Dig Data Management Assignment 5

信科四 郑元嘉 1800920541

In this report, we're about to carry out three identical tasks on the same data `student.csv` via three various databases, `Redis` , `MongoDB` and `Neo4j` .
To get it straight for our quantity analysis of consumed time, we declare the three tasks here :

1. Insertion task: insert all data into the empty corresponding databases
2. Update task: update all data with all `sex` -> `'F'` and all `Mjob` -> `'teacher'`
3. Search task: find all original data where `sex` == `'F'` , `paid` == `'no'` and `internet` == `'yes'` , and then return all matched results

Task1 - Redis

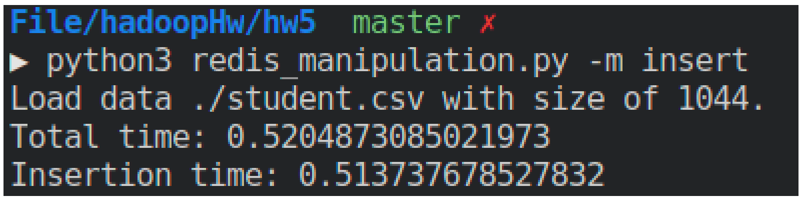
- **Insertion Task**

  (Partial source code)

```python
def insert_bulk(file_path, redis_host, redis_port, redis_db_index):
    global timestamp
    client = redis.Redis(host=redis_host, port=redis_port, db=redis_db_inde

    data = pd.read_csv(file_path, delimiter=';')
    print('Load data {} with size of {}.'.format(file_path, data.shape[0]))

    timestamp.stamp('insert')

    for i, row in data.iterrows():
        mapping = dict(row)
        del mapping['id']
        client.hmset(row['id'], mapping)

    timestamp.stamp('insert')
```

  By the means of Redis, we can only insert one datum each time by function `hmset()` , which means hardly could we run a 'bulk' task.
  Here's some point worth attention. The csv file `student.csv` applies an abnormal delimiter `;` instead of `,` , so we need to set parameter `delimiter` of function `read_csv()` .

Consumed Time :



```
File/hadoopHw/hw5  master ✗
▶ python3 redis_manipulation.py -m insert
Load data ./student.csv with size of 1044.
Total time: 0.5204873085021973
Insertion time: 0.513737678527832
```

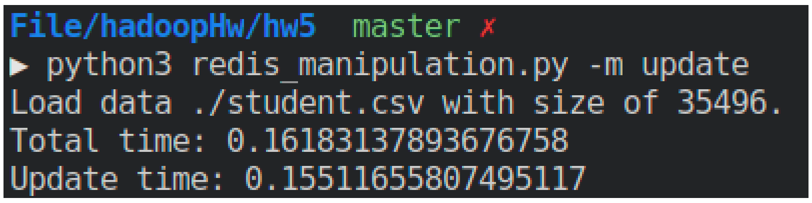
- **Update Task**

  (Partial source code)

```python
def update_bulk(file_path, redis_host, redis_port, redis_db_index):
    global timestamp
    client = redis.Redis(host=redis_host, port=redis_port, db=redis_db_inde

    data = pd.read_csv(file_path, delimiter=';')
    print('Load data {} with size of {}.'.format(file_path, data.size))

    timestamp.stamp('update')

    new_job, new_sex = 'teacher', 'F'
    for i, row in data.iterrows():
        mapping = {"Mjob": new_job, "sex": new_sex}
        client.hmset(row['id'], mapping)

    timestamp.stamp('update')
```

Similar to insertion task, we can only update single one datum each time merely by function `hmset()` .

