

**大数据管理理论综合报告**

|  |  |
| --- | --- |
|  |  |
| 姓 名： | 董玲晶 |
| 学 院： | 计算机科学与技术学院 |
| 专 业： | 计算机科学与技术 |
| 班 级： | CS2005 |
| 学 号： | U202090063 |

|  |  |
| --- | --- |
| 分数 |  |
| 教师签名 |  |

2023 年 04月29日

**目录**

[任务一：MongoDB实验 4](#_Toc133713218)

[启动 4](#_Toc133713219)

[1-1 4](#_Toc133713220)

[1-2 4](#_Toc133713221)

[1-3 5](#_Toc133713222)

[1-4 5](#_Toc133713223)

[1-5 5](#_Toc133713224)

[1-6 6](#_Toc133713225)

[1-7 7](#_Toc133713226)

[1-8 7](#_Toc133713227)

[1-9 8](#_Toc133713228)

[1-10 10](#_Toc133713229)

[1-11 10](#_Toc133713230)

[1-12 12](#_Toc133713231)

[1-13 13](#_Toc133713232)

[1-14 15](#_Toc133713233)

[1-15 16](#_Toc133713234)

[任务二：Neo4j实验 18](#_Toc133713235)

[启动 18](#_Toc133713236)

[2-1 18](#_Toc133713237)

[2-2 18](#_Toc133713238)

[2-3 19](#_Toc133713239)

[2-4 19](#_Toc133713240)

[2-5 19](#_Toc133713241)

[2-6 20](#_Toc133713242)

[2-7 20](#_Toc133713243)

[2-8 20](#_Toc133713244)

[2-9 22](#_Toc133713245)

[2-10 22](#_Toc133713246)

[2-11 23](#_Toc133713247)

[2-12 24](#_Toc133713248)

[2-13 25](#_Toc133713249)

[2-14 27](#_Toc133713250)

[2-15 28](#_Toc133713251)

[2-16 29](#_Toc133713252)

[2-17 30](#_Toc133713253)

[2-18 34](#_Toc133713254)

[任务三：多数据库交互应用实验 35](#_Toc133713255)

[3-1 35](#_Toc133713256)

[3-2 35](#_Toc133713257)

[3-3 39](#_Toc133713258)

[任务四：不同类型数据库MVCC多版本并发控制对比实验 41](#_Toc133713259)

[MySQL 41](#_Toc133713260)

[MongoDB 44](#_Toc133713261)

[不同之处 46](#_Toc133713262)

## 任务一：MongoDB实验

### 启动

代码1.1：启动mongodb命令

|  |
| --- |
| // 命令行启动mongod服务 mongod --dbpath /var/lib/mongodb/ --logpath /var/log/mongodb/mongodb.log --logappend & mongo use yelp |

### 1-1

#### **题目**

查询review集合的2条数据，跳过第1条和第2条

#### **解析**

使用了skip()方法跳过前两条数据，然后使用limit()方法来限制结果集的大小为2。如果review集合中有少于4条数据，则返回所有匹配的数据。

代码1.2

|  |
| --- |
| db.review.find().skip(2).limit(2) |

### 1-2

#### **题目**

查询business集合中city是Goodyear的5条数据。

#### **解析**

db.business表示选择business集合；{ city: "Goodyear" }是查询条件，表示选择city字段等于Goodyear的文档；.limit(5)表示限制返回的文档数量为5条。

代码1.3

|  |
| --- |
| db.business.find({ city: "Goodyear" }).limit(5) |

### 1-3

#### **题目**

查询user集合中name是Tanya的user，只需要返回useful和cool,限制10条数据。

#### **解析**

db.user表示选择user集合；{ name: "Tanya" }是查询条件，表示选择name字段等于Tanya的文档；{ \_id: 0, useful: 1, cool: 1 }表示仅返回useful和cool字段，同时将\_id字段排除在外；.limit(10)表示限制返回的文档数量为10条。

代码1.4

|  |
| --- |
| db.user.find({ name: "Tanya" }, { \_id: 0, useful: 1, cool: 1 }).limit(10) |

### 1-4

#### **题目**

查询user集合中funny位于[82，83，84]的user，只需返回name和funny，限制20条数据。

#### **解析**

db.user表示选择user集合；{ funny: { $in: [82, 83, 84] } }是查询条件，表示选择funny字段值在数组[82, 83, 84]中的文档；{ \_id: 0, name: 1,funny: 1 }表示仅返回name和funny字段，同时将\_id字段排除在外；.limit(20)表示限制返回的文档数量为20条。

代码1.5

|  |
| --- |
| db.user.find({ funny: { $in: [82, 83, 84] } }, { \_id: 0, name: 1, funny: 1 }).limit(20) |

### 1-5

#### **题目**

查询user集合中5≤cool<10且useful≥20的user，限制10条。

#### **解析**

db.user表示选择user集合；{ cool: { $gte: 5, $lt: 10 }, useful: { $gte: 20 } }是查询条件，表示选择cool字段值大于等于5且小于10，且useful字段值大于等于20的文档；.limit(10)表示限制返回的文档数量为10条。

代码1.6

|  |
| --- |
| db.user.find({ cool: { $gte: 5, $lt: 10 }, useful: { $gte: 20 } }).limit(10) |

### 1-6

#### **题目**

统计business一共有多少条数据，并使用explain查询执行计划，了解MongoDB对集函数的执行方式。

#### **解析**

先统计，再用.explain("executionStats")查询MongoDB对count()命令的执行计划

代码1.7

|  |
| --- |
| db.business.count() > 192609 db.business.explain("executionStats").count() |



图1.1：explain()查询执行计划图

包含了优化选择的查询计划和查询的执行统计集合

* "winningPlan": {"stage": "COUNT"} 表示优化器选择了一个简单的计数操作作为最优查询计划，即对集合进行 COUNT 操作来统计文档数。
* "rejectedPlans": [] 表示优化器没有拒绝任何其他查询计划。
* "nReturned": 0 表示返回的文档数为 0。
* "executionTimeMillis": 0 表示执行该查询所用的时间为 0 毫秒。
* "totalKeysExamined": 0 表示索引键被查询的次数为 0。
* "totalDocsExamined": 0 表示被查询的文档数为 0。
* "executionStages": {"stage": "COUNT", ...} 描述了查询的执行阶段，包括：COUNT 阶段、估计执行时间、文档计数、跳过的文档数等。

### 1-7

#### **题目**

查询business集合city为Phoenix或者Charlotte的数据。

#### **解析**

使用*$or*操作符

代码1.8

|  |
| --- |
| db.business.find({$or: [{city: "Phoenix"}, {city: "Charlotte"}]}).count() > 28275 db.business.find({$or: [{city: "Phoenix"}, {city: "Charlotte"}]}) |

### 1-8

#### **题目**

查询business集合中，categories为7种的商户信息，显示这7种类别，限制10条

#### **解析**

$size表示匹配数组大小的条件操作符，7表示指定的数组大小。{\_id: 0, name: 1, categories: 1}表示只返回文档中的name和categories字段，不返回\_id字段。limit(10)表示只返回10条匹配的结果。

代码1.9

|  |
| --- |
| db.business.find({categories: {$size: 7}}, {\_id: 0, name: 1, categories: 1}).limit(10) |

### 1-9

#### **题目**

使用explain看db.business.find({business\_id: "5JucpCfHZltJh5r1JabjDg"})的执行计划，了解该查询的执行计划及查询执行时间，并给出物理优化手段，以提高查询性能，通过优化前后的性能对比展现优化程度。

#### **解析**

先用explain()得到查询执行计划

代码1.10

|  |
| --- |
| // 原始查询执行计划 db.business.find({business\_id: "5JucpCfHZltJh5r1JabjDg"}).explain("executionStats") |

MongoDB优化器发现没有*business\_id*的索引可以使用，因此选择了*COLLSCAN*，即遍历整个集合。在执行阶段，我们可以看到查询语句扫描了192609个文档。查询返回了1个文档，执行时间为51ms。

* "nReturned": 1 表示返回的文档数为 1。
* "executionTimeMillis": 51 表示执行该查询所用的时间为 51 毫秒。
* "totalKeysExamined": 0 表示索引键被查询的次数为 0。
* "totalDocsExamined": 192609 表示被查询的文档数为 192609。
* "executionStages": {"stage": "COLLSCAN", ...} 描述了查询的执行阶段，包括：全集合扫描阶段、过滤条件、文档计数、跳过的文档数等。

代码1.11

|  |
| --- |
| // 没有建立索引时的explain结果 > "stage": "COLLSCAN" > "queryPlanner": {  "indexFilterSet": false,  "parsedQuery": {  "business\_id": {  "$eq": "5JucpCfHZltJh5r1JabjDg"  }  },  "winningPlan": {  "stage": "COLLSCAN",  "filter": {  "business\_id": {  "$eq": "5JucpCfHZltJh5r1JabjDg"  }  },  },  "rejectedPlans": []  } > "executionStats": {  "executionSuccess": true,  "executionTimeMillis": 51,  "totalDocsExamined": 192609,  "executionStages": {  "stage": "COLLSCAN",  "filter": {  "business\_id": {  "$eq": "5JucpCfHZltJh5r1JabjDg"  }  },  }  } |

针对这个查询，可以尝试如下的物理优化手段：  
创建索引：对于经常查询的字段，可以创建索引以加快查询速度。在本例中，*business\_id*字段是查询条件，可以考虑为该字段创建一个索引。

代码1.12

|  |
| --- |
| // 创建business\_id字段的索引，注：原来的一条索引是id默认索引 db.business.createIndex({business\_id: 1}) > {  "createdCollectionAutomatically" : false,  "numIndexesBefore" : 1,  "numIndexesAfter" : 2,  "ok" : 1  } |

通过索引*business\_id\_1*进行扫描并返回结果，因为该索引是唯一的，并且查询只需要扫描一次。在执行过程中，总共检查了一个键和一个文档，并且执行时间为 2 ms，比原来的51ms快很多。

