Data Warehouse Project

Design Document

电影查询系统

数据存储设计说明文档

组长：赵敏

组员：陈一帆 时守格 王瀚林

目 录

I 项目简介 4

II 实现功能 4

III 系统架构 5

3.1 数据库 5

3.1.1 MySQL 5

3.1.1 Neo4j 5

3.1.1 Hive 5

3.2 前后端技术 5

3.2.1 前端技术 6

3.2.2 后端技术 6

IV ETL 6

4.1 数据爬取 6

4.1.1 爬取过程 6

4.1.2 反爬措施 8

4.2 数据清洗 9

4.2.1 数据解析 9

4.2.2 信息规范化 9

4.2.3 删除干扰信息 9

4.2.4 过滤非电影数据 9

4.2.5 缺失值处理 10

V 存储模型 10

5.1 逻辑存储模型 10

5.1.1 ERD 10

5.1.2 星型模型 11

5.2 物理存储模型 11

5.3 图数据库存储模型 12

VI 查询统计 12
6.1 条件组合查询 12
6.2 电影版本查询 13
6.3 用户评价查询 13
6.4 合作关系查询 13
VII 存储优化 13
7.1 MySQL 13
7.1.1 单表优化 14
7.1.2 多表优化 14
7.2 Hive 14
7.3 Neo4j 15
VIII 性能对比 15
8.1 MySQL 15
8.1.1 单个查询 16
8.1.2 组合查询 16
8.1.3 合作查询 16
8.2 Hive 17
8.3 Neo4j 18
8.3.1 优化前后对比 18
8.3.2 MySQL与Neo4j合作关系查询性能对比 18
IX 总结 19

I 项目简介

本项目的原始数据来源是Amazon Review Data（2018）的Movies and TV 5-core (3,410,019 reviews)数据集。我们从中提取出了有关评论的asin，并从亚马逊网站爬取了电影的详细信息。以此为基础，我们分别建立了关系型数据库，分布式文件型数据库和图数据库，建立电影查询系统，实现我们的查询业务，并且对不同数据库的性能进行了对比与分析，以此来探究各种存储方式在本项目中适合的查询业务。此外，我们根据本项目的特定业务，在存储和查询两个方面都做了优化，并对比了优化前后的查询效率。

II 实现功能

2.1 条件组合查询

1. 按照发行时间查询统计
2. 按照电影名称查询统计
3. 按照导演名查询统计
4. 按照主演名查询统计
5. 按照演员名查询统计
6. 按照编剧名查询统计
7. 按照电影类别查询统计
8. 按照电影评分查询统计
9. 按照上述条件组合查询统计

2.2 电影版本查询

1. 按照电影名称，查询同名电影与电影版本

2.3 用户评价查询

1. 查询不含负面评价的电影
2. 根据电影名称查询评价

2.4 人物关系查询

1. 查询演员与导演合作关系Top榜
2. 查询演员和演员合作关系Top榜
3. 查询演员和编剧合作关系Top榜
4. 查询导演和编剧合作关系Top榜
5. 查询指定演员合作导演Top榜
6. 查询指定演员合作演员Top榜
7. 查询指定演员合作编剧Top榜
8. 查询指定导演合作编剧Top榜

III 系统架构

3.1 数据库

3.1.1 MySQL

项目中我们使用MySQL作为关系型数据库，它也是当前使用率较高的一种关系型数据库，体积小且性能稳定。作为我们的主要数据库，MySQL完成了所有业务逻辑的查询。

3.1.1 Neo4j

我们使用Neo4j作为图数据库，Neo4j是一个高性能的，NOSQL数据库，它将结构化数据存储在网络上而不是表中，利用它我们完成了大部分导演，演员之间合作关系的查询业务。

3.1.1 Hive

Hive 作为一个分布式数据库，在大数据处理方面相对其他类型的数据库比较占优势。由于用户评论数据的数量级较大，这部分业务的查询基本都在 Hive 中完成。此外,部分电影的简单条件查询也可以在 Hive 中执行。

3.2 前后端技术

3.2.1 前端技术

技术/框架	说明
Node.Js	基于 Chrome V8 引擎的 JavaScript 运行环境
Webpack	前端资源模块化管理和打包工具
Vue	渐进式框架，用于构建用户界面
Element UI	基于 Vue 2.0 的桌面端组件库
EChart	基于 JavaScript 的可视化图表库
Axios	基于promise可用于浏览器和node.js 的网络请求库

