Lectures 11 and 12 – Big Data and CouchDB

Luca Morandini
Data Architect – AURIN Project
University of Melbourne
luca.morandini@unimelb.edu.au

Outline of this Lecture

Part 1: "Big data" challenges and architectures

- Consistency and availability in distributed environments
- DBMSs for distributed environments
- MapReduce algorithms

Part 2: Introduction to CouchDB (needed for assignment 2)

- Admin user interface
- Managing documents
- HTTP API
- View, List and Show functions
- Other APIs for CouchDB
- Geo-spatial queries
- Replication and compaction

Part 3: Workshop on CouchDB

Part 1: “Big data” Challenges and Architectures

“Big data” Is Not Just About “Bigness”

The “Vs” :

- **Volume**: yes, volume (Giga, Tera, Peta, …) is a criteria, but not the only one
- **Velocity**: the frequency of new data being brought in to the system and analysis performed
- **Variety**: the variability and complexity of data schema. The more complex the data schema(s) you have, the higher the probability of them changing along the way, adding more complexity.
- **Veracity**: the level of trust in the data accuracy; the more diverse sources you have, the more unstructured they are, the less veracity you have.

Consistency, Availability, Partition-Tolerance

- Consistency

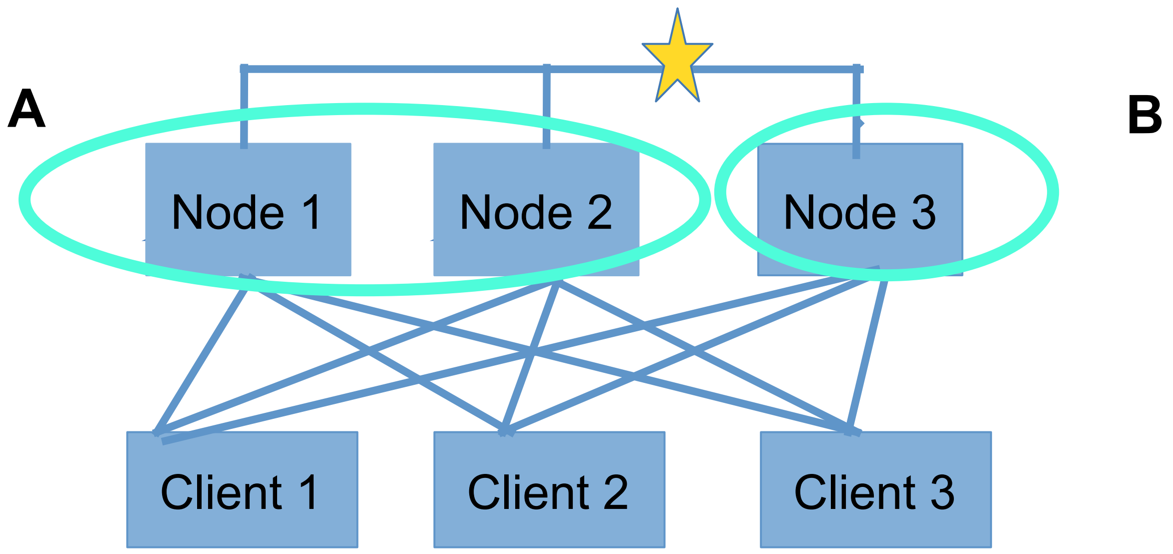
  Every client receiving an answer receives the same answer from all nodes in the cluster

- Availability

  Every client receives an answer from any nodes in the cluster

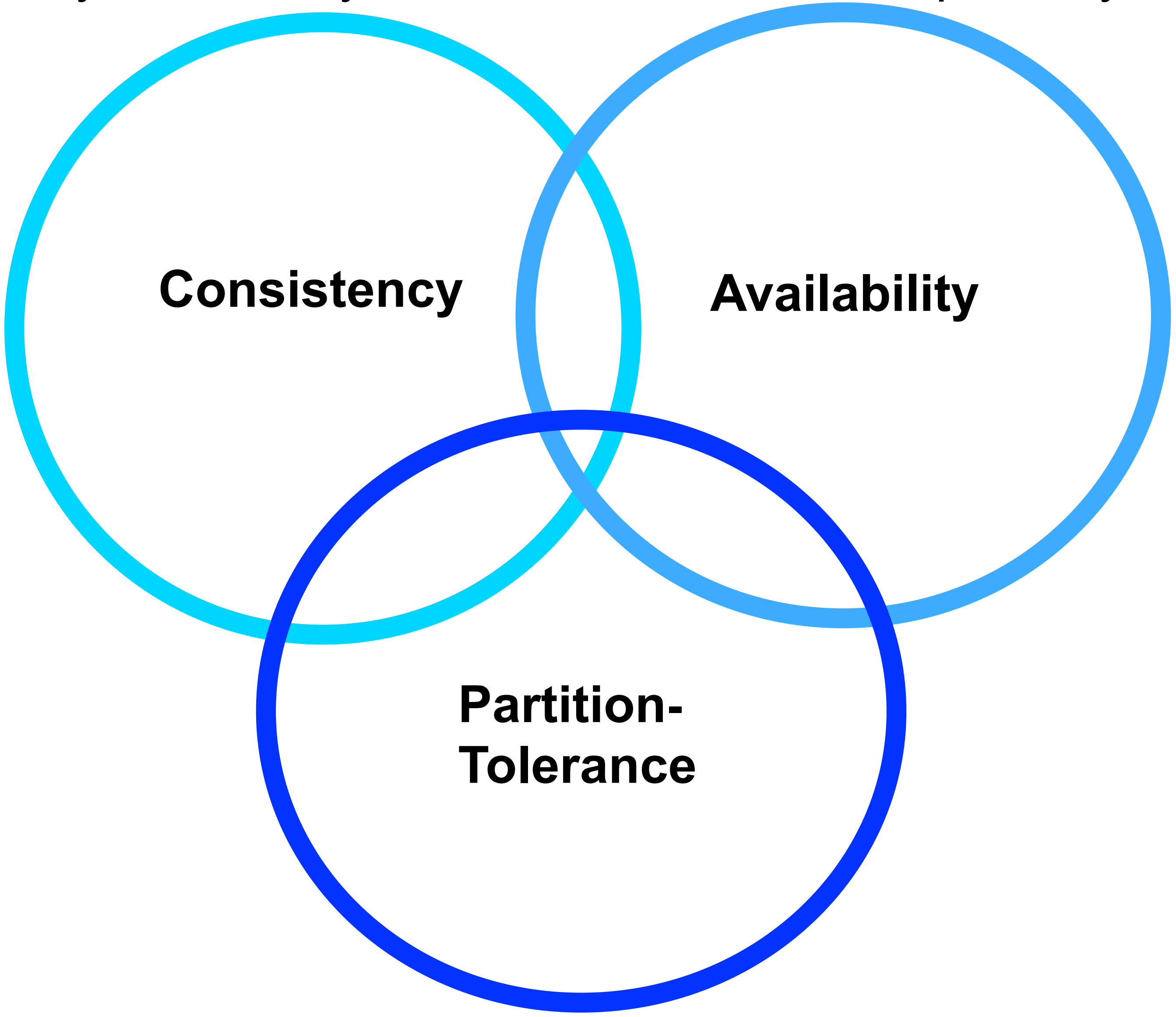
- Partition-tolerance

  The cluster keeps on operating when one or more nodes cannot communicate with the rest of the cluster



Brewer’s CAP Theorem

Consistency, Availability and Partition-Tolerance: pick any two...