代码1.13

|  |
| --- |
| // 建立索引后再次explain后的结果 > "executionStats": {  "executionSuccess": true,  "nReturned": 1,  "executionTimeMillis": 2,  "totalKeysExamined": 1,  "totalDocsExamined": 1,  "executionStages": {  "stage": "FETCH",  "inputStage": {  "stage": "IXSCAN",  "keyPattern": {  "business\_id": 1  },  "indexName": "business\_id\_1",  }  }  } |

### 1-10

#### **题目**

统计各个星级的商店的个数，返回星级数和商家总数，按照星级降序排列。

#### **解析**

使用MongoDB的聚合管道来实现这个需求。首先，需要用*$group*操作符按照*stars*字段进行分组，并使用*$sum*操作符统计每个星级的商店个数。然后，再用*$sort*操作符按照星级数降序排列。

代码1.14

|  |
| --- |
| db.business.aggregate([{$group:{\_id:"$stars", count:{$sum:1}}}, {$sort:{\_id:-1}}]) |

### 1-11

#### **题目**

创建一个review的子集合Subreview(取review的前五十万条数据)，分别对评论的内容建立全文索引，对useful建立升序索引，然后查询评价的内容中包含关键词delicious且useful大于8的评价。插入数据过程耗时约150s，建索引耗时约60s。

#### **解析**

首先使用 *$limit* 管道操作符将 review 集合中的文档限制为前 500,000 条，然后使用 *$out* 管道操作符将结果输出到一个名为Subreview的新集合中，这样就创建一个新的子集合 Subreview。

代码1.15

|  |
| --- |
| db.review.aggregate([{$limit: 500000}, {$out: "Subreview"}]) |

创建前后都可以使用*show collections*来查看目前数据库中的集合，可以看到执行以上语句后数据库中多了一条叫做Subreview的集合，且我们查询其文档数目，为50w。

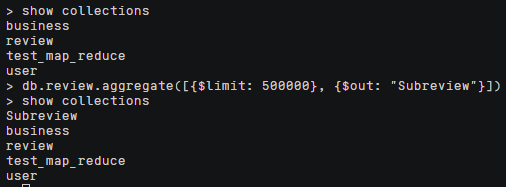


图1.2：创建新集合的结果图

fig:

图1.3：新集合文档条数查询

我们先查看一下Subreview中的数据，评论字段叫做text

代码1.16

|  |
| --- |
| db.Subreview.find().limit(1) > {  "\_id": ObjectId("600d7ea4f5e9bd91d7c30018"),  "review\_id": "Q1sbwvVQXV2734tPgoKj4Q",  "user\_id": "hG7b0MtEbXx5QzbzE6C\_VA",  "business\_id": "ujmEBvifdJM6h6RLv4wQIg",  "stars": 1,  "useful": 6,  "funny": 1,  "cool": 0,  "text": "Total bill for this horrible service? Over $8Gs. These crooks actually had the nerve to charge us $69 for 3 pills. I checked online the pills can be had for 19 cents EACH! Avoid Hospital ERs at all costs.",  "date": "2013-05-07 04:34:36"  } |

对评论的内容建立全文索引，对useful建立升序索引

代码1.17

|  |
| --- |
| // 对评论的内容建立全文索引 db.Subreview.createIndex({text: "text"}) > {  "createdCollectionAutomatically" : false,  "numIndexesBefore" : 1,  "numIndexesAfter" : 2,  "ok" : 1  }  // MongoDB默认为字段"useful"创建了升序索引,降序索引通过指定-1来实现。 db.Subreview.createIndex({useful: 1}) > {  "createdCollectionAutomatically" : false,  "numIndexesBefore" : 2,  "numIndexesAfter" : 3,  "ok" : 1  } |

查询

代码1.18

|  |
| --- |
| db.Subreview.find({$text: {$search: "delicious"}, useful: {$gt: 8}}) db.Subreview.find({$text: {$search: "delicious"}, useful: {$gt: 8}}).count() |

fig:

图1.4：查询结果条数

### 1-12

#### **题目**

在Subreview集合中统计评价中useful、funny和cool都大于5的商家，返回商家id及平均打星，并按商家id降序排列

#### **解析**

$match 阶段：筛选出 useful、funny 和 cool 都大于 5 的评价。

$group 阶段：按照 business\_id 字段分组，计算每个商家的平均打星数。

$sort 阶段：按照商家 id 降序排列。

代码1.19

|  |
| --- |
| // 题目要求 db.Subreview.aggregate([  {$match: {useful: {$gt: 5}, funny: {$gt: 5}, cool: {$gt: 5}}},  {$group: {\_id: "$business\_id", average\_stars: {$avg: "$stars"}}},  {$sort: {\_id: -1}} ]) |

为了得到结果的数量，可以添加一个额外的$group聚合管道阶段，来计算匹配的文档总数。它将所有文档分组为一个组，\_id: null意味着将所有文档放在同一个分组中。然后使用$sum操作符来计算文档数量，并将其存储在新的count字段中。可以看到结果为1894

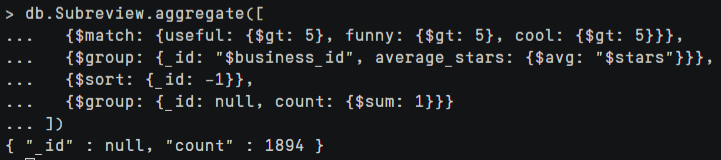


图1.5：结果条数图

### 1-13

#### **题目**

查询距离商家 *xvX2CttrVhyG2z1dFg\_0xw(business\_ id)* 120米以内的商家，只需要返回商家名字，地址和星级。提示：使用2dsphere建立索引、获取商家地理坐标、使用坐标进行查询

#### **解析**

2dsphere索引是MongoDB支持的一种地理位置索引，用于存储和查询包含地理位置信息的数据。

首先还是先看一下business\_id为*xvX2CttrVhyG2z1dFg\_0xw*的商家信息，并且可以得到地理位置信息在loc字段。

代码1.20

|  |
| --- |
| db.business.find({business\_id: "xvX2CttrVhyG2z1dFg\_0xw"}) > {  "\_id": ObjectId("6016c6b4af81085b0f2183c4"),  "business\_id": "xvX2CttrVhyG2z1dFg\_0xw",  "postal\_code": "85338",  "latitude": 33.4556129678,  "longitude": -112.3955963552,  "loc": {  "type": "Point",  "coordinates": [  -112.3955963552,  33.4556129678  ]  }  } |

所以我们在loc字段上建立一个2dsphere索引，原来的两条一条是id（默认的），一条是business\_id（前面建立的）

代码1.21

|  |
| --- |
| db.business.createIndex({ loc: "2dsphere" }) > {  "createdCollectionAutomatically" : false,  "numIndexesBefore" : 2,  "numIndexesAfter" : 3,  "ok" : 1  } |

最后，使用*$near*操作符来查找附近的商家。该操作符接受一个*$geometry*参数来指定查询的中心点（即商家 xvX2CttrVhyG2z1dFg\_0xw 的坐标），并接受一个*$maxDistance*参数来指定查询的最大距离

代码1.22

|  |
| --- |
| db.business.find({  loc: {  $near: {  $geometry: {  type: "Point",  coordinates: [-112.3955963552, 33.4556129678]  },  $maxDistance: 120  }  } }, {  \_id: 0,  name: 1,  address: 1,  stars: 1 }) |

结果有三条

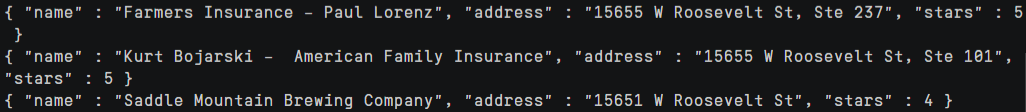


图1.6：查询结果展示

### 1-14

#### **题目**

在集合Subreview上建立索引，统计出用户从2016年开始发出的评价有多少，按照评价次数降序排序，需要返回用户id和评价总次数，只显示前20条结果。

#### **解析**

由于要查询从2016年开始的评价，前面我们打印过Subview的数据条目的结构，包含有一个date字段，因此可以考虑在data上建立一个索引。同时由于要统计每个用户的评价次数，可以考虑再增加一个对于用户id的索引，因此我们要做的是建立一个用户id和评价时间的复合索引。

代码1.23

|  |
| --- |
| db.Subreview.createIndex({user\_id: 1, date: 1}) > {  "createdCollectionAutomatically" : false,  "numIndexesBefore" : 3,  "numIndexesAfter" : 4,  "ok" : 1  } |

先用*$substr*提取出年份，再用*$toInt*转成int类型，命名为year字段；然后使用*$match*以及*$gte*匹配所有年份大于2016年的数据。然后使用*$group*统计每个用户的评价次数，其中 \_id 字段表示按照用户id进行分组，count字段使用*$sum*操作符计算评价次数的总和。接着使用*$sort*操作符对评价次数降序排序，最后使用*$limit*限制结果集大小为前20条。

代码1.24

|  |
| --- |
| db.Subreview.aggregate([  { $project: { year: { $toInt: { $substr: ["$date", 0, 4] } } } },  { $match: { year: {$gte: 2016} } },  { $group: { \_id: "$user\_id", count: {$sum: 1} } },  { $sort: { count: -1 } },  { $limit: 20 },  { $project: { \_id: 1, count: 1 } } ]) |

### 1-15

#### **题目**

使用mapreduce计算每个商家的评价的平均分（建议在Subreview集合上做，review过于大）,不要直接使用聚合函数。

#### **解析**

通过map函数将每条评价按照商家id进行分组，输出商家id和该评价的星级以及计数器初始值为1。然后通过reduce函数将同一商家的评价星级累加，并将计数器进行累加，返回商家id和星级总和以及评价数总和。最后通过finalize函数计算出每个商家的平均评分，并输出到一个叫做Average\_Stars的新集合中。