3.2.2 后端技术

技术/框架	说明
Spring Boot	基础框架
Mybatis Plus	持久层框架，负责和数据库的交互
Lombok	注解生成
Fast JSON	前后端数据传输
Swagger	前后端接口文档
Maven	依赖管理

IV ETL

4.1 数据爬取

4.1.1 爬取过程

1、数据预处理:

在进行数据爬取前，需要对数据源做预处理，具体为，用python提取出所有评论对应的asin，并去重。

2、URL拼接和请求头构造

(1) 观察可知，爬取的网页网址有以下特征：

```
url="http://www.amazon.com/dp/"+asin
```

(2) 请求头构造

```
header = {
 "User-Agent": head_user_agent[random.randrange(0,
len(head_user_agent1))],
 "Connection": "closer",
 "Upgrade-Insecure-Requests": "1",
 "Cache-Control": "max-age=0",
 "Referer":"http://www.google.com"  }
```

上述代码中head_user_agent是存放了多条用户代理的列表，在爬取过程中随机取一个用户代理构成请求头，其目的在于防止网站反爬虫机制捕获到异常，封禁IP。

构造完成后，就可以使用requests库内方法访问网站，得到该网页全部信息，但上述构造仍然很容易被反爬机制捕获，因此我们使用了多种方法防止被封禁，具体措施将在第三部分详述。

3、并发优化爬取速度

利用多线程来加速爬取，使用的并发模块为concurrent.futures模块，设置多线程的个数为20个。

4、爬取对象

我们在爬取中不会直接对 html 页面进行解析拿到具体的电影信息，而是在本地存成 html，等待爬取结束后再对页面进行解析，获取电影信息。

4.1.2 反爬措施

1、反爬原理

使用单一IP，单一用户代理爬取时，前面的数据是正常的，但是后面的数据全是空，究其原因，其实是亚马逊网站监视到异常后，会返回一个验证码网页，对于这种情况有两种解决办法，一是直接识别验证码，填写并提交，二是绕过验证码。我们这里采用第二种方法，即搭建代理池，每次访问网站的IP不一样，会大大减少异常的出现。

2、使用代理池

基于https://github.com/Python3WebSpider/ProxyPool可以快速搭建一个代理池，在使用时只需下载docker，打开该项目所在目录运行命令docker-compose up，然后访问http://127.0.0.1:5555/random即可获取一个代理池内的随机IP。

3、其他措施

在使用代理池之后爬取，异常情况大大降低，但其数量还是不容小觑，为了提高效率，我们又采取了以下措施：

(1) 随机休息

由于固定的程序执行时间依然是网站识别爬虫的一种方法，为了模拟人访问网站的行为，我们在每次爬取完都会随机休眠0.2秒以内的时间，这样即保证了爬取速度不会太慢，也保证了爬取的成功性。

(2) 失败情况

为了解放双手，让程序一次成功地爬出所有页面，我们使用了两个循环：

① 只有成功的网页才存到本地，因此每轮爬取都将现有的网页数量与应有的网页数量比较，如果不相等则求出其差集继续下一轮爬取。

② 对于每个网页，如果爬取一次失败就将其略过的话，显然第一种循环次数就会增加，因此这里设置了每个网页有10次爬取的机会，如果一次成功则爬取下一个页面，如果不成功则选取其他随机IP，随机用户代理尝试第2次，第3次，直

到成功或10次机会用完。

这样可以一次将所有页面全部爬出，且不需要人工干预。

4.2 数据清洗

4.2.1 数据解析

我们对爬取到的网页做了所需信息的提取，包括asin，商品名，电影名，导演，运行时长，发行时间，上架时间，演员，主演，工作室，监制，编剧，标签，评分，imdb评分以及参与评分的人数，并将其导出为csv文件，注意到网页中还有某一商品对应着的多个版本（即多个asin对应同一个商品）的信息，我们还对提取到的信息运用并查集做了去重工作，合并了指向同一商品但asin不同的行，并将原先asin字段改为该商品关联的所有的asin字段。

4.2.2 信息规范化

由于标题信息的不标准携带"DVD"、“VHS”等旁支信息，以及网页格式影响提取内容会伴随“\\xa0”等符号，我们使用正则表达式对解析后的数据进行了规范化处理，比如删除标题中的伴随信息，将电影时长统一为分钟为单位，删除演员、导演等名字中携带的多余符号。

