DUT STID, Université de la Côte d'Azur

Bases de données avancées

MongoDB et bases noSQL

Prof. Dario Malchiodi

UNIVERSITÀ DEGLI STUDI DI MILANO
DIPARTIMENTO DI INFORMATICA

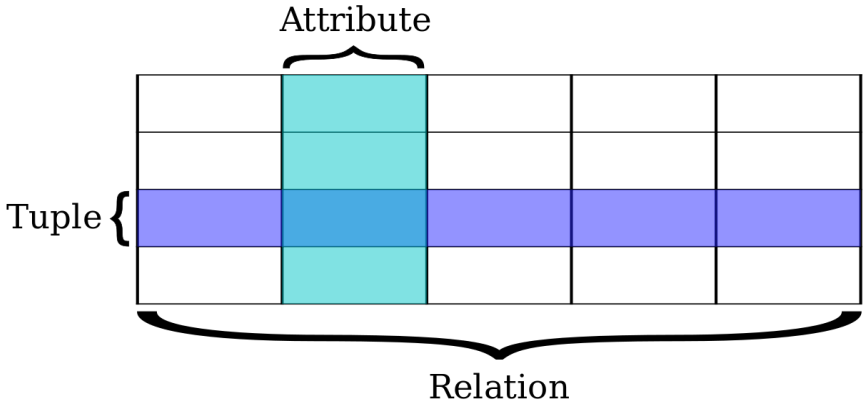
UNIVERSITÉ CÔTE D'AZUR

SGBD

Données organisées en termes de *relations* (qui signifie ensembles de tuples)

Entre les plus utilisés:

- Oracle
- MySQL
- PostgreSQL
- SQLite



Pourquoi noSQL?

Parce que le monde réel n'est guère en ordre!

- Fréquemment, les données **ne suivent pas** un schéma prédéfini
- et ça c'est plutôt la règle dans le domaine de la data science

En plus, noSQL a des avantages très intéréssants:

- utiliser directement des objéts complex/structurés
- extensibilité horizontale (sharding)
- robustesse en exploitant la redondance

Voilà MongoDB

MongoDB (http://www.mongodb.com (http://www.mongodb.com)) est un SGBD noSQL open source **orienté aux documents**.

- Developpé dès 2007
- En production à partir de la version 1.4 (mars 2010)
- Dernière version stable: 4.0, publiée en october 2018
- (le nom fait référence à «humongous»)

Voilà MongoDB

MongoDB a, quand même, des points critiques:

- ses index sont organisés en termes de structures données qui utilisent beaocoup de RAM
- les DB nécéssitent d'habitude plus space de disque dur que leurs homologues dans un SGBD classique
- il n'y a pas un suuport direct aux transactions, aux jointures, ...

In a SGBD classique

- Une BD est un ensemble de tables
- Une table est un ensemble de lignes
- Une ligne est une séquence de valeurs pour des attributes (fixés)
- Un **schéma** definis la structures des lignes dans une table
- On peut effectuer des jointures entre tables

In a SGBD ~~classique~~ orienté aux documents

- Une BD est un ensemble de ~~tables~~ collections
- Une ~~table~~ collection est un ensemble de ~~lignes~~ documents
- ~~Une ligne~~ Un document est ~~une séquence des valeurs pour des attributes (fixés)~~ un ensemble (variable) des couples (nom, valeur)
- On **ne peut pas** effectuer des jointures entre tables (bon, pas directement)
- ~~Un **schéma** definis la structure des ligne~~ dans une table les lignes ne sont pas structurées

Réprésentation des données

- Dans une BD relationnelle les lignes sont codifiées comme une séquence de valeurs dont le type est fixé
- Ça n'est pas possible avec MongoDB, parce que il n'y a pas une structure prédéfinie pour les documents
- C'est pour ça que les documents sont réprésentés en utilisant le format JSON format
- JSON (JavaScript Object Notation, http://www.json.org (http://www.json.org)) est un format d'échange des données *light*, facile à lire (par les humains) et à analyser syntactiquement (par les ordinateurs)

Light?

