

# 数据仓库文档

2050865 黄彦铭

2052727 吕天成

2052334 杨严

2051196 刘一飞

## 目录

数据仓库文档 .....	1
项目要求: .....	3
1. 数据爬取 .....	3
1.1 原始数据获取 .....	3
1.2 数据预处理 .....	4
1.3 数据爬取 .....	5
2. ETL .....	15
2.1 电影合并 .....	15
2.2 数据清洗 .....	16
2.3 人名合并 .....	18
2.4 筛除 TV 数据 .....	18
3. 存储模型 .....	18
3.1 关系型数据库 .....	18
3.1.1 逻辑模型 .....	18
3.1.2 物理模型 .....	21
3.2 分布式文件型数据库 .....	21
3.3 图数据库 .....	22
4. 优化 .....	24
4.1 关系型数据库优化 .....	24
4.1.1 建立索引 .....	24
4.2 分布式数据库优化 .....	27
4.3 图数据库优化 .....	30
4.3.1 建立索引 .....	30

4.3.2 将电影类型与演员之间建立联系 .....	31
5. 数据治理 .....	31
5.1 数据质量 .....	31
5.1.1 数据完整性 .....	32
5.1.2 数据准确性 .....	32
5.1.3 数据一致性 .....	32
5.2 数据血缘 .....	33
5.2.1 非电影数据 .....	33
5.2.2 电影数据 .....	34
.....	35
6. 查询比较 .....	36
6.1 特定查询应当选择适合的存储模型 .....	36
6.1.1 关系型数据库 .....	36
6.1.2 分布式文件型数据库 .....	36
6.1.3 图数据库 .....	37
6.2 测试用例及对比 .....	37
6.2.1 简单查询 .....	37
6.2.2 总量查询（Action 分类一共有多少电影） .....	39
.....	41
6.2.3 关系查询（经常合作的导演和演员） .....	41
7. 前端展示页面 .....	43

## 项目要求：

数据来源：<http://snap.stanford.edu/data/web-Movies.html> (Links to an external site.)

- ETL 要求：

- 1) 获取用户评价数据中的 7,911,684 个用户评价
- 2) 从 Amazon 网站中利用网页中所说的方法利用爬虫获取 253,059 个 Product 信息页面
- 3) 挑选其中的电影页面，通过 ETL 从数据中获取

- 电影 ID，评论用户 ID，评论用户 ProfileName，评论用户评价 Helpfulness，评论用户 Score，评论时间 Time，评论结论 Summary，评论结论 Text，电影上映时间，电影风格，电影导演，电影主演，电影演员，电影版本等信息

4) 在网页中不同网页可能是相同的电影（如同一部电影的蓝光、DVD 版本，同一部电影的不同语言的版本等），通过 ETL 对相同的电影（需要给出你所认为的相同的定义）进行合并

5) 在网页中电影演员、电影导演、电影主演等会出现同一个人但有不同的名字的情况（如 middle name，名字缩写等），通过 ETL 对相同的人名进行合并

4) 在网页中部分电影没有上映时间，可以通过第三方数据源（如 IMDB、豆瓣等）或者从评论时间来获取

5) 通过 ETL 工具存储 Amazon 页面和最终合并后的电影之间的数据血缘关系，即可以知道某个电影的某个信息是从哪些网站或者数据源获取的，在合并的过程中最终我们采用的信息是从哪里来的。

- 可以参考的工具：

1) Pentaho Data Integration: <https://sourceforge.net/projects/pentaho/> (Links to an external site.)

2) Web 爬虫: <https://scrapy.org>

## 1. 数据爬取

### 1.1 原始数据获取

数据来源：<http://snap.stanford.edu/data/web-Movies.html>

该数据集主要由来自 Amazon 的电影评论组成，包括了 1997 年 8 月至 2012 年 10 月之间的

7911684 条评论。

Dataset statistics	
Number of reviews	7,911,684
Number of users	889,176
Number of products	253,059
Users with > 50 reviews	16,341
Median no. of words per review	101
Timespan	Aug 1997 - Oct 2012

对网页中给出的压缩包下载后进行解压，得到了完整的数据集。其中数据集中的一条评论由如下结构组成：

```
product/productId: B00006HAXW
review/userId: A1RSDE90N6RSZF
review/profileName: Joseph M. Kotow
review/helpfulness: 9/9
review/score: 5.0
review/time: 1042502400
review/summary: Pittsburgh - Home of the OLDIES
review/text: I have all of the doo wop DVD's and this one is as good or better than the
1st ones. Remember once these performers are gone, we'll never get to see them again.
Rhino did an excellent job and if you like or love doo wop and Rock n Roll you'll LOVE
this DVD !!
```

- product/productId: Amazon 产品的 ASIN，可以通过 ASIN 直接获得商品详情页的 url：  
amazon.com/dp/+asin
- review/userId: 评论的用户 ID
- review/profileName: 评论的用户昵称
- review/helpfulness: 评论是否有帮助
- review/score: 评论的评分
- review/time: 评论的时间
- review/summary: 评论的总结
- review/text: 评论的内容

## 1.2 数据预处理

1. 将上一个步骤中 txt 文件中的评论信息进行结构化的读取，保存到数据库中

index	productId	userId	profileName	helpfulness	score	time	summary	text
0	B003AI2VGA	A141HP4LYPWM5R	Brian E. Erland "Rainbow 5	7/7	3.0	1182729600	"There Is So Much Darkne	Synopsis: On the daily tre
1	B003AI2VGA	A328S9RN3U5M68	Grady Harp	4/4	3.0	1181952000	Worthwhile and Importan	THE VIRGIN OF JUAREZ is
2	B003AI2VGA	A117QGUDP043DG	Chrissy K. McVay "Writer"	8/10	5.0	1164844800	This movie needed to be	The scenes in this film can
3	B003AI2VGA	A1M5405JH9THP9	golgotha.gov	1/1	3.0	1197158400	distantly based on a real	t THE VIRGIN OF JUAREZ (2
4	B003AI2VGA	ATXL536YX71TR	KerrLines "&#34;Movies,I	1/1	3.0	1188345600	"What's going on down ir	Informationally, this SHO
5	B003AI2VGA	A3QYDL5CDNYN66	abra "a devoted reader"	0/0	2.0	1229040000	Pretty pointless fictionaliz	The murders in Juarez are
6	B003AI2VGA	AQJVNDW6YZFQS	Charles R. Williams	3/11	1.0	1164153600	This is junk, stay away	Mexican men are macho r
7	B00006HAXW	AD4CDZK7D31XP	Anthony Accordin	64/65	5.0	1060473600	A Rock N Roll History Les	Over the past few years, p
8	B00006HAXW	A3Q4S5DFVPB70D	Joseph P. Aiello	26/26	5.0	1041292800	A MUST-HAVE video if y	I recvd this video (DVD ve
9	B00006HAXW	A2P7UB02HAVEPB	"bruce_from_la"	24/24	5.0	1061164800	If You Like DooWop You	(Wow! When I saw this shc
10	B00006HAXW	A2TX99AZKDK0V7	Henrique Peirano	22/23	4.0	1039564800	I expected more.	I have the Doo Wop 50 an

2. 获取评论信息中所包含的产品信息，使用 kettle 完成，流程如下

- 表输入，获取所有评论信息
- 字段选择，仅保留 productId 列
- 按照 productId 排序记录
- 去除重复记录
- 表输出，输出去重后的所有产品 ID 信息



## 1.3 数据爬取

常见的数据爬取有两种策略：

- 爬虫框架爬取：requests 或 scrapy 等
- 模拟浏览器点击爬取：webdriver+selenium 库

在这个项目的流程中，共涉及到了三次爬虫，我们根据所爬取网站的反扒的不同措施针对每一次爬虫选择了不同的爬取策略。

### 1.3.1 Amazon 数据爬取

在经过上述的数据预处理后，我们获得了评论中所包含的产品 asin 信息，需要到 Amazon 网站中进入商品详情页获取详细的产品数据信息。

#### 1. 目标 url

可以通过 ASIN 拼接得到 Amazon 商品详情页的 url

- amazon.com/dp/+asin

#### 2. 爬虫策略

Amazon 网站的反扒机制比较严格，因此在此轮爬虫中，选择了使用 webdriver 搭配 selenium 库模拟浏览器点击进行爬虫，从而大大降低了 IP 被封的概率。

与此同时，为了同时保证爬取的速度，我们使用多线程的方式并行进行爬虫，代码如下

```
# 开启多线程加速爬虫
# todo 修改线程数
thread_num = 2

thread_list = []
for i in range(thread_num):
    thread = threading.Thread(target=crawl, args=(i, d))
    thread.start()
    thread_list.append(thread)

for t in thread_list:
    t.join()
```

### 3. 爬虫过程中的验证码验证

针对于 Amazon 网站中的验证码问题,我们使用了 Amazon Captcha 库中已经训练好的网络模型,将验证码的图片传入并返回字符串,此后使用 `webdriver` 模拟输入和点击,完成验证码验证过程,代码如下:

```
verification = driver.find_elements_by_xpath('//*[@class="a-row a-text-center"]//img')
if len(verification) > 0:
    pic_url = verification[0].get_attribute('src')
    captcha = AmazonCaptcha.fromlink(pic_url)
    code = captcha.solve(keep_logs=True)
    print(code)
    input_element = driver.find_element_by_id("captchacharacters")
    input_element.send_keys(code)
    button = driver.find_element_by_xpath("//button")
    button.click()
    continue
```

#### 4. 404 页面记录

对于其中不存在的页面，即网页中显示 404 小狗图像的产品，在代码中进行了特殊判断，并且将 404 数据记录到 csv 文件中，代码如下：

```
dog_img = driver.find_elements_by_xpath('//a[@href="/dogsofamazon"]')
if len(dog_img) > 0:
    # 404 网页
    print("id={},404".format(productId))
    # todo 404 网页记录
    writer.writerow(['\t' + productId, 404])
    csvfile.flush()
```

