Lecture 13/14 – 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

- DBMSs for distributed environments
- What distributed DBMSs look like: MongoDB vs CouchDB
- Consistency and availability in distributed environments
- MapReduce algorithms
- Sharding

Part 2: Introduction to CouchDB

- Managing documents
- HTTP API
- Queries (Views, Mango and Full-text Search)

Part 3: Workshop on CouchDB

- Setting up a 3-node cluster with Docker
- Storing and retrieving data using CouchDB

Part 1: “Big data” Challenges and Architectures

“Big data” Is Not Just About “Bigness”

The four “**V**s” :

- **Volume:** yes, volume (Giga, Tera, Peta, …) is a criteria, but not the only one
- **Velocity**: the frequency at which new data is being brought into the system and analytics 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 (provenance); the more diverse sources you have, the more unstructured they are, the less veracity you have.

Big Data Calls for Ad hoc Solutions

- While Relational DBMSs are extremely good at ensuring consistency, they rely on normalized data models that, in a world of big data (think about Veracity and Variety) can no longer be taken for granted.

- Therefore, it makes sense to use DBMSs that are built upon data models that are not relational (relational model: tables and relationships amongst tables).

- While there is nothing preventing SQL to be used in distributed environments, alternative query languages have been used for distributed databases, hence they are sometimes called *NoSQL DBMSs*

DBMSs for Distributed Environments

- A *key-value store* is a DBMS that allows the retrieval of a chunk of data given a key: fast, but crude (e.g. Redis, PostgreSQL Hstore, Berkeley DB)

- A *BigTable DBMS* stores data in columns grouped into *column families*, with rows potentially containing different columns of the same family (e.g. Apache Cassandra, Apache Accumulo)

- A *Document-oriented DBMS* stores data as structured documents, usually expressed as XML or JSON (e.g. Apache CouchDB, MongoDB)

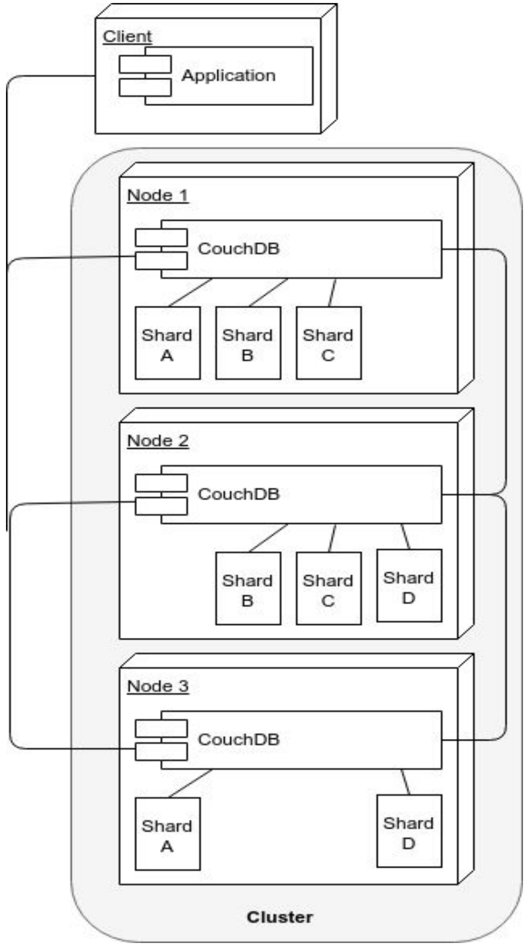
A Tale of Two Clusters

- Distributed databases are run over “clusters”, that is, sets of connected computers

- Clusters are needed to:
    - Distribute the computing load over multiple computers, e.g. to improve *availability*
    - Storing multiple copies of data, e.g. to achieve *redundancy*

