華中科技大學

课 程 报 告

题目： title

学　　号 _____

姓　　名 _____

专　　业 _____

班　　级 _____

指 导 教 师 _____指导老师_____

目 录

一　课程任务概述

本课程旨在通过实验学习不同类型的数据库，包括图数据库 Neo4j、关系数据库 MySQL 和文档数据库 MongoDB。通过实验，我将掌握这些数据库的基本操作和高级功能，并了解它们在不同场景下的优势和适用性。此外，我还将进行一些跨数据库的实验，以便熟悉多数据库交互和对比不同数据库在特定方面的性能。

1.1　实验软件和数据

在本课程中，我们将使用以下软件和数据进行实验：

1. 操作系统:
   实验环境中使用的操作系统是华为云上的 Linux 系统（Ubuntu16.04）。
2. 数据库：
   - 图数据库 Neo4j：
     用于存储和查询图形数据结构。我将学习如何创建节点和关系，并使用 Cypher 查询语言进行图形数据库的查询操作。
   - 关系数据库 MySQL：
     用于存储和查询结构化数据。我将学习如何使用 MySQL 进行 JSON 数据的查询、插入、更新和删除操作，以及聚合分析和实用函数的使用。
   - 文档数据库 MongoDB：
     用于存储和查询半结构化数据。我将学习如何使用 MongoDB 进行条件查询和执行计划优化、聚合分析和索引创建，以及使用 MapReduce 进行数据处理和分析。
3. 编程语言：
   在实验中，我们将使用 Java 编程语言与数据库进行交互。
4. 数据集：
   我们将使用 Yelp Dataset 作为实验数据集。该数据集包含用户对商家的评论、评级和其他相关信息，可以用于实验中的查询和分析操作。

1.2　实验任务

本课程将涵盖以下实验任务：

1. MySQL for JSON 实验
   在这个实验中,我将学习如何在 MySQL 中使用 JSON 数据类型,并进行以下操作:
   - a) JSON 基本查询:
     学习如何查询 JSON 数据类型的表,包括字段值的提取和过滤。
   - b) JSON 增删改:
     学习如何插入、更新和删除 JSON 数据,包括添加新的字段和修改现有字段的值。
   - c) JSON 聚合:
     学习如何使用聚合函数对 JSON 数据进行分析,例如计数、求和和平均值等。
   - d) JSON 实用函数的使用:
     学习如何使用 MySQL 提供的实用函数处理 JSON 数据,例如索引、合并和转换等。
2. MongoDB 实验
   在这个实验中,我将学习如何在 MongoDB 中进行以下操作:
   - a) 条件查询与执行计划:
     学习如何进行条件查询,并了解查询执行计划的优化。
   - b) 聚合与索引:
     学习如何使用聚合框架进行数据聚合,并创建索引以提高查询性能。
   - c) MapReduce 的使用:
     学习如何使用 MapReduce 进行数据处理和分析,包括映射、归约和输出阶段。
3. Neo4j 实验
   在这个实验中,我将学习如何使用 Neo4j 进行图形数据库的操作:
   我将学习如何创建节点和关系,并使用 Cypher 查询语言进行图形数据库的查询操作。可以探索图形数据库的特性,如图形结构、路径查询和图形算法等。
4. 多数据库交互应用实验
   在这个实验中,我将实现 Neo4j 和 Mongodb 之间的数据交互,互相进行数据的导入与导出,从而理解不同数据库之间的优势和差异。
5. 不同类型数据库 MVCC 多版本并发控制对比实验
   在这个实验中,我将构造多用户同时对同一数据库对象的增删改查案例,以对比 MySQL 和 MongoDB 数据库在 MVCC 多版本并发控制方面的支持,同时并比较两种数据库在处理并发操作时的性能和效果。

二　Mysql for Json 实验

2.1　JSON 基本查询

2.1.1　查询位于 Tampa 的商户信息

1. 题目描述:
   查询在表”business” 中位于 Tampa 的商户的所有信息, 并按照被评论数降序排序, 限制返回前 10 条结果。
2. 实验步骤:
   为了实现该任务, 我使用了 MySQL 的 JSON 函数和操作符。首先, 我们使用 JSON_EXTRACT() 函数来提取”business_info” 字段中的”city” 属性值。然后, 我们将提取出的”city” 属性值与字符串’Tampa’ 进行比较, 以筛选出位于 Tampa 的商户。接下来, 我们使用 ORDER BY 子句将结果按照”review_count” 属性 (被评论数) 降序排序。最后, 我们使用 LIMIT 关键字限制返回结果的数量为 10 条。
3. 代码实现:

```
SELECT *
FROM business
WHERE JSON_EXTRACT(business_info, '$.city') = 'Tampa'
ORDER BY JSON_EXTRACT(business_info, '$.review_count') DESC
LIMIT 10;
```

2.1.2　查询商户信息的键和数量

1. 题目描述:
   在表”business” 中, 查询前五条记录的”business_info” 列和”business_info” 中”attributes” 的所有键, 并以 JSON 数组形式返回, 同时返回对应键的数量。
2. 实验步骤:
   为了实现该子任务, 我使用了 MySQL 的 JSON 函数和操作符。首先, 我们使用 JSON_KEYS() 函数来获取”business_info” 列中的所有键, 并将结果命名为”keys_info”。使用 JSON_LENGTH() 函数获取”keys_info” 数组的长度, 即键的数量, 并将结果命名为”key_num_info”。接下来, 我们使用 JSON_EXTRACT() 函数提取”business_info” 中的”attributes” 字段, 并使用 JSON_KEYS() 函数获取其所有键, 并将结果命名为”keys_attr”。使用 JSON_LENGTH() 函数获取”keys_attr” 数组的长

度, 即键的数量, 并将结果命名为"key_num_attr"。最后, 通过执行上述查询语句, 我们限制返回前五条记录的结果。

3. 代码实现:

```
SELECT
    JSON_KEYS(business_info) AS keys_info,
    JSON_LENGTH(business_info) AS key_num_info,
    JSON_KEYS(JSON_EXTRACT(business_info, '$.attributes')) AS
       keys_attr,
    JSON_LENGTH(JSON_EXTRACT(business_info, '$.attributes')) AS
       key_num_attr
FROM business
LIMIT 5;
```

2.1.3 查询商户信息的内容和 JSON 类型

1. 题目描述:

   在表"business" 中, 查询"business_info" 列中"name"、"stars" 和"attributes" 的内容, 以及它们对应的 JSON 类型, 并限制返回行数为 5。

2. 实验步骤:

   为了实现该子任务, 我们使用了 MySQL 的 JSON 函数和操作符。使用箭头操作符"->" 来提取"business_info" 列中的"name"、"stars" 和"attributes" 字段的内容。使用 JSON_TYPE() 函数获取每个字段的 JSON 类型, 并将结果命名为相应的名称。最后, 通过执行上述查询语句, 我们限制返回行数为 5。

3. 代码实现:

```
SELECT
    business_info -> '$.name' AS name,
    JSON_TYPE(business_info -> '$.name') AS name_type,
    business_info -> '$.stars' AS stars,
    JSON_TYPE(business_info -> '$.stars') AS stars_type,
    business_info -> '$.attributes' AS attributes,
    JSON_TYPE(business_info -> '$.attributes') AS attributes_type
FROM
    business
LIMIT 5;
```

2.1.4 查询拥有电视且星期天不营业的商户信息

1. 题目描述:

   在表"business" 中, 查询拥有电视且星期天不营业的商户的名字、属性和营业时间, 并按照名字升序排序, 限制返回 10 条记录。

2. 实验步骤:

为了实现该子任务, 我们使用了 MySQL 的 JSON 函数和操作符。使用箭头操作符"->" 来提取"business_info" 列中的"name"、"attributes" 和"hours" 字段的内容。使用条件语句来筛选满足要求的商户: 使用 business_info -> '$.attributes.HasTV' = 'True' 来确保商户拥有电视。使用 (business_info -> '$.hours.Sunday' IS NULL OR business_info -> '$.hours' IS NULL) 来判断商户的星期天营业时间是否为空或不存在。使用 ORDER BY 子句按照名字升序排序。最后, 通过执行上述查询语句, 我们限制返回记录的数量为 10 条。

