -g -X GET 'http://localhost:5984/accu-food/
_design/food/_view/food-count?startkey=["veg","a"]
&endkey=["veg","z"]'

{"rows":[
{"key":null,"value":7}
]}
```

- Collation is unicode order
- Uses IBM's ICU library
- Can skip ICU collation using raw=true

Grouping

- Querying a reduce view normally shows final reduction i.e. that at the root node of the B-tree
- Can show "lower" results by adding group=true and group_level

Final scores

```
$ curl -g -X GET 'http://localhost:5984/accu-food/_design/food/_view/
food-count?group=true'
```

```
{"rows":[
{"key":["fruit","apples"],"value":2},
{"key":["fruit","bananas"],"value":1},
{"key":["fruit","durian"],"value":1},
{"key":["fruit","kiwis"],"value":1},
{"key":["fruit","kumquats"],"value":1},
{"key":["fruit","pears"],"value":4},
{"key":["veg","broccoli"],"value":1},
{"key":["veg","carrots"],"value":3},
{"key":["veg","celeriac"],"value":1},
{"key":["veg","potatoes"],"value":2}
]}
```

```
$ curl -g -X GET 'http://localhost:5984/accu-food/_design/food/_view/
food-count?group=true&group_level=1'
```

```
{"rows":[
{"key":["fruit"],"value":10},
{"key":["veg"],"value":7}
]}
```

Reduce Parameters

- keys: n keys that correspond to the…
- values: n values of the map results to be reduced.
- rereduce: true if we're reducing non-leaf nodes

Reduce Caveats

- Reduce should produce a scalar value
- Don't try and produce a complex value
  - e.g. a hash of unique keys and their counts
  - Complex values munch storage, often becoming bigger than the document B-Tree itself

Map/Reduce wrap-up

Map/Reduce wrap-up

- Map/Reduce functions are side-effect free

Map/Reduce wrap-up

- Map/Reduce functions are side-effect free
- Only scratched the surface in terms of techniques

Map/Reduce wrap-up

- Map/Reduce functions are side-effect free
- Only scratched the surface in terms of techniques
- Haven't covered handling different document types (although we've hinted at it)

Map/Reduce wrap-up

- Map/Reduce functions are side-effect free
- Only scratched the surface in terms of techniques
- Haven't covered handling different document types (although we've hinted at it)
- 0.11 includes linked documents

Formatting results

- Output seen so far
  - JSON
  - Futon-formatted
- Wouldn't it be nice if we could get CouchDB to format our data?

Show Functions

- Format a document
- Live in the shows key of a design document
- function(doc, req)
  - doc is the doc
  - req is the HTML Request object
  - return HTTP Response object or body string
- GET **db**/_design/**design**/_show/**show**/**id**

```
function(doc, req) {
  log(req);
  send("<!DOCTYPE html>");
  send("<html><head><title>" + doc._id + "</title></head><body>");
  send("<h1>" + doc._id + "</h1>");
  for (type in doc.preferences) {
    send("<h2>" + type + "</h2><ul>");
    for (i in doc.preferences[type]) {
      send("<li>" + doc.preferences[type][i] + "</li>");
    }
    send("</ul>");
  }
  send("</body></html>");
  return "";
}
```

List Functions

- Format a view
- Lives in the lists key of a _design document
- function(head, req)
  - head is {total_rows: **n**, offset: **n**}
  - req is the HTTP Request object
  - return HTTP Response object or body string

```
function(head, req) {
  start({"headers": {"Content-Type": "text/html"}});

  send("<!DOCTYPE html>\n");
  send("<html><head><title>People</title></head><body>");

  var row = null;
  var curtype = null;
  var curitem = null;

  while (row = getRow()) {
    if (row.key[0] !== curtype) {
      curtype = row.key[0];
      curitem = null;
      send("<h1>" + curtype + "</h1>");
    }
    if (row.key[1] !== curitem) {
      curitem = row.key[1];
      send("<h2>" + curitem + " are enjoyed by</h2>");
    }
    send("<p>" + row.value + "</p>");
  }

