NoSQL Databases

Laszlo SZATHMARY
University of Debrecen
Faculty of Informatics

- using indexes

(last update: 2023-11-15 [yyyy-mm-dd])

2023-2024, 1st semester

Using indexes

Normally, when we are looking for something, we traverse the whole collection (*collection scan*). If we have lots of records, then it will be very slow.

Idea: let's use indexes. In an index, keys are stored in an ordered way. Search can be performed on an index faster, and we can also use efficient searching algorithms, e.g. binary search (MongoDB uses B-trees). A key in the index can be found quickly, and following the pointer next to the index key allows us to retrieve the required document.

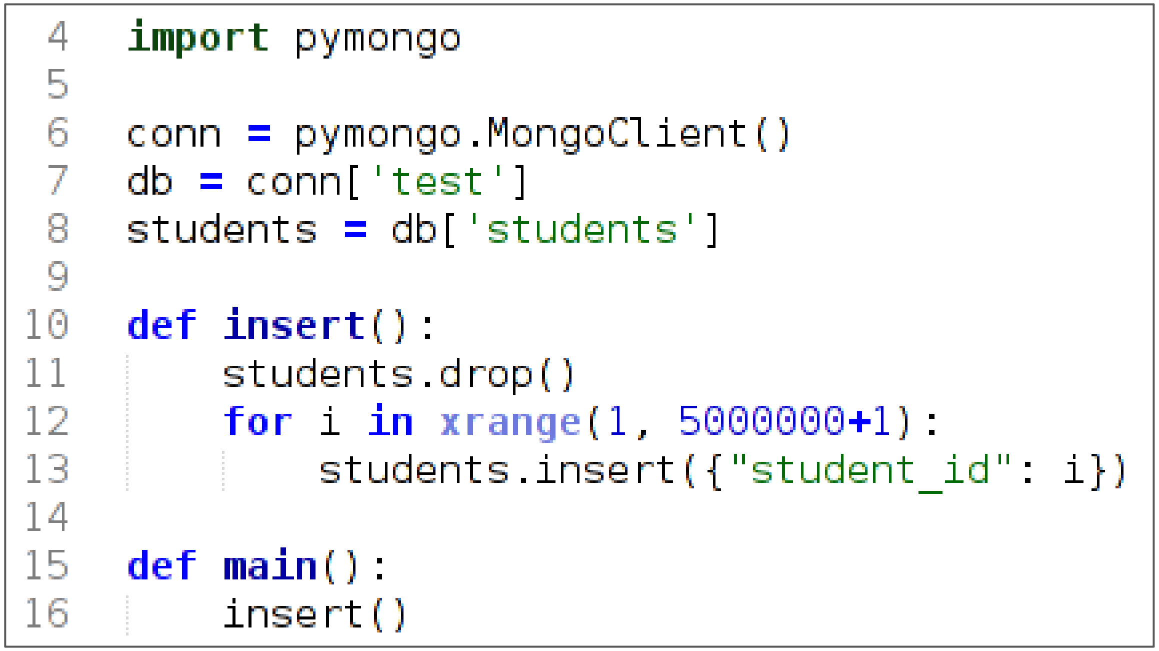
If we have an index, then:

- Reading is very fast.
- Writing will be a bit slower, because after each insert, the index must be updated.

It means that you don't need to put an index on everything! Put an index only on those fields that you query often.

Indexes have a huge impact on the performance of a database system. If indexes are chosen well, then only few queries need to scan the whole collection.

Run the following script. It inserts 5 million documents in the *students* collection:



```python
import pymongo

conn = pymongo.MongoClient()
db = conn['test']
students = db['students']

def insert():
    students.drop()
    for i in xrange(1, 5000000+1):
        students.insert({"student_id": i})

def main():
    insert()
```

Run some queries:

```
> db.students.find({student_id: 4000000})
```

It will take a few seconds. *find()* will find all the occurrences, not just the first one, i.e. it traverses the whole collection.

Further queries:

```
> db.students.findOne({student_id: 10})
```

findOne() goes until the first occurrence. Since this document is there at the beginning of the collection, it will finish very quickly.

```
> db.students.findOne({student_id: 3000000})
```

This will be slow. Although it goes until the first occurrence, this document is not at the beginning of the collection.

creating an index

```
> db.students.createIndex({student_id: 1})
```

It will create an index on the collection *students*. Since we have 5 million documents, building the index will take some minutes. The value „1” means: put in index on the *student_id* field in ascending order (-1 would mean descending order).

When creating a new index, we get some feedback:

```
> db.students.createIndex({student_id: 1})
{
    "createdCollectionAutomatically" : false,
    "numIndexesBefore" : 1,
    "numIndexesAfter" : 2,
    "ok" : 1
}
```

That is: before this command, we already had an index, and after the command we have 2 indexes. Every collection has an index, which is built on the field „_id”.