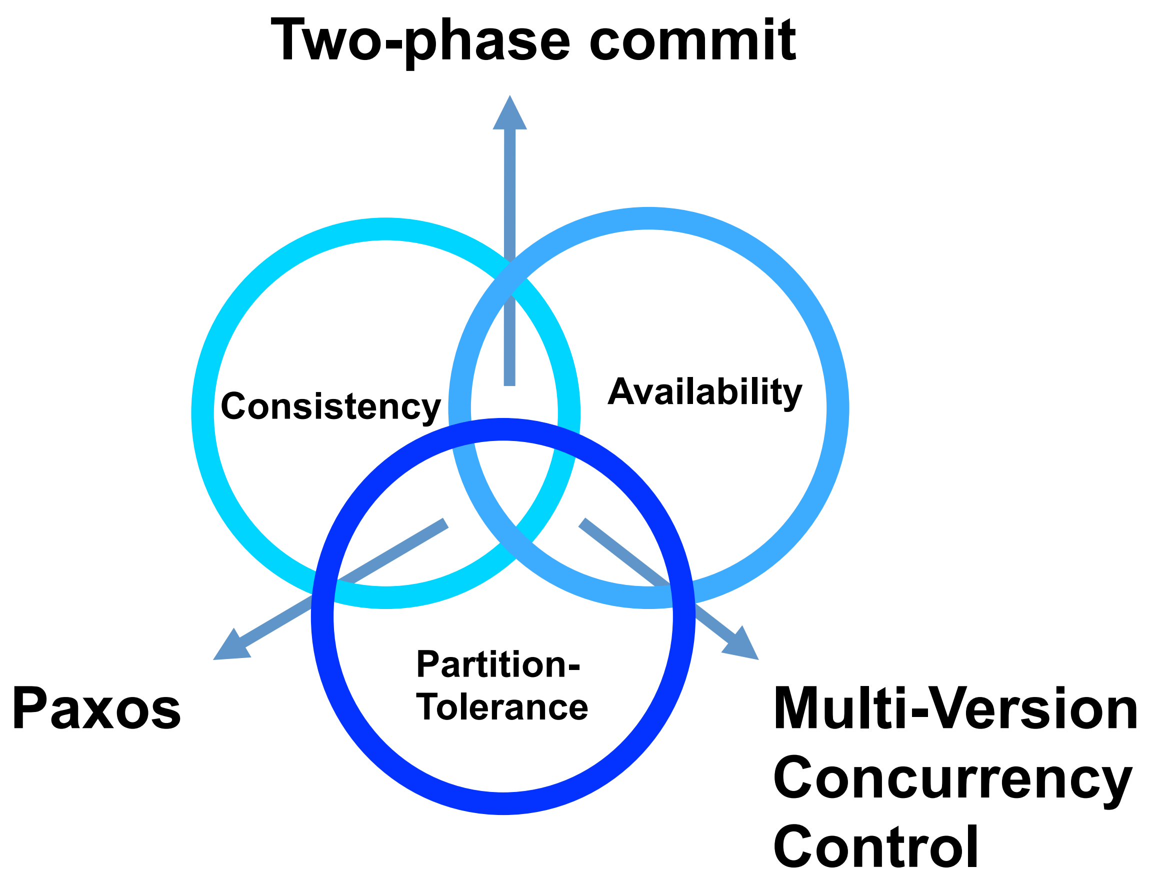


Brewer’s CAP Theorem

...but not quite:

- While the theorem shows all three qualities as symmetrical, Consistency and Availability are at odds when a Partition happens
- “Hard” network partitions may be rare, but “soft” ones are not (a slow node may be considered *dead* even if it is not); ultimately, every partition is detected by a *timeout*
- Can have consequences that impact the cluster as a whole, e.g. a distributed join is only complete when all sub-queries return
- A collection of nodes may include a node that is partition-prone, like a mobile phone (strictly speaking, not a cluster)
- Traditional DBMS architectures were not concerned with network partitions, since all data were supposed to be in a small co-located cluster of servers
- The emphasis on numerous *commodity servers*, can result in an increased number of hardware failures
- The CAP theorem forces us to consider trade-offs among different options

CAP Theorem and the Classification of Distributed Processing Algorithms



Consistency and Availability

Two phase commit

This is the usual algorithm used in relational DBMS's, it enforces consistency by:

- locking data that are within the transaction scope
- performing transactions on write-ahead logs
- completing transactions (commit) only when all nodes in the cluster have performed the transaction
- aborts transactions (rollback) when a partition is detected

This procedure entails the following:

- reduced availability (data lock, stop in case of partition)
- enforced consistency (every database is in a consistent state, and all are left in the same state)

Therefore, two-phase commit is a good solution when the cluster is co-located, less then good when it is distributed

Consistency and Partition-Tolerance

Paxos

- This family of algorithms - is driven by consensus, and is both partition-tolerant and consistent
- In Paxos, every node (replica) is either a proposer or an accepter:
    - a proposer proposes a value (with a timestamp)
    - an accepter can accept or refuse it (if the accepter receives a more current value)
- when a proposer has received a sufficient number of acceptances (a quorum is reached)
- it sends a confirmation message to accepters with the agreed value
- Paxos clusters can recover from partitions and maintain consistency, but the smaller part of a partition (the part that is not in the replicas' quorum) will not send responses, hence the availability is compromised

Availability and Partition-tolerance

Multi-Version Concurrency Control (MVCC)

- Is a method to ensure availability (every node in a cluster always accepts requests), and some sort of recovery from a partition by reconciling the single databases with *revisions* (data are not replaced, they are just given a new revision number)

- In MVCC, concurrent updates are possible without distributed locks (in optimistic locking only the local copy of the object is locked), since the updates will have different revision numbers; the transaction that completes last will get a higher revision number, hence will be considered the “current” value

- Coarse-grained DBMS models, like document-oriented DBMSs, fit the optimistic locking of the MVCC method, since the number of transactions is less than in a comparable relational design

Multi-Version Concurrency Control Example

An example of how two clients do not lock data and still avoid inconsistency by using a revision number for sequence transactions (similar to revision control systems, e.g. Git or Subversion).