  send("</body></html>");
  return "";
}
```

Helper functions

- toJSON(obj)
  - Converts obj to a JSON string
- JSON.parse(string)
  - Converts JSON string to an object
- send(chunk)
  - Append chunk to the body of the result
- start(obj)
  - Like calling return obj

Format Helpers

- registerType(name, content-type, …)
  - registers name as a format mapped to a list of content-types
- provides(name, function() { … })
  - registers function as providing the rendering code for the given format. Formats can be specified explicitly as a query parameter ?format=name or by the HTTP Accepts header.

Result object keys

- code
- headers
- body
- stop
  - Returning {"stop": true} terminates list function early

Update functions

- Can change document data!
- Useful for adding timestamps
- Live in updates key of design document
- GET **db**/_design/**design**/_update/**update/id**

Update functions

- function(curdoc, req)
  - curdoc is the current contents of the document identified by req.id
  - may be null if no such document exists
  - can create one!
  - return [newdoc, resp]
  - resp is as the return value in show and list functions

Validation

- validate_doc_update key in design doc
- One per design doc
- All validation functions in db are called on an update

Validation

- function(new-doc, old-doc, user-context)
  - user-context contains {db, name, roles}
  - throw({"forbidden": reason}) on failure

Replication

Replication API

Replication API

- One-shot
  - POST db/_replicate
    {"source": "http://host-a/db",
     "target": "http://host-b/db"}
- Continuous
  - POST db/_replicate
    {"source": "…", "target": "…",
     "continuous": true}

Changes API

- Continuous replication requires host-b to monitor host-a
- Done via Changes API

Changes API

- GET **db**/_changes
  - since=**db-seq**
  - feed=[longpoll|continous]
    - default is immediate return; longpoll waits for 1 change; continuous keeps on truckin'
  - heartbeat=**millis**
  - returns
    {"seq": **db-seq**, "id": **doc-id**,
    "changes": [{"rev": **rev**, …}, …]

Conflicts

- Occur when a document id is changed on host-a and host-b
- Conflicting revs in the invisible _conflicts key
  - Need a conflicts view to find them
- CouchDB always resolves conflicts automatically using it's own algorithm
- We may want to do something else

Resolving Conflicts

- Look at the conflicting and current revs
  - GET **db/id**
  - GET **db/id**?rev=**conflict-rev**
- Merge/replace in your client code
- Put the winner back
  - PUT **db/id**
- Delete the conflicting revs
  - DELETE **db/id**?rev=**conflict-rev**

Security

Authentication

- Both HTTP BasicAuth and Cookie-based authentication is available

Permissions & Roles

- Users have read or read/write privs.
- Also have roles
  - Available in user-context parameter in validate_doc_update, so…
  - You can implement your own write security scheme

What's not there

- No SSL
- Everyone can read
- …so you may want to reverse-proxy CouchDB and apply your own stricter rules

Performance & Tuning

View speed

- Allow use of old view data with stale=ok
  - If all requests use stale=ok view will never update!
  - Refresh views with externally scheduled job

Update speed

- Use the _bulk_docs endpoint with user-defined, monotonic document IDs
- Queue updates with batch=ok
  - Commit to disk controlled by batch_save_size/interval .ini params or…
  - POST **db**/_ensure_full_commit
- Require full sync-on-commit (more reliable, but slower): delayed_commits .ini param

CouchDB benchmark

