

# Analysis of unstructured data

## Lecture 6 - data storage

Janusz Szwabiński

Overview:

- CVS file
- Relational databases
  - SQLite
  - PostgreSQL
  - MySQL
  - Firebird
  - SQLAlchemy
- Case study - SQLite, Pandas and big data sets
- No-SQL databases
  - CouchDB
  - MongoDB
- Case study - CouchDB, Python and Twitter

In [51]:

```
%matplotlib inline
import matplotlib.pyplot as plt
```

## CSV files

### csv module

In [2]:

```
import csv

with open('python_to_csv.csv', 'w') as f:
    fieldnames = ['first_name', 'last_name']
    writer = csv.DictWriter(f, fieldnames=fieldnames, delimiter="|")

    writer.writeheader()
    writer.writerow({'first_name': 'Baked', 'last_name': 'Beans'})
    writer.writerow({'first_name': 'Lovely', 'last_name': 'Spam'})
    writer.writerow({'first_name': 'Wonderful', 'last_name': 'Spam'})
```

In [3]:

```
!cat python_to_csv.csv
```

## Pandas

In [4]:

```
import pandas as pd
```

```
df = pd.DataFrame({'AAA' : [4,5,6,7], 'BBB' : [10,20,30,40], 'CCC' :  
[100,50,-30,-50]})  
df
```

Out[4]:

	AAA	BBB	CCC
0	4	10	100
1	5	20	50
2	6	30	-30
3	7	40	-50

In [5]:

```
with open('pandas_to_csv.csv','w') as f:  
    df.to_csv(f,sep='\t')
```

In [6]:

```
!cat pandas_to_csv.csv
```

	AAA	BBB	CCC
0	4	10	100
1	5	20	50
2	6	30	-30
3	7	40	-50

# Relational databases

## Databases - short introduction

### Definitions

- **a database** - an organized collection of data (in the narrower sense - a digital collection of data)
- **a database-management system (DBMS)** - a computer-software application that interacts with end-users, other applications, and the database itself to capture and analyze data

### Types

- **flat file database** - a database stored as an ordinary unstructured file (a "flat file"). To access the structure of the data and manipulate it on a computer system, the file must be read in its entirety into the computer's memory
- **hierarchical database** - data is organized into a tree-like structure. The data is stored as records which are connected to one another through links. A record is a collection of fields, with each field containing only one value. The entity type of a record defines which fields the record contains. Examples: file systems, IBM IMS (since 1966)
- **relational database** - data organized into one or more tables (or "relations") of columns and rows, with a unique key identifying each row. Virtually all relational database systems use SQL (Structured Query Language) for querying and maintaining the database
- **object-oriented database** - data is represented in the form of objects as used in object-oriented programming. From the conceptional point of view very popular in the 1990s. Examples: Versant, db4o, LoXiM
- **object-relational database** - similar to a relational database, but with an object-oriented database model: objects, classes and inheritance are directly supported in database schemas and in the query language. Examples: Omniscience, UniSQL, Valentina, PostgreSQL
- **streaming databases** - to manage continuous data streams (with queries which are continuously executed until they are explicitly uninstalled)
- **temporal databases** - a relational database offering timestamps determining e.g. the time span in which data is valid
- **non-relational databases (NoSQL)** - data stored in structures different from relational databases (e.g. key-value, wide column, graph, or document)

## SQLite

- a relational database management system contained in a C programming library
- a popular choice as embedded database software for local/client storage
- the most widely deployed database engine

### Basic features

- no server infrastructure required
- no configuration needed
- bindings to many programming languages (Perl, PHP, Ruby, C++, Python, Java, .NET)
- support for ODBC
- single binary file for each database (up to 140 TB)
- ACID-compliant (*Atomicity, Consistency, Isolation, Durability*)
- most of the SQL 92 standard implemented
- very efficient (in single user mode)

### Creation of databases

In [7]:

```
import sqlite3

# create database (in does not exist) and establish a connection
conn = sqlite3.connect('moja_baza.db')
c = conn.cursor()

# create a table
c.execute('''CREATE TABLE my_table
            (id TEXT, my_var1 TEXT, my_var2 INT)''')

# insert one data row
c.execute("INSERT INTO my_table VALUES ('ID_2352532','YES', 4)")

# insert multiple rows
multi_lines = [ ('ID_2352533','YES', 1),
                 ('ID_2352534','NO', 0),
                 ('ID_2352535','YES', 3),
                 ('ID_2352536','YES', 9),
                 ('ID_2352537','YES', 10)
               ]
c.executemany('INSERT INTO my_table VALUES (?,?)', multi_lines)

# commit changes
conn.commit()

# close connection
conn.close()</pre
```

Let us check the content of the working directory:

In [8]:

```
! ls -lh
```

razem 1,6M

```
-rwxrwxr-x 1 szwabin szwabin 256K lis 13 09:44 6_data_storage.ipynb
-rwxrwxr-x 1 szwabin szwabin  82 lis 13 09:44 config.ini
-rwxrwxr-x 1 szwabin szwabin 130K lis 13 09:44 demodb.png
-rwxrwxr-x 1 szwabin szwabin  40K lis 13 09:44 exampledb.png
-rwxrwxr-x 1 szwabin szwabin  79K lis 13 09:44 futon_new_db_2.png
-rwxrwxr-x 1 szwabin szwabin  16K lis 13 09:44 futon_new_db.png
-rwxrwxr-x 1 szwabin szwabin  89K lis 13 09:44 futon_new_doc_2.png
-rwxrwxr-x 1 szwabin szwabin  86K lis 13 09:44 futon_new_doc.png
-rwxrwxr-x 1 szwabin szwabin  75K lis 13 09:44 futon.png
-rwxrwxr-x 1 szwabin szwabin  98K lis 13 09:44 mapreduce.png
-rwxrwxr-x 1 szwabin szwabin 120K lis 13 09:44 mapreduce_res.png
-rw-r--r-- 1 szwabin szwabin 2,0K lis 13 09:45 moja_baza.db
-rwxrwxr-x 1 szwabin szwabin 129K lis 13 09:44 mongo_find.png
-rwxrwxr-x 1 szwabin szwabin  44K lis 13 09:44 mongo_konsola.png
-rwxrwxr-x 1 szwabin szwabin  56 lis 13 09:45 pandas_to_csv.csv
-rwxrwxr-x 1 szwabin szwabin 8,3K lis 13 09:44 python_mysql.sql
-rwxrwxr-x 1 szwabin szwabin  64 lis 13 09:45 python_to_csv.csv
-rw-rw-r-- 1 szwabin szwabin 127K lis 13 09:44 robo3t.png
-rwxrwxr-x 1 szwabin szwabin 132K lis 13 09:44 robomongo.png
-rwxrwxr-x 1 szwabin szwabin  11K lis 13 09:44 vincent example.ipynb
-rwxrwxr-x 1 szwabin szwabin  201 lis 13 09:44 wyklad6_dane.txt
```

**Warning!** Re-running of the above code will cause the following error:

In [9]:

```
import sqlite3

# create database (in does not exist) and establish a connection
conn = sqlite3.connect('moja_baza.db')
c = conn.cursor()

# create a table
c.execute('''CREATE TABLE my_table
            (id TEXT, my_var1 TEXT, my_var2 INT)''')

# insert one data row
c.execute("INSERT INTO my_table VALUES ('ID_2352532','YES', 4)")

# insert multiple rows
multi_lines = [ ('ID_2352533','YES', 1),
                 ('ID_2352534','NO', 0),
                 ('ID_2352535','YES', 3),
                 ('ID_2352536','YES', 9),
                 ('ID_2352537','YES', 10)
               ]
c.executemany('INSERT INTO my_table VALUES (?,?)', multi_lines)

# commit changes
conn.commit()

# close connection
conn.close()</pre

```

ERROR:root:An unexpected error occurred while tokenizing input  
 The following traceback may be corrupted or invalid  
 The error message is: ('EOF in multi-line string', (1, 49))

```
-----
-----
OperationalError                                Traceback (most recent call
last)
<ipython-input-9-a2fd36608d0e> in <module>()
      7 # create a table
      8 c.execute('''CREATE TABLE my_table
----> 9             (id TEXT, my_var1 TEXT, my_var2 INT)''')
     10
     11 # insert one data row
```

OperationalError: table my\_table already exists

That is why it is a good practice to put

```
DROP TABLE IF EXISTS my_table;
```

before creating a new table.

In [10]:

```
import sqlite3

# create database (in does not exist) and establish a connection
conn = sqlite3.connect('moja_baza.db')
c = conn.cursor()

#just in case there is already my_table
c.execute('''DROP TABLE IF EXISTS my_table''')
conn.commit()

# create a table
c.execute('''CREATE TABLE my_table
            (id TEXT, my_var1 TEXT, my_var2 INT)''')

# insert one data row
c.execute("INSERT INTO my_table VALUES ('ID_2352532','YES', 4)")

# insert multiple rows
multi_lines =[ ('ID_2352533','YES', 1),
                ('ID_2352534','NO', 0),
                ('ID_2352535','YES', 3),
                ('ID_2352536','YES', 9),
                ('ID_2352537','YES', 10)
              ]
c.executemany('INSERT INTO my_table VALUES (?,?,?)', multi_lines)

# commit changes
conn.commit()

# close connection
conn.close()
```

## Updating data

In [11]:

```
import sqlite3

# open a connection
conn = sqlite3.connect('moja_baza.db')
c = conn.cursor()

# update data
t = ('NO', 'ID_2352533', )
c.execute("UPDATE my_table SET my_var1=? WHERE id=?", t)
print("Total number of rows changed:", conn.total_changes)

# remove row
t = ('NO', )
c.execute("DELETE FROM my_table WHERE my_var1=?", t)
print("Total number of rows changed: ", conn.total_changes)

# add column
c.execute("ALTER TABLE my_table ADD COLUMN 'my_var3' TEXT")

# commit changes
conn.commit()

# print list of columns
c.execute("SELECT * FROM my_table")
col_name_list = [tup[0] for tup in c.description]
print(col_name_list)

# close connection
conn.close()
```

```
Total number of rows changed: 1
Total number of rows changed: 3
['id', 'my_var1', 'my_var2', 'my_var3']
```

## Queries



In [12]:

```
import sqlite3

# open connection
conn = sqlite3.connect('moja_baza.db')
c = conn.cursor()

# print all rows, ordered by my_var2 column
print('-'*30)
for row in c.execute('SELECT * FROM my_table ORDER BY my_var2'):
    print(row)

# print all rows with the value "YES" in column my_var1
# and value <= 7 in my_var2
print('-'*30)
t = ('YES',7,)
for row in c.execute('SELECT * FROM my_table WHERE my_var1=? AND my_var2 <= ?',
t):
    print(row)

# same thing, different method
print('-'*30)
t = ('YES',7,)
c.execute('SELECT * FROM my_table WHERE my_var1=? AND my_var2 <= ?', t)
rows = c.fetchall()
for r in rows:
    print(r)

# close connection
conn.close()
```

```
-----
('ID_2352535', 'YES', 3, None)
('ID_2352532', 'YES', 4, None)
('ID_2352536', 'YES', 9, None)
('ID_2352537', 'YES', 10, None)
-----
('ID_2352532', 'YES', 4, None)
('ID_2352535', 'YES', 3, None)
-----
('ID_2352532', 'YES', 4, None)
('ID_2352535', 'YES', 3, None)
```

## MySQL

- an open-source relational database management system (RDBMS)
- for proprietary use, several paid editions are available
- a central component of the LAMP open-source web application software stack (LAMP is an acronym for "Linux, Apache, MySQL, Perl/PHP/Python")
- used in many applications, e.g. TYPO3, MODx, Joomla, WordPress, phpBB, MyBB, and Drupal
- used in many large-scale websites, including Google (though not for searches), Facebook, Twitter, Flickr and YouTube
- written with efficiency in mind rather than with compliance with SQL standards
- features as available in MySQL 5.6 (some of them missing in earlier versions):
  - a broad subset of ANSI SQL 99, as well as extensions
  - cross-platform support
  - stored procedures, using a procedural language that closely adheres to SQL/PSM
  - triggers
  - cursors
  - updatable views
  - transactions with savepoints when using the default InnoDB Storage Engine
  - ACID compliance when using InnoDB and NDB Cluster Storage Engines
  - query caching
  - sub-SELECTs (i.e. nested SELECTs)
  - built-in replication support configurations using Galera Cluster.[73]
  - full-text indexing and searching
  - unicode support
  - commit grouping
- minor updates released every 2 months
- MySQL and Python:
  - a third-party module required:
    - MySQL Connector/Python  
(<http://dev.mysql.com/downloads/connector/python/2.0.html>  
(<http://dev.mysql.com/downloads/connector/python/2.0.html>))
    - MySQLdb (<http://mysql-python.sourceforge.net/MySQLdb.html> (<http://mysql-python.sourceforge.net/MySQLdb.html>))

### Database connection

In [2]:

```
import mysql.connector
conn = mysql.connector.connect(host='localhost', database='python_mysql', user='demouser', password='python')
print(conn)
```

```
<mysql.connector.connection.MySQLConnection object at 0x7fcf20181c18>
```

In [3]:

```
conn.close()
```

In [4]:

```
%%writefile config.ini
[mysql]
host = localhost
database = python_mysql
user = demouser
password = python
```

Overwriting config.ini

In [5]:

```
from configparser import ConfigParser

def read_db_config(filename='config.ini', section='mysql'):
    """ Read connection configuration file, return the data as a dict"""
    # create the parser, read the file
    parser = ConfigParser()
    parser.read(filename)

    # read sections
    db = {}
    if parser.has_section(section):
        items = parser.items(section)
        for item in items:
            db[item[0]] = item[1]
    else:
        raise Exception('{0} not found in the {1} file'.format(section,
filename))

    return db
```

In [6]:

```
read_db_config()
```

Out[6]:

```
{'database': 'python_mysql',
 'host': 'localhost',
 'password': 'python',
 'user': 'demouser'}
```

In [7]:

```
conn_data = read_db_config()
conn = mysql.connector.connect(**conn_data)
```

In [8]:

```
print(conn)
```

```
<mysql.connector.connection.MySQLConnection object at 0x7fcf200fe3c8>
```

## Reading data from database

In [10]:

```
cursor = conn.cursor()  
cursor.execute("SELECT * FROM books")
```

In [11]:

```
row = cursor.fetchone()  
print(row)
```

```
(1, 'Bel and the Dragon ', '123828863494')
```

In [12]:

```
while row is not None:  
    print(row)  
    row = cursor.fetchone()
```