	Client-1	Client-2
time-1	POST obj1,{name:”a”} ==> OK,rev:1	
time-2		PUT obj1,rev:1,{name:”b”} ==> OK,rev:2
time-3	PUT obj1,rev:1,{name:”c”} ==> ERROR	
time-4	GET obj1 ==> OK,rev:2, {name:“b”}	
time-5	PUT obj1,rev:2,{name:”bc”} ==> OK,rev:3	

MVCC relies on monotonous increasing revision numbers and preservation of old object versions to ensure availability (i.e. when an object is updated, the old versions can still be read).

ACID-to-BASE: Continuum

Atomicity
Consistency
Isolation
Durability

Basically
Available
Soft-state
Eventual
consistency

ACID has been the common set of properties that every proper DBMS was expected to show in traditional transaction processing. BASE is a relaxed set of properties that is more in tune with the less tidy realities of big data.

There are systems that are in-between BASE and ACID, like Paxos, which is Basically-Available and always Consistent

Why DBMSs for Distributed Environments?

While Relational DBMS are extremely good for ensuring consistency and availability, and there is nothing preventing them implementing partition-tolerant algorithms, the normalization that lies at the heart of a relational database model produces fine-grained data, which are less partition-tolerant than coarse-grained data.

Example:

- A typical contact database in a relational data model may include: a person table, a telephone table, an email table, an address table.
- The same database in a document-oriented database would entail one document type only, with telephones numbers, email addresses, etc., nested as arrays in the same document.

The document-oriented option needs less synchronization and is easily “shardable” (*shards* are partitions of a database, which could be based on an attribute, e.g.: sharding tweets by city means every city has one database holding all the Tweets from it).

DBMSs for Distributed Environments

- Key-values stores: Redis, PostgreSQL Hstore
- BigTable DBMSs: Google BigTable, Apache Accumulo
- Document-oriented DBMSs: Apache CouchDB, MongoDB

Examples of Distributed DBMSs

- PostgreSQL-XL: a clustered relational DBMS (based on the open source PostgreSQL DBMS) that implements MVCC to keep consistency.

- Cloudera's Impala: in-memory heterogeneous-format SQL database based on Hadoop, used for analytics

- Apache Drill: heterogeneous-format SQL, no metadata required, used for analytics

Apache Accumulo

Apache Accumulo is an example of a BigTable DBMS:

- Original developed by the National Security Agency (NSA)
- The key of every piece of data is composed of the following parts: *Row ID, Column Family, Column Qualifier, Timestamp*
- Every operation on the same row and column is called a *mutation*, and older mutations are hidden by default
  - (only the one with the most recent timestamp is shown)
- Visibility is fine-grained:
  - it can be set for every column of every row, or even by timestamp
- Rows can be scanned (filtered) by key and then iterated over
- Rows can be combined with iterators, a sort of reduce

Apache Accumulo, sample session

Apache Accumulo is an example of a BigTable DBMS

```
createtable contacts
contacts> insert "joe" "emails" "work" "joe@aaa.com"
contacts> insert "joe" "emails" "private" "joe@gmail.com"
contacts> insert "joe" "emails" "work" "joe@bbb.com"
contacts> insert "bob" "emails" "private" "bob@gmail.com”

contacts> scan -st
bob emails:private [] 1426472398864    bob@gmail.com
joe emails:private [] 1426469439544    joe@gmail.com
joe emails:work [] 1426469449671    joe@bbb.com

contacts> scan -row bob
bob emails:private []    bob@gmail.com
```

MapReduce Algorithms

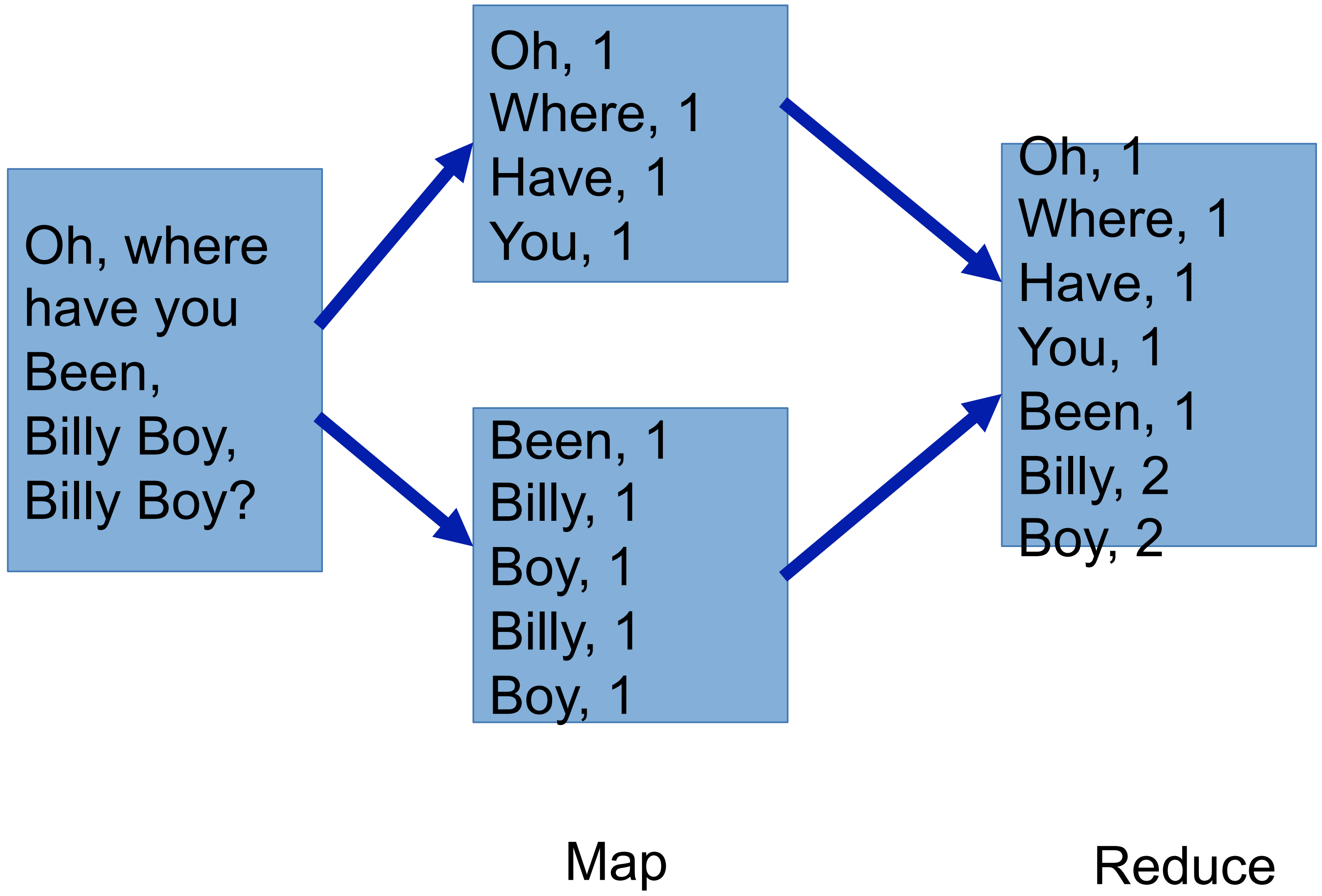
- This family of algorithms, pioneered by Google, is particularly suited to parallel computing of the Single-Instruction, Multiple-Data type (see Flynn's taxonomy – lecture 3).

