



华中科技大学

大数据管理实验报告

姓 名：涂畅
学 院：计算机科学与技术学院
专 业：计算机科学
班 级：201803 班
学 号：X2021H1002

分数	
教师签名	

2021 年 6 月 29 日

教师评分页

子目标	子目标评分
1	
2	
3	
4	
5	
6	
总分	

目 录

1 课程任务概述.....	1
2 MongoDB 实验任务.....	2
2.1 任务要求.....	2
2.2 完成过程.....	2
2.3 任务小结.....	5
3 Neo4j 的实验.....	6
3.1 任务要求.....	6
3.2 完成过程.....	6
3.3 任务小结.....	12
4 验证 MYSQL 在 InnoDB 存储引擎下 MVCC 多版本并发控制.....	13
4.1 任务要求.....	13
4.2 完成过程.....	13
4.3 任务小结.....	15
5 程序设计.....	16
5.1 任务要求.....	16
5.2 完成过程.....	16
5.3 任务小结.....	19
6 课程总结.....	20
附录.....	21

1 课程任务概述

第一个任务和第二个任务主要是在 MongoDB 和 Neo4j 中进行实验操作，详情请见任务书。

第三个任务是验证 MySQL 在 InnoDB 存储引擎下的 MVCC 多版本并发控制，最后一个任务是实现一个功能用来在 MySQL 和 Neo4j 中添加一个新的用户，需要满足事务的一致性，保证在添加的用户在上述两个数据库中状态一致。

2 MongoDB 实验任务

2.1 任务要求

首先进行 MongoDB 的安装，然后在安装好的软件下和设定好的环境下进行 MongoDB 实验操作。

2.2 完成过程

2.2.1 查询 user 集合从第二条记录开始的三条记录。

```
db.user.find().limit(3).skip(1).pretty()
```

用 `find()` 进行查询，`skip(1)` 跳过第一条记录，`limit(3)` 显示第二条记录开始的三条记录（限制三条记录），`pretty()` 使得语句的格式整齐。

2.2.2 查询 user_id 是 q-0Um-wv45DcHdNIyhTUOw 的 user。

```
db.user.find({user_id:"q-0Um-wv45DcHdNIyhTUOw"}).pretty()
```

2.2.3 查询 fans 不小于 50 且 review_count 大于 10 的 user 限制 10 条

```
db.user.find({'fans':{'$gte':50},$and:[{'review_count':{'$gt':10}}]}).limit(10).pretty()
```

用 `find()` 进行查询，`'fans':{'$gte':50}` 不小于 50 也就是大于等于 50，`'review_count':{'$gt':10}` review_count 大于 10，`limit(10)` user 限制 10 条。

2.2.4 查询所有名字为 rashmi 的用户名，不区分大小写。

```
db.user.find({'name':{'$regex':'rashmi',$options:'$i'}}).pretty()
```

`$options:'$i'` 不区分大小写，`$regex:'rashmi'` 为查询中的模式匹配字符串提供正则表达式功能，查询所有名字为 `rashmi` 的用户名。

2.2.5 查询同时是 user_id 为 xjrUcid6Ymq0DoTJELkYyw 和

6yCWjFPtp_AD4x93WAwmnw 的朋友的用户，限制 20 条。

```
db.user.find({'friends':{'$all':['xjrUcid6Ymq0DoTJELkYyw',"6yCWjFPtp_AD4x93WAwmnw"]}},{'user_id':1});
```

`$all` 打印出所有共同用户。

2.2.6 统计 user 一共有多少条数据。

```
db.user.count()
```

2.2.7 计用户名重名的次数，返回姓名重复的次数和对应的姓名，按照次数降序排列。

```
db.user.aggregate({"$project":{"name":1,"user_id":1}},
{"$group":{"_id":"$name","counts":{"$sum":1}}},
{"$sort":{"counts":-1}})
```

用 `aggregate` 的函数整合，`"counts":{"$sum":1}` 重复的次数求和，`"$sort":{"counts":-1}` 求和降序排列。

2.2.8 查询评价对应的 user 的信息【提示 2 个表关联查询\$lookup】

```
db.user.aggregate([{$lookup:{from:"review",
localField:"user_id", //from user
```

```
foreignField:"user_id",           //review
as:"review"}},
{$unwind:"$review"}})
```

Join 两个 table，入手点是 `user_id`，首先先建立 index 使得查询速度更快。

2.2.9 使用 explain 看 `db.user.find({user_id: "z4xZolh3zzBmPEW6RuGuJQ"})` 执行计划，进行查询优化

①建立 index 优化查询速度：

```
db.user.createIndex({user_id:1})
```

②在终端或 MongoDB Compass 里面输入：

```
db.user.find({user_id:"z4xZolh3zzBmPEW6RuGuJQ"}).explain()
```

③然后我们可以看到在

```
db.user.find({user_id:'q-0Um-wv45DcHdNIyhTUOw'}).explain().queryPlanner.winningPlan
```

出来的结果中 stage 是“IXSCAN”也就是进行了 index search scan，我们还可以用 `db.user.find({user_id:'z4xZolh3zzBmPEW6RuGuJQ'}).explain("executionStats")` 中结果给出的 `totalKeysExamined` 查看用了多少个 index，在这里我们用了一个。

2.2.10 创建一个 review 的子集合 reviewSub(取 review 的前一百万条数据)，对评论的内容建立全文索引，进行查询评价的内容中包含关键词 **ambience good lively** 的评价

```
db.reviewSub.createIndex({text:1})
```

```
db.reviewSub.find( { 'text':/ambience good lively/i})
```