Consumed Time :



```
File/hadoopHw/hw5  master ✗
▶ python3 redis_manipulation.py -m update
Load data ./student.csv with size of 35496.
Total time: 0.16183137893676758
Update time: 0.15511655807495117
```

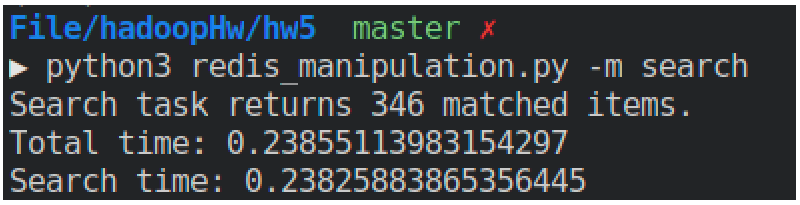
- **Search Task**

  (Partial source code)

```python
def search_bulk(file_path, redis_host, redis_port, redis_db_index):
    global timestamp
    client = redis.Redis(host=redis_host, port=redis_port, db=redis_db_inc

    timestamp.stamp('search')
    sex_filter, paid_filter, internet_filter = 'F', 'no', 'yes'
    output = []

    keys = client.keys('*')
    for key in keys:
        datum = client.hgetall(key)
        if datum['sex'] == sex_filter and datum['paid'] == paid_filter anc
            output.append(datum)

    print('Search task returns {} matched items.'.format(len(output)))

    timestamp.stamp('search')
```

Search task on Redis is quite unfriendly and inelegant. We query all keys first, and then query the data according to the keys, and ultimately we filter every datum by comparing the values of selected fields. On the whole, the filtering task is run on the client side which would potentially cause unnecessary and redundant transmission of data. Yet it can release the burden of Redis. It could be a trade-off.

Consumed Time :



```
File/hadoopHw/hw5  master ✗
▶ python3 redis_manipulation.py -m search
Search task returns 346 matched items.
Total time: 0.23855113983154297
Search time: 0.23825883865356445
```