- The first step (Map), distributes data across machines, while the second (Reduce) hierarchical summarizes them until the result is obtained. Apart from parallelism, its advantage lies in moving the process where data are, greatly reducing network traffic.

- Example (unashamedly taken from Wikipedia):

```
function map(name, document):
    for each word w in document:
      emit (w, 1)
function reduce(word, partialCounts):
    sum = 0
    for each pc in partialCounts:
      sum += pc
    emit (word, sum)
```

MapReduce Wordcount in a Picture



MapReduce and Distributed DBMS

Since it is horizontally scalable, MapReduce is the tool of choice when operations on big datasets are to be done
Let's see how the staple of database querying, the inner join, works on MapReduce (fileL and fileR have to be joined on userId field):

```
function map(fileL, fileR):
  forEach line in fileL:
    emit ({key: line.userId, {type: “L”, line: line})
  forEach line in fileR:
    emit ({key: line.userId, {type: “R”, line: line})

function reduce(key, values):
  forEach value in values:
    if value.type == “L”
      lineL= value.line
    else
      lineR= value.line

  return {userid: key, left: lineL, right: lineR}
```

Part 2: Introduction to CouchDB

CouchDB Main Features

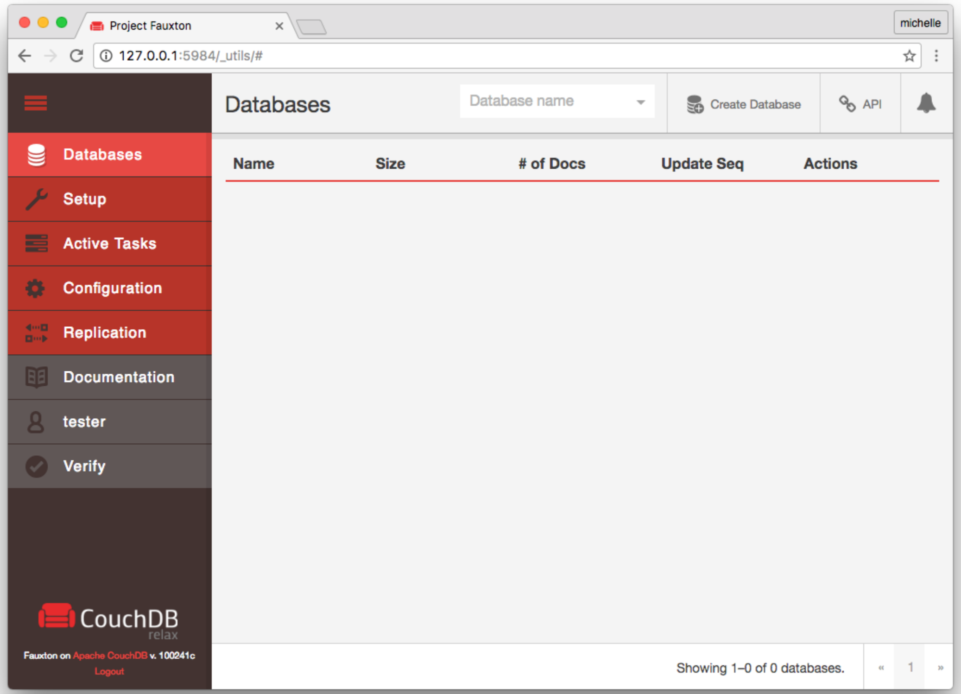
The main features of CouchDB are:

- Document-oriented DBMS (Documents expressed in JavaScript Object Notation)
- HTTP ReST API (more on ReST in later lectures!)
- Web-based admin interface
- Web-ready: since it talks HTTP and produces JSON (it can also produce HTML or XML), it can be both the data and logic tier of a three-tier application, hence avoiding the marshalling and unmarshalling of data objects
- Support for MapReduce algorithms, including aggregation at different levels
- JavaScript as the default data manipulation language
- Run Mango queries (MongoDB query language)
- Schema-less data model with JSON as the data definition language
- Support for replication
- Support (albeit limited!) for geo-spatial data (for 2nd assignment!)

Fauxton User Interface

Typing http://localhost:5984/_utils into a browser opens the admin user interface, which lets you do most operations easily, including:

- Create/delete databases
- Edit documents
- Edit design documents
- Run views (MapReduce)
- Run Mango queries
- Modify the configuration
- Manage users
- Set up replications



Databases

- A CouchDB instance can have many databases; each database can have its own set of functions (grouped into *design documents*) and stored in different datafiles

- Adding and deleting a database is through a HTTP call:

```
curl -X PUT “http://localhost:5984/exampledb“
curl -X DELETE “http://localhost:5984/exampledb“
```

- Listing all databases of an instance is even simpler

```
curl -X GET “http://localhost:5984/_all_dbs”
```

- Where, every response's body is a JSON object:

```
["_replicator","_users","exampledb"]
```

- In every CouchDB instance there are two system databases:
  - _replicator database holds the documents defining replication flows
  - _users database is for CouchDB users

Insertion and retrieval of documents

- To insert a document:

```
curl -X POST "http://localhost:5984/exampledb" --header "Content-Type:application/json" --data '{"type": "account", "holder": "Alice", "initialbalance": 1000}'
```

Response: 201 (202 if less than the prescribed write operations were successfully performed)

```
{"ok":true,"id":"c43bcff2cdbb577d8ab2933cdc0011f8","rev":"1-b8a039a8143b474b3601b389081a9eec"}
```