首先我们还是先建立索引以后再查询语句使得查询速度更快。

2.2.11 体验文档的结构是自由的【同一个集合中可以存在异构的文档】

首先我们向 user 里面插入一个 array list（异构文档），

```
> use yelp
< 'switched to db yelp'
> db.user.insert( [
  { _id: 11, item: "pencil", qty: 50, type: "no.2" },
  { item: "pen", qty: 20 },
  { item: "eraser", qty: 25 }
])
< 'DeprecationWarning: Collection.insert() is deprecated. Use insertOne, insertMany or bulkWrite.'
```

图 2.1

然后用 find 查询此异构文档，证明能够插入数据库中

```
< { acknowledged: true,
  insertedIds:
    { '0': 11,
      '1': ObjectId("60cf0c55e6374518c239687a"),
      '2': ObjectId("60cf0c55e6374518c239687b") } }
> db.user.find({item:"pen"})
< [ { _id: ObjectId("60cf0c55e6374518c239687a"),
  item: 'pen',
  qty: 20 }
```

图 2.2

2.2.12 查询距离店铺 6016c6b4af81085b0f2183d5(object id) 10 米以内的商家

```
db.business.find(  
  { loc: { $geoWithin: { $center: [ [ -115.140685, 36.169993 ], 10 ] } } }  
)
```

想象一个半径为 10 的圆,在这个圆里面进行搜索查询商家,使用函数\$geoWithin。

2.2.13 实时统计一个网站的访问次数,可以这样设计

db.page_counter.insertOne({cnt: 0}), 如何更新这个字段, 如果使用 MySQL 怎么实现, 性能怎么样, 如何优化, 和 MongoDB 对比的情况如何

MongoDB 比 MySQL 要更快一些因为 NoSQL 增加字段不用改表的结构。优化方案: 删除现有索引, 一旦添加越来越多的索引, 向 MySQL 表插入数据的速度会变慢。批量插入数据, 将许多小操作合并为一个操作能够优化插入速度。

2.2.14 使用 map reduce 计算每位用户做出评价的平均分 (建议在 reviewSub 集合上做, review 过于大)。

①首先在终端里面建立索引: db.reviewSub.createIndex({user_id:1,stars:1})

```
> db.createIndex({user_id:1,stars:1});  
uncaught exception: TypeError: db.createIndex is not a function :  
@(shell):1:1  
> db.reviewSub.createIndex({user_id:1,stars:1});  
{  
  "numIndexesBefore" : 5,  
  "numIndexesAfter" : 5,  
  "note" : "all indexes already exist",  
  "ok" : 1  
}
```

图 2.3

②其次用 MapReduce 计算数据保存为 average_stars

```
var map=function(){emit(this.user_id,this.stars)};  
var reduce=function(user_id,stars){return Array.avg(stars)};  
db.reviewSub.mapReduce(map,reduce,{out:"avgstars"});  
  
> use yelp  
switched to db yelp  
> var map=function(){emit(this.user_id,this.stars)};  
uncaught exception: SyntaxError: illegal character :  
@(shell):1:49  
> var map=function(){emit(this.user_id,this.stars)};;  
> var reduce=function(user_id,stars){return Array.avg(stars)};;  
> db.reviewSub.mapReduce(map,reduce,{out:"average_stars"});  
{ "result" : "average_stars", "ok" : 1 }
```

图 2.4

③最后可查看用户 id 和平均值

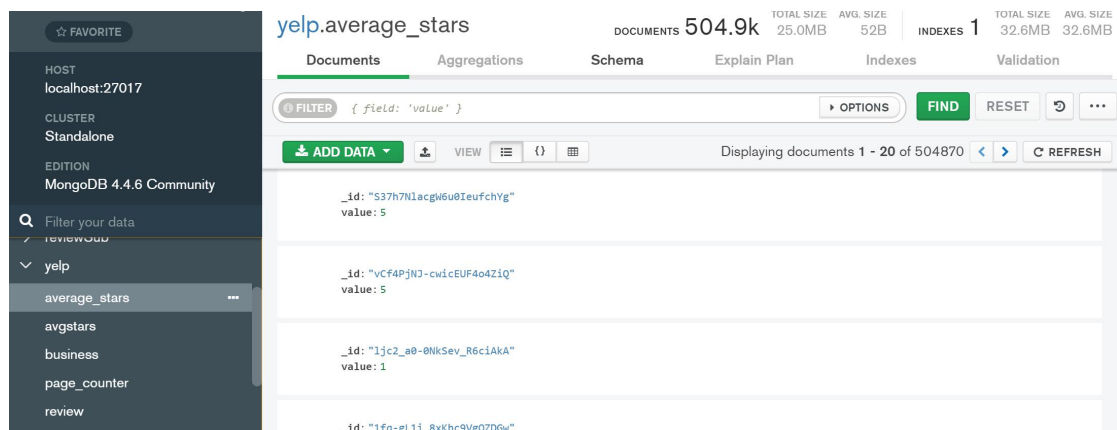


图 2.5

2.3 任务小结

首先第一个遇到的就是查询速度的问题，这个可以通过建立索引的方式来解决。其次在 MongoDB compass 运行 `this.user` 显示错误，在终端运行解决了此问题。

3 Neo4j 的实验

3.1 任务要求

首先进行 Neo4j 的安装，然后在安装好的软件下和设定好的环境下进行 Neo4j 的实验操作

3.2 完成过程

3.2.1 使用 `import.bat` 命令把数据导入到 `neo4j` 中

在终端把数据导入即可，见指导书。

3.2.2 查询标签是 `UserNode` 的节点，限制 10 个

```
MATCH (n:UserNode)
```

```
RETURN n
```

```
LIMIT 10
```

返回的是节点的数量，limit 限制节点为十个。

总查询时间：13ms

3.2.3 查询名字是 `Dwi` 的节点。

```
MATCH (n:UserNode {name: 'Dwi'})
```

```
RETURN n
```