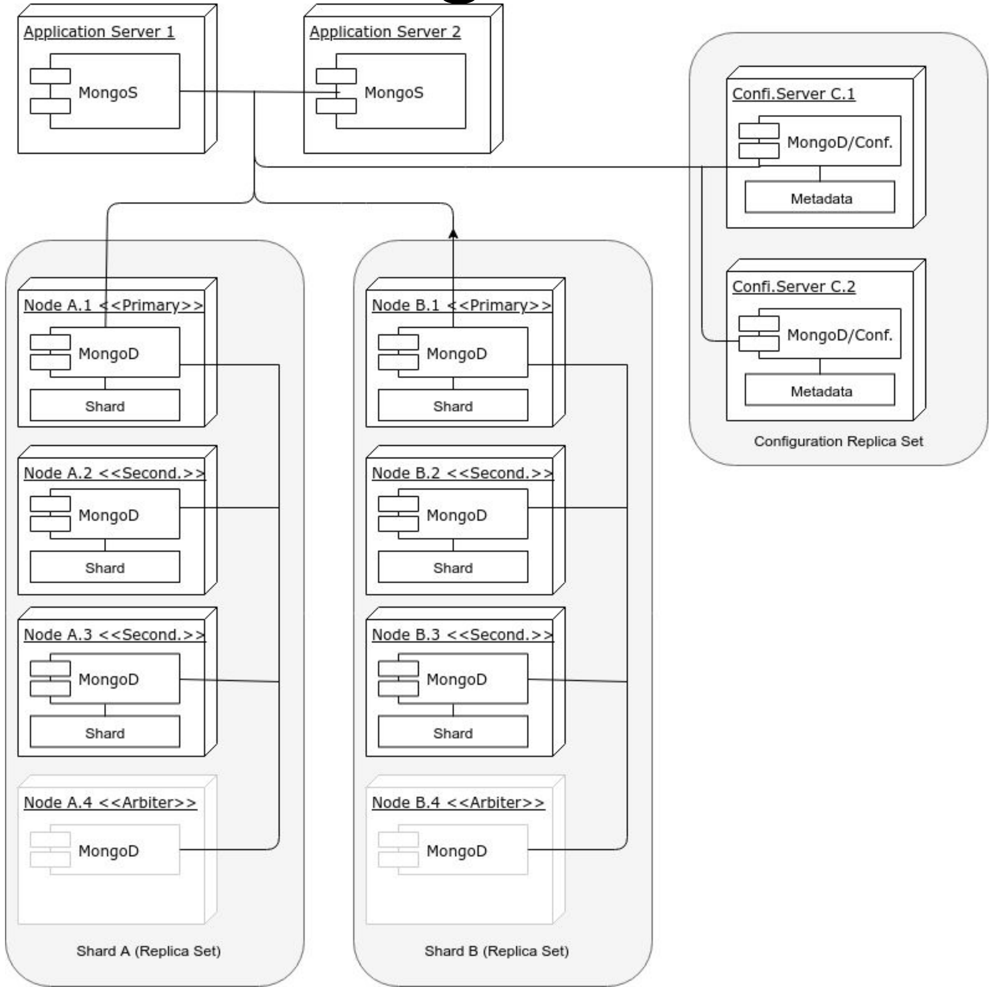
- Consider two document-oriented DBMSs (*CouchDB* and *MongoDB*) and their typical cluster architectures

CouchDB Cluster Architecture



- All nodes answer requests (read or write) at the same time
- Sharding (splitting of data across nodes) is done on every node
- When a node does not contain a document (say, a document of Shard A is requested to Node 2), the node requests it from another node (say, Node 1) and returns it to the client
- Nodes can be added/removed easily, and their shards are re-balanced automatically upon addition/deletion of nodes
- In this example there are 3 nodes, 4 shards and a replica number of 2

MongoDB Cluster Architecture



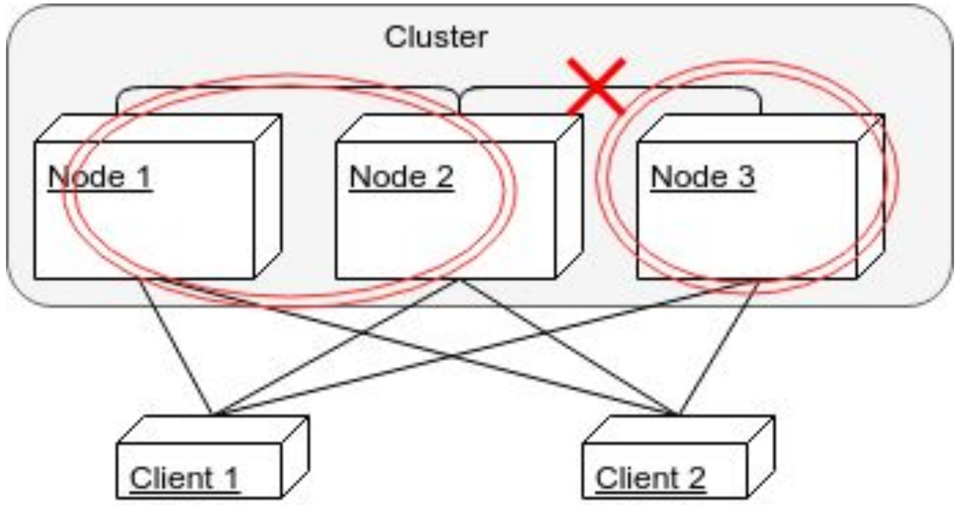
- Sharding (splitting of data) is done at the *replica set* level, hence it involves more than one cluster (a shard is on top of a replica set)
- Only the *primary node* in a replica set answers write requests, but read requests can -depending on the specifics of the configuration- be answered by every node (including *secondary nodes*) in the set
- Updates flow only from the primary to the secondary
- If a primary node fails, or discovers it is connected to a minority of nodes, a secondary of the same replica set is elected as the primary
- Arbiters (MongoDB instances without data) can assist in breaking a tie in elections.
- Data are balanced across replica sets
- Since a quorum has to be reached, it is better to have an odd number of voting members (the *arbiter* in this diagram is only illustrative)

