



# 华中科技大学

## 大数据管理概论实验报告

姓 名： 瞿明睿  
学 院： 计算机科学与技术学院  
专 业： 计算机科学与技术  
班 级： CS2207  
学 号： U202215561

分数	
教师签名	

2025 年 1 月 6 日

## 教师评分页

子目标	子目标评分
1	
2	
3	
总分	

# 目 录

<b>1 课程任务概述 .....</b>	<b>1</b>
<b>2 MySQL for JSON 实验 .....</b>	<b>2</b>
2.1 任务要求.....	2
2.2 完成过程.....	2
2.3 任务小结.....	8
<b>3 MongoDB 实验 .....</b>	<b>9</b>
3.1 任务要求.....	9
3.2 完成过程.....	9 <a href="#">s</a>
3.3 任务小结.....	13
<b>4 Neo4j 实验.....</b>	<b>14</b>
4.1 任务要求.....	14
4.2 完成过程.....	14
4.3 任务小结.....	17
<b>5 课程总结 .....</b>	<b>18</b>

## 1 课程任务概述

本实验主要围绕大数据管理，涉及三种数据库的操作和修改包含 MySQL、MongoDB 和 Neo4j 三种数据库，让学生完成任务，理解不同数据库的特性、功能以及适用场景。

MySQL for JSON 实验关注的是 MySQL 数据库中对于 JSON 处理操作。在查询方面，要求运用多种函数，如 JSON\_EXTRACT、JSON\_UNQUOTE、JSON\_TYPE 等，查询符合特定条件的数据，并进行排序、限制返回数量等操作。JSON 增删包括新增键值对、修改属性值、插入新记录等复杂操作。JSON 聚合通过 JSON\_OBJECTAGG 和 JSON\_ARRAYAGG 等函数实现。此外，实验还包含一些使用 JSON 实用函数进行条件判断、数据转换以及合并 JSON 文档等操作，对比查询执行计划和效率，探讨索引对查询性能的影响。

MongoDB 实验围绕 MongoDB 数据库展开，涵盖条件查询与执行计划分析、聚合操作以及 MapReduce 的运用。在条件查询部分需要使用多种查询操作符按照要求条件查询集合，使用 skip、limit 等控制返回结果。聚合操作要求学生利用 project、\$sort 等聚合管道操作符，实现诸如统计各州商店数量、计算平均打星、按条件分组统计等要求。MapReduce 则要求自定义 map 和 reduce 函数，计算子集集合中每个商店的平均得分，进一步拓展对 MongoDB 数据集合的处理运用方式。

Neo4j 实验针对的是图数据库的操作与应用。在查询主要使用 MATCH 语句结合各种条件进行查询，排序限制返回结果等操作。实验要求使用 PROFILE 查看执行计划，理解查询执行过程及性能优化方向。此外还需进行索引操作，包括创建索引以提升查询性能，反映索引对不同操作的影响，以及通过多关系联合查询和 with 语句实现复杂的关联查询，以及不同操作场景下的性能表现。

## 2 MySQL for JSON 实验

### 2.1 任务要求

在使用 MySQL 数据中，使用 JSON 函数进行复杂查询，如按州、城市、属性等条件筛选商户信息，并进行排序与结果限制，数据的增删改操作，包括新增、修改属性值、插入新记录并修改。利用聚合函数实现按州聚合商户信息及将用户的 tips 聚合成 JSON 数组。通过对比执行计划和效率，理解索引对查询性能的影响。

### 2.2 完成过程

#### 2.2.1 1-a-5 执行计划对比

任务描述：使用 explain 查看 `select * from user where user_info->'$.cool' > 200` 的执行计划,其中执行计划按 JSON 格式输出;并且实际执行一次该查询,请注意观察语句消耗的时间并与 MongoDB 的查询方式进行对比(MongoDB 要执行此查询要求,相应的语句是什么?执行计划是怎样的?并给出查询效率对比).最后，在 MySQL 中为 `user_info` 的字段加索引来优化提高查询效率，对比一下 MySQL 加索引查询前后的查询效率，分析加索引前后的执行计划。