## 5. 网络延迟重爬数据

对于网络延迟的情况，通过识别弹窗来识别到这种情况，采取重爬的方式。如果重爬 3 次依然不能获取数据，才放弃这条数据。

```
break
# 是否需要刷新页面
alert = driver.find_elements_by_xpath('//div[@class="a-alert-content"]')
if len(alert) > 0:
    reload = False
    for a in alert:
        if "reload" in a.text:
            reload = True
            break
    if retry > 0 and reload:
        continue
```

## 6. 数据概览

7. 在该轮爬虫过程中，我们所获得的数据字段为：

- productId: 产品 asin
- productTitle: 产品名称
- category: 产品类型
- date: 上架日期
- genre: 体裁
- score: 评分
- score\_distribution: 各评分占比
- director: 导演
- star: 主演
- actor: 演员
- commentNum: 评论总数
- producer: 制作者



- studio: 发行商
- runtime: 运行时长
- rating: 电影评级
- language: 语言
- relatedProduct: 相关同类型产品

考虑到爬虫所耗费的时间代价, 因此在一次爬取中最大程度的获取到了产品的各类信息, 虽然上述某些提到的字段在该项目的后续过程中没有用到, 但是为后续需要其他维度的数据分析提供了数据支撑。

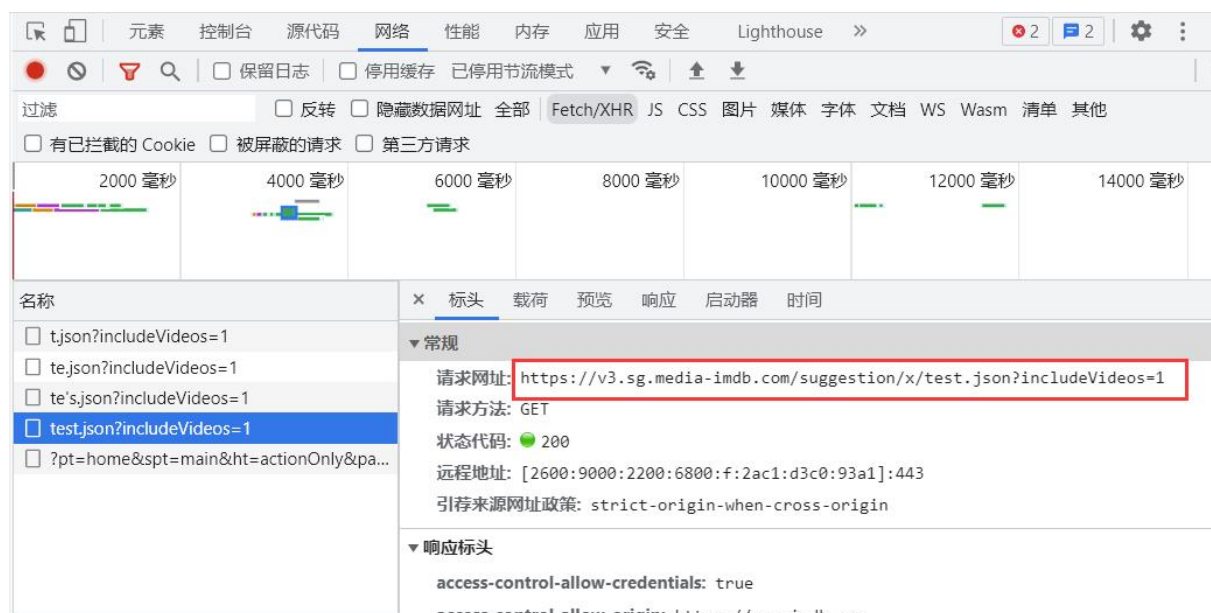
完成此轮爬虫后, 我们获得了近 12 万 7 千条 Amazon 数据。

### 1.3.2 IMDB 数据第一次爬取

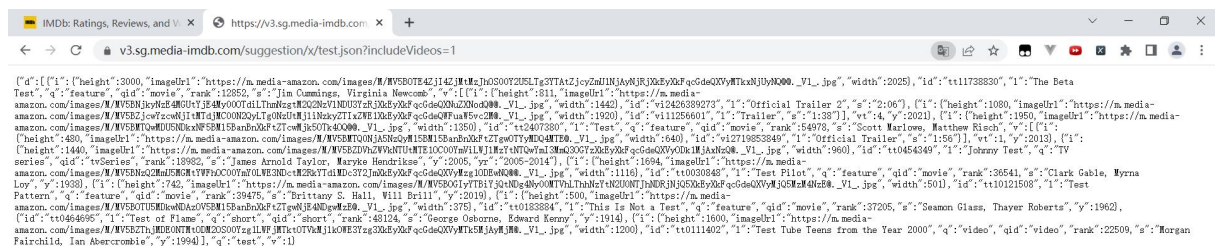
在完成了对 Amazon 的爬虫后, 鉴于数据信息存在大量缺值或值不准确的情况, 我们选择了 imdb 作为第二个数据信息源, 以提升整体数据质量。

#### 1. 目标 url

由于我们没有 IMDB 的电影 ID, 而是只有电影名称, 因此需要利用搜索框进行查找。在使用 IMDB 网站的搜索框时, 发现了其向后端请求的 url



当我们在搜索框中输入 test 时, IMDB 向该 url 发起了请求, 当我们访问这个请求, 可以看到它返回了通过关键词 test 搜索到的电影信息, 并且以 json 的格式返回。而这个数据对应了搜索栏中的下拉框内容。



因此我们可以根据电影的名称拼接处需要爬取的目标 url

## 2. 爬取策略

### 3. 切换 wifi 解除 ip 封禁

```
# 切换 wifi（在 ip 被封之后切换 wifi 便能够重新爬取）
def auto_switch_wifi():
    print("正在切换 wifi")
    cmd = 'netsh wlan connect name="%s" % "TJ-wifi"'
    res = os.system(cmd)
    print("切换完毕")
    return 'ok' if res == 0 else 'failed'

if response.content == b'Too many network requests. Try again in a few seconds.':
    print("IP 被封了，休眠十分钟")
    # 切换 wifi
    auto_switch_wifi()
    continue
```

#### 4. 添加随机 User-Agent

为了使爬虫的行为更难以被发现，在每次请求时，都使用了随机的 user-agent，以模拟不同浏览器访问，相关代码如下：

```
url =  
'https://v3.sg.media-imdb.com/suggestion/titles/x/{}.json?includeVideos=1'.format(name)  
params = {  
    'type': '5',  
    'interval_id': '100:90',  
    'action': '',  
    'start': '0',  
    'limit': '20'  
}  
header = {  
    'User-Agent': random.choice(USER_AGENTS_LIST)  
}
```

## 5. 名称匹配策略

在接口返回的数据中, 我们通过比较名称和主演来确定是否 imdb 的数据和 Amazon 的数据是匹配的, 匹配原则为名称和主演要同时满足包含或被包含的关系, 相关代码如下:

```

# 检查名称是否相同
def name_check(right, wait):
    right = name_process(right)
    wait = name_process(wait)
    if (wait == right) or (right in wait) or (wait in right):
        return True
    else:
        return False

# 检查主演是否相同
def stars_check(right, wait):
    if right is None:
        return True
    right_ = right.lower().split(',')
    right_list = []
    for i in right_:
        right_list.append(i.strip())
    wait_ = wait.lower().split(',')
    wait_list = []
    for i in wait_:
        wait_list.append(i.strip())

    for i in wait_list:
        if i in right_list:
            return True
    return False

```

## 6. 别名查询

由于某些电影存在别名，如果用主名称没有查询到，则会尝试继续使用电影的所有别名进行递归搜索，直到搜索到匹配项或所有别名已经搜索结束。

## 7. 数据概览

在该轮爬虫过程中，我们所获得的数据字段为：

- movie\_id: 电影的 ID
- imdb\_id: imdb 网站的电影 ID
- name: 电影名称

- style: 电影风格
- type: 电影类型
- star: 电影主演
- release\_data: 电影上映日期

由于所访问的接口为 imdb 网站搜索栏的数据，因此所获得数据有限，一些其他的核心数据没有包含在此接口的返回值中，但是获得了最为重要的 imdb 电影 id 信息。

完成此轮爬虫后，我们获得了近 6 万 3 千条和 Amazon 产品匹配的 imdb 电影信息。

### 1.3.3 IMDB 数据第二次爬取

在完成了 imdb 第一轮爬虫后，我们发现仍然缺少关键信息，因此不能仅仅使用搜索栏的数据，而是仍然要访问 imdb 的电影详情页来进行详细信息的获取。

#### 1. 目标 url

imdb 的电影详情页 url 也可以通过 imdb 电影 id 拼接来得到

- `imdb.com/title/ + movie_id`

#### 2. 爬取策略

对于 imdb 电影详情页的爬取，我们仍然使用了 request 搭配多线程的方式，并且一定程度上降低了线程数，从而使得在整个爬虫过程中都没有出现 ip 被封的情况。

#### 3. 数据概览

在该轮爬虫过程中，我们所获得的数据字段为：

- movie\_id: 电影 ID
- imdb\_id: imdb 网站的电影 ID
- name: 电影名称
- director: 导演
- writers: 编剧
- star: 主演
- actor: 演员
- genre: 体裁
- release\_date: 上映日期
- country\_of\_origins: 原产国
- languages: 语言
- production\_companies: 制作公司
- run\_time: 时长

- rating: IMDB 评分

和对于 Amazon 的爬虫相同，在爬取 imdb 详情页信息时，我们同样尽可能获取了所有可能的数据信息，虽然在后续的过程中某些属性信息没有被用到，但是能够为拓展其他维度的数据分析提供了数据基础

完成此轮爬虫后，我们同样获得了近 6 万 3 千条和 Amazon 产品匹配的 imdb 电影详情信息。

## 2. ETL

对于 ETL 的过程，我们采用了 pentaho 和 python 来进行处理。在经过 ETL 之后，我们一共得到了 163250 条电影。



### 2.1 电影合并

在爬取 amazon 电影数据的时候，把同一部电影的其他 format，edition 的 productId 都爬下来了。每个产品都具有其他格式、版本的产品信息，通过这些关联信息，可以在产品间建立起并查集。通过这种并查集的方式，可以给所有产品分组，同一组的产品就认为是相同的电影。通过 python 脚本对所有产品打上分组的标签。