4.2.3 删除干扰信息

对于一些不合法的值，比如演员、导演等名字中的“Various”，“-”，“--”，进行了删除处理。

4.2.4 过滤非电影数据

在过滤非电影信息时，我们考虑到亚马逊网站标签的不规范性，所以对于亚马逊网站标签中明确存在“Movie”或“Film”，且不是“Film History”，我们认定为电影，其余的部分，我们查询电影名是否存在于IMDB的电影数据来判断。而考虑到同名

不同一的情况，我们进一步考虑导演信息，作为鉴别依据。

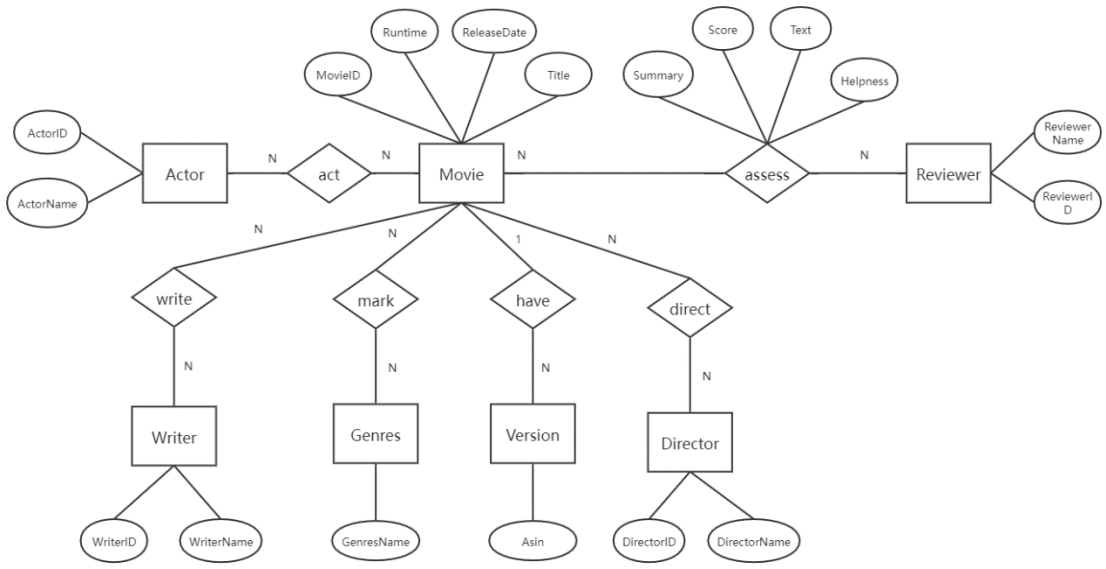
4.2.5 缺失值处理

因为数据中可能存在一些缺失值，所以我们需要对缺失值进行合理补充来让数据更适应于业务。比如针对电影发行时间的缺失，我们首先使用IMDB电影数据库的日期来填充，填充后仍存在缺失的数据，则采取使用最早的评论时间来进行第二层填充。