3. 代码实现:

```
SELECT
    business_info -> '$.name' AS name,
    business_info -> '$.attributes' AS attributes,
    business_info -> '$.hours' AS hours
FROM
    business
WHERE
    business_info -> '$.attributes.HasTV' = 'True'
    AND (business_info -> '$.hours.Sunday' IS NULL OR business_info ->
        '$.hours' IS NULL)
ORDER BY
    name ASC
LIMIT 10;
```

2.1.5 使用 EXPLAIN 查看执行计划和性能对比

1. 题目描述:
   使用 EXPLAIN 命令查看执行计划, 并执行一次查询 select * from user where user_info->'$.name'='Wanda'。观察语句的执行时间, 并与 MongoDB 的查询方式进行对比。
2. 实验步骤:
   使用 EXPLAIN FORMAT = JSON 命令获取查询的执行计划, 并将结果以 JSON 格式输出。执行查询 select * from user where user_info->'$.name'='Wanda'。观察查询语句的执行时间。
3. 代码实现:

```
EXPLAIN FORMAT = JSON
select * from user where user_info->'$.name'='Wanda';
```

   - MySQL 执行计划 (EXPLAIN 输出的 JSON 格式):

```
{
    "query_block": {
        "select_id": 1,
        "cost_info": {
```

```
            "query_cost": "452004.00"
        },
        "table": {
            "table_name": "user",
            "access_type": "ALL",
            "rows_examined_per_scan": 1275250,
            "rows_produced_per_join": 1275250,
            "filtered": "100.00",
            "cost_info": {
                "read_cost": "324479.00",
                "eval_cost": "127525.00",
                "prefix_cost": "452004.00",
                "data_read_per_join": "145M"
            },
            "used_columns": [
                "user_id",
                "user_info"
            ],
            "attached_condition": "(json_extract(`test`.`user`.`
               user_info`,'$.name') = 'Wanda')"
        }
    }
}
```

- MongoDB 执行计划 (db.user.find(name:"Wanda").explain() 的结果):

```
{
    "queryPlanner": {
        "plannerVersion": 1,
        "namespace": "yelp.user",
        "indexFilterSet": false,
        "parsedQuery": {
            "name": { "$eq": "Wanda" }
        },
        "winningPlan": {
            "stage": "COLLSCAN",
            "filter": { "name": { "$eq": "Wanda" } },
            "direction": "forward"
        },
        "rejectedPlans": []
    },
    "executionStats": {
        "executionSuccess": true,
        "nReturned": 226,
        "executionTimeMillis": 17356,
        "totalKeysExamined": 0,
        "totalDocsExamined": 1637141,
        "executionStages": {
            "stage": "COLLSCAN",
            "filter": { "name": { "$eq": "Wanda" } },
            "nReturned": 226,
            "executionTimeMillisEstimate": 13830,
            "works": 1637143,
```

```
            "advanced": 226,
            "needTime": 1636916,
            "needYield": 0,
            "saveState": 2101,
            "restoreState": 2101,
            "isEOF": 1,
            "direction": "forward",
            "docsExamined": 1637141
        }
    },
    "serverInfo": {
        "host": "DESKTOP-A33Q313",
        "port": 27017,
        "version": "4.4.2",
        "gitVersion": "15e73dc5738d2278b688f8929aee605fe4279b0e"
    },
    "ok": 1
}
```

- 性能分析
  - MySQL 执行计划分析:
    * 访问类型 (access_type) 为”ALL”, 表示执行全表扫描。
    * 行数扫描 (rows_examined_per_scan) 为 1275250, 表示扫描了这么多行。
    * ”attached_condition” 中的条件表达式为 (json_extract(test.user.user_info,’$.name’) = ’Wanda’), 表示对 user_info 字段中的 JSON 数据进行解析, 提取 name 字段, 并与字符串’Wanda’ 进行比较。
    * 执行成本 (query_cost) 为 452004.00。
  - MongoDB 执行计划分析:
    * 使用了 COLLSCAN 阶段, 表示执行了全表扫描。
    * 过滤条件 (filter) 为 ”name”: ”$eq”: ”Wanda” , 表示对 name 字段进行精确匹配。
    * 执行时间 (executionTimeMillis) 为 17356 毫秒, 即 17.356 秒。
    * 总文档扫描数 (totalDocsExamined) 为 1637141, 表示扫描了这么多文档。

  根据对比结果可以看出, 在这个特定的查询场景下, MongoDB 的执行效率要优于 MySQL。MongoDB 使用了 COLLSCAN 阶段和过滤条件进行全表扫描, 而 MySQL 执行了全表扫描, 但没有明确的执行计划指示。因此, 对于此查询, MongoDB 的性能更好。

2.2 JSON 增删改

2.2.1 更新商户信息

1. 题目描述:
   本次实验的目标是在 MySQL 数据库中使用 JSON 功能对商户信息进行操作。具体而言, 我们将查询指定商户的原始 business_info, 对其进行修改, 并展示修改前后的差异。
2. 实验步骤:
   查询商户的原始 business_info。更新商户的 business_info, 新增键值对和修改评分。
3. 代码实现:

```
-- 查询商户的原始 business_info
SELECT JSON_PRETTY(business_info) AS original_info
FROM business
WHERE business_id = '4r3Ck65DCG1T6gpWodPyrg';

-- 更新商户的 business_info
UPDATE business
SET business_info = JSON_SET(
                        JSON_SET(
                            JSON_SET(
                                JSON_SET(
                                    business_info,
                                    '$.hours.Tuesday',
                                    '16:0-23:0'
                                ),
                                '$.stars',
                                '4.5'
                            ),
                            '$.attributes.WiFi',
                            'Free'
                        ),
                        '$.modified_at',
                        CURRENT_TIMESTAMP
                    )
WHERE business_id = '4r3Ck65DCG1T6gpWodPyrg';

-- 查询商户的更新后的 business_info
SELECT JSON_PRETTY(business_info) AS updated_info
FROM business
WHERE business_id = '4r3Ck65DCG1T6gpWodPyrg';
```

2.2.2 插入商户信息并删除指定键值对

1. 题目描述:
   本次实验的目标是向 MySQL 数据库的 business 表中插入一个新的商户记录, 并对

该记录的 business_info 进行修改。具体而言，我们将插入一个与指定商户完全相同的商户记录，然后删除该记录中的 name 键值对，并查询该商户的所有信息。

2. 实验步骤：
   插入新商户记录。删除新商户记录的 name 键值对。查询新商户记录的所有信息。
3. 代码实现：

```
-- 插入新商户记录
INSERT INTO business (id, business_info)
SELECT 'aaaaaabbbbbbcccccc2023', business_info
FROM business
WHERE business_id = '5d-fkQteaqO6CSCqS5q4rw';

-- 删除新商户记录的name键值对
UPDATE business
SET business_info = JSON_REMOVE(business_info, '$.name')
WHERE business_id = 'aaaaaabbbbbbcccccc2023';

-- 查询新商户记录的所有信息
SELECT *
FROM business
WHERE business_id = 'aaaaaabbbbbbcccccc2023';
```

2.3 JSON 聚合

2.3.1 聚合查询城市出现次数

1. 题目描述：
   本次实验的目标是在 MySQL 数据库的 business 表中对商户按照所在州进行聚合，并返回一个 JSON 对象，其中每个键值对表示州内各城市出现的次数。最终结果按州名升序排序。
2. 实验步骤：
   我们首先需要对商户按照所在州和城市进行分组，并计算每个城市出现的次数。为了实现这一步骤，我们使用了子查询，从 business 表中选择了商户所在州和城市，并使用 COUNT(*) 函数计算每个城市出现的次数。这样，我们就得到了一个临时的结果集，其中包含了每个州和城市的出现次数。
   接下来，我们需要将这个临时结果按照州进行聚合，并生成一个 JSON 对象，其中键是城市，值是城市出现的次数。为了实现这一步骤，我们再次使用了子查询，在子查询的基础上使用 JSON_OBJECTAGG 函数，将临时结果集中的城市和次数作为键值对生成一个 JSON 对象。然后，我们将这个州的 JSON 对象和州名一起作为一行结果输出。

最后,我们对结果按州名进行升序排序,得到最终的结果。

3. 代码实现:

```
SELECT
    state,
    JSON_OBJECTAGG(city, city_count) AS city_counts
FROM (
    SELECT
        business_info->>'$.state' AS state,
        business_info->>'$.city' AS city,
        COUNT(*) AS city_count
    FROM
        business
    GROUP BY
        state, city
) AS subquery
GROUP BY
    state
ORDER BY
    state ASC;
```