Très simplement:

```
['one', 'two', 'three']
```

est bien plus simple que

```
<array>
  <element>one</element>
  <element>two</element>
  <element>three</element>
</array>
```

Réprésentation interne

MongoDB effectue le stockage en utilisant le format BSON format (Binary JSON, http://bsonspec.org/ (http://bsonspec.org)), conçu pour:

- serialiser de façon efficace les documents JSON
- traverser aisement ces documents
- être toujours *light*

Components de MongoDB

`mongod`	serveur (en écoute sur le port 27017)
mongo	client javascript
API	pour Python, Java, Scala, Ruby, C, C++, ... (http://api.mongodb.org (http://api.mongodb.org))
installation	https://docs.mongodb.com/manual/installation (https://docs.mongodb.com/manual/installation)

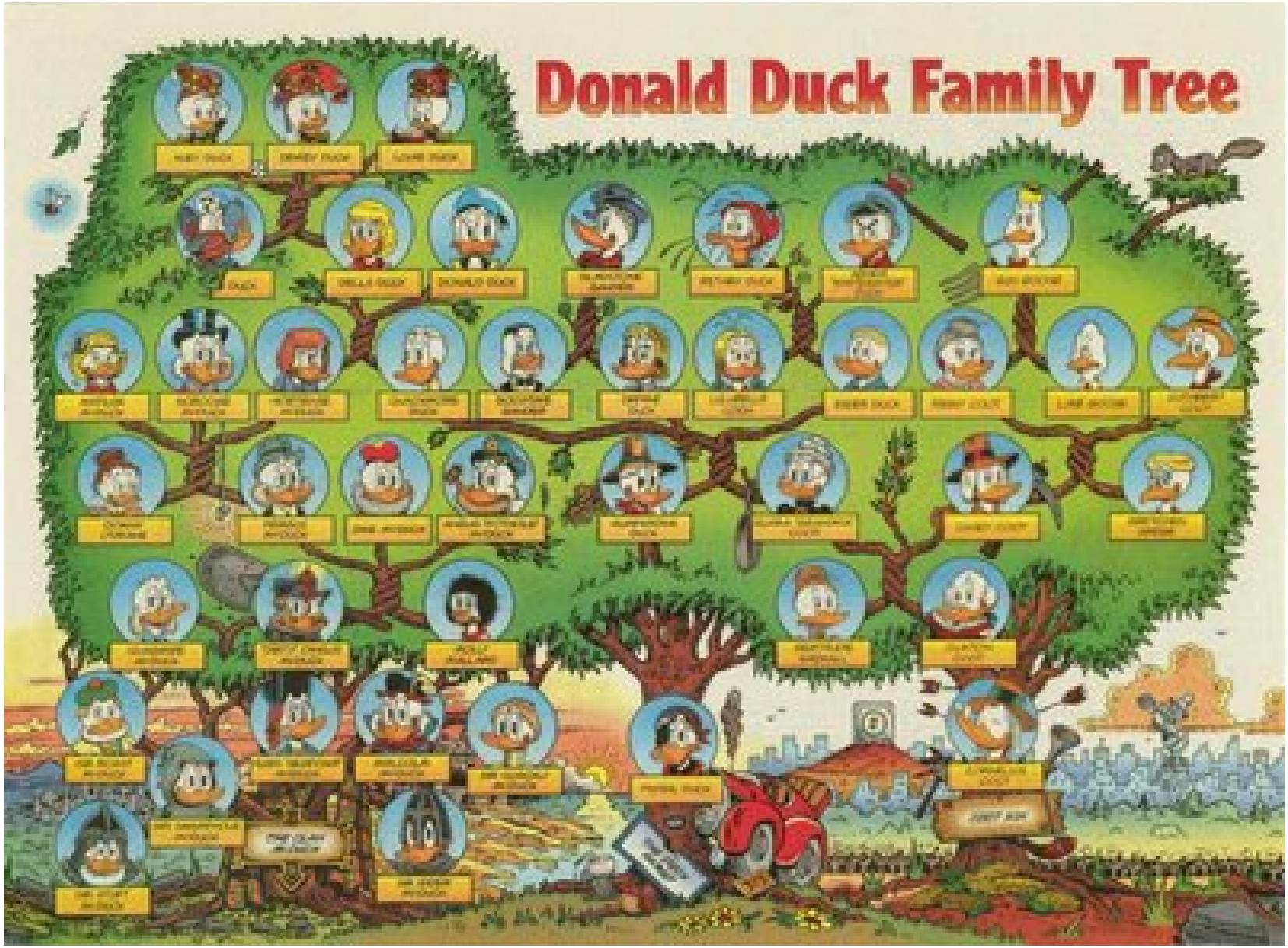
Exécution manuelle

```
$ mongod --dbpath .
2018-05-28T08:47:04.813+0200 I CONTROL  [initandlisten] MongoDB starting : pid=5172 port=27017 dbpath
=. ...
...

$ mongo
MongoDB shell version: 3.6.5
...
>
```

Il nous faut des données

Voyons qui habite Donaldville



Opérations préliminaires

In [11]:
```
import pymongo
client = pymongo.MongoClient('mongodb://localhost:27017/')
duckburg_db = client.duckburg # silently creates db
```