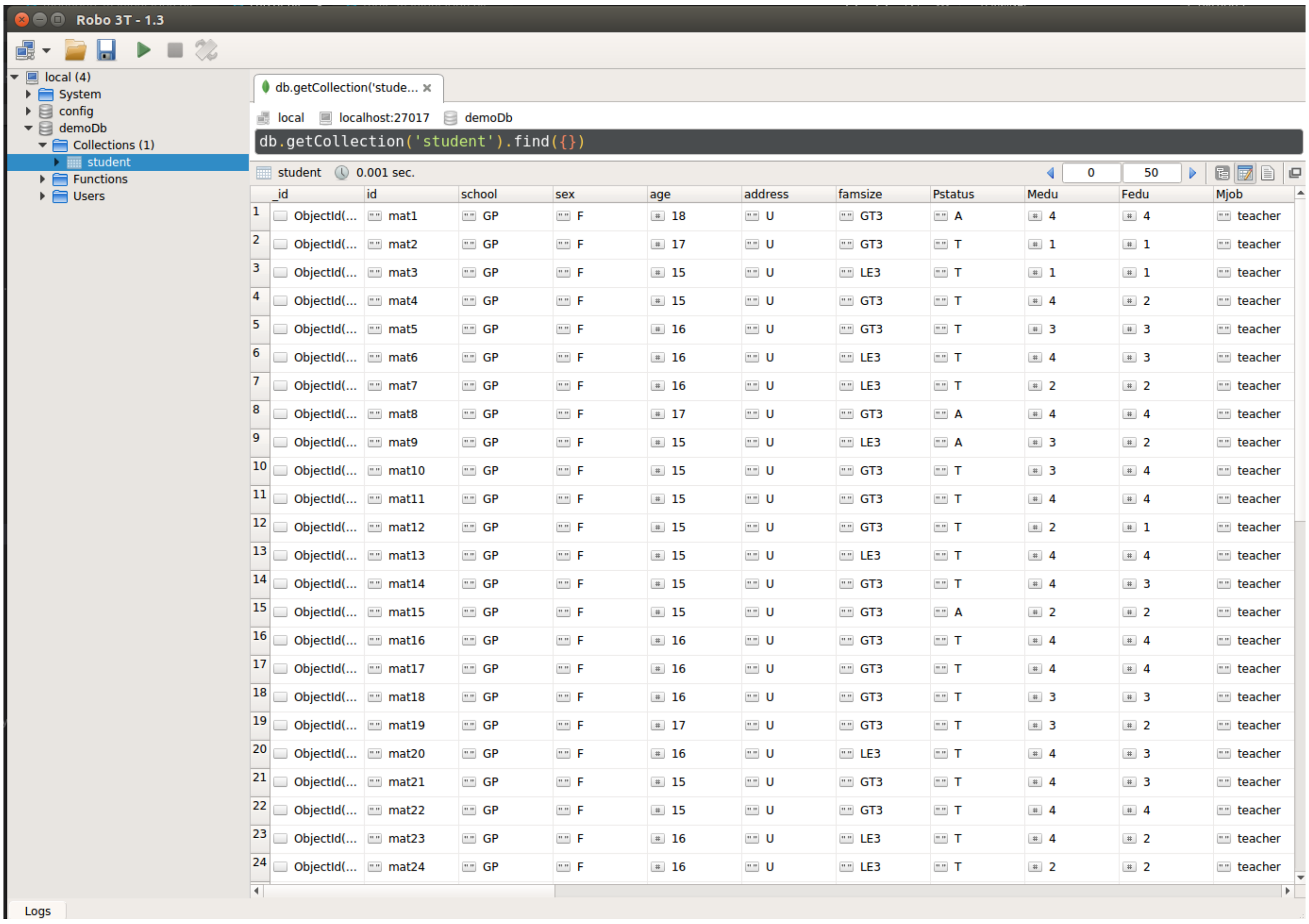
Task2 - MongoDB

- **Insertion Task**

  (Partial source code)

```
def insert_bulk(file_path, db_host, db_port, db_name, db_col_name):
    client = MongoClient(db_host, db_port)

    db = client[db_name]
    collection = db[db_col_name]

    data = pd.read_csv(file_path, delimiter=';')
    print('Load data {} with size of {}.'.format(file_path, data.shape[0]))

    timestamp.stamp('insert')

    bulk = []
    for i, row in data.iterrows():
        datum = dict(row)
        bulk.append(datum)

    collection.insert_many(bulk)

    timestamp.stamp('insert')
```

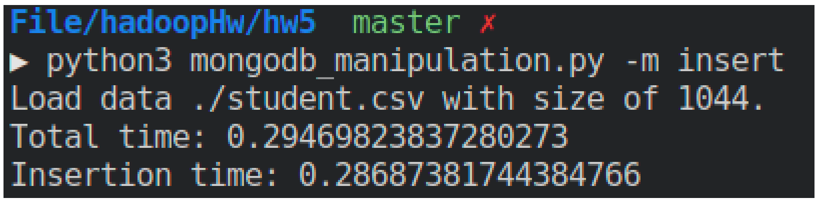
First of all, we create one collection named `student` in our database. Unlike Redis, MongoDB supports bulk operation which reduce the waste of transmission time in a leap. We read the file `student.csv` , convert them into `dictionary` type, and call function `insert_many()` to be done with this task with ease.
Plus, we can use `Robot 3T` – a GUI tool for MongoDB – to have a look at the data.



```
db.getCollection('student').find({})
```