打上分组标签以后，相同标签的产品要进行合并。合并时，要关注以下两点：

- 每个字段的合并方式不同。若是采用字符串拼接的方式，在拼接后需要进行去重。
  - 电影名称：选用第一个非空值
  - 上映日期：选用最早的时间
  - 人名信息：采用字符串连接
  - 评论数：选用第一个非空值
  - 产品溯源信息：采用字符串连接
  - 电影类型：采用字符串连接

5	genre2	genre	使用, 连接同组字符串
6	directors2	directors	使用, 连接同组字符串
7	starring2	starring	使用, 连接同组字符串
8	studio2	studio	使用, 连接同组字符串

- 记录数据血缘。每条合并后的数据，都应该能查询到它是由哪些产品合并而来的。

明确了以上两点以后，在 pentaho 中进行以下过程。将 python 打好标签的所有产品输入，按照标签号排序后进行合并，选择需要的字段输出。输出时，将分组标签保留，作为每个电影的主码。



## 2.2 数据清洗

在 amazon 爬下来的数据中，许多字段出现空值、乱字符串的情况。对于出现乱字符串的情况，我们利用 python 脚本将每一个字段的非法值进行了清理。

第一，在一些纯数字的字段上，可能爬取数据的时候会爬下来一些非数字的信息。例如，电影评分 score 字段，在爬取下来的原始数据中会出现一些英文字母。对于这些错误字段，直接置为空，以免影响这些数字的字段参与后续的数值运算。

第二，一部电影的主演应当是参演的子集。而在 amazon 爬下来的数据中，并不都满足这个约束。所以在数据清洗时，用主演的数据来补充参演的数据。

第三，在人名字段中，容易混杂着一些莫名其妙的信息。

- 对于一些未知的人名，网页中有时会用 N/A, x, Various 等字符串进行占位。
- 有些人名在网页中的分隔符不是逗号，而是&或/。
- 有些人名甚至还夹杂着一些网页中的其他 DOM 元素，例如点击展开按钮的"more..."。

由于人名是按照字符串拼接进行存储的，所以不能将整个字段置空，应当将每个字段存储的所有人名进行筛选，过滤掉不合规范的人名。在 python 中处理的示例代码如下：



```

# step1: 按,分割
directors = raw_directors.split(',')

# step2: 去除每个人的首尾空格
for i in range(len(directors)):
    directors[i] = directors[i].strip(' ')

# step3: 删除列表中的, n/a N.A. --- * =
directors = list(filter(lambda x: x != 'n/a' and
                        x != 'n/A' and
                        x != 'N/a' and
                        x != 'N/A' and
                        x != 'nan' and
                        x != ';' and
                        x != " " and
                        x != '---' and
                        x != '=' and
                        x != '*' and
                        x != 'x', directors))

# step4: 按 & / 分割
directors = listOfStr_split_by_char(directors, '&')
directors = listOfStr_split_by_char(directors, '/')

# step5: 去除每个人的首尾空格
for i in range(len(directors)):
    directors[i] = directors[i].strip(' ')

# step6: 删去人名后面的括号
# step7: 删去字符串中的 more...子串
for i in range(len(directors)):
    directors[i] = directors[i].split('(')[0]
    directors[i] = directors[i].replace('more...', '')

# step8: 删去 various Various
directors = list(filter(lambda x: x != 'various' and
                        x != 'Various', directors))

```

## 2.3 人名合并

经过排查爬下来的数据，发现同一个人的不同名字记录，基本上都是人名缩写。而这里的人名缩写又没有那么复杂，可能有的名字表示为两个单词，而有的名字表现为一个单词。所以只要判断一个名字是否为另一个名字的子串就行了，如果是，则只保留长串。通过这种方法，对每一部电影内部的人名进行合并。示例代码如下：

```
for i in range(len(directors)):
    for j in range(len(directors)):
        if directors[i] in directors[j] and i != j:
            directors[i] = "" # 标记为空串

directors = list(filter(lambda x: x != "", directors))
```

## 2.4 筛除 TV 数据

由于在 amazon 的商品分类中，Movie 和 TV 是被分到同一类的，所以在 amazon 中判断一个商品是 TV 还是 Movie 就非常困难。为了筛除 TV 数据，我们利用了 imdb 爬下来的数据。由于 imdb 的分类更为专业和准确，所以如果一部电影在 imdb 中被分类为 TV 或是其他非 Movie 的类型，我们就直接将这条商品筛除掉。