In [12]:
```
client.list_database_names()       # it's empty, thus only default dbs will show up
```

Out[12]:
```
['admin', 'config', 'local']
```

In [13]:
```
ducks = duckburg_db.ducks       # silently creates collection within db
duckburg_db.list_collection_names()  # it will not show up, either
```

Out[13]:
```
[]
```

Ajouter des documents dans une collection

La façon la plus facile d'insérer un document dans une BD est celle d'utiliser la méthode `insert_one`

In [14]:
```
r = ducks.insert_one({'first_name': 'Donald',
                      'last_name': 'Duck',
                      'gender': 'M',
                      'car': {'model': 'American Bantam',
                              'license_plate': 313},
                      'birth_year': 1920,
                      'first_appearance': 1934})
```

In [15]:
```
r
```

Out[15]:
```
<pymongo.results.InsertOneResult at 0x7f9e400af608>
```

La méthode `insert_one` retourne un objet qui réunis des info sur le résultat

- `acknowledged` est un valeur booléen qui devient étal à `True` quand l'écriture est terminée (il y aurait plus à dire sur ça, mais nous ne le ferons pas)

In [16]:
```
r.acknowledged
```

Out[16]:
```
True
```

- `inserted_id` est un attribut qui contient un ID univoque pour le document qui vient d'être créé (on va détailler ça dans un moment)

In [17]:
```
r.inserted_id
```

Out[17]:
```
ObjectId('5c804c86fc4b1718037992f9')
```

Qu'est-ce qu'il s'est passé après cette première écriture?

In [18]:
```
print(client.list_database_names())
print(duckburg_db.list_collection_names())
```

```
['admin', 'config', 'duckburg', 'local']
['ducks']
```

Quelques détails techniques

- Taille maximale des documents: 16MB
- L'opération d'insertion peut aussi se réaliser avec un *upsert* (on verra ça plus tard)

Ajoutons quelques documents à la BD

In [19]:
```
characters = [
    {'first_name': 'Grandma', 'last_name': 'Duck', 'gender': 'F',
     'car': {'model': 'Detroit Electric'},
     'birth_year': 1833, 'first_appearance': 1943,
     'hobbies': ['cooking', 'gardening']},
    {'first_name': 'Scrooge', 'last_name': 'McDuck', 'gender': 'M',
     'birth_year': 1867, 'first_appearance': 1947,
     'hobbies': ['finance', 'savings', 'swimming in money bins']},
    {'first_name': 'Gyro', 'last_name': 'Gearloose', 'gender': 'M',
     'first_appearance': 1952, 'hobbies': ['invention', 'study']},
    {'first_name': 'Ludwig', 'last_name': 'Von Drake', 'gender': 'M',
     'first_appearance': 1961, 'hobbies': ['study']}
]

r = ducks.insert_many(characters)
r
```

Out[19]:
```
<pymongo.results.InsertManyResult at 0x7f9e400becc8>
```

In [20]:
```
r.inserted_ids
```

Out[20]:
```
[ObjectId('5c804c97fc4b1718037992fa'),
 ObjectId('5c804c97fc4b1718037992fb'),
 ObjectId('5c804c97fc4b1718037992fc'),
 ObjectId('5c804c97fc4b1718037992fd')]
```

Lire une collection I

In [21]:
```
duck = ducks.find_one()
duck
```

Out[21]:
```
{'_id': ObjectId('5c804c86fc4b1718037992f9'),
 'first_name': 'Donald',
 'last_name': 'Duck',
 'gender': 'M',
 'car': {'model': 'American Bantam', 'license_plate': 313},
 'birth_year': 1920,
 'first_appearance': 1934}
```

Il n'y a aucune garantie sur *lequel* document sera obtenu!

Notez que le format JSON a été directement converti en un objet python, et donc on peut accéder au document en utilisant la syntaxe des dictionnaires

In [22]:
```
duck['first_name']
```

Out[22]:
```
'Donald'
```

C'est quoi _id? Et ObjectId?