V 存储模型

5.1 逻辑存储模型

5.1.1 ERD

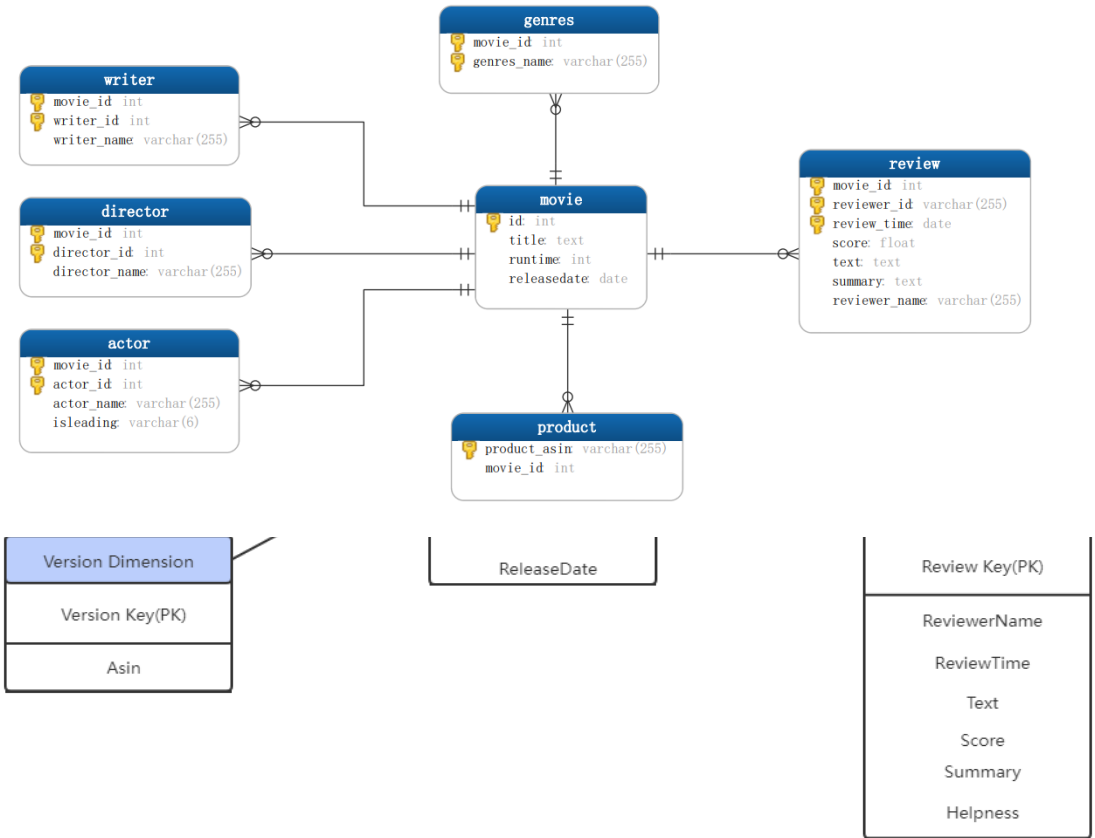


5.1.2 星型模型

从维度建模的角度分析，我们采用星形模型作为我们数据仓库的逻辑存储模型。相比于存储规范化数据的雪花存储模型，星形存储模型是一种反规范化的结构，多维数据集的每一个维度都直接与事实表相连接，不存在渐变维度，所以数据有一定的冗余。因为存在数据冗余，所以星型模型中很多统计查询不需要做外部的连接，因此一般情况下效率比雪花型模型要高。考虑到在我们的数据集中，电影数据的总量仅在1.8万左右，进行数据冗余完全在可接受的范围内，因此我们采用了更有效率的星形模型。除了性能上的考虑之外，星形模型加载维度表，不需要在维度之间添加附属模型，因此ETL的复杂度更低，可以实现高度的并行化。

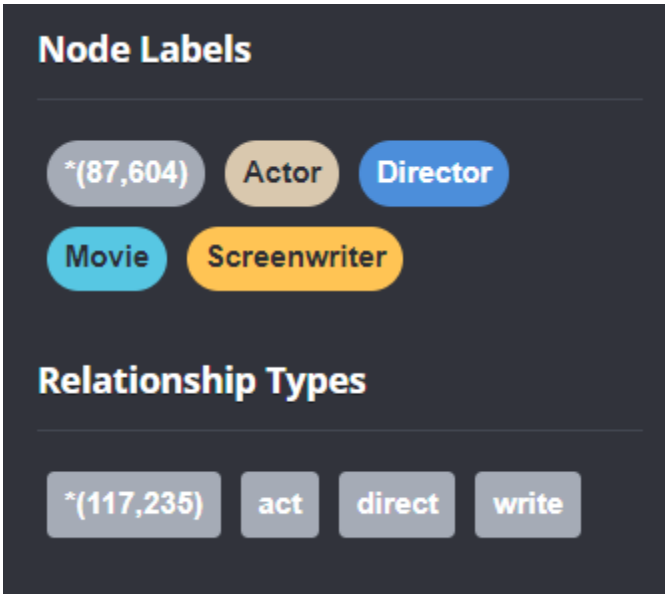
5.2 物理存储模型

基于星型逻辑存储模型，我们在MySQL和Hive中建立了如下物理存储模型。



5.3 图数据库存储模型

为了查询演员与导演，演员与演员，演员与编剧之间的合作关系，在图数据库中我们建立了Movie, Actor, Director, Scresswriter四种节点，并建立了三种关系：



act(演员参演电影), direct(导演执导电影), write(编剧编写电影剧本)。如果要查询演员和演员，演员和导演，演员和编剧的合作关系。就是看两个结点有没有共同参与同一部电影，共同参与了多少次。对于无起点查询，挑选出合作次数位于前20的组合即可。