# 3. 存储模型

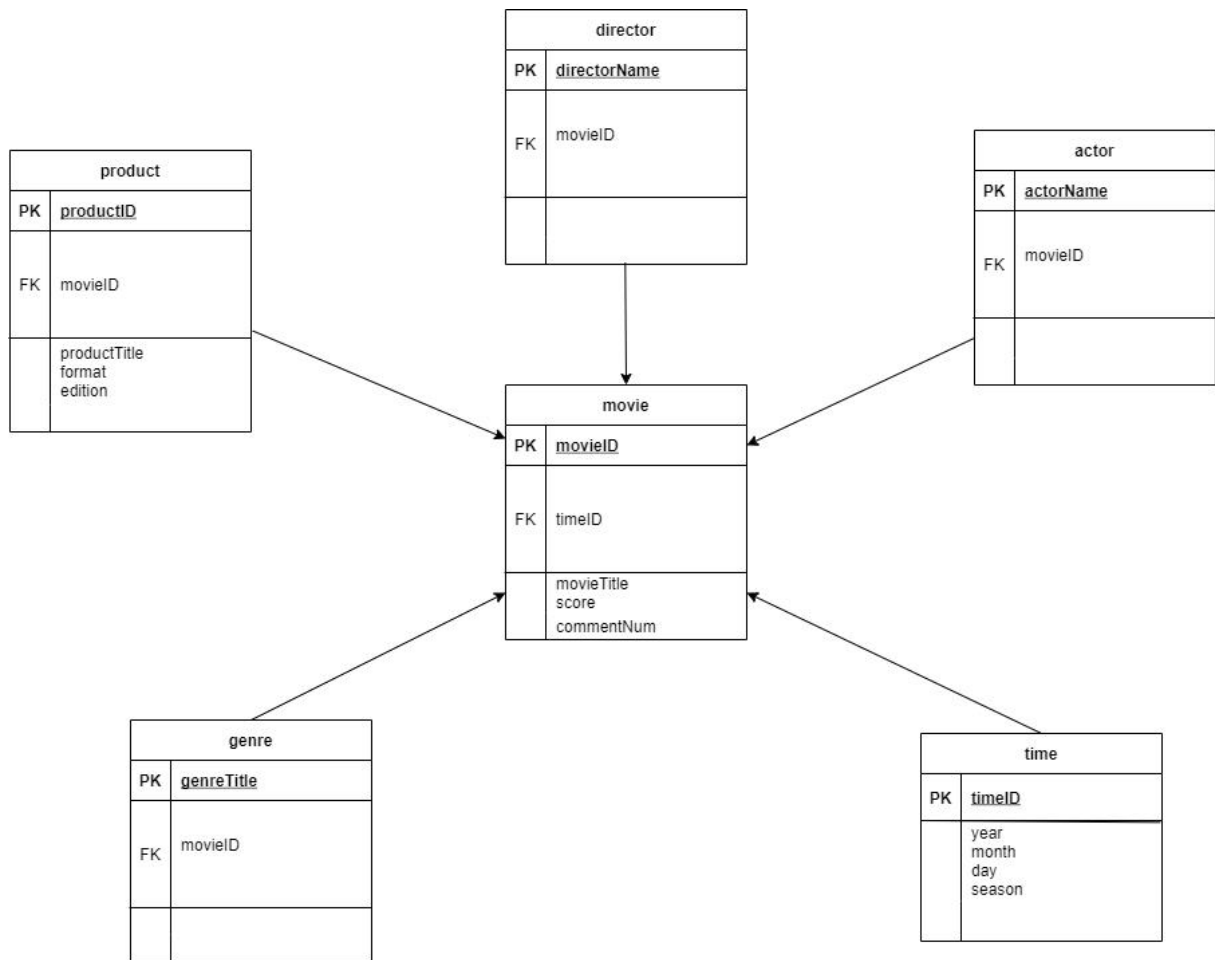
## 3.1 关系型数据库

### 3.1.1 逻辑模型

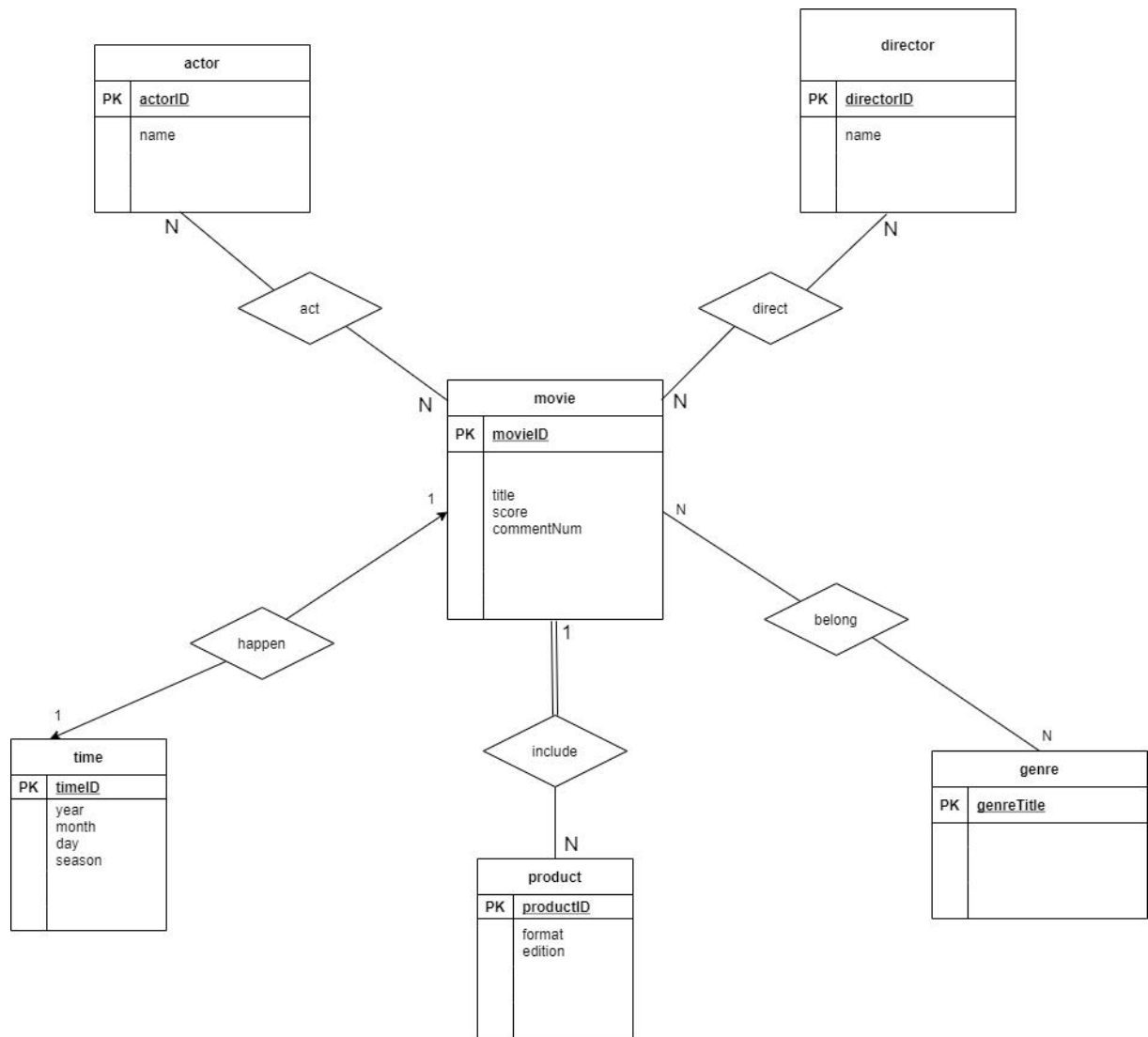
由于在实际应用场景中，电影信息面临着大量查询需求。若只用一张大表存储电影的所有信息，则面临大量查询时会产生过多的 I/O 操作，效率较低。所以我们采用冗余的存储，选用星型模式(Star Schema)来进行关系型数据库的组织。

在星型模式存储中，电影的 ID、名称等基本信息被存放在维度表中，大量的产品、导演、演员、类型、时间信息被存放在各个事实表中，维度表和事实表通过外键进行联系。

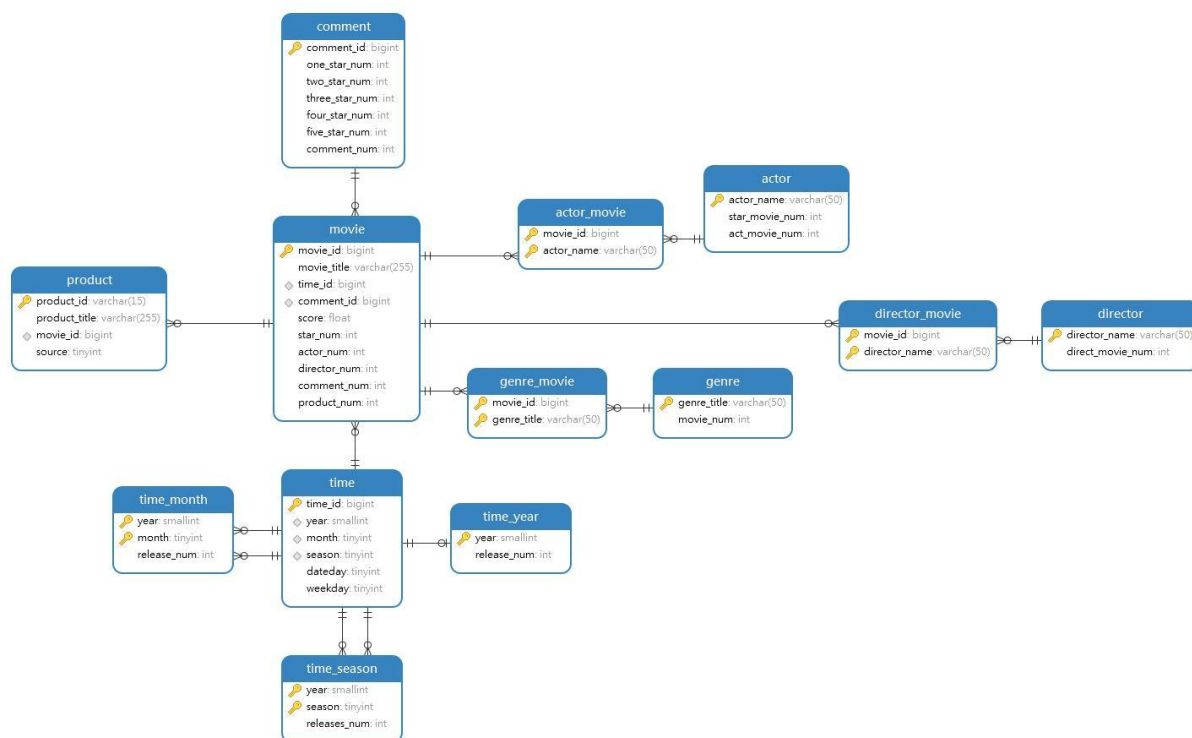
**星型模型：**



E-R图:



### 3.1.2 物理模型



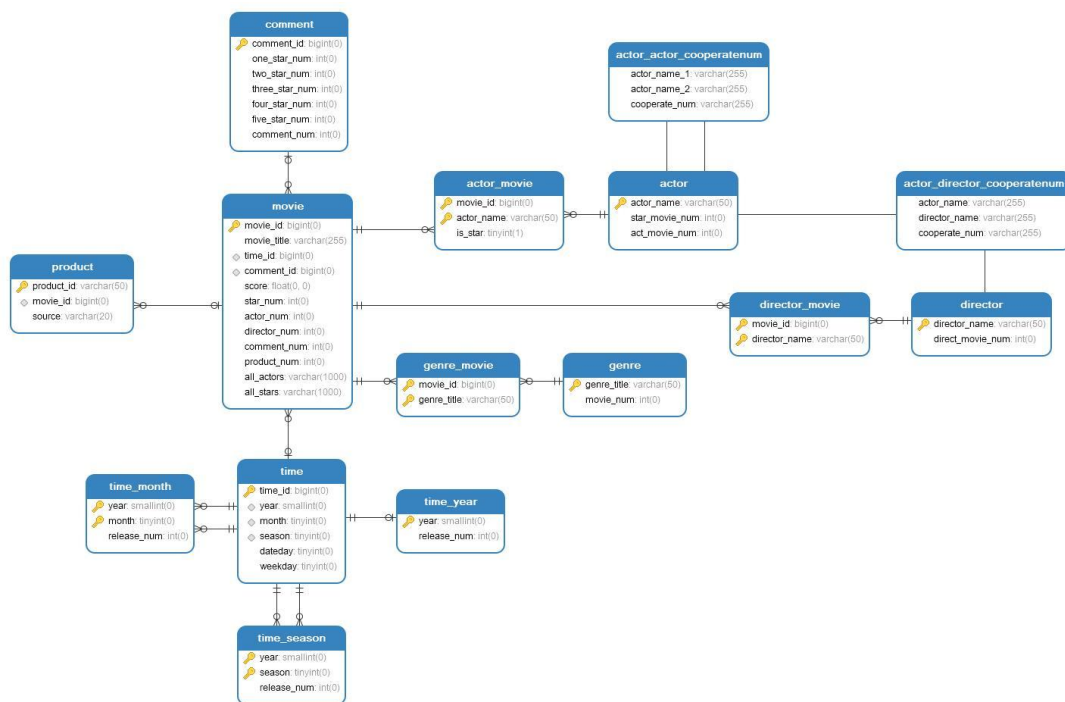
物理模型如上。因为一个电影存在多个演员和导演，因此这里颠倒了存储方式，在演员表和维度表中将电影 id 作为主键，这一冗余信息仅存在于列数相对较少的维度表中，同时根据演员、导演查询电影的效率也会大大提高。

同时，表中存储了一些元数据，主要用于记录某些总量的数据。这些元数据，避免了许多 join 操作才能查询出来的总量。例如，查询导演执导过的电影总数，如果不存储元数据，就需要 movie 和 director 表做连接再做 COUNT(\*)；但是如果存储了元数据，就转变为了对单表一个属性的查询，大大提高了总量信息的查询效率。

### 3.2 分布式文件型数据库

本项目采用了 Hadoop 这一分布式文件系统进行分布式数据库的存储，并使用 Spark 进行操作，我们建立了一个 NameNode 以及两个 DataNode，实现了数据的分布式存储。

分布式数据库的优越之处在于其能通过可用的计算机集群分配数据，完成存储和计算任务，同时可以在各个节点之间进行动态的移动数据，保证各个节点的动态平衡，多个副本存储也使其具有高容错性。根据存储和查询需求，我们建立存储模型如下图所示。