In [23]:
```
ducks.find_one()['_id']
```

Out[23]:
```
ObjectId('5c804c86fc4b1718037992f9')
```

L'attribut `_id` est automatiquement ajouté à chaque document; il contient un ID unique qui vient utilisé pour indexer les collections. MongoDB utilise pour ça les objets de la classe `ObjectId`

In [24]:
```
from bson.objectid import ObjectId
[ObjectId(), ObjectId(), ObjectId()]
```

Out[24]:
```
[ObjectId('5c804ca4fc4b1718037992fe'),
 ObjectId('5c804ca4fc4b1718037992ff'),
 ObjectId('5c804ca4fc4b171803799300')]
```

Lire une collection II

La sélection de *plusieurs* documents qui correspondent à une requête est faite avec la méthode `find`

In [25]:
```
r = ducks.find()
r
```

Out[25]:
```
<pymongo.cursor.Cursor at 0x7f9e400ca470>
```

On va parler des curseurs dans une minute. Pour l'instant il suffit de dire que il peuvent être converti en liste pour afficher les documents sélectionnés

In [26]:
```
list(ducks.find())
```

Out[26]:
```
[{'_id': ObjectId('5c804c86fc4b1718037992f9'),
  'first_name': 'Donald',
  'last_name': 'Duck',
  'gender': 'M',
  'car': {'model': 'American Bantam', 'license_plate': 313},
  'birth_year': 1920,
  'first_appearance': 1934},
 {'_id': ObjectId('5c804c97fc4b1718037992fa'),
  'first_name': 'Grandma',
  'last_name': 'Duck',
  'gender': 'F',
  'car': {'model': 'Detroit Electric'},
  'birth_year': 1833,
  'first_appearance': 1943,
  'hobbies': ['cooking', 'gardening']},
 {'_id': ObjectId('5c804c97fc4b1718037992fb'),
  'first_name': 'Scrooge',
  'last_name': 'McDuck',
  'gender': 'M',
  'birth_year': 1867,
  'first_appearance': 1947,
  'hobbies': ['finance', 'savings', 'swimming in money bins']},
 {'_id': ObjectId('5c804c97fc4b1718037992fc'),
  'first_name': 'Gyro',
  'last_name': 'Gearloose',
  'gender': 'M',
  'first_appearance': 1952,
  'hobbies': ['invention', 'study']},
 {'_id': ObjectId('5c804c97fc4b1718037992fd'),
  'first_name': 'Ludwig',
  'last_name': 'Von Drake',
  'gender': 'M',
  'first_appearance': 1961,
  'hobbies': ['study']}]
```

Un affichage plus compacte des documents

Écrivons une fonction qui nous permet d'afficher les canard de façon plus compacte

In [27]:
```
def show_duck(d):
    try:
        return '{0} {1}'.format(d['first_name'], d['last_name'])
    except KeyError:
        return d['first_name']

show_duck(duck)
```

Out[27]:
```
'Donald Duck'
```

Lire une collection: selecteurs simples I

Basés sur l'existence de champs spécifiques

In [28]:
```
[show_duck(d) for d in ducks.find({'car': {'$exists': True}})]
```

Out[28]:
```
['Donald Duck', 'Grandma Duck']
```

Lire une collection: selecteurs simples II

Qui selectionnent des valeurs spécifiques

In [29]:
```
[show_duck(d) for d in ducks.find({'gender': 'F'})]
```

Out[29]:
```
['Grandma Duck']
```

In [30]:
```
[show_duck(d) for d in ducks.find({'first_appearance': 1961})]
```

Out[30]:
```
['Ludwig Von Drake']
```

Lire une collection: selecteurs simples III

Basés sur une relation

In [31]:
```
[show_duck(d) for d in ducks.find({'first_appearance': {'$gt': 1950}})]
```

Out[31]:
```
['Gyro Gearloose', 'Ludwig Von Drake']
```

Qui filtrent plusieurs valeurs pour un attribut

In [32]:
```
[show_duck(d) for d in ducks.find({'first_appearance': {'$in': [1947,
                                                                1961]}})]
```

Out[32]:
```
['Scrooge McDuck', 'Ludwig Von Drake']
```

Lire une collection: selecteurs complexes I

Conjunction logique (AND)

In [33]:
```
[show_duck(d) for d in ducks.find({'first_appearance': {'$lt': 1950},
                                   'gender': 'M'})]
```

Out[33]:
```
['Donald Duck', 'Scrooge McDuck']
```

Disjunction logique (OR)

In [34]:
```
[show_duck(d) for d in ducks.find({'$or': [{'birth_year': {'$gt': 1900}},
                                           {'gender': 'F'}]})]
```

Out[34]:
```
['Donald Duck', 'Grandma Duck']
```

Lire une collection: selecteurs complexes II

Basés sur le contenu d'un vecteur

In [35]:
```
[show_duck(d) for d in ducks.find({'hobbies': 'study'})]
```

Out[35]:
```
['Gyro Gearloose', 'Ludwig Von Drake']
```

In [36]:
```
[show_duck(d) for d in ducks.find({'hobbies': ['study']})]
```

Out[36]:
```
['Ludwig Von Drake']
```

In [37]:
```
[show_duck(d) for d in ducks.find({'hobbies.1': 'study'})]
```

Out[37]:
```
['Gyro Gearloose']
```

Lire une collection: selecteurs complexes III

Basés sur documents incorporés

In [38]:
```
[show_duck(d) for d in ducks.find({'car.model': 'American Bantam'})]
```