代码1.25

|  |
| --- |
| db.Subreview.mapReduce(  function() {  emit(this.business\_id, { stars: this.stars, count: 1 });  },   function(key, values) {  var totalStars = 0;  var totalCount = 0;  values.forEach(function(value) {  totalStars += value.stars;  totalCount += value.count;  });  return { stars: totalStars, count: totalCount };  },   {  out: "Average\_Stars",  finalize: function(key, reducedValue) {  return { avgStars: reducedValue.stars / reducedValue.count };   // 如果按输出参考文档里的格式：  var avgStars = reducedValue.stars / reducedValue.count;  return { stars: reducedValue.stars, count: reducedValue.count, avg: avgStars };  }  } ) |

执行结果如下

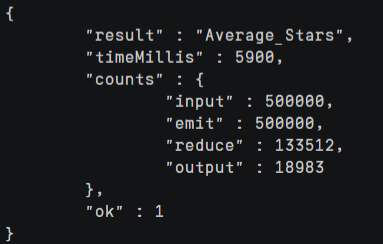


图1.7：查询结果展示

查看一下Average\_Stars集合中的数据

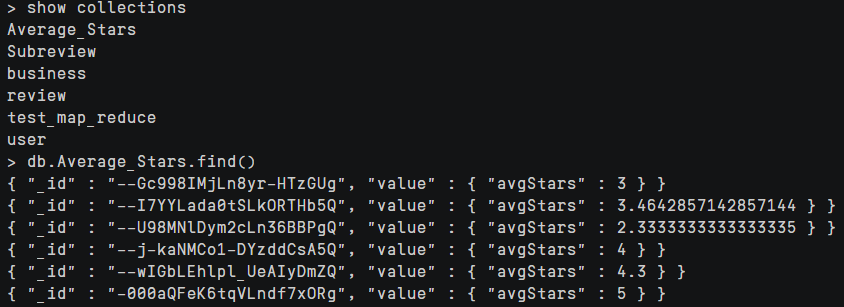


图1.7：结果展示

## 任务二：Neo4j实验

### 启动

进入的默认目录在~

代码2.1

|  |
| --- |
| cd ./neo4j-community-4.0.9/bin ./neo4j console |

打开浏览器，输入http://124.71.146.178:7474，服务器IP地址+7474端口号，登录账号为neo4j。

### 2-1

#### **题目**

查询标签是UserNode的节点，限制10个

#### **解析**

代码2.2

|  |
| --- |
| MATCH (n:UserNode) RETURN n LIMIT 10 |

### 2-2

#### **题目**

查询城市是Chandler的商家节点。

#### **解析**

代码2.3

|  |
| --- |
| MATCH (business:BusinessNode {city: 'Chandler'}) RETURN business |

### 2-3

#### **题目**

查询reviewid是xkVveYJIL1Eiwl46cP\_VBg对应的bussiness信息。

#### **解析**

代码2.4

|  |
| --- |
| MATCH(:ReviewNode {reviewid:'xkVveYJIL1Eiwl46cP VBg'})-[:Reviewed]->(business:BusinessNode) RETURN business |

### 2-4

#### **题目**

查询评价过businessid是5ykOWYZ44sUvu9qxD8rPeg商家的用户的名字和粉丝数。

#### **解析**

代码2.5

|  |
| --- |
| MATCH (user:UserNode)-[:Review]->(:ReviewNode)  -[:Reviewed]->(:BusinessNode {businessid: '5ykOWYZ44sUvu9qxD8rPeg'}) RETURN user.name, user.fans |

### 2-5

#### **题目**

查询被userid为0kSXMbNFo7mdwTPj4iQv9A的用户评论为5星的商家名称和地址。

#### **解析**

代码2.6

|  |
| --- |
| MATCH (:UserNode {userid: '0kSXMbNFo7mdwTPj4iQv9A'})  -[:Review]->(:ReviewNode {stars: '5.0'})  -[:Reviewed]->(business:BusinessNode) RETURN business.name, business.address |

### 2-6

#### **题目**

查询商家名及对应的星级和地址，按照星级降序排序（限制15条）。

#### **解析**

代码2.7

|  |
| --- |
| MATCH (business:BusinessNode) RETURN business.name, business.stars, business.address ORDER BY business.stars DESC LIMIT 15 |

### 2-7

#### **题目**

使用where查询粉丝数大于100的用户的名字和粉丝数（限制10条）

#### **解析**

代码2.8

|  |
| --- |
| MATCH (user:UserNode) WHERE toInteger(user.fans) > 100 RETURN user.name, user.fans LIMIT 10 |

### 2-8

#### **题目**

查询businessid是5ykOWYZ44sUvu9qxD8rPeg商家包含的种类数,并使用PROFILE查看执行计划，进行说明

#### **解析**

代码2.9

|  |
| --- |
| MATCH (:BusinessNode {businessid: '5ykOWYZ44sUvu9qxD8rPeg'})  -[:IN\_CATEGORY]->(c:CategoryNode) RETURN COUNT(DISTINCT c)  PROFILE MATCH (:BusinessNode {businessid: '5ykOWYZ44sUvu9qxD8rPeg'})  -[:IN\_CATEGORY]->(c:CategoryNode) RETURN COUNT(DISTINCT c) |

查询计划图

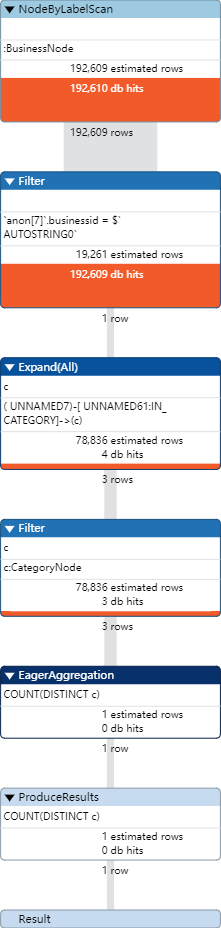


图2.1：查询计划图

### 2-9

#### **题目**

查询businessid是5ykOWYZ44sUvu9qxD8rPeg商家包含的种类,以list的形式返回。

#### **解析**

代码2.10

|  |
| --- |
| MATCH (:BusinessNode {businessid: '5ykOWYZ44sUvu9qxD8rPeg'})  -[:IN\_CATEGORY]->(c:CategoryNode) RETURN COLLECT(c.category) |

是以list的形式返回

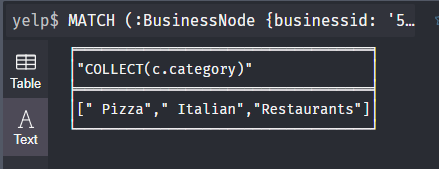


图2.2：查询结果图

### 2-10

#### **题目**

查询Victoria的朋友（直接相邻）分别有多少位朋友。(考察：使用with传递查询结果到后续的处理)

#### **解析**

先找到名为"Victoria"的用户节点，从该节点出发，通过关系类型"HasFriend"找到其所有朋友节点，对于每一个朋友节点，计算其名称和其朋友的数量

代码2.11

|  |
| --- |
| MATCH(:UserNode{name:'Victoria'})-[:HasFriend]->(friend) WITH friend.name as friendsList, size((friend)-[:HasFriend]-()) as number0fFoFs RETURN friendsList,number0fFoFs |

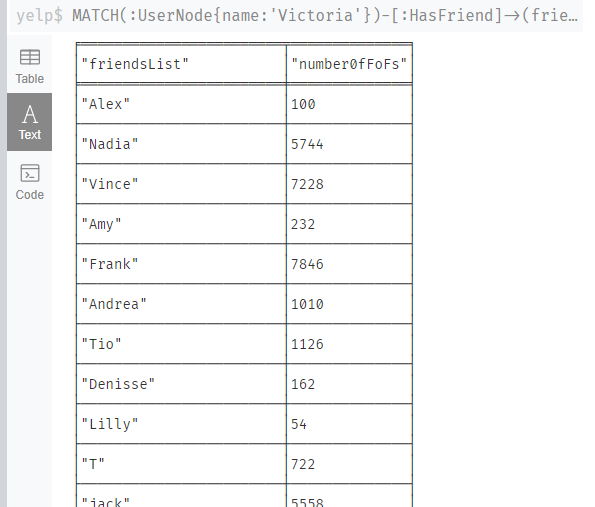


图2.3：查询结果图

### 2-11

#### **题目**

查询城市是Chandler的商家节点。

#### **解析**

* 使用MATCH子句匹配类别为Hot Pot的CategoryNode节点和它们所对应的 BusinessNode节点，并将匹配到的BusinessNode节点的city属性作为变量cityName传递到下一个子句。
* 使用WITH子句对每个城市进行分组，统计该城市中类别为Hot Pot的商家数量，并将城市名称和商家数量传递到下一个子句。
* 使用ORDER BY子句将商家数量按照降序排序，以便选择前5个数量最多的城市。
* 使用RETURN子句输出城市名称和对应的商家数量，并限制结果数量为前5条。

代码2.12

|  |
| --- |
| MATCH (b:BusinessNode)-[:IN\_CATEGORY]->(:CategoryNode {category: 'Hot Pot'}) WITH b.city AS cityName, COUNT(\*) AS numberOfBusiness RETURN cityName, numberOfBusiness ORDER BY numberOfBusiness DESC |