```
$ test/bench/run
1..6
# Single doc inserts: 162.75 docs/second
ok 1 single_doc_insert
# Single doc inserts with batch=ok: 466.09 docs/second
ok 2 batch_ok_doc_insert
# Bulk docs - 100: 4115.23 docs/second
ok 3 bulk_doc_100
# Bulk docs - 1000: 4202.56 docs/second
ok 4 bulk_doc_1000
# Bulk docs - 5000: 3934.68 docs/second
ok 5 bulk_doc_5000
# Bulk docs - 10000: 3942.67 docs/second
ok 6 bulk_doc_10000
```

View Servers

- Some view servers are faster than others
- Python is faster than JavaScript
  [http://www.mikealrogers.com/archives/673]
- JSON serialisation is a bottleneck
  - Can use Erlang native view server
  - …but only as last resort!
    BIG trade-off in convenience.

Client API

- Native Erlang access via hovercraft
- Bypasses HTTP transport overhead

DB Space

- B-Tree grows and grows
- Need to remove old revs
- Compaction must be triggered manually
  - POST **db**/_compact
  - POST **db**/_compact/**design**

Old View Data

- Changing a view implementation creates a new view B-Tree
- Old B-Trees are left hanging around
- Manual cleanup is required
  - POST **db**/_view_cleanup

Couch Apps

Couch Apps

- You can host a full application stack in CouchDB alone

Couch Apps

- You can host a full application stack in CouchDB alone
  - Futon

Couch Apps

- You can host a full application stack in CouchDB alone
  - Futon
- MVC perspective

Couch Apps

- You can host a full application stack in CouchDB alone
  - Futon
- MVC perspective
  - Controller: JavaScript in the browser

Couch Apps

- You can host a full application stack in CouchDB alone
  - Futon
- MVC perspective
  - Controller: JavaScript in the browser
  - Model: documents + views

Couch Apps

- You can host a full application stack in CouchDB alone
  - Futon
- MVC perspective
  - Controller: JavaScript in the browser
  - Model: documents + views
  - View: show + list + static attachments (CSS, JS)

Problems

Problems

- No way to re-use code in map/reduce/ show/list

Problems

- No way to re-use code in map/reduce/ show/list
- This is unlikely to happen on the server side, since it will hamper internal caching of design documents

Problems

- No way to re-use code in map/reduce/ show/list
- This is unlikely to happen on the server side, since it will hamper internal caching of design documents
- Even unsophisticated list/show functions will duplicate big chunks of code

couchapp

- Build a design document in the filesystem
- couchapp push http: //host/**db**
- ...will create a design document at _id with directory hierarchy mapped to JSON object hierarchy
  - json files become JSON objects
  - other files become strings

Code re-use

- Macros in .js files are expanded prior to upload
- !code path/to/file.js
  - Includes the specified javascript in your function
- !json path.to.file
  - Creates an object hierarchy with only the leafmost node populated

Cloning

- You can extract a pushed couchapp back into the filesystem
- This opens up the idea of easily sharing your couchapps
- Not a substitute for proper DVCS though!

Clustering

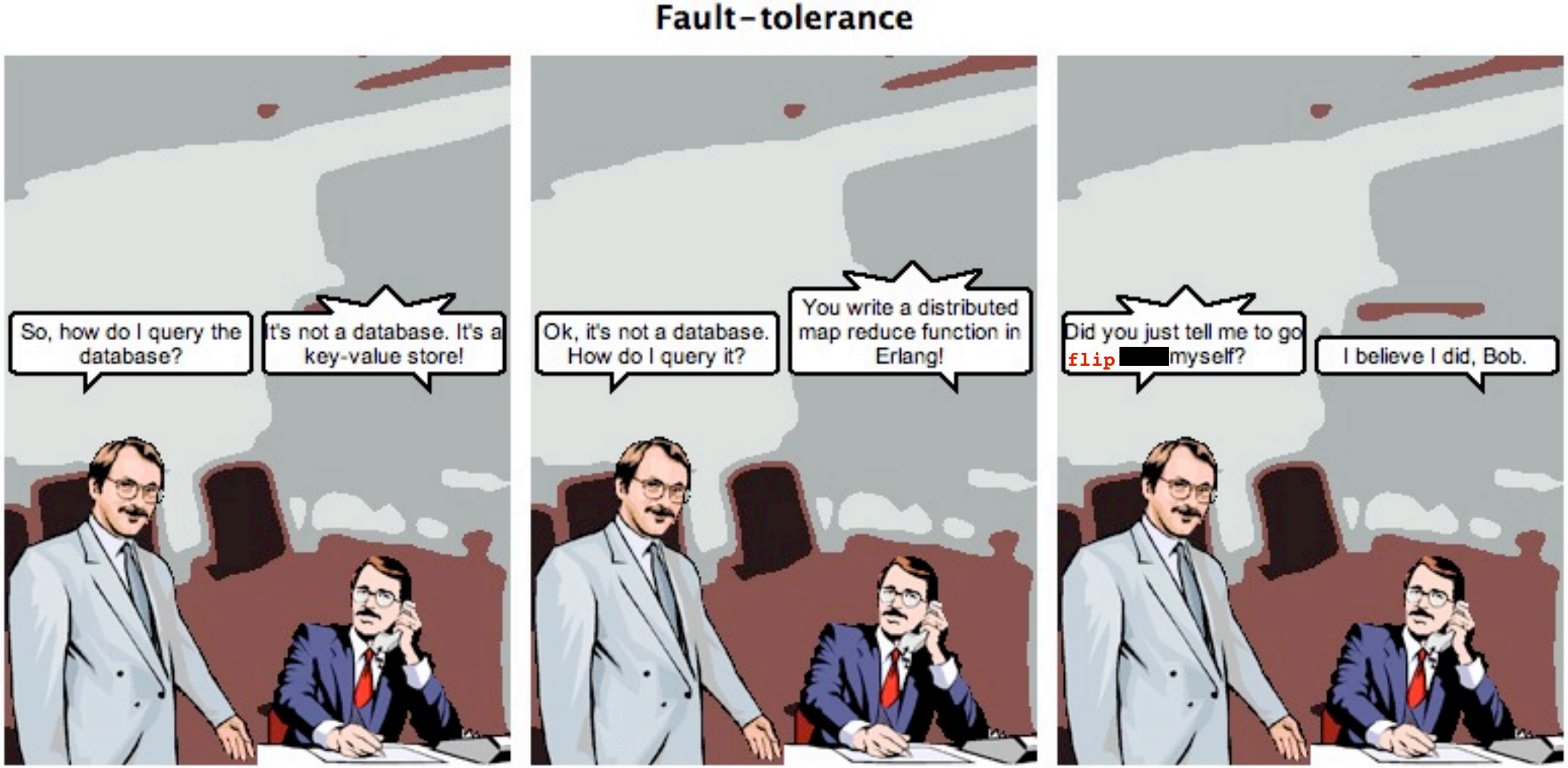
- Roll-your-own, or…
- Use couchdb-lounge
  - Sharded CouchDB instances
  - python+twisted proxy
    - Delegates requests
    - Joins results
  - Still need to roll-your-own replication

Hosting

- Cloudant have a big clustering solution in private beta
- Couchio offer individual CouchDB instances

Why?

- You have a good idea of how the relationships in your model work
- You want to be able to deploy on desktop and server, and synchronise between them
- You may need to scale to handle big data at some point
- You want a flexible B-tree to build your own clustering solution on: Cloudant



Bowdlerised from http://browsertoolkit.com/fault-tolerance.png