Out[38]:
```
['Donald Duck']
```

Lire une collection: selecteurs complexes IV

Basés sur expressions régulières

In [39]:
```
import re
regx = re.compile("uck$", re.IGNORECASE)

[show_duck(d) for d in ducks.find({'last_name': regx})]
```

Out[39]:
```
['Donald Duck', 'Grandma Duck', 'Scrooge McDuck']
```

Prêter attention à la flexibilité

L'utilisation d'expressions régulières, ainsi que de `$lt` ou `$gt`, peut être source d'inefficacités dans l'execution des requêtes

Projections I

Implementées par un deuxième argument de `find`

- qui spécifie les attributs à montrer...

In [40]:
```
list(ducks.find({'hobbies.1': 'study'},
                {'first_name': 1, 'last_name': 1}))
```

Out[40]:
```
[{'_id': ObjectId('5c804c97fc4b1718037992fc'),
  'first_name': 'Gyro',
  'last_name': 'Gearloose'}]
```

Projections II

- ...ou ceux à exclure

In [41]:
```
list(ducks.find({'hobbies.1': 'study'},
                {'_id': 0, 'gender': 0, 'birth_year': 0,
                 'first_appearance': 0, 'hobbies': 0}))
```

Out[41]:
```
[{'first_name': 'Gyro', 'last_name': 'Gearloose'}]
```

Projections III

- Selection et exclusion sont mutuellement exclusives
- Seule exception: l'attribut `_id` peut toujours être exclus:

In [42]:
```
list(ducks.find({'hobbies.1': 'study'},
                {'first_name': 1, 'last_name': 1, '_id': 0}))
```

Out[42]:
```
[{'first_name': 'Gyro', 'last_name': 'Gearloose'}]
```

Indexation I

Les collections peuvent être indexées en spécifiant un ensemble d'attributs et le corréspondant ordre de tri

In [43]:
```
ducks.create_index([('first_name', pymongo.ASCENDING),
                    ('first_appearance', pymongo.DESCENDING)])
```

Out[43]:
```
'first_name_1_first_appearance_-1'
```

Indexation II

Si une requête est *couverte* par un index, ce qui signifie

- tous les attributs sur lesquels on effectue un filtre,
- et tous les attributs affichés

sont dans l'index, ceci vient automatiquement utilisé pour traverser les documents quand la requête est exécutée.

In [44]:
```
regx = re.compile("^G.*", re.IGNORECASE)
list(ducks.find({'first_name': regx, 'first_appearance': {'$lt': 1955}},
                {'_id': 0, 'first_name': 1, 'first_appearance': 1}))
```

Out[44]:
```
[{'first_name': 'Grandma', 'first_appearance': 1943},
 {'first_name': 'Gyro', 'first_appearance': 1952}]
```

Curseurs

`find` retourne un *curseur* pour le résultat de la requête, et ceci est accessible via

- l'invocation de la méthode `next` (jusqu'à `StopIteration` soit déclenchée)

In [45]:
```
cursor = ducks.find()
try:
    while True:
        print(show_duck(cursor.next()))
except StopIteration:
    pass
```

```
Donald Duck
Grandma Duck
Scrooge McDuck
Gyro Gearloose
Ludwig Von Drake
```

- accès positionnel *à la liste* (aka `[]`)

In [46]:
```
cursor = ducks.find()
show_duck(cursor[1])
```

Out[46]:
```
'Grandma Duck'
```

In [47]:
```
show_duck(cursor[4])
```

Out[47]:
```
'Ludwig Von Drake'
```

(et on peut aussi remonter à l'arrière)

In [48]:
```
show_duck(cursor[2])
```

Out[48]:
```
'Scrooge McDuck'
```

- conversion a liste, boucle `for`

In [49]:
```
cursor = ducks.find()

for d in cursor:
    print(show_duck(d))
```

```
Donald Duck
Grandma Duck
Scrooge McDuck
Gyro Gearloose
Ludwig Von Drake
```

- invocation des méthodes `skip` , `limit` , `sort` et `count`

In [50]:
```
cursor = ducks.find().sort('last_name', pymongo.DESCENDING)
for d in cursor:
    print(show_duck(d))
```

```
Ludwig Von Drake
Scrooge McDuck
Gyro Gearloose
Donald Duck
Grandma Duck
```

In [51]:
```
cursor.rewind()
```

Out[51]:
```
<pymongo.cursor.Cursor at 0x7f9e40056898>
```

La éthode `update` est mauvais!