总查询时间：13ms

3.2.4 查询 `userid` 是 `q-0Um-wv45DcHdNIyhTUOw` 发表的评价。

```
MATCH (u:UserNode)-[r:Review]->(rn:ReviewNode)
```

```
WHERE u.userid='q-0Um-wv45DcHdNIyhTUOw'
```

```
RETURN rn.stars
```

这里的关系是用户进行了 review 然后在 review 节点中查找评价。

总查询时间：53ms

3.2.5 查询被 `userid` 是 `q-0Um-wv45DcHdNIyhTUOw` 评价的商家的名称。

```
MATCH (u:UserNode)-[r:Review]->(rn:ReviewNode)
```

```
WHERE u.userid='q-0Um-wv45DcHdNIyhTUOw'
```

```
WITH rn
```

```
MATCH (rn:ReviewNode)-[rd:Reviewed]->(b:BusinessNode)
```

```
WHERE exists(rn.reviewid)
```

```
RETURN b.name
```

每一次 review 都会有一个 reviewid, 通过这个 reviewid 我们可以找到商家的名字。

总查询时间：17ms

3.2.6 查询评论过 `businessid` 为 `_9kpamfhbsG5NVNpP-ED3w` 的 `user`。

```
MATCH (rn:ReviewNode)-[rd:Reviewed]->(b:BusinessNode)
```

```
WHERE b.businessid="_9kpamfhbsG5NVNpP-ED3w"
```

```
WITH rn
```

```
MATCH (u:UserNode)-[r:Review]->(rn:ReviewNode)
```

```
WHERE exists(rn.reviewid)
```

```
RETURN u.userid
```

这里是反向的一个查询，还是通过 reviewid。

总查询时间：251ms

3.2.7 查询用户，按照粉丝数降序排序限制 20 条即可。

```
CREATE INDEX index_fans FOR (u:UserNode) ON (u.fans)
```

```
MATCH (u:UserNode)
```

```
RETURN u.fans
```

```
ORDER BY u.fans DESC
```

```
LIMIT 20
```

找到 UserNode 里面的粉丝降序排列（DESC）以后用 limit 限制返回数量即可。

总查询时间：7022ms

3.2.8 使用 where 查询城市在 North Las Vegas 的商家名字。

```
MATCH (b:BusinessNode)-[:IN_CITY]->(c:CityNode)
```

```
WHERE c.city="North Las Vegas"
```

```
RETURN b.name
```

通过商家所在的城市进行反向查询。

总查询时间：370ms

3.2.9 查询 userid 是 q-0Um-wv45DcHdNIyhTUOw 的朋友数量。

```
MATCH (u:UserNode
```

```
{userid:"q-0Um-wv45DcHdNIyhTUOw"})-[:HasFriend]->(friend:UserNode)
```

```
RETURN size(collect(friend.name))
```

这个关系就是用户有朋友，收集所有朋友返回数组大小即可。

总查询时间：10ms

3.2.10 查询 userid 是 q-0Um-wv45DcHdNIyhTUOw 的朋友的名字，以一个 list 的形式返回。

```
MATCH (u:UserNode
```

```
{userid:"q-0Um-wv45DcHdNIyhTUOw"})-[:HasFriend]->(friend:UserNode)
```

```
RETURN collect(friend.name)
```

用户有朋友，收集所有朋友返回数组即可。

总查询时间：4ms

3.2.11 使用 with 传递查询结果到后续的处理。查询出 Dwi 的朋友（直接相邻）分别都有多少位朋友（直接相邻）（Dwi 的 userid 为 q-0Um-wv45DcHdNIyhTUOw）。

```
MATCH (u0:UserNode
```

```
{userid:"q-0Um-wv45DcHdNIyhTUOw"})-[:HasFriend]->(u1:UserNode)
```

```
WITH u1
```

```
MATCH (u1)-[:HasFriend]->(u3:UserNode)
```

```
RETURN u1.name, size(collect(u3.name))
```

先写出用户关系，返回列表包含姓名和朋友数量。

总查询时间：98ms

3.2.12 统计商家在每个城市的分布情况，返回城市中商家的数量和城市名称。

```
MATCH (b:BusinessNode)-[i:IN_CITY]->(c:CityNode)
```

```
WHERE b.city=c.city
```

```
RETURN c.city, size(collect(b.name))
```

总查询时间：10ms

3.2.13 统计包含类别是 **Asian Fusion** 的商家的城市，按照每个城市中商家的数量降序排序，返回城市和相应城市中的商家的数量。

```
MATCH (b:BusinessNode)-[i:IN_CATEGORY]->(c:CategoryNode{category:"Asian Fusion"})
```

```
WITH b
```

```
MATCH (b:BusinessNode)-[i:IN_CITY]->(c:CityNode)
```

```
WHERE b.city=c.city
```

```
RETURN c.city, size(collect(b.name))
```

总查询时间：125ms

3.2.14 统计用户名重名的次数，返回姓名重复的次数和对应的姓名，按照次数降序排列。

```
CREATE INDEX index_username FOR (u:UserNode) ON (u.userid)
```

```
match (u:UserNode)
```

```
return u.name,count(*) order by count(*) desc
```