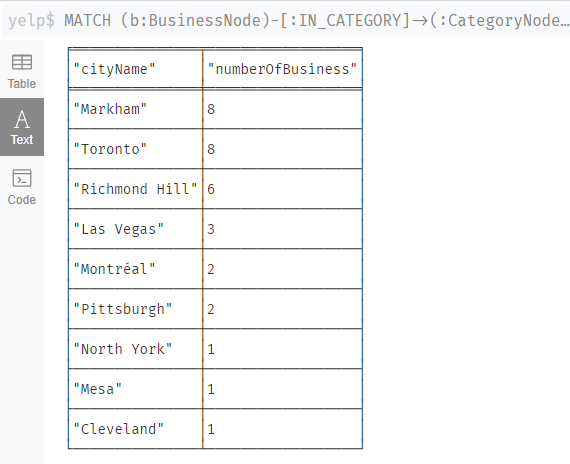


图2.4：查询结果图

### 2-12

#### **题目**

查询商家名重复次数前10的商家名及其次数。

#### **解析**

首先匹配所有的 BusinessNode 节点，并将每个节点的 name 属性作为变量 name 传递到下一个子句。然后使用 WITH 子句对每个商家名称进行分组，统计重复出现的次数，并将商家名称和出现次数传递到下一个子句。接着使用 WHERE 子句筛选出出现次数大于 1 的商家名称，以便选择重复出现的商家名称。最后使用输出商家名称和对应的出现次数，并按照出现次数降序排序，以便选择前 10 个重复出现次数最多的商家名称。

代码2.13

|  |
| --- |
| MATCH (b:BusinessNode) WITH b.name AS name, COUNT(\*) AS count WHERE count > 1 RETURN name, count ORDER BY count DESC LIMIT 10 |

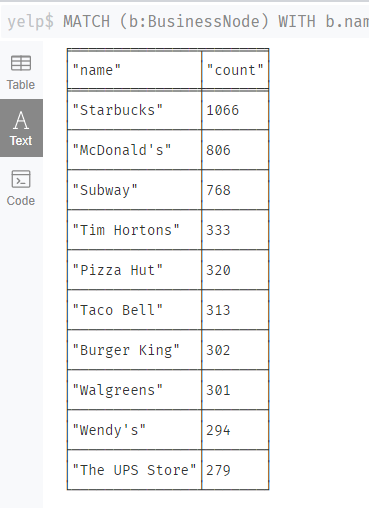


图2.5：查询结果图

### 2-13

#### **题目**

统计评价数大于4000的商家名热度（名字的重复的次数在所有的商家名中的占比），按照评价数量排序，返回热度和商家名和评价数。

#### **解析**

首先筛选出评价数大于4000的商家，使用toInteger函数将business.reviewcount属性转换为整型。接着，使用WITH COUNT(DISTINCT business) AS cnt语句统计所有商家的数量，将数量保存到变量cnt中。然后，再次使用MATCH语句筛选评价数大于4000的商家，并在WITH子句中将这些商家的名字、评价数、以及前面统计得到的商家总数cnt传递给下一步操作。最后，使用WITH子句对商家名、热度（即名字的重复次数在所有商家名中的占比）和评价数进行处理，并使用RETURN语句返回结果。结果按照评价数降序排序，其中每行的内容依次为热度、商家名和评价数。

代码2.14

|  |
| --- |
| MATCH (business:BusinessNode) WHERE toInteger(business.reviewcount) > 4000 WITH COUNT(DISTINCT business) AS cnt MATCH (business:BusinessNode) WHERE toInteger(business.reviewcount) > 4000 WITH business, COUNT(\*) AS count, cnt WITH business.name AS name, count, count\*1.0/cnt AS popularity, business.reviewcount AS reviewcount RETURN popularity, name, reviewcount ORDER BY reviewcount DESC |

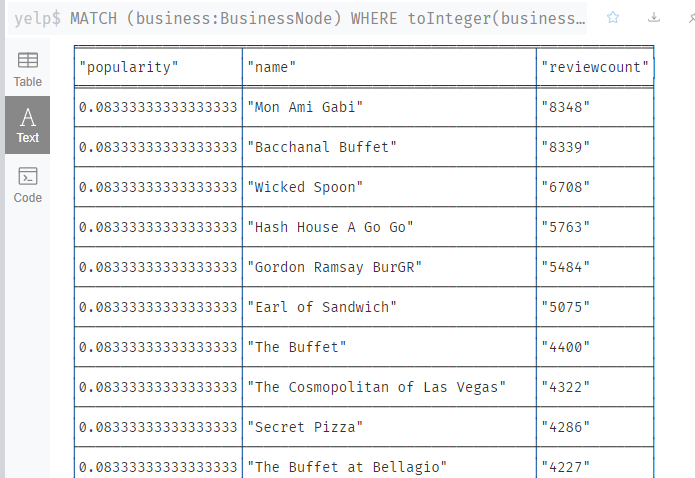


图2.6：查询结果图

### 2-14

#### **题目**

查询具有评分为5.0的Hot Pot类别的商铺所在城市。

#### **解析**

通过MATCH语句匹配所有评分为5.0且分类为Hot Pot的商铺节点和Hot Pot类别节点，然后通过-[:IN\_CATEGORY]->语句指定这两种节点之间的关系为"IN\_CATEGORY"关系，表示商铺属于Hot Pot类别。最后通过RETURN语句返回去重后的商铺所在城市，即business.city。

代码2.15

|  |
| --- |
| MATCH (:UserNode)-[:Review]->(r:ReviewNode)  -[:Reviewed]->(b:BusinessNode)  -[:IN\_CATEGORY]->(c:CategoryNode {category: 'Hot Pot'}) WHERE r.stars = '5.0' RETURN DISTINCT b.city as city |



图2.7：查询结果图

### 2-15

#### **题目**

统计每个用户评价过的商家数量，按照数量降序排列，返回用户id，用户名和评价过的商家的数量（需要对商家去重）。

#### **解析**

首先找到所有的UserNode节点，这些节点和BusinessNode节点通过Reviewed关系相连，并且这些BusinessNode节点和ReviewNode节点之间通过Review关系相连。然后将找到的节点按照user进行分组，并计算每个用户评价过的商家数量，将结果保存为count。

代码2.16

|  |
| --- |
| MATCH (user:UserNode)-[:Review]->(:ReviewNode)  -[:Reviewed]->(b:BusinessNode) WITH user, COUNT(DISTINCT b) AS count RETURN user.userid, user.name, count ORDER BY count DESC LIMIT 10 |

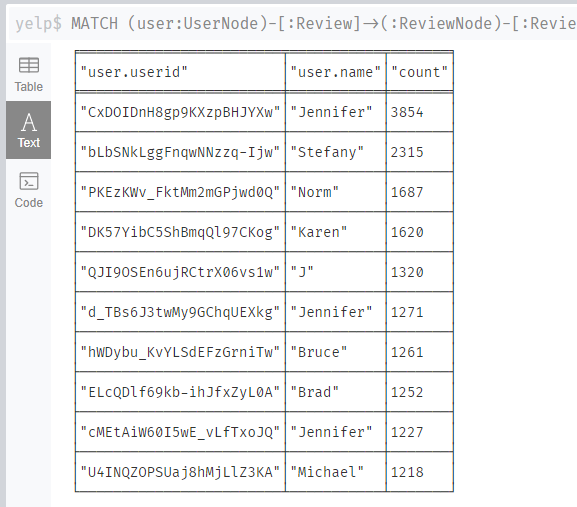


图2.8：查询结果图

### 2-16

#### **题目**

体会建立索引对查询带来的性能提升，但会导致插入，删除等操作变慢（需要额外维护索引代价）。

#### **解析**

1.首先为UserNode增加flag属性，由于Neo4j服务器的Java Heap空间不足，只为fans值大于3000的数据添加flag属性，值等于其fans值

代码2.17

|  |
| --- |
| MATCH (user:UserNode) WHERE toInteger(user.fans) > 300 SET user.flag = user.fans |

2.对UserNode的flag属性执⾏查询（flag>300）

代码2.18

|  |
| --- |
| MATCH (user:UserNode) WHERE toInteger(user.flag) > 300 RETURN user |

3.把所有的flag值改为8001（更新操作）

代码2.19

|  |
| --- |
| MATCH (user:UserNode) WHERE toInteger(user.flag) > 300 SET user.flag = 8001 |

4.删除（flag>8000），删除后查询一下看看，结果为空，删除成功

代码2.20

|  |
| --- |
| MATCH (user:UserNode) WHERE user.flag > 8000 REMOVE user.flag |

5.重新执行操作1，然后在flg属性上建立索引

代码2.21

|  |
| --- |
| CREATE INDEX FOR (user:UserNode) ON (user.flag) // 备注，查找索引名字并删除索引的方法 CALL db.indexes() DROP INDEX `name with no quote` |

6.重复上述查询、修改、删除操作

### 2-17

#### **题目**

查询与用户user1（userid: 0kSXMbNFo7mdwTPj4iQv9A) 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，了解该查询的执行计划及查询执行时间，并给出物理优化手段，以提高查询性能，通过优化前后的性能对比展现优化程度。）

#### **解析**

首先，查询出u1评价过的商家列表，使用COLLECT函数对结果进行聚合，得到一个列表u1\_businesses。然后，匹配所有评价过与u1相同商家的用户u2，并检查u2是否与u1有朋友关系（使用NOT操作符），将结果返回。返回结果中包括u1和u2的名称，以及二者共同评价商家的数量。最后按照共同评价商家数量进行降序排序。

代码2.22

|  |
| --- |
| MATCH (u1:UserNode {userid: '0kSXMbNFo7mdwTPj4iQv9A'})  -[:Review]->(:ReviewNode)-[:Reviewed]->(b:BusinessNode) WITH u1, COLLECT(DISTINCT b) AS u1\_businesses MATCH (u2:UserNode)-[:Review]->(:ReviewNode)-[:Reviewed]->(b:BusinessNode) WHERE NOT (u1)-[:HasFriend]->(u2) AND NOT (u2)-[:HasFriend]->(u1) AND b IN u1\_businesses RETURN u1.name, u2.name, COUNT(b) AS sum ORDER BY sum DESC |