In [52]:
```
ducks.insert_one({'first_name': 'Magica', 'last_name': 'De Spell',
                  'first_appearance': 1961})
ducks.update({'first_name': 'Magica'}, {'first_appearance': 2000})
```

```
/home/malchiodi/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:3: DeprecationWarning: update is
deprecated. Use replace_one, update_one or update_many instead.
  This is separate from the ipykernel package so we can avoid doing imports until
```

Out[52]:
```
{'n': 1, 'nModified': 1, 'ok': 1.0, 'updatedExisting': True}
```

Il est important de noter que:

- la méthode est obsolète;
- elle retourne un dictionnaire (pas un objet) qui résume les résultats de l'operation

Vérifions si le document a été effectivement modifié:

In [53]:
```
ducks.find_one({'first_name': 'Magica'})
```

In [54]:
```
ducks.find_one({'first_appearance': 2000})
```

Out[54]:
```
{'_id': ObjectId('5c804cfcfc4b171803799301'), 'first_appearance': 2000}
```

Donc cett forme (obsolète) de `update` a complètement remplacé le document!

Solution

Uiliser les méthodes `update_one` et `update_many` en place de `update`

In [55]:
```
# OK, let's revert things
ducks.find_one_and_delete({'first_appearance': 2000})
ducks.insert_one({'first_name': 'Magica', 'last_name': 'De Spell',
                  'first_appearance': 1961})

ducks.update_one({'first_name': 'Magica'}, {'first_appearance': 2000})
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-55-6a0ad72fdd9c> in <module>()
      4                   'first_appearance': 1961})
      5
----> 6 ducks.update_one({'first_name': 'Magica'}, {'first_appearance': 2000})

~/anaconda3/lib/python3.6/site-packages/pymongo/collection.py in update_one(self, filter, update, upsert, by
pass_document_validation, collation, array_filters, session)
    983         """
    984         common.validate_is_mapping("filter", filter)
--> 985         common.validate_ok_for_update(update)
    986         common.validate_list_or_none('array_filters', array_filters)
    987

~/anaconda3/lib/python3.6/site-packages/pymongo/common.py in validate_ok_for_update(update)
    492     first = next(iter(update))
    493     if not first.startswith('$'):
--> 494         raise ValueError('update only works with $ operators')
    495
    496

ValueError: update only works with $ operators
```

WTF(irst_appearance)?

- L'erreur est due au fait que ces méthodes ne permettent pas de remplacer entierement un document
- Au lieu de ça, elles imposent l'utilisation de `$set`, qui permet de modifier un ou plusieurs attributs sans rien toucher du reste du document

In [56]:
```
r = ducks.update_one({'first_name': 'Magica'},
                     {'$set': {'first_appearance': 2000}})
r
```

Out[56]:
```
<pymongo.results.UpdateResult at 0x7f9e40053d08>
```

Aussi dans ce cas, l'opération retourne un objet qui réunit des infos sur le résultat:

In [57]:
```
# n. of matched documents
print(r.matched_count)
```

```
1
```

In [58]:
```
# n. of modified documents
r.modified_count
```

Out[58]:
```
1
```

In [59]:
```
# None if no upsertion, id of created document otherwise
# this deals with upsertions, will be explained later on
r.upserted_id
```

In [60]:
```
# results in the update-style
r.raw_result
```

Out[60]:
```
{'n': 1, 'nModified': 1, 'ok': 1.0, 'updatedExisting': True}
```

Maintenant on peut effectivement vérifier que la méthode se comporte comme prévu

In [61]:
```
ducks.find_one({'first_name': 'Magica'})
```

Out[61]:
```
{'_id': ObjectId('5c804d0ffc4b171803799302'),
 'first_name': 'Magica',
 'last_name': 'De Spell',
 'first_appearance': 2000}
```

`update_one` va modifier un seul des documents identifiés (et on ne peut pas savoir lequel), `update_many` va les modifier tous.

Éliminer des couples

Il y a aussi la possibilité d'*éliminer* une couple clé/valeur dans un document, en utilisant l'operateur `$unset` .