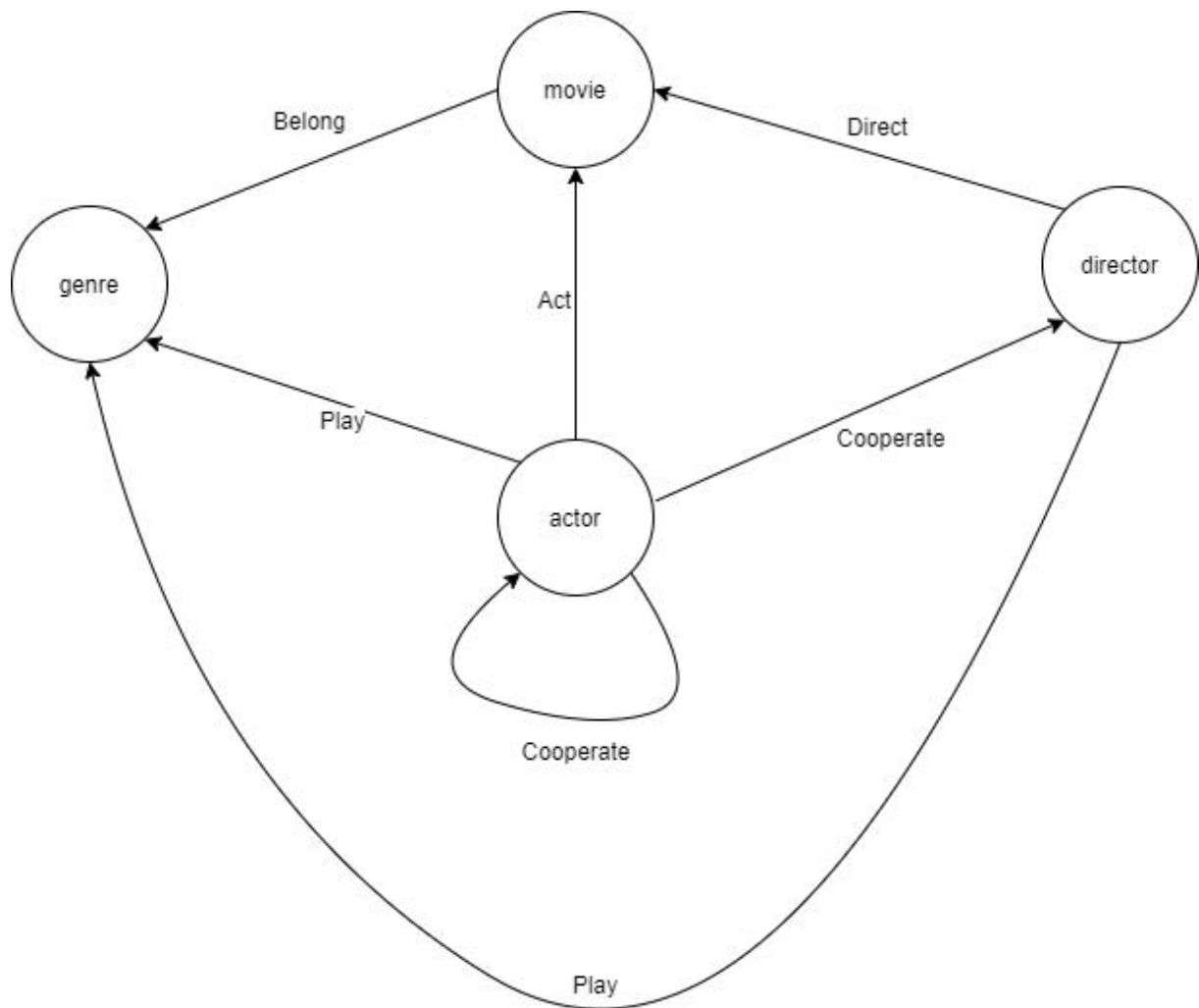
因为 Spark 的 join 操作非常耗时，为尽快查找到人和人之间的关系，我们新建了 2 张表进行存储，分别存储演员与演员、导演与演员之间的合作关系。人与人之间关系本来可以使用一张表进行存储，但为了加快查找速度，我们将其拆分为两个表，由于每次操作都会扫描全表，所以表行数的减少会使查询速度更快。

同时，针对 movie 的全部演员查询，也是出于减少 join 操作的目的，我们在 movie 中新增 2 个字段 all\_actors 与 all\_stars，将电影的全部演员、主演分别拼接成一个字符串存储，以逗号分割。

### 3.3 图数据库

4 种结点：Movie, Actor, Director, Genre

5 种边：Act, Direct, Belong, Play, Cooperate



● **Movie 结点：**

属性	类型	说明
movie_id	long	电影 ID
name	string	电影名
score	float	电影评分
comment_num	int	电影评论数
date	string	上映时间

● **Genre 结点：**

属性	类型	说明
name	string	电影类型

● **Director 结点：**

属性	类型	说明
----	----	----

name	string	导演名字
------	--------	------

● **Actor 结点：**

属性	类型	说明
name	string	演员名字

● **Act 边：**

属性	类型	说明
is_star	bool	这部电影的该演员是否为主演

● **Play 边：**

属性	类型	说明
play_genre_comment_sum	int	这个演员所参演过的该类型电影评论数加和

● **Direct 边：无属性**

● **Belong 边：无属性**

## 4. 优化

### 4.1 关系型数据库优化

#### 4.1.1 建立索引

建立索引是一种常用且有效的查询优化的方式，它通过牺牲空间来换取时间。如果遇到以下几种情况，比较适合建立索引。

- 该属性为主键或外键
- WHERE 子句中经常出现的属性
- ORDER BY, GROUP BY 中经常出现的属性
- 不经常增删改查的属性。对于数据仓库的应用场景，符合这一情况

其中，索引还分为单一索引和复合索引，我们针对这两种情形分别做了优化。

##### 4.1.1.1 建立单一索引

以查询 xx 评分以上的所有电影信息为例，在 movie 表中的 score 字段建立索引。可以看到，在



建立索引前后，查询时间由 1579ms 下降到了 885ms。加速比=1579/885=1.78，获得了将近一倍的效率提升。

## 关系型数据库：获取xx评分以上的所有电影

评分下限

4.5

提交

查询时间

1579ms

总条数

70693

## 关系型数据库：获取xx评分以上的所有电影

评分下限

4.5

提交

查询时间

885ms

总条数

70693

### 4.1.1.2 建立组合索引

以查询某位演员主演过的所有电影为例，查询语句如下：

```
SELECT movie_id FROM actor_movie WHERE actor_name='Aarno Sulkanen' AND is_star=1
```

可以看到, 查询 actor\_movie 表, 而且 WHERE 子句中包含两个属性。所以对 actor\_name 和 is\_star 两个属性建立组合索引。

优化前, 仅仅对 actor\_name 这一外键建立单一索引; 优化后, 对于 actor\_name 和 is\_star 两个属性建立组合索引。可以看到, 建立组合索引前的查询速度基本在 10ms 以上, 建立组合索引后基本都在 1ms-2ms 之间, 极大提高了查询速度。

## 关系型数据库：查询一个演员主演过的所有电影

演员名 Abelardo De Constantine

提交

查询时间 13ms

总条数 1

## 关系型数据库：查询一个演员主演过的所有电影

演员名 Abelardo De Constantine

提交

查询时间 1ms

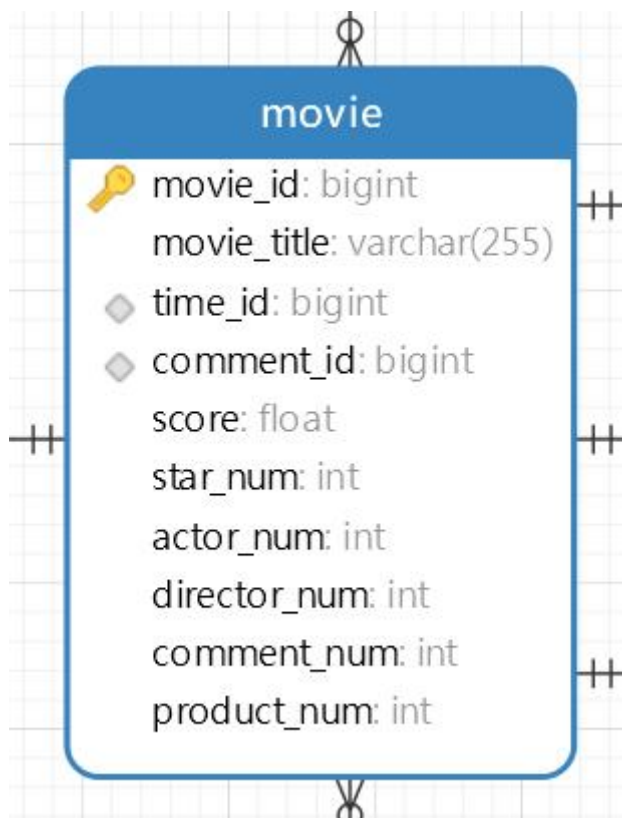
总条数 1

### 4.1.2 冗余存储

在数据仓库的应用场景中, 为了面对大量查询, 加快查询速度, 我们需要采取“反范式”的策略, 建立冗余字段。

第一, 我们采用星型模式, 将数据分散到各个事实表上, 以避免宽列所带来的大量 I/O 操作。

第二, 我们存储了许多的代表总量的数据。例如在 movie 表中, 除了电影的基本信息, 还存储了 star\_num, actor\_num, director\_num 等字段, 避免查询这些相关信息时需要 JOIN 再 COUNT(\*)。



第三，有些相同的字段，我们将其在不同表中重复存储。例如，在 movie 表和 comment 表中均有 comment\_num 字段。这样就使得在面临对 movie 表或 comment 表进行单表查询时，避免了 JOIN 操作。

## 4.2 分布式数据库优化

### 4.2.1 local 模式运行

大多数的 Spark Job 是需要 Hadoop 提供的完整的可扩展性来处理大数据集的。不过，有时输入数据量是非常小的。在这种情况下，为查询触发执行任务消耗的时间可能会比实际 job 的执行时间要多的多。对于大多数这种情况，Spark 可以通过本地模式在单台机器上处理所有的任务。对于小数据集，执行时间可以明显被缩短。

用于测试的 **movie** 表数据总数为 169434 条，属于较小的数据集，所以我们可以开启本地模式进行单机处理，我们可以将 **spark.master** 的值设置为 **local** 来实现此优化。优化实现语句如下所示：

```
SparkLauncher.setMaster("local")
```