MongoDB vs CouchDB Clusters

- MongoDB clusters are considerably more complex than CouchDB ones
- MongoDB clusters are less available, as - by default - only primary nodes can talk to clients for read operations, (and exclusively so for write operations)
- MongoDB software routers (MongoS) must be embedded in application servers, while any HTTP client can connect to CouchDB
- Losing two nodes out of three in the CouchDB architecture shown, means losing access to one quarter of data
- Losing two nodes in the MongoDB example implies losing *write access* to half the data (although there are ten nodes in the cluster instead of three), and possibly *read access* too, depending on the cluster configuration parameters and the nature (primary or secondary) of the lost nodes
- Some features (such as unique indexes) are not supported in MongoDB sharded environments

These differences are rooted in different approaches to an unsolvable problem, a problem defined by *Brewer's CAP Theorem*

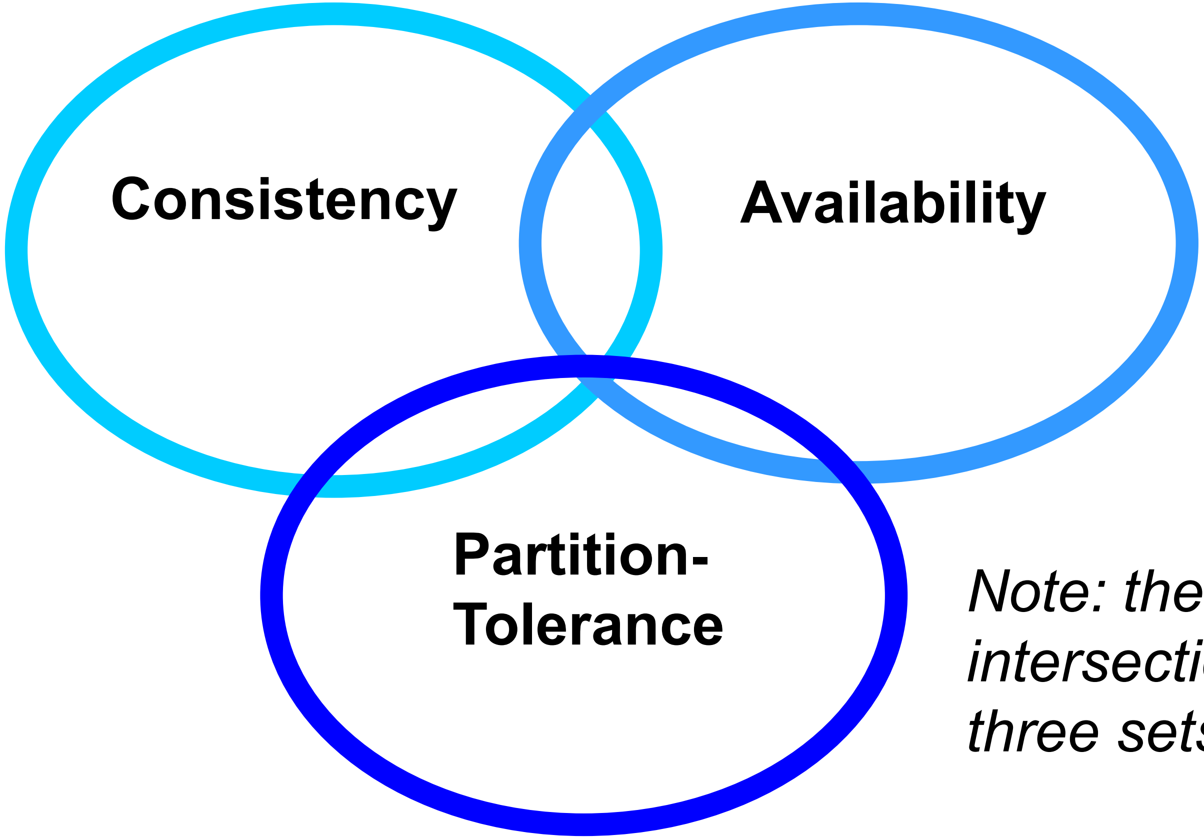
Consistency, Availability, Partition-Tolerance



- **Consistency**: every client receiving an answer receives the same answer from all nodes in the cluster
- **Availability**: every client receives an answer from any node 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...