VI 查询统计

6.1 条件组合查询

Mysql 数据库存储了完整的电影相关的数据，因此可以在 Mysql 数据库中进行电影条件组合查询。

支持的查询字段包括：电影名称、导演、演员、类别、评分、年、月、季度。

进行查询后，程序会把在数据库中的查询耗时以数值和图表的形式在前端界面展示，便于进行性能的对比分析。我们在 Mysql 数据库中分别以 Normalization（按照第三范式拆表）的方式和 Denormalization（不进行拆表）的方式存储了两

套电影数据，程序也会把在两种存储模型中的查询耗时在前端界面进行展示，可以对比两种方式在查询性能上的差异。

从用户使用的角度考虑，我们也把查询结果的电影详细数据以分页的方式进行展示，并显示查询结果总数。

6.2 电影版本查询

基于在 ETL 阶段完成的电影版本 Mapping 工作，我们把电影版本数据在 Mysql 数据库中进行了冗余存储，从而为该业务功能提供高的查询性能。用户输入电影名称，系统将显示该电影共有多少个发行版本。程序将显示该功能在 Mysql 数据库中的查询耗时，用于体现进行冗余存储带来的性能优势。

6.3 用户评价查询

支持三种用户评价查询：" 用户评分高于某个值的所有电影"、" 用户评价数最多的电影 " 和" 不含负面评价的电影 "。

针对于不含负面评价的电影的查询，我们定义当用户给出的评分小于三时，视作此评价为负面评价，进而实现了不含负面评价的电影查询。

6.4 合作关系查询

我们基于 Neo4j 中存储的电影数据，实现了合作关系查询功能。用户可以进行不指定起点的查询，例如查询合作 TOP 榜，也可以进行指定起点的查询，例如查询与某个演员合作次数最多的导演和演员有哪些。 同样我们基于Mysql也实现了上述查询，并将其查询时间进行对比，结果以图表的形式在前端展示。

Ⅶ 存储优化

7.1 MySQL

7.1.1 单表优化

1、数据字段

通过对数据字段的类型和内容进行优化，减少数据表中单行记录的存储大小，从而提高单表查询的速度。

（1） 尽量减少单表的字段数，去除不必要的字段。

（2） 避免使用NULL字段，避免NULL字段占用额外索引空间和引起引擎放弃索引采用全表扫描

2、索引

根据数据库索引的原理与结构，合理地为数据表建立索引，提高查询效率。

例如进行根据演员查询电影，查询主要依据是演员名，因此对演员冗余表中的演员名称字段建立了索引。

7.1.2 多表优化

1、多表Join

如果根据范式对数据表进行切分，需要慎重考虑对查询性能带来的影响。因为只要是进行切分，一部分查询必然会涉及到多表 join，多表 Join 的问题是不可避免的，并且多表 join 很可能带来严重的性能影响，但是良好的设计和切分却可以减少此类情况的发生。比如把涉及到多个表的查询分成两次查询，在第一次查询的结果集中找到关联电影的id，在根据这些id发起第二次请求得到关联数据。

7.2 Hive

Apache Hive 支持 Apache Hadoop 中使用的几种常见的文件格式，如 TextFile， RCFile，SequenceFile，AVRO，ORC 和 Parquet 格式。不同的存储格式在内存中有 不同的存储方式和性能特点。向 Hive 导入数据是默认采用 TextFile格式，该格式存储空间消耗较大，并且压缩的 text 无法分割和合并。因为可以直接存储，所以加载数 据的速度最高，但查询的效率最低。这显然不符合我们数据仓

库对查询性能要求高的 特点。 我们把存储格式更换为 ORCFile，以寻求更高的查询性能。ORC 文件是一种优化 的行列存储相结合的存储方式，数据按行分块，每块按照列存储，其中每个块都有一 个索引，数据压缩率非常高，同时能够获得很高的查询性能。