	_id	id	school	sex	age	address	famsize	Pstatus	Medu	Fedu	Mjob
1	ObjectId(...	mat1	GP	F	18	U	GT3	A	4	4	teacher
2	ObjectId(...	mat2	GP	F	17	U	GT3	T	1	1	teacher
3	ObjectId(...	mat3	GP	F	15	U	LE3	T	1	1	teacher
4	ObjectId(...	mat4	GP	F	15	U	GT3	T	4	2	teacher
5	ObjectId(...	mat5	GP	F	16	U	GT3	T	3	3	teacher
6	ObjectId(...	mat6	GP	F	16	U	LE3	T	4	3	teacher
7	ObjectId(...	mat7	GP	F	16	U	LE3	T	2	2	teacher
8	ObjectId(...	mat8	GP	F	17	U	GT3	A	4	4	teacher
9	ObjectId(...	mat9	GP	F	15	U	LE3	A	3	2	teacher
10	ObjectId(...	mat10	GP	F	15	U	GT3	T	3	4	teacher
11	ObjectId(...	mat11	GP	F	15	U	GT3	T	4	4	teacher
12	ObjectId(...	mat12	GP	F	15	U	GT3	T	2	1	teacher
13	ObjectId(...	mat13	GP	F	15	U	LE3	T	4	4	teacher
14	ObjectId(...	mat14	GP	F	15	U	GT3	T	4	3	teacher
15	ObjectId(...	mat15	GP	F	15	U	GT3	A	2	2	teacher
16	ObjectId(...	mat16	GP	F	16	U	GT3	T	4	4	teacher
17	ObjectId(...	mat17	GP	F	16	U	GT3	T	4	4	teacher
18	ObjectId(...	mat18	GP	F	16	U	GT3	T	3	3	teacher
19	ObjectId(...	mat19	GP	F	17	U	GT3	T	3	2	teacher
20	ObjectId(...	mat20	GP	F	16	U	LE3	T	4	3	teacher
21	ObjectId(...	mat21	GP	F	15	U	GT3	T	4	3	teacher
22	ObjectId(...	mat22	GP	F	15	U	GT3	T	4	4	teacher
23	ObjectId(...	mat23	GP	F	16	U	LE3	T	4	2	teacher
24	ObjectId(...	mat24	GP	F	16	U	LE3	T	2	2	teacher

Consumed Time :



```
File/hadoopHw/hw5  master ✗
▶ python3 mongodb_manipulation.py -m insert
Load data ./student.csv with size of 1044.
Total time: 0.29469823837280273
Insertion time: 0.286873817443847 66
```

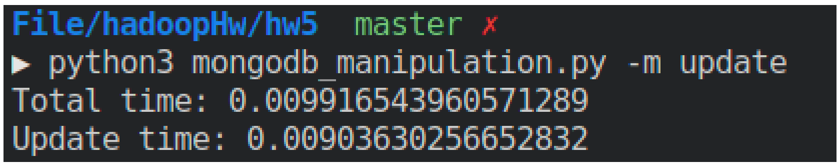
- **Update Task**

  (Partial source code)

```
def update_bulk(file_path, db_host, db_port, db_name, db_col_name):
    client = MongoClient(db_host, db_port)

    db = client[db_name]
    collection = db[db_col_name]

    global timestamp
    timestamp.stamp('update')

    new_job, new_sex = 'teacher', 'F'
    res = collection.update_many({}, {'$set': {'Mjob': new_job, 'sex': nev

    timestamp.stamp('update')
```

The story is almost the same. MongoDB supports bulk operation of update (modification). Thus we're allowed to call function `update_many()` as soon as prepared for our conversion preprocess. Because the client side only sends the filter query and wanted modification, all operations such as retrieving and filtering are undergone in MongoDB which lessens the workload on client sides and makes transmission more efficient.

Consumed Time :



```
File/hadoopHw/hw5  master ✗
▶ python3 mongodb_manipulation.py -m update
Total time: 0.009916543960571289
Update time: 0.009036302566528320
```

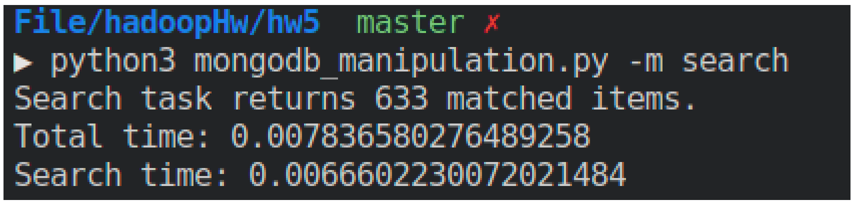
- **Search Task**

  (Partial source code)

```
def search_bulk(file_path, db_host, db_port, db_name, db_col_name):
    client = MongoClient(db_host, db_port)

    db = client[db_name]
    collection = db[db_col_name]

    global timestamp
    timestamp.stamp('search')

    sex_filter, paid_filter, internet_filter = 'F', 'no', 'yes'
    cursor = collection.find({'sex': sex_filter, 'paid': paid_filter, 'inte
    output = list(cursor)

    print('Search task returns {} matched items.'.format(len(output)))

    timestamp.stamp('search')
```