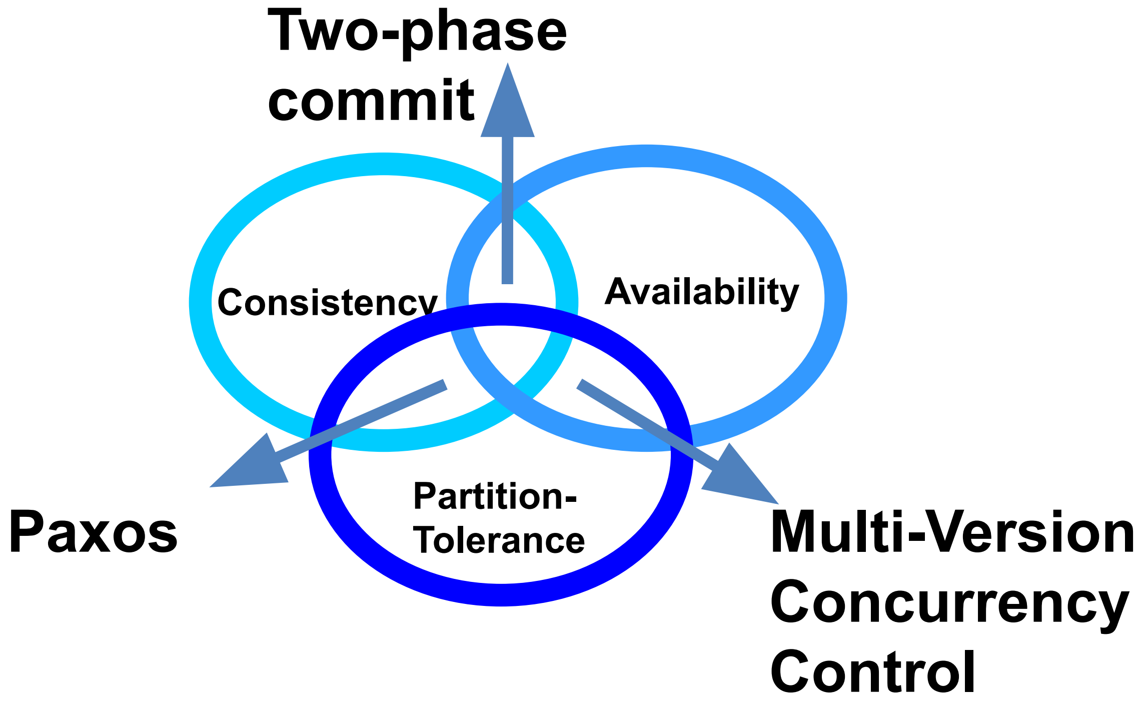
Note: the intersection of the three sets is empty

Brewer’s CAP Theorem

...but not quite:

- While the theorem shows all three qualities are 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
- 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 (and MongoDB, to same extent), 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 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 is either a *proposer* or an *accepter* :
  - a *proposer* proposes a value (with a timestamp)
  - an *accepter* can accept or refuse it (e.g. if the *accepter* receives a more recent value)
- When a proposer has received a sufficient number of acceptances (a *quorum* is reached), and a confirmation message is sent to the *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 quorum) will not send responses, hence the availability is compromised

Availability and Partition-tolerance: Multi-Version Concurrency Control (MVCC)

- 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 as the *current value*.
- In case of cluster partition and concurrent requests with the same revision number going to two partitioned nodes, both are accepted, but once the partition is solved, there would be a *conflict*. Conflict that would have to be solved somehow (CouchDB returns a list of all current conflicts, which are then left to be solved by the application).

Addendum: The Peculiar Case of the Blockchain

- Blockchains can be described as *distributed, inalterable, verifiable, databases*. So, how do they map into this classification? (To fix ideas, let's focus just on the Bitcoin distributed ledger.)
- Bitcoin works on a cluster of peer-to-peer nodes, each containing a copy of the entire database, operated by different -possibly malicious- actors.
- Since new nodes can enter the system at any time, and every node has the entire database, availability is not an issue even in case of a partition, but consistency cannot be assured, since you cannot trust a single node.
- To achieve consistency, Bitcoin uses a form of MVCC based on *proof-of-work* (which is a proxy for the computing power used in a transaction) and on repeated *confirmations* by a majority of nodes of a history of transactions.
- The guarantee of Bitcoin database security is that no single actor can amass enough of the cluster computing power (with 6 confirmations, an actor that controls 18% of the computing power has a 1% probability of compromising a transaction.