PROFILE查看执行计划后得到如下图

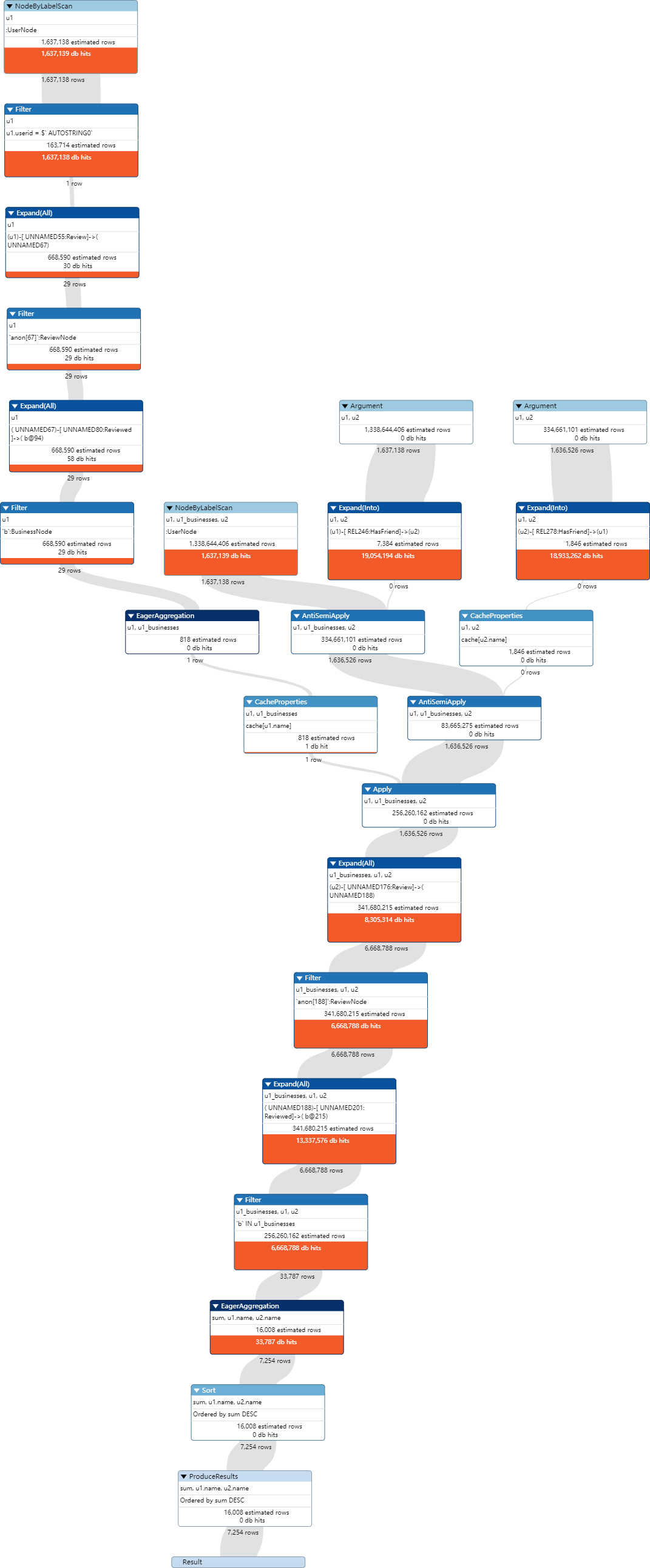


图2.9优化前的查询计划图

运行时间如下

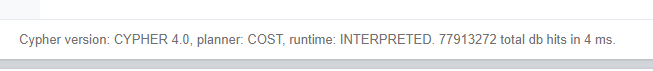


图2.10：运行时间查询

#### **优化**

为UserNode的userid属性和BusinessNode的business\_id属性创建索引，以加速节点的查找和匹配操作。

代码2.23

|  |
| --- |
| CREATE INDEX FOR (user:UserNode) ON (user.userid) CREATE INDEX FOR (b:BusinessNode) ON (b.businessid) |

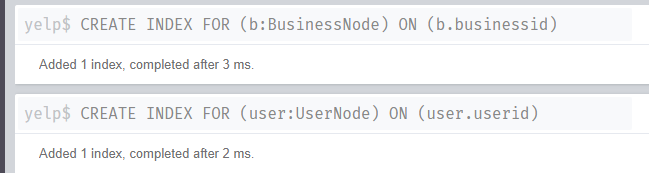


图2.11：建立索引结果图

将子查询中的COLLECT操作改为使用节点标签进行聚合，以减少内存使用。

代码2.24

|  |
| --- |
| MATCH (u1:UserNode {userid: '0kSXMbNFo7mdwTPj4iQv9A'})  -[:Review]->(:ReviewNode)-[:Reviewed]->(b:BusinessNode) WITH u1, COLLECT(DISTINCT b.businessid) AS u1\_businesses MATCH (u2:UserNode)-[:Review]->(:ReviewNode)  -[:Reviewed]->(b:BusinessNode) WHERE NOT (u1)-[:HasFriend]->(u2)  AND NOT (u2)-[:HasFriend]->(u1)  AND b.businessid IN u1\_businesses RETURN u1.name, u2.name, COUNT(b.businessid) AS sum ORDER BY sum DESC |

重新PROFILE查看执行计划

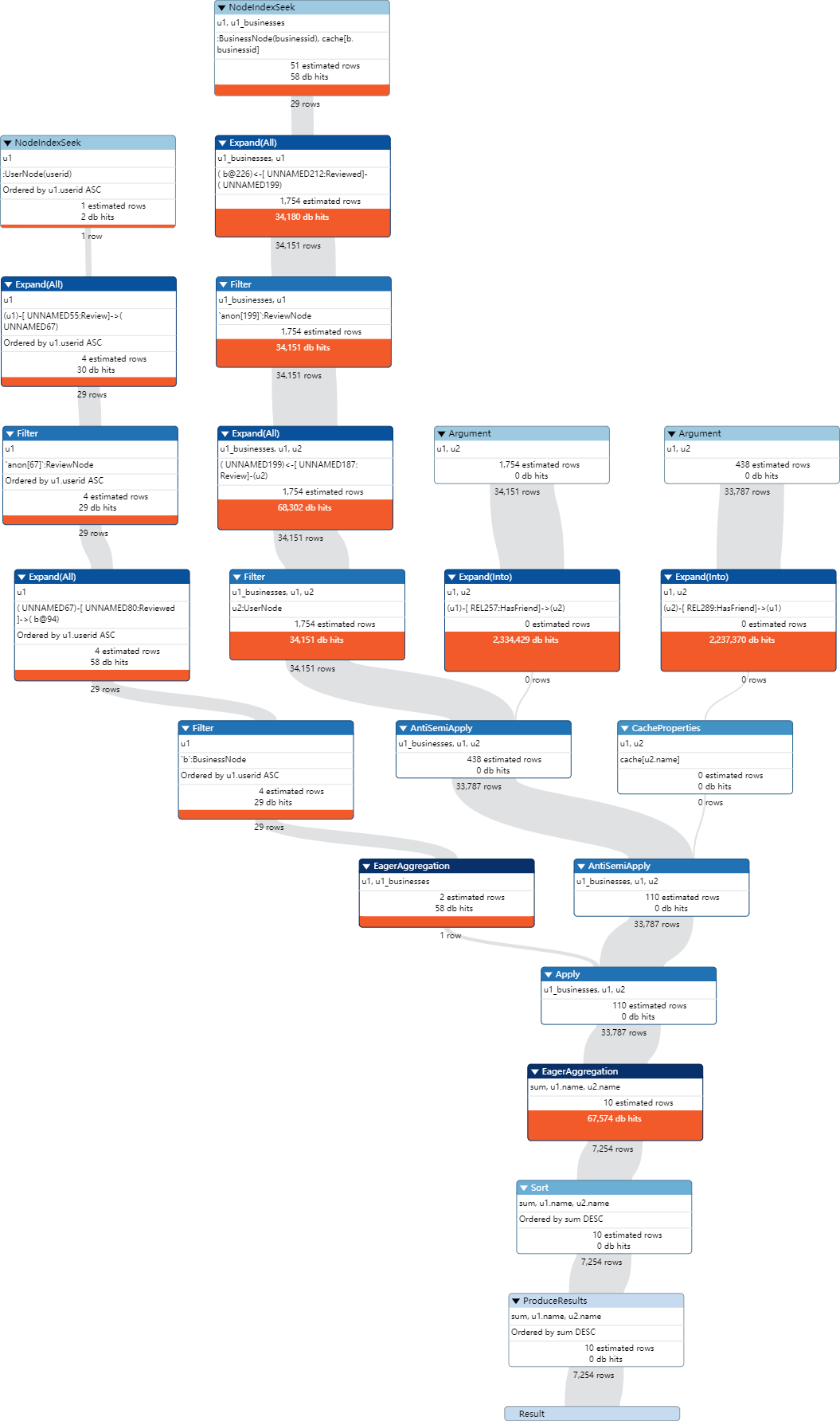


图2.12：建立索引后查询计划图

运行时间如下，可以看到比建立索引前短了特别多

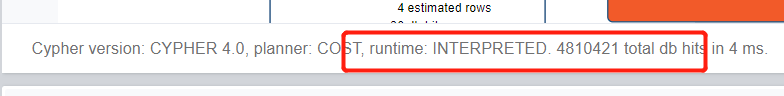


图2.13：建立索引后运行时间图

### 2-18

#### **题目**

分别使用Neo4j和MongoDB查询review\_id为Q1sbwvVQXV2734tPgoKj4Q对应的user信息，比较两者查询时间，指出Neo4j和MongoDB主要的适用场景。

#### **解析**

##### Neo4j

代码2.25

|  |
| --- |
| MATCH (user:UserNode)-[:Review]->(r:ReviewNode {reviewid: 'Q1sbwvVQXV2734tPgoKj4Q'})RETURN user |

##### MongoDB

代码如下

代码2.26

|  |
| --- |
| var review\_uid = db.review.findOne({"review\_id": "Q1sbwvVQXV2734tPgoKj4Q"}).user\_id db.user.findOne({"user\_id": review\_uid}).explain() |

