In this session:

- You will learn how to use the NoSQL database MongoDB and its `map-reduce` implementation using using the python interface `pymongo`
- You will implement a *simple association rule miner* for market basket analysis on top of MongoDB

1 Installing and running MongoDB

Download and install MongoDB from `http://www.mongodb.org/`. It is also possible to install it using package managers like `apt-get` in Linux or `brew` for Mac OS.

You will have two executables that we will use: the server (`mongod`) and the client (`mongo`).

To start the server on your machine, type on a command-line terminal:

```
mongod -dbpath <yourpath>
```

The database files will be placed where you indicated. If you don't give the mongo server the argument `dbpath`, it stores the databases somewhere of its choosing. You should leave this console open.

To start the client, open a new console and invoke the client

```
mongo
```

You should see the mongo client's interpreter prompt, it is just waiting for your instructions. For example you can tell it to show the databases it holds by typing `show dbs`. If it is your first time using MongoDB you may have no databases yet. [1]

2 Using MongoDB

Each MongoDB database consists of "collections" (vague analogy: tables). Each collection consists of "documents" (vague analogy: tuples). The crucial differences with relational databases are that, first, there is no schema at all and, second, that we are not even in first normal form: each document is actually an arbitrary mapping of keys (also called "properties") to values, and values can be anything, for instance... other documents again.

(If you happen to know the JSON format, this is exactly what we are talking about.)

Things you can say to your mongo client:

```
> db                                    # tells you the database you are connected to
> show dbs                              # shows the available databases on the server,
```

[1] In this lab session we are using MongoDB locally. Of course, this database is designed to be replicated and distributed for scalability.

```
                                        # "test" among them
> use foo                               # connects to database named 'foo'
> show collections                      # shows the collections on the database you are
                                        # connected to
> db.testing.insert({"a":1})            # inserts document {"a":1} in collection "testing"
                                        # of current database
                                        # (creates collection testing if it does not exist)
> db.new_coll.insert({"b":0,"c":"hello"})   # creates new collection new_coll in database test
                                        # and inserts the document in it
> db.new_coll.insert({"b":0,"d":"again"})   # further insertions
> db.new_coll.insert({"b":0,"c":"goodbye"})
> db.new_coll.find()                    # retrieves all documents in collection new_coll
> db.new_coll.find({"b":0})             # retrieves matching documents
> db.new_coll.find().sort({"c":-1})     # sort the result according to property "c"
                                        # (1:ascending, -1:descending)
> db.new_coll.remove({"b":0})           # removes from collection all matching documents
> db.new_coll.drop()                    # deletes the collection
> exit
```

Have a look at the console where the server was left running.

3 Installing and using pymongo

Please install the `pymongo` library to your installation of Python; "sudo pip install pymongo" or a related command may work if you have "pip"; or, go to `http://api.mongodb.org/python/current/index.html` (or simply search for "Python" in the mongodb.org searchbox) and follow the Installation indications.

The `pymongo` API to handle MongoDB databases from Python works very well and is well-documented `http://api.mongodb.org/python/current/`.

You can run each of the following snippets from the python prompt, or save the python code into a file and run it. Understand the action of each instruction!

```
from pymongo import MongoClient

# server on localhost
conn = MongoClient()
db = conn.test    # db = conn.foo would connect to database 'foo'

db.testing.insert({"j":5, "k":6, "m":"nopqr" })
db.testing.insert({"j":15, "k":6, "m":"nopqr" })

d = {}
d['j'] = 10
d['k'] = 6
for i in range(4):
  d['a'*i] = i
db.testing.insert(d)

d = {}
d['j'] = 8
d['k'] = 6
```

```
d['b'] = "Hi there!"
db.testing.insert(d)

docs = db.testing.find({"j":10})

for doc in docs:
  print "(j:10)", doc

for doc in db.testing.find({"k":{"$lt":7}}):
    print "(k<7)", doc

for doc in db.testing.find({"j":{"$gt":9}}):
    print "(j>9)", doc

db.testing.remove({ "k" : 6 })
```

Try also incorporating the other methods you have seen earlier (.sort(), .drop()...) into the Python code. You may need to consult the online help for `pymongo`.

4 An example of `map-reduce`

Suppose we have created a database called 'foo' and that within it we have a collection called 'corpus'. Each document in this collection has a sigle property (besides the '_id' property): 'contents', whose value is a list of words constituting the text.

Something like that could be achieved with the following python code, for example:

```
from os import path
from codecs import encode, decode
from pymongo import MongoClient

conn = MongoClient()
db = conn.foo

with open("text.txt") as f:
    text = []
    for line in f:
        for word in line.strip().split():
          word = decode(word.strip(),'latin2','ignore')
          text.append(word)
        d = {}
        d['content'] = text
        db.corpus.insert(d)
```

The following map-reduce program counts the number of occurrences of each word in the whole corpus.

```
from pymongo import MongoClient
from bson.code import Code

conn = MongoClient()
db = conn.foo

mapper = Code("""
function() {
    for (var i = 0; i < this.content.length; i++) {
        emit(this.content[i],1);
        }
    }
                """)

reducer = Code("""
function(key,values) {
    var total = 0;
    for (var i = 0; i < values.length; i++) {
        total += values[i];
        }
        return total;
    }
                 """)

r = db.corpus.map_reduce(mapper, reducer, "counts")
```

There are a few things worth noting.