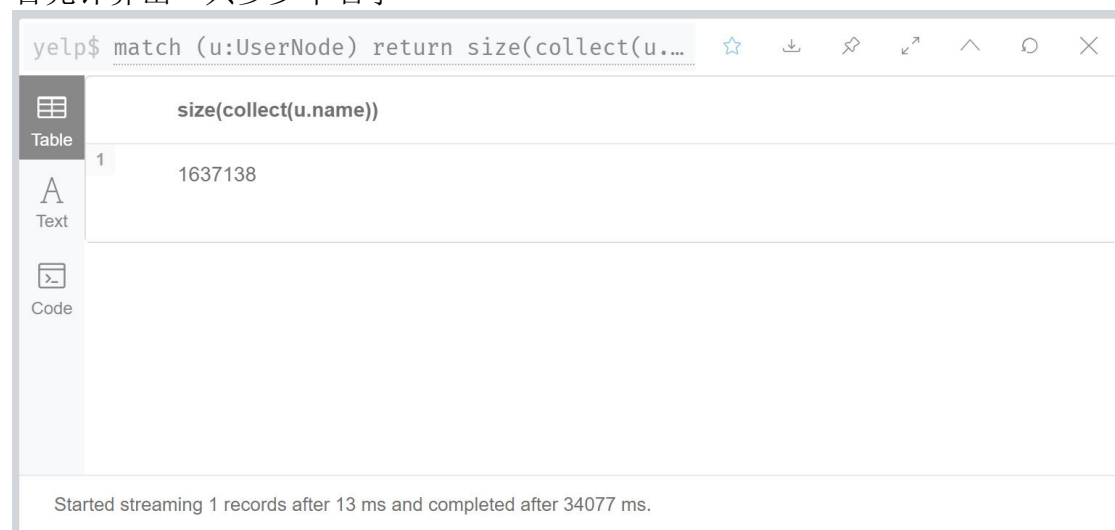
```
limit 3
```

总查询时间：6863ms

3.2.15 统计每个用户名热度（名字的重复的次数在所有的不同的用户名中的占比），返回热度和用户名。

```
CREATE INDEX index_name FOR (u:UserNode) ON (u.name)
```

首先计算出一共多少个名字



The screenshot shows a query execution interface. At the top, the query is: `yelp$ match (u:UserNode) return size(collect(u.name))`. Below the query, there is a table view. The table has one column named `size(collect(u.name))` and one row with the value `1637138`. The table is labeled `Table` and `1`. Below the table, there is a text area labeled `Text` and a code area labeled `Code`. At the bottom, a status message says: `Started streaming 1 records after 13 ms and completed after 34077 ms.`

size(collect(u.name))
1637138

图 3.1

```
match (u:UserNode)
```

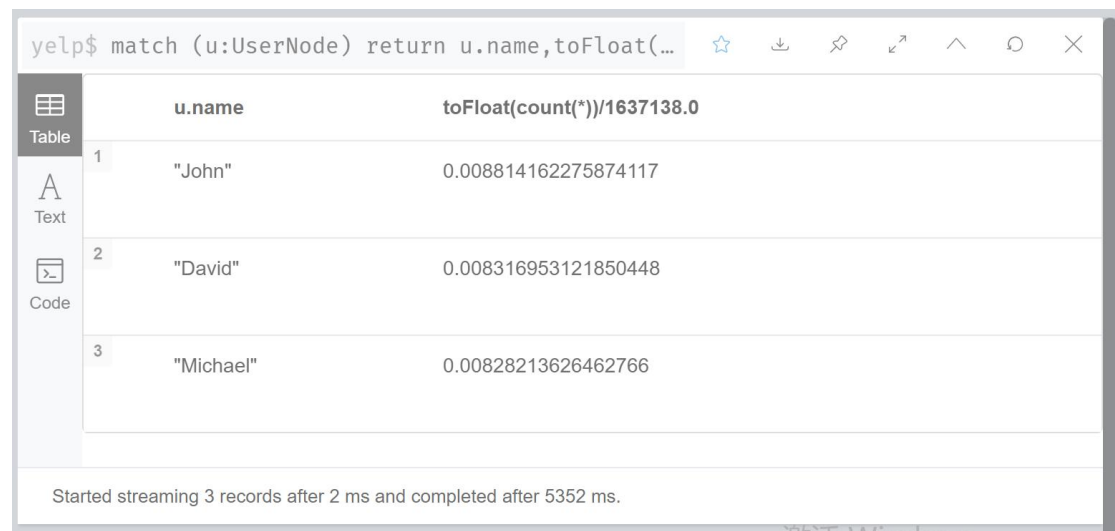
```
return u.name,toFloat(count(*)/1637138.0
```

```
order by toFloat(count(*)/1637138.0 desc
```

```
limit 3
```

首先计算出总用户名字，然后 count 转换成 float 因为 count 是整数无法除，最后返回用户名和结果即可。

总查询时间：10345ms



The screenshot shows a web interface with a query editor at the top containing the Cypher query: `yelp$ match (u:UserNode) return u.name,toFloat(count(*)/1637138.0`. Below the editor is a table view showing the results of the query. The table has two columns: `u.name` and `toFloat(count(*)/1637138.0`. There are three rows of data, numbered 1, 2, and 3. At the bottom of the interface, a status message reads: "Started streaming 3 records after 2 ms and completed after 5352 ms."

	u.name	toFloat(count(*)/1637138.0
1	"John"	0.008814162275874117
2	"David"	0.008316953121850448
3	"Michael"	0.00828213626462766

图 3.2

3.2.16 统计每个用户做出的评价的数量，按照用户评价的数量降序排列，返回用户名和评价的数量。

```
CREATE INDEX index_stars FOR (rn:ReviewNode) ON (rn.stars)
```

```
CREATE INDEX index_id FOR (u:UserNode)- ON (u.userid)
```

首先建立索引

```
MATCH(u:UserNode)-[:Review]->(rn:ReviewNode)
```

```
Return u.userid, count(rn.stars) order by count(rn.stars) desc
```

总查询时间：54916ms

yelp\$ MATCH(u:UserNode)-[:Review]→(rn:ReviewNode) Ret...

	u.userid	count(rn.stars)
1	"CxDOIDnH8gp9KXzpBHJYXw"	4129
2	"bLbSNkLggFnqwNNzzq-ljw"	2354
3	"PKEzKWv_FktMm2mGPjwd0Q"	1822
4	"ELcQDI69kb-ihJfxZyL0A"	1764
5	"DK57YibC5ShBmqQl97CKog"	1727

Started streaming 5 records after 2 ms and completed after 54916 ms.