MongoDB vs CouchDB Clusters

- While CouchDB uses MVCC, MongoDB uses a mix of two-phase commit (for replicating data from primary to secondary nodes) and Paxos-like (to elect a primary node in a replica-set)

- From the MongoDB 3.6. Documentation: <<*A network partition may segregate a primary into a partition with a minority of nodes. When the primary detects that it can only see a minority of nodes in the replica set, the primary steps down as primary and becomes a secondary. Independently, a member in the partition that can communicate with a majority of the nodes (including itself) holds an election to become the new primary.*>>

- The different choices of strategies explains the different cluster architectures of these two DBMSs

Why Document-oriented DBMS for Big data?

While Relational DBMSs are extremely good for ensuring consistency and availability, the normalization that lies at the heart of a relational database model implies fine-grained data, which are less conducive to partition-tolerance 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 and an address table, all relate to each other.
- 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.

Sharding

- Sharding is the partitioning of a database “*horizontally*”, i.e. the database rows (or documents) are partitioned into subsets that are stored on different servers. Every subset of rows is called a *shard.*
- Usually the number of shards is larger than the number of *replicas*, and the number of nodes is larger than the number of replicas (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 it easier to move data files around, e.g. when adding new nodes to the cluster
- The number of shards that split a database dictates the (meaningful) number of nodes: the maximum number of nodes is equal to the number of shards (lest a node contains the same shard file twice)
- 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) that are stored on the same node

Replication and Sharding

- Replication is the action of storing the same row (or document) 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.

- A bit of nomenclature:
    - `n` is the number of replicas (how many times the same data item is repeated across the cluster)
    - `q` is the number of shards (how many files a database is split)
    - `n * q` is the total number of shard files distributed in the different nodes of the cluster

How Shards Look in CouchDB

- This is the content of the `data/shards` directory on a node of a three-node cluster
- The `test` database has q=8, n=2, hence 16 shards files
- The `*.couch` files are the actual files where data are stored
- The sub-directories are named after the document `_ids` ranges

```
NODE 1
|-- 00000000-1fffffff
|   `-- test.1520993373.couch
|-- 20000000-3fffffff
|   `-- test.1520993373.couch
|-- 60000000-7fffffff
|   `-- test.1520993373.couch
|-- 80000000-9fffffff
|   `-- test.1520993373.couch
|-- c0000000-dfffffff
|   `-- test.1520993373.couch
`-- e0000000-ffffffff
    `-- test.1520993373.couch
```

```
NODE 2
|-- 20000000-3fffffff
|   `-- test.1520993373.couch
|-- 40000000-5fffffff
|   `-- test.1520993373.couch
|-- 80000000-9fffffff
|   `-- test.1520993373.couch
|-- a0000000-bfffffff
|   `-- test.1520993373.couch
`-- e0000000-ffffffff
    `-- test.1520993373.couch
```

```
NODE 3
|-- 00000000-1fffffff
|   `-- test.1520993373.couch
|-- 40000000-5fffffff
|   `-- test.1520993373.couch
|-- 60000000-7fffffff
|   `-- test.1520993373.couch
|-- a0000000-bfffffff
|   `-- test.1520993373.couch
|-- c0000000-dfffffff
   `-- test.1520993373.couch
```

Partitions

- A partition is a grouping of logically related rows in the same shard (for instance, all the tweets of the same user)

- Partitioning improves performance by restricting queries to a single shard

- To be effective, partitions have to be relatively small (certainly smaller than a shard)