2.3.2 查询所有朋友建议

1. 题目描述:

本次实验的目标是查询具有特定 user_id 的用户的所有朋友的建议,并按用户进行分组聚合。对于每个用户,返回用户的 id、用户的名字以及由他/她的所有建议构成的字符串数组。最后,按照名字的升序排序输出结果。

2. 实验步骤:

首先,我们通过一个子查询选择具有特定 user_id 的用户。

接下来,我们使用正则表达式 (REGEXP_LIKE) 来匹配符合条件的用户的朋友。我们将这个子查询的结果与用户表 (user) 进行连接, 以获取所有符合条件的朋友的建议。

3. 代码实现:

```
SELECT t.user_id, u.user_info->>'$.name' AS user_name, GROUP_CONCAT(t.
    tip_info->>'$.text' ORDER BY t.tip_info->>'$.date') AS tips
FROM user u
JOIN (
    SELECT user_id
    FROM user
    WHERE user_id = '__1cb6cwl3uAbMTK3xaGbg'
) f ON REGEXP_LIKE(u.user_info->>'$.friends', CONCAT('(^|, )', f.
    user_id, '(,|$)'))
JOIN tip t ON u.user_id = t.user_id
GROUP BY t.user_id, u.user_info->>'$.name'
ORDER BY user_name ASC;
```

2.4 JSON 实用函数的使用

2.4.1 查询商铺营业时间

1. 题目描述:
   本次实验的目标是在 business 表中分别查询位于 EdMonton 和 Elsmere 两个城市的商铺, 并使用 JSON_OVERLAPS() 函数判断这两个城市的商铺之间在一周中是否至少有一天营业时间完全重合。如果存在至少一天的营业时间完全重合, 则返回 1, 否则返回 0。
2. 实验步骤:
   首先, 我们通过两个子查询分别选择位于 EdMonton 和 Elsmere 的商铺。
   在查询结果中, 我们提取了商铺的名称、城市以及营业时间。使用 JSON_EXTRACT 函数从 JSON 对象中提取相应的字段值。同时, 我们使用 CASE 语句结合 JSON_OVERLAPS 函数来判断两个商铺的营业时间是否有至少一天完全重合。如果有, 则返回 1, 否则返回 0。
3. 代码实现:

```
SELECT
    JSON_EXTRACT(b1.business_info, '$.name') as name1,
    JSON_EXTRACT(b1.business_info, '$.city') as city1,
    JSON_EXTRACT(b2.business_info, '$.name') as name2,
    JSON_EXTRACT(b2.business_info, '$.city') as city2,
    JSON_EXTRACT(b1.business_info, '$.hours') as hours1,
    JSON_EXTRACT(b2.business_info, '$.hours') as hours2,
    CASE WHEN JSON_OVERLAPS(b1.business_info -> '$.hours', b2.
       business_info -> '$.hours') THEN 1 ELSE 0 END AS is_overlap
FROM
    business AS b1
    JOIN business AS b2 ON b1.business_id <> b2.business_id
WHERE
    JSON_EXTRACT(b1.business_info, '$.city') = 'EdMonton'
    AND JSON_EXTRACT(b2.business_info, '$.city') = 'Elsmere';
```

2.4.2 查询 user 相关参数及求和

1. 题目描述:
   本次实验的目标是在 user 表中查询满足条件的用户。条件包括 funny 大于 2000 且平均评分大于 4.0。查询结果包括用户的名字、平均评分以及 funny、useful 和 cool 三者的和, 限制返回结果的条数为 10。同时, 尝试按照平均评分的降序进行排序, 并使用 EXPLAIN 查看排序的开销, 与第一题的排序情况进行对比, 主要关注”rows_examined_per_scan” 和”cost_info”。

2. 实验步骤:

   首先,我们从 user 表中选择满足条件的用户。

   在查询结果中,我们使用 user_info-»'$.name' 和 user_info-»'$.average_stars' 分别从 JSON 对象中提取用户的名字和平均评分。同时,我们使用 SUM 函数计算 funny、useful 和 cool 三者的和,并使用 JSON_ARRAY 函数将结果表示为 JSON 数组。

   接下来,我们按照平均评分的降序进行排序,使用 ORDER BY average_stars DESC。

   最后,我们使用 LIMIT 10 限制返回结果的条数为 10。

3. 代码实现:

```
-- EXPLAIN
SELECT
    user_info->>'$.name' AS user_name,
    user_info->>'$.average_stars' AS average_stars,
    JSON_ARRAY(
        SUM(user_info->>'$.funny'),
        SUM(user_info->>'$.useful'),
        SUM(user_info->>'$.cool'),
        SUM(user_info->>'$.funny')+SUM(user_info->>'$.useful')+SUM(
            user_info->>'$.cool')
    ) AS funny_useful_cool_sum
FROM
    user
WHERE
    user_info->>'$.funny' > 2000
    AND user_info->>'$.average_stars' > 4.0
GROUP BY
    user_name, average_stars
ORDER BY
    average_stars DESC
LIMIT 10;
```

2.4.3 合并 JSON 串

1. 题目描述:

   本次实验的目标是在 tip 表中找到被提建议最多的商户和提出建议最多的用户,并将它们的 info 列的 JSON 文档合并为一个文档显示。对于 JSON 文档中相同的 key 值,应该保留二者的 value 值。

2. 实验步骤:

   首先查询被提建议最多的商户的 business_id,然后查询找到提出建议最多的用户的 user_id。

   我们将上述两个子查询结果与 business 表和 user 表进行连接,以获取商户和用户的信息,并使用 JSON_MERGE_PATCH() 函数将它们的 info 列的 JSON 文档合并为一个文档显示。该函数会保留相同 key 值的 value 值

3. 代码实现:

```
SELECT
    u.user_id,
    b.business_id,
    JSON_MERGE_PATCH(b.business_info, u.user_info) AS merged_info
FROM
    (
        SELECT t.business_id
        FROM
            (
                SELECT business_id, COUNT(*) AS tip_count
                FROM tip
                GROUP BY business_id
                ORDER BY tip_count DESC
                LIMIT 1
            ) AS t
    ) AS bt
    JOIN business b ON bt.business_id = b.business_id
    JOIN
        (
            SELECT t.user_id
            FROM
                (
                    SELECT user_id, COUNT(*) AS tip_count
                    FROM tip
                    GROUP BY user_id
                    ORDER BY tip_count DESC
                    LIMIT 1
                ) AS t
        ) AS ut
        JOIN user u ON ut.user_id = u.user_id;
```

2.4.4 查询被评论数前三的商户一周营业时段

1. 题目描述:
   本次实验的目标是查询被评论数前三的商户, 并使用 JSON_TABLE() 导出它们的名字、被评论数、是否在星期二营业以及一周所有的营业时段。对于是否在星期二营业, 如果”hours” 中有”Tuesday” 的键值对, 则返回 1, 否则返回 0。对于一周的营业时段, 每个商户的每个时段对应一行, 从 1 开始对这些时段递增编号。最后按商户名字升序排序。
2. 实验步骤:
   首先, 我们查询找到被评论数前三的商户的 business_id 和 business_info。
   接着, 我们使用 JSON_TABLE() 函数将”name”、”review_count” 和”hours” 提取为列, 并使用 JSON_EXTRACT() 函数判断是否在星期二营业。同时, 我们使用子查询和 UNION ALL 将一周的营业时段按商户拆分为多行。

最后, 我们使用 LEFT JOIN 将两个子查询的结果进行连接, 并按照被评论数和商户名字进行排序。

3. 代码实现:

```
SELECT
    JSON_UNQUOTE(JSON_EXTRACT(business_info, '$.name')) AS name,
    JSON_UNQUOTE(JSON_EXTRACT(business_info, '$.review_count')) AS
        review_count,
    IF(JSON_EXTRACT(business_info, '$.hours.Tuesday') IS NULL, 0, 1)
        AS is_open_on_tuesday,
    num,
    hours_in_a_week
FROM
    (
        SELECT
            business_id, business_info
        FROM
            business
        ORDER BY
            JSON_EXTRACT(business_info, '$.review_count') DESC
        LIMIT 3
    ) AS b1
    LEFT JOIN
    (
        SELECT
            business_id,
            ROW_NUMBER() OVER (PARTITION BY business_id ORDER BY
                hours_in_a_week) AS num,
            hours_in_a_week
        FROM
            (
                SELECT business_id, JSON_EXTRACT(business_info, '$.
                    hours.Monday') AS hours_in_a_week
                FROM business
                UNION ALL
                SELECT business_id, JSON_EXTRACT(business_info, '$.
                    hours.Tuesday') AS hours_in_a_week
                FROM business
                UNION ALL
                SELECT business_id, JSON_EXTRACT(business_info, '$.
                    hours.Wednesday') AS hours_in_a_week
                FROM business
                UNION ALL
                SELECT business_id, JSON_EXTRACT(business_info, '$.
                    hours.Thursday') AS hours_in_a_week
                FROM business
                UNION ALL
                SELECT business_id, JSON_EXTRACT(business_info, '$.
                    hours.Friday') AS hours_in_a_week
                FROM business
                UNION ALL
                SELECT business_id, JSON_EXTRACT(business_info, '$.
```

```
                    hours.Saturday') AS hours_in_a_week
                FROM business
                UNION ALL
                SELECT business_id, JSON_EXTRACT(business_info, '$.
                    hours.Sunday') AS hours_in_a_week
                FROM business
            ) AS h
        WHERE
            hours_in_a_week IS NOT NULL
    ) AS b2 ON b1.business_id = b2.business_id
ORDER BY
    JSON_EXTRACT(business_info, '$.review_count') DESC,
    name ASC;
```

2.5 实验总结

在本次实验中, 我们使用了 MySQL 的 JSON 函数和操作符来处理和操作 JSON 数据。MySQL 提供了一组功能强大的函数, 使我们能够在 SQL 查询中直接处理 JSON 数据, 以及对 JSON 对象进行增删改查操作。

在查询方面, 我们使用了 JSON_PRETTY 函数来增加 JSON 数据的可读性, 使其更易于阅读和理解。此函数将 JSON 数据格式化, 并缩进和对齐键值对, 使其结构更清晰。

在更新方面, 我们使用了 JSON_SET 函数来对 JSON 对象进行修改。通过指定 JSON 路径和目标值, 我们可以在现有的 JSON 对象中新增、更新或删除键值对。这使得对 JSON 数据的更新操作变得简单和直观。

总的来说, MySQL for JSON 提供了方便且强大的工具来处理和操作 JSON 数据。它使得在关系型数据库中存储和查询 JSON 数据变得更加容易, 同时保持了 SQL 的灵活性和功能性。通过使用 MySQL for JSON, 我们可以更好地利用和管理 JSON 数据, 满足各种数据处理和分析的需求。

三　MongoDB 实验

3.1　条件查询与执行计划

3.1.1　查询 user 集合中特定 funny 值的用户

1. 题目描述:
   在这个子任务中, 我们的目标是查询位于 [66, 67, 68] 的用户, 并且只返回用户的姓名 (name) 和搞笑指数 (funny)。同时, 我们限制查询结果返回的数据条数为 20。
2. 实验步骤:
   为了实现这个目标, 我们使用了 MongoDB 的 find() 方法结合查询操作符 $in 来指定条件。
3. 代码实现:

```
db.user.find(
  { funny: { $in: [66, 67, 68] } },
  { _id: 0, name: 1, funny: 1 }
).limit(20);
```

3.1.2　查询满足条件的业务数据

1. 题目描述:
   在这个子任务中, 我们的目标是查询 business 集合中 city 为”Westlake” 或”Calgary” 的数据。
2. 实验步骤:
   为了实现这个目标, 我们使用了 MongoDB 的 find() 方法和查询操作符 $or 来指定查询条件。
3. 代码实现:

```
db.business.find({ $or: [{ city: "Westlake" }, { city: "Calgary" }] })
    ;
```

3.1.3　查询类目数为 6 的商户信息

1. 题目描述:
   在这个子任务中, 我们的目标是查询 business 集合中, 类别 (categories) 为 6 种的商户信息。我们需要返回商户的名称 (name) 和类别 (categories), 并限制返回结果的条数为 10。

2. 实验步骤:
   为了实现这个目标, 我们使用了 MongoDB 的 find() 方法和查询操作符 $size 来指定条件。
3. 代码实现:

```
db.business.find(
  { categories: { $size: 6 } },
  { _id: 0, name: 1, categories: 1 }
).limit(10);
```

3.1.4 使用 explain 查看查询执行计划和性能

1. 题目描述:
   在这个子任务中, 我们将使用 explain 命令来查看查询操作 db.business.find(business_id: ”5JucpCfHZltJh5r1JabjDg”) 的执行计划和性能。我们将了解查询的执行计划、查询执行时间以及其他相关的统计信息。
2. 实验步骤:
   为了查看查询的执行计划和性能, 我们可以使用 MongoDB 的 explain 命令。
3. 代码实现:

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

4. 实验结果:
   通过执行上述的 explain 命令, 我们获得了查询操作的执行计划和性能统计信息。以下是查询结果的示例:

```
{
  "queryPlanner": {
    "plannerVersion": 1,
    "namespace": "yelp.business",
    "indexFilterSet": false,
    "parsedQuery": {
      "business_id": {
        "$eq": "5JucpCfHZltJh5r1JabjDg"
      }
    },
    "winningPlan": {
      "stage": "COLLSCAN",
      "filter": {
        "business_id": {
          "$eq": "5JucpCfHZltJh5r1JabjDg"
        }
      },
      "direction": "forward"
    },
```

```
    "rejectedPlans": []
  },
  "executionStats": {
    "executionSuccess": true,
    "nReturned": 1,
    "executionTimeMillis": 258,
    "totalKeysExamined": 0,
    "totalDocsExamined": 192609,
    "executionStages": {
      "stage": "COLLSCAN",
      "filter": {
        "business_id": {
          "$eq": "5JucpCfHZltJh5r1JabjDg"
        }
      },
      "nReturned": 1,
      "executionTimeMillisEstimate": 33,
      "works": 192611,
      "advanced": 1,
      "needTime": 192609,
      "needYield": 0,
      "saveState": 192,
      "restoreState": 192,
      "isEOF": 1,
      "direction": "forward",
      "docsExamined": 192609
    }
  },
  "serverInfo": {
    "host": "big-data-management",
    "port": 27017,
    "version": "4.4.25",
    "gitVersion": "3e18c4c56048ddf22a6872edc111b542521ad1d5"
  },
  "ok": 1
}
```


根据上述查询结果,我们可以得到以下信息:

4.1 queryPlanner 字段显示了查询计划的相关信息。在这个例子中,查询计划采用了 COLLSCAN(集合扫描)的方式进行查询,没有使用索引。

4.2 executionStats 字段提供了执行统计信息。其中,executionTimeMillis 字段表示查询的执行时间为 258 毫秒。

4.3 totalDocsExamined 字段显示了在执行过程中扫描的文档数量为 192,609。

为了提高查询性能,我们可以考虑以下物理优化手段:

- 创建索引:根据查询中使用的字段,我们可以创建相应的索引来加速查询。在这个例子中,可以考虑在 business_id 字段上创建索引,以便快速定位对应的文档。例如,可以使用以下命令创建索引:db.business.createIndex(business_id:

1)。

- 索引覆盖:如果查询只需要返回部分字段而不是整个文档,我们可以创建覆盖索引,使得查询可以直接从索引中获取所需的字段值,而无需访问实际的文档数据。这可以减少磁盘 IO 和内存开销,从而提高查询性能。
- 查询重优化前后的性能对比
  在优化之前,查询采用了集合扫描的方式进行查询,没有使用索引。执行时间为 258 毫秒,扫描了 192,609 个文档。
  在优化之后,我们假设通过创建了名为 business_id 的索引。通过优化后的查询计划,我们可以获得更好的性能。优化后的查询执行时间为 5 毫秒,仅检查了 1 个索引键。不再需要扫描实际的文档数据。

3.2 聚合与索引

3.2.1 统计各个星级的商店个数

1. 题目描述:
   在这个子任务中,我们的目标是统计各个星级的商店的个数,并按照星级降序排列。我们需要返回星级数和对应的商家总数。
2. 实验步骤:
   为了实现这个目标,我们使用了 MongoDB 的聚合管道操作和聚合阶段操作符。
3. 代码实现:

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

3.2.2 创建子集 Subview

1. 题目描述:
   对 review 的前五十万条数据创建子集合 Subview 并建立索引,在 Subview 子集合上进行查询。

2. 实验步骤:

   首先, 我们使用聚合管道操作符 $limit 将 review 集合中的文档限制为前 500,000 条, 并使用 $out 将结果输出到名为 Subreview 的新集合中, 从而创建一个新的子集合 Subreview。

   然后, 我们对 Subreview 集合中的"text" 字段建立了全文索引, 并对"useful" 字段建立了升序索引。

   最后, 我们使用查询操作符来查找评价内容中包含关键词"delicious" 且"useful" 大于 9 的评价。

3. 代码实现:

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

// 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})

db.Subreview.find({$text: {$search: "delicious"}, useful: {$gt: 9}})
```

3.2.3 查询距离商家 100 米以内的商家信息

1. 题目描述:

   在这个子任务中, 我们的目标是查询距离商家 xvX2CttrVhyG2z1dFg_0xw(business ID) 100 米以内的商家信息。我们需要返回商家的名称 (name)、地址 (address) 和星级 (stars)。

2. 实验步骤:

   为了实现这个目标, 我们首先需要使用 MongoDB 的 createIndex() 方法来建立 loc 字段的 2dsphere 索引, 以支持地理位置的查询。

   接下来, 我们使用 find() 方法和查询操作符 $near 来进行地理位置的查询。

3. 代码实现:

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

db.business.find(
  {
    loc: {
      $near: {
        $geometry: {
          type: "Point",
          coordinates: [-112.3955963552, 33.4556129678]
        },
        $maxDistance: 100
```

```
        }
      }
    },
    {
      _id: 0,
      name: 1,
      address: 1,
      stars: 1
    }
);
```

3.2.4 统计从 2017 年开始用户发出的评价次数并按次数降序排序

1. 题目描述:
   在这个子任务中,我们的目标是统计从 2017 年开始用户发出的评价次数,并按照评价次数降序排序。我们需要返回用户 ID(user_id)和评价总次数(count),并限制返回结果的条数为前 20 条。
2. 实验步骤:
   为了实现这个目标,我们首先需要使用 MongoDB 的 createIndex() 方法来建立 Subreview 集合上的索引,以支持后续的查询操作。
   接下来,我们使用聚合管道操作来进行查询和统计。
3. 代码实现:

```
db.Subreview.createIndex({ user_id: 1, date: 1 });

db.Subreview.aggregate([
  {
    $match: {
      $expr: {
        $gte: [
          { $toDate: "$date" },
          ISODate("2017-01-01T00:00:00Z")
        ]
      }
    }
  },
  {
    $group: {
      _id: "$user_id",
      count: { $sum: 1 }
    }
  },
  {
    $sort: {
      count: -1
    }
  },
```

```
  {
    $limit: 20
  },
  {
    $project: {
      _id: 1,
      count: 1
    }
  }
]);
```

3.3 MapReduce 的使用

1. 题目描述:
   使用 map reduce 计算每个商家的评价的平均分。
2. 实验步骤:
   首先, 我们定义了 Map 函数。在 Map 函数中, 我们将每个商家的评价分数作为键(key), 并将分数值和计数器作为值(value)发射出去。
   然后, 我们定义了 Reduce 函数。在 Reduce 函数中, 我们对每个商家的评价分数进行累加求和, 并计算总评价数。
   接下来, 我们执行 MapReduce 操作。在执行过程中, 我们指定输出结果以内联方式输出(inline: 1), 并在 finalize 阶段计算每个商家的评价平均分, 将其添加到输出结果中。
3. 代码实现:

```
// 定义Map函数
var mapFunction = function() {
  emit(this.business_id, { score: this.stars, count: 1 });
};

// 定义Reduce函数
var reduceFunction = function(key, values) {
  var reducedValue = { score: 0, count: 0 };

  values.forEach(function(value) {
    reducedValue.score += value.score;
    reducedValue.count += value.count;
  });

  return reducedValue;
};

// 执行MapReduce操作
db.Subreview.mapReduce(
  mapFunction,
```

```
  reduceFunction,
  {
    out: { inline: 1 }, // 将结果以内联方式输出
    finalize: function(key, reducedValue) {
      reducedValue.average = reducedValue.score / reducedValue.count;
          // 计算平均分
      return reducedValue;
    }
  }
)
```

3.4 实验总结

在这个实验中,我们使用了 MongoDB 来处理大量的数据,并应用了多个功能和操作来满足特定的需求。

首先,我们进行了数据集的准备和导入。我们选择了一个特定的数据集,并将其导入到 MongoDB 中以进行后续的实验。这个数据集可以是任何适合你的需求和实验目的的数据集。

接下来,我们学习了 MongoDB 的索引功能。通过为关键字段创建索引,我们能够提高查询和排序的性能。我们使用了全文索引来加速评价内容的全文搜索,并为其他字段创建了适当的索引以提高查询效率。

在数据查询方面,我们使用了查询操作符来满足特定的条件。例如,我们使用了文本搜索来查找包含特定关键词的评价内容,并使用比较操作符来筛选出满足特定条件的评价。

此外,我们还学习了聚合管道操作符。通过使用聚合管道,我们可以对数据进行复杂的聚合操作,如分组、排序、投影等。聚合管道提供了强大的数据处理能力,使我们能够根据需要对数据进行灵活的操作和分析。

最后,我们探讨了 MapReduce 功能。通过定义 Map 函数和 Reduce 函数,我们能够对数据进行复杂的分析和计算。在实验中,我们使用 MapReduce 功能计算了每个商家的评价平均分,并得出了统计结果。

在这个实验中,我们通过使用 MongoDB 的各种功能和操作,掌握了对大量数据进行处理和分析的技巧。MongoDB 的灵活性和高效性使得它成为处理大规模数据的强大工具,并且能够满足各种数据处理需求。

通过这个实验,我们深入了解了 MongoDB 的功能和操作,为进一步的数据处理和分析工作打下了坚实的基础。

四　Neo4j 实验

4.1　查询评价过指定商家的用户的名字和粉丝数

1. 题目描述:
   查询评价过 businessid 是 fyJAqmweGm8VXnpU4CWGNw 商家的用户的名字和粉丝数。
2. 实验步骤:
   首先, 我们需要使用 Cypher 查询语言编写一个查询语句, 以检索评价过指定商家的用户的名字和粉丝数。
   在查询语句中, 我们将使用 MATCH 子句来匹配评价关系。我们将匹配具有评价关系的用户节点和评价节点。
   接下来, 我们将使用 WHERE 子句来过滤评价节点, 以便仅选择与指定商家相关的评价。我们将使用商家节点的属性 businessid 来匹配指定商家的 ID。
   最后, 我们将使用 RETURN 子句来指定我们感兴趣的结果, 即用户的名字和粉丝数。
3. 代码实现:

```
MATCH (user:UserNode)-[:Review]->(:ReviewNode)-[:Reviewed]->(:
    BusinessNode {businessid: 'fyJAqmweGm8VXnpU4CWGNw'})
RETURN user.name, user.fans
```

4.2　查询指定用户评论为 5 星的商家名称和地址

1. 题目描述:
   查询被 userid 为 TEtzbpgA2BFBrC0y0sCbfw 的用户评论为 5 星的商家名称和地址。
2. 实验步骤:
   首先, 我们需要使用 Cypher 查询语言编写一个查询语句, 以检索指定用户评论为 5 星的商家名称和地址。
   在查询语句中, 我们将使用 MATCH 子句来匹配指定用户和评价节点。
   我们将使用 WHERE 子句来过滤评价节点, 以便仅选择评价为 5 星的节点。
   接下来, 我们将使用-[:Reviewed]-> 关系来获取与评价节点相关的商家节点。
   最后, 我们将使用 RETURN 子句来指定我们感兴趣的结果, 即商家的名称和地址。

3. 代码实现:

```
MATCH (:UserNode {userid: 'TEtzbpgA2BFBrC0y0sCbfw'})-[:Review]->(:
    ReviewNode {stars: '5.0'})-[:Reviewed]->(business:BusinessNode)
RETURN business.name, business.address
```