图 3.3

3.2.17 实验建立索引对查询带来的提升，但会导致插入，删除等操作变慢（需要额外维护索引代价）。

以添加城市节点进行实验，

没有 index 插入删除：

Insert operation:

Create (n:Citynode {city: "undefined city"})

yelp\$ Create (n:Citynode {city: "undefined city"})

Added 1 label, created 1 node, set 1 property, completed after 10 ms.	

图 3.4

Deletion operation:

MATCH (n:Citynode {city: "undefined city"
})
DELETE n

yelp\$ MATCH (n:Citynode {city: "undefined city" }) D...

Deleted 1 node, completed after 13 ms.	

图 3.5

After create index:

Index creation process:

CREATE INDEX index_city FOR (n:Citynode) ON (n.city)

yelp\$ CREATE INDEX index_city FOR (n:Citynode) ON (n...

Added 1 index, completed after 45 ms.	

图 3.6

Insert operation:

Create (n:Citynode {city: "undefined city"})

yelp\$ Create (n:Citynode {city: "undefined city"})

Added 1 label, created 1 node, set 1 property, completed after 47 ms.	

图 3.7

Deletion operation:

```
MATCH (n:Citynode {city: "undefined city"
})
DELETE n
```

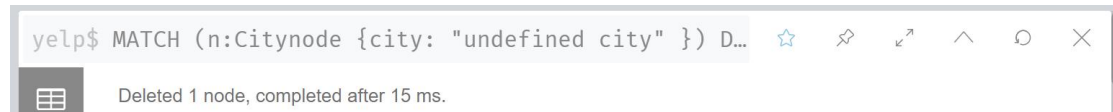


图 3.8

从上述实验可以看出，在创建了索引的情况下操作变慢，特别是 insert 操作。

3.2.18 查询与用户 user1 (userid: DXDeGUM_L7tt3u_5IkFRLw) 不是朋友关系的用户中和 user1 评价过相同的商家的用户，返回用户名，共同评价的商家的数量，按照评价数量降序排序[查看该查询计划，并尝试根据查询计划优化]。

```
CREATE INDEX index_businessid FOR (b:BusinessNode) ON (b.businessid)
CREATE INDEX index_userid FOR (u:UserNode) ON (u.userid)
```

同之前的实验，先建立索引

```
MATCH
(u:UserNode)-[r:Review]->(rn:ReviewNode)-[rd:Reviewed]->(b:BusinessNode)-[:Reviewed]-(:ReviewNode)-[:Review]-(:UserNode
{userid:"DXDeGUM_L7tt3u_5IkFRLw"})
Where not
(u:UserNode)-[:HasFriend]->(:UserNode{userid:"DXDeGUM_L7tt3u_5IkFRLw"})
and not
(:UserNode{userid:"DXDeGUM_L7tt3u_5IkFRLw"})-[:HasFriend]->(u:UserNode)
```

```
Return u.userid, count(b.businessid)
order by count(b.businessid) desc
```

为了使得代码看起来简单，两个 relationship 写在了一起，一条是从 UserNode 到 ReviewNode 再到 BusinessNode，另外一条是 user1 评价过的商家。Where 语句里面是除开朋友的关系，因为朋友是双向关系，有的数据可能没有全部导入到这个数据库所以双向关系都需要考虑。Return 的是用户 id 以及商家数量，最后用 order by 排倒序。

总查询时间：877ms

yelp\$ MATCH (u:UserNode)-[r:Review]-(rn:Review...

	u.userid	count(b.businessid)
1	"pJSnRKQU6CIKwKKaQAGd8w"	3
2	"XEh2VtVx5ZYATKEM4jZOXg"	3
3	"KavG2RBI77vRLqG1_6RBvQ"	3
4	"Z7gX8p-qvWQLsaEHwsqQCw"	3
5	"F_5_UNX-wrAFCXuAkBZRDw"	3

Started streaming 5 records after 7 ms and completed after 877 ms.

图 3.9

3.3 任务小结

写 Neo4j 代码时候不要用 MySQL 的思维写，chained with 语句会使得查询变很慢，代码也会变复杂，Neo4j 在查找 relationship 的情况下相对而言比较快。

画关系图可以很好地看出节点之间的关系从而写出代码。

4 验证 MySQL 在 InnoDB 存储引擎下 MVCC 多版本并发控制

4.1 任务要求

MySQL 体验 MySQL 在 InnoDB 存储引擎下的 MVCC 多版本并发控制，实现事务的 ACID 特性。

1. 创建一个数据库 test，在此数据库中创建一个表 student，属性只有一个 name 类型是 varchar，然后启动两个(win+r，再输入 cmd)窗口，登陆 MySQL。两个窗口可以都使用 root 账号登陆(mysql -uroot -pxxxx)xxx 是密码。

准备如下：

```
create database test; use test;
create table `student` (
    `name` varchar(64) not null
) engine=INNODB;
insert into `student` values("Alice");
```

4.2 完成过程

使用 show global variables 可以查看 InnoDB 隔离级别。（可重复读）

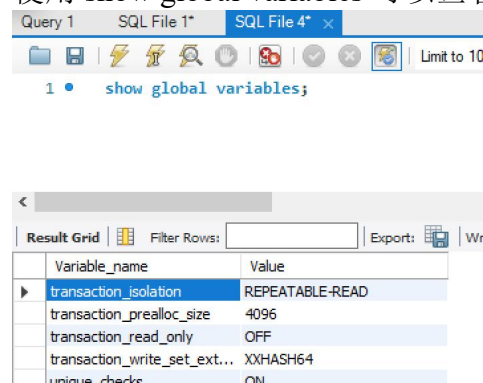


图 4.1

我们首先测试一下可重复的读的隔离级别下是否会出现不可以重复读的情况
为了实验的清晰，user1,user2 我都设置为自动提交等于零：

```
mysql> set autocommit = 0;
```