In [62]:
```
ducks.update_one({'first_name': 'Magica'},
                 {'$unset': {'first_appearance': ''}})
```

Out[62]:
```
<pymongo.results.UpdateResult at 0x7f9e4003d388>
```

In [63]:
```
ducks.find_one({'first_name': 'Magica'})
```

Out[63]:
```
{'_id': ObjectId('5c804d0ffc4b171803799302'),
 'first_name': 'Magica',
 'last_name': 'De Spell'}
```

Usage avancé des méthodes de modification

- `$inc` et `$dec` incrémentent and décrémentent respectivement une quantité numerique
- `$push` ajoute des éléments aux vecteurs

In [64]:
```
ducks.update_one({'first_name': 'Gyro'}, {'$push': {'hobbies': 'lamps'}})
ducks.find_one({'first_name': 'Gyro'})
```

Out[64]:
```
{'_id': ObjectId('5c804c97fc4b1718037992fc'),
 'first_name': 'Gyro',
 'last_name': 'Gearloose',
 'gender': 'M',
 'first_appearance': 1952,
 'hobbies': ['invention', 'study', 'lamps']}
```

- `$pop` enlève le premier ou le dernier élément d'un vecteur
- `$pull` élimine toutes les occurrences d'un élément dans un vecteur

L'opérateur $

Quand on utilise les vecteurs, `$` devient un opérateur spécial qui fait référence à la première occurrence d'une valeur trouvé en appliquant une requête de mise à jour

In [65]:
```
ducks.update_one({'hobbies': 'lamps'},
                 {'$set': {'hobbies.$': 'robotic lamps'}})
ducks.find_one({'first_name': 'Gyro'})['hobbies']
```

Out[65]:
```
['invention', 'study', 'robotic lamps']
```

Usage avancé de $

L'operateur `$` peut aussi être utilisé pour se déplacer dans une hiérarchie. Supposez que un document contient la valeur suivante pour `hobbies` :

```
[{'name': 'invention', 'day': 'Monday'},
 {'name': 'study', 'day': 'Friday'}]
```

La requête de mise à jour suivante va modifier le jour pour le passe-temps *study*:

```
 {{'hobbies.name': 'study'},
  {'$set': {'hobbies.$.day': 'Tuesday'}}}
```

Upsert

Que va se passer si l'on essaie de mettre à jour un document qui n'existe pas?

- rien...
- ...sauf si le paramètre optionnel `upsert=True` est spécifié quand on appelle `update_one` ou `update_many` : dans ce cas l'effet de la requête est celui d'insérer un nouveau documentn (on appelle ça *insert-upon-update* oou *upsert*).

Éliminer...

- un des documents (s'ils exsistent) identifiés par une requête (sans savoir lequel)

In [66]:
```
ducks.delete_one({'first_name': 'Ludwig'})
```

Out[66]:
```
<pymongo.results.DeleteResult at 0x7f9e4003d988>
```

- tous les documents identifiés par une requête

In [67]:
```
ducks.delete_many({'first_name': 'Ludwig'})
```

Out[67]:
```
<pymongo.results.DeleteResult at 0x7f9e4003dbc8>
```

- tous les documents dans une collection

In [68]:
```
ducks.delete_many({})
```

Out[68]:
```
<pymongo.results.DeleteResult at 0x7f9e4003dc88>
```

- une collection

In [69]:
```
ducks.drop()
```

- une BD

In [70]:
```
client.drop_database('duckburg')

# or

duckburg_db.command('dropDatabase')
```

Out[70]:
```
{'ok': 1.0}
```

Jointures et extensibilité (scalability)

- Dans MongoDB l'extensibilité exploite le **sharding**, qui signifie distribuer et dupliquer les données sur plusierurs ordinateurs
- Cepedant, ça ne permet pas d'effectuer automatiquement des jointures

Jointures faites maison I

On peut utiliser `ObjectId` pour récréer à la main les jointures *un-à-plusieurs*...

In [ ]:
```
donald_id = ducks.find_one({'first_name': 'Donald'})['_id']
for d in ['Huey', 'Dewey', 'Louie']:
    ducks.insert_one({'first_name': d, 'uncle': donald_id})
```

...au prix de devoir effectuer une requête additionnelle (dans ce cas, pour pouvoir récuperer l'oncle).

Jointures faites maison II

On peut implémenter les jointures *plusieurs-à-plusieurs*

- avec des vecteurs

```
ducks.insert_one({
    'first_name': 'Donald',
    'last_name': 'Duck',
    'relatives': [ObjectId('516405b8b356cd6125b74e89'),
                  ObjectId('516405b8b356cd6125b74e8a'),
                  ObjectId('516405b8b356cd6125b74e8b')]})
ducks.find({
    'relatives': ObjectId('516405b8b356cd6125b74e8a')})
```

- ou des documents incorporés

```
ducks.insert_one({
    'first_name': 'Donald',
    'last_name': 'Duck',
    'relatives': [{'first_name': 'Huey'},
                  {'first_name': 'Dewey'},
                  {'first_name': 'Louie'}]})
ducks.find_one({'relatives.first_name': 'Dewey'})
```

