

**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 |  |

20 年 月 日

**教师评分页**

|  |  |
| --- | --- |
| 子目标 | 子目标评分 |
| 1 |  |
| 2 |  |
| 3 |  |
|  |  |
|  |  |
|  |  |
|  |  |
| 总分 |  |

**目 录**

[1 课程任务概述 1](#_Toc187069996)

[2 MySQL for JSON 实验 2](#_Toc187069997)

[2.1 任务要求 2](#_Toc187069998)

[2.2 完成过程 2](#_Toc187069999)

[2.3任务小结 5](#_Toc187070000)

[3 MongoDB 实验 6](#_Toc187070001)

[3.1 任务要求 6](#_Toc187070002)

[3.2 完成过程 6](#_Toc187070003)

[3.3任务小结 8](#_Toc187070004)

[4 Neo4j 实验 10](#_Toc187070005)

[4.1 任务要求 10](#_Toc187070006)

[4.2 完成过程 10](#_Toc187070007)

[4.3任务小结 11](#_Toc187070008)

[5 课程总结 13](#_Toc187070009)

# 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，计算子集合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.

分析如下

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) |

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

filtere=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"); |
|  |

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

|  |
| --- |
|  |
|  |
|  |

但一般来说，如果 MongoDB 在cool字段上没有索引，其查询效率可能与 MySQL 的全表扫描类似，甚至可能更低，因为 MongoDB 在处理复杂查询条件时可能需要更多的内部操作。而如果 MongoDB 在cool字段上建立了索引，其查询效率将显著高于 MySQL 的原始查询，因为索引可以快速定位满足条件的文档，减少不必要的文档扫描。

原始的 MySQL 查询由于缺乏合适的索引，执行计划显示全表扫描，导致查询效率低下，耗时较长。MongoDB 在没有索引的情况下查询效率可能类似，但在建立合适索引后会有显著提升。通过为 MySQL 的user\_info字段添加针对cool条件的索引，查询执行计划得到优化，从全表扫描变为使用索引快速定位记录，查询效率大幅提高。这充分体现了索引在优化数据库查询性能方面的重要性，在实际数据库应用中，合理创建索引是提高查询性能的关键手段之一。同时，也需要注意索引的维护成本，避免过度创建索引导致数据插入、更新和删除操作的性能下降。

### 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

1. 内层查询执行：首先，`business`表中的数据按`review\_count`降序排序，并返回前三条记录的`business\_info`字段。返回结果是一个包含`business\_info`字段的结果集。

2. `JSON\_TABLE`展开：然后，`JSON\_TABLE`函数处理这些`business\_info`字段，将其中的`attributes`字段展开成表格形式。每个属性变成单独的行，并且`attribute\_value`字段提取属性值。