Bulk search task can be carried out via mere function `find()` supported by MongoDB. It's noteworthy that the return of the function `find()` is a class `Cursor` rather than the actual data, so we need to call extra functions triggering conversion like `list()` to retrieve the actual data if needed. I dig into the source code of `pymongo` packages and find that `Cursor` contains no actual data in RAM while when we call extra functions like `list()` on `Cursor` instance, `pymongo` would send another request to MongoDB to retrieve the actual ones which is a clever and time-saving design in most cases.

Consumed Time :



```
File/hadoopHw/hw5  master ✗
▶ python3 mongodb_manipulation.py -m search
Search task returns 633 matched items.
Total time: 0.007836580276489258
Search time: 0.0066602230072021484
```

Task3 - Neo4j

- **Insertion Task**

  (Partial source code)

```python
def insert_bulk(file_path, uri, acc, pwd):
    driver = GraphDatabase.driver(uri, auth=(acc, pwd))

    data = pd.read_csv(file_path, delimiter=';')
    print('Load data {} with size of {}.'.format(file_path, data.shape[0]))

    global timestamp
    timestamp.stamp('insert')

    statement = 'CREATE (:student{{{}}})'.format(','.join(['{v}: {{{v}}}'.f

    with driver.session() as session:
        for i, row in data.iterrows():
            session.run(statement, dict(row))

    timestamp.stamp('insert')
```

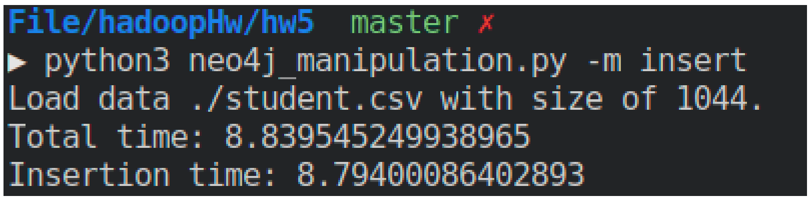
All manipulation of Neo4j is gone through scripts on Python. Due to lack of bulk insertion, we need to iteratively run the `CREATE` operation to insert all data. Plus, the rules of script format are kind of unfriendly and flexible which causes redundant programming codes for Python than other databases APIs.

Of course, you may attempt to write all `CREATE` clauses in one script to run which would reduce the times of transmission. The partial source code is like

```python
def insert_bulk(file_path, uri, acc, pwd):
    driver = GraphDatabase.driver(uri, auth=(acc, pwd))

    data = pd.read_csv(file_path, delimiter=';')

    print('Load data {} with size of {}.'.format(file_path, data.shape[0]

    global timestamp
    timestamp.stamp('insert')

    with driver.session() as session:
        s = ''
        for i, row in data.iterrows():
            # print(dict(row))
            statement = 'CREATE (:student{{{}}}) '.format(','.join(['{v}:
            s += statement
        session.run(s)

    timestamp.stamp('insert')
```

However, much to our surprise, the consumed time increases badly somehow. Here's a guess - one script line would create only one process in Neoj4 to finish all clauses which means the process is overwhelmed by a great amount of memory and operation sources.