- To retrieve a document:

```
curl -X GET “http://localhost:5984/exampledb/c43bcff2cdbb577d8ab2933cdc0011f8”
```

Response: 200

```
{"_id":"c43bcff2cd577d8ab2933cdc0011f8","_rev":"1-b8a039a8143b474b3601b389081a9eec","type":"account","holder":"Alice","initialbalance":1000}
```

System Properties of Documents

- _id: is the ID of a single document
  - while the ID can be set during the document load, by default it is generated by CouchDB and it is guaranteed to be unique

- _rev: revision number of a document,
  - guaranteed to be increasing per-document
  - every database instance will pick up the same revision as the “live” version of the document

- Request to set the ID of a document (note PUT instead of POST):

```
curl -X PUT "http://localhost:5984/exampledb/charlie" --header "Content-
  Type:application/json" --data '{"type": "account", "holder": "Charlie",
  "initialbalance": 100}'
```

Response: 200

```
{"ok":true,"id":"charlie","rev":"1-faff5448bf3051ac4fb8f1cc2b04bc51"}
```

Note that the Id of the doc is “charlie”, and not a UUID

CouchDB can generate and serve UUIDs

- If you need an UUID, just ask CouchDB to generate one:

```
curl -X GET “http://localhost:5984/_uuids”
```

Response: 200

```
{"uuids":["c43bcff2cdbb577d8ab2933cdc18e83b"]}
```

- Or, if you need more than one:

```
curl -X GET “http://localhost:5984/_uuids?count=3”
```

Response: 200

```
{"uuids":
  ["c43bcff2cdbb577d8ab2933cdc18d06c","c43bcff2cdbb577d8ab2933cd
  c18d815","c43bcff2cdbb577d8ab2933cdc18dc0f"]}
```

Error 409 when updating documents

- What happens when we update a document?

```
curl -X PUT "http://localhost:5984/exampledb/charlie" --header "Content-Type:application/json" --data '{"type": "account", "holder": "Charlie", "initialbalance": 200}'
```

Response: 409

```
{"error":"conflict","reason":"Document update conflict."}
```

(HTTP status code 409)

Why?

The way to avoid conflicts in MVCC is to state which revision the update refers to

- Try with the revision number as returned by POST:

```
curl -X PUT "http://localhost:5984/exampledb/charlie?rev=1-
  faff5448bf3051ac4fb8f1cc2b04bc51" --header "Content-Type:application/
  json" --data '{"type": "account", "holder": "Charlie", "initialbalance": 200}'
```

  Response: 200

```
{"ok":true,"id":"charlie","rev":”2c0716f36b7cb2b2d31102fe807697573"}
```

- Better now (note the increased revision number)

What happens when a conflict happens on a cluster of CouchDB nodes?

When the cluster gets partitioned, and two nodes receives two different updates of the same document, two different revisions are added. However, only one of this is returned as the current revision. The “winning” revision is computed deterministically, and it is guaranteed to be the same on any node of the cluster.

To help the merging of conflicting revisions, CouchDB can return all the conflicts in a database

```
GET /exampledb/_all_docs?include_docs=true&conflicts=true
```

Deletion of documents... but not quite

- How to delete document (note the revision number):

```
curl -X DELETE "http://localhost:5984/exampledb/charlie?rev=2-
  c0716f36b7cb2b2d31102fe807697573"
```

  Response:

```
{"ok":true,"id":"charlie","rev":"3-320d11c2d78a18ccc0220086c418cc41"}
```

- To check the deletion:

```
curl -X GET "http://localhost:5984/exampledb/charlie"
```

  Response (note, the reason is not “missing”, it is “deleted”):

```
{"error":"not_found","reason":"deleted"}
```

- Actually, documents are not deleted until they are “purged”, hence they could be retrieved, with a bit of effort (add document with the same id, then retrieve the old revision).

Deletion of documents... for good

- How to delete a document permanently:

```
curl -X POST "http://localhost:5984/exampledb/_purge" --header "Content-
   Type:application/json" --data '{"charlie":
   ["3-320d11c2d78a18ccc0220086c418cc41"]}'
```

  Response:

```
{"purge_seq":2,"purged":{"charlie":
   ["3-320d11c2d78a18ccc0220086c418cc41"]}}
```

- Every document and revision has to be specified for CouchDB to delete them, e.g.:

```
url -X GET "http://localhost:5984/exampledb/charlie"
```

  Response (now document is not just “deleted”, it is “missing”):

```
 {"error":"not_found","reason":"missing"}
```

Deletion of Old Revisions

- The accumulation of old revisions can bloat a database, but automatic deletion and compaction is available:

```
curl -X POST
  “http://localhost:5984/my_db/_compact“ --header "Content-Type:
  application/json"
```

- The compaction can be tailored by setting a limit on the number of revisions stored before deletion, time of day for compaction, automatic triggering conditions (based on the percentage of old documents in views or documents)

Documents can be bulk-managed

- Documents can be bulk loaded, deleted, updated via the CouchDB bulk docs API:

```
curl -v -X POST "http://localhost:5984/exampledb/_bulk_docs" --header
  "Content-Type:application/json" --data '{"docs":[{"name":"joe"},
  {"name":"bob"}]}'
```

Response:

```
[{"ok":true,"id":"c43bcff2cdbb577d8ab2933cdc18f402","rev":"1-
c8cae35f4287628c696193172096c988"},
{"ok":true,"id":"c43bcff2cdbb577d8ab2933cdc18f796","rev":"1-
c9f1576d06e12af51ece1bac75b26bad"}]
```

The same POST request can be used to update documents (revision numbers and ids need to be provided in the JSON with `_id` and `_rev` attributes respectively)

Documents can have attachments

- A document can have one (or more) attachments of whatever MIME-type is needed, including binary ones (in RDBMS, they would be called BLOBs):

```
curl -X PUT "http://localhost:5984/exampledb/text/original?
  rev=1-26074febbe9a4a0e818f7d5587d7411a" --header "Content-
  Type:image/png" --data @./scannedtext.png
```

- Attachments are listed in the _attachments attribute of a document, together with content-type, hash code, and length

- Attachments can be very handy for sharing binary data or big JSON documents that do not require parsing

Views, but not relational ones!

- CouchDB *views* are not:
  - Queries (they are not volatile)
  - Views (the selected data are persisted)
  - Indexes (data are persisted together with the index)

- What are they?
  - CouchDB views are similar to *Index-organized tables* in Oracle, which are defined in the Oracle documentation as:

    An index-organized table has a storage organization that is a variant of a primary B-tree. Unlike an ordinary (heap-organized) table whose data is stored as an unordered collection (heap), data for an index-organized table is stored in a B-tree index structure in a primary key sorted manner. Each leaf block in the index structure stores both the key and non-key columns.