3. 外层查询：接着，外层查询处理这些展开的数据：

- 提取`business\_name`和`HasTV`属性。

- 计算每个业务的属性值行号（`ROW\_NUMBER()`），按`attribute\_value`排序。

- 返回每条记录的`attribute\_value`。

4. 排序：最后，结果按`business\_name`进行升序排序。

|  |
| --- |
| 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; |

这里简单看一下结果

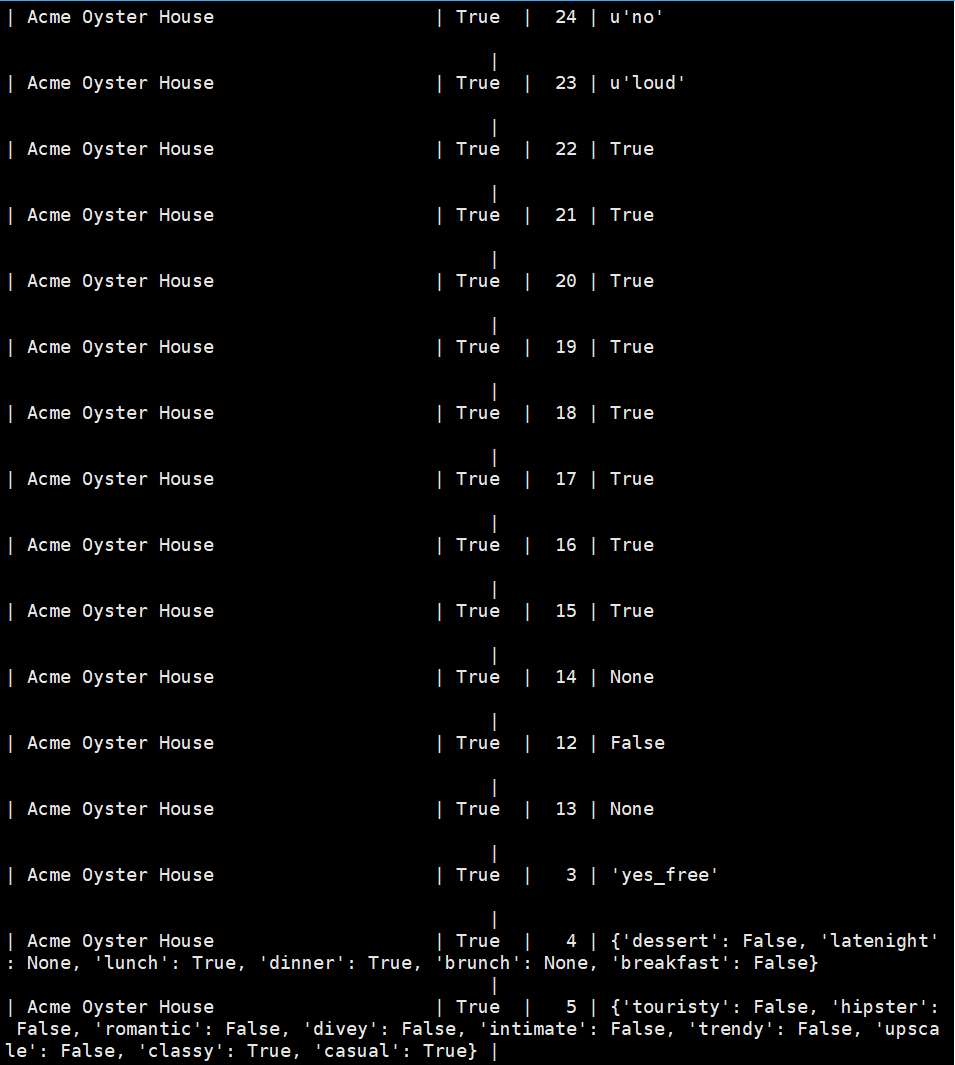


图2.1 结果缩略图

该查询通过巧妙地结合子查询、JSON\_TABLE 函数和窗口函数，实现了对 JSON 数据的复杂处理和转换，将其以关系型表格的形式呈现，并按照商户名字升序排序。这不仅展示了如何在关系型数据库中处理 JSON 数据的能力，也体现了在处理复杂数据结构时如何通过合理的查询设计来满足特定的数据分析需求，同时要注意数据质量和性能方面的问题，以确保查询的正确性和高效性

内层查询（获取被评论数前 3 的商户信息）

使用ORDER BY JSON\_EXTRACT(business\_info, '$.review\_count') DESC按照商户的评论数降序排序，然后通过LIMIT 3获取评论数最多的前 3 条记录的business\_info字段。这一步确定了要处理的目标商户信息，是整个查询的基础，确保后续操作只针对被评论数最高的商户。

JSON\_TABLE 展开（处理商户属性）

使用JSON\_TABLE(t.business\_info, '$.attributes.\*' COLUMNS (attribute\_value VARCHAR(255) PATH '$')) AS attr将内层查询返回的每条business\_info中的attributes字段展开为关系型表格形式。$.attributes.\*表示展开attributes对象中的所有属性，每个属性变为单独的一行，COLUMNS (attribute\_value VARCHAR(255) PATH '$')定义了展开后的表格包含一个名为attribute\_value的列，其数据类型为VARCHAR(255)，路径为$表示获取属性的值。这一步将复杂的 JSON 格式的属性数据转换为关系型表格结构，方便后续处理和展示。