##### 比较

* Neo4j：适用于复杂的图形结构数据查询，如社交网络、推荐系统、知识图谱等。由于其使用基于图论的查询语言Cypher，支持关系型数据的快速查询和分析。
* MongoDB：适用于海量非结构化或半结构化数据存储和查询，如日志、传感器数据、文档数据库等。由于其使用文档数据库模型，支持高效的数据插入和查询，并支持分布式数据库集群的横向扩展

## 任务三：多数据库交互应用实验

### 3-1

#### **题目**

使用Neo4j查找：找出包含的商家种类超过10类的城市（记得去重），并在Neo4j以表格形式输出满足以上条件的每个城市中的商家信息：城市，商家名称，商家地址。

#### **解析**

代码3.1

|  |
| --- |
| MATCH (c:CityNode)<-[:IN\_CITY]-(b:BusinessNode)-[:IN\_CATEGORY]->(a:CategoryNode) WITH c, count(DISTINCT a) AS category\_count WHERE category\_count > 10 MATCH (c)<-[:IN\_CITY]-(b:BusinessNode) RETURN c.city AS city, b.name AS name, b.address AS address |

### 3-2

#### **题目**

将1得到的结果导入MongoDB，并使用该表格数据，统计其中所有出现的商家名及该商家名对应的出现次数，并按照出现次数降序排序:  
(1) 使用aggregate和mapreduce两种方式实现  
(2) 比较这两种方式的执行效率并分析其原因。

#### **解析**

##### Neo4j结果导入MongoDB

1.将刚刚neo4j中的输出结果用csv格式下载到本地。打开终端，cd到刚刚csv保存的目录下，输入以下命令将数据导入到服务器上。

* 这里的3-1.csv是刚刚保存的输出结果的文件名
* root@124.71.146.178是服务器用户名和IP地址
* :/root/ 代表上传目标路径

代码3.2

|  |
| --- |
| scp ./3-1.csv root@124.71.146.178:/root/ |

运行命令后，输入服务器密码即可，如下

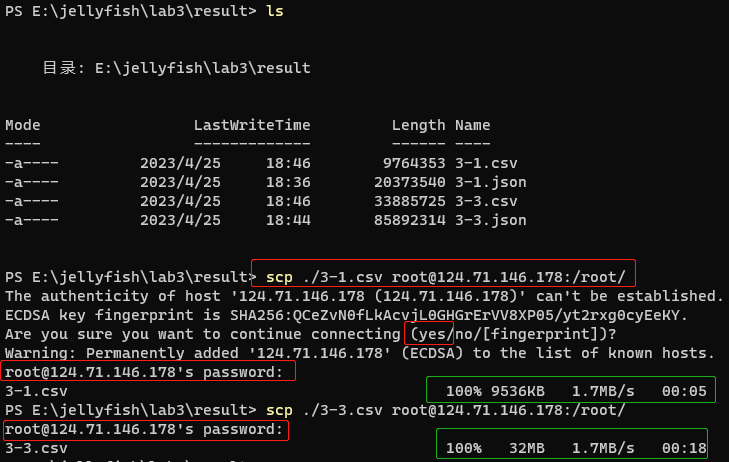


图3.1：本地上传文件到服务器

在服务器上可以看到root/下以有我们导入的数据

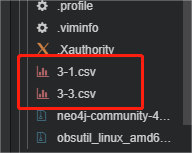


图3.2：上传结果图

2.切换到mongoDB的yelp数据集并创建一个新的集合，这里叫做CityBusiness

代码3.3

|  |
| --- |
| use yelp db.createCollection("CityBusiness") show collections |

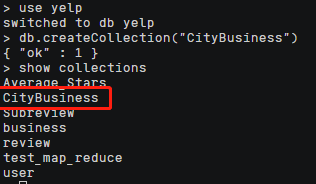


图3.3：创建集合结果

3.退出mongoDB，回到~，把数据导入到mongoDB中的yelp数据集的CityBusiness集合中。

代码3.4

|  |
| --- |
| mongoimport -d=yelp -c=CityBusiness --type=csv --headerline ./3-1.csv |

运行后显示导入成功

fig:

图3.4：新数据导入yelp数据集

##### 数据统计

1. Aggregate  
按照name字段对文档进行分组，使用*$sum*操作符对每组文档的数量进行求和，生成一个新的count字段。

代码3.5

|  |
| --- |
| db.CityBusiness.aggregate([  { $group: { \_id: '$name', count: { $sum: 1 } } },  { $sort: { count: -1 } } ]) |

2. Mapreduce

代码3.6

|  |
| --- |
| db.CityBusiness.mapReduce(  function() {  emit(this.name, 1);  },  function(key, values) {  return Array.sum(values);  },  {  out: { inline: 1 },  finalize: function(key, reducedValue) {  return { count: reducedValue };  }  } ) |

##### 比较分析

结果是aggregate方法特别快、无等待感，但是mapreduce要特别久，下图是mapreduce输出结果

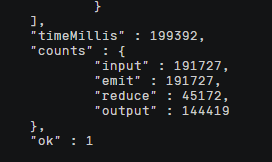


图3.5：mapreduce执行结果

分析如下

* 操作数据的过程中不需要频繁地读写磁盘，aggregate操作可以在内存中完成。相比之下，mapreduce 需要对所有输入数据进行读取和写入，对磁盘的操作比较频繁，效率相对较低。
* aggregate 支持多个操作符的串联使用，因此可以将多个操作合并在一起执行，减少操作次数和磁盘 I/O。而 mapreduce 的过程中，由于需要进行两个阶段的操作，会产生较多的数据中间输出和磁盘读写操作，导致效率相对较低。

### 3-3

#### **题目**

在Neo4j中查找所有商家，要求返回商家的名字，所在城市、商铺类。  
(1) 将查找结果导入MongoDB中实现对数据的去重（提示：使用aggregate，仅保留城市、商铺类型即可）  
(2) 将去重后的结果导入Neo4j中的新库result中，完成（City-[Has]->Category）图谱的构建。

#### **解析**

1.Neo4j数据库查询操作

代码3.7

|  |
| --- |
| MATCH (b:BusinessNode)-[:IN\_CATEGORY]->(c:CategoryNode) RETURN b.name AS business\_name, b.city AS city, c.category AS category |

2.导入服务器和MongoDB操作不再赘述，请看本文档的3-2章节。这里新的集合名叫做BusinessAll

3.去重操作  
使用*$group*将BusinessAll集合中所有的文档按照city和category字段进行分组，然后用*$forEach*将前面结果中的数据插入到BusiDistinct集合中；BusiDistinct集合中的所有文档，即为不重复的城市和类别组合。

代码3.8

|  |
| --- |
| db.createCollection("BusiDistinct") db.BusinessAll.aggregate([  { $group: { \_id: { city: '$city', category: '$category' } } } ]).forEach((item) => { db.BusiDistinct.insert( item.\_id ) } ) |

查看结果

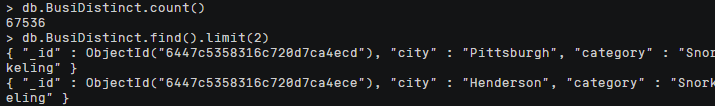


图3.5：去重并导入新集合

4.导出BusiDistinct集合的内容为csv文件

代码3.9

|  |
| --- |
| mongoexport -d yelp -c BusiDistinct --type=csv --fields city,category --out result.csv |

5.上一步导出后，result.csv文件位于~路径下，用cp命令把他放到neo4j安装目录的的import路径下

代码3.10

|  |
| --- |
| cd ~/neo4j-community-4.0.9/import cp /root/result.csv ./ |

6.在neo4j网页数据库中输入以下命令，要对空值做处理

代码3.11

|  |
| --- |
| LOAD CSV WITH HEADERS FROM "file:///result.csv" AS f MERGE (c:CityNode {city: COALESCE(f.city, "")}) MERGE (a:CategoryNode {category: COALESCE(f.category, "")}) CREATE (c) -[:Has]-> (a) |

查看一下图谱

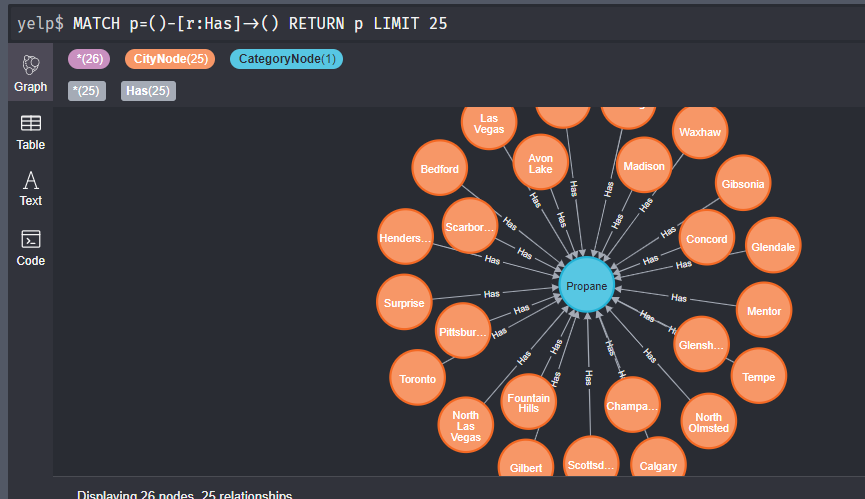


图3.6：新建图谱图

## 任务四：不同类型数据库MVCC多版本并发控制对比实验

### MySQL

创建一个叫做testdb的数据库

代码4.1

|  |
| --- |
| create database testdb use testdb |

在testdb数据库中创建一个新表以用于后续实验，设置engine=InnoDB;

代码4.2

|  |
| --- |
| create table `Lingjing` (  `id` int(11) not null,  `idol` varchar(20) not null,  `age` int(5) not null,  primary key(`id`) ) engine=InnoDB; |

设置事务隔离等级为可重复读

代码4.3

|  |
| --- |
| set session transaction isolation level repeatable read; |

向表中插入一些数据

代码4.4

|  |
| --- |
| insert into Lingjing (`id`, `idol`, `age`) values (1, 'Yunjin', 100); insert into Lingjing (`id`, `idol`, `age`) values (2, 'Kazuha', 19); insert into Lingjing (`id`, `idol`, `age`) values (3, 'Yeji', 22); insert into Lingjing (`id`, `idol`, `age`) values (4, 'Lia', 22); insert into Lingjing (`id`, `idol`, `age`) values (5, 'Jennie', 26); insert into Lingjing (`id`, `idol`, `age`) values (6, 'Jennie', 26); |

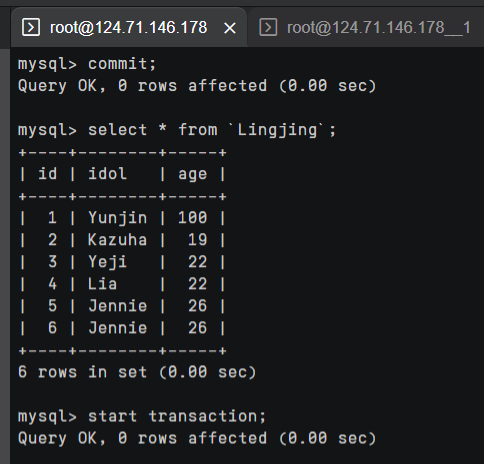


图4.1：插入结果图

开始一个事务

代码4.5

|  |
| --- |
| start transaction; |

再开一个终端，修改idol = Jnenie的数据的age为101

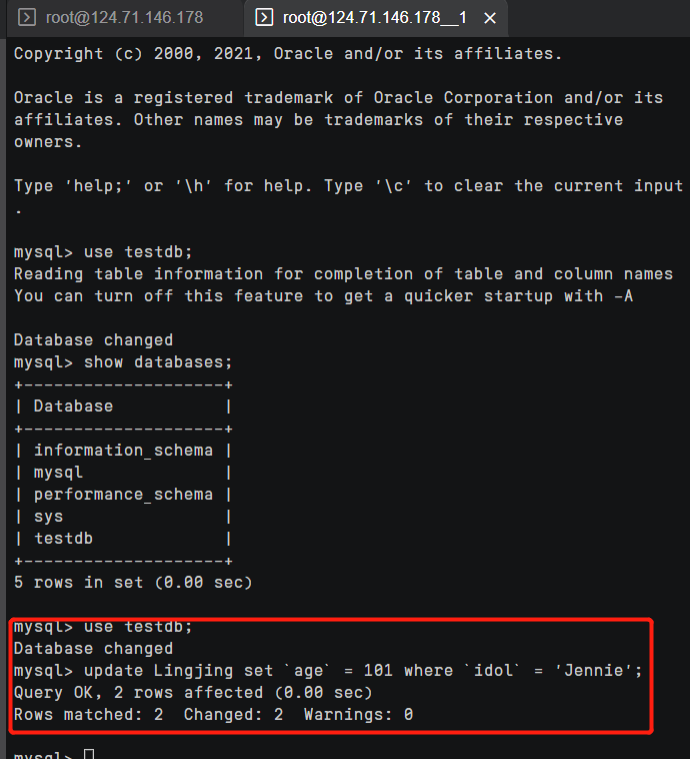


图4.2：更新结果图

在原来的终端里查询数据，发现已经可以查询到修改的数据，如下所示。

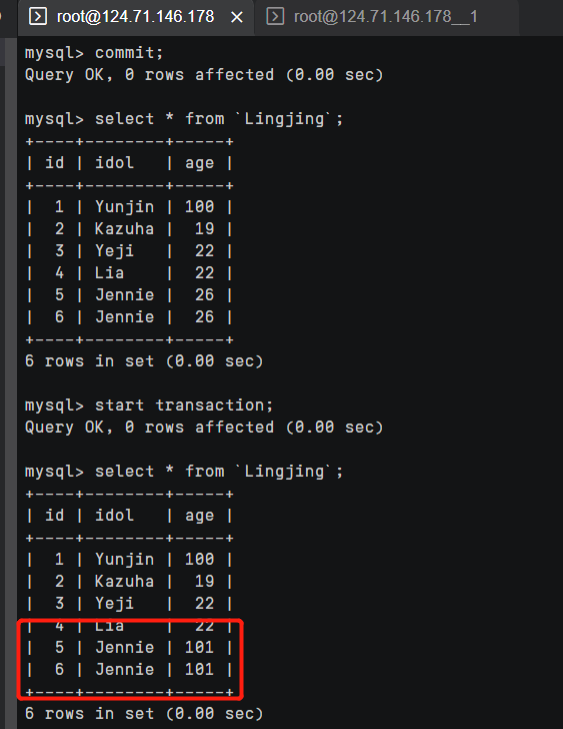


图4.3：查询到修改后的数据

在原来的终端里修改数据，修改*idol = Jnenie*的数据的*age*为203

代码4.6

|  |
| --- |
| update Lingjing set `age` = 233 where `idol` = 'Jennie'; |

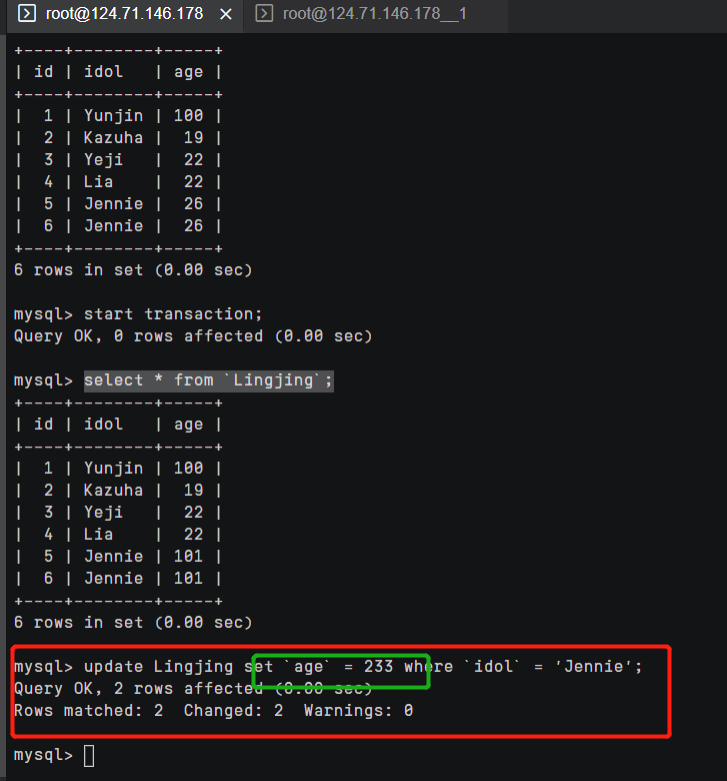


图4.4：修改数据

先不要commit，在另一个终端中查看一下表中的数据，读到的是原来的数据101

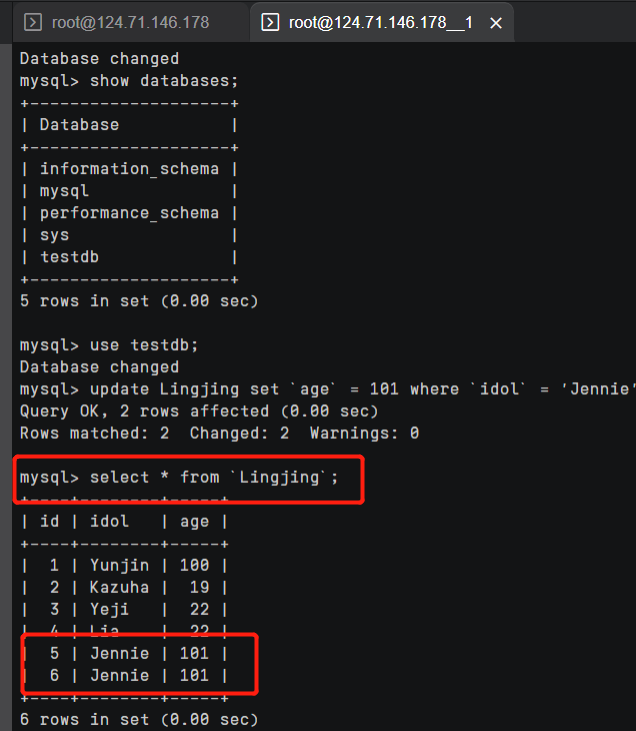


图4.5：读到原来的值

回到原来的终端中进行commit，到另一个终端中查询，已经可以查到修改后的数据。

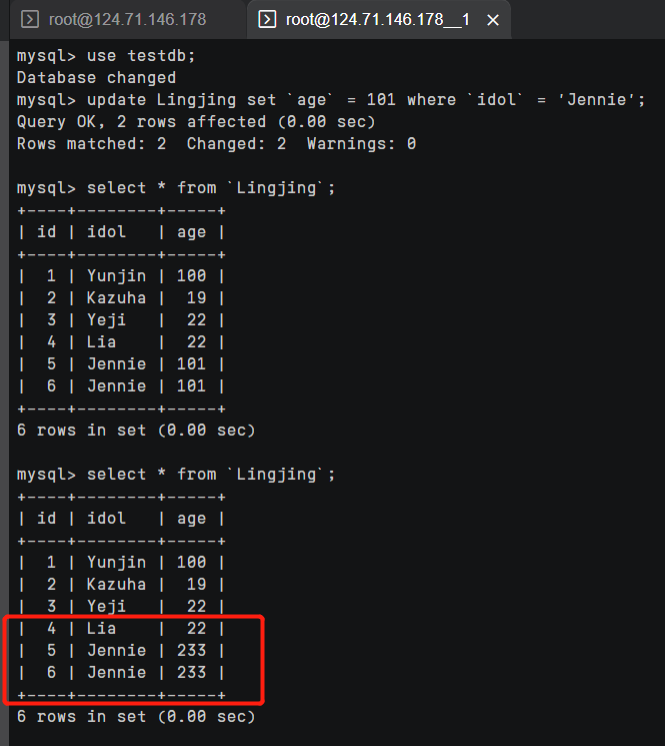


图4.6：commit后读到新值

### MongoDB

分片教程大部分按助教的文档来就行，以下是修改及注意点：

* MongoDB 4.4版本安装，只要把软件源list文件里所有的3.2换成4.4就行
* *su - mongodb* 失败没有得到预期结果的话先执行 *su root* 就行
* 初始化replica set的时候实验文档里的shard1写成shard2了，端口号也要改成27017
* 使用mongos时，端口号是前面设置的30000不是20000，执行addshard时字符串里的逗号之间不能有空格，IP换成自己的三个 （教程后两个ip是一样的，它写错了）。

使用mongos，连接上两个服务器，在第一个连接终端里创建testdb数据，并创建一个集合，创建完后插入一些数据，效果如下

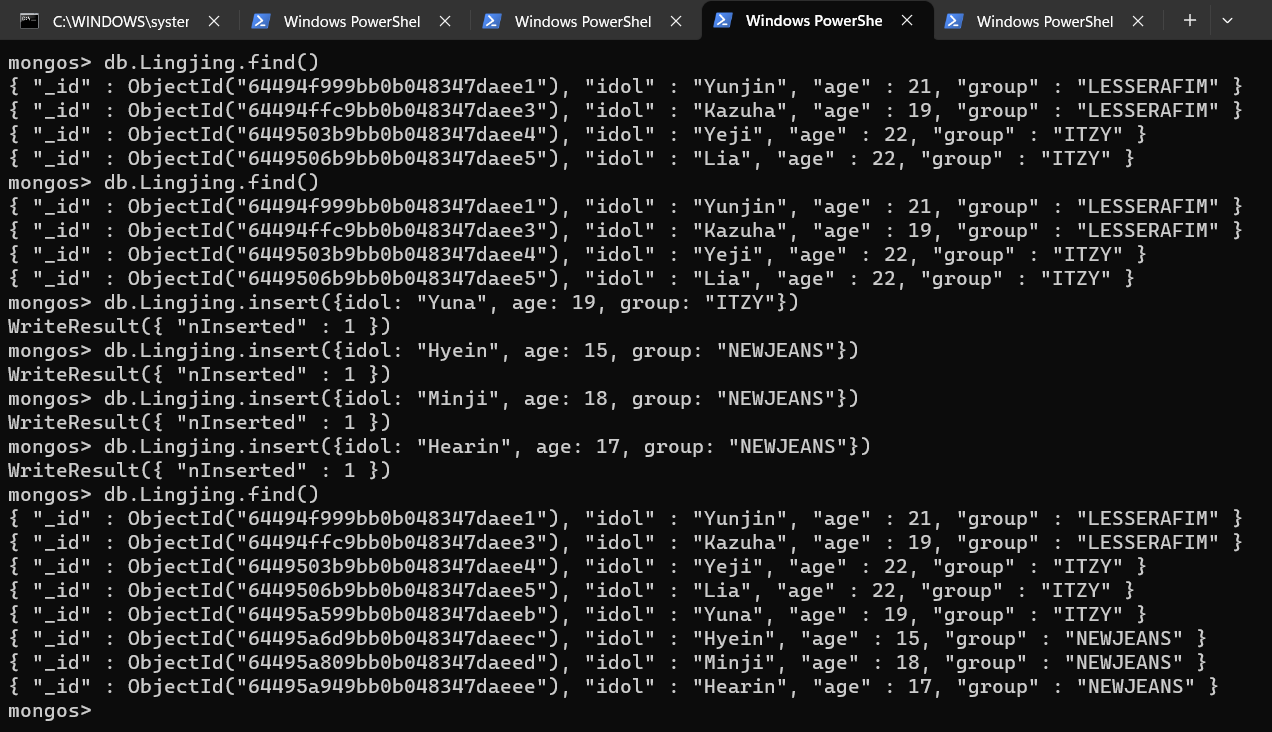


图4.7：插入结果图

在另一个终端窗口中查询数据

代码4.7

|  |
| --- |
| db.getSiblingDB("testdb").getCollection("Lingjing").find({}); |

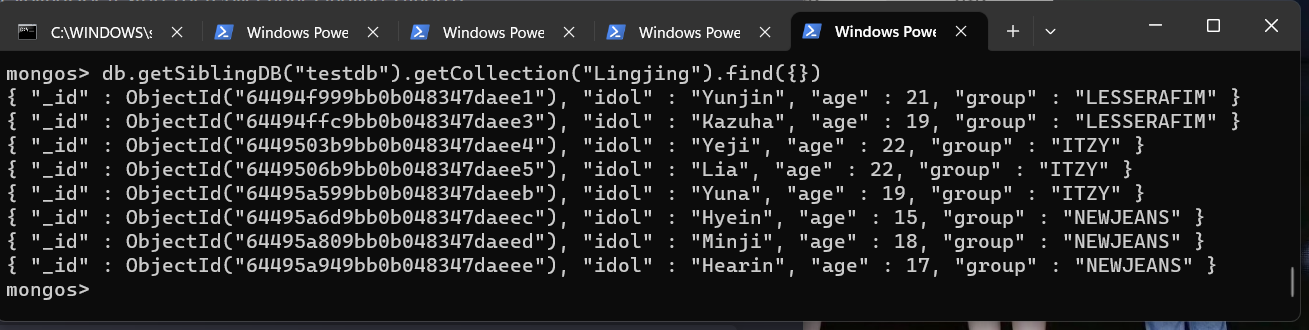


图4.8：在新端口里查询数据

删除一条数据

代码4.8

|  |
| --- |
| db.getSiblingDB("testdb")  .getCollection("Lingjing")  .deleteOne({  \_id: ObjectId("64495a949bb0b048347daeee")  }) |

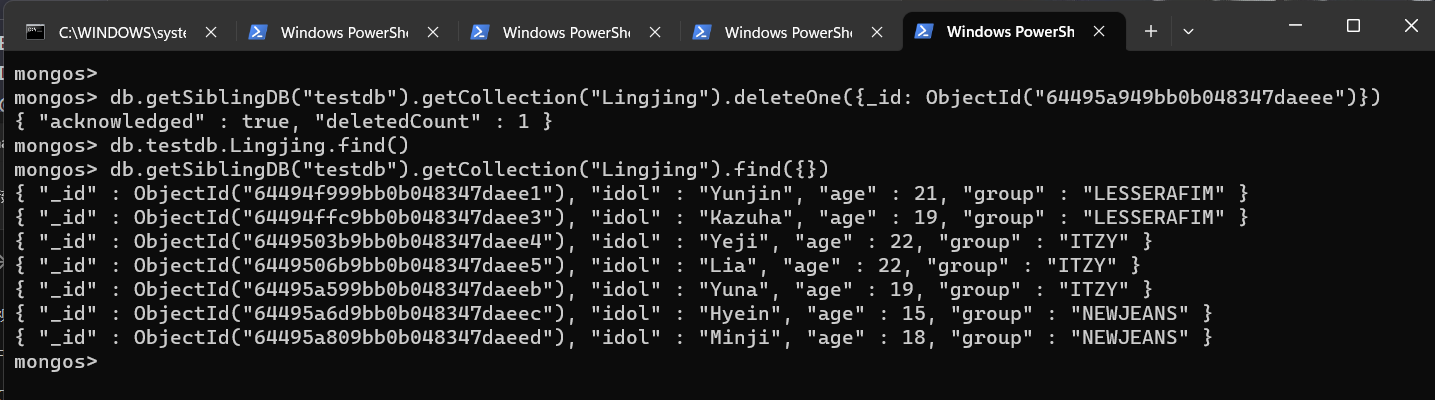


图4.9：新端口中删除数据

回到第一个终端窗口中查询数据，可以发现已经删除成功

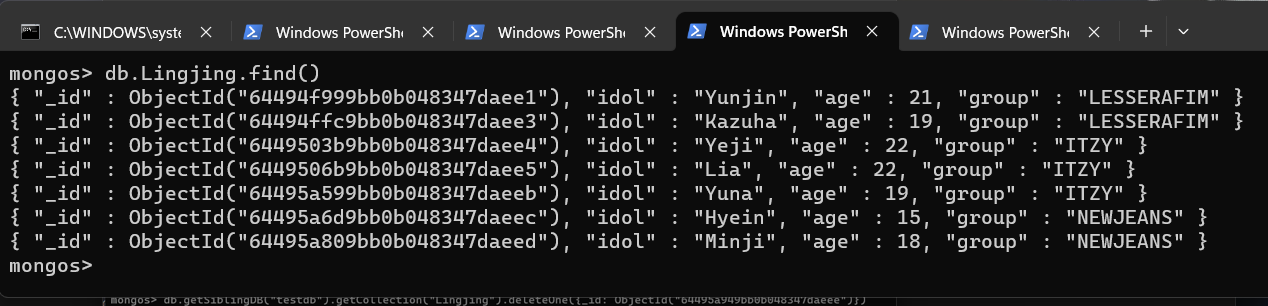


图4.10：原端口中查询数据

### 不同之处

MySQL的MVCC是通过在每行数据后面存储版本号来实现的。每次修改数据时，MySQL会将新的版本号与修改后的数据一起写入数据库，保留旧版本的数据。这样，在并发读取数据时，MySQL可以使用版本号来判断读取哪个版本的数据，从而实现MVCC。

MongoDB的MVCC是通过在每个文档上保留历史版本来实现的。MongoDB中的文档是一组键值对，每个文档都有一个唯一的\_id字段作为主键。在MongoDB中，每个文档都会维护一个“版本号”，用来表示该文档最近一次被修改的版本。