分析：查询的代码语句较为简单，我们直接来进行比较

执行以下代码可获得以下结果

```
EXPLAIN FORMAT = JSON
SELECT * FROM user
WHERE JSON_EXTRACT(user_info, '$.cool') > 200;
```

```
{
  "query_block": {
    "select_id": 1,
    "cost_info": {
      "query_cost": "501899.80"
    },
    "table": {
      "table_name": "user",
      "access_type": "ALL",
      "rows_examined_per_scan": 1842688,
      "rows_produced_per_join": 1842688,
      "filtered": "100.00",
      "cost_info": {
        "read_cost": "317631.00",
        "eval_cost": "184268.80",
        "prefix_cost": "501899.80",
```

```

        "data_read_per_join": "210M"
      },
      "used_columns": [
        "user_id",
        "user_info"
      ],
      "attached_condition": "(json_extract(`test`.`user`.`user_info`, '$.cool') >
200)"
    }
  }
} |
26981 rows in set (34.20 sec)

```

表 2.1 执行计划查询

这里分析一下这里查询的执行计划，可以看到以下字段比较重要：

**access\_type=ALL:** 表明此查询进行了全表扫描，没有使用索引，因为我们并没有建立索引，所以需要遍历整个 **user** 表来查找满足条件的记录。

**rows\_examined\_per\_scan=1842688:** 表明在每次扫描表时需要检查的行数即整个 **user** 表的行数。因为没有索引可以快速定位满足条件的记录，所以需要逐一检查每一行。

**rows\_produced\_per\_join=1842688:** 表示连接操作产生的行数为 1503145 行，由于是全表扫描，所以所有行都被视为满足连接条件

**filter=100.00:** 在应用条件后，没有对行进行额外的过滤，所有扫描的行都需要进行 **json\_extract** 函数计算来判断是否满足 **cool > 200** 的条件。

**query\_cost=501899.80:** 预估的查询成本，包括读取数据的成本和评估每行数据是否满足条件的成本。这么高的成本表明没有索引的情况下效率低。

那么我们对比一下 **mongo** 的查询，执行以下语句可以获得这样的结果

```

mongodb:
db.user.find({ "cool": { $gt: 200 } });
db.user.find({ "cool": { $gt: 200 } }).explain("executionStats");
{
  "queryPlanner" : {
    "plannerVersion" : 1,
    "namespace" : "yelp.user",
    "indexFilterSet" : false,
    "parsedQuery" : {
      "cool" : {
        "$gt" : 200
      }
    }
  }
}

```

```

    },
    "winningPlan" : {
      "stage" : "COLLSCAN",
      "filter" : {
        "cool" : {
          "$gt" : 200
        }
      },
      "direction" : "forward"
    },
    "rejectedPlans" : [ ]
  },
  "executionStats" : {
    "executionSuccess" : true,
    "nReturned" : 22923,
    "executionTimeMillis" : 16430,
    "totalKeysExamined" : 0,
    "totalDocsExamined" : 1637141,
    "executionStages" : {
      "stage" : "COLLSCAN",
      "filter" : {
        "cool" : {
          "$gt" : 200
        }
      },
      "nReturned" : 22923,
      "executionTimeMillisEstimate" : 12413,
      "works" : 1637143,
      "advanced" : 22923,
      "needTime" : 1614219,
      "needYield" : 0,
      "saveState" : 1874,
      "restoreState" : 1874,
      "isEOF" : 1,
      "direction" : "forward",
      "docsExamined" : 1637141
    }
  },
  "serverInfo" : {
    "host" : "sanstoolow",

```

```

    "port" : 27017,
    "version" : "4.4.29",
    "gitVersion" : "f4dda329a99811c707eb06d05ad023599f9be263"
  },
  "ok" : 1
}

```

表 2.2 mongo 计划查询

这里再给出 mongo 的执行计划中的关键字段的分析如下

stage: 执行计划的阶段，这里是 COLLSCAN，表示集合扫描，即 MongoDB 扫描了整个集合来查找符合条件的文档。

nReturned: 返回的文档数量，这里是 22923，表示有 22923 个文档符合查询条件。

executionTimeMillis: 查询执行的总时间，单位是毫秒，这里是 16430 毫秒

totalKeysExamined: 扫描的索引键数量，这里是 0，表示没有使用索引。

totalDocsExamined: 扫描的文档数量，这里是 1637141，表示 MongoDB 扫描了 1637141 个文档来找到符合条件的 22923 个文档。