外层查询（整合并返回结果）

选择商户的name（使用JSON\_UNQUOTE(JSON\_EXTRACT(business\_info, '$.name')) AS business\_name提取并去除引号）、HasTV属性（使用JSON\_UNQUOTE(JSON\_EXTRACT(business\_info, '$.attributes.HasTV')) AS HasTV提取并去除引号）以及展开后的attributes字段的值（attribute\_value）。

使用ROW\_NUMBER() OVER (PARTITION BY JSON\_UNQUOTE(JSON\_EXTRACT(business\_info, '$.name')) ORDER BY attr.attribute\_value) AS num为每个商户的属性行分配编号。PARTITION BY JSON\_UNQUOTE(JSON\_EXTRACT(business\_info, '$.name'))表示按照商户name进行分区，确保每个商户的属性行编号独立，ORDER BY attr.attribute\_value表示在每个分区内按照attribute\_value升序排序，为属性行分配递增的编号。

最后按照business\_name升序排序结果，使得输出按照商户名字有序排列，每个商户的属性行按照编号顺序依次列出。

## 2.3 任务小结

# 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计算子集合中商店的平均得分

## 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

|  |
| --- |
| db.business.find({business\_id:"5JucpCfHZltJh5r1JabjDg" }).explain("executionStats")  db.business.createIndex({ business\_id: 1 })  db.business.find({business\_id:"5JucpCfHZltJh5r1JabjDg" }).explain("executionStats")  //来自AI:  //查找  db.business.getIndexes()  //删  db.business.dropIndex("indexName") |

//在未创建索引时，会进行全集合扫描，有了索引，查询应该使用索引扫描，会少很多文档。

未优化执行计划详情

当执行db.business.find({business\_id: "5JucpCfHZltJh5r1JabjDg"}).explain("executionStats")时，在未创建索引的情况下，执行计划可能显示为全集合扫描（CollectionScan）。这意味着 MongoDB 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。

全集合扫描的性能较低，因为它需要读取大量不必要的数据，消耗较多的系统资源（如 CPU、内存和磁盘 I/O），导致查询执行时间较长。例如，假设business集合中有大量的文档，每次查询都要对所有文档进行检查，即使目标文档可能在集合的开头或中间位置，也需要遍历整个集合才能找到，这显然效率低下。

创建索引语句及作用

执行db.business.createIndex({business\_id: 1})创建business\_id字段的升序索引。索引的作用是为 MongoDB 提供一种快速定位数据的方式，类似于书籍的目录。通过创建索引，MongoDB 可以直接根据business\_id的值快速定位到对应的文档，而无需进行全集合扫描。

索引的数据结构通常是 B 树或 B + 树（在 MongoDB 中），这种结构可以高效地进行数据查找、插入和删除操作。对于business\_id索引，它会按照business\_id的值进行排序存储，使得查询时能够快速定位到目标值所在的位置，大大提高查询性能。

优化后执行计划变化

创建索引后再次执行查询并查看执行计划，会发现查询使用了索引（IndexScan）。此时，MongoDB 直接通过索引定位到business\_id为"5JucpCfHZltJh5r1JabjDg"的文档，大大减少了需要读取的数据量。

执行计划中的扫描文档数（nReturned）将显著减少，可能从全集合扫描时的大量文档（取决于集合大小）减少到仅 1 个（如果business\_id唯一）。同时，查询执行时间也会大幅缩短，可能从原来的较长时间（如几百毫秒或几秒，取决于集合大小和服务器性能）缩短到几毫秒甚至更短。

### 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：

|  |
| --- |
| //这里AI误导我好久，我逃他猴子，loc字段自己建立索引，我以为本身就有，这样就可以用地理空间查询  db.business.createIndex({ loc: "2dsphere" });  let targetBusiness;//还能有变量真想不到，真无敌了孩子  //寻找目标商家  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  },  $maxDistance: 2000 //这里指定距离  }  }  }, {  \_id: 0,  name: 1,  address: 1,  stars: 1  }).sort({ stars: -1 }).limit(20).forEach(printjson);  }  //AI:`$nearSphere`操作符查找距离目标商家坐标（`targetCoordinates`）2 公里以内的商家文档 |