图 4.2

我们先进行 User1 查询：

```
mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from student;
+-----+
| name |
+-----+
| Alice |
+-----+
1 row in set (0.00 sec)
```


图 4.3

然后进行 User2 查询，User2 执行一个更新语句：

```
mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from student ;
+-----+
| name |
+-----+
| Alice |
+-----+
1 row in set (0.00 sec)

mysql> update student set name = "123";
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> commit;
Query OK, 0 rows affected (0.00 sec)
```

图 4.4

User2 commit a change to set the username to 123

User2 提交了事务。

```
mysql> select * from student; #repeatable
+-----+
| name |
+-----+
| Alice |
+-----+
1 row in set (0.00 sec)
```

图 4.5

我们返回 User1 的窗口进行查询，数据没有改动。（可重复读，MVCC）

The result returned by user 1 is corresponding to the result returned by user 1 at the beginning, so this means that the committed change user 2 made did not affect user 1.

然后 User1 也进行更改提交。

After committed user 1

```
mysql> commit;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from student;
+-----+
| name |
+-----+
| 123 |
+-----+
1 row in set (0.00 sec)
```

图 4.6

我们发现显示的是 User2 更改过后的数据。When the user 1 change has been committed, the name has been updated since user 2 has committed the change and user 1 also committed the change on their server.

4.3 任务小结

本实验验证 MYSQL 在 InnoDB 存储引擎下 MVCC 多版本并发控制，在同一个时间点不同的事务读取的数据版本是不同的。

5 程序设计

5.1 任务要求

尝试实现一个功能用来在 MySQL 和 neo4j 中添加一个新的用户，需要满足事务的一致性，保证在添加的用户在上述两个数据库中状态一致(同时添加成功，或者都不能添加，不能出现只在某一个数据库添加成功)

5.2 完成过程

首先建立好 python 环境，并且 *import* 需要的函数：*py2neo*，*mysql*，*datetime* 等，*install the mysql-connector-python package: pip install mysql-connector-python*

```
C:\Users\ctu28>pip install mysql-connector-python
Collecting mysql-connector-python
  Downloading mysql_connector_python-8.0.25-cp39-cp39-win_amd64.whl (794 kB)
    | 794 kB 819 kB/s
Collecting protobuf>=3.0.0
  Downloading protobuf-3.17.3-py2.py3-none-any.whl (173 kB)
    | 173 kB 437 kB/s
Collecting six>=1.9
  Downloading six-1.16.0-py2.py3-none-any.whl (11 kB)
Installing collected packages: six, protobuf, mysql-connector-python
Successfully installed mysql-connector-python-8.0.25 protobuf-3.17.3 six-1.16.0
```

图 5.1

尝试 insert 值到数据库确保能够成功工作

Trying to do the insertion operation insert user node to the mysql database

```
Help  python_connector.py

13  from datetime import datetime
14
15  #establishing the connection
16  conn = mysql.connector.connect(
17      user='root', password='5678857', host='127.0.0.1', database='yelp')
18
19  #Creating a cursor object using the cursor() method
20  cursor = conn.cursor()
21
22  #create date and time match timestamp/format
23  now = datetime.now()
24  formatted_date = now.strftime('%Y-%m-%d')
25  print(formatted_date)
26
27  # Preparing SQL query to INSERT a record into the database.
28  insert_stmt = ("INSERT INTO user"
29      "(userid, NAME, yelp_since, review_count,average_stars)"
30      "VALUES (%s, %s, %s, %s, %s)")
31
32  data = ('abcdefg', 'undefined', formatted_date, int(99), float(4.4))
33

Search  Stack Data  Debug I/O  Python Shell

Commands execute without debug. Use arrow keys for history.

Type "help", "copyright", "credits" or "license" for more information.
>>> [evaluate python_connector.py]
2021-06-19
Data inserted
>>>
```

图 5.2

Insertion succeed

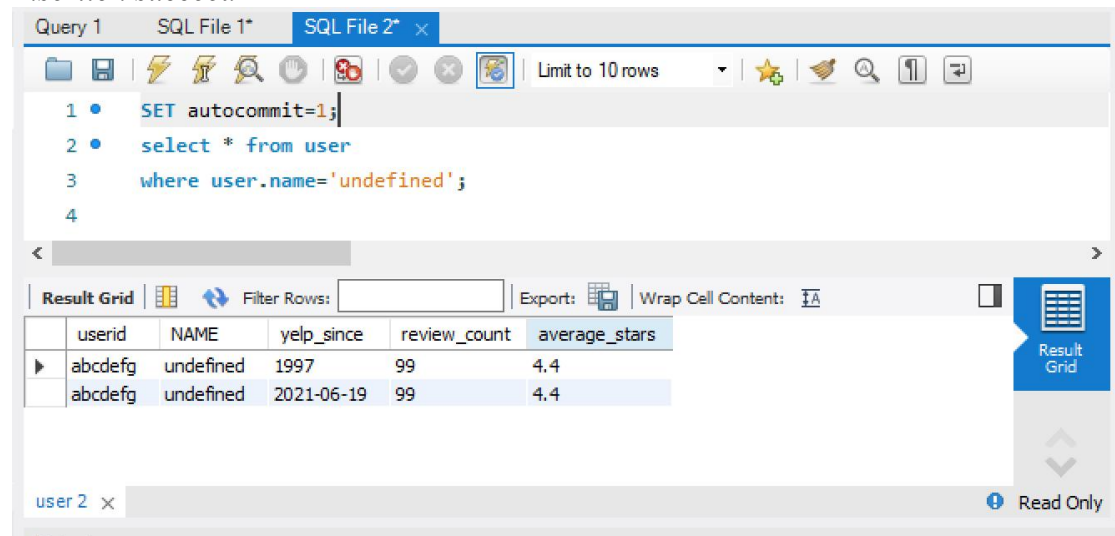


图 5.3

So we can find the node that we have inserted in the table.