executionTimeMillisEstimate: 执行时间的估计值，这里是 12413 毫秒

advanced: 传递给父阶段的文档数量，这里是 22923。

docsExamined: 扫描的文档数量，这里是 1637141。

最后我们在 MySQL 中加入 user\_info 字段索引，此时我们再次看一下执行计划和查询的结果如下

```

ALTER TABLE user
ADD COLUMN cool INT GENERATED ALWAYS AS
(CAST(JSON_UNQUOTE(JSON_EXTRACT(user_info, '$.cool')) AS
UNSIGNED)) VIRTUAL;
CREATE INDEX idx_cool ON user (cool);
EXPLAIN FORMAT = JSON
SELECT * FROM user WHERE cool > 200;
{
  "query_block": {
    "select_id": 1,
    "cost_info": {
      "query_cost": "66483.79"
    },
    "table": {
      "table_name": "user",
      "access_type": "range",
      "possible_keys": [

```



```

        "idx_cool"
    ],
    "key": "idx_cool",
    "used_key_parts": [
        "cool"
    ],
    "key_length": "5",
    "rows_examined_per_scan": 55674,
    "rows_produced_per_join": 55674,
    "filtered": "100.00",
    "cost_info": {
        "read_cost": "60916.39",
        "eval_cost": "5567.40",
        "prefix_cost": "66483.79",
        "data_read_per_join": "6M"
    },
    "used_columns": [
        "user_id",
        "user_info",
        "cool"
    ],
    "attached_condition": "(`test`.`user`.`cool` > 200)"
    }
}
} |
26981 rows in set (17.78 sec)

```

表 2.3 mysql 计划查询

这里再看一下关键字段

**access\_type:** 访问类型，这里是 **range**，表示查询使用了索引范围扫描

**possible\_keys:** 可能使用的索引列表，这里是 ["idx\_cool"]，表示 MySQL 考虑使用 idx\_cool 索引。

通过对比建立索引前后，数据库对于前后的 **cost** 的预测都小了很多，实际上的查询时间也少了很多。这里可以提及：使用虚拟列并加上了索引，无论是需要扫描的数目还是使用的时间都减少了很多

### 2.2.2 1-c-13 JSON 和关系型表格转换

**任务要求：** 查询被评论数前 3 的商户,使用 **JSON\_TABLE()** 可以将 json 型数据转换为关系型表格，请使用 **JSON\_TABLE()** 将商户的 **name**, **HasTV**, 和所有的 **attributes** (不考虑顺序，一个属性就对应一行，对每个商户，从 1 开始对这些时段

递增编号), 最后按商户名字升序排序.

分析: 这个问题可以用嵌套查询解决

首先需要查询 business 表中按 review\_count 降序排序前三条记录, 拿出被评论数前 3 的商户记录, 然后这里需要将 business\_info 字段的 attributes 字段展开成表格形式。也就是使用 JSON\_TABLE 函数, 每个属性变成单独的行, 通过 attribute\_value 字段给予属性值。最后更要求我们得分为 business\_name 和 HasTV 属性, 计算 attribute\_value, 最后使用 business\_name 升序排序

```
SELECT
    JSON_UNQUOTE(JSON_EXTRACT(business_info, '$.name')) AS
business_name,
    JSON_UNQUOTE(JSON_EXTRACT(business_info, '$.attributes.HasTV'))
AS HasTV,
    ROW_NUMBER() OVER (PARTITION BY
JSON_UNQUOTE(JSON_EXTRACT(business_info, '$.name')) ORDER BY
attr.attribute_value) AS num, attr.attribute_value
FROM (
    SELECT
        business_info
    FROM business
    ORDER BY JSON_EXTRACT(business_info, '$.review_count') DESC
    LIMIT 3
) AS t
JOIN JSON_TABLE(
    t.business_info,
    '$.attributes.*' COLUMNS (
        attribute_value VARCHAR(255) PATH '$'
    )
) AS attr ON true
ORDER BY business_name ASC;
```

表 2.4 JSON 和关系型表格转换

这里简单分析一下数据流向, 在这里中间的部分为子查询, 外层的查询的得到 business\_name, HasTV, 并且使用对两者的独立编号, 在形成表格的时候 attribute\_value VARCHAR(255) PATH '\$' 中的 VARCHAR(255) 指明数据类型, \$ 指名数据就是 \$.attributes.

这里简单看一下结果, 如下图 2.1

```

| Acme Oyster House | True | 24 | u'no'
| Acme Oyster House | True | 23 | u'loud'
| Acme Oyster House | True | 22 | True
| Acme Oyster House | True | 21 | True
| Acme Oyster House | True | 20 | True
| Acme Oyster House | True | 19 | True
| Acme Oyster House | True | 18 | True
| Acme Oyster House | True | 17 | True
| Acme Oyster House | True | 16 | True
| Acme Oyster House | True | 15 | True
| Acme Oyster House | True | 14 | None
| Acme Oyster House | True | 12 | False
| Acme Oyster House | True | 13 | None
| Acme Oyster House | True | 3 | 'yes_free'
| Acme Oyster House | True | 4 | {'dessert': False, 'latenight':
: None, 'lunch': True, 'dinner': True, 'brunch': None, 'breakfast': False}
| Acme Oyster House | True | 5 | {'touristy': False, 'hipster':
False, 'romantic': False, 'divey': False, 'intimate': False, 'trendy': False, 'upscale': False, 'classy': True, 'casual': True}

```

图 2.1 结果缩略图

## 2.3 任务小结

Mysql 对于 json 数据的支持还是比较重要的，其中很多操作相比传统的，之前有所了解的数据库查询还是很不一样的，有结构的数据还是会更有操作性，内容上更丰富

## 3 MongoDB 实验

### 3.1 任务要求

运用多种查询操作符进行条件查询，结合 skip、limit 控制结果返回。

利用聚合管道操作符实现统计、计算平均值、分组统计等聚合操作。

运用 MapReduce 计算子集中商店的平均得分

### 3.2 完成过程

#### 3.2.1 2-a-9 查询优化

任务：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。

分析：我们先查询执行计划，建立索引后在查询一次执行计划比较即可

```
db.business.find({business_id:"5JucpCfHZltJh5r1JabjDg" }).explain("executionStats")
```

```
{
  "queryPlanner" : {
    "plannerVersion" : 1,
    "namespace" : "yelp.business",
    "indexFilterSet" : false,
    "parsedQuery" : {
      "business_id" : {
        "$eq" : "5JucpCfHZltJh5r1JabjDg"
      }
    },
  },
  "executionStats" : {
    "nReturned" : 1,
    "executionTimeMillis" : 567,
    "totalDocsExamined" : 192609,
    "executionStages" : {
      "stage" : "COLLSCAN",
      "filter" : {
        "business_id" : {
          "$eq" : "5JucpCfHZltJh5r1JabjDg"
        }
      },
      "nReturned" : 1,
      "executionTimeMillisEstimate" : 125,
```

<pre>         "works" : 192611,         "advanced" : 1,         "needTime" : 192609,         "docsExamined" : 192609       }     },   } } </pre>
<pre> db.business.createIndex({ business_id: 1 }) db.business.find({business_id:"5JucpCfHZltJh5r1JabjDg" }).explain("executionSta ts") </pre>
<pre> {   "executionStats" : {     "executionSuccess" : true,     "nReturned" : 1,     "executionTimeMillis" : 2,     "totalKeysExamined" : 1,     "totalDocsExamined" : 1,     "executionStages" : {       "stage" : "FETCH",       "nReturned" : 1,       "executionTimeMillisEstimate" : 0,       "inputStage" : {         "stage" : "IXSCAN",         "nReturned" : 1,         "executionTimeMillisEstimate" : 0,         "works" : 2,         "advanced" : 1,         "isEOF" : 1,         "keyPattern" : {           "business_id" : 1         },         "indexName" : "business_id_1",         "isMultiKey" : false,         "multiKeyPaths" : {           "business_id" : [ ]         },         "indexVersion" : 2,         "direction" : "forward",         "indexBounds" : {           "business_id" : [ </pre>

```

        ["5JucpCfHZltJh5r1JabjDg",
        "5JucpCfHZltJh5r1JabjDg"]
    ],
    },
    "keysExamined" : 1,
    "seeks" : 1,
  }
}
},
}

```

表 3.1 执行结果（精简）

在未创建索引的情况下，执行计划显示为 COLLSCAN 全集合扫描。这意味着需要遍历整个 business 集合中的所有文档，逐一检查 business\_id 字段是否与指定值匹配。全集合扫描读取大量不必要的数据，导致查询执行时间较长。

db.business.createIndex({business\_id: 1})创建 business\_id 字段的升序索引。用过查看执行计划，会发现查询使用了 IndexScan 即索引，直接通过索引定位到 business\_id 为"5JucpCfHZltJh5r1JabjDg"的文档。同时扫描文档数 nReturned 显著减少，从全集合扫描时的大量文档减少到仅 1 个。

### 3.2.2 2-b-13 地图索引

**任务：**在 business 表中，查询距离商家 smkZUv\_IeYYj\_BA6-Po7oQ(business\_id) 2 公里以内的所有商家，返回商家名字，地址和星级，按照星级降序排序，限制返回 20 条。

**分析：**建立一个关于位置的索引寻找

```

db.business.createIndex({ loc: "2dsphere" });
let targetBusiness;

targetBusiness = db.business.findOne({ business_id: "smkZUv_IeYYj_BA6-Po7oQ" });
if (!targetBusiness) {
  print("Target business not found");
} else {

  let targetCoordinates = targetBusiness.loc.coordinate

  db.business.find({
    loc: {
      $nearSphere: {
        $geometry: {

```

```

        type: "Point",
        coordinates: targetCoordinates
    },
    $maxDistance: 2000
}
}
}, {
    _id: 0,
    name: 1,
    address: 1,
    stars: 1
}).sort({ stars: -1 }).limit(20).forEach(printjson);
}

```

表 3.2 空间索引建立

使用 `db.business.createIndex({loc: "2dsphere"})` 创建 `loc` 字段的 2dsphere 索引。获取商家坐标，执行地理空间查询。

`db.business.find({loc: {$nearSphere: {$geometry: {type: "Point", coordinates: targetCoordinates}, $maxDistance: 2000}}}, {_id: 0, name: 1, address: 1, stars: 1}).sort({stars: -1}).limit(20).forEach(printjson)` 进行地理空间查询。`$nearSphere` 操作符用于查找距离指定点距离的文档，`$geometry` 指定查询的坐标点类型为 `Point` 并传入目标商家坐标，`$maxDistance` 设定最大距离为 2000 米。

这里强调一下 2dsphere 索引，它是索引，所以可以快速定位到与给定地理坐标点在一定距离范围内的文档，避免全表扫描，同时还有没用到的功能，比如：判断点是否在多边形内、计算两个地理区域的交集。

### 3.2.3 2-c-15 MapReduce 使用

**任务描述：**使用 map reduce 计算 Subreview 集合中每个商店的平均得分, (不要直接使用聚合函数), 输出为一个集合 Map\_Reduce, 其中应该包括 `business_id` 以及 `values{count(打分次数), sum_stars(总的打分), avg_stars(平均打分)}`, 最后查询 Map\_Reduce, 返回前 20 条数据。

**分析：**

```

//Map 函数
var mapfunc = function() {
    emit(this.business_id, {
        count: 1,
        sum_stars: this.stars
    });
}

```

```

});
//Reduce 函数
var reducefunc = function(key, values) {
    var result = { count: 0, sum_stars: 0 };
    values.forEach(function(value) {
        result.count += value.count;
        result.sum_stars += value.sum_stars;
    });
    result.avg_stars = result.sum_stars / result.count;
    return result;
};
//MapReduce 应用两个函数
db.Subreview.mapReduce(
    mapfunc,
    reducefunc,
    {out: "Map_Reduce"}
);
//查询
db.Map_Reduce.find().limit(20).pretty();

```

表 3.3 MapReduce

**Map:** emit 逻辑是传入值返回一个键值对并输出，这里就是 Subreview 集合中的每个文档转换为一个键值对 {business\_id, (sum\_stars 和 count)} 比如这里假设有三个对 A, B 和 C 评价的记录，经过 MAP 之后形成(A, {count: 1, sum\_stars: stars })、(B, {count: 1, sum\_stars: stars })、(C, {count: 1, sum\_stars: stars })。

**Reduce:** 接收相同 business\_id 的多个值对象。通过累加键值对第二个对象中的 count 和 stars 值最后取平均得到平均打分 avg\_stars，具体评论数目 count 最后都是一个键值对形式保存下来了。最终返回结果对象将作为聚合结果，包含了每个商店的打分次数、总打分和平均打分信息。

执行 MapReduce 操作，将结果输出到名为 Map\_Reduce 的集合中。最后使用 db.Map\_Reduce.find().limit(20).pretty();查询 Map\_Reduce 集合

### 3.3 任务小结

Mongo 的逻辑和 mysql 完全不一样，这种类似于 js 的语法让人眼前一亮，但是这也让查询可以更加复杂，能做的事情更多了。就比如最后 MapReduce 函数的实现，就是对于 mongo 的灵活性的体现。



## 4 Neo4j 实验

### 4.1 任务要求

运用 MATCH 语句结合条件进行查询，掌握排序与结果限制操作。

通过 PROFILE 查看执行计划，优化查询性能。

进行索引操作，观察其对各种操作的影响。

利用多关系联合查询和 with 语句实现复杂关联查询。

### 4.2 完成过程

#### 4.2.1 3-10 查询关系边

任务描述：查询 userid 为 d7D4dYzF6THtOx9imf-wPw 的用户的朋友（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)，返回前 20 条数据。

分析：查询节点 id == d7D4dYzF6THtOx9imf-wPw，cha 训导这样的用户节点之后，我们再次分析一下这样的用户的朋友的朋友有多少个。

```
MATCH (u:UserNode {userid: 'd7D4dYzF6THtOx9imf-wPw'})-[:HasFriend]->
(f:UserNode)
WITH f.name AS fname, size((f)-[:HasFriend]->()) AS fofCount
RETURN fname, fofCount
LIMIT 20
//Started streaming 20 records after 1 ms and completed after 36 ms.

MATCH (u:UserNode {userid: 'd7D4dYzF6THtOx9imf-wPw'})-[:HasFriend]->
(f:UserNode)
WITH f, f.name AS friend_name
MATCH (f)-[:HasFriend]->(fof:UserNode)//不用 size 方法，就得再做一次 match
WITH f.userid AS uid , friend_name, COUNT(fof) AS numberOfFs
RETURN friend_name, numberOfFs
LIMIT 20
//Started streaming 20 records after 1 ms and completed after 4042 ms.
```

表 4.1 查询两种思路

这里描述两个问题：

1.思路问题：这两段代码都可以查询到结果，且肯定是一样的，但是这两段代码使用了不一样的思路，第一段代码使用 size()函数，统计列表中元素的数量。第二段代码则是通过用户节点得到第一组朋友节点后，再次查询朋友节点并统计。

2.效率问题：显然通过在代码尾巴的注释可以了解到，这里的执行结果第一个思路速度大大快于第二个思路，原因在于 size()的执行效果等于在执行完成后，

统计了朋友节点的满足(f)-[:HasFriend]->()统计模板的边的数量,这样就不用就列表朋友节点的朋友节点具体有多少个,从而大大提升了查询效率,从第二个查询结果来看,就是慢了很多,相当于再次做了 n 次 (n 应该是第一次查询得到的节点数目) 第一遍查询,效率很低。

#### 4.2.2 3-17 多关系查询和查询优化

任务描述: 查询与用户 user1 (userid: 4i4lyXBigT2HShIjw7TbDw) 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。

分析:

```
MATCH (user1:UserNode {userid: '4i4lyXBigT2HShIjw7TbDw'})-[:Review]->
(r1:ReviewNode)-[:Reviewed]->(b:BusinessNode)
MATCH (user2:UserNode)-[:Review]->(r2:ReviewNode)-[:Reviewed]->(b)
WHERE NOT (user1)-[:HasFriend]->(user2) AND user1 <> user2 //AI:: 条件确
保找到的`user2`与`user1`不是朋友关系且不是`user1`本身
WITH user1.name AS user1_name, user2.name AS user2_name,
COUNT(DISTINCT b) AS sum
ORDER BY sum DESC
LIMIT 10
RETURN user1_name, user2_name, sum
//Started streaming 10 records after 2 ms and completed after 18101 ms.
//索引
CREATE INDEX FOR (u:UserNode) ON (u.userid);
//Added 1 index, completed after 41 ms.

PROFILE
MATCH (user1:UserNode {userid: '4i4lyXBigT2HShIjw7TbDw'})-[:Review]->
(r1:ReviewNode)-[:Reviewed]->(b:BusinessNode)
MATCH (user2:UserNode)-[:Review]->(r2:ReviewNode)-[:Reviewed]->(b)
WHERE NOT (user1)-[:HasFriend]->(user2) AND user1 <> user2
WITH user1.name AS user1_name, user2.name AS user2_name,
COUNT(DISTINCT b) AS sum
ORDER BY sum DESC
LIMIT 10
RETURN user1_name, user2_name, sum
//Cypher version: CYPHER 4.0, planner: COST, runtime: INTERPRETED. 52813
total db hits in 405 ms.
```

表 4.2 查询计划以及优化

这里先找到 user1 评价的商家 MATCH (user1:UserNode {userid:

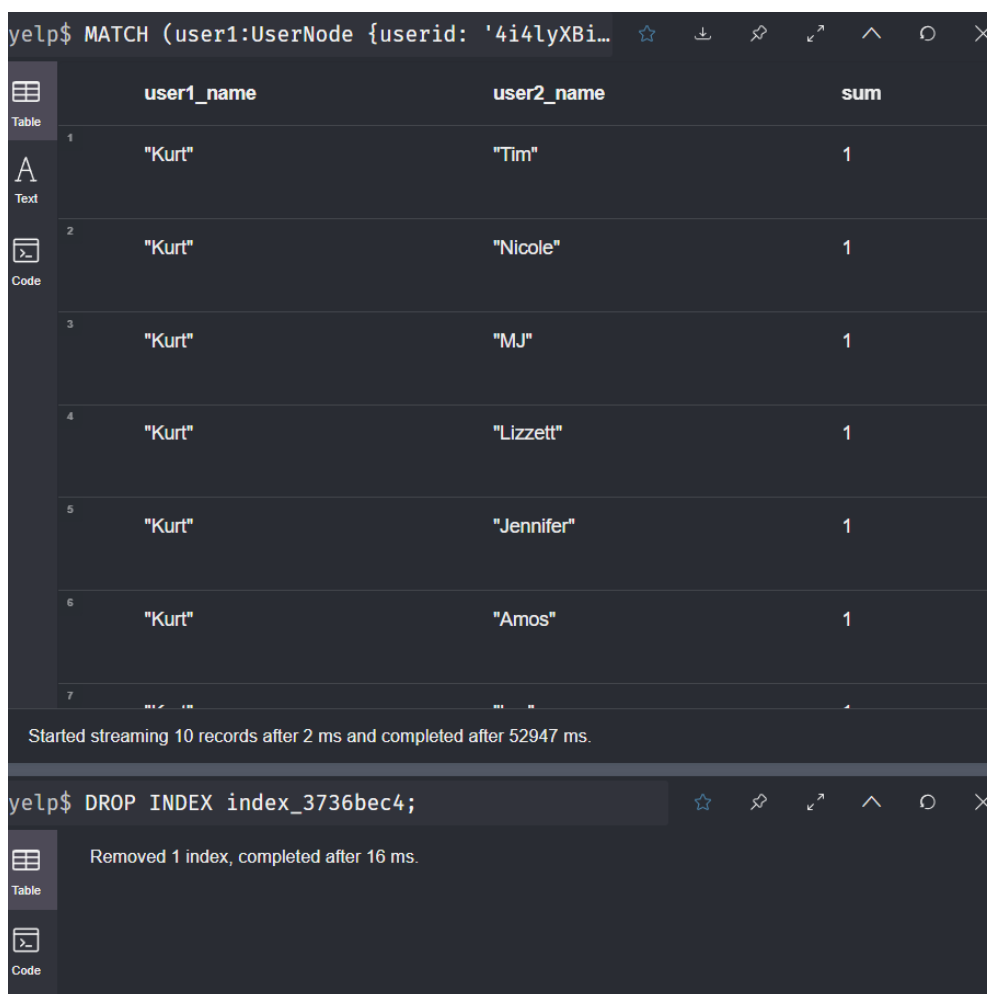
'4i4lyXBigT2HShIjw7TbDw'}})-[:Review]->(r1:ReviewNode)-[:Reviewed]->(b:BusinessNode)通过 Review 关系找到 user1 评价的 ReviewNode 节点，再通过 Reviewed 关系找到对应的 BusinessNode 节点 b，确定了 user1 评价过的商家集合。

然后我们再找到与 user1 不是朋友关系且评价过相同商家的用户

MATCH (user2:UserNode)-[:Review]->(r2:ReviewNode)-[:Reviewed]->(b)找到所有评价过商家 b 的 user2，然后通过 WHERE NOT (user1)-[:HasFriend]->(user2) AND user1 <> user2 条件筛选出与 user1 不是朋友关系且不是 user1 本身的用户。

你们这里我们可以计算共同评价的商家数量并返回结果 WITH user1.name AS user1\_name, user2.name AS user2\_name, COUNT(DISTINCT b) AS sum 使用 WITH 子句将 user1 和 user2 的名字以及共同评价的商家数量进行分组统计，COUNT(DISTINCT b)确保计算的是不同商家的数量。最后通过 ORDER BY sum DESC 按照共同评价的商家数量降序排序，返回用户名和共同评价的商家数量

在这里我们第一次查询的结果非常不好，如下图 4.1



	user1_name	user2_name	sum
1	"Kurt"	"Tim"	1
2	"Kurt"	"Nicole"	1
3	"Kurt"	"MJ"	1
4	"Kurt"	"Lizzett"	1
5	"Kurt"	"Jennifer"	1
6	"Kurt"	"Amos"	1

Started streaming 10 records after 2 ms and completed after 52947 ms.

yelp\$ DROP INDEX index\_3736bec4;

Removed 1 index, completed after 16 ms.

图 4.1 无索引的查询结果

优化过结果查询图如下

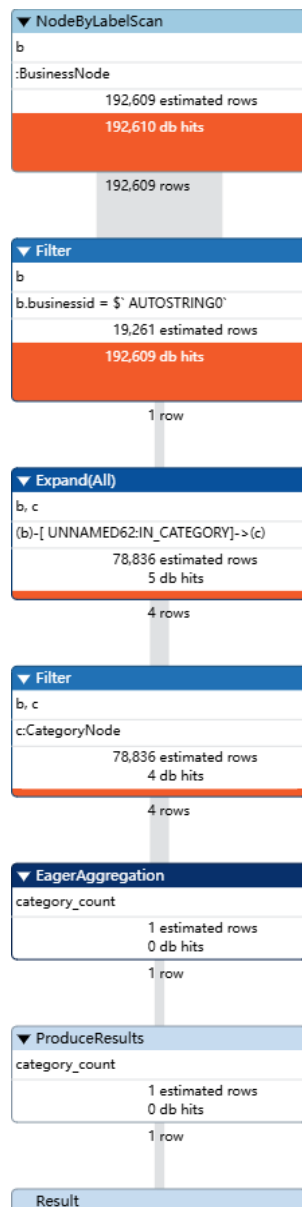


图 4.2 优化后查询计划

通过在表中的查询时间可知，通过建立索引，查询速度快非常多。

### 4.3 任务小结

我们通过使用 `neo4j` 这样的图数据库，又感受到了一种别的查询方式的新奇感，通过关系查找结点，从语法上更加符合人的思路，也更容易编写代码，但是往往这样的代码很容易出现，效率的问题，如 3-10 查询边的过程，多写一次 `MATCH` 会增加非常多的开销。

## 5 课程总结

本次大数据管理课程实践涵盖了 MySQL、MongoDB 和 Neo4j 三种数据库的操作与应用。完成各项任务，需要掌握了不同数据库的基本操作，理解了它们的特性、适用场景及性能优化方法。比如增加针对特定数据的索引，增加虚拟列并且在虚拟列上再次索引等等方法。

了解了 MySQL、MongoDB 和 Neo4j 这样三个数据库的操作与应用在实践过程中，遇到函数使用不熟练、查询逻辑错误、性能优化困难等问题，但通过查阅文档、分析错误信息和不断调试，都得到了有效解决。其中询问人工智能的过程中有遇到各种各样的问题，所以查询文档的方法还是很有效果的，不容易发生错误，特定语法在数据库的特定版本可能不再使用，但是人工智能有时会给出这样错误的答案导致调试苦难，所以更需要自己的思考的理解。

总的来说还是比较有收获的，其次在服务器上还可以通过使用 next.js 装 mongo 模块实现网页服务器开设，在 js 内实现对于数据库的操作，还是非常具有使用价值的一次体验。