Consumed Time :



```
File/hadoopHw/hw5  master ✗
▶ python3 neo4j_manipulation.py -m insert
Load data ./student.csv with size of 1044.
Total time: 8.839545249938965
Insertion time: 8.79400086402893
```

- **Update Task**

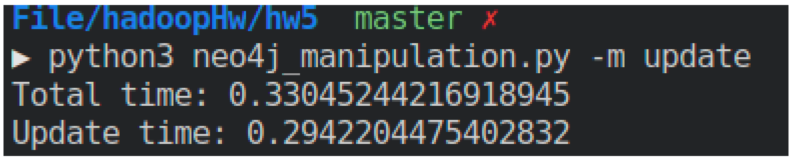
  (Partial source code)

```
def update_bulk(file_path, uri, acc, pwd):
    driver = GraphDatabase.driver(uri, auth=(acc, pwd))

    global timestamp
    timestamp.stamp('update')

    new_job, new_sex = 'teacher', 'F'
    statement = 'MATCH (s:student{}) SET s.Mjob = {Mjob} SET s.sex = {sex}

    with driver.session() as session:
        res = session.run(statement, Mjob=new_job, sex=new_sex)
        data = list(res.records())

    timestamp.stamp('update')
```

Neo4j supports bulk update in some extent. The only information in the script is our matching conditions and the new values which is highly similar to MongoDB.
Note that the resulting return is not a friendly object we're familiar with, thus we can access

the values of the returned data via some tricks.

Consumed Time :



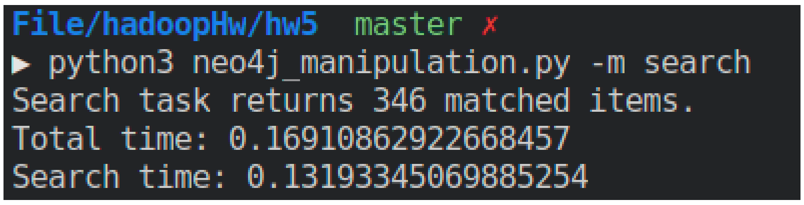
```
File/hadoopHw/hw5  master ✗
▶ python3 neo4j_manipulation.py -m update
Total time: 0.33045244216918945
Update time: 0.2942204475402832
```

- **Search Task**
- (Partial source code)

```
def search_bulk(file_path, uri, acc, pwd):
    driver = GraphDatabase.driver(uri, auth=(acc, pwd))

    global timestamp
    timestamp.stamp('search')

    sex_filter, paid_filter, internet_filter = 'F', 'no', 'yes'
    statement = 'MATCH (s:student{sex: {sex}, paid: {paid}, internet: {inte

    with driver.session() as session:
        res = session.run(statement, sex=sex_filter, paid=paid_filter, inte
        data = list(res.records())

        print('Search task returns {} matched items.'.format(len(data)))

    timestamp.stamp('search')
```

Neo4j provides `MATCH` clause to run our search task as well. We call `MATCH` clause and `RETURN` clause to retrieve the results. The call function `.records()` on the returned result to access the matched data.

Consumed Time :



```
File/hadoopHw/hw5  master ✗
▶ python3 neo4j_manipulation.py -m search
Search task returns 346 matched items.
Total time: 0.16910862922668457
Search time: 0.13193345069885254
```

Consumed Time Analysis

	Insertion Task	Update Task	Search Task
Redis	0.5137	0.1551	0.2383

	Insertion Task	**Update Task**	**Search Task**
MongoDB	**0.2869**	**0.0090**	**0.0067**
Neo4j	8.7940	0.2942	0.1320

- Insertion Task
  The rare support of bulk insertion mainly contributes to the prominent performance of MongoDB. The simplified structure of storage helps Redis surpass Neo4j while the more complicated design of structure enables Neo4j to manipulate a rather more complex searching (matching) task.
- Update Task
  The cost of update is quite similar to the one of insertion, so the ranking is the same as the one for insertion task. It's worth attention that the consumed time of Neo4j shrinks the most significantly which shows the overhead of creating one node is a heavy workload.
- Search Task
  The relative ranking between Redis and Neo4j switches owing to lack of bulk research. Plus, the filtering task is run on the client side in the case of Redis, so we need to retrieve all data back and scan through every single one which is such an inefficiency in transmission.

Last but not least, keep in mind that we only go through extremely limited experiments to analyze which gives a huge bias towards MongoDB and doesn't give a guarantee of good performance in any cases.