Neo4j 部分：过程与上述相似，确保数据能够成功插入数据库。

Install neo4j tool

```
C:\Users\ctu28>pip install neo4j
Collecting neo4j
  Downloading neo4j-4.3.1.tar.gz (74 kB)
    | 74 kB 258 kB/s
Collecting pytz
  Downloading pytz-2021.1-py2.py3-none-any.whl (510 kB)
    | 510 kB 2.2 MB/s
Using legacy 'setup.py install' for neo4j, since package 'wheel' is not installed.
Installing collected packages: pytz, neo4j
  Running setup.py install for neo4j ... done
Successfully installed neo4j-4.3.1 pytz-2021.1
```

图 5.4

Install py2neo

```
C:\Users\ctu28>pip install --upgrade py2neo
Collecting py2neo
  Downloading py2neo-2021.1.5-py2.py3-none-any.whl (204 kB)
    | 204 kB 1.1 MB/s
  Running setup.py install for neotime ... done
Successfully installed certifi-2021.5.30 cffi-1.14.5 chardet-4.0.0 cryptography-3.4.7 docker-5.0.0 english-20
e-2.10 monotonic-1.6 neotime-1.7.4 packaging-20.9 pansy-2020.7.3 prompt-toolkit-3.0.19 py2neo-2021.1.5 pypars
ments-2.9.0 pyparsing-2.4.7 pywin32-227 requests-2.25.1 urllib3-1.26.5 wcwidth-0.2.5 websocket-client-1.1.0
```

图 5.5

程序思路：

创建两个 boolean 来表示两个数据库 insertion 的状态，同时设定为 False；执行 SQL 插入，如果成功，MySQL_data_inserted 状态变成 True，否则执行 rollback；如果 MySQL 插入成功，执行 Neo4j 插入：如果 Neo4j 成功插入，Neo4j_data_inserted 状态变成 True，否则 rollback 之前的 MySQL 数据；最后如果两者都成功了，commit the change，关闭程序。

程序测试:

①两者都成功 insert (在 MySQL 和 Neo4j 都成功 insert 的情况下)

```
>> [evaluate python_connector.py]
data has been inserted into MySQL
data has been inserted into neo4j
database commitment succeed!
```

图 5.6

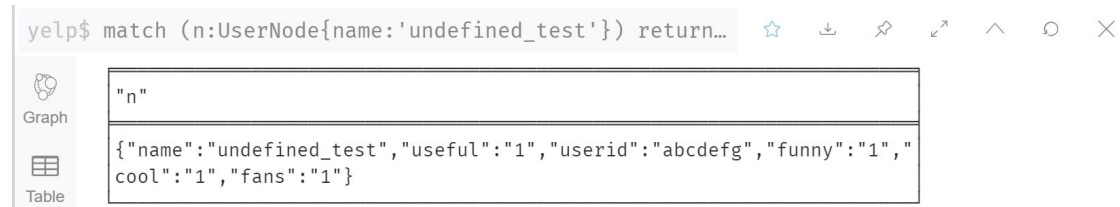


图 5.7

在 MySQL 数据库中我们查询到插入的数据

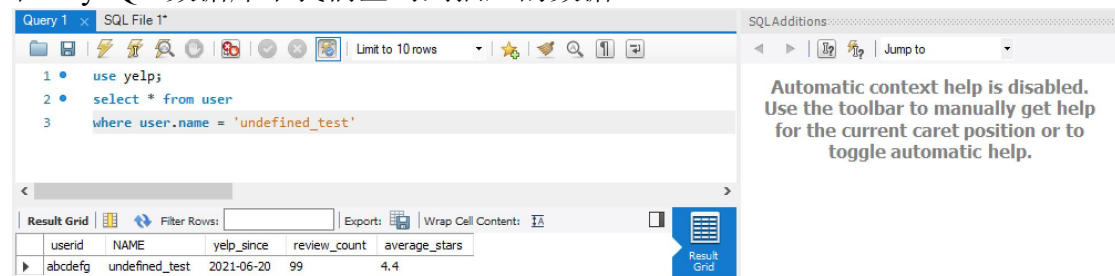


图 5.8

②Neo4j 没有成功情况下, 两者都查询不到数据

```
>> [evaluate python_connector.py]
data has been inserted into MySQL
neo4j insertion failed
```

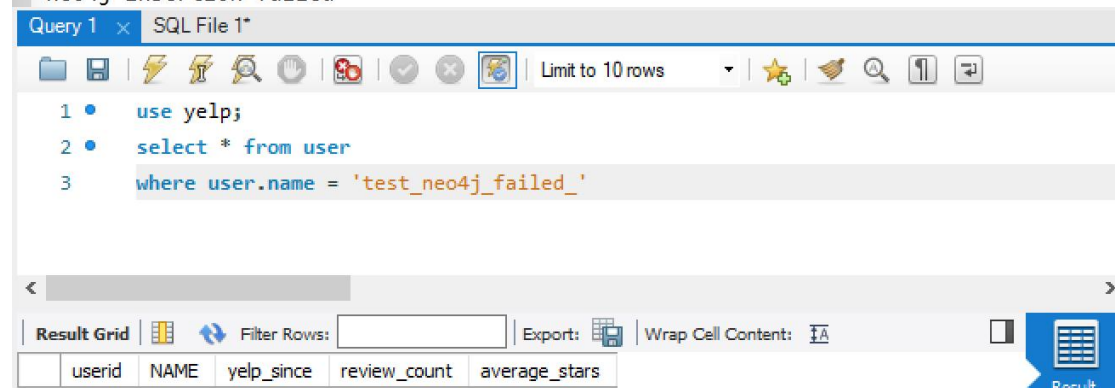


图 5.9

③MySQL 没有成功的情况下, 不会进行 Neo4j 的数据插入, 直接 rollback

```
Python 3.9.5 (tags/v3.9.5:0a7dcdb, May 3 2021, 17:27:52) [MSC v.1928 64 bit (AMD64)]
Type "help", "copyright", "credits" or "license" for more information.
>>> [evaluate python_connector.py]
MySQL data insertion failed
```