- A database has to be declared “partitioned” during its creation

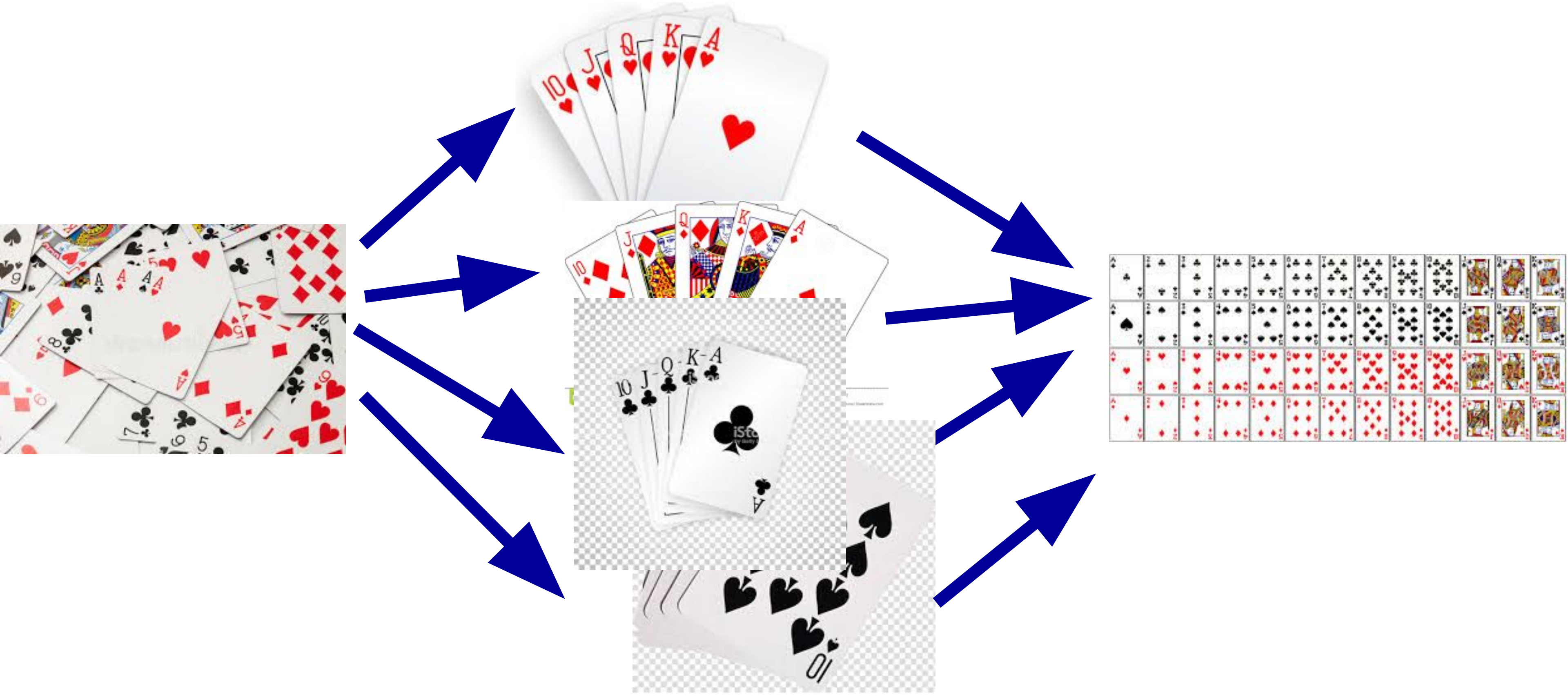
- Partitions are a new feature of CouchDB 3.x

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

Ordering a Deck of Cards with MapReduce



Map

Reduce

Part 2: Introduction to CouchDB

Why Using CouchDB in This Course?

- Is open-source, hence you can peruse the source code and see how things work
- It has MapReduce queries, hence you can understand how this programming paradigm works
- It is easy to setup a cluster
- It has sharding, replication, and partitions
- The HTTP API makes it easy to interact with it

CouchDB Main Features

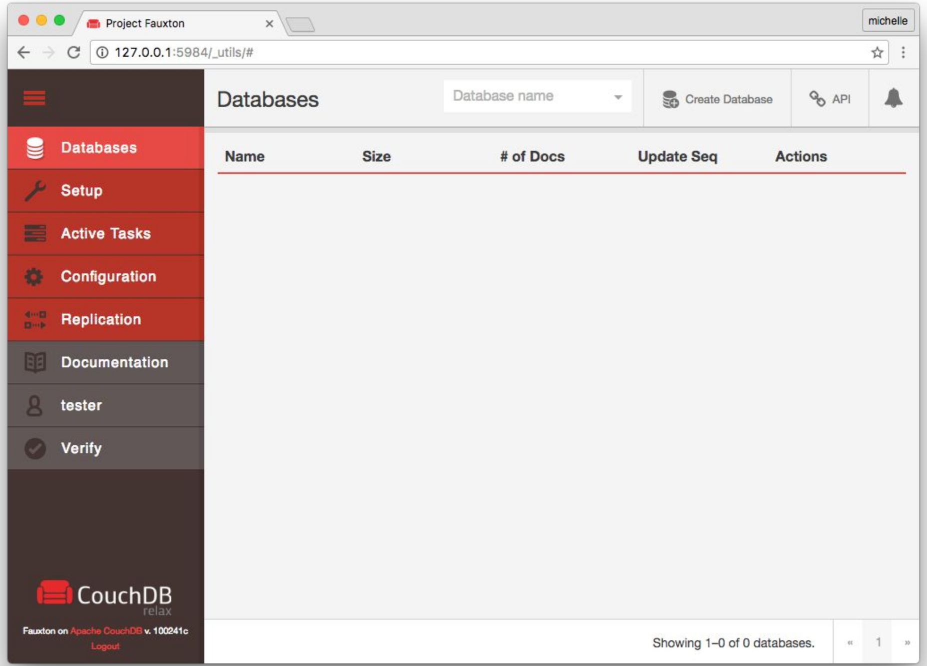
The main features of CouchDB 3.x are:

- Document-oriented DBMS, where documents are expressed in JavaScript Object Notation (JSON)
- 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 marshaling and unmarshaling of data objects
- Support for MapReduce algorithms, including aggregation at different levels
- JavaScript as the default data manipulation language
- Full-text search
- Support of MongoDB query language
- Support of replication
- Support of partitions
- Support of sharding
- Support of clusterized databases

Fauxton User Interface

Typing `http://<hostname>: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
- Run full-text searches
- Modify the configuration
- Manage users
- Set up replications



Name	Size	# of Docs	Update Seq	Actions

Showing 1–0 of 0 databases.