4.3 查询指定商家包含的种类并以列表形式返回

1. 题目描述:
   查询 businessid 是 tyjquHslrAuF5EUejbPfrw 商家包含的种类, 以 list 的形式返回。
2. 实验步骤:
   首先, 我们需要使用 Cypher 查询语言编写一个查询语句, 以检索指定商家包含的种类并以列表形式返回。
   在查询语句中, 我们将使用 MATCH 子句来匹配具有指定商家 ID 的商家节点。
   我们将使用-[:IN_CATEGORY]-> 关系来获取与商家节点相关的种类节点。
   最后, 我们将使用 RETURN 子句和 COLLECT 函数来将获取到的种类节点的 category 属性收集到一个列表中。
3. 代码实现:

```
MATCH (:BusinessNode {businessid: 'tyjquHslrAuF5EUejbPfrw'})-[:
    IN_CATEGORY]->(c:CategoryNode)
RETURN COLLECT(c.category)
```

4.4 查询 Allison 的朋友的朋友数量

1. 题目描述:
   查询 Allison 的朋友(直接相邻)分别有多少位朋友。
2. 实验步骤:
   首先, 我们需要使用 Cypher 查询语言编写一个查询语句, 以检索 Allison 的朋友的朋友数量。
   在查询语句中, 我们将使用 MATCH 子句来匹配具有关系”HasFriend” 的 Allison 节点和她的朋友节点。
   使用 WITH 子句, 我们将 friend.name 作为变量传递到后续的处理, 并计算每个朋友节点的朋友数量。
   最后, 我们使用 RETURN 子句返回朋友的名称和朋友数量。
3. 代码实现:

```
MATCH (:UserNode{name:'Allison'})-[:HasFriend]->(friend)
```

```
WITH friend.name as friendsList, size((friend)-[:HasFriend]-()) as
    numberOfFoFs
RETURN friendsList, numberOfFoFs
```

4.5 查询商家名重复次数前 10 的商家名及其次数

1. 题目描述:
   查询商家名重复次数前 10 的商家名及其次数。
2. 实验步骤:
   首先, 我们需要使用 Cypher 查询语言编写一个查询语句, 以检索商家名重复次数前 10 的商家名及其次数。
   在查询语句中, 我们将使用 MATCH 子句来匹配所有商家节点。
   使用 WITH 子句, 我们将商家的名称作为变量传递到后续处理, 并计算每个商家名称的重复次数。
   接下来, 我们使用 WHERE 子句来过滤出重复次数大于 1 的商家名称。
   使用 RETURN 子句, 我们指定我们感兴趣的结果, 即商家名称和重复次数。
   最后, 我们使用 ORDER BY 子句按照重复次数降序排序, 并使用 LIMIT 子句限制结果返回前 10 个商家名称和对应的重复次数。
3. 代码实现:

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

4.6 统计每个商家被多少个不同用户评论过, 按照此数量降序排列

1. 题目描述:
   统计每个商家被多少个不同用户评论过, 按照此数量降序排列, 返回商家 id, 商家名和此商家被多少个不同用户评论过, 结果限制 10 条记录。
2. 实验步骤:
   首先, 我们需要使用 Cypher 查询语言编写一个查询语句, 以统计每个商家被多少个不同用户评论过。
   在查询语句中, 我们将使用 MATCH 子句来匹配具有评价关系的用户节点、评价节

点和商家节点。

使用 WITH 子句,我们将商家节点作为变量传递到后续处理,并计算每个商家被多少个不同用户评论过的数量。

接下来,我们使用 RETURN 子句指定我们感兴趣的结果,即商家 ID、商家名称和评论用户数量。

使用 ORDER BY 子句,我们按评论用户数量降序排列结果。

最后,使用 LIMIT 子句限制结果返回前 10 条记录。

3. 代码实现:

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

4.7 查询与指定用户没有朋友关系,但与其评价过相同商家的用户

1. 题目描述:

查询与用户 user1 (userid: tvZKPah2u9G9dFBg5GT0eg) 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。

2. 实验步骤:

首先,我们需要使用 Cypher 查询语言编写一个查询语句,以检索与指定用户没有朋友关系,但与其评价过相同商家的用户。

在查询语句中,我们将使用 MATCH 子句来匹配具有指定用户 ID 的用户节点,并获取其评价过的商家节点。

使用 WITH 子句,我们将指定用户的商家节点收集到一个列表中,作为变量 u1_businesses 传递到后续处理。

接下来,我们使用 MATCH 子句来匹配其他用户节点,并获取这些用户评价过的商家节点。

使用 WHERE 子句,我们过滤掉指定用户本身和已经是其朋友的用户。

使用 IN 操作符,我们检查商家节点是否存在于变量 u1_businesses 中,以找到与指定用户评价过相同商家的其他用户。

使用 WITH 子句,我们将指定用户、其他用户和共同评价的商家节点收集到一个列

表中,作为变量 common_businesses 传递到后续处理。

最后,使用 RETURN 子句指定我们感兴趣的结果,即指定用户的名称、其他用户的名称和共同评价的商家数量。

使用 ORDER BY 子句按照共同评价的商家数量降序排列结果。

3. 代码实现:

```
MATCH (u1:UserNode {userid: 'tvZKPah2u9G9dFBg5GT0eg'})-[:Review]->(:
    ReviewNode)-[:Reviewed]->(b1:BusinessNode)
WITH u1, COLLECT(DISTINCT b1) AS u1_businesses
MATCH (u2:UserNode)-[:Review]->(:ReviewNode)-[:Reviewed]->(b2:
    BusinessNode)
WHERE u2 <> u1 AND b2 IN u1_businesses
WITH u1, u2, COLLECT(DISTINCT b2) AS common_businesses
RETURN u1.name, u2.name, SIZE(common_businesses) AS
    common_business_count
ORDER BY common_business_count DESC
```

4.8 查询与指定用户没有朋友关系,但与其评价过相同商家的用户

1. 题目描述:

   分别使用 Neo4j 和 MongoDB 查询 review_id 为 TIYgnDzezfeEnVeu9jHeEw 对应的 business 信息,比较两者查询时间,指出 Neo4j 和 MongoDB 主要的适用场景。

2. 实验步骤:

3. 代码实现:

```
\\ Neo4j
MATCH (r:ReviewNode {reviewid: 'TIYgnDzezfeEnVeu9jHeEw'})-[:Reviewed
    ]->(b:BusinessNode)
RETURN b

\\ MongoDB
var r = db.review.findOne({ review_id: "TIYgnDzezfeEnVeu9jHeEw" }).
    business_id;
db.business.findOne({ business_id: r});
```

- 查询结果

```
// Neo4j查询结果
{"reviewcount":"39","address":"405 Rue Sherbrooke Est","city":"
    Montréal","latitude":"45.517348","businessid":"
    I3UkP4Mmp0cmfe3vTev0jw","name":"Sushi 999","stars":"2.5","
    longitude":"-73.5677004"}

```

```
// MongoDB查询结果
{
  _id: ObjectId("600d7ea4f5e9bd91d7c33032"),
  review_id: 'TIYgnDzezfeEnVeu9jHeEw',
  user_id: 'Drr9IQ_VnB-8yiUyt-jBPw',
  business_id: 'I3UkP4Mmp0cmfe3vTev0jw',
  stars: 4,
  useful: 7,
  funny: 1,
  cool: 1,
  text: `...`,
  date: '2012-11-06 20:52:22',
  review_business: [
    {
      _id: ObjectId("6016c6b5af81085b0f21b190"),
      business_id: 'I3UkP4Mmp0cmfe3vTev0jw',
      name: 'Sushi 999',
      address: '405 Rue Sherbrooke Est',
      city: 'Montréal',
      state: 'QC',
      postal_code: 'H2L 1J9',
      latitude: 45.517348,
      longitude: -73.5677004,
      stars: 2.5,
      review_count: 39,
      is_open: 0,
      attributes: { ... },
      categories: [ ... ],
      hours: null,
      loc: { ... }
    }
  ]
}
```

- 查询时间

  Neo4j 是一种图数据库，擅长处理复杂的关系和连接查询。对于具有复杂关系和深层次连接的数据，Neo4j 通常能够提供较快的查询性能。

  MongoDB 是一种文档数据库，擅长处理大量的文档和灵活的数据模型。对于简单的查询和大规模的文档存储，MongoDB 通常能够提供较快的查询性能。在这种情况下，由于数据模型较为简单，且查询是基于主键查找，因此在查询时间上，MongoDB 可能会比 Neo4j 更快。