- Long story short:
  - views are fast, but inflexible and waste a lot of storage

CouchDB Views #1

- Views are definitions of MapReduce jobs that are updated as new data come in and are persisted.
  - ➢ e.g.(a view for counting occurrences of words in a text):
  - ➢ Example of a document:

```
{"_id": "dan",
 "name": "Chris White",
 "address": "4, Macaulay Rd, Melbourne",
 "type": "account",
 "balance": 100}
```

- Example of a view:

…the Map part:

```
function(doc) {
    emit([doc.name], doc.balance);
}
```

...the Reduce part:

```
function(keys, values, rereduce) {
   return sum(values);
}
```

CouchDB Views #2

Results of the view defined in the previous page:

```
["Alice Smith"],500
["Bob Dole"],1000
["Charlie Black"],1500
["Chris White"], 100
```

Note: results are ordered by an ascending key

- keys parameter: an array of keys, as returned by the view (null when rereduce is true)
- values parameter: an array of values, as returned by the view
- rereduce parameter: if false, the reduce is still in its first stage (keys are values are the disaggregated ones); if true, the reduce has already happened at least once, and the function works on aggregated keys and values (hence the keys parameter is null)

CouchDB Views #3

- Keys can be composite:

```
function(doc) {
 if (doc.type === "text" ) {
   var words= doc.contents.split(/[\s\.,]+/);
   for (var i in words)  {
     if (words[i].length > 1) {
       emit([words[i].substring(0,1),words[i]],
       1);
    …
["a", "ad"], 1
["a", "adipisicing"], 1
["a", "aliqua"], 1
…
```

- Results can grouped by key, hence aggregate data at different levels:

```
["a"], 7
["c"], 6
["d"], 6
...
```

CouchDB Views #4

- Views can be called from HTTP

http://localhost:5984/exampledb/_design/example/_view/wc2
?group_level=2&startkey=[“a”,null]&endkey=[“c”,{}]

- Views
  - are grouped into *design documents* (example).
  - can be passed the level of aggregation (group_level).
  - can return only a subset of keys (start_key, end_key parameters).
  - are computed once they are called - not when they are defined - and updated every time a document is inserted, deleted or updated. Hence, the first invocation is slow.
  - are persisted to disk, hence adding an entire document in the view's result would use a lot of disk space: emit(words[i],doc) (don't, unless necessary... use include_docs=true instead).
  - Since there is no schema and documents of different types are often stored in the same database, it is useful to add a *type* attribute to docs, which comes in handy when defining views.

CouchDB Views #5

- Views
  - can be defined in languages other than JavaScript
  - can share libraries
  - cannot be passed custom parameters, either during computation or during selection

- Computation of views can be influenced only by the document itself (referentially transparency):
  - no reading of other documents
  - no passing of parameters during computation

- In short…
  - Views are not influenced by the state of the system
  - this ensures consistency of results

CouchDB Views #6

- Keys are case-sensitive, as they are in RDBMS indexes
- Pagination is available through the use of skip and limit parameters
- Ad-hoc queries, useful during development, can be done using “Temporary Views”, which are not saved to the database
- Sort order can be reversed by using the descending parameter
- The Reduce part of a View can be more complex than adding values (sum, count, min and max can be computed in one go by using the stat function)
- Reduce function must be referentially transparent, associative and commutative. In other words, the order of computations must not influence the result:

```
f(Key, Values) === f(Key, [ f(Key, Values) ])
```

List and Show Functions

- However powerful, views are limited, since they can produce only JSON and cannot change their behavior (except for sub-setting the output with skip, limit, group_level, etc)
- To address these shortcomings, CouchDB offers List and Show functions
    - Both these two classes of functions can modify their behavior when HTTP request parameters are sent, and both can produce non-JSON output
    - List functions transform a view into a list of something (can be a list of HTML <li> tags, or a list of <doc> XML tags.
    - Show functions transform an entire document into something else (like an entire HTML page).
- To sum it up:
    - Show functions are applied to the output of a single document query
    - List functions are applied to the output of Views
    - List and Show functions can be seen as the equivalent of JEE servlets

How to do joins #1

- Views can be used to simulate joins; that is, to order rows by a common key.
- Suppose we have bank accounts and operations:

```
{"_id":"bob","name":"Bob Dole","address":"1,Collins
  St,Melbourne","type":"account","balance":1000}
…
{"_id":"94b6eb8511034068a5bdc6bcf1002ce7","account":"bob","type":"de
  posit","amount":200}
{"_id":"94b6eb8511034068a5bdc6bcf100244b","account":"bob","type":"wit
  hdrawal","amount":100}
```

account can act as Foreign Key from operations (withdrawal and deposit document types) to the account document type

How to do joins #2

A view like this.

```
function(doc) {
  if (doc.type==="account") {
    emit([doc._id, doc.type], doc.balance);
  }
  if (doc.type==="deposit") {
    emit([doc.account, doc.type], doc.amount);
  }
  if (doc.type==="withdrawal") {
    emit([doc.account, doc.type], -doc.amount);
  }
}
```

Returns a key formed by an account ID and a document type, hence ordering the documents, while the value is the balance or the operations amount (negative for withdrawals)

How to do joins #3

The view returns this at group level 2:

```
{"rows":[
{"key":["alice","account"],"value":500},
{"key":["alice","deposit"],"value":200},
{"key":["alice","withdrawal"],"value":-100},
{"key":["bob","account"],"value":1000},
{"key":["bob","deposit"],"value":100},
{"key":["charlie","account"],"value":1500},
{"key":["charlie","deposit"],"value":200},
{"key":["charlie","withdrawal"],"value":-400}
]}
```

...and this at group level 1 (note the changed balance):

```
{"rows":[
{"key":["alice"],"value":600},
{"key":["bob"],"value":1100},
{"key":["charlie"],"value":1300}
]}
```

Hold on a sec, isn't that a transaction ?

The previous view returns the balance as the arithmetic sum of all operations on a given account:

```
{"rows":[
{"key":["alice"],"value":600},
{"key":["bob"],"value":1100},
{"key":["charlie"],"value":1300}
]}
```

Is this result different from the one given by a relational transaction that locks the account's row, adds/subtracts the value of the financial operation, and then unlocks the account's row?