Denormalisation

- il s'agit d'une solution possible (faut pas toujours diaboliser la redondance)
- limite critique: la taille maximale pour un document est de 16 MBytes
- (note de côté: le texte entier du Hamlet de Shakespeare n'occupe que 200 Kbytes, donc on peut mémoriser dans un document l'équivalent de environs 80 livres)

- le vrai prix à payer est plutôt celui de devoir effetuer un nombre plus haut de requêtes (comme par exemple quand il faut mettre à jour des données qui sont dupliqués)

Map-reduce

```
resource    date
index       Jan 20 2010 4:30
index       Jan 20 2010 5:30
about       Jan 20 2010 6:00
index       Jan 20 2010 7:00
about       Jan 21 2010 8:00
about       Jan 21 2010 8:30
index       Jan 21 2010 8:30
about       Jan 21 2010 9:00
index       Jan 21 2010 9:30
index       Jan 22 2010 5:00
```

À partir de ces données...

Map-reduce

...nous voulons obtenir

```
resource    year    month   day     count
index       2010    1       20      3
about       2010    1       20      1
about       2010    1       21      3
index       2010    1       21      2
index       2010    1       22      1
```

On va utiliser un algorithme map-reduce où

- pour chaque clé (ressource, date) *map* va émettre la valeur 1
- *reduce* sommes les valeurs

Map-reduce

Pour faire ça, il nous faut tout d'abord insérer les données dans la BD

In [108]:
```
import datetime

ducks.hits.insert_one({'resource': 'index',
                       'date': datetime.datetime(2010, 1, 20, 4, 30, 0, 0)})
ducks.hits.insert_one({'resource': 'index',
                       'date': datetime.datetime(2010, 1, 20, 5, 30, 0, 0)})
ducks.hits.insert_one({'resource': 'about',
                       'date': datetime.datetime(2010, 1, 20, 6, 0, 0, 0)})
ducks.hits.insert_one({'resource': 'index',
                       'date': datetime.datetime(2010, 1, 20, 7, 0, 0, 0)})
ducks.hits.insert_one({'resource': 'about',
                       'date': datetime.datetime(2010, 1, 21, 8, 0, 0, 0)})
ducks.hits.insert_one({'resource': 'index',
                       'date': datetime.datetime(2010, 1, 21, 8, 30, 0, 0)})
ducks.hits.insert_one({'resource': 'about',
                       'date': datetime.datetime(2010, 1, 21, 9, 0, 0, 0)})
ducks.hits.insert_one({'resource': 'index',
                       'date': datetime.datetime(2010, 1, 21, 9, 30, 0, 0)})
_ = ducks.hits.insert_one({'resource': 'index',
                           'date': datetime.datetime(2010, 1, 22, 5, 0, 0, 0)})
```

Map-reduce

Afin de pouvoir lancer un job map-reduce, il faut écrire les fonctions *map* et *reduce* (en javascript!)

In [109]:
```
from bson.code import Code

map = Code('''function() {
  var key = {
    resource: this.resource,
    year: this.date.getFullYear(),
    month: this.date.getMonth(),
    day: this.date.getDate()
  };
  emit(key, {count: 1});
}''')
```

Map-reduce

Afin de pouvoir lancer un job map-reduce, il faut écrire les fonctions *map* et *reduce* (en javascript!)

In [110]:
```
reduce = Code('''function(key, values) {
  var sum = 0;
  values.forEach(function(value) {
     sum += value['count'];
  });
  return {count: sum};
}''')
```

Map-reduce

Le reste est automatiquement géré par MongoDB

In [111]:
```
result = ducks.hits.map_reduce(map, reduce, 'hit_stats')
result
```

Out[111]:
```
Collection(Database(MongoClient(host=['mongodb:27017'], document_class=dict, tz_aware=False, connect=True),
'duckburg'), 'hit_stats')
```

In [112]:
```
for doc in result.find():
    print(doc)
```

```
{'_id': {'resource': 'about', 'year': 2010.0, 'month': 0.0, 'day': 20.0}, 'value': {'count': 1.0}}
{'_id': {'resource': 'about', 'year': 2010.0, 'month': 0.0, 'day': 21.0}, 'value': {'count': 2.0}}
{'_id': {'resource': 'index', 'year': 2010.0, 'month': 0.0, 'day': 20.0}, 'value': {'count': 3.0}}
{'_id': {'resource': 'index', 'year': 2010.0, 'month': 0.0, 'day': 21.0}, 'value': {'count': 2.0}}
{'_id': {'resource': 'index', 'year': 2010.0, 'month': 0.0, 'day': 22.0}, 'value': {'count': 1.0}}
```

In [ ]: