

华中科技大学

大数据管理实验报告

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
1-1	4
1-2	4
1-3	4
1-4	4
1-5	4
1-6	4
1-7	5
1-8	5
1-9	6
1-10	7
1-11	8
1-12	9
1-13	10
1-14	10
1-15	11
任务二：Neo4j 实验.....	13
2-1	13
2-2	13
2-3	13
2-4	13
2-5	13
2-6	13

2-7	13
2-8	13
2-9	15
2-10	15
2-11	15
2-12	16
2-13	17
2-14	18
2-15	18
2-16	19
2-17	20
2-18	24
任务三：多数据库交互应用实验	25
3-1	25
3-2	25
3-3	28
任务四：不同类型数据库 MVCC 多版本并发控制对比实验.....	30
MySQL	30
MongoDB	32
不同之处.....	33

任务一：MongoDB 实验

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 题目较简单，略

1-2

题目 查询 business 集合中 city 是 Goodyear 的 5 条数据。

解析 题目较简单，略

1-3

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

解析 题目较简单，略

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 题目较简单，略

1-5

题目 查询 user 集合中 $5 \leq cool < 10$ 且 $useful \geq 20$ 的 user，限制 10 条。

解析 题目较简单，略

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

```
      "winningPlan" : {
        "stage" : "COUNT"
      },
      "rejectedPlans" : [ ]
    },
    "executionStats" : {
      "executionSuccess" : true,
      "nReturned" : 0,
      "executionTimeMillis" : 0,
      "totalKeysExamined" : 0,
      "totalDocsExamined" : 0,
      "executionStages" : {
        "stage" : "COUNT",
        "nReturned" : 0,
        "executionTimeMillisEstimate" : 0,
```

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 题目较简单，略

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。

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 1.1

```
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 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 1.2

```
// 没有建立索引时的 explain 结果  
> "stage": "COLLSCAN"  
> "queryPlanner": {  
  "indexFilterSet": false,
```

```
"winningPlan": {
  "stage": "COLLSCAN",},}
> "executionStats": {
  "executionSuccess": true, "executionTimeMillis": 51,
  "totalDocsExamined": 192609,
  "executionStages": {
    "stage": "COLLSCAN",}}
```

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 1.3

```
// 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})
```

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 1.4

```
// 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",
      "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。

```
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。

```
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。

```
> show collections
business
review
test_map_reduce
user
> db.review.aggregate([{$limit: 500000}, {$out: "Subreview"}])
> show collections
Subreview
business
review
test_map_reduce
user
```

图 1.2: 创建新集合的结果图

```
> db.Subreview.find().count()
500000
```

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。对评论的内容建立全文索引, 对 useful 建立升序索引

代码 1.7

```
// 对评论的内容建立全文索引
db.Subreview.createIndex({text: "text"})

// 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})
```

查询

代码 1.8

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

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 1.9

```
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

```
> db.Subreview.aggregate([
...   {$match: {useful: {$gt: 5}, funny: {$gt: 5}, cool: {$gt: 5}}},
...   {$group: {_id: "$business_id", average_stars: {$avg: "$stars"}}},
...   {$sort: {_id: -1}},
...   {$group: {_id: null, count: {$sum: 1}}}
... ])
{ "_id" : null, "count" : 1894 }
```

图 1.4: 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。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）

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 1.10

```
db.business.find({
  loc: { $near: { $geometry: { type: "Point", coordinates: [-112.39559635
52, 33.4556129678]}, $maxDistance: 120 } }
}, { _id: 0, name: 1, address: 1, stars: 1})
```

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 1.11

```
db.Subreview.createIndex({user_id: 1, date: 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 1.12

```
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。

```

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 };
    }
  }
)

```

执行结果如下

```

{
  "result" : "Average_Stars",
  "timeMillis" : 5900,
  "counts" : {
    "input" : 500000,
    "emit" : 500000,
    "reduce" : 133512,
    "output" : 18983
  },
  "ok" : 1
}

```

图 1.5: 查询结果展示

查看一下 Average_Stars 集合中的数据

```

> show collections
Average_Stars
Subreview
business
review
test_map_reduce
user
> db.Average_Stars.find()
{ "_id" : "--Gc998IMjLn8yr-HTz6Ug", "value" : { "avgStars" : 3 } }
{ "_id" : "--I7YYLada0tSLkORTHb5Q", "value" : { "avgStars" : 3.4642857142857144 } }
{ "_id" : "--U98MNlDym2cLn36BBPgQ", "value" : { "avgStars" : 2.3333333333333335 } }
{ "_id" : "--j-kaNMCo1-DYzddCsA5Q", "value" : { "avgStars" : 4 } }
{ "_id" : "--wIGbLEhlpl_UeAIyDmZQ", "value" : { "avgStars" : 4.3 } }
{ "_id" : "--000aQFeK6tqVLndf7x0Rg", "value" : { "avgStars" : 5 } }

```

图 1.6: 结果展示

任务二：Neo4j 实验

2-1

题目 查询标签是 UserNode 的节点，限制 10 个

解析 题目较简单，略

2-2

题目 查询城市是 Chandler 的商家节点。

解析 题目较简单，略

2-3

题目 查询 reviewid 是 xkVveYJIL1Eiwl46cP_VBg 对应的 bussiness 信息。

解析 题目较简单，略

2-4

题目 评价过 businessid 是 5ykOWYZ44sUvu9qx D8rPeg 商家的用户名和粉丝数。

解析 题目较简单，略

2-5

题目 被 userid: 0kSXMbNFo7mdwTPj4iQv9A 的用户评论为 5 星的商家名和地址。

解析 题目较简单，略

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 题目较简单，略

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 题目较简单，略

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 2.1

```
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

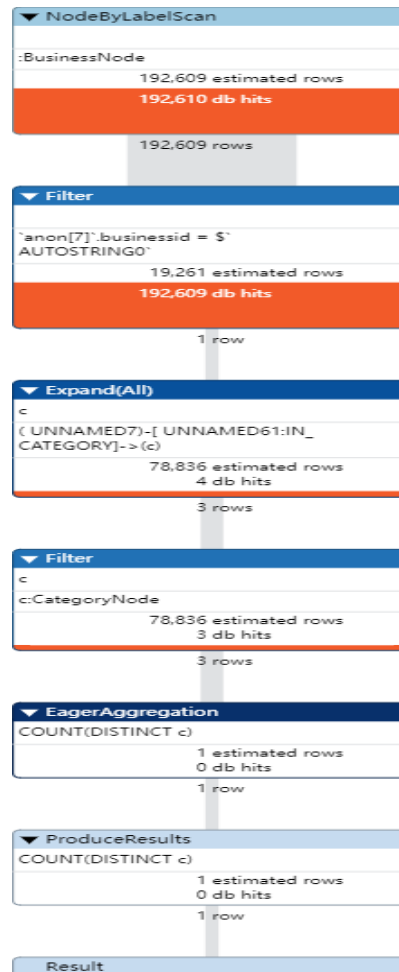


图 2.1: 查询计划图

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 2.2

```
MATCH (:BusinessNode {businessid: '5ykOWYZ44sUvu9qxD8rPeg'})  
-[:IN_CATEGORY]->(c:CategoryNode) RETURN COLLECT(c.category)
```

用 collect()返回, 查看结果, 确实是以 list 的形式返回

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 2.3

```
MATCH(:UserNode{name:'Victoria'})-[:HasFriend]->(friend)  
WITH friend.name as friendsList, size((friend)-[:HasFriend]-()) as number  
ofFoFs RETURN friendsList,numberofFoFs
```

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 2.4

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

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 2.5

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


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 2.6

```
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
```

Table	"popularity"	"name"	"reviewcount"
Text	0.08333333333333333	"Mon Ami Gabi"	"8348"
Code	0.08333333333333333	"Bacchanal Buffet"	"8339"
	0.08333333333333333	"Wicked Spoon"	"6708"
	0.08333333333333333	"Hash House A Go Go"	"5763"
	0.08333333333333333	"Gordon Ramsay BurGR"	"5484"
	0.08333333333333333	"Earl of Sandwich"	"5075"
	0.08333333333333333	"The Buffet"	"4400"
	0.08333333333333333	"The Cosmopolitan of Las Vegas"	"4322"
	0.08333333333333333	"Secret Pizza"	"4286"
	0.08333333333333333	"The Buffet at Bellagio"	"4227"

图 2.2: 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 2.7

```
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
```

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 2.8

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

Table			
Text			
Code			
	"user.userid"	"user.name"	"count"
	"CxDOIDnH8gp9KXzpBHJYXw"	"Jennifer"	3854
	"bLbSNkLggFnqwNNzzq-Ijw"	"Stefany"	2315
	"PKEzKWv_FktMm2mGPjwd0Q"	"Norm"	1687
	"DK57YibC5ShBmqQL97CKog"	"Karen"	1620
	"QJI90SEn6ujRCtrX06vs1w"	"J"	1320
	"d_TBs6J3twMy9GChqUEXkg"	"Jennifer"	1271

图 2.3: 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 2.9

```
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 2.10

```
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 2.11

```
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 2.12

```
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 2.13

```
CREATE INDEX FOR (user:UserNode) ON (user.flag)
```

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 2.14

```
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

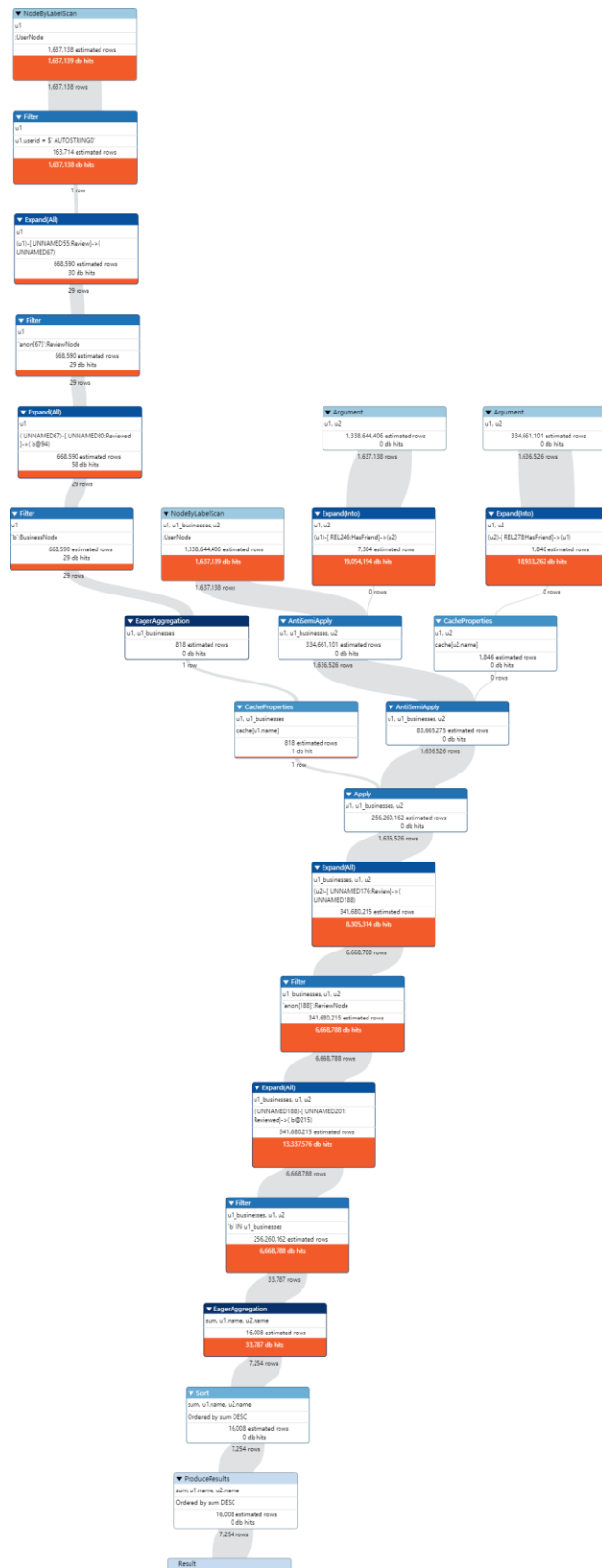


图 2.4 优化前的查询计划图

运行时间如下

```
Cypher version: CYPHER 4.0, planner: COST, runtime: INTERPRETED. 77913272 total db hits in 4 ms.
```

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 2.15

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

```
yelp$ CREATE INDEX FOR (b:BusinessNode) ON (b.businessid)
```

```
Added 1 index, completed after 3 ms.
```

```
yelp$ CREATE INDEX FOR (user:UserNode) ON (user.userid)
```

```
Added 1 index, completed after 2 ms.
```

图 2.5: 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 2.16

```
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

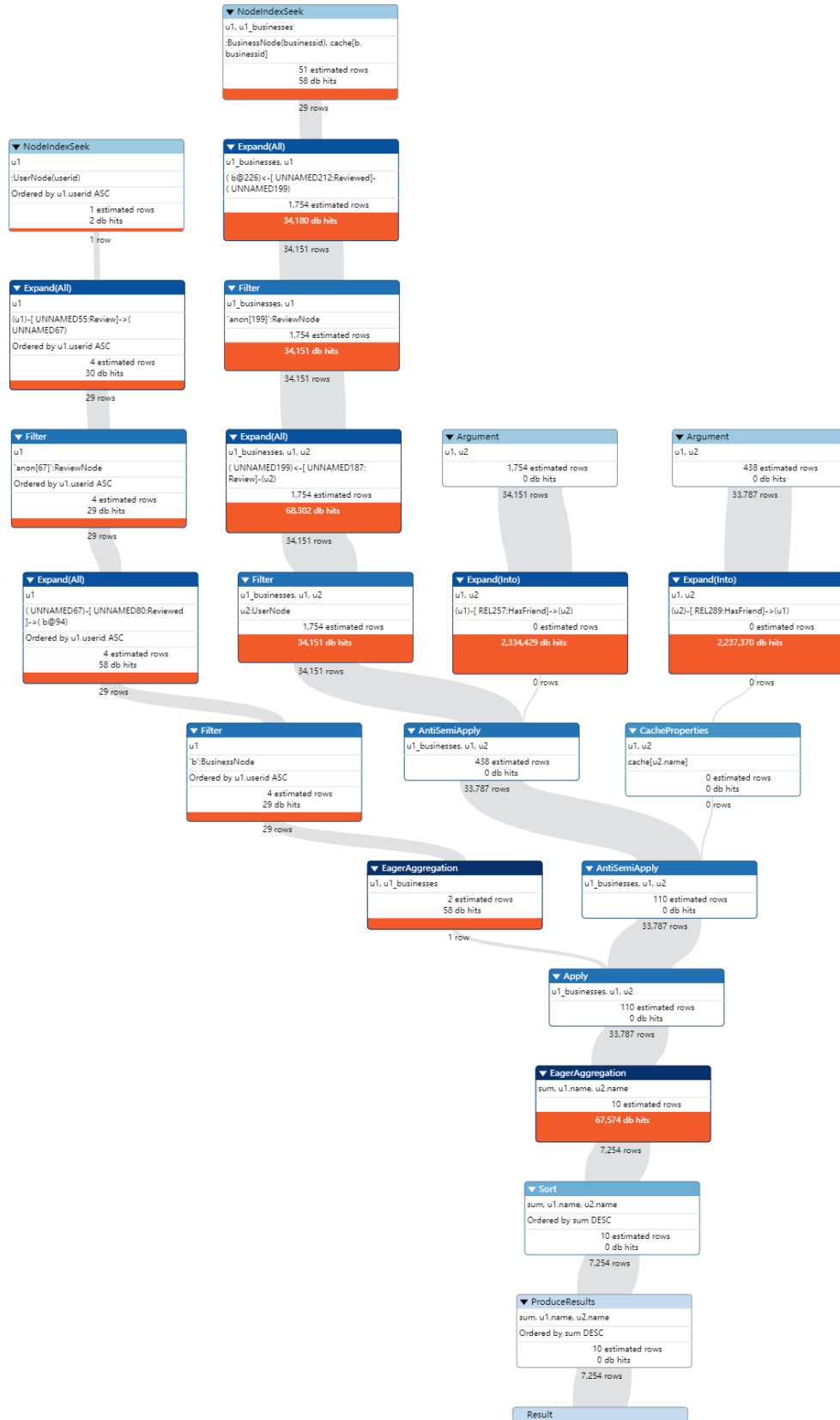


图 2.6: 建立索引后查询计划图

运行时间如下，可以看到比建立索引前短了特别多

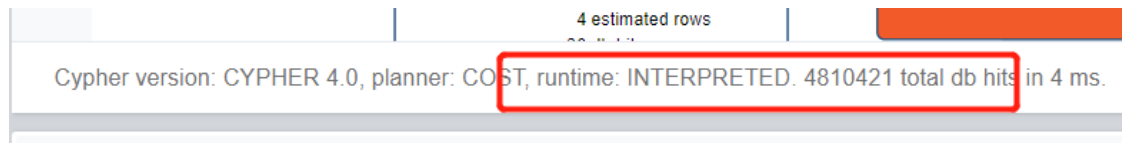


图 2.7：建立索引后运行时间图

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 2.17

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

MongoDB

代码如下

代码 2.18

```
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

```
PS E:\jellyfish\lab3\result> ls

目录: E:\jellyfish\lab3\result

Mode                LastWriteTime         Length Name
----                -
-a----             2023/4/25      18:46          9764353 3-1.csv
-a----             2023/4/25      18:36          20373540 3-1.json
-a----             2023/4/25      18:46          33885725 3-3.csv
-a----             2023/4/25      18:44          85892314 3-3.json

PS E:\jellyfish\lab3\result> scp ./3-1.csv root@124.71.146.178:/root/
The authenticity of host '124.71.146.178 (124.71.146.178)' can't be established.
ECDSA key fingerprint is SHA256:QCeZvN0fLkAcvjL0GHGrErVv8XP05/yt2rxg0cyEeKY.
Are you sure you want to continue connecting (yes/no/[fingerprint])?
Warning: Permanently added '124.71.146.178' (ECDSA) to the list of known hosts.
root@124.71.146.178's password:
3-1.csv                               100% 9536KB   1.7MB/s   00:05
PS E:\jellyfish\lab3\result> scp ./3-3.csv root@124.71.146.178:/root/
root@124.71.146.178's password:
3-3.csv                               100% 32MB     1.7MB/s   00:18
```

图 3.1: 本地上传文件到服务器

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

```
db.createCollection("CityBusiness")
```

```
> use yelp
switched to db yelp
> db.createCollection("CityBusiness")
{ "ok" : 1 }
> show collections
Average Stars
CityBusiness
Subreview
business
review
test_map_reduce
user
```

图 3.2: 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

```
root@ashleytung:~# mongoimport -d=yelp -c=CityBusiness --type=csv --headerline ./3-1.csv
2023-04-25T19:16:33.149+0800 connected to: localhost
2023-04-25T19:16:34.861+0800 imported 191727 documents
```

图 3.3: 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 } }])
```

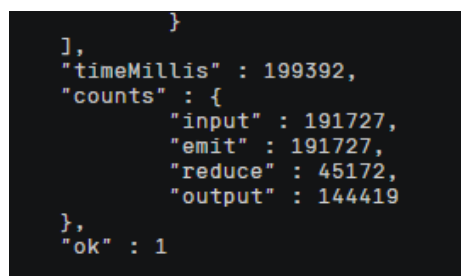
2. Mapreduce

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



```
}
],
"timeMillis" : 199392,
"counts" : {
  "input" : 191727,
  "emit" : 191727,
  "reduce" : 45172,
  "output" : 144419
},
"ok" : 1
```

图 3.4: 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

```
> db.BusiDistinct.count()
67536
> db.BusiDistinct.find().limit(2)
{ "_id" : ObjectId("6447c5358316c720d7ca4ecd"), "city" : "Pittsburgh", "category" : "Snorkeling" }
{ "_id" : ObjectId("6447c5358316c720d7ca4ece"), "city" : "Henderson", "category" : "Snorkeling" }
```

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 --output 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

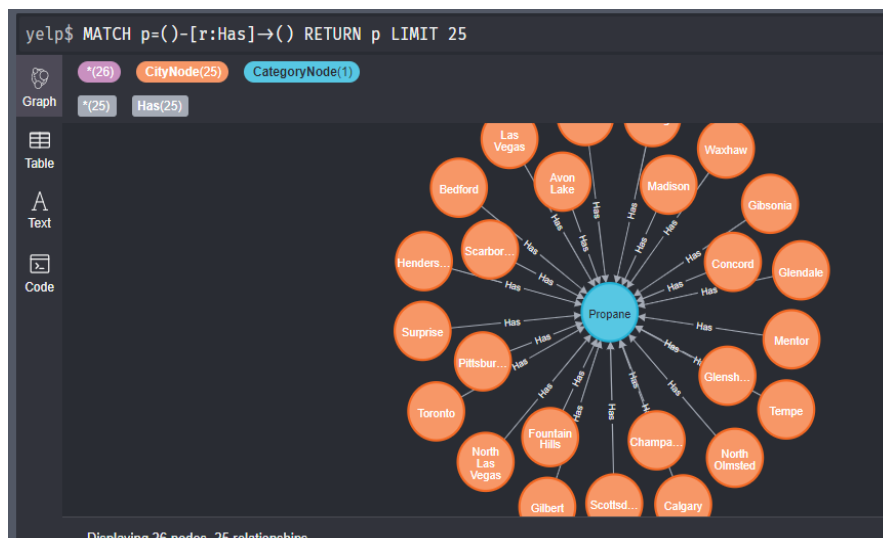


图 3.6：新建图谱图

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

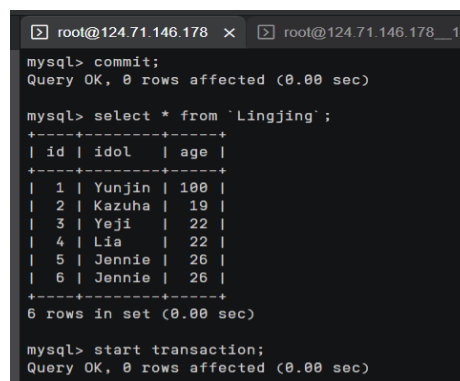
```
create table `Lingjing` (  
  `id` int(11) not null, `idol` varchar(20) not null,  
  `age` int(5) not null, primary key(`id`)  
) engine=InnoDB;
```

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



```
mysql> commit;  
Query OK, 0 rows affected (0.00 sec)  
  
mysql> select * from `Lingjing`;  
+----+-----+-----+  
| id | idol  | age |  
+----+-----+-----+  
| 1  | Yunjin | 100 |  
| 2  | Kazuha | 19  |  
| 3  | Yeji   | 22  |  
| 4  | Lia    | 22  |  
| 5  | Jennie | 26  |  
| 6  | Jennie | 26  |  
+----+-----+-----+  
6 rows in set (0.00 sec)  
  
mysql> start transaction;  
Query OK, 0 rows affected (0.00 sec)
```

图 4.1：插入结果图

开始一个事务

再开一个终端，修改 idol = Jnenie 的数据的 age 为 101

```
mysql> use testdb;
Database changed
mysql> update Lingjing set `age` = 101 where `idol` = 'Jennie';
Query OK, 2 rows affected (0.00 sec)
Rows matched: 2  Changed: 2  Warnings: 0
```

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。

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

mysql> select * from 'Lingjing';
+----+-----+-----+
| id | idol  | age |
+----+-----+-----+
| 1  | Yunjin | 100 |
| 2  | Kazuha | 19  |
| 3  | Yeji   | 22  |
| 4  | Lia    | 22  |
| 5  | Jennie | 101 |
| 6  | Jennie | 101 |
+----+-----+-----+
6 rows in set (0.00 sec)
```

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 4.4

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

```
mysql> select * from 'Lingjing';
+----+-----+-----+
| id | idol  | age |
+----+-----+-----+
| 1  | Yunjin | 100 |
| 2  | Kazuha | 19  |
| 3  | Yeji   | 22  |
| 4  | Lia    | 22  |
| 5  | Jennie | 101 |
| 6  | Jennie | 101 |
+----+-----+-----+
6 rows in set (0.00 sec)

mysql> update Lingjing set `age` = 233 where `idol` = 'Jennie';
Query OK, 2 rows affected (0.00 sec)
Rows matched: 2  Changed: 2  Warnings: 0

mysql>
```

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

```
mysql> update Lingjing set 'age' = 101 where 'idol' = 'Jennie'
Query OK, 2 rows affected (0.00 sec)
Rows matched: 2  Changed: 2  Warnings: 0

mysql> select * from 'Lingjing';
+----+-----+-----+
| id | idol  | age |
+----+-----+-----+
| 1  | Yunjin | 100 |
| 2  | Kazuha | 19  |
| 3  | Yeji  | 22  |
| 4  | Lia   | 22  |
| 5  | Jennie | 101 |
| 6  | Jennie | 101 |
+----+-----+-----+
6 rows in set (0.00 sec)
```

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，可以查到修改后的数据。

```
mysql> select * from 'Lingjing';
+----+-----+-----+
| id | idol  | age |
+----+-----+-----+
| 1  | Yunjin | 100 |
| 2  | Kazuha | 19  |
| 3  | Yeji  | 22  |
| 4  | Lia   | 22  |
| 5  | Jennie | 233 |
| 6  | Jennie | 233 |
+----+-----+-----+
6 rows in set (0.00 sec)
```

图 4.6: commit 后读到新值

MongoDB

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

```
mongos> db.Lingjing.insert({idol: "Hearin", age: 17, group: "NEWJEANS"})
WriteResult({ "nInserted" : 1 })
mongos> db.Lingjing.find()
{ "_id" : ObjectId("64494f999bb0b048347daee1"), "idol" : "Yunjin", "age" : 21, "group" : "LESSERAFIM" }
{ "_id" : ObjectId("64494ffc9bb0b048347daee3"), "idol" : "Kazuha", "age" : 19, "group" : "LESSERAFIM" }
{ "_id" : ObjectId("6449503b9bb0b048347daee4"), "idol" : "Yeji", "age" : 22, "group" : "ITZY" }
{ "_id" : ObjectId("6449506b9bb0b048347daee5"), "idol" : "Lia", "age" : 22, "group" : "ITZY" }
{ "_id" : ObjectId("64495a599bb0b048347daeeb"), "idol" : "Yuna", "age" : 19, "group" : "ITZY" }
{ "_id" : ObjectId("64495a6d9bb0b048347daeecc"), "idol" : "Hyein", "age" : 15, "group" : "NEWJEANS" }
{ "_id" : ObjectId("64495a809bb0b048347daeed"), "idol" : "Minji", "age" : 18, "group" : "NEWJEANS" }
{ "_id" : ObjectId("64495a949bb0b048347daeee"), "idol" : "Hearin", "age" : 17, "group" : "NEWJEANS" }
mongos>
```

图 4.7: 插入结果图

在另一个终端窗口中查询数据

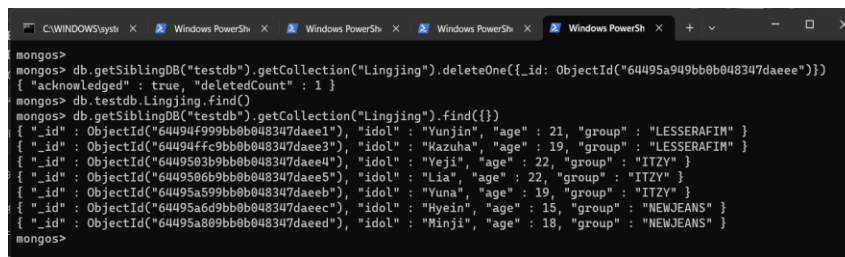
```
C:\WINDOWS> x Windows Powe x Windows Powe x Windows Powe x Windows Powe x
mongos> db.getSiblingDB("testdb").getCollection("Lingjing").find({})
{ "_id" : ObjectId("64494f999bb0b048347daee1"), "idol" : "Yunjin", "age" : 21, "group" : "LESSERAFIM" }
{ "_id" : ObjectId("64494ffc9bb0b048347daee3"), "idol" : "Kazuha", "age" : 19, "group" : "LESSERAFIM" }
{ "_id" : ObjectId("6449503b9bb0b048347daee4"), "idol" : "Yeji", "age" : 22, "group" : "ITZY" }
{ "_id" : ObjectId("6449506b9bb0b048347daee5"), "idol" : "Lia", "age" : 22, "group" : "ITZY" }
{ "_id" : ObjectId("64495a599bb0b048347daeeb"), "idol" : "Yuna", "age" : 19, "group" : "ITZY" }
{ "_id" : ObjectId("64495a6d9bb0b048347daeecc"), "idol" : "Hyein", "age" : 15, "group" : "NEWJEANS" }
{ "_id" : ObjectId("64495a809bb0b048347daeed"), "idol" : "Minji", "age" : 18, "group" : "NEWJEANS" }
{ "_id" : ObjectId("64495a949bb0b048347daeee"), "idol" : "Hearin", "age" : 17, "group" : "NEWJEANS" }
mongos>
```

图 4.8: 在新端口里查询数据

删除一条数据

代码 4.5

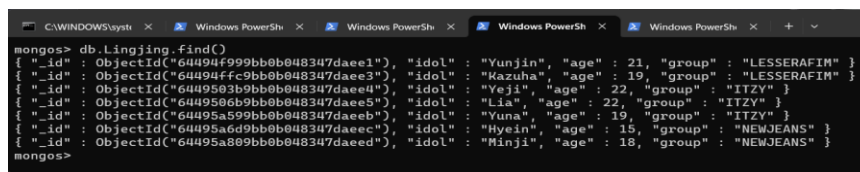
```
db.getSiblingDB("testdb").getCollection("Lingjing")
.deleteOne({ _id: ObjectId("64495a949bb0b048347daee")})
```



```
mongos> db.getSiblingDB("testdb").getCollection("Lingjing").deleteOne({_id: ObjectId("64495a949bb0b048347daee")})
{"acknowledged": true, "deletedCount": 1}
mongos> db.testdb.Lingjing.find()
mongos> db.getSiblingDB("testdb").getCollection("Lingjing").find({})
{ "_id" : ObjectId("64494f999bb0b048347daee1"), "idol" : "Yunjin", "age" : 21, "group" : "LESSERAFIM" }
{ "_id" : ObjectId("64494ffcf9bb0b048347daee3"), "idol" : "Kazuha", "age" : 19, "group" : "LESSERAFIM" }
{ "_id" : ObjectId("6449503b9bb0b048347daee4"), "idol" : "Yeji", "age" : 22, "group" : "ITZY" }
{ "_id" : ObjectId("6449506b9bb0b048347daee5"), "idol" : "Lia", "age" : 22, "group" : "ITZY" }
{ "_id" : ObjectId("64495a599bb0b048347daeeb"), "idol" : "Yuna", "age" : 19, "group" : "ITZY" }
{ "_id" : ObjectId("64495a6d9bb0b048347daee"), "idol" : "Hyein", "age" : 15, "group" : "NEWJEANS" }
{ "_id" : ObjectId("64495a809bb0b048347daee"), "idol" : "Minji", "age" : 18, "group" : "NEWJEANS" }
mongos>
```

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



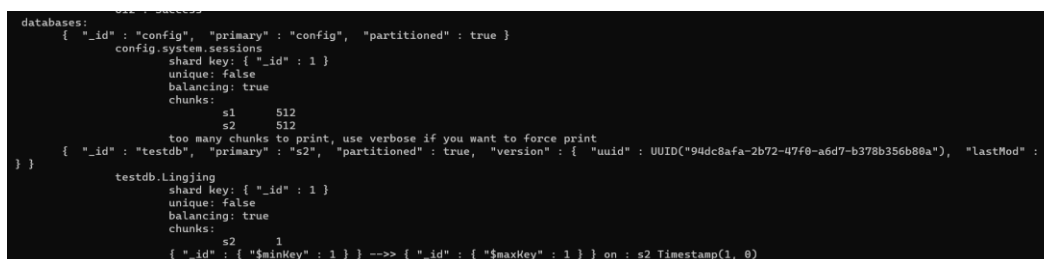
```
mongos> db.Lingjing.find()
{ "_id" : ObjectId("64494f999bb0b048347daee1"), "idol" : "Yunjin", "age" : 21, "group" : "LESSERAFIM" }
{ "_id" : ObjectId("64494ffcf9bb0b048347daee3"), "idol" : "Kazuha", "age" : 19, "group" : "LESSERAFIM" }
{ "_id" : ObjectId("6449503b9bb0b048347daee4"), "idol" : "Yeji", "age" : 22, "group" : "ITZY" }
{ "_id" : ObjectId("6449506b9bb0b048347daee5"), "idol" : "Lia", "age" : 22, "group" : "ITZY" }
{ "_id" : ObjectId("64495a599bb0b048347daeeb"), "idol" : "Yuna", "age" : 19, "group" : "ITZY" }
{ "_id" : ObjectId("64495a6d9bb0b048347daee"), "idol" : "Hyein", "age" : 15, "group" : "NEWJEANS" }
{ "_id" : ObjectId("64495a809bb0b048347daee"), "idol" : "Minji", "age" : 18, "group" : "NEWJEANS" }
mongos>
```

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。



```
databases:
  { "_id" : "config", "primary" : "config", "partitioned" : true }
  config.system.sessions
    shard key: { "_id" : 1 }
    unique: false
    balancing: true
    chunks:
      s1 512
      s2 512
    too many chunks to print, use verbose if you want to force print
  { "_id" : "testdb", "primary" : "s2", "partitioned" : true, "version" : { "uuid" : UUID("94dc8afa-2b72-47f8-a6d7-b378b356b88a"), "lastMod" : 1 } }
    testdb.Lingjing
      shard key: { "_id" : 1 }
      unique: false
      balancing: true
      chunks:
        s2 1
        { "_id" : { "$minKey" : 1 } } --> { "_id" : { "$maxKey" : 1 } } on : s2 Timestamp(1, 0)
```