How to do OLAP (sort of) #1

- Given a fake db of tweets (time and location properties added):

```
{
   "_id": "224077c21b5c2d61235bb7d1fc002e9e",
   "_rev": "1-ef0192025d040f3632ccaa73294013b4",
   "time": "15:19:05",
   "location": "kensington, melbourne",
   "text": "I am anxious"
},
{
   "_id": "224077c21b5c2d61235bb7d1fc001911",
   "_rev": "3-eb49d8e7113348a57ccf9edc7be459b5",
   "time": "10:10:02",
   "location": "carlton, melbourne",
   "text": "I feel angry and frustrated"
},
{
   "_id": "224077c21b5c2d61235bb7d1fc003f31",
   "_rev": "1-bcfcbb7724c2eb34ff0a38d01b6a45d4",
   "time": "10:19:05",
   "location": "auburn, sydney",
   "text": "I am happy"
}
```

How can we aggregate by: City, Suburb, Time of day (morning, afternoon) and a mood extracted from the text ?

How to do OLAP (sort of) #2

- CouchDB views can aggregate across multiple dimensions:

```
function(doc) {
  var moods= [
     {name: "negative", count: 0, synonyms: ["angry", "anxious", "unhappy", "frustrated"]},
     {name: "positive", count: 0, synonyms: ["happy", "elated"]}
    ];
  var i;
  var words= doc.text.split(" ");
  for (i in words)  {
    if (moods[0].synonyms.indexOf(words[i]) > -1) moods[0].count++;
    if (moods[1].synonyms.indexOf(words[i]) > -1) moods[1].count++;
  }
  var suburb= doc.location.split(",")[0];
  var city= doc.location.split(",")[1];
  var timeofday;
  if (Number(doc.time.split(":")[0]) < 12) {
    timeofday= "am";
  } else {
    timeofday= "pm";
  }
  emit([city, suburb, timeofday], moods[1].count - moods[0].count);
}
```

How to do OLAP (sort of) #3

- CouchDB views can aggregate across multiple dimensions: group_level 3 (aggregation by City, Suburb, Time of day)

```
["melbourne", "carlton", "am"], -3
["melbourne", "kensington", "am"], 1
["melbourne", "kensington", "pm"]  , -1
["sydney", "auburn", "am"], 1
["sydney", "auburn", "pm"], -1
```

group_level 2 (aggregation by City and Suburb)

```
["melbourne", "carlton"], -3
["melbourne", "kensington"], 0
["sydney", "auburn"], 0
```

group_level 1 (aggregation by City)

```
["melbourne"], -3
["sydney"], 0
```

How to do OLAP (sort of) #4

- The order of dimensions and levels is fixed; hence, if we want to aggregate by Time of day, City, Suburb, we have to add another view.

- Line to change in previous view:

```
emit([timeofday, city, suburb], moods[1].count - moods[0].count);
```

- group_level 2 (aggregation by Time of day, City)

```
["am", "melbourne"], -2
["am", "sydney"], 1
["pm", "melbourne"], -1
["pm", "sydney"], -1
```

- group_level 2 (aggregation by Time of day)

```
["am"], -1
["pm"], -2
```

CouchDB Application Development

- Java: Ektorp
- Node.js: Nano, Cradle
- Client Javascript: jQuery plugin,
- Plenty of others: Perl, Python, Ruby, Clojure, Common LISP, .Net...
- Worthy of mention:
  - ➢ PouchDB is a client JavaScript software that mimics CouchDB and synchronizes data with a CouchDB instance, ideal for stand-alone applications development

Replication

- There is a system _replicator database that holds all the replications to be performed on the CouchDB instance; adding a replication is just a POST away:

```
curl -H 'Content-Type: application/json' -X POST http://localhost:5984/
  _replicate -d ' {"source": "http://myserver:5984/foo", "target": "bar",
  "create_target": true, "continuous": true}'
```

- Note the continuous attribute is set to true; if this were false, the database instance would be replicated only once

- To cancel a replication, just issue the same exact JSON, but with an additional cancel attribute set to true

- Replications are uni-directional, for a properly balanced system, you need to add two replications

Note: this example is taken from the Apache CouchDB Wiki

Clustering CouchDB Instances

- So far we have considered CouchDB instances in isolation, but how could they work as part of a cluster?

- Some highlights:
  - During database creation, it is possible to define the number of shards (q) and the replicas (n):

```
curl -X PUT http://localhost:5984/test?n=3&q=4
```

  - Write operations (w) complete successfully only if the document is committed to a quorum of replicas (usually a simple majority)
  - Read operations (r) complete successfully only if a quorum of replicas return matching documents
  - Views are distributed
  - The default of these parameters (n, q, r, w) are set in the cluster section of the *.ini configuration file

Sharding

- Sharding is the partitioning of a database “horizontally”; that is, the rows are partitioned in subsets that are stored on different servers.

- Usually the number of shards (q) is larger than the number if replicas (n), and the number of nodes is larger than the replicas (which are usually set to 3)

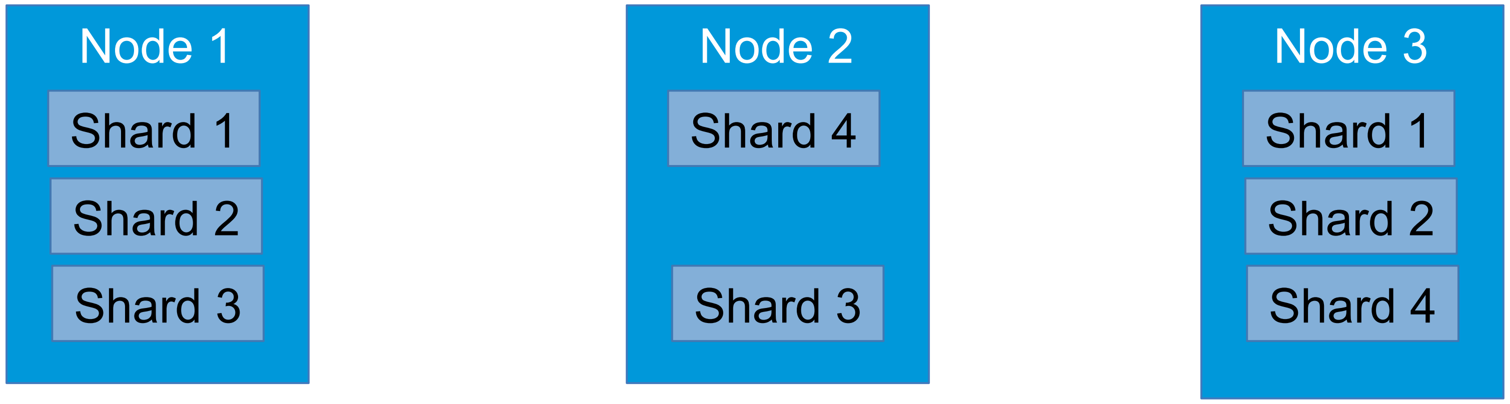
- The main advantage of a sharded database lies in the improvement of performance through the distribution of computing load across nodes. In addition, it makes easier to move data files around, for instance, when adding new nodes to the cluster

- There are different sharding strategies, most notably:
    - Hash sharding: to distribute rows evenly across the cluster
    - Range sharding: similar rows (say, tweets coming for the same area) are stored on the same node (or sub-set of nodes)

Replication and Sharding

Replication is the action of storing the same row on different nodes to make the database fault-tolerant. Replication and sharding can be combined with the objective of maximizing availability while maintaining a minimum level of data safety.