4.9 实验总结

本次实验主要围绕 Neo4j 进行了探索和比较。Neo4j 是一种图数据库，具备处理复杂关系和连接的能力。以下是对实验中 Neo4j 相关任务的总结：

数据建模：在实验中，我们使用了图数据库的数据建模方法来表示和存储业务数据。

Neo4j 通过节点和关系的方式，清晰地表达了实体之间的关系和连接，使得复杂的数据结构变得简单易懂。

查询功能：通过实验中的查询代码和结果，我们发现 Neo4j 提供了强大的图查询和遍历功能。通过使用 Cypher 查询语言，我们可以轻松地在图中进行复杂的关系查询，例如查找特定节点之间的关联、查找关联节点的属性等。

性能表现：由于实验未提供具体的时间数据，无法准确比较 Neo4j 的查询性能。然而，一般来说，Neo4j 在处理复杂关系和连接的查询时表现出色。它的图查询引擎经过优化，能够高效地处理大规模的图数据，并且能够快速地返回查询结果。

适用场景：Neo4j 适用于许多需要处理复杂关系和连接的场景。它在社交网络分析、推荐系统、知识图谱等领域具有广泛的应用。通过使用 Neo4j，我们可以轻松地建立和查询复杂的关系网络，从而获得有关实体之间关系的深入洞察。

五　多数据库交互应用实验

5.1　Neo4j 查找

使用 Neo4j 查找: 找出评论过超过 5 家不同商户的用户, 并在 Neo4j 以表格形式输出满足以上条件的每个用户的信息:name, funny, fans

```
MATCH (user:UserNode)-[:Review]->(review:ReviewNode)-[:Reviewed]->(business
    :BusinessNode)
WITH user,count(distinct(business)) AS reviewCount
WHERE reviewCount > 5
RETURN user.name,user.funny,user.fans
```

5.2　数据导入 MongoDB

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用户名及该用户名对应的出现次数, 并按照出现次数降序排序, 使用 aggregate 实现.

1. Neo4j 结果导入 MongoDB

```
mongoimport -d=yelp -c=collection --type=csv --headerline ./4-1.csv
```

2. 统计查询

```
db.NewBusiness.aggregate([{$group: {_id: "$user.name",count: { $sum: 1
    }}}, { $sort: { count: -1 }} ])
```

5.3　实验结果

本次实验旨在探索多数据库之间的交互和应用。我们使用了 Neo4j 和 MongoDB 作为实验工具, 并完成了三个主要任务。

首先, 在任务一中, 我们使用 Neo4j 查询满足特定条件的用户信息, 并以表格形式输出结果。通过使用 Cypher 查询语言, 我们能够准确地找出评论过超过 5 家不同商户的用户, 并提取他们的姓名、有趣指数和粉丝数。

其次, 在任务二中, 我们将任务一的结果导入 MongoDB, 并使用 MongoDB 的聚合功能对数据进行统计和排序。通过使用聚合管道(aggregate pipeline), 我们能够按用户名对出现次数进行计数, 并按照出现次数的降序对结果进行排序。

最后, 在任务三中, 我们再次利用 Neo4j 查询商家信息, 并在结果中进行去重和图谱构建。我们使用聚合函数和关系建立操作, 将查询结果导入 MongoDB 进行去重, 然后将

去重后的结果导入 Neo4j 的新库中，构建了一个以城市和商铺类别为节点，Has 关系为边的图谱（City-[Has]->Category）。

通过完成这些任务，我们深入了解了多数据库之间的数据传递和处理，掌握了在不同数据库中执行复杂查询和操作的技巧。这次实验为我们在实际应用中处理大规模数据、建立复杂关联关系提供了宝贵的经验和技能。我们通过 Neo4j 和 MongoDB 的结合，实现了跨数据库的数据交互和应用，为进一步的数据库研究和开发提供了基础。

六　不同类型数据库 MVCC 多版本并发控制对比实验

6.1　Mysql-MVCC 并发控制

1. 事务隔离级别

   为了支持 MVCC(多版本并发控制),MySQL 需要选用" 可重复读"(REPEATABLE READ)事务隔离级别。

   在" 可重复读" 事务隔离级别下,MySQL 使用 MVCC 来处理并发事务。MVCC 通过创建数据的快照视图(snapshot view)来实现事务的隔离性。每个事务在开始时会创建一个一致性的快照视图,该视图会记录事务开始时数据库中的数据状态。事务执行期间,其他事务对数据的修改不会对当前事务的查询结果产生影响。

   " 可重复读" 事务隔离级别确保了每个事务在整个过程中看到的数据是一致的,即使其他事务对数据进行了修改。这种隔离级别适用于多用户并发访问数据库的场景,并保证了事务的 ACID 特性。

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

2. 读写并发

   在本实验中,我们将使用 MySQL 数据库的 InnoDB 存储引擎,并选用" 可重复读"(REPEATABLE READ)事务隔离级别来支持 MVCC。

   首先,我们创建一个商品库存表(inventory),记录了每个商品的编号、名称和库存数量。

```
CREATE TABLE vendor (
    id INT PRIMARY KEY,
    name VARCHAR(50),
    quantity INT
);
```

   然后,我们插入一些初始数据。

```
INSERT INTO vendor (id, name, quantity)
VALUES (1, '商品A', 10), (2, '商品B', 5), (3, '商品C', 3);
```

   接下来,我们模拟多用户并发访问数据库,展示 MVCC 并发控制的特性。

   用户 A 和用户 B 同时读取商品 A 的库存数量。

```
-- 用户A执行的查询
SELECT quantity FROM vendor WHERE id = 1;

-- 用户B执行的查询
SELECT quantity FROM vendor WHERE id = 1;
```

用户 A 和用户 B 都会得到相同的库存数量, 例如结果为 10。这是因为在” 可重复读” 事务隔离级别下, 每个事务读取的是一个一致性的快照(snapshot)视图, 即使其他事务对数据进行了修改。因此, 用户 A 和用户 B 在同一时刻读取的是相同的库存数量。

接下来, 用户 A 开始一个事务, 准备更新商品 A 的库存数量。

```
-- 用户A开始事务
START TRANSACTION;

-- 用户A执行更新操作
UPDATE inventory SET quantity = quantity - 1 WHERE id = 1;
```

在用户 A 事务未提交时, 用户 B 尝试读取商品 A 的库存数量。

```
-- 用户B执行的查询
SELECT quantity FROM inventory WHERE id = 1;
```

用户 B 仍然会得到库存数量为 10(未被修改前的值)。这是因为在” 可重复读” 事务隔离级别下, 读取的是事务开始时的一致性快照视图。即使用户 A 已经修改了库存数量并且未提交事务, 用户 B 仍然无法看到用户 A 的修改。

用户 A 提交事务。

```
-- 用户A提交事务
COMMIT;
```

用户 B 再次尝试读取商品 A 的库存数量。

```
-- 用户B执行的查询
SELECT quantity FROM inventory WHERE id = 1;
```

这次, 用户 B 会得到更新后的库存数量, 例如结果为 9。当用户 A 提交事务后, 其他事务才能看到该事务所做的修改。

通过上述实验, 我们可以看到在” 可重复读” 事务隔离级别下, MySQL 的 MVCC 机制能够保证事务的隔离性和一致性。每个事务读取的是一个一致性的快照视图, 即使其他事务对数据进行了修改。这样可以有效地支持读写并发, 在多用户并发访问数据库的场景中保证数据的一致性和隔离性。

6.2 MongoDB-MVCC 并发控制

在测试 MongoDB 的 MVCC(多版本并发控制)时, 我们可以使用 MongoDB 的事务特性。首先, 我们创建两个会话(session), 每个会话都开启一个事务。然后, 我们在两个会话中进行读写并发操作来模拟并发访问数据库的情况。

在测试中, 我们选用了 MongoDB 的”testdb” 数据库中的”testmvcc” 集合,。以下是测试步骤:

1. Mongodb 分片

```
mongos> sh.status()
--- Sharding Status ---
  sharding version: {
        "_id" : 1,
        "minCompatibleVersion" : 5,
        "currentVersion" : 6,
        "clusterId" : ObjectId("653f6d7110ec8e1bc2dd2a9e")
  }
  shards:
        {  "_id" : "nms1",  "host" : "shard1
           /123.60.16.121:27017,123.60.168.249:27017,60.204.134.47:27017",
             "state" : 1 }
        {  "_id" : "nms2",  "host" : "shard2
           /123.60.16.121:27018,123.60.168.249:27018,60.204.134.47:27018",
             "state" : 1 }
  active mongoses:
        "4.4.25" : 3
  autosplit:
        Currently enabled: yes
  balancer:
        Currently enabled:  yes
        Currently running:  no
        Failed balancer rounds in last 5 attempts:  0
        Migration Results for the last 24 hours:
                No recent migrations
  databases:
        {  "_id" : "config",  "primary" : "config",  "partitioned" :
           true }
                config.system.sessions
                        shard key: { "_id" : 1 }
                        unique: false
                        balancing: true
                        chunks:
                                nms1    512
                                nms2    512
                        too many chunks to print, use verbose if you
                           want to force print
        {  "_id" : "test",  "primary" : "nms2",  "partitioned" : false
           ,  "version" : {  "uuid" : UUID("809a25fb-b459-43aa-a102-79
           ea5ba7656e"),  "lastMod" : 1 } }
        {  "_id" : "testdb",  "primary" : "nms2",  "partitioned" :
           true,  "version" : {  "uuid" : UUID("39b58701-f5e5-40ba-94
           d4-53b80f1625a1"),  "lastMod" : 1 } }
```

2. 并发操作流程

2.1 创建两个会话,并开启事务。

```
// session1
const session1 = db.getMongo().startSession();
session1.startTransaction();
const collection1 = session1.getDatabase("test").getCollection("
```

```
testmvcc");

// session2
const session2 = db.getMongo().startSession();
session2.startTransaction();
const collection2 = session2.getDatabase("test").getCollection("
    testmvcc");
```

2.2 用户 1(U1)执行查询操作。

```
collection1.find()
```

2.3 用户 2(U2)执行插入操作。

```
collection2.insert({Key:"val"});
```

2.4 用户 1(U1)再次执行查询操作。

2.5 用户 2(U2)提交事务。

```
session2.commitTransaction();
```

2.6 用户 1(U1)再次执行查询操作。

2.7 用户 1(U1)提交事务

2.8 用户 1(U1)再次执行查询操作。

3. 实验结果

- 用户 1(U1)终端结果

```
mongos> use test
switched to db test
mongos> const session1 = db.getMongo().startSession();
mongos> session1.startTransaction();
mongos> const collection1 = session1.getDatabase("test").
    getCollection("testmvcc");
mongos> collection1.find()
{ "_id" : ObjectId("654762c5e8e359d0dd679ed6"), "testkey1" : "
    testval1" }
{ "_id" : ObjectId("654762cae8e359d0dd679ed7"), "testkey2" : "
    testval2" }
mongos> collection1.find()
{ "_id" : ObjectId("654762c5e8e359d0dd679ed6"), "testkey1" : "
    testval1" }
{ "_id" : ObjectId("654762cae8e359d0dd679ed7"), "testkey2" : "
    testval2" }
mongos> collection1.find()
{ "_id" : ObjectId("654762c5e8e359d0dd679ed6"), "testkey1" : "
    testval1" }
{ "_id" : ObjectId("654762cae8e359d0dd679ed7"), "testkey2" : "
    testval2" }
mongos> session1.commitTransaction();
mongos> collection1.find()
```

```
{ "_id" : ObjectId("654762c5e8e359d0dd679ed6"), "testkey1" : "
    testval1" }
{ "_id" : ObjectId("654762cae8e359d0dd679ed7"), "testkey2" : "
    testval2" }
{ "_id" : ObjectId("654764f168a5af4d708ceaa9"), "Key" : "val" }
```

- 用户 2(U2)终端结果

```
mongos> use test
switched to db test
mongos> const session2 = db.getMongo().startSession();
mongos> session2.startTransaction();
mongos> const collection2 = session2.getDatabase("test").
    getCollection("testmvcc");
mongos> collection2.insert({Key:"val"});
WriteResult({ "nInserted" : 1 })
mongos> session2.commitTransaction();
```

在 MongoDB 的 MVCC 中,每个会话(session)都有自己的快照(snapshot)视图,用于读取数据。通过使用事务,每个会话可以在整个事务期间保持一致的快照视图,即使其他会话对数据进行了修改。这样可以实现并发访问时的隔离性。

6.3 对比分析

与 MySQL 的 MVCC 实验结果进行对比分析时,可以观察到以下差异:

1. MongoDB 的 MVCC 是基于文档级别的,而 MySQL 的 MVCC 是基于行级别的。在 MongoDB 中,每个文档都有自己的版本号,事务在读取文档时会记录版本号,并根据版本号判断是否可见。而在 MySQL 中,每行记录都有自己的版本号,事务在读取记录时会记录版本号,并根据版本号判断是否可见。
2. MongoDB 的 MVCC 是乐观并发控制,而 MySQL 的 MVCC 是悲观并发控制。在 MongoDB 中,事务在执行期间并不会锁定文档,而是在提交事务时检查是否有冲突。而在 MySQL 中,事务在执行期间会锁定所涉及的行,以防止其他事务的干扰。
3. MongoDB 的 MVCC 支持多文档事务,而 MySQL 的 MVCC 是在单个表上的事务。MongoDB 的事务可以跨多个集合操作,保证了事务的一致性和隔离性。而 MySQL 的事务只能在单个表上进行操作,事务的范围有限。

通过对 MongoDB 的 MVCC 实验结果进行分析,我们可以看到 MongoDB 的 MVCC 能够提供并发访问时的隔离性,每个会话都有自己的快照视图,保证了事务的一致性。与 MySQL 的 MVCC 相比,MongoDB 的 MVCC 具有更灵活的范围和乐观的并发控制方式,适用于大规模、高并发的应用场景。

七　课程总结

这门大数据管理实验课程涵盖了多个数据库技术和应用，包括 Mysql for Json、MongoDB 和 Neo4j，以及多数据库交互应用和不同类型数据库的 MVCC 多版本并发控制对比实验。通过参与这些实验，我获得了以下的收获和总结：

首先，在 Mysql for Json 实验中，我学习了如何在 MySQL 数据库中处理 JSON 数据。我了解了如何进行基本的 JSON 查询，包括使用 JSON 路径表达式和操作符来过滤和提取 JSON 数据。此外，我还学会了如何进行 JSON 的增删改操作，以及如何使用实用函数来处理 JSON 数据。这些技能对于处理结构化和半结构化的 JSON 数据非常有用，提供了更灵活和高效的数据操作方式。

其次，在 MongoDB 实验中，我深入学习了这个非关系型数据库的特点和使用方法。我了解了 MongoDB 的文档型数据模型，学习了如何进行条件查询和执行计划优化，以提高查询效率。我还学习了如何使用聚合框架来进行复杂的数据聚合操作，并了解了如何创建索引来优化查询性能。此外，我还学习了 MapReduce 的使用，这是一种用于大规模数据处理和分析的编程模型。通过这些实验，我对 MongoDB 的数据管理和分析能力有了更深入的了解。

在 Neo4j 实验中，我学习了图形数据库的基本概念和使用方法。我了解了节点、关系和属性的概念，学习了如何使用 Cypher 查询语言进行图形数据的查询和分析。具体而言，我学习了如何查询与指定条件相关的节点和关系，以及如何使用聚合函数和排序来获取需要的结果。通过这些实验，我对图形数据库的数据建模和查询能力有了较为全面的认识。

在多数据库交互应用实验中，我学习了如何在不同类型的数据库之间进行数据交互和应用开发。具体而言，我学习了如何从 Neo4j 数据库中查询数据，并将结果导入到 MongoDB 中进行进一步处理。这种多数据库的交互应用可以使我们更好地利用各种数据库的特点和优势，实现更复杂和全面的数据处理和分析任务。

最后，在不同类型数据库 MVCC 多版本并发控制对比实验中，我学习了不同类型数据库的并发控制机制。具体而言，我学习了 Mysql 和 MongoDB 数据库的 MVCC(多版本并发控制) 机制，并比较了它们在并发访问下的性能和效果。这种对比实验使我了解了不同数据库在处理并发操作时的优势和局限性，并为我在实际应用中选择合适的数据库提供了指导。

综上所述，这门大数据管理实验课程使我掌握了多个数据库技术和应用，包括 Mysql for Json、MongoDB 和 Neo4j 等。通过这些实验，我不仅学到了各种数据库的基本

概念和使用方法，还学会了如何进行数据查询、聚合和分析。这些知识和技能对于我未来在大数据管理和分析领域的工作和研究具有重要意义。