Databases

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

- Adding and deleting a database is done 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”
```

  ...every response's body is a JSON object:

```
["exampledb", "twitter", “instagram”]
```

- In every CouchDB instance there are system databases. These are prefixed by underscore, such as `_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-b8a039a8143b474b
3601b389081a9eec"}
```

- To retrieve a document:

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

Response: `200`

```
{"_id":"c43bcff2cd577d8ab2933cdc0011f8","_rev":"1-b8a039a8143b474b3601b38908
1a9eec","type":"account","holder":"Alice","initialbalance":1000}
```

System Properties of Documents

- `_id`: is the ID of a single document which can be set during the document load; by default it is generated by CouchDB and guaranteed to be unique
- `_rev`: revision number of a document. It is guaranteed to be increasing per-document, i.e. every database instance will pick up the same revision as the current 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` status code, and body equal to:

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

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

Error 409 when Updating Documents

- What happens when we update document `charlie`?

```
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."}
```

Why?

MVCC and Revision Numbers

An example of how two clients querying a single database instance do not lock data and still avoid inconsistency by using a revision number for transactions (e.g. software revision control systems such as Git or Subversion).

	CLIENT 1	CLIENT 2
t1	POST obj1,{email:”a@x.au”} ==> OK, rev:1	
t2		PUT obj1,rev:1,{email:”b@x.au”} ==> OK, rev:2
t3	PUT obj1,rev:1,{email:”c@x.au”} ==> ERROR	
t4	GET obj1 ==> OK, rev:2, {name:“b@x.au”}	
t5	PUT obj1,rev:2,{name:”c@x.au”} ==> OK, rev:3	

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

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). This is just the application of MVCC, and <u>has nothing to do</u>, per se, with clustered databases

What Happens When a Conflict Happens on a Cluster of CouchDB Nodes?

- When the revision number is not sent when documents are updated a `409` is raised in a single-node database, but something similar may happen on a clustered database even if the revision number is sent
- When a cluster is partitioned and two nodes receive two different updates of the same document, two different revisions are added. However, only one of these is returned as the current revision (the “winning” revision is computed deterministically, hence guaranteed to be the same on any node of the cluster). At any rate, the “losing” revision is still stored in the database, and can be used to solve the conflict.
- To help in 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: `200`

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

- To check the deletion:

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

Response: `404` (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 can be retrieved with a bit of effort (e.g. 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: `200`

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

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

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

Response: `200` (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, e.g. 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 or 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: `200`

```
[{"ok":true,"id":"c43bcff2cdbb577d8ab2933cdc18f402","rev":"1-c8cae35f4287628c6
  96193172096c988"},{"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=1-26074febbe9a4a0e818f7d5587
  d7411a" --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 are useful for sharing binary data or big JSON documents that do not require parsing

Querying a CouchDB Database

CouchDB has three mechanisms to select a set of documents that exhibit certain features:

- *MapReduce Views*: results of MapReduce processes that are written as B-tree indexes to disk and become part of the database

- *Mango Queries*: queries expressed in JSON, following the MongoDB queries syntax (Mango queries can also use B-tree indexes to speed-up computations)

- *Full-text search*: queries that can search form specific works or portions of words(via the *Closueau* plugin, running in a separate JVM instance)

Views, but not Relational Ones!

- **CouchDB *views* <u>are not</u>:**
  - *Relational SQL Queries* (they are not volatile)
  - *Relational 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 and store aggregated data (which is great for analytics), but are inflexible and use a lot of storage

CouchDB Views #1

- Views are definitions of MapReduce jobs that are updated as new data come in and are persisted.

- Let’s suppose we have a database composed of document such as:

```
{"_id": "c43bcff2cdbb577d8ab2933cdc18f402",
 "name": "Chris White",
 "type": "transaction",
 "amount": 100}
```

- Let’s write a view that returns the balance of every account holder:
    - map part: `function(doc) {emit([doc.name], doc.amount);}`
    - reduce part:

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

CouchDB Views #2

- `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 (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**)

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 key ascending

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 aggregating data at different levels:

```
["a"], 7
["c"], 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`);
  - may be passed the level of aggregation (`group_level`);
  - may return only a subset of keys (`start_key, end_key` parameters);
  - 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)` (<u>don't</u>! 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
  - are computed (indexed) in the background by a daemon called *ken*, hence it may take some time before an update operation is refleced in the views (this feature has been introduced in CouchDB 3.x);
  - are re-computed every time one of the views in the same the design document is updated, hence be careful in packing too many views in the same design document;
  - can be defined in languages other than JavaScript
  - can use libraries (only in the *map* part of the view definition)
  - cannot be passed custom parameters, either during indexing or during querying
- Computation of views can be influenced only by the document itself (referential transparency):
  - no reading of other documents
  - no passing of parameters during indexing

CouchDB Views #6

- Keys are case-sensitive, as they are in RDBMS indexes

- Pagination is available through the use of `skip` and `limit` parameters

- Since views are updated only when used (lazy evaluation), their results would have to wait for the view update… unless `stale` parameter is set to `update_after` (forcing the update of the view to happen <u>after</u> the results are returned to the client)

- 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) ])`

CouchDB Views #7

- For instance, this is not a referentially transparent way to compute averages:

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

- But this is (both examples taken from the class of 2017):

```
function (keys, values, rereduce) {
    results = {'sum':0, 'count':0};
    if (rereduce) {
        for(var i = 0; i < values.length; i++){
            results.sum += values[i].sum;
            results.count += values[i].count;}
        return results;
    } else {
        for(var j = 0; j < values.length; j++){
            results.sum += values[j].sum;
        }
        results.count = values.length;
    return results;  }}
```

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":"deposit
  ","amount":200}
{"_id":"94b6eb8511034068a5bdc6bcf100244b","account":"bob","type":"withdra
  wal","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 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 aggregated balance):

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

How to do OLAP (sort of) #1

- Given a 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"
},
...
```

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.
- For example, 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) #3

- 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
```

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

Update and Validate Functions

- Documents can be be enriched prior to storing into the database using an *update function* (for instance, by adding a timestamp):

```
function(doc, req){
    doc['tstamp'] = new Date();
    return [doc, 'Added timestamp!']
}
```

- Authorization can be checked with a *validation function*:

```
function(newDoc, oldDoc, userCtx, secObj) {
    if (!newDoc.name) {
        throw({forbidden: 'doc.name is required'});
    }
    if (userCtx.roles.indexOf('manager') === -1) {
        throw({forbidden: 'only managers can update'});
    }
}
```

Mango Queries

- Queries are expressed in JSON, as in:

```
{"selector": {
        "year": {"$gt": 2010}
    },
    "fields": ["_id", "_rev", "year", "title"],
    "sort": [{"year": "asc"}],
    "limit": 2,
    "skip": 0,
    "execution_stats": true
}
```

- The query above selects all documents with the `year` attribute greater then `2010`, displaying `_id, _rev, year, title` attributes, sorted by `year` (ascending), limited to the first two documents, and displaying statistics about the query

Mango Indexes

- To speed-up queries, Mango can use B-tree indexes on attributes
- These indexes are defined as JSON, as in:

```
{"index": {
       "fields": ["foo"]
   },
   "name" : "foo-index",
   "type" : "json"}
```

- This index (`foo-index`) orders documents by the `foo` attribute
- More than one attribute can be used in an index

Full-text Queries

- Queries are expressed either as GET or POST (JSON) queries, as in:

```
curl -XGET
"http://${user}:${pass}@${masternode}:5984/twitter/_design/text
search/_search/text?q=language:en+AND+(text:weekend+AND+text:da
ys)"
```

- The query above selects all documents in English that contains either the term `weekend` or the term `days`

- These queries use the Lucene search engine (as a separate JMV process), and can index: text fields, continuous fields (such as geographic coordinates), categorical fields (*faceted searches)* such as the language field above

CouchDB Application Development

- Java: Ektorp, LightCouch
- 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

Partitioned Databases

- A partitioned database can be created by adding the parameter *partitioned* :

```
curl -X PUT
"http://localhost:5984/testpart?partitioned=true"
--user '<username>:<password>'
```

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 of replicas (`n`): `curl -XPUT "http://<hostname>:5984/test?n=3&q=4"`
  - Write operations complete successfully only if the document is committed to a quorum of replicas (usually a simple majority, parameter `w`)
  - Read operations complete successfully only if a quorum of replicas (parameter `r`) 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

Part 3: Workshop on CouchDB

Installation and Operation of a 3-node Cluster with Docker

- Clone this Git repo: `https://github.com/AURIN/comp90024`
- Open the `README.md` and install the required software (as I did)
- Open the `couchdb/README.md` and follow the instructions, as I am about to do...

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] *Analysis of hashrate-based double-spending*,Meni Rosenfeld, 2012

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