以年份和月份来查找电影信息这一查询来做对比。

存储格式	查询耗时
TextFile	21755ms
ORCFile	1589ms

7.3 Neo4j

对于图数据库来讲，其基于业务的存储逻辑是不便改变的，因此性能优化应从其他地方考虑，这里我们想到了加入索引提高查询速度。Neo4j数据库的索引一般分为三类：手动索引，自动索引，模式索引。前两种索引方式较为麻烦，并且目前的Neo4j版本已不支持。因此这里采用加入模式索引提高性能。 我们在Screenwriter，Actor，Director结点上都加入了索引。

Index Name	Type	Uniqueness	EntityType	LabelsOrTypes	Properties	State
index_2586c141	BTREE	NONUNIQUE	NODE	["Screenwriter"]	["name"]	ONLINE
index_343aff4e	LOOKUP	NONUNIQUE	NODE	[]	[]	ONLINE
index_ad47116a	BTREE	NONUNIQUE	NODE	["Actor"]	["name"]	ONLINE
index_c2eaf96f	BTREE	NONUNIQUE	NODE	["Director"]	["name"]	ONLINE
index_f7700477	LOOKUP	NONUNIQUE	RELATIONSHIP	[]	[]	ONLINE

除此之外，在实际查询中，在正式查询所需数据前，可以先查询所有数据，进行预热，这样可以触发图中所有的点和关系。整个图会被缓存起来。后续查询的速度将会非常快。

Ⅷ 性能对比

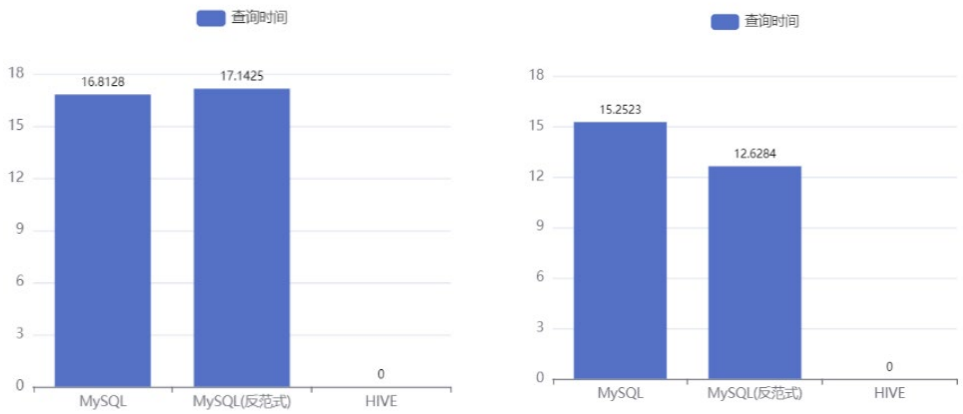
8.1 MySQL

8.1.1 单个查询

	遵循范式	反范式
标签查询 电影	60ms	59ms
电影查询 评论	54ms	18ms

在查询单个业务时，我们发现遵循范式的数据库和反范式的数据库性能相差不大。

8.1.2 组合查询



在组合条件查询中，左图是条件为一时的对比，右图是条件为三时的对比，可以发现在条件数增多时，反范式数据库相对遵循范式数据库性能好很多。

8.1.3 合作查询

	遵循范式	反范式
有起点	20ms	13ms
无起点	8536ms	2402ms

在查询实体关系的业务中，由于反范式需要做的join相对于符合3NF要少，可以发现反范式的数据库速度要快很多。

此外，当不同的查询语句对于无起点的实体关系查询性能的影响也非常显著。我们发现，如果先对应名字，再判断是否合作，通过电影id连接再通过双方名字分组，对比先判断合作，通过电影id连接再通过双方id分组，再对应名字要快很多，我们认为主要是varchar相对于int在查询性能中表现较差。

	先对应名字	先判断合作
无起点	8536ms	858ms

8.2 Hive

查询条件	存储格式	查询时间	结果数目
根据年月查询电影	TextFile	21.755s	128
根据年月查询电影	ORCFile	1.589s	128
根据演员查询电影	TextFile	167.104s	10
根据演员查询电影	ORCFile	137.104s	10

通过以上结果可见，Hive 进行相同的条件组合查询的性能明显低于 Mysql 关系型数据库。

首先，通过查看执行计划，可以看到在 Hive 中进行查询时，但凡涉及到 group by、聚合函数、多表 join 等操作，都会调用 Map Reduce 分布式计算框架进行计算。其次，进一步分析其原因，Hive 本身的特点是在大型数据集上会表现出优越的性能，考虑到我们的项目数据集中，最多的数据集是 700 多万条的用户评论数据，而基本功能的实现都是操作在数据量仅有 25 万余条的电影数据，我们猜测是数据量限制了 Hive 体现其性能优越性。

第三，可以对Hive的存储格式进一步探究，发现在TextFile和ORCFile存储格式下Hive的查询性能差异极大，ORCFile不但数据压缩效果较好，文件存储大小明显更小，而且查询性能更优，甚至查询耗时已经不在一个数量级。

8.3 Neo4j

8.3.1 优化前后对比

预热对查询性能的影响

是否预热	是	否
查询时间	877ms	1349ms

索引对查询性能的影响（预热后）

	有索引	无索引
有起点	6ms	52ms
无起点	741ms	877ms

针对以上数据可以看出，两种优化措施的性能优化体现在以下方面：

对于添加索引的性能优化措施，其对是否有起点的查询优化效果较为明显，有起点的查询在增加索引后查询速度提升非常快，对于无起点的查询优化效果不大。

对于预热的性能优化措施，其针对两种查询角度，性能优化都比较明显。

因此，进行有起点查询时，使用索引是比较好的方法，其次，对于任何查询，都可以在正式查询前进行预热，提高性能效果较好。

8.3.2 MySQL与Neo4j合作关系查询性能对比

	MySQL	MySQL（反范式）	Neo4j
第一次	20.6412ms	8.6129ms	676.7394ms
第二次	10067.7199ms	10047.424ms	2720.6706ms

第三次	37.8242ms	38.3078ms	75.4129ms

针对以上查询，可以看出无起点查询时，Neo4j的性能明显好于关系型数据库，因为图数据库找到节点后不需做任何join即可以直接找到与该节点有关系的其他节点，而不论是否反范式的MySQL存储模型或多或少都需要做join操作，浪费了时间。因此，无起点查询时，Neo4j性能更好。

而有起点查询时，我们发现了一个奇怪的现象，第一次是将后端与MySQL数据库部署到一台机器上运行，查询前已进行过预热，第二次是将后端与Neo4j数据库部署到一台机器上运行，并且查询前未进行过预热，第三次是在第二次的基础上进行预热过后的结果，可以看出，后端访问同一台机器上的数据库时间是明显要比访问另一台机器上的数据库的时间短的，第一次Neo4j时间较长的原因可能是后端访问不在同一台机器上的数据库连接时间较长，第二次三种时间都很大的原因可能是首次连接数据库花费了较多时间，图3可能是MySQL底层优化使得对于查询做了缓存，为了验证这一猜想，我们使用同一条件组合查询多次，发现MySQL数据库越到后面的查询就会越快，于是猜测或许是MySQL的底层优化对查询做了缓存。

从数据库的角度来说，有起点查询时间Neo4j和MySQL的查询速度是不相上下的，都是个位数毫秒级别。因此有起点查询两个数据库性能都比较好。

IX 总结

通过实现电影数据获取、存储、查询相关的性能实践，我们分析了各类数据库的存储特点和查询性能。在此过程中，我们以上课所学为依托，同时查询相关资料，尝试了星型模型、雪花模型等各类存储模型，学习了逆规范化方法在OLAP业务中的有效性和重要性，并分析了各种模型、存储结构、优化方法的优势与缺陷。

三种数据库在查询时存在一些相同的特点，例如，数据库中执行时间的变化趋势都是先较多然后减少，我们认为主要是由于MySQL底层的缓存机制，Neo4j的预

热，此外，jdbc在首次连接时需要较多时间进行网络通信，当一次连接建立后，在此基础上执行后续事务才是真实的操作时间。

三种数据库的存储模型分别具有各自的存储特点，适用于不同的查询处理。例如，对于查询实体属性关系型数据库性能较好，而对于查询实体间关系的业务，关系型数据库可能需要进行多表连接查询，这会消耗较多时间，图数据库Neo4j则在这类查询上能获得比较优秀的表现。而Hive数据库适合数据量百万级别时的查询业务，因为项目数据量限制，未通过本项目证实Hive的优越性。

除此之外，对于各个数据库，我们都了解了各种从存储以及查询语句写法层面优化数据库性能。例如建立索引、避免多表Join等。

总的来说，通过此项目，我们提高了对于三种数据库的认识。