整体查询思路分析

建立地理空间索引

首先使用db.business.createIndex({loc: "2dsphere"})创建loc字段的2dsphere索引。这是进行地理空间查询的关键步骤，2dsphere索引能够有效地支持对地理坐标数据的查询操作，使得后续查询距离范围内的商家时可以快速定位到相关数据，大大提高查询效率。

获取目标商家坐标

let targetBusiness = db.business.findOne({business\_id: "smkZUv\_IeYYj\_BA6-Po7oQ"})查找指定business\_id的商家文档，然后从该文档中获取其地理坐标信息targetCoordinates = targetBusiness.loc.coordinates。获取目标商家坐标是后续计算距离并筛选附近商家的基础，确保查询能够准确地围绕目标商家进行。

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（目标商家坐标）一定范围内的文档，$geometry指定查询的坐标点类型为Point并传入目标商家坐标，$maxDistance设定最大距离为 2000 米（2 公里）。同时，通过投影操作符{\_id: 0, name: 1, address: 1, stars: 1}指定返回的字段为商家名字、地址和星级，排除默认的\_id字段，然后使用sort({stars: -1})按照星级降序排序，最后使用limit(20)限制返回 20 条结果，并使用forEach(printjson)遍历查询结果并打印，以便查看和分析。

2dsphere 索引的作用与原理

作用

提高地理空间查询性能：2dsphere索引能够快速定位到与给定地理坐标点在一定距离范围内的文档，避免了全表扫描，大大减少了查询时间。在处理大量地理数据（如商家位置信息）时，索引的存在与否对查询效率的影响非常显著。

支持复杂地理空间操作：除了简单的距离查询，还可以支持其他地理空间操作，如判断点是否在多边形内、计算两个地理区域的交集等，为更复杂的地理数据分析提供了基础。

原理

基于空间数据结构：2dsphere索引使用了特定的空间数据结构（如四叉树或 R 树）来组织地理坐标数据。这些数据结构将地理空间划分为不同的区域，使得在查询时可以快速定位到可能包含目标点的区域，然后进一步在该区域内搜索满足条件的文档。例如，四叉树将空间递归地划分为四个子区域，每个子区域可以根据需要进一步细分，从而有效地组织和索引地理数据。

### 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;  });//类似于for循环，这里应该是对应数据流的感觉，处理完最后返回平均值，欸也不是，就是result这一个类，或者叫object  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"}//输出到新的集合  );  //查询MapReduce函数结果  db.Map\_Reduce.find().limit(20).pretty(); |

Map 阶段

数据转换逻辑：在map函数var mapfunc = function() { emit(this.business\_id, {count: 1, sum\_stars: this.stars}); };中，对于Subreview集合中的每个文档（代表一个评价），将其转换为一个键值对。键为business\_id，表示该评价所属的商店，值为一个包含两个属性的对象，count初始化为 1，表示该商店的打分次数，sum\_stars为文档中的stars字段值，表示该次评价的得分。这一步的关键在于将原始的评价数据按照商店进行分组，为后续的计算做准备。例如，如果有三个评价分别属于商店 A、B、C，那么经过map阶段后，会生成三个键值对，分别为 (A, {count: 1, sum\_stars: 该评价的 stars 值})、(B, {count: 1, sum\_stars: 该评价的 stars 值})、(C, {count: 1, sum\_stars: 该评价的 stars 值})。

数据分发机制：emit函数的作用是将生成的键值对输出，以便后续的reduce阶段进行处理。在分布式环境下（MongoDB 的 MapReduce 通常在分布式环境中运行），相同business\_id的键值对会被分发到同一个reduce任务中进行合并计算，确保每个商店的评价数据能够正确地进行聚合。