优化前和优化后对比如下图所示：

cluster 模式运行时，一次电影的组合查询需要 9.2s

```
dinator stopped!
2022-12-17 07:32:16 INFO  SparkContext:54 - Successfully stopped SparkContext
end time:9248
yyut3#9248
2022-12-17 07:32:16 INFO  ShutdownHookManager:54 - Shutdown hook called
2022-12-17 07:32:16 INFO  ShutdownHookManager:54 - Deleting directory /tmp/spark-e0c9
5-f9fd29c1b625
2022-12-17 07:32:16 INFO  ShutdownHookManager:54 - Deleting directory /tmp/spark-132f
7-b1a269a73c93
root@hadoop-master:/jars#
```

local 模式运行时，同样的查询需要 6.9s

```
2022-12-17 06:55:52 INFO  MemoryStore:54 - MemoryStore cleared
2022-12-17 06:55:52 INFO  BlockManager:54 - BlockManager stopped
2022-12-17 06:55:52 INFO  BlockManagerMaster:54 - BlockManagerMaster stopped
2022-12-17 06:55:52 INFO  OutputCommitCoordinator$OutputCommitCoordinatorEndpoint
dinator stopped!
2022-12-17 06:55:52 INFO  SparkContext:54 - Successfully stopped SparkContext
end time:6952
yyut3#6952
2022-12-17 06:55:52 INFO  ShutdownHookManager:54 - Shutdown hook called
2022-12-17 06:55:52 INFO  ShutdownHookManager:54 - Deleting directory /tmp/spark-
8-9812dacd413d
2022-12-17 06:55:52 INFO  ShutdownHookManager:54 - Deleting directory /tmp/spark-
c-7c41e5627477
root@hadoop-master:/jars#
```

#### 4.2.2 列存储文件

最初我们的分布式文件型数据库使用 csv 格式存储文件，直接将关系型数据库的表导出为 csv 格式传递到 hdfs 中。

现更换文件存储类型，将 csv 文件转化为 parquet 文件。用 spark 程序转化 csv 文件



```

Dataset<Row> csv_df = sparkSession
    .read()
    .format("csv")
    .option("header", "true")
    .load(csv_path);
csv_df.show();
csv_df.write().format("parquet").save(p_path);

```

后续 spark 查询读取 parquet 文件。

parquet 作为列式存储文件，存储查询中只涉及到部分列，所以只需读取这些列对应的数据块，而不需要读取整个表的数据，从而减少 I/O 开销，提高查询的性能表现。

下面是同一个查询的两种结果，前者读取 csv 文件，后者读取 parquet 文件。

```

2022-12-17 06:55:52 INFO MemoryStore:54 - MemoryStore cleared
2022-12-17 06:55:52 INFO BlockManager:54 - BlockManager stopped
2022-12-17 06:55:52 INFO BlockManagerMaster:54 - BlockManagerMaster stopped
2022-12-17 06:55:52 INFO OutputCommitCoordinator$OutputCommitCoordinatorEndpoint:54 - OutputCommitCoordinator stopped!
2022-12-17 06:55:52 INFO SparkContext:54 - Successfully stopped SparkContext
end time:6952
yyut3#6952
2022-12-17 06:55:52 INFO ShutdownHookManager:54 - Shutdown hook called
2022-12-17 06:55:52 INFO ShutdownHookManager:54 - Deleting directory /tmp/spark-8-9812dacd413d
2022-12-17 06:55:52 INFO ShutdownHookManager:54 - Deleting directory /tmp/spark-c-7c41e5627477
root@hadoop-master: /jars#

show time:5539
2022-12-17 08:52:06 INFO AbstractConnector:318 - Stopped Spark@23671a96{HTTP/1.1,[
2022-12-17 08:52:06 INFO SparkUI:54 - Stopped Spark web UI at http://hadoop-master
2022-12-17 08:52:06 INFO MapOutputTrackerMasterEndpoint:54 - MapOutputTrackerMasterEndpoint stopped!
2022-12-17 08:52:06 INFO MemoryStore:54 - MemoryStore cleared
2022-12-17 08:52:06 INFO BlockManager:54 - BlockManager stopped
2022-12-17 08:52:06 INFO BlockManagerMaster:54 - BlockManagerMaster stopped
2022-12-17 08:52:06 INFO OutputCommitCoordinator$OutputCommitCoordinatorEndpoint:54 - OutputCommitCoordinator stopped!
2022-12-17 08:52:06 INFO SparkContext:54 - Successfully stopped SparkContext
end time:5572
yyut3#5572
2022-12-17 08:52:06 INFO ShutdownHookManager:54 - Shutdown hook called
2022-12-17 08:52:06 INFO ShutdownHookManager:54 - Deleting directory /tmp/spark-a936e
2022-12-17 08:52:06 INFO ShutdownHookManager:54 - Deleting directory /tmp/spark-ce882
root@hadoop-master: /jars#

```


## 4.3 图数据库优化

### 4.3.1 建立索引

Neo4j 的 B 树索引提供了单一索引和组合索引的方式，它限于对单个 LABEL 建立索引。这里以单一索引为例，查询评分在 4.5 以上的电影数量。Cypher 语句如下：

```
MATCH (m:Movie)
WHERE m.score>4.5
RETURN COUNT(m)
```

在:Movie 标签的 score 字段上建立索引，建立索引前后的查询时间对比如下。可以看到，查询时间从 103ms 降到了 17ms，加速比=103/17=6.06，获得了巨大的性能提升。

\$ MATCH (m:Movie) WHERE m.score>4.5 RETURN COUNT(m)	
 Table	<b>COUNT(m)</b>
 Text	55250
 Code	
Started streaming 1 records after 103 ms and completed after 103 ms.	

```
$ MATCH (m:Movie) WHERE m.score>4.5 RETURN COUNT(m)
```

Table	COUNT(m)
Text	55250
Code	

Started streaming 1 records after 17 ms and completed after 17 ms.

### 4.3.2 将电影类型与演员之间建立联系

在老师提供的需求中，需要查询：如果要拍一部 xx 类型的电影，最合适的 n 人演员组合是什么？如果不建立演员和电影类型的关系，那么在查询时需要查询两个关系：演员和电影的关系，电影和类型的关系。

假设演员和电影之间共有 m 条 Act 边，电影和类型之间共有 n 条 Belong 边。那么查询演员和类型的时间复杂度为： $O(\log m * \log n)$ 。

如果建立了演员和类型的边，假设演员和类型之间共有 p 条 Play 边，那么查询演员和类型的时间复杂度为： $O(\log p)$

又有关系式： $p < m * n$

所以， $O(\log p) = O(\log(m * n)) = O(\log m + \log n) < O(\log m * \log n)$

有效降低了这类查询的时间复杂度。

## 5. 数据治理

### 5.1 数据质量

对于数据质量，在本项目中主要是在爬虫和 ETL 的过程中进行控制。我们主要考虑了以下 3 个方面。

### 5.1.1 数据完整性

在本项目中，数据的完整性可以体现在两方面：数据条目是否遗漏，已有的数据中的缺值比例。

#### 5.1.1.1 减少数据条目的遗漏（字段大小写，用 in；模拟点击展开）

数据条目最有可能丢失的阶段是在爬虫的时候。在爬虫阶段，并不是每一个产品的页面都能够爬取到信息。有的是因为产品下架了，本来就不应该爬取该网页的信息；有的是因为网络延迟或是 IP 被封，导致被验证码拦截或是访问超时。所以必须要通过观察网页的行为，编写一套完整的逻辑判断来分辨各个情况。

- 对于产品下架的情况，直接跳过这条产品，不爬取数据。
- 对于验证码拦截，通过破解验证码的库来访问到页面，再爬取数据。
- 对于网络请求超时，重新访问网页，爬取数据。

#### 5.1.1.2 减少数据的缺值

- 利用另一数据源——IMDB 的数据，对 amazon 的缺值进行填充。
- 利用主演信息填充演员信息。

### 5.1.2 数据准确性

数据准确性是我们要考虑的一个重要因素。对此，我们考虑了以下 2 个方面。

- 字段中的一些混乱的字符可能会影响数据的准确性。比如，某些产品中没有记载人名，就用一些占位符如 N/A, Various 来填充；在进行数据清洗的时候要进行筛除；甚至还有一些字段混入了网页中其他 DOM 元素的文字，也要进行筛除。再者，还有一些字段夹杂了隐藏的字符，如空格和换行符，难以发现而且会真正地影响到查询，所以也需要筛除。
- amazon 与 imdb 的数据，有些不能进行合并。例如，amazon 和 imdb 的电影评分绝对不能进行合并。因为这两个平台的评分完全是基于两套评价体系，它们之间没有任何可比性，所以对于评分数据，不能盲目地合并到一个字段中，否则这个字段将失去意义。由于在 imdb 上爬到的条目较少，我们最终只选用了 amazon 的数据。

但是，由于时间原因，我们在保证数据准确性这一块还做的并不是很好。有些字段中依然会存在一些乱的字符。我们原来的解决方案是通过审阅数据，把肉眼能观察到的错误情况写进脚本中筛除。但是，面对这样数据量较大的场景，这种方法并不能取得很好的效果。我们未来对此的改进方案是，对每个字段的数据规定一种规范的格式，将不符合这种规范的数据全部剔除，以保证数据的准确性。

### 5.1.3 数据一致性

对于数据一致性，我们主要探讨人名合并。我们在 ETL 时先是对同一部电影，不同产品之间的人名做了一个字符串拼接。由于这些名字信息来源于不同的产品页面，这势必导致同一个人可