- the mapper and the reducer must be written in JavaScript (to match easily the JSON format of the documents),
- the map-reduce call is associated to a specific collection in the database,
- each mapper is associated to a document in the collection, available as "this",
- and some magic lines are needed to import into Python the Mongo map-reduce handlers.

In this case, the mappers just traverse the words in the list that forms the 'content' property of their associated document, and emit them with a count of 1. The output is obtained as a new collection within the same Mongo database: it is given as name the last parameter in the map_reduce call, here 'counts'. Run `db.counts.find().sort("value":-1)` to see them after running map_reduce, most frequent words first.

5 Market basket analysis

Market basket analysis refers to the type of analysis that one does over a database of customer's purchases (typically called *transactions*). In this lab session, we will do this over the cvs file `groceries.csv`. If you look at the first few lines of this file, you will see that each line is a transaction, which is basically the set of items that a customer has purchased together.

```
$ head -n 5 groceries.csv
citrus fruit,semi-finished bread,margarine,ready soups
tropical fruit,yogurt,coffee
whole milk
pip fruit,yogurt,cream cheese ,meat spreads
other vegetables,whole milk,condensed milk,long life bakery product
```

For example, the first customer (line 1) purchased fruit, bread, margarine and soup. The second customer (line 2) purchased fruit, yogurt and coffee. And so on.

The idea of market basket analysis is to figure out consumer's buying habits in order to exploit this knowledge in marketing campaigns or product sales etc.

5.1 Association rules

Association rules offer a way to understand consumer buying habits. In this lab session we will consider a simplified version of association rules having the following general form:

$$\textbf{\textit{IF}}\ item_1\ \textit{is purchased}\ \textbf{\textit{THEN}}\ item_2\ \textit{is also purchased}$$

This means that you will have to find pairs of items that are typically bought together. The validity of an association rule is measured using two parameters: *support* and *confidence*.

Support. The support of a rule "$item1 \to item2$" is the number of times $item1$ appears in transactions. It can be given in absolute terms or in relative terms (as a percentage).

Confidence. The confidence of a rule "$item1 \to item2$" is the percentage of transactions containing $item1$ that also contain $item2$.

As a simple example to make sure you understand the concepts, suppose our transaction database is:

```
a,b,c
a,c
a,d
```

Then the association rule "$a \to d$" has support 100% and confidence 33%, and the rule "$b \to c$" has support 33% and confidence 100%

5.2 Association rule mining

The problem of association rule mining is the following: given a *support threshold* s and a *confidence threshold* c (in percentage), find all the rules that have support at least s and confidence at least c.

As a general strategy to solve the poblem, it will be useful to have the support counts of single items (to figure out potential left-hand-sides of rules), and support counts of **pairs** of items (to figure out

6 Your task

You will have to write programs that solve the association rule mining problem. It is compulsory to use the *mapreduce* mechanism to do the counting of "single item" supports and "pair of items" supports.

I propose that you write the following programs (in `python` preferably). This is just a suggestion, you can organize your code as you wish.

- A program that reads the input file `groceries.csv` and creates a MongoDB collection where documents will be the transactions, that is, each document will hold the set of items for that transaction.
- A program with map reduce code that "counts" in how many transactions each item appears, and also in how many transactions pairs of items appear.
- A program that given support and confidence thresholds, uses the counts computed by the mapreduce script above to generate the list of association rules that pass both support and confidence thresholds.

Task 1. With these programs, fill out the missing values of the following table:

Row	*Support*	*Confidence*	*Nr. of association rules found*
1	1	25	?
2	1	50	?
3	1	75	?
4	5	50	?
5	10	25	?
6	20	25	?
7	50	25	?

Task 2. Give the list of association rules found corresponding to rows 4, 5, and 6 of the table of Task 1.

7 Deliverables

Rules: 1. You can solve the problem alone or with one other person, but at most one lab assignment with the same person in the whole course. 2. No plagiarism; don't discuss your work with others, except your teammate if you are solving the problem in two; if in doubt about what is allowed, ask us. 3. If you feel you are spending much more time than the rest of the group, ask us for help. Questions can be asked either in person or by email, and you'll never be penalized by asking questions, no matter how stupid they look in retrospect.

To deliver: You must deliver a brief report describing your results and the main difficulties/-choices you had while implementing this lab session's work. You also have to hand in the source code of your implementations.

Procedure: Submit your work through the raco platform as a single zipped file.

Deadline: Work must be delivered within 3 weeks from the lab session you attend. Late submissions risk being penalized or not accepted at all. If you anticipate problems with the deadline, tell me as soon as possible.