For instance, in a cluster of 3 nodes, a replication factor of 2 and a number of shards equal to 4 would look like:



Geo-spatial indexing: GeoCouch

- The geo-spatial capabilities of CouchDB are, for the time being, rather poor (and not very performant)
  - ➢ GeoCouch is just another way to index documents via views (R-trees instead of B-trees) with geometries defined in JSON. Actually, GeoJSON http://geojson.org/geojson-spec.html
  - ➢ GeoCouch views are stored under “_spatial”, not under “_view”.
  - ➢ GDAL supports loading of data in CouchDB via GeoCouch

- Example of spatial-enabled view:

```
function(doc)  {
        emit({type: "Point",coordinates: [doc.loc[0],doc.loc[1]]},
    [doc._id,doc.loc]);
}
```

- Selection of documents by location (bounding-box):

```
curl -X GET 'http://localhost:5984/places/_design/main/_spatial/points?
  bbox=70,40,80,50'
```

Part 3: Workshop

Hands-on CouchDB

- Now for a practical tour of CouchDB using Futon (it’s web-based administration user interface)

- Topics:
  - ➢ Creating/deleting, making a backup/restoring a database
  - ➢ Adding, updating and deleting documents
  - ➢ Adding documents in bulk
  - ➢ Writing and executing views
  - ➢ Writing and testing show and list functions
  - ➢ Setting up replication and continuous replication

Transforming and Adding documents in Bulk

```
head -1 twitau4.json | sed "s/ObjectId( //g" | sed "s/ )//g" | sed 's/{ "_id"/
{"oldid"/g'> twitau4.head.json

head -1000 twitau4.json | tail -999 | sed "s/ObjectId( //g" | sed "s/ )//g" |
sed 's/{ "_id"/ , {"oldid"/g' > twitau4.sample.json

echo '{"docs":[' > twitau4.sample2.json
cat twitau4.head.json >> twitau4.sample2.json
cat twitau4.sample.json >> twitau4.sample2.json
echo ']}' >> twitau4.sample2.json
curl -X DELETE "http://localhost:5984/twit"
curl -X PUT "http://localhost:5984/twit"
curl -X POST "" --data @twitau4.sample2.json --header "Content-
Type:application/json"
```

Writing and Executing Views (examples with Twitter data)

- Editing views using Futon
- Use of CouchApp/Erica [1]
- Read the logs, and use the “log” function
- Develop with a small, but meaningful, subset of your data
- Simple: calculate statistics on languages (twitter/language)
- More complex: use of a JS library (twitter/languagegroup)

- [1] To install CouchApp, go here and follow the instructions: https://github.com/couchapp/couchapp

- Alternatively you may want to look at Erica: https://github.com/benoitc/erica

Writing and Executing Show & List Functions

- How to add a show function

- Calling a show function:

```
curl -X GET
"http://localhost:5984/twit/_design/twitter/_show/html/docid"
```

- How to add a list function

- Calling the a list function:

```
curl -X GET
"http://localhost:5984/twit/_design/twitter/_list/html/languagegroup?
  limit=10&group_level=2"
```

Replication

Easy...

- Local or remote
- Continuous or not (done only once)
- Every replication's instance is mono-directional
- Futon's replications does not survive a Server's restart, add them instead to the _replicator's database

Some Anti-Patterns...

- A collection of real-world examples, to show that data modeling has its place in a document-oriented DBMS too.
- Let's spot the anti-patterns...

Anti-Pattern #1 (at least three issues)

```
{
    _id: "b738b38106f2d53f1bc1e48299a5ef91",
    _rev: "2-c9e0ba0d0d7ce40647f008f7252b021d",
    id: "7f23b9e5-62cd-41d7-9595-94a79e05fdb2",
    data: {
        type: "meta",
        value: {
          availability: false,
          docId: "7f23b9e5-62cd-41d7-9595...”,
          projectId: "project-47e4-9713-8eab6ff...",
          projectName: "Untitled Project"
        }
    },
    userId: "John+Smith",
    projectId: "project-meta"
}
```

Anti-Pattern #2 (two issue)

```
_id: "68b0f64b8f6c7801fcd4cb01a2ca0291",
_rev: "4-f70af527df033fa7cd5cbc0df3d8563d",
timestamp: "2013-12-13 13:42:53.415",
jobId: "000ce3cb-d2f5-42b5-8057-1a794dc15f51",
jobStatus: "DONE",
job: {
    datasetParams: {
        name: "phidu:SD_Labourforce",
        filter: {
            type: "tabular",
            expression: “population > 1000”
        }
}
```

Anti-Pattern #3 (one issue)

```
function(doc) {
 if(doc.id && doc.timestamp) {
  if(doc.projectId && doc.projectId=='user-meta')
     emit(doc.timestamp, doc);
 }
}
```

Anti-Patterns Solutions

- Anti-pattern #1: id should not be there, since CouchDB already provides an id (_Id); projectId is repeated; the type of the document should be top-level
- Anti-pattern #2: “status” and “id” should be in “job”, since they are both attributes of it; the type of the document should be defined
- Anti-pattern #3: the entire document is replicated in the view, doubling the database size and slowing down view creation and update

Bibliography

[1] *3D Data Management: Controlling Data Volume, Velocity, and Variety*, Doug Laney, Gartner Group, 2001

[2] *Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services*, Seth Gilbert and Nancy Lynch, 2001

[3] *The Growing Impact of the CAP Theorem*, February 2012 issue of *IEEE Computer* magazine, IEEE

[4] *Paxos made simple*, ACM SIGACT News, 121, December 2001, 51-58

[5] *Mapreduce: Simplified data processing on large clusters*, Jeffrey Dean and Sanjay Ghemawat. In OSDI 2004, pages 137-150, 2004

[6] *Spanner: Google's Globally-Distributed Database*, James C. Corbett, Google Inc.

[7] *CouchDB: The Definitive Guide: Time to Relax*, J. Chris Anderson, Jan Lehnardt and Noah Slater, O'Reilly, 2010

[8] *An Introduction to Apache Accumulo*, Dr. Dennis Patrone, 2014