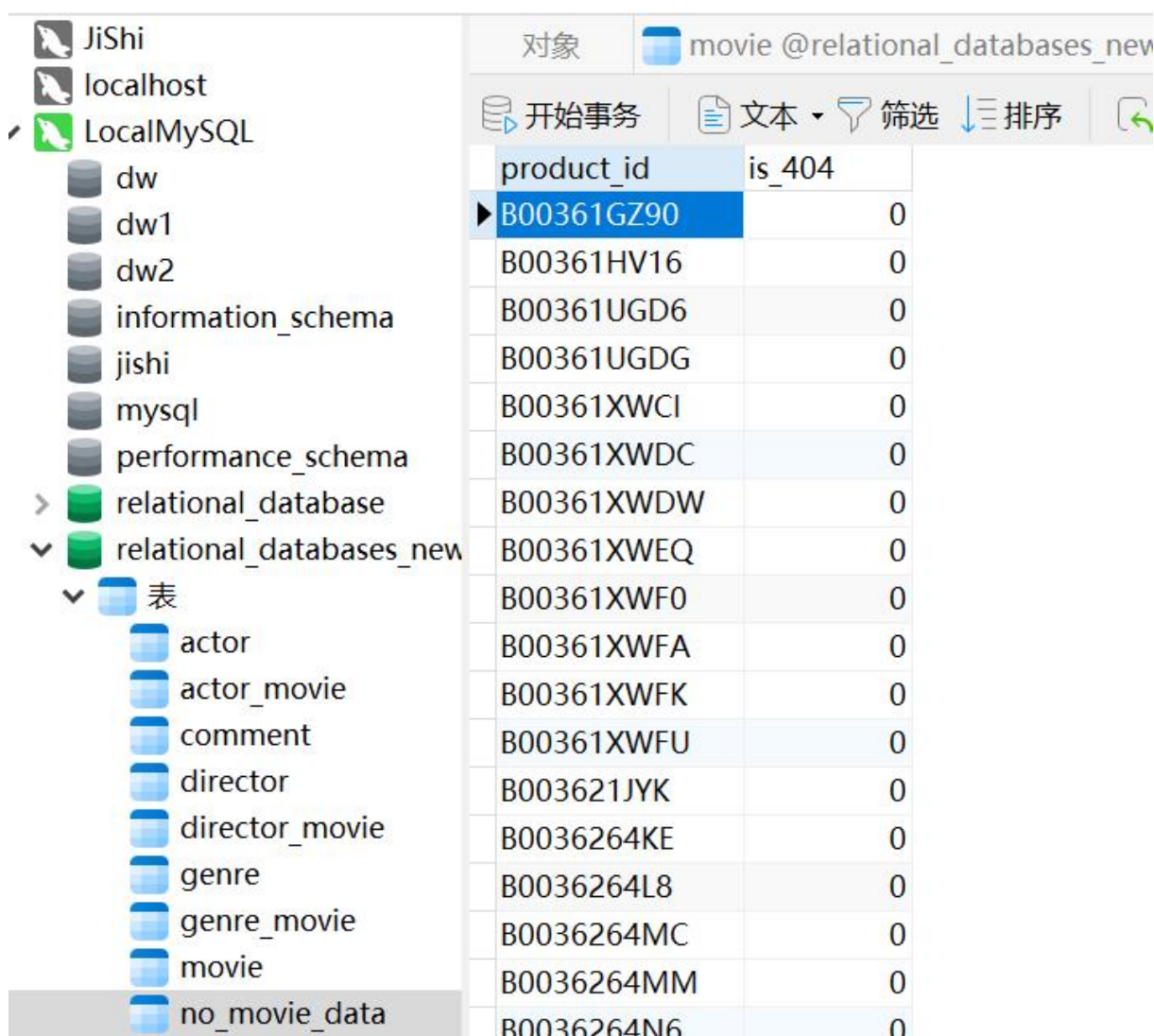
能以两种名字的形式存在于这一组数据中。为了解决这个问题，我们通过判断子串的方式，来处理人名缩写的问题，以保证人名信息的一致性。

## 5.2 数据血缘

### 5.2.1 非电影数据

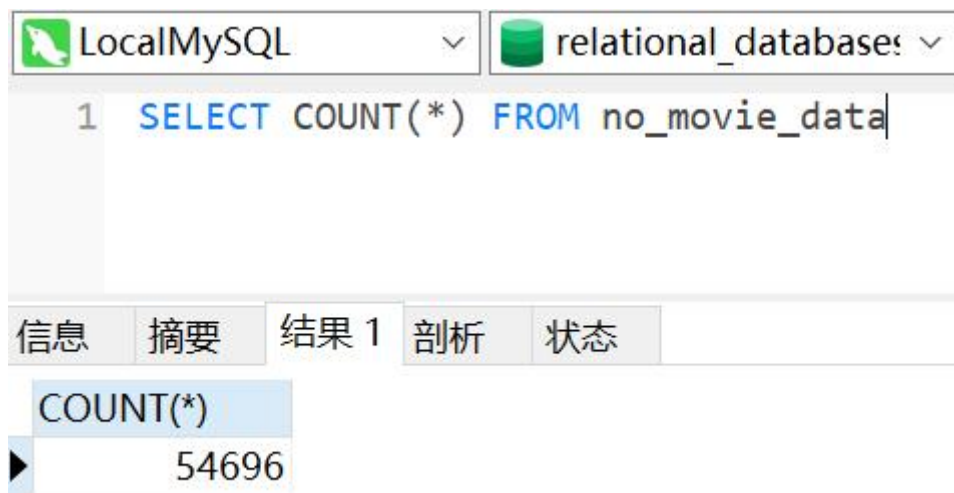
在原始的 txt 文档中，我们一共提取出了 253061 条产品。但这些产品并不都是电影，有些是已经下架的产品，还有一些是其他类型如 TV 的产品。我们把这些数据统称为非电影数据，并把它们存储在一张单独的表里。

在其中设置一个字段，标识是已经下架的商品（404 页面）还是被认定为不是 Movie 的商品。



product_id	is_404
B00361GZ90	0
B00361HV16	0
B00361UGD6	0
B00361UGDG	0
B00361XWCI	0
B00361XWDC	0
B00361XWDW	0
B00361XWEQ	0
B00361XWFO	0
B00361XWFA	0
B00361XWFK	0
B00361XWFU	0
B003621JYK	0
B0036264KE	0
B0036264L8	0
B0036264MC	0
B0036264MM	0
B0036264N6	0

在这张表中，我们一共存储了 54696 条非电影产品的 amazon 产品号。



## 5.2.2 电影数据

数据血缘，指的是探寻数据的根本、源头。我们分析处理的数据，可能来源很广泛，不同来源的数据，其数据质量参差不齐，对分析处理的结果影响也不尽相同。当数据发生异常，我们需要能追踪到异常发生的原因，把风险控制在适当的水平。

在我们的数据获取过程中，用到了两个数据源：amazon 和 imdb。我们存储的数据血缘信息中，能够追溯电影是来源于哪一个平台的哪一个产品号，能够直接根据该信息跳转到溯源的网页。

以哈利波特的电影数据为例，我们可以回答：我们找到了多少哈利波特系列的电影？这个电影有多少版本？多少网页？

我们在数据血缘的应用界面输入 Harry Potter 进行模糊搜索，显示出了 71 条结果，这说明经过我们的合并，最终 Harry Potter 系列电影的结果有 71 部。并且对于每一部电影，我们都能显示相应的版本数，并能溯源到网页的 URL。

Harry Potter										
电影ID	电影名称	电影分数	上映时间	导演	演员	评论数量	电影对应版本数	操作		
3640	Harry Potter: Witchcraft Repackaged VHS	4.2	2002-1-1		Caryl,Matrisciana	22	1	<a href="#">查看</a>		
34404	Harry Potter & Sorcerer's Stone	4.4	2002-5-28	Chris Columbus,Phil Alden Robinson	Al Vandecruys,Alison Darcy,Ben Affleck,Bruce McGill,Dale Godboldo,Daniel Radcliffe,Emma Watson,Ian Mongrain,James Cromwell,John Beasley,John Cleese,John Hurt,Ken Jenkins,Lee Garlington,Maggie Smith,Morgan Freeman,Ostap Soroka,Philip Baker Hall,Philip Pretten,Richard Cohee,Richard Marner,Robert Martin Robinson,Rupert Grint,Russell Bobbitt	9	1	<a href="#">查看</a>		
40669	Harry Potter and the Prisoner of Azkaban	4.9	2004-11-23	Alfonso Cuar'n,Alfonso Cuaron	Abby Ford,Adrian Rawlins,Daniel Radcliffe,Emma Watson,Fiona Shaw,Gary Oldman,Geraldine Somerville,Harry Melling,James Phelps,Jim Tavar,Jimmy Gardner,Lee Ingleby,Lenny Henry,Michael Gambon,Oliver Phelps,Pam Ferris,Richard Griffiths,Robbie Coltrane,Robert Hardy,Rupert Grint	201	7	<a href="#">查看</a>		
40864	Harry Potter and the Order of the Phoenix (Two-Disc Special Edition)	4.7	2013-4-9	David Yates	Adrian Rawlins,Brendan Gleeson,Daniel Radcliffe,Emma Watson,Fiona Shaw,George Harris,Geraldine Somerville,Harry Melling,Helena Bonham Carter,Jason Boyd,Jessica Hynes,Kathryn Hunter,Miles Jupp,Natalia Tena,Ralph Fiennes,Richard Griffiths,Richard Macklin,Robbie Coltrane,Robert Pattinson,Rupert Grint	2	2	<a href="#">查看</a>		

Dashboard / 溯源查询

Harry Potter

电影溯源信息

Harry Potter and the Prisoner of Azkaban

共合并 7 个版本数量的电影,点击url可访问源网址

以下为未合并前的电影数据信息:

产品ID	来源	url
B00005JMAH	amazon	<a href="https://www.amazon.com/dp/B00005JMAH">https://www.amazon.com/dp/B00005JMAH</a>
B0002TT0NW	amazon	<a href="https://www.amazon.com/dp/B0002TT0NW">https://www.amazon.com/dp/B0002TT0NW</a>
B0002VB24K	amazon	<a href="https://www.amazon.com/dp/B0002VB24K">https://www.amazon.com/dp/B0002VB24K</a>
B0008KLW7W	amazon	<a href="https://www.amazon.com/dp/B0008KLW7W">https://www.amazon.com/dp/B0008KLW7W</a>
B000FAOCHW	amazon	<a href="https://www.amazon.com/dp/B000FAOCHW">https://www.amazon.com/dp/B000FAOCHW</a>
B000W745CU	amazon	<a href="https://www.amazon.com/dp/B000W745CU">https://www.amazon.com/dp/B000W745CU</a>
tt0304141	imdb	<a href="https://www.imdb.com/title/tt0304141">https://www.imdb.com/title/tt0304141</a>