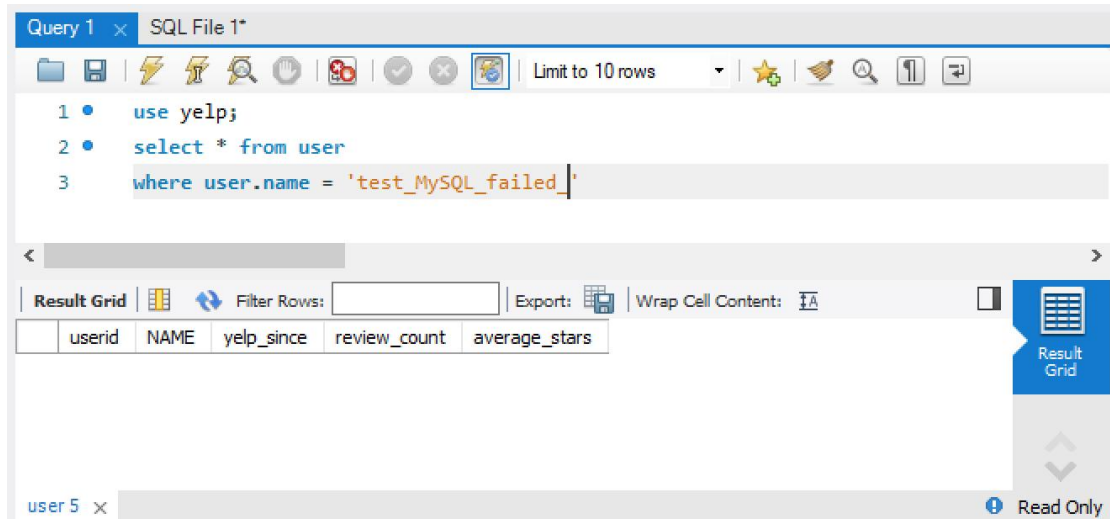


图 5.10

④如果两者都数据插入失败了，情况和 MySQL 失败类似，不会进行后续操作。

代码详见本报告附录或者 Github:

https://github.com/changtu28/Big-data-course-HUST-/blob/main/python_connector.py

5.3 任务小结

本实验的程序实现了在 python 里面对 MySQL 和 neo4j 中同时添加新用户的操作，也满足了事务的一致性，保证节点同时添加成功。

6 课程总结

本实验介绍了在 MongoDB 和 Neo4j 里面的基本操作，比如如何导入数据，基本的增删改，语句查询。在数据量大的时候如何通过增加索引或者改变语句优化搜索，任务三验证了 MySQL 在 InnoDB 存储引擎下的 MVCC 多版本并发控制，再次强调了 ACID 特性的重要性，系统完整的重要性。最后用程序实际操作数据库，同时在 MySQL 和 Neo4j 里面插入数据，在理论上进行了实操实验。

在未来的实验中或许还可以考虑涉及到三种并发问题——脏读，不可重复读，幻读；日志的重做和回滚，checkpoint 的好处，锁的使用方法等等。

附录

```
#-----
#   Program that insert node simultaneously into the MySQL DB and Neo4j DB
#   Author: Chang Tu
#-----

from py2neo import Graph, Node, Relationship
import mysql.connector
from datetime import datetime

#-----MySQL part-----
# set the status of the data to be inserted into the database
MySQL_data_inserted = False; Neo4j_data_inserted = False

#establishing the connection to MySQL server
conn = mysql.connector.connect(
    user='root', password='5678857', host='127.0.0.1', database='yelp')

#Creating a cursor object using the cursor() method(MySQL)
cursor = conn.cursor()

#create date and time match timestamp/format
now = datetime.now()
formatted_date = now.strftime('%Y-%m-%d')

# Preparing SQL query to INSERT a record into the database.
insert_stmt = ("INSERT INTO user"
               "(userid, NAME, yelp_since, review_count,average_stars)"
               "VALUES (%s, %s, %s, %s, %s)")

try:
    SQLdata = ('abcdefg', 'test_MySQL_failed_', formatted_date, int(99), float(4.4))
    # Executing the SQL command
    cursor.execute(insert_stmt, SQLdata)

    # change the status of MySQL_data_inserted
    MySQL_data_inserted = True

    # Commit changes in the SQL database
    print("data has been inserted into MySQL")

except:
    # Rolling back in case of error
    conn.rollback()
    print("MySQL data insertion failed")

#-----Neo4j part-----
# if the MySQL data has been successfully inserted into the database
if MySQL_data_inserted:

    # Insert new nodes in neo4j
    try:
        graph = Graph("bolt://localhost:7687", auth=("neo4j", "Tc5678857"))
        tx=graph.begin()
        neo4j_data = Node('UserNode', name=
'test_MySQL_failed_',userid="abcdefg",useful="1",funny="1",cool="1",fans="1")
        tx.create(neo4j_data)
        Neo4j_data_inserted = True
        print("data has been inserted into neo4j")
    except:
        conn.rollback()
        print("neo4j insertion failed and rollback for MySQL")

# committed changes in two databases
if MySQL_data_inserted and Neo4j_data_inserted:
    try:
        conn.commit()
```



```
graph.commit(tx)
print("database commitment succeed!")
except:
    print("database commitment failed!")

## Closing the connection for MySQL
conn.close()
```