(1, 'Bel and the Dragon ', '123828863494')  
(2, 'Daughters of Men ', '1234404543724')  
(3, 'The Giant on the Hill ', '1236400967773')  
(4, 'Marsh Lights ', '1233673027750')  
(5, 'Mr. Wodehouse and the Wild Girl ', '1232423190947')  
(6, 'The Fairy Castle ', '1237654836443')  
(7, 'The Girl Who Walked a Long Way ', '1230211946720')  
(8, 'The Runaway ', '1238155430735')  
(9, 'The Shrubbery ', '1237366725549')  
(10, 'Tom Underground a play ', '1239633328787')  
(11, 'Anemones of the British Coast ', '1233540471995')  
(12, 'Ask to Embla poem-cycle ', '1237417184084')  
(13, 'Cassandra verse drama ', '1235260611012')  
(14, 'Chidiok Tichbourne ', '1230468662299')  
(15, 'The City of Is ', '1233136349197')  
(16, 'Cromwell verse drama ', '1239653041219')  
(17, 'Debatable Land Between This World and the Next ', '1235927658929')  
(18, 'The Fairy Melusina epic poem ', '1232341278470')  
(19, 'The Garden of Proserpina ', '1234685512892')  
(20, 'Gods Men and Heroes ', '1233369260356')  
(21, 'The Great Collector ', '1237871538785')  
(22, 'The Grecian Way of Love ', '1234003421055')  
(23, 'The Incarcerated Sorceress ', '1233804025236')  
(24, 'Last Tales ', '1231588537286')  
(25, 'Last Things ', '1239338429682')  
(26, 'Mummy Possest poem ', '1239409501196')  
(27, 'No Place Like home ', '1239416066484')  
(28, 'Pranks of Priapus ', '1231359225882')  
(29, 'Ragnarök ', '1230741986307')  
(30, 'The Shadowy Portal ', '1232294350642')  
(31, 'Jan Swammerdam poem ', '1238329678939')  
(32, 'St. Bartholomew's Eve verse drama ', '1230082140880')  
(33, 'Tales for innocents ', '1234392912372')  
(34, 'Tales Told in November ', '1234549242464')  
(35, 'Bel and the Dragon ', '1239374496485')  
(36, 'Daughters of Men ', '1235349316660')  
(37, 'The Giant on the Hill ', '1235644620578')  
(38, 'Marsh Lights ', '1235736344898')  
(39, 'Mr. Wodehouse and the Wild Girl ', '1232744187226')  
(40, 'The Fairy Castle ', '1233729213076')  
(41, 'The Girl Who Walked a Long Way ', '1237641884608')  
(42, 'The Runaway ', '1233964452155')  
(43, 'The Shrubbery ', '1231273626499')  
(44, 'Tom Underground a play ', '1238441018900')  
(45, 'In A Future Chalet School Girl: Mystery at Heron Lake ', '1231377433718')  
(46, 'In Althea Joins the Chalet School: The Secret of Castle Dancin g ', '1232395135758')  
(47, 'In Carola Storms the Chalet School: The Rose Patrol in the Alps ', '1234185299775')  
(48, 'In The Chalet School Goes To It: Gipsy Jocelyn ', '1234645928899')  
(49, 'In Gay from China at the Chalet School: Indian Holiday and Nancy Meets a Nazi ', '1230275004688')  
(50, 'In Jo Returns to the Chalet School: Cecily Holds the Fort and Malvina Wins Through ', '1230839327111')  
(51, 'In Joey Goes to Oberland: Audrey Wins the Trick and Dora of the Lower Fifth ', '1237588408519')  
(52, 'In The Chalet School and the Island: The Sea Parrot ', '1236495378720')

(53, 'In The Chalet School in Exile: Tessa in Tyrol ', '1236588981768')  
(54, 'In The Mystery at the Chalet School: The Leader of the Lost Cause ', '1231308608691')  
(55, "In The New Mistress at the Chalet School: King's Soldier Maid and Swords Crossed ", '1230312140169')  
(56, 'In A Problem for the Chalet School: A Royalist Soldier-Maid and Werner of the Alps ', '1230967619568')  
(57, 'In Three Go to the Chalet School: Lavender Laughs in Kashmir ', '1230127072745')  
(58, 'In Tom Tackles the Chalet School: The Fugitive of the Salt Cave and The Secret House ', '1234238103911')  
(59, 'In Two Sams at the Chalet School: Swords for the King! ', '1230886230089')  
(60, 'In Maids of La Rochelle: Guernsey Folk Tales ', '1233675376783')  
(61, 'Bacon Death ', '1236766330719')  
(62, 'Breakfast First ', '1236432913317')  
(63, 'The Culinary Dostoevski ', '1234582103529')  
(64, 'The Egg Laid Twice ', '1236148226462')  
(65, 'He Kissed All Night ', '1237321964604')  
(66, 'A History of Nebraska ', '1239609581078')  
(67, 'Hombre ', '1235105625585')  
(68, "It's the Queen of Darkness Pal ", '1237435357811')  
(69, 'Jack The Story of a Cat ', '1233766820792')  
(70, 'Leather Clothes and the History of Man ', '1236346938182')  
(71, 'Love Always Beautiful ', '1233800248087')  
(72, 'Moose ', '1232083986943')  
(73, 'My Dog ', '1236297974136')  
(74, 'My Trike ', '1237550454699')  
(75, 'The Need for Legalized Abortion ', '1238912644528')  
(76, 'The Other Side of My Hand ', '1239707352212')  
(77, 'Pancake Pretty ', '1234761413168')  
(78, "Printer's Ink ", '1230702325223')  
(79, 'The Quick Forest ', '1236002513635')  
(80, 'Sam Sam Sam ', '1239666823646')  
(81, 'The Stereo and God ', '1231316672178')  
(82, 'UFO vs. CBS ', '1239778693754')  
(83, 'Vietnam Victory ', '1237098200581')

Other method - fetchall:

In [13]:

```
cursor = conn.cursor()
cursor.execute("SELECT * FROM books")
rows = cursor.fetchall()

print('Total Row(s):', cursor.rowcount)
print(rows[:5])
print(rows[-5:])
```

```
Total Row(s): 83
[(1, 'Bel and the Dragon ', '123828863494'), (2, 'Daughters of Men ', '1234404543724'), (3, 'The Giant on the Hill ', '1236400967773'), (4, 'Marsh Lights ', '1233673027750'), (5, 'Mr. Wodehouse and the Wild Girl ', '1232423190947')]
[(79, 'The Quick Forest ', '1236002513635'), (80, 'Sam Sam Sam ', '1239666823646'), (81, 'The Stereo and God ', '1231316672178'), (82, 'UFO vs. CBS ', '1239778693754'), (83, 'Vietnam Victory ', '1237098200581')]
```

Use fetchall with care - for large databases it may lead to a memory overflow. In this case it is better to use the fetchmany function, which fetches a sample of data of a given size:



In [14]:

```
cursor.execute("SELECT * FROM books")
while True:
    print('- '*40)
    rows = cursor.fetchmany(10)
    if not rows:
        break
    for row in rows:
        print(row)
```

```

-----
(1, 'Bel and the Dragon ', '123828863494')
(2, 'Daughters of Men ', '1234404543724')
(3, 'The Giant on the Hill ', '1236400967773')
(4, 'Marsh Lights ', '1233673027750')
(5, 'Mr. Wodehouse and the Wild Girl ', '1232423190947')
(6, 'The Fairy Castle ', '1237654836443')
(7, 'The Girl Who Walked a Long Way ', '1230211946720')
(8, 'The Runaway ', '1238155430735')
(9, 'The Shrubbery ', '1237366725549')
(10, 'Tom Underground a play ', '1239633328787')
-----
(11, 'Anemones of the British Coast ', '1233540471995')
(12, 'Ask to Embla poem-cycle ', '1237417184084')
(13, 'Cassandra verse drama ', '1235260611012')
(14, 'Chidiok Tichbourne ', '1230468662299')
(15, 'The City of Is ', '1233136349197')
(16, 'Cromwell verse drama ', '1239653041219')
(17, 'Debatable Land Between This World and the Next ', '12359276589
29')
(18, 'The Fairy Melusina epic poem ', '1232341278470')
(19, 'The Garden of Proserpina ', '1234685512892')
(20, 'Gods Men and Heroes ', '1233369260356')
-----
(21, 'The Great Collector ', '1237871538785')
(22, 'The Grecian Way of Love ', '1234003421055')
(23, 'The Incarcerated Sorceress ', '1233804025236')
(24, 'Last Tales ', '1231588537286')
(25, 'Last Things ', '1239338429682')
(26, 'Mummy Possest poem ', '1239409501196')
(27, 'No Place Like home ', '1239416066484')
(28, 'Pranks of Priapus ', '1231359225882')
(29, 'Ragnarök ', '1230741986307')
(30, 'The Shadowy Portal ', '1232294350642')
-----
(31, 'Jan Swammerdam poem ', '1238329678939')
(32, "St. Bartholomew's Eve verse drama ", '1230082140880')
(33, 'Tales for innocents ', '1234392912372')
(34, 'Tales Told in November ', '1234549242464')
(35, 'Bel and the Dragon ', '1239374496485')
(36, 'Daughters of Men ', '1235349316660')
(37, 'The Giant on the Hill ', '1235644620578')
(38, 'Marsh Lights ', '1235736344898')
(39, 'Mr. Wodehouse and the Wild Girl ', '1232744187226')
(40, 'The Fairy Castle ', '1233729213076')
-----
(41, 'The Girl Who Walked a Long Way ', '1237641884608')
(42, 'The Runaway ', '1233964452155')
(43, 'The Shrubbery ', '1231273626499')
(44, 'Tom Underground a play ', '1238441018900')
(45, 'In A Future Chalet School Girl: Mystery at Heron Lake ', '1231
377433718')
(46, 'In Althea Joins the Chalet School: The Secret of Castle Dancin
g ', '1232395135758')
(47, 'In Carola Storms the Chalet School: The Rose Patrol in the Alp
s ', '1234185299775')
(48, 'In The Chalet School Goes To It: Gipsy Jocelyn ', '12346459288
99')
(49, 'In Gay from China at the Chalet School: Indian Holiday and Nan
cy Meets a Nazi ', '1230275004688')
(50, 'In Jo Returns to the Chalet School: Cecily Holds the Fort and

```

Malvina Wins Through ', '1230839327111')

(51, 'In Joey Goes to Oberland: Audrey Wins the Trick and Dora of the Lower Fifth ', '1237588408519')

(52, 'In The Chalet School and the Island: The Sea Parrot ', '1236495378720')

(53, 'In The Chalet School in Exile: Tessa in Tyrol ', '1236588981768')

(54, 'In The Mystery at the Chalet School: The Leader of the Lost Cause ', '1231308608691')

(55, 'In The New Mistress at the Chalet School: King's Soldier Maid and Swords Crossed ', '1230312140169')

(56, 'In A Problem for the Chalet School: A Royalist Soldier-Maid and Werner of the Alps ', '1230967619568')

(57, 'In Three Go to the Chalet School: Lavender Laughs in Kashmir ', '1230127072745')

(58, 'In Tom Tackles the Chalet School: The Fugitive of the Salt Cave and The Secret House ', '1234238103911')

(59, 'In Two Sams at the Chalet School: Swords for the King! ', '1230886230089')

(60, 'In Maids of La Rochelle: Guernsey Folk Tales ', '1233675376783')

(61, 'Bacon Death ', '1236766330719')

(62, 'Breakfast First ', '1236432913317')

(63, 'The Culinary Dostoevski ', '1234582103529')

(64, 'The Egg Laid Twice ', '1236148226462')

(65, 'He Kissed All Night ', '1237321964604')

(66, 'A History of Nebraska ', '1239609581078')

(67, 'Hombre ', '1235105625585')

(68, 'It's the Queen of Darkness Pal ', '1237435357811')

(69, 'Jack The Story of a Cat ', '1233766820792')

(70, 'Leather Clothes and the History of Man ', '1236346938182')

(71, 'Love Always Beautiful ', '1233800248087')

(72, 'Moose ', '1232083986943')

(73, 'My Dog ', '1236297974136')

(74, 'My Trike ', '1237550454699')

(75, 'The Need for Legalized Abortion ', '1238912644528')

(76, 'The Other Side of My Hand ', '1239707352212')

(77, 'Pancake Pretty ', '1234761413168')

(78, 'Printer's Ink ', '1230702325223')

(79, 'The Quick Forest ', '1236002513635')

(80, 'Sam Sam Sam ', '1239666823646')

(81, 'The Stereo and God ', '1231316672178')

(82, 'UFO vs. CBS ', '1239778693754')

(83, 'Vietnam Victory ', '1237098200581')

## Inserting data

In [15]:

```
query = "INSERT INTO books(title,isbn) " \
        "VALUES(%s,%s)"
args = ('A Sudden Light','9781439187036')

cursor.execute(query, args)
conn.commit()
```

In [16]:

```
cursor.execute("SELECT * FROM books")
rows = cursor.fetchall()

print('Total Row(s):', cursor.rowcount)
print(rows[-1])
```

```
Total Row(s): 84
(84, 'A Sudden Light', '9781439187036')
```

### Updating (changing) data in the database

In [17]:

```
query = """ UPDATE books
            SET title = %s
            WHERE id = %s """
data = ('The Giant on the Hill *** TEST ***',37)
cursor.execute(query,data)
conn.commit()
```

In [18]:

```
cursor.execute("SELECT * FROM books WHERE id = 37")
print(cursor.fetchone())
```

```
(37, 'The Giant on the Hill *** TEST ***', '1235644620578')
```

### Deleting data

In [19]:

```
query = "DELETE FROM books WHERE id = %s"
cursor.execute(query,(89,))
conn.commit()
```

In [20]:

```
cursor.execute("SELECT * FROM books")
rows = cursor.fetchall()

print(rows[-1])
```

```
(84, 'A Sudden Light', '9781439187036')
```

### Closing a connection

In [21]:

```
cursor.close()  
conn.close()
```

## PostgreSQL

- object-relational database management system (ORDBMS)
- SQL:2011 compliant
- stored procedures written in many programming languages, including Python and R (via extensions)
- built-in support for many types of indexes
- triggers
- ACID compliant (*Atomicity, Consistency, Isolation, Durability* – Atomowość, Spójność, Izolacja, Trwałość)

Most popular Python modules for working with PostgreSQL:

- Psycopg2 (<http://initd.org/psycopg/> (<http://initd.org/psycopg/>))
- pg8000 (<https://github.com/mfenniak/pg8000> (<https://github.com/mfenniak/pg8000>))

## Firebird

- ANSI SQL-92 compliant
- supports many elements from SQL-99 and SQL:2003 standards
- forked from Borland's open source edition of InterBase 6.0
- stored procedures and triggers
- regular expressions
- ACID compliant
- small footprint (minimal installation is 4Mb, standard is 33Mb)
- no configuration required in the default installation

Python modules:

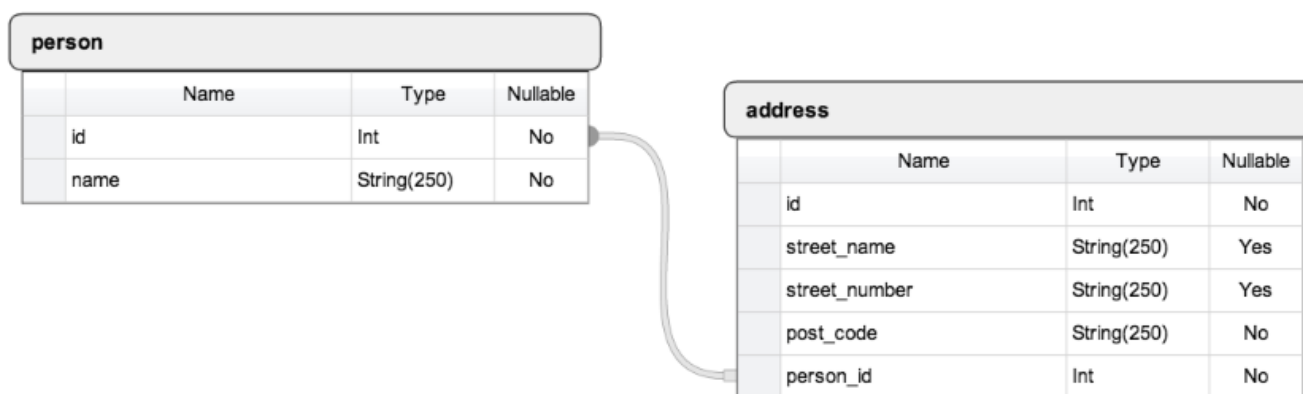
- fdb (<https://pypi.python.org/pypi/fdb/> (<https://pypi.python.org/pypi/fdb/>))
- KInterbasDB (<http://www.firebirdsql.org/en/python-driver/> (<http://www.firebirdsql.org/en/python-driver/>))

## SQLAlchemy

- Python SQL toolkit and Object Relational Mapper
- offers the full power and flexibility of SQL without using the raw SQL queries
- supported databases:
  - SQLite
  - Postgresql
  - MySQL
  - Oracle
  - MS-SQL
  - Firebird
  - Sybase

### Example

We are going to create a database with two tables `person` and `address` in the following design:



In this design, `address.person_id` is the foreign key to the `person` table.