If we try the function *find()* again, we can notice that the answer arrives immediately.

(I made some tests on my laptop: the query without index took 2114 ms and it dropped to 34 ms when an index was used).

deleting an index

```
> db.students.dropIndex({student_id: 1})
```

Deleting an index is similar to its creation. The only difference is that we need to call the function *dropIndex()*.

discovering indexes

```
> db.system.indexes.find()
```

gives information of all the collections

```
> db.students.getIndexes()
[
    {
        "v" : 1,
        "key" : {
            "_id" : 1
        },
        "name" : "_id_",
        "ns" : "test.students"
    },
    {
        "v" : 1,
        "key" : {
            "student_id" : 1
        },
        "name" : "student_id_1",
        "ns" : "test.students"
    }
]
```

shows the indexes of the given collection only

integrity constraints

Suppose we only want to allow unique values in a field. This constraint can be forced by creating an index on the given field that is specified to be *unique*.
For example, two students cannot have the same ninckname:

```
> db.students.createIndex({nickname: 1}, {unique: true})
```

Trying to insert a duplicate will result in an error.

If the collection already contains duplicates on a field, and we want to put a unique index on it, we will get an error.
With the option *dropDups*, duplicates can be removed. However, it's a dangerous operation! It's not deterministic which document is kept as the unique document!
Usage example:

```
> db.students.createIndex({nickname: 1}, {unique: true, dropDups:true})
```

explain

The function *explain()* gives extra information about a query. From this, for instance, we can figure out which indexes were used in a query.

```
> db.students.find({student_id: 400}).explain()
{
        "cursor" : "BtreeCursor student_id_1",
        "isMultiKey" : false,
        "n" : 1,
        "nscannedObjects" : 1,
        "nscanned" : 1,
        "nscannedObjectsAllPlans" : 1,
        "nscannedAllPlans" : 1,
        "scanAndOrder" : false,
        "indexOnly" : false,
        "nYields" : 1,
        "nChunkSkips" : 0,
        "millis" : 27,
        "indexBounds" : {
                "student_id" : [
                        [
                                400,
                                400
                        ]
                ]
        },
        "server" : "pybox:27017",
        "filterSet" : false
}
```

BasicCursor: no index was used

BtreeCursor: index was used (here, the one that was built on the field *student_id*)

n: number of found records

nscanned: number of scanned documents (here, thanks to the index, the element was found right away)

millis: execution time of the query in msec

index size

Information about the size of a collection and its index(es):

```
> db.students.stats()
{
        "ns" : "test.students",
        "count" : 5000000,
        "size" : 240000208,
        "avgObjSize" : 48,
        "storageSize" : 410312704,
        "numExtents" : 14,
        "nindexes" : 2,
        "lastExtentSize" : 114012160,
        "paddingFactor" : 1,
        "systemFlags" : 1,
        "userFlags" : 1,
        "totalIndexSize" : 288032304,
        "indexSizes" : {
                "_id_" : 162228192,
                "student_id_1" : 125804112
        },
        "ok" : 1
}
```

count: number of documents (here: 5 million elements)

avgObjSize: average document size (here: 48 bytes)

storageSize: size of the collection on the storage (here: 410 MB)

totalIndexSize: sum of the sizes of the indexes (here: 288 MB)

That is: indexes are not for free! You must think it over on which field to put an index.

Full Text Search index

Suppose that we store large texts in the database, and we would like to perform searches on these texts efficiently. In this case, you can put an index of type *text* on the field that contains the text (in the example it’s the field *words*) :

```
> db.sentences.createIndex({‘words’: ‘text’})
```

Query:

```
> db.sentences.find({$text: {$search: ‘prince’}})
```

The search term ‘prince’ can be anywhere in the text, the query will find it.

```
> db.sentences.find({$text: {$search: ‘prince cat key’}})
```

If multiple search terms are given, then these are connected with a logical OR operator. That is: we are looking for those documents, in which one of the following words is present: *prince* or *cat* or *key*.

Conclusion

- We saw how to use MongoDB, which is the most popular NoSQL database management system.
- We saw that we can work with (JSON) documents too, not only with tables (see SQL).
- We saw how to use the command-line shell.
- There are also GUI administration tools, e.g. Studio 3T.
- We saw how to use MongoDB from an application (Python).
- MongoDB is an ideal choice for rapid prototyping, when we don’t know the database schema in advance, and the structure of the database evolves during the development process (dynamic schemas).
- Who uses MongoDB? See https://www.mongodb.com/who-uses-mongodb