Reduce 阶段

数据合并逻辑：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; };接收相同business\_id的多个值对象（这些值对象是在map阶段由不同的评价生成的）。函数首先初始化一个结果对象result，包含count（打分次数总和）和sum\_stars（总打分），初始值都为 0。然后遍历值对象数组，将每个值对象中的count和sum\_stars累加到结果对象中。最后计算平均打分avg\_stars，即总打分除以打分次数。例如，如果有两个值对象 {count: 1, sum\_stars: 3} 和 {count: 1, sum\_stars: 4}，经过reduce阶段后，将得到 {count: 2, sum\_stars: 7, avg\_stars: 3.5}。

输出结果格式：reduce函数返回的结果对象将作为最终的聚合结果，包含了每个商店的打分次数、总打分和平均打分信息，符合任务要求的输出格式，即{business\_id: value {count, sum\_stars, avg\_stars}}。

输出阶段

结果存储：通过db.Subreview.mapReduce(mapfunc, reducefunc, {out: "Map\_Reduce"});执行 MapReduce 操作，将结果输出到名为Map\_Reduce的集合中。这个集合将存储每个商店的聚合信息，包括business\_id以及对应的count、sum\_stars和avg\_stars。

结果查询与展示：最后使用db.Map\_Reduce.find().limit(20).pretty();查询Map\_Reduce集合，获取每个商店的平均得分信息，并使用limit(20)限制返回前 20 条数据，使用pretty()格式化输出结果，使其更易读，展示了每个商店的business\_id以及包含打分次数、总打分和平均打分的values对象。

## 3.3 任务小结

（注意：若有不止两项实验任务，则可继续增加章节，后续的综合实践任务五等等章节的编号顺延

（总结本章实验中遇到的主要问题及所采用的解决方案，篇幅不宜过长，可不超过一页。）

# 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 numberOfFoFs  RETURN friend\_name, numberOfFoFs  LIMIT 20  //Started streaming 20 records after 1 ms and completed after 4042 ms. |

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（标题改为具体任务名称）

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.  //AI：WHERE NOT (user1)-[:HasFriend]->(user2) AND user1 <> user2：条件确保找到的`user2`与`user1`不是朋友关系且不是`user1`本身  //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. |

查找 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。

查找与 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。这一步通过多次匹配和条件筛选，准确找到了目标用户群体。

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

优化过结果查询图如下

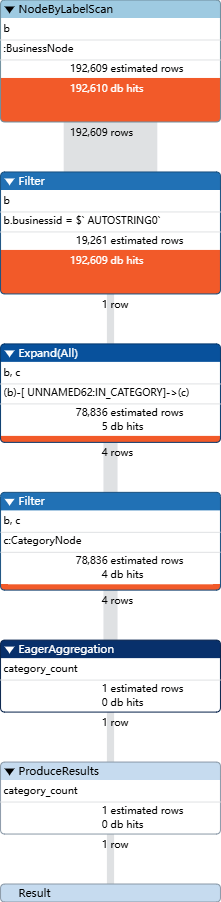


图4.1 优化后查询计划

## 4.3 任务小结

# 5 课程总结

本章篇幅不宜过长，不超过大半页即可。

逐条概括、总结此次课程实践的主要工作，阐述此次课程实践的心得体会，展望此次课程实践的有待改进和完善的工作。

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。通过完成各项任务，不仅掌握了不同数据库的基本操作，还深入理解了它们的特性、适用场景及性能优化方法。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。未来，希望在数据库设计、复杂查询优化、大数据处理等方面进一步提升能力，将所学知识更好地应用于实际项目中，如构建更高效的数据存储系统、优化数据分析流程等，以适应大数据时代对数据管理与处理的高要求。

注意：

整个报告总页数不建议超过25页，但也不要写得太短。每个任务选择部分题目进行报告阐述。

//实验报告（电子版）名称：大数据管理实验\_CST010x\_学号\_姓名

另：要提交实验源码：应包括你所完成的所有题目。建议以任务为单位分源文件（mysql\_json，mongoDB，neo4j，综合，MVCC 5个实验源码文件，如.md文件格式）