In order to create this database in SQLite we may use the following code:

In [22]:

```
import sqlite3
conn = sqlite3.connect('example.db')
c = conn.cursor()

c.execute('''DROP TABLE IF EXISTS person''')
c.execute('''DROP TABLE IF EXISTS address''')

conn.commit()

c.execute('''
CREATE TABLE person
(id INTEGER PRIMARY KEY ASC, name varchar(250) NOT NULL)
''')
c.execute('''
CREATE TABLE address
(id INTEGER PRIMARY KEY ASC, street_name varchar(250), street_number v
archar(250),
post_code varchar(250) NOT NULL, person_id INTEGER NOT NULL,
FOREIGN KEY(person_id) REFERENCES person(id))
''')

c.execute('''
INSERT INTO person VALUES(1, 'pythoncentral')
''')
c.execute('''
INSERT INTO address VALUES(1, 'python road', '1', '000000', 1)
''')

conn.commit()
conn.close()
```

Since we have inserted already one record into each table, we can query the database:

In [23]:

```
import sqlite3
conn = sqlite3.connect('example.db')

c = conn.cursor()
c.execute('SELECT * FROM person')
print(c.fetchall())
c.execute('SELECT * FROM address')
print(c.fetchall())
conn.close()
```

```
[(1, 'pythoncentral')]
[(1, 'python road', '1', '000000', 1)]
```

Similar query, but with relations:

In [24]:

```
import sqlite3
conn = sqlite3.connect('example.db')

c = conn.cursor()
c.execute('SELECT * FROM person, address WHERE person.id = address.person_id ')
print(c.fetchall())
conn.close()
```

```
[(1, 'pythoncentral', 1, 'python road', '1', '00000', 1)]
```

In order to use SQLAlchemy for that job, we have to map person and address tables into Python classes:

In [25]:

```
import os
import sys
from sqlalchemy import Column, ForeignKey, Integer, String
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import relationship
from sqlalchemy import create_engine

Base = declarative_base()

class Person(Base):
    __tablename__ = 'person'
    # columns of 'person' table
    id = Column(Integer, primary_key=True)
    name = Column(String(250), nullable=False)

class Address(Base):
    __tablename__ = 'address'
    # columns of 'address' table
    id = Column(Integer, primary_key=True)
    street_name = Column(String(250))
    street_number = Column(String(250))
    post_code = Column(String(250), nullable=False)
    person_id = Column(Integer, ForeignKey('person.id'))
    person = relationship(Person)

# connect to database
engine = create_engine('sqlite:///sqlalchemy_example.db')

# create tables
Base.metadata.create_all(engine)
```

In this way, a new empty sqlite3 database called `sqlalchemy_example.db` has been created. Since the database is empty, let us write some code to insert records into it:



In [26]:

```
from sqlalchemy.orm import sessionmaker

engine = create_engine('sqlite:///sqlalchemy_example.db')
# Bind the engine to the metadata of the Base class so that the
# declaratives can be accessed through a DBSession instance
Base.metadata.bind = engine

DBSession = sessionmaker(bind=engine) #all communication with the database
session = DBSession()

# insert a person in the `person` table
new_person = Person(name='new person')
session.add(new_person)
session.commit()

# insert an address
new_address = Address(post_code='00000', person=new_person)
session.add(new_address)
session.commit()
```

Now, we can check the content of the database:

In [27]:

```
session.query(Person).all()
```

Out[27]:

```
[<__main__.Person at 0x7fcf200b7d68>]
```

In [28]:

```
person = session.query(Person).first()
```

In [29]:

```
person.name
```

Out[29]:

```
'new person'
```

In [30]:

```
person.id
```

Out[30]:

```
1
```

In [31]:

```
session.query(Address).filter(Address.person == person).all()
```

Out[31]:

```
[<__main__.Address at 0x7fcf200b1080>]
```

In [32]:

```
address = session.query(Address).filter(Address.person == person).one()
```

In [33]:

```
address.post_code
```

Out[33]:

```
'00000'
```

## Case study - SQLite and pandas

We are going to use a subset of 311 service requests from NYC Open Data again. But this time a larger subset of all requests from 2010 till March 2016. The database is in CSV format. Let us check its size:

In [34]:

```
! ls -lh ~/Data/*.csv  
  
-rwxrwxrwx 1 szwabin szwabin 6,5G mar 25 2016 /home/szwabin/Data/311_Service_Requests_from_2010_to_Present.csv
```

The dataset is too large to load into a pandas dataframe! So, instead we'll perform out-of-memory aggregations with SQLite and load the result directly into a dataframe with Panda's iotools.

## Converting data from CSV file to SQLite database

Our first task is to stream the data from a CSV into SQLite. We can use pandas to complete the task. We divide it into following steps:

- load the CSV, chunk-by-chunk, into a DataFrame
- process the data a bit, strip out uninteresting columns
- append it to the SQLite database

Let us have a look at the data first:

In [35]:

```
!wc -l < ~/Data/311_Service_Requests_from_2010_to_Present.csv #number of lines  
10971582
```

In [36]:

```
import pandas as pd
```

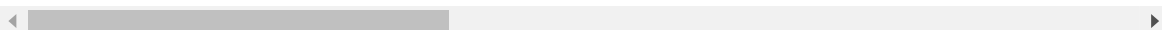
In [37]:

```
pd.read_csv('~Data/311_Service_Requests_from_2010_to_Present.csv', nrows=2).head()
```

Out[37]:

	Unique Key	Created Date	Closed Date	Agency	Agency Name	Complaint Type	Descriptor	Location
0	32950265	03/21/2016 02:08:09 AM	NaN	DHS	Operations Unit - Department of Homeless Services	Homeless Person Assistance	NaN	Street/...
1	32950917	03/21/2016 02:05:03 AM	NaN	NYPD	New York City Police Department	Illegal Parking	Commercial Overnight Parking	Street/...

2 rows × 53 columns



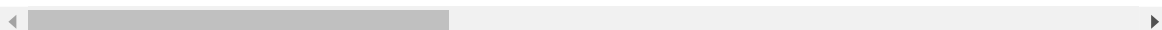
In [38]:

```
pd.read_csv('~Data/311_Service_Requests_from_2010_to_Present.csv', nrows=2).tail()
```

Out[38]:

	Unique Key	Created Date	Closed Date	Agency	Agency Name	Complaint Type	Descriptor	Location
0	32950265	03/21/2016 02:08:09 AM	NaN	DHS	Operations Unit - Department of Homeless Services	Homeless Person Assistance	NaN	Street/...
1	32950917	03/21/2016 02:05:03 AM	NaN	NYPD	New York City Police Department	Illegal Parking	Commercial Overnight Parking	Street/...

2 rows × 53 columns



We can use SQLAlchemy to establish a connection to the database (this time we use it simply as an engine, without the mapping to objects):

In [39]:

```
from sqlalchemy import create_engine  
import datetime as dt
```

First, we initialize the sqlite database with the filename 311.db:

In [40]:

```
disk_engine = create_engine('sqlite:///home/szwabin/Data/311.db')
```

Then we read the data chunk by chunk from the CSV file, remove spaces from the column names, convert dates to datetime format, change the first index to 1, select some interesting columns and save the result to the sqlite database:

In [41]:

```
start = dt.datetime.now()
chunksize = 200000
j = 0
index_start = 1

for df in pd.read_csv('~/.Data/311_Service_Requests_from_2010_to_Present.csv', chunksize=chunksize, low_memory=False, iterator=True, encoding='utf-8'):

    df = df.rename(columns={c: c.replace(' ', '') for c in df.columns}) # remove spaces

    df['CreatedDate'] = pd.to_datetime(df['CreatedDate']) # convert to datetime
    df['ClosedDate'] = pd.to_datetime(df['ClosedDate']) #

    df.index += index_start # start counting from 1, like humans usually do

    # Select the most interesting columns
    columns = ['Agency', 'CreatedDate', 'ClosedDate', 'ComplaintType', 'Descriptor',
               'CreatedDate', 'ClosedDate', 'TimeToCompletion',
               'City']

    for c in df.columns:
        if c not in columns:
            df = df.drop(c, axis=1)

    j+=1
    print('{} seconds: completed {} rows'.format((dt.datetime.now() - start).seconds, j*chunksize))

    df.to_sql('data', disk_engine, if_exists='append') #write to database
    index_start = df.index[-1] + 1
```

61 seconds: completed 200000 rows  
129 seconds: completed 400000 rows  
198 seconds: completed 600000 rows  
268 seconds: completed 800000 rows  
337 seconds: completed 1000000 rows  
407 seconds: completed 1200000 rows  
477 seconds: completed 1400000 rows  
547 seconds: completed 1600000 rows  
618 seconds: completed 1800000 rows  
689 seconds: completed 2000000 rows  
759 seconds: completed 2200000 rows  
830 seconds: completed 2400000 rows  
899 seconds: completed 2600000 rows  
970 seconds: completed 2800000 rows  
1041 seconds: completed 3000000 rows  
1113 seconds: completed 3200000 rows  
1181 seconds: completed 3400000 rows  
1252 seconds: completed 3600000 rows  
1322 seconds: completed 3800000 rows  
1392 seconds: completed 4000000 rows  
1463 seconds: completed 4200000 rows  
1534 seconds: completed 4400000 rows  
1606 seconds: completed 4600000 rows  
1676 seconds: completed 4800000 rows  
1745 seconds: completed 5000000 rows  
1813 seconds: completed 5200000 rows  
1883 seconds: completed 5400000 rows  
1953 seconds: completed 5600000 rows  
2023 seconds: completed 5800000 rows  
2094 seconds: completed 6000000 rows  
2165 seconds: completed 6200000 rows  
2234 seconds: completed 6400000 rows  
2303 seconds: completed 6600000 rows  
2375 seconds: completed 6800000 rows  
2444 seconds: completed 7000000 rows  
2514 seconds: completed 7200000 rows  
2584 seconds: completed 7400000 rows  
2654 seconds: completed 7600000 rows  
2725 seconds: completed 7800000 rows  
2794 seconds: completed 8000000 rows  
2864 seconds: completed 8200000 rows  
2934 seconds: completed 8400000 rows  
2999 seconds: completed 8600000 rows  
3062 seconds: completed 8800000 rows  
3128 seconds: completed 9000000 rows  
3197 seconds: completed 9200000 rows  
3269 seconds: completed 9400000 rows  
3336 seconds: completed 9600000 rows  
3405 seconds: completed 9800000 rows  
3476 seconds: completed 10000000 rows  
3544 seconds: completed 10200000 rows  
3612 seconds: completed 10400000 rows  
3682 seconds: completed 10600000 rows  
3751 seconds: completed 10800000 rows  
3810 seconds: completed 11000000 rows

Let us have a look at the database:

In [42]:

```
df = pd.read_sql_query('SELECT * FROM data LIMIT 3', disk_engine)
df.head()
```

Out[42]:

	index	CreatedDate	ClosedDate	Agency	ComplaintType	Descriptor	
0	1	2016-03-21 02:08:09.000000	None	DHS	Homeless Person Assistance	None	NEW YO
1	2	2016-03-21 02:05:03.000000	None	NYPD	Illegal Parking	Commercial Overnight Parking	RIDGEW
2	3	2016-03-21 02:03:38.000000	None	NYPD	Illegal Parking	Blocked Hydrant	BROOKL

Select multiple columns:

In [43]:

```
df = pd.read_sql_query('SELECT Agency, Descriptor FROM data LIMIT 3', disk_engine)
df.head()
```

Out[43]:

	Agency	Descriptor
0	DHS	None
1	NYPD	Commercial Overnight Parking
2	NYPD	Blocked Hydrant

In [44]:

```
df = pd.read_sql_query('SELECT ComplaintType, Descriptor, Agency '
                        'FROM data '
                        'LIMIT 10', disk_engine)
df
```

Out[44]:

	ComplaintType	Descriptor	Agency
0	Homeless Person Assistance	None	DHS
1	Illegal Parking	Commercial Overnight Parking	NYPD
2	Illegal Parking	Blocked Hydrant	NYPD
3	Noise - Street/Sidewalk	Loud Music/Party	NYPD
4	Noise - Vehicle	Car/Truck Music	NYPD
5	Blocked Driveway	No Access	NYPD
6	Damaged Tree	Branch Cracked and Will Fall	DPR
7	Fire Safety Director - F58	On Site Test	FDNY
8	Animal Abuse	Neglected	NYPD
9	New Tree Request	For One Address	DPR

Filter rows with WHERE:

In [45]:

```
df = pd.read_sql_query('SELECT ComplaintType, Descriptor, Agency '
                        'FROM data '
                        'WHERE Agency = "NYPD" '
                        'LIMIT 10', disk_engine)
df.head()
```

Out[45]:

	ComplaintType	Descriptor	Agency
0	Illegal Parking	Commercial Overnight Parking	NYPD
1	Illegal Parking	Blocked Hydrant	NYPD
2	Noise - Street/Sidewalk	Loud Music/Party	NYPD
3	Noise - Vehicle	Car/Truck Music	NYPD
4	Blocked Driveway	No Access	NYPD

Filter multiple values with WHERE and IN:



In [46]:

```
df = pd.read_sql_query('SELECT ComplaintType, Descriptor, Agency '
                        'FROM data '
                        'WHERE Agency IN ("NYPD", "DOB") '
                        'LIMIT 10', disk_engine)
df.head()
```

Out[46]:

	ComplaintType	Descriptor	Agency
0	Illegal Parking	Commercial Overnight Parking	NYPD
1	Illegal Parking	Blocked Hydrant	NYPD
2	Noise - Street/Sidewalk	Loud Music/Party	NYPD
3	Noise - Vehicle	Car/Truck Music	NYPD
4	Blocked Driveway	No Access	NYPD

Find the unique values in a column with DISTINCT:

In [47]:

```
df = pd.read_sql_query('SELECT DISTINCT City FROM data', disk_engine)
df.head()
```

Out[47]:

	City
0	NEW YORK
1	RIDGEWOOD
2	BROOKLYN
3	STATEN ISLAND
4	BRONX

Count the number of rows:

In [48]:

```
df = pd.read_sql_query('SELECT Agency, COUNT(*) as `num_complaints`  
                        'FROM data '  
                        'GROUP BY Agency ', disk_engine)  
  
df.head()
```

Out[48]:

	Agency	num_complaints
0	3-1-1	23091
1	ACS	6
2	AJC	8
3	CAU	10
4	CCRB	10

Order the results with ORDER BY:

In [49]:

```
df = pd.read_sql_query('SELECT Agency, COUNT(*) as `num_complaints`  
                        'FROM data '  
                        'GROUP BY Agency '  
                        'ORDER BY -num_complaints', disk_engine)  
  
df.head()
```

Out[49]:

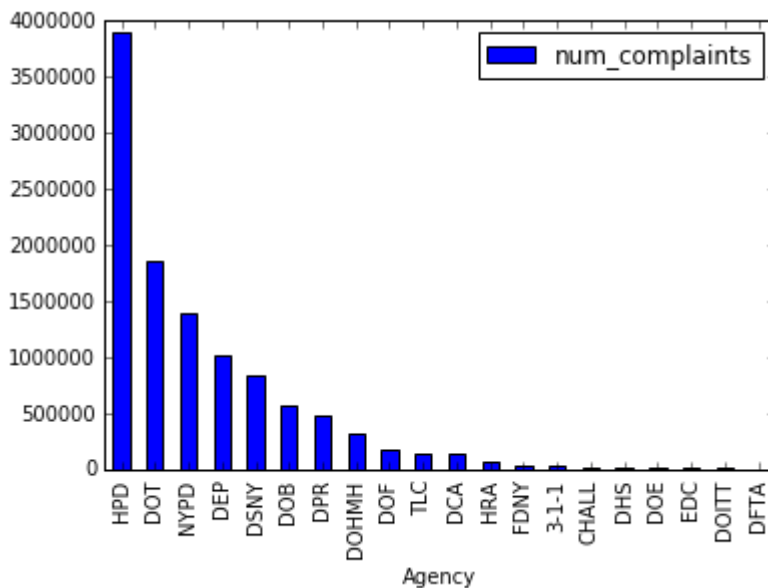
	Agency	num_complaints
0	HPD	3894650
1	DOT	1851584
2	NYPD	1386943
3	DEP	1017030
4	DSNY	830939

In [52]:

```
df[:20].plot(kind='bar',x='Agency')
```

Out[52]:

<matplotlib.axes.\_subplots.AxesSubplot at 0x7fceebb43780>



### Most common complaint

In [53]:

```
df = pd.read_sql_query('SELECT ComplaintType, COUNT(*) as `num_complaints`, Agency
                        FROM data
                        GROUP BY `ComplaintType`
                        ORDER BY -num_complaints', disk_engine)

most_common_complaints = df # will be used later

df.head()
```

Out[53]:

	ComplaintType	num_complaints	Agency
0	HEATING	887869	HPD
1	Street Light Condition	618087	DOT
2	Street Condition	612937	DOT
3	PLUMBING	530650	HPD
4	GENERAL CONSTRUCTION	500867	HPD

### The most common complaint in each city

Let us see first how many cities are recorded in the dataset:

In [54]:

```
len(pd.read_sql_query('SELECT DISTINCT City FROM data', disk_engine))
```

Out[54]:

1945

Cities with most complaints (top 10):

In [55]:

```
df = pd.read_sql_query('SELECT City, COUNT(*) as `num_complaints` '
                        'FROM data '
                        'GROUP BY `City` '
                        'ORDER BY -num_complaints '
                        'LIMIT 10 ', disk_engine)
df
```

Out[55]:

	City	num_complaints
0	BROOKLYN	3229867
1	NEW YORK	2075756
2	BRONX	1946605
3	None	849204
4	STATEN ISLAND	526233
5	JAMAICA	162533
6	FLUSHING	131220
7	ASTORIA	101470
8	Jamaica	96719
9	RIDGEWOOD	76054

Case insensitive queries with COLLATE NOCASE:

In [71]:

```
df = pd.read_sql_query('SELECT City, COUNT(*) as `num_complaints` '
                        'FROM data '
                        'GROUP BY `City` '
                        'COLLATE NOCASE '
                        'ORDER BY -num_complaints '
                        'LIMIT 11 ', disk_engine)
```

Let us check the list of the cities:

In [72]:

```
cities = list(df.City)
cities.remove(None)
print(cities)
```

```
['BROOKLYN', 'NEW YORK', 'BRONX', 'STATEN ISLAND', 'JAMAICA', 'FLUSH
ING', 'ASTORIA', 'RIDGEWOOD', 'CORONA', 'WOODSIDE']
```

For every city from the above list create a dataframe:

In [73]:

```
city = cities[0]
df = pd.read_sql_query('SELECT ComplaintType, COUNT(*) as `num_complaints` '
                      'FROM data '
                      'WHERE City = "{}" COLLATE NOCASE '
                      'GROUP BY `ComplaintType` '
                      'ORDER BY -num_complaints'.format(city), disk_engine)
df.columns = ['ComplaintType', city]

df.head()
```

Out[73]:

	ComplaintType	BROOKLYN
0	HEATING	175181
1	HEAT/HOT WATER	136422
2	PLUMBING	132681
3	Blocked Driveway	124735
4	Street Condition	118186

In [74]:

```
df2 = df.copy()
```

In [75]:

```
for city in cities[1:]:
    df = pd.read_sql_query('SELECT ComplaintType, COUNT(*) as `num_complaints` '
                          'FROM data '
                          'WHERE City = "{}" COLLATE NOCASE '
                          'GROUP BY `ComplaintType` '
                          'ORDER BY -num_complaints'.format(city), disk_engine)
    df.columns = ['ComplaintType', city]
    df2 = pd.merge(df2, df, on='ComplaintType')
```

In [76]:

df2.head()

Out[76]:

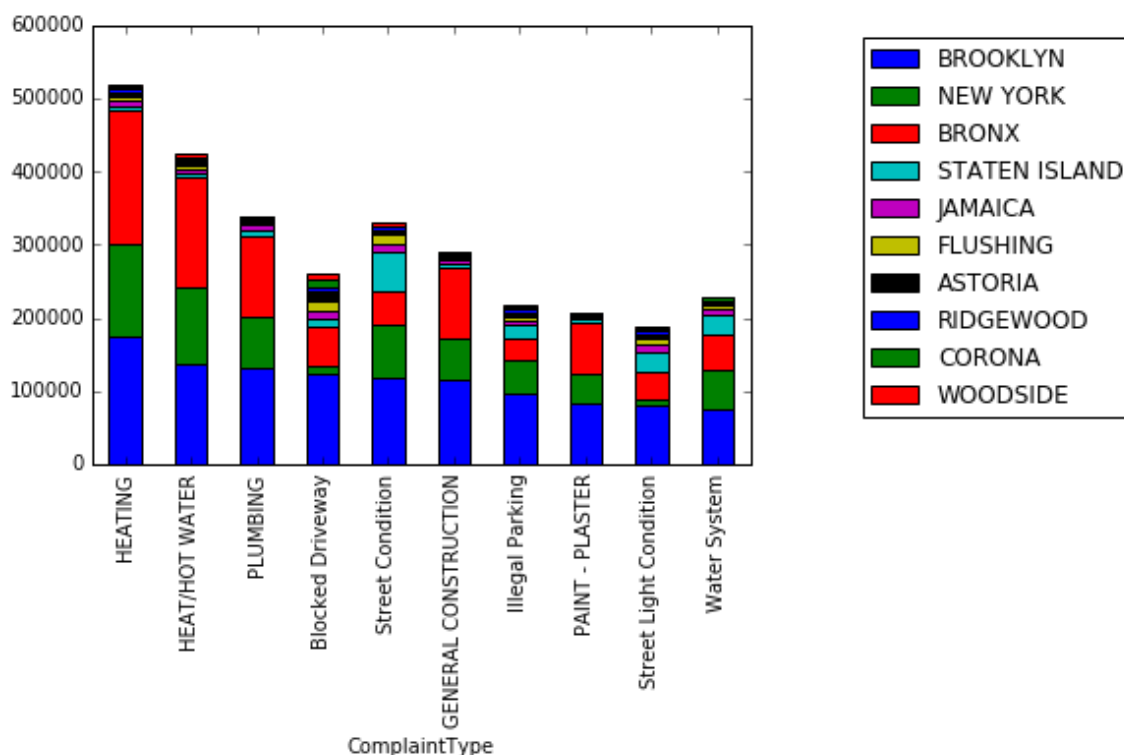
	ComplaintType	BROOKLYN	NEW YORK	BRONX	STATEN ISLAND	JAMAICA	FLUSHING	AST
0	HEATING	175181	126803	180983	5507	7860	6325	6416
1	HEAT/HOT WATER	136422	106256	150353	4344	6093	5503	5932
2	PLUMBING	132681	68934	110682	7885	7207	3303	2956
3	Blocked Driveway	124735	9885	53069	10082	12507	12545	1261
4	Street Condition	118186	73196	44964	54992	10872	11398	5380

In [77]:

```
df2[:10].plot(kind='bar', x='ComplaintType', stacked=True, label=city).legend(bbox_x_to_anchor=(1.6,1.0))
```

Out[77]:

&lt;matplotlib.legend.Legend at 0x7fd3c5bc57f0&gt;



### Number of complaints by hour

Filter rows with timestamp strings:

In [78]:

```
df = pd.read_sql_query('SELECT ComplaintType, CreatedDate, City '
                      'FROM data '
                      'WHERE CreatedDate < "2014-11-16 23:47:00" '
                      'AND CreatedDate > "2014-11-16 23:45:00"', disk_engine)

df
```

Out[78]:

	ComplaintType	CreatedDate	City
0	Derelict Vehicle	2014-11-16 23:46:57.000000	RIDGEWOOD
1	Noise - Commercial	2014-11-16 23:46:08.000000	ASTORIA
2	Derelict Vehicles	2014-11-16 23:46:00.000000	Jamaica
3	Benefit Card Replacement	2014-11-16 23:45:55.000000	None
4	Blocked Driveway	2014-11-16 23:45:43.000000	BROOKLYN
5	School Maintenance	2014-11-16 23:45:41.000000	BRONX
6	Noise - Street/Sidewalk	2014-11-16 23:45:10.000000	NEW YORK

Extract the hour from the timestamp (strftime):

In [79]:

```
df = pd.read_sql_query('SELECT CreatedDate, '
                      'strftime(\'%H\', CreatedDate) as hour, '
                      'ComplaintType '
                      'FROM data '
                      'LIMIT 5 ', disk_engine)

df.head()
```

Out[79]:

	CreatedDate	hour	ComplaintType
0	2016-03-21 02:08:09.000000	02	Homeless Person Assistance
1	2016-03-21 02:05:03.000000	02	Illegal Parking
2	2016-03-21 02:03:38.000000	02	Illegal Parking
3	2016-03-21 02:03:29.000000	02	Noise - Street/Sidewalk
4	2016-03-21 02:02:33.000000	02	Noise - Vehicle

Count the number of complaint by hour:

In [80]:

```
df = pd.read_sql_query('SELECT CreatedDate, '
                        'strftime(\'%H\', CreatedDate) as hour, '
                        'count(*) as `Complaints per Hour`'
                        'FROM data '
                        'GROUP BY hour', disk_engine)

df.head()
```

Out[80]:

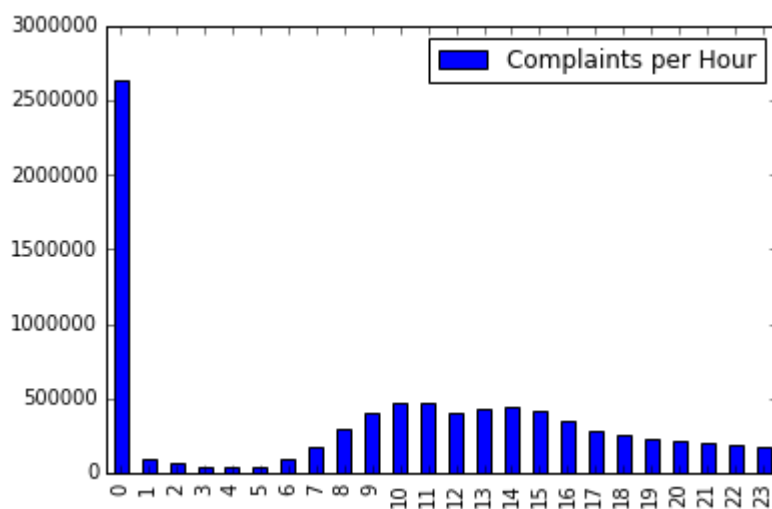
	CreatedDate	hour	Complaints per Hour
0	2011-06-27 00:00:00.000000	00	2635837
1	2011-06-27 01:00:22.000000	01	89689
2	2011-06-27 02:00:00.000000	02	63554
3	2011-06-27 03:00:40.000000	03	40602
4	2011-06-27 04:00:26.000000	04	37244

In [81]:

```
df.plot(kind="bar")
```

Out[81]:

<matplotlib.axes.\_subplots.AxesSubplot at 0x7fd3c75a0470>



Noise complaints



In [82]:

```
df = pd.read_sql_query('SELECT CreatedDate, '
                        'strftime(\'%H\', CreatedDate) as `hour`, '
                        'count(*) as `Complaints per Hour` '
                        'FROM data '
                        'WHERE ComplaintType IN ("Noise", '
                                                '"Noise - Street/Sidewalk", '
                                                '"Noise - Commercial", '
                                                '"Noise - Vehicle", '
                                                '"Noise - Park", '
                                                '"Noise - House of Worship", '
                                                '"Noise - Helicopter", '
                                                '"Collection Truck Noise") '
                        'GROUP BY hour', disk_engine)
```

In [83]:

```
df.head()
```

Out[83]:

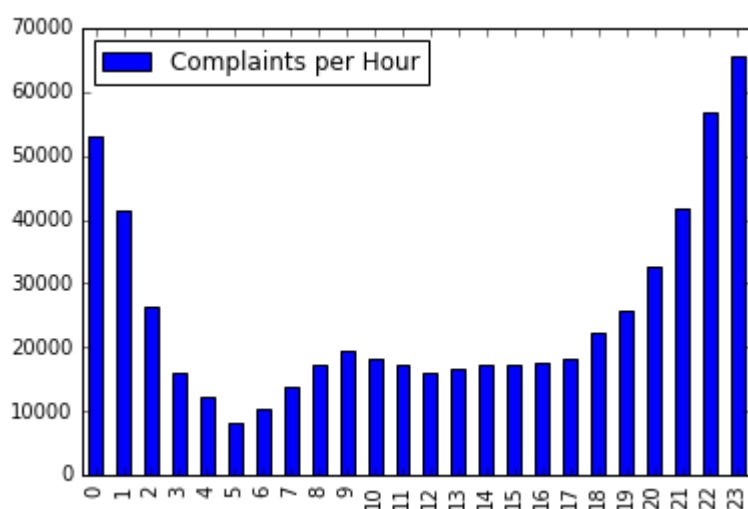
	CreatedDate	hour	Complaints per Hour
0	2011-06-27 00:03:34.000000	00	52971
1	2011-06-27 01:23:53.000000	01	41386
2	2011-06-27 02:11:00.000000	02	26501
3	2011-06-27 03:06:50.000000	03	15967
4	2011-06-27 04:00:26.000000	04	12342

In [84]:

```
df.plot(kind='bar')
```

Out[84]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fd3c9360e80>
```



## NoSQL databases

- non-relational databases
- for data that is modeled in means other than the tabular relations
- very often no predefined structure of data
- usually best choice for big data
- features:
  - not always ACID compliant
  - scalability
  - objects of different types and structures
  - map-reduce for data aggregation
  - support for query languages
  - limited support for transactions
- types:
  - **key\_value** - the simplest NoSQL data stores to use from an API perspective. The client can either get the value for the key, put a value for a key, or delete a key from the data store. They generally have great performance and can be easily scaled
  - **column family store** - data is stored in column families as rows that have many columns associated with a row key. Column families are groups of related data that is often accessed together
  - **document store** - the database stores and retrieves documents, which can be XML, JSON, BSON, and so on. These documents are self-describing, hierarchical tree data structures which can consist of maps, collections, and scalar values. The documents stored are similar to each other but do not have to be exactly the same
  - **graph databases** - allow to store entities and relationships between these entities. Entities are also known as nodes, which have properties. Relations are known as edges that can have properties. Edges have directional significance
  - **object databases** - data is stored in objects
  - **multi-model store** - hybrid solutions
- examples
  - Redis (key-value, <http://redis.io/> (<http://redis.io/>))
  - CouchDB (document, <http://couchdb.apache.org/> (<http://couchdb.apache.org/>))
  - MongoDB (document, <https://www.mongodb.org/> (<https://www.mongodb.org/>))
  - IBM Domino (document, <http://www-03.ibm.com/software/products/pl/ibmdomino> (<http://www-03.ibm.com/software/products/pl/ibmdomino>))
  - Cassandra (column family, <http://cassandra.apache.org/> (<http://cassandra.apache.org/>))
  - Hypertable (column family, <http://www.hypertable.com/> (<http://www.hypertable.com/>))
  - Hadoop/HBase (column family, <https://hbase.apache.org/> (<https://hbase.apache.org/>))
  - Neo4j (graph, <http://neo4j.com/> (<http://neo4j.com/>))
  - db4o (object, <https://sourceforge.net/projects/db4o/> (<https://sourceforge.net/projects/db4o/>))
  - EyeDB (object, <http://www.eyedb.org/> (<http://www.eyedb.org/>))
  - OrientDB (hybrid, <http://orientdb.com/orientdb/> (<http://orientdb.com/orientdb/>))

## CouchDB

- document-oriented NoSQL database
- implemented in Erlang language
- REST API (POST, GET, PUT, and DELETE methods from HTTP)
- JSON format
- JavaScript as query language
- queries in MapReduce form
- adding other query languages including Python possible
- ACID semantics
- advanced replication and synchronization of data
- Fauxton (formerly Futon) - a web-based application for administration

### First steps

Installation in Ubuntu is easy:

```
sudo apt-get install couchdb
```

After installation, the database system in the default setting will be available at <http://127.0.0.1:5984/>

We may use curl to work with CouchDB directly from CLI:

In [88]:

```
!curl http://127.0.0.1:5984/
```

```
{"couchdb": "Welcome", "uuid": "38de869a5c71966079cba2bfee28a492", "version": "1.6.0", "vendor": {"name": "Ubuntu", "version": "15.10"}}
```

First, we check existing databases:

In [89]:

```
!curl -X GET http://127.0.0.1:5984/_all_dbs
```

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

We add a new one:

In [95]:

```
!curl -X PUT szwabin:analiza@localhost:5984/new_database
```

```
{"ok": true}
```

In [96]:

```
!curl -X GET http://127.0.0.1:5984/_all_dbs
```

```
["_replicator", "_users", "demodb", "new_database"]
```

## Futon

- built-in administration interface
- accessible at [http://127.0.0.1:5984/\\_utils/](http://127.0.0.1:5984/_utils/) ([http://127.0.0.1:5984/\\_utils/](http://127.0.0.1:5984/_utils/))

The screenshot shows the Apache CouchDB Futon interface in a web browser. The address bar displays `127.0.0.1:5984/_utils/`. The main content area is titled "Overview" and features a "Create Database ..." button. Below this is a table listing existing databases:

Name	Size	Number of Documents	Update Seq
<code>_replicator</code>	4.1 KB	1	1
<code>_users</code>	4.1 KB	1	1
<code>new_database</code>	79 bytes	0	0

Below the table, it indicates "Showing 1-3 of 3 databases" and provides navigation links for "Previous Page", "Rows per page: 10", and "Next Page".

The right sidebar contains a CouchDB logo with the text "CouchDB relax" and a "Tools" menu with links to Overview, Configuration, Replicator, and Status. Below this is a "Documentation" section with a link to the Manual, and a "Diagnostics" section with a link to Verify Installation. At the bottom of the sidebar, it says "Recent Databases".

At the very bottom of the interface, a message reads: "Welcome to Admin Party! Everyone is admin. [Fix this](#)". The footer indicates "Futon on Apache CouchDB 1.6.0".

Creating a new database::

The "Create New Database" dialog box prompts the user to enter a database name. It includes the following text: "Please enter the name of the database. Note that only lowercase characters (a-z), digits (0-9), or any of the characters `_`, `$`, `(`, `)`, `+`, `-`, and `/` are allowed."

Below the text is a text input field labeled "Database Name:". At the bottom right of the dialog are two buttons: "Create" and "Cancel".

View of the new database:

The screenshot shows the Apache CouchDB Futon web interface in a browser window. The address bar displays the URL `127.0.0.1:5984/_utils/database.html?demodb`. The browser's bookmark bar shows folders like 'Aplikacje', 'Prywatne', 'Służbowe', 'Dydaktyka', 'Projekty', and 'Complex Global'. The interface has a dark header with 'Overview' and 'demodb' tabs. Below the header, there are controls for 'Jump to: [Document ID]', 'View: All documents', and 'Stale views'. Action buttons include '+ New Document', 'Security...', 'Compact & Cleanup...', and 'Delete Database...'. A table with columns 'Key' and 'Value' is shown, indicating 'Showing 0--1 of 0 rows'. The right sidebar contains the CouchDB logo, navigation links for Tools (Overview, Configuration, Replicator, Status), Documentation (Manual), Diagnostics (Verify Installation), and Recent Databases (listing '\_users' and 'demodb'). At the bottom right, a welcome message for 'szwabin!' is displayed with links to 'Setup more admins or Change password or Logout', and the footer reads 'Futon on Apache CouchDB 1.6.0'.

Inserting New Document:

The screenshot shows the Apache CouchDB Futon web interface. The browser address bar displays `127.0.0.1:5984/_utils/document.html?demodb`. The breadcrumb navigation shows `Overview > demodb > e9dc8c71913c5d4c82695a927800041e`. The main content area features a document editor with a table of fields:

Field	Value	
<input checked="" type="checkbox"/> <code>_id</code>	<input type="text" value="e9dc8c71913c5d4c82695a927800041e"/>	<input checked="" type="checkbox"/> <input type="checkbox"/>
<input checked="" type="checkbox"/> <code>task</code>	<input type="text" value="Wykład z analizy danych"/>	<input checked="" type="checkbox"/> <input type="checkbox"/>
<input checked="" type="checkbox"/> <code>done</code>	<input type="text" value="False"/>	<input checked="" type="checkbox"/> <input type="checkbox"/>

Below the table are links for `← Previous Version` and `Next Version →`. The right sidebar contains the CouchDB logo, navigation links (Tools, Documentation, Diagnostics), and a list of recent databases including `_users` and `demodb`. At the bottom, a welcome message for user `szwabin!` is displayed.

The document is automatically added to the view:

The screenshot shows the Apache CouchDB Futon web interface in a browser window. The address bar shows the URL `127.0.0.1:5984/_utils/database.html?demodb`. The page title is "Apache CouchDB - Futon". The main content area displays the "demodb" database overview. At the top, there are controls for "Jump to: [Document ID]", "View: All documents", and "Stale views". Below these are buttons for "New Document", "Security...", "Compact & Cleanup...", and "Delete Database...". A table shows the document details:

Key	Value
"e9dc8c71913c5d4c82695a927800041e"	{rev: "2-d52168193dd61f7cb60667e4f0fe88b6"}

Below the table, it says "Showing 1-1 of 1 row" and "Rows per page: 10". On the right side, there is a sidebar with the CouchDB logo and "relax" text. The sidebar contains links for "Tools" (Overview, Configuration, Replicator, Status), "Documentation" (Manual), "Diagnostics" (Verify Installation), and "Recent Databases" (listing "\_users" and "demodb"). At the bottom of the sidebar, there is a welcome message: "Welcome szwabin! Setup more admins or Change password or Logout" and "Futon on Apache CouchDB 1.6.0".

Now the document can be read, edited or deleted.

## Views

Let us first add some new documents to the database:

In [97]:

```
!curl -X POST -d '{"task":"task 1", "done":"False"}' http://localhost:5984/demodb -H "Content-Type:application/json"
```

```
{"ok":true,"id":"63c531a07440d16ae3d3ef96ef000a94","rev":"1-7562778599d4a5f29b72e4c2149290a5"}
```

In [98]:

```
!curl -X POST -d '{"task":"task 2", "done":"True"}' http://localhost:5984/demodb -H "Content-Type:application/json"
```

```
{"ok":true,"id":"63c531a07440d16ae3d3ef96ef00108c","rev":"1-0d3e347f634f5a809842ed4cad65de8a"}
```

In [99]:

```
!curl -X POST -d '{"task":"task 3", "done":"True"}' http://localhost:5984/demodb  
-H "Content-Type:application/json"
```

```
{"ok":true,"id":"63c531a07440d16ae3d3ef96ef001453","rev":"1-61cad97d  
c4f61f47297071c96bca0f42"}
```

In [100]:

```
!curl -X POST -d '{"task":"task 4", "done":"True"}' http://localhost:5984/demodb  
-H "Content-Type:application/json"
```

```
{"ok":true,"id":"63c531a07440d16ae3d3ef96ef001e02","rev":"1-571e099b  
7d5ae990f2e8d02d20a97cba"}
```



Check the content of the database:

The screenshot shows the Apache CouchDB Futon interface for a database named 'demodb'. The browser address bar shows the URL '127.0.0.1:5984/\_utils/database.html?demodb'. The interface includes a navigation bar with 'Overview' and 'demodb'. Below the navigation bar, there is a 'Jump to' field for Document ID, a 'View' dropdown set to 'All documents', and a 'Stale views' checkbox. The main content area displays a table with 5 rows of documents. The table has two columns: 'Key' and 'Value'. The 'Key' column shows document keys and IDs, and the 'Value' column shows document values. The right sidebar contains links for Tools, Documentation, Diagnostics, and Recent Databases. The bottom of the sidebar shows a welcome message for 'szwabin!' and a footer indicating 'Futon on Apache CouchDB 1.6.0'.

Key	Value
"e9dc8c71913c5d4c82695a927800041e" ID: e9dc8c71913c5d4c82695a927800041e	{rev: "2-d52168193dd61f7cb60667e4f0fe88b6"}
"e9dc8c71913c5d4c82695a92780007ef" ID: e9dc8c71913c5d4c82695a92780007ef	{rev: "1-7562778599d4a5f29b72e4c2149290a5"}
"e9dc8c71913c5d4c82695a9278000d05" ID: e9dc8c71913c5d4c82695a9278000d05	{rev: "1-0d3e347f634f5a809842ed4cad65de8a"}
"e9dc8c71913c5d4c82695a9278000d0e" ID: e9dc8c71913c5d4c82695a9278000d0e	{rev: "1-61cad97dc4f61f47297071c96bca0f42"}
"e9dc8c71913c5d4c82695a9278001615" ID: e9dc8c71913c5d4c82695a9278001615	{rev: "1-571e099b7d5ae990f2e8d02d20a97cba"}

Showing 1-5 of 5 rows    ← Previous Page    Rows per page: 10    Next Page →

Tools  
Overview  
Configuration  
Replicator  
Status

Documentation  
Manual

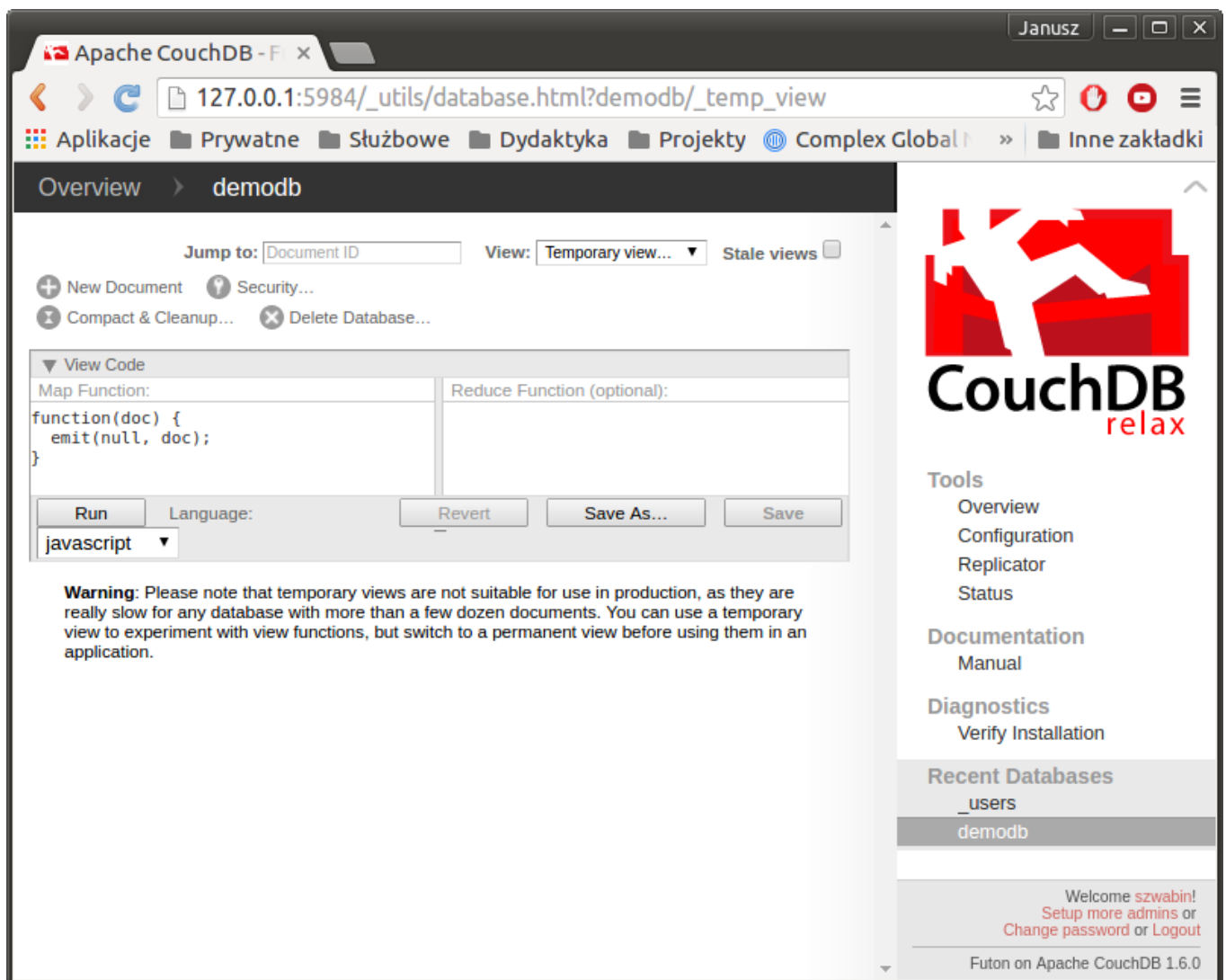
Diagnostics  
Verify Installation

Recent Databases  
\_users  
demodb

Welcome szwabin!  
Setup more admins or  
Change password or Logout

Futon on Apache CouchDB 1.6.0

Our goal is know to find all tasks with the value False in the field done. To this end we select Temporary view... in the View list:



The screenshot shows the Apache CouchDB Futon web interface in a browser window. The address bar shows the URL `127.0.0.1:5984/_utils/database.html?demodb/_temp_view`. The page title is "Apache CouchDB - Futon". The interface is for the "demodb" database. At the top, there are buttons for "New Document", "Security...", "Compact & Cleanup...", and "Delete Database...". Below these, there is a "View Code" section with a "Map Function" field containing the following JavaScript code:

```
function(doc) {  
  emit(null, doc);  
}
```

Below the code field, there are buttons for "Run", "Language:" (set to "javascript"), "Revert", "Save As...", and "Save". A warning message is displayed below the code editor:

**Warning:** Please note that temporary views are not suitable for use in production, as they are really slow for any database with more than a few dozen documents. You can use a temporary view to experiment with view functions, but switch to a permanent view before using them in an application.

On the right side, there is a sidebar with the CouchDB logo and the word "relax". Below the logo, there are links for "Tools" (Overview, Configuration, Replicator, Status), "Documentation" (Manual), "Diagnostics" (Verify Installation), and "Recent Databases" (listing "\_users" and "demodb"). At the bottom of the sidebar, there is a welcome message for "szwabin!" and links for "Setup more admins or Change password or Logout". The footer of the page says "Futon on Apache CouchDB 1.6.0".

We change the Map function in the following way,

```
function(doc) {  
  if (doc.done === "False"){  
    emit(doc.task);  
  }  
}
```

and run it. In this simple example the Reduce function may be empty.

The screenshot shows the Apache CouchDB web interface in a browser. The address bar displays the URL `127.0.0.1:5984/_utils/database.html?demodb/_temp_view`. The page title is "Apache CouchDB - F". The interface includes a navigation bar with "Overview" and "demodb". Below this, there are controls for "Jump to: [Document ID]", "View: Temporary view...", and a "Stale views" checkbox. A sidebar on the right contains links for "Tools" (Overview, Configuration, Replicator, Status), "Documentation" (Manual), "Diagnostics" (Verify Installation), and "Recent Databases" (\_users, demodb). The main content area shows a "View Code" section with a Map Function and a Reduce Function (optional). The Map Function is a JavaScript function that emits `doc.task` if `doc.done` is not "False". Below the code editor are buttons for "Run", "Language" (set to "javascript"), "Revert", "Save As...", and "Save". A warning message states: "Warning: Please note that temporary views are not suitable for use in production, as they are really slow for any database with more than a few dozen documents. You can use a temporary view to experiment with view functions, but switch to a permanent view before using them in an application." Below the warning is a table with two columns: "Key" and "Value". The table contains two rows: one with key `"task 1"` and value `null`, and another with key `"Wykład z analizy danych"` and value `null`. The table is paginated, showing 1-2 of 2 rows. At the bottom, it says "View request duration: 00:00:00.015".

## CouchDB and Python

The `couchdb-python` module offers access to CouchDB databases from Python.

### First steps

In [101]:

```
import couchdb
```

First, we have to create a server object. If the server uses the default setting, no argument in the constructor is required:

In [102]:

```
couch = couchdb.Server()
print(couch)
```

```
<Server 'http://localhost:5984/'>
```

A new database can be created with the `create` method:

In [103]:

```
db = couch.create('python_test')
```

```

-----
-----
Unauthorized                                Traceback (most recent call last)
<ipython-input-103-74f1ba73401d> in <module>()
----> 1 db = couch.create('python_test')

/usr/local/lib/python3.5/dist-packages/couchdb/client.py in create(self, name)
    205         :raise PreconditionFailed: if a database with that name already exists
    206         """
--> 207         self.resource.put_json(name)
    208         return self[name]
    209

/usr/local/lib/python3.5/dist-packages/couchdb/http.py in put_json(self, path, body, headers, **params)
    568     def put_json(self, path=None, body=None, headers=None, **params):
    569         return self._request_json('PUT', path, body=body, headers=headers,
--> 570                                     **params)
    571
    572     def _request(self, method, path=None, body=None, headers=None, **params):

/usr/local/lib/python3.5/dist-packages/couchdb/http.py in _request_json(self, method, path, body, headers, **params)
    583     def _request_json(self, method, path=None, body=None, headers=None, **params):
    584         status, headers, data = self._request(method, path, body=body,
--> 585                                             headers=headers,
s, **params)
    586         if 'application/json' in headers.get('content-type', ''):
    587             data = json.decode(data.read().decode('utf-8'))

/usr/local/lib/python3.5/dist-packages/couchdb/http.py in _request(self, method, path, body, headers, **params)
    579         return self.session.request(method, url, body=body,
    580                                     headers=all_headers,
--> 581                                     credentials=self.credentials)
    582
    583     def _request_json(self, method, path=None, body=None, headers=None, **params):

/usr/local/lib/python3.5/dist-packages/couchdb/http.py in request(self, method, url, body, headers, credentials, num_redirects)
    411         error = ''
    412         if status == 401:
--> 413             raise Unauthorized(error)
    414         elif status == 404:
    415             raise ResourceNotFound(error)

Unauthorized: ('unauthorized', 'You are not a server admin.')

```

It does not work, because a system admin was already added to CouchDB. Let us try it once again:

In [104]:

```
couch = couchdb.Server('http://szwabin:analiza@localhost:5984')
db = couch.create('python_test2')
```

Check the content of the database:

In [105]:

```
for id in db:
    print(id)
```

Since the database is empty, there are no results. Let us add a couple of documents:

In [106]:

```
doc = {'foo' : 'bar'}
db.save(doc)
```

Out[106]:

```
('63c531a07440d16ae3d3ef96ef002aba', '1-4c6114c65e295552ab1019e2b046b10e')
```

In [107]:

```
doc = {'foo' : 'rab'}
db.save(doc)
```

Out[107]:

```
('63c531a07440d16ae3d3ef96ef0039ad', '1-35d2652fde877351da88b4424d07d87a')
```

Let us check the content again:

In [108]:

```
for id in db:
    print(id, db[id]['foo'])
```

```
63c531a07440d16ae3d3ef96ef002aba bar
63c531a07440d16ae3d3ef96ef0039ad rab
```

Deleting a database is easy:

In [109]:

```
couch.delete('python_test2')
```

We may access other databases with a dict style too:

In [110]:

```
db = couch['demodb']
```

In [111]:

```
for id in db:  
    print(id, db[id]['task'])
```

```
63c531a07440d16ae3d3ef96ef000a94 task 1  
63c531a07440d16ae3d3ef96ef00108c task 2  
63c531a07440d16ae3d3ef96ef001453 task 3  
63c531a07440d16ae3d3ef96ef001e02 task 4
```

## Document mapping

In [112]:

```
couch = couchdb.Server('http://szwabin:analiza@localhost:5984')  
db = couch['demodb']
```

In [113]:

```
from couchdb.mapping import Document, TextField
```

In [114]:

```
class Todo(Document):  
    task = TextField()  
    done = TextField()
```

In [115]:

```
newtask = Todo(task="Seminarium2",done="True")
```

In [116]:

```
newtask.store(db)
```

Out[116]:

```
<Todo '63c531a07440d16ae3d3ef96ef0041f6'@'1-48d03fd7587a2498ee48a46e  
c5307c60' {'done': 'True', 'task': 'Seminarium2'}>
```

In [117]:

```
for id in db:  
    print(id, db[id]['task'])
```

```
63c531a07440d16ae3d3ef96ef000a94 task 1  
63c531a07440d16ae3d3ef96ef00108c task 2  
63c531a07440d16ae3d3ef96ef001453 task 3  
63c531a07440d16ae3d3ef96ef001e02 task 4  
63c531a07440d16ae3d3ef96ef0041f6 Seminarium2
```



## Building new views in Python

The couchdb-python offers a view server as well, which allows to create views directly in Python. To this end, one has to add the following lines to the `/etc/couchdb/local.ini` file:

```
[query_servers]
python=/usr/local/bin/couchpy
```

After restarting CouchDB Python should be available in Futon as one of the languages.

**Warning!** On my computer Python indeed appeared on the list of the available languages. However, even the simplest example from the couchdb-python docs (<https://pythonhosted.org/CouchDB/views.html>) did not work. That is why we will use here a different approach - we will insert some JavaScript code from Python:

In [118]:

```
couch = couchdb.Server('http://szwabin:analiza@localhost:5984')
db = couch['demodb']
```

In [119]:

```
map_fun = ''' function(doc) {
  if (doc.done === "False"){
    emit(doc.task);
  }
}
'''
results = db.query(map_fun)
```

In [120]:

```
results
```

Out[120]:

```
<ViewResults <TemporaryView ' function(doc) {\n  if (doc.done === "F
alse"){\n    emit(doc.task);\n  }\n}' None> {}>
```

In [121]:

```
for res in results:
    print(res.key)
```

```
task 1
```

## MongoDB

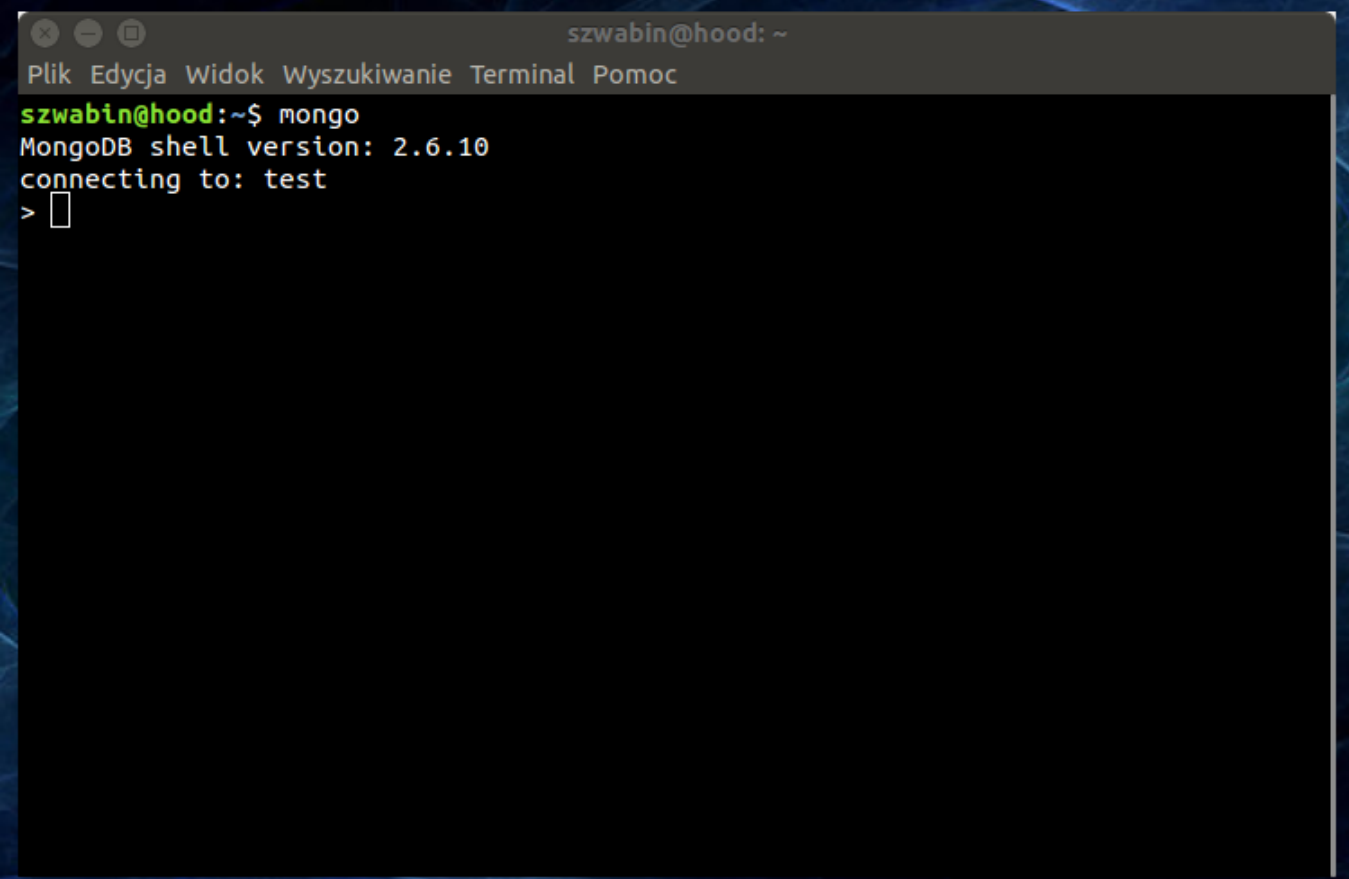
- document-oriented NoSQL database
- implemented in C++
- efficient
- high scalability
- data format similar to JSON
- JavaScript for user-defined queries
- transactions supported at a single-document level

### First steps

Installation on Ubuntu is easy:

```
sudo apt-get install mongodb
```

After installation, the mongo command may be used to start the MongoDB console:



```
szwabin@hood: ~  
Plik Edycja Widok Wyszukiwanie Terminal Pomoc  
szwabin@hood:~$ mongo  
MongoDB shell version: 2.6.10  
connecting to: test  
> 
```

### Creating new database or collection

A database will be created automatically when we start to insert some data records. For instance, you may copy the following code and paste it in the mongo shell in order to create a database with some records inside (example taken from <http://code.tutsplus.com/tutorials/getting-started-with-mongodb-part-1--net-22879> (<http://code.tutsplus.com/tutorials/getting-started-with-mongodb-part-1--net-22879>)):

```
db.nettuts.insert({
  first: 'matthew',
  last: 'setter',
  dob: '21/04/1978',
  gender: 'm',
  hair_colour: 'brown',
  occupation: 'developer',
  nationality: 'australian'
});
db.nettuts.insert({
  first: 'james',
  last: 'caan',
  dob: '26/03/1940',
  gender: 'm',
  hair_colour: 'brown',
  occupation: 'actor',
  nationality: 'american'
});
db.nettuts.insert({
  first: 'arnold',
  last: 'schwarzenegger',
  dob: '03/06/1925',
  gender: 'm',
  hair_colour: 'brown',
  occupation: 'actor',
  nationality: 'american'
});
db.nettuts.insert({
  first: 'tony',
  last: 'curtis',
  dob: '21/04/1978',
  gender: 'm',
  hair_colour: 'brown',
  occupation: 'developer',
  nationality: 'american'
});
db.nettuts.insert({
  first: 'jamie lee',
  last: 'curtis',
  dob: '22/11/1958',
  gender: 'f',
  hair_colour: 'brown',
  occupation: 'actor',
  nationality: 'american'
});
db.nettuts.insert({
  first: 'michael',
  last: 'caine',
  dob: '14/03/1933',
  gender: 'm',
  hair_colour: 'brown',
  occupation: 'actor',
```

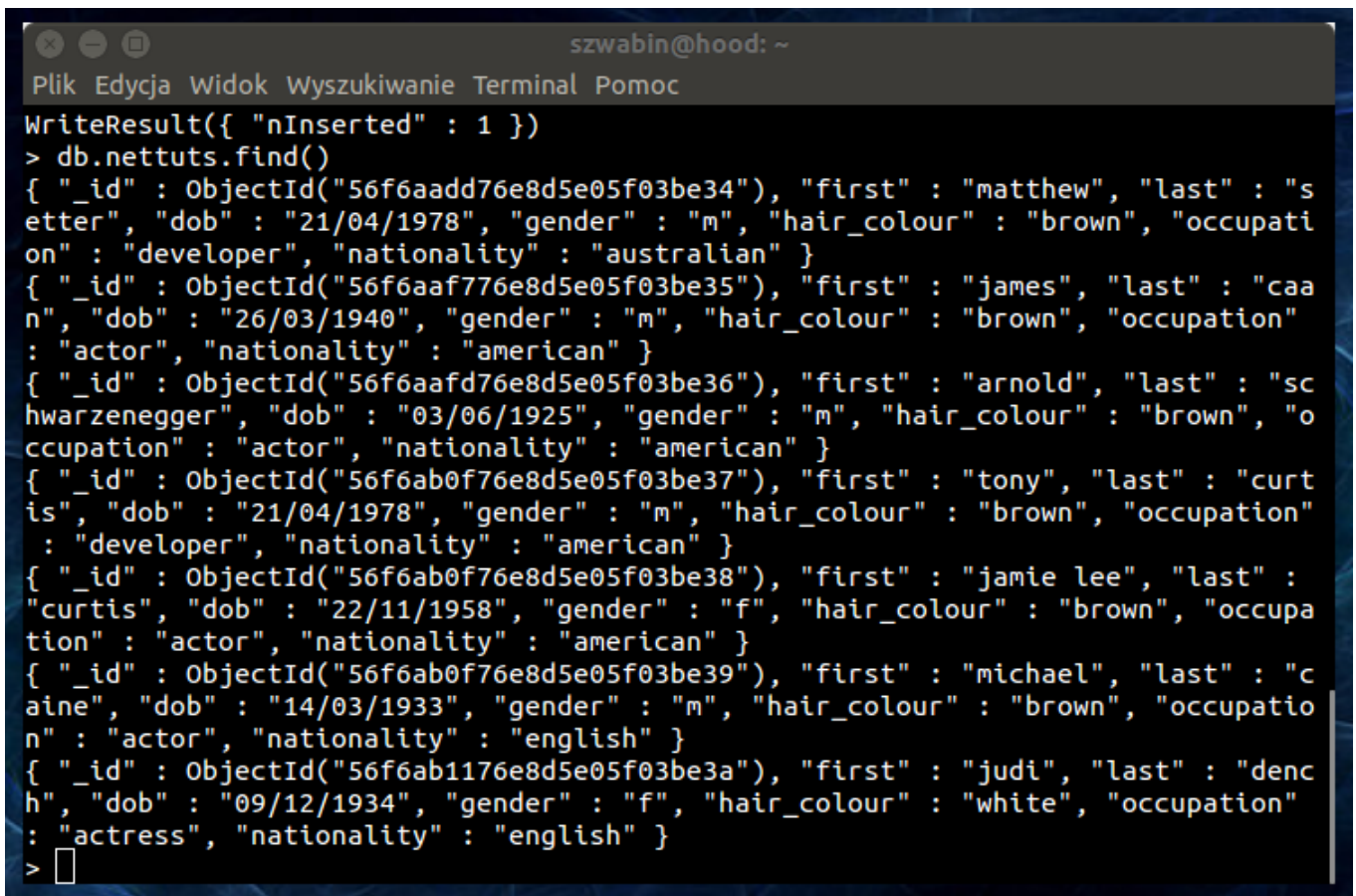
```

    nationality: 'english'
  });
db.nettuts.insert({
  first: 'judi',
  last: 'dench',
  dob: '09/12/1934',
  gender: 'f',
  hair_colour: 'white',
  occupation: 'actress',
  nationality: 'english'
});

```

Let us check if the database and accompanying records have been created:

```
db.nettuts.find()
```



The screenshot shows a terminal window with the title 'szwabin@hood: ~'. The terminal output displays the result of a MongoDB insert operation and a subsequent find operation. The insert operation returned a 'WriteResult' object indicating 1 record was inserted. The find operation returned a list of 8 records, each represented as a JSON object with fields for '\_id', 'first', 'last', 'dob', 'gender', 'hair\_colour', 'occupation', and 'nationality'.

```

WriteResult({ "nInserted" : 1 })
> db.nettuts.find()
{ "_id" : ObjectId("56f6aadd76e8d5e05f03be34"), "first" : "matthew", "last" : "s
etter", "dob" : "21/04/1978", "gender" : "m", "hair_colour" : "brown", "occupati
on" : "developer", "nationality" : "australian" }
{ "_id" : ObjectId("56f6aaf776e8d5e05f03be35"), "first" : "james", "last" : "caa
n", "dob" : "26/03/1940", "gender" : "m", "hair_colour" : "brown", "occupation"
: "actor", "nationality" : "american" }
{ "_id" : ObjectId("56f6aafd76e8d5e05f03be36"), "first" : "arnold", "last" : "sc
hwarzenegger", "dob" : "03/06/1925", "gender" : "m", "hair_colour" : "brown", "o
ccupation" : "actor", "nationality" : "american" }
{ "_id" : ObjectId("56f6ab0f76e8d5e05f03be37"), "first" : "tony", "last" : "curt
is", "dob" : "21/04/1978", "gender" : "m", "hair_colour" : "brown", "occupation"
: "developer", "nationality" : "american" }
{ "_id" : ObjectId("56f6ab0f76e8d5e05f03be38"), "first" : "jamie lee", "last" :
"curtis", "dob" : "22/11/1958", "gender" : "f", "hair_colour" : "brown", "occupa
tion" : "actor", "nationality" : "american" }
{ "_id" : ObjectId("56f6ab0f76e8d5e05f03be39"), "first" : "michael", "last" : "c
aine", "dob" : "14/03/1933", "gender" : "m", "hair_colour" : "brown", "occupatio
n" : "actor", "nationality" : "english" }
{ "_id" : ObjectId("56f6ab1176e8d5e05f03be3a"), "first" : "judi", "last" : "denc
h", "dob" : "09/12/1934", "gender" : "f", "hair_colour" : "white", "occupation"
: "actress", "nationality" : "english" }
>

```

### Searching for records with selectors

Selectors are to MongoDB what WHERE clauses are to SQL. If we want to find all female actors for instance, we run the following command:

```
db.nettuts.find({gender : 'f'})
```

We should get the following result:

```
> db.nettuts.find({gender : 'f'})
{ "_id" : ObjectId("56f6ab0f76e8d5e05f03be38"), "first" : "jamie lee", "last" : "curtis", "dob" : "22/11/1958", "gender" : "f", "hair_colour" : "brown", "occupation" : "actor", "nationality" : "american" }
{ "_id" : ObjectId("56f6ab1176e8d5e05f03be3a"), "first" : "judi", "last" : "dench", "dob" : "09/12/1934", "gender" : "f", "hair_colour" : "white", "occupation" : "actress", "nationality" : "english" }
```

Now we look for actors that are male or American:

```
> db.nettuts.find({gender: 'm', $or: [{nationality: 'american'}]});
{ "_id" : ObjectId("56f6aaf776e8d5e05f03be35"), "first" : "james", "last" : "caan", "dob" : "26/03/1940", "gender" : "m", "hair_colour" : "brown", "occupation" : "actor", "nationality" : "american" }
{ "_id" : ObjectId("56f6aafd76e8d5e05f03be36"), "first" : "arnold", "last" : "schwarzenegger", "dob" : "03/06/1925", "gender" : "m", "hair_colour" : "brown", "occupation" : "actor", "nationality" : "american" }
{ "_id" : ObjectId("56f6ab0f76e8d5e05f03be37"), "first" : "tony", "last" : "curtis", "dob" : "21/04/1978", "gender" : "m", "hair_colour" : "brown", "occupation" : "developer", "nationality" : "american" }
```

The queries can be sorted:

```
> db.nettuts.find({gender: 'm', $or: [{nationality: 'english'}, {nationality: 'american'}]}).sort({nationality: -1});
{ "_id" : ObjectId("56f6ab0f76e8d5e05f03be39"), "first" : "michael", "last" : "caine", "dob" : "14/03/1933", "gender" : "m", "hair_colour" : "brown", "occupation" : "actor", "nationality" : "english" }
{ "_id" : ObjectId("56f6aaf776e8d5e05f03be35"), "first" : "james", "last" : "caan", "dob" : "26/03/1940", "gender" : "m", "hair_colour" : "brown", "occupation" : "actor", "nationality" : "american" }
{ "_id" : ObjectId("56f6aafd76e8d5e05f03be36"), "first" : "arnold", "last" : "schwarzenegger", "dob" : "03/06/1925", "gender" : "m", "hair_colour" : "brown", "occupation" : "actor", "nationality" : "american" }
{ "_id" : ObjectId("56f6ab0f76e8d5e05f03be37"), "first" : "tony", "last" : "curtis", "dob" : "21/04/1978", "gender" : "m", "hair_colour" : "brown", "occupation" : "developer", "nationality" : "american" }
```

This time with descending order in nationality and ascending one in first names:

```
> db.nettuts.find({gender: 'm', $or: [{nationality: 'english'}, {nationality: 'american'}]}).sort({nationality: -1, first: 1});
{ "_id" : ObjectId("56f6ab0f76e8d5e05f03be39"), "first" : "michael", "last" : "caine", "dob" : "14/03/1933", "gender" : "m", "hair_colour" : "brown", "occupation" : "actor", "nationality" : "english" }
{ "_id" : ObjectId("56f6aafd76e8d5e05f03be36"), "first" : "arnold", "last" : "schwarzenegger", "dob" : "03/06/1925", "gender" : "m", "hair_colour" : "brown", "occupation" : "actor", "nationality" : "american" }
{ "_id" : ObjectId("56f6aaf776e8d5e05f03be35"), "first" : "james", "last" : "caan", "dob" : "26/03/1940", "gender" : "m", "hair_colour" : "brown", "occupation" : "actor", "nationality" : "american" }
{ "_id" : ObjectId("56f6ab0f76e8d5e05f03be37"), "first" : "tony", "last" : "curtis", "dob" : "21/04/1978", "gender" : "m", "hair_colour" : "brown", "occupation" : "developer", "nationality" : "american" }
```

### Limiting records

```
> db.nettuts.find({gender: 'm'}).limit(2);
{ "_id" : ObjectId("56f6aadd76e8d5e05f03be34"), "first" : "matthew", "last" : "setter", "dob" : "21/04/1978", "gender" : "m", "hair_colour" : "brown", "occupation" : "developer", "nationality" : "australian" }
{ "_id" : ObjectId("56f6aaf776e8d5e05f03be35"), "first" : "james", "last" : "caan", "dob" : "26/03/1940", "gender" : "m", "hair_colour" : "brown", "occupation" : "actor", "nationality" : "american" }
```

If we want to skip over the first two:

```
> db.nettuts.find({gender: 'm'}).limit(2).skip(2);
{ "_id" : ObjectId("56f6aaafd76e8d5e05f03be36"), "first" : "arnold", "last" : "schwarzenegger", "dob" : "03/06/1925", "gender" : "m", "hair_colour" : "brown", "occupation" : "actor", "nationality" : "american" }
{ "_id" : ObjectId("56f6ab0f76e8d5e05f03be37"), "first" : "tony", "last" : "curtis", "dob" : "21/04/1978", "gender" : "m", "hair_colour" : "brown", "occupation" : "developer", "nationality" : "american" }
```

### Updating records

```
> db.nettuts.update({first: 'james', last: 'caan'}, {$set: {hair_colour: 'black'}});
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

### Deleting records

```
> db.nettuts.remove({first: 'james', last: 'caan'});
WriteResult({ "nRemoved" : 1 })
> db.nettuts.find({first: 'james', last: 'caan'});
```

If we want to delete all the records from the database:

```
db.nettuts.remove();
```

## MapReduce

We want to find the total count of all the females in the group. First we define the map function:

```
> var map = function() {  
...   emit( { gender: this.gender }, { count: 1 } );  
... }
```

Our reduce function is going to take the output from the map function and use it to keep a running total of the count for each gender:

```
> var reduce = function(key, values) {  
...   var result = { count : 0 };  
...  
...   values.forEach(function(value){  
...     result.count += value.count;  
...   })  
...  
...   return result;  
... }
```

We put them together by calling the mapReduce function in the database. The result will be stored in a new collection called gender:

```
> var res = db.nettuts.mapReduce( map, reduce, { out : 'gender' } );
```

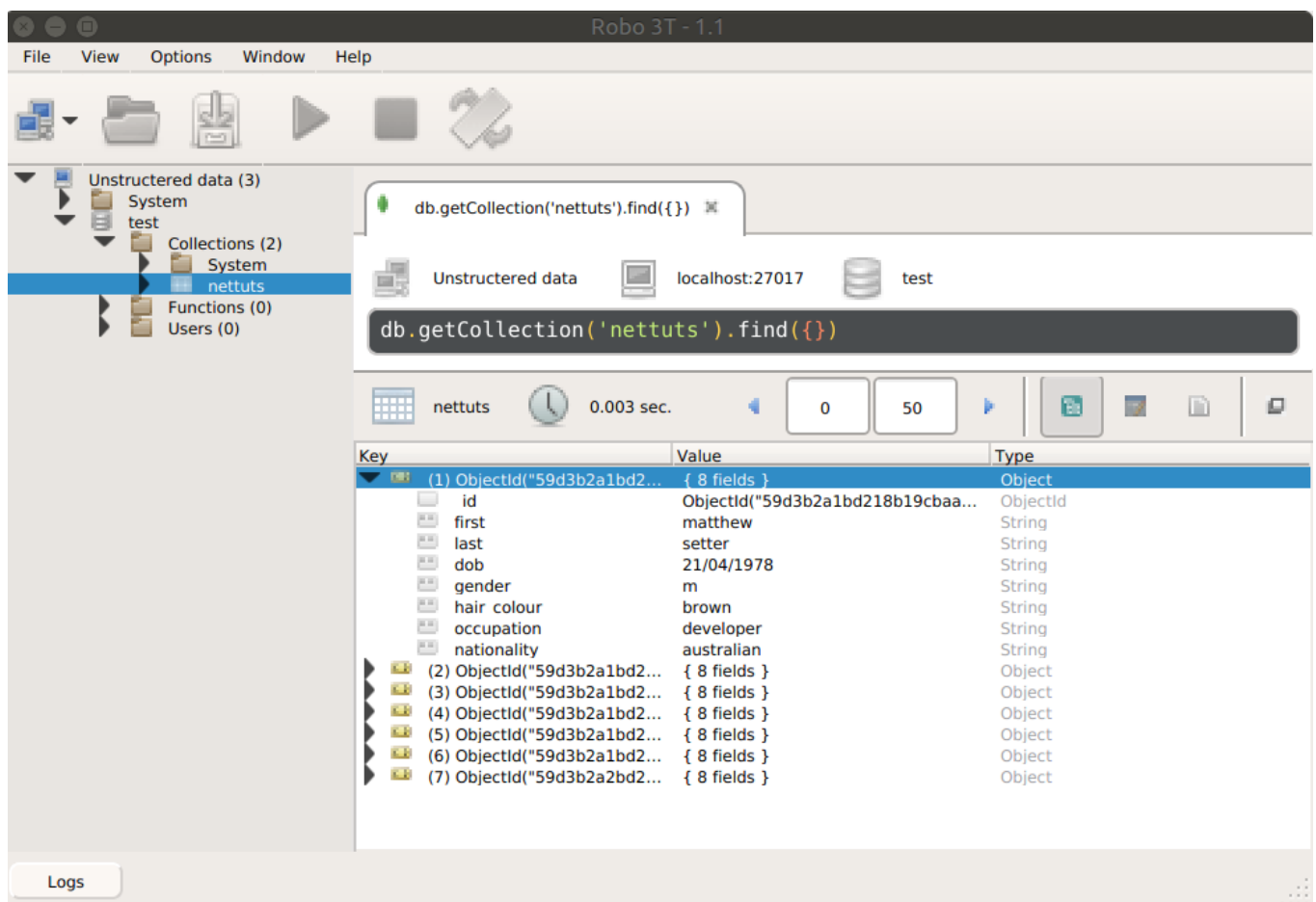
Let us have a look at the results:

```
> db.gender.find()  
{ "_id" : { "gender" : "f" }, "value" : { "count" : 2 } }  
{ "_id" : { "gender" : "m" }, "value" : { "count" : 4 } }
```



## GUI

- list of administrative tools in MongoDB ecosystem: <https://docs.mongodb.com/ecosystem/tools/> (<https://docs.mongodb.com/ecosystem/tools/>)
- Robo 3T (formerly Robomongo): <https://robomongo.org/> (<https://robomongo.org/>)



## MongoDB and Python

- pymongo module required

### Making a connection

In [122]:

```
from pymongo import MongoClient
client = MongoClient()
```

### Getting (or creating) a database

In [123]:

```
db = client["test"] # client.test works as well
```

### Checking the data

In [124]:

```
cursor = db.nettuts.find()
for doc in cursor:
    print(doc)
```

```
{'first': 'matthew', 'gender': 'm', '_id': ObjectId('59d3b2a1bd218b19cbaa4dc1'), 'nationality': 'australian', 'hair_colour': 'brown', 'occupation': 'developer', 'dob': '21/04/1978', 'last': 'setter'}
{'first': 'james', 'gender': 'm', '_id': ObjectId('59d3b2a1bd218b19cbaa4dc2'), 'nationality': 'american', 'hair_colour': 'brown', 'occupation': 'actor', 'dob': '26/03/1940', 'last': 'caan'}
{'first': 'arnold', 'gender': 'm', '_id': ObjectId('59d3b2a1bd218b19cbaa4dc3'), 'nationality': 'american', 'hair_colour': 'brown', 'occupation': 'actor', 'dob': '03/06/1925', 'last': 'schwarzenegger'}
{'first': 'tony', 'gender': 'm', '_id': ObjectId('59d3b2a1bd218b19cbaa4dc4'), 'nationality': 'american', 'hair_colour': 'brown', 'occupation': 'developer', 'dob': '21/04/1978', 'last': 'curtis'}
{'first': 'jamie lee', 'gender': 'f', '_id': ObjectId('59d3b2a1bd218b19cbaa4dc5'), 'nationality': 'american', 'hair_colour': 'brown', 'occupation': 'actor', 'dob': '22/11/1958', 'last': 'curtis'}
{'first': 'michael', 'gender': 'm', '_id': ObjectId('59d3b2a1bd218b19cbaa4dc6'), 'nationality': 'english', 'hair_colour': 'brown', 'occupation': 'actor', 'dob': '14/03/1933', 'last': 'caine'}
{'first': 'judi', 'gender': 'f', '_id': ObjectId('59d3b2a2bd218b19cbaa4dc7'), 'nationality': 'english', 'hair_colour': 'white', 'occupation': 'actress', 'dob': '09/12/1934', 'last': 'dench'}
```

In [125]:

```
cursor = db.nettuts.find({'gender' : 'm'})
for doc in cursor:
    print(doc)
```

```
{'first': 'matthew', 'gender': 'm', '_id': ObjectId('59d3b2a1bd218b19cbaa4dc1'), 'nationality': 'australian', 'hair_colour': 'brown', 'occupation': 'developer', 'dob': '21/04/1978', 'last': 'setter'}
{'first': 'james', 'gender': 'm', '_id': ObjectId('59d3b2a1bd218b19cbaa4dc2'), 'nationality': 'american', 'hair_colour': 'brown', 'occupation': 'actor', 'dob': '26/03/1940', 'last': 'caan'}
{'first': 'arnold', 'gender': 'm', '_id': ObjectId('59d3b2a1bd218b19cbaa4dc3'), 'nationality': 'american', 'hair_colour': 'brown', 'occupation': 'actor', 'dob': '03/06/1925', 'last': 'schwarzenegger'}
{'first': 'tony', 'gender': 'm', '_id': ObjectId('59d3b2a1bd218b19cbaa4dc4'), 'nationality': 'american', 'hair_colour': 'brown', 'occupation': 'developer', 'dob': '21/04/1978', 'last': 'curtis'}
{'first': 'michael', 'gender': 'm', '_id': ObjectId('59d3b2a1bd218b19cbaa4dc6'), 'nationality': 'english', 'hair_colour': 'brown', 'occupation': 'actor', 'dob': '14/03/1933', 'last': 'caine'}
```

### Inserting new documents

In [126]:

```
result = db.nettuts.insert_one(
{
    'first': 'james',
    'last': 'caan',
    'dob': '26/03/1940',
    'gender': 'm',
    'hair_colour': 'brown',
    'occupation': 'actor',
    'nationality': 'american'
})
```

In [127]:

```
print(result.inserted_id)
```

59e909c8f445e8142edc6cb0

In [128]:

```
cursor = db.nettuts.find({'last' : 'caan'})
for doc in cursor:
    print(doc)
```

```
{'first': 'james', 'gender': 'm', '_id': ObjectId('59d3b2a1bd218b19c
baa4dc2'), 'nationality': 'american', 'hair_colour': 'brown', 'occup
ation': 'actor', 'dob': '26/03/1940', 'last': 'caan'}
{'first': 'james', 'gender': 'm', 'occupation': 'actor', 'dob': '26/
03/1940', 'nationality': 'american', 'hair_colour': 'brown', '_id':
ObjectId('59e909c8f445e8142edc6cb0'), 'last': 'caan'}
```

## Document update

In [129]:

```
result = db.nettuts.update_one({'first': 'james', 'last': 'caan'}, {'$set': {'ha
ir_colour': 'red'}})
```

In [130]:

```
print(result.modified_count)
```

1

In [131]:

```
cursor = db.nettuts.find({'last' : 'caan'})
for doc in cursor:
    print(doc)
```

```
{'first': 'james', 'gender': 'm', '_id': ObjectId('59d3b2a1bd218b19c
baa4dc2'), 'nationality': 'american', 'hair_colour': 'red', 'occupat
ion': 'actor', 'dob': '26/03/1940', 'last': 'caan'}
{'first': 'james', 'gender': 'm', 'occupation': 'actor', 'dob': '26/
03/1940', 'nationality': 'american', 'hair_colour': 'brown', '_id':
ObjectId('59e909c8f445e8142edc6cb0'), 'last': 'caan'}
```

In [132]:

```
result = db.nettuts.update_many({'gender': 'f'}, {'$set': {'gender': 'female'}})
```

In [133]:

```
cursor = db.nettuts.find()
for doc in cursor:
    print(doc)
```

```
{'first': 'matthew', 'gender': 'm', '_id': ObjectId('59d3b2a1bd218b19cbaa4dc1'), 'nationality': 'australian', 'hair_colour': 'brown', 'occupation': 'developer', 'dob': '21/04/1978', 'last': 'setter'}
{'first': 'james', 'gender': 'm', '_id': ObjectId('59d3b2a1bd218b19cbaa4dc2'), 'nationality': 'american', 'hair_colour': 'red', 'occupation': 'actor', 'dob': '26/03/1940', 'last': 'caan'}
{'first': 'arnold', 'gender': 'm', '_id': ObjectId('59d3b2a1bd218b19cbaa4dc3'), 'nationality': 'american', 'hair_colour': 'brown', 'occupation': 'actor', 'dob': '03/06/1925', 'last': 'schwarzenegger'}
{'first': 'tony', 'gender': 'm', '_id': ObjectId('59d3b2a1bd218b19cbaa4dc4'), 'nationality': 'american', 'hair_colour': 'brown', 'occupation': 'developer', 'dob': '21/04/1978', 'last': 'curtis'}
{'first': 'jamie lee', 'gender': 'female', '_id': ObjectId('59d3b2a1bd218b19cbaa4dc5'), 'nationality': 'american', 'hair_colour': 'brown', 'occupation': 'actor', 'dob': '22/11/1958', 'last': 'curtis'}
{'first': 'michael', 'gender': 'm', '_id': ObjectId('59d3b2a1bd218b19cbaa4dc6'), 'nationality': 'english', 'hair_colour': 'brown', 'occupation': 'actor', 'dob': '14/03/1933', 'last': 'caine'}
{'first': 'judi', 'gender': 'female', '_id': ObjectId('59d3b2a2bd218b19cbaa4dc7'), 'nationality': 'english', 'hair_colour': 'white', 'occupation': 'actress', 'dob': '09/12/1934', 'last': 'dench'}
{'first': 'james', 'gender': 'm', 'occupation': 'actor', 'dob': '26/03/1940', 'nationality': 'american', 'hair_colour': 'brown', '_id': ObjectId('59e909c8f445e8142edc6cb0'), 'last': 'caan'}
```

## Removing data

In [134]:

```
result = db.nettuts.delete_many({"last": "caan"})
print(result.deleted_count)
```

2

In [135]:

```
cursor = db.nettuts.find()
for doc in cursor:
    print(doc)
```

```
{'first': 'matthew', 'gender': 'm', '_id': ObjectId('59d3b2a1bd218b19cbaa4dc1'), 'nationality': 'australian', 'hair_colour': 'brown', 'occupation': 'developer', 'dob': '21/04/1978', 'last': 'setter'}
{'first': 'arnold', 'gender': 'm', '_id': ObjectId('59d3b2a1bd218b19cbaa4dc3'), 'nationality': 'american', 'hair_colour': 'brown', 'occupation': 'actor', 'dob': '03/06/1925', 'last': 'schwarzenegger'}
{'first': 'tony', 'gender': 'm', '_id': ObjectId('59d3b2a1bd218b19cbaa4dc4'), 'nationality': 'american', 'hair_colour': 'brown', 'occupation': 'developer', 'dob': '21/04/1978', 'last': 'curtis'}
{'first': 'jamie lee', 'gender': 'female', '_id': ObjectId('59d3b2a1bd218b19cbaa4dc5'), 'nationality': 'american', 'hair_colour': 'brown', 'occupation': 'actor', 'dob': '22/11/1958', 'last': 'curtis'}
{'first': 'michael', 'gender': 'm', '_id': ObjectId('59d3b2a1bd218b19cbaa4dc6'), 'nationality': 'english', 'hair_colour': 'brown', 'occupation': 'actor', 'dob': '14/03/1933', 'last': 'caine'}
{'first': 'judi', 'gender': 'female', '_id': ObjectId('59d3b2a2bd218b19cbaa4dc7'), 'nationality': 'english', 'hair_colour': 'white', 'occupation': 'actress', 'dob': '09/12/1934', 'last': 'dench'}
```

## Data aggregation

In [136]:

```
cursor = db.nettuts.aggregate(
    [
        {"$group": {"_id" : "$gender", "count" : {"$sum": 1}}} # look at $ in front of gender!!!!
    ]
)
```

In [137]:

```
for doc in cursor:
    print(doc)
```

```
{'_id': 'female', 'count': 2}
{'_id': 'm', 'count': 4}
```

## Case study - CouchDB, Python and Twitter

### Class definition

In [29]:

```
import couchdb
import couchdb.design

class TweetStore(object):
    def __init__(self, dbname, url='http://localhost:5984/'):
        try: #establish a connection
            self.server = couchdb.Server(url=url)
            self.db = self.server.create(dbname)
            self._create_views()
        except couchdb.http.PreconditionFailed: #if database exists, connect to
it
            self.db = self.server[dbname]

    def _create_views(self):
        #first view - number of tweets
        count_map = 'function(doc) { emit(doc.id, 1); }'
        count_reduce = 'function(keys, values) { return sum(values); }'
        view = couchdb.design.ViewDefinition('twitter',
                                             'count_tweets',
                                             count_map,
                                             reduce_fun=count_reduce)

        #insert view into the database
        view.get_doc(self.db)
        view.sync(self.db)

        #second view - show tweets
        get_tweets = 'function(doc) { emit(("00000000000000000000"+doc.id).slice
(-19), doc); }'
        view = couchdb.design.ViewDefinition('twitter', 'get_tweets',
get_tweets)
        view.get_doc(self.db)
        view.sync(self.db)

    def save_tweet(self, tw):
        tw['_id'] = tw['id_str']
        self.db.save(tw)

    def count_tweets(self):
        for doc in self.db.view('twitter/count_tweets'):
            return doc.value

    def get_tweets(self):
        return self.db.view('twitter/get_tweets')
```

Looking for tweets

In [32]:

```
import tweepy

consumer_key="your_consumer_key"
consumer_secret="your_secret_consumer_key"
access_token="your_access_token"
access_token_secret="your_secret_acces_token"

auth = tweepy.auth.OAuthHandler(consumer_key, consumer_secret)
auth.set_access_token(access_token,access_token_secret)
api = tweepy.API(auth)
query = "python"
max_tweets = 20
search_results = [status._json for status in tweepy.Cursor(api.search,
q=query).items(max_tweets)]

storage =
TweetStore('twitter_mining',url='http://name:password@127.0.0.1:5984/')

for item in search_results:
    storage.save_tweet(item)
    print('done!')
```

```
done!
done!
done!
done!
done!
done!
done!
done!
done!
done!
done!
done!
done!
done!
done!
done!
done!
done!
done!
done!
```

### Using the views

In [33]:

```
print(storage.count_tweets())
```

20