电影ID	电影名称	电影分数	上映时间	导演	演员	评论数量	电影对应版本数	操作
3640	Harry Potter: Witchcraft Repackaged VHS	4.2	2002-1-1		Caryl,Matrisciana	22	1	<a href="#">查看</a>
34404	Harry Potter & Sorcerer's Stone	4.4	2002-5-28	Chris Columbus,Phil Alden Robinson	c,Bruce McGill,Emma Watson,Ian Beasley,John Cleese,n,Maggie Smith,p Baker Hall,Marner,Robert ell Bobbitt	9	1	<a href="#">查看</a>
40669	Harry Potter and the Prisoner of Azkaban	4.9	2004-11-23	Alfonso Cuar'n,Alfonso Cuaron	Radcliffe,Emma Watson,Geraldine Somerville,r,Jimmy Gardner,Gambon,Oliver Phelps,Robbie Coltrane,	201	7	<a href="#">查看</a>
40864	Harry Potter and the Order of the Phoenix (Two-Disc Special Edition)	4.7	2013-4-9	David Yates	el Radcliffe,E s,Geraldine S n Carter,Jason Miles Jupp,Na ths,Richard M acklin,Robbie Coltrane,Robert Pattinson,Rupert Grint	2	2	<a href="#">查看</a>

## 6. 查询比较

### 6.1 特定查询应当选择适合的存储模型

#### 6.1.1 关系型数据库

关系型数据库是最经典的数据库，它能够将同一业务范围的的事物组合成一张表。关系型数据库适合做单一条件的查询，在面对这种单一查询时，加了索引以后往往非常高效。比如查询同一类别的所有电影。但是，关系型数据库在面对多表的 join 操作时，会非常耗时。

#### 6.1.2 分布式文件型数据库

##### 6.1.2.1 查询缓慢的原因

可以注意到本项目的 spark 查询，性能远远低于其他两种数据库，下面是我们分析的原因

##### 1. spark 任务启动需要时间

下面是一次查询的时间节点打印，创建 sparkSession 就耗时 1.2s

单个 spark 任务的启动就需要占据相当的时间

```
yyut4#yyut4# create sparksession done:1271
yyut4#yyut4# read file done:4710
yyut1#1
yyut2#[{"actor_name":"Tom","star_movie_num":"0","act_movie_num":"2"}]
yyut4#yyut4# query done:7918
yyut4#yyut4# stop done:7951
yyut3#7952
```

##### 2. spark 读取文件耗时

由于我们没有使用 hive 数据库，所有查询都是基于 hdfs 文件系统的，每一次查询都需要重新读取文件，从上图可以看到，读取文件耗时 4.5s。同时也是出于这个原因，我们把 csv 存储文件替换成更适合读取的 parquet 列存储文件

##### 6.1.2.2 适合的查询

从本项目的结果来看，spark 分布式查询除了利用冗余表避免 join 操作时可以胜过关系型数据库，在基本相同的查询操作下，我们的 spark 分布式查询并不占任何时间上的优势。

尽管在我们的项目里无法体现分布式的优势，但我们认为分布式查询的优势在于处理较大数据量时，分布式数据库与查询的可靠性与可拓展性。

在做好异常处理的情况下，个别场地或个别通信链路发生的故障，不致于导致整个系统的崩溃，而且系统的局部故障不会引起全局失控。

分布式更容易扩展，在分布式环境中，在添加更多数据、增加数据库大小方面扩展系统，增加数据库大小或添加更多处理器要容易得多。

### 6.1.3 图数据库

图数据库一个非常明显的优势在于查询多个实体之间的关系。在进行“关系”的查询时，图数据库要比关系型数据库快上不少。此外，图数据库处理非结构化数据，十分灵活。

## 6.2 测试用例及对比

### 6.2.1 简单查询

以查询所有评分 4.5 分以上的电影为例：

#### 6.2.1.1 测试代码

(1) 关系型数据库

```
SELECT movie_id, movie_title  
FROM movie  
WHERE score > 4.5
```

## (2) 分布式数据库

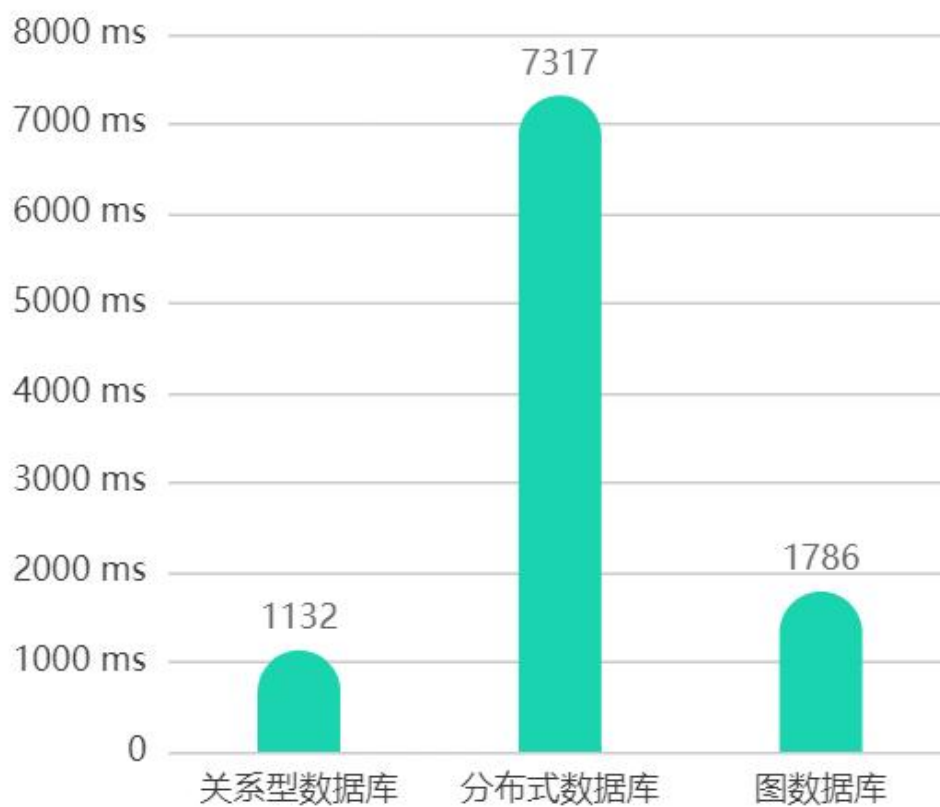
```
Dataset<Row> sql_df=csv_df  
.select("movie_id","movie_title").where("score >= 4.5");
```

## (3) 图数据库

```
MATCH (n:Movie)  
WHERE n.score > 4.5  
RETURN n.movie_id, n.movie_title
```

### 6.2.1.2 结果截图

#### 性能比较查询



### 6.2.2 总量查询 (Action 分类一共有多少电影)

#### 6.2.2.1 测试代码

(1) 关系型数据库

```
SELECT COUNT(*)  
FROM genre_movie  
WHERE genreTitle = 'Action'
```

## (2) 分布式数据库

```
sql_df=csv_df  
.where("genre_title='Action'");
```

## (3) 图数据库

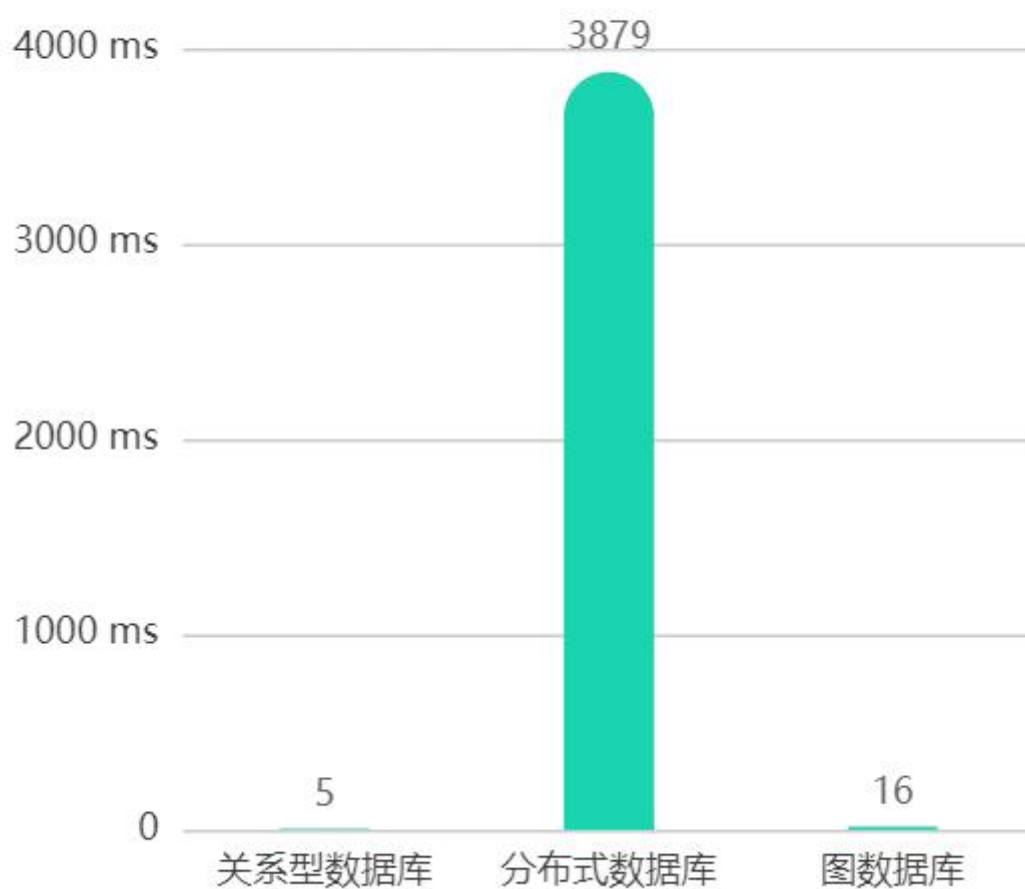
```
MATCH(movie:Movie)-[r:Belong]->(genre:Genre{name:$genreTitle})  
RETURN COUNT(*)
```

### 6.2.2.2 结果截图

这里需要特别说明一下，关系型低达 2ms 的原因，是存储了总量的元数据。



## 性能比较查询



### 6.2.3 关系查询（经常合作的导演和演员）

#### 6.2.3.1 测试代码

(1) 关系型数据库

```
SELECT actor_name,director_name,COUNT(DISTINCT movie_id) AS cooperate_num  
FROM actor_movie JOIN director_movie USING(movie_id)  
GROUP BY actor_name,director_name  
HAVING COUNT(DISTINCT movie_id) > 25
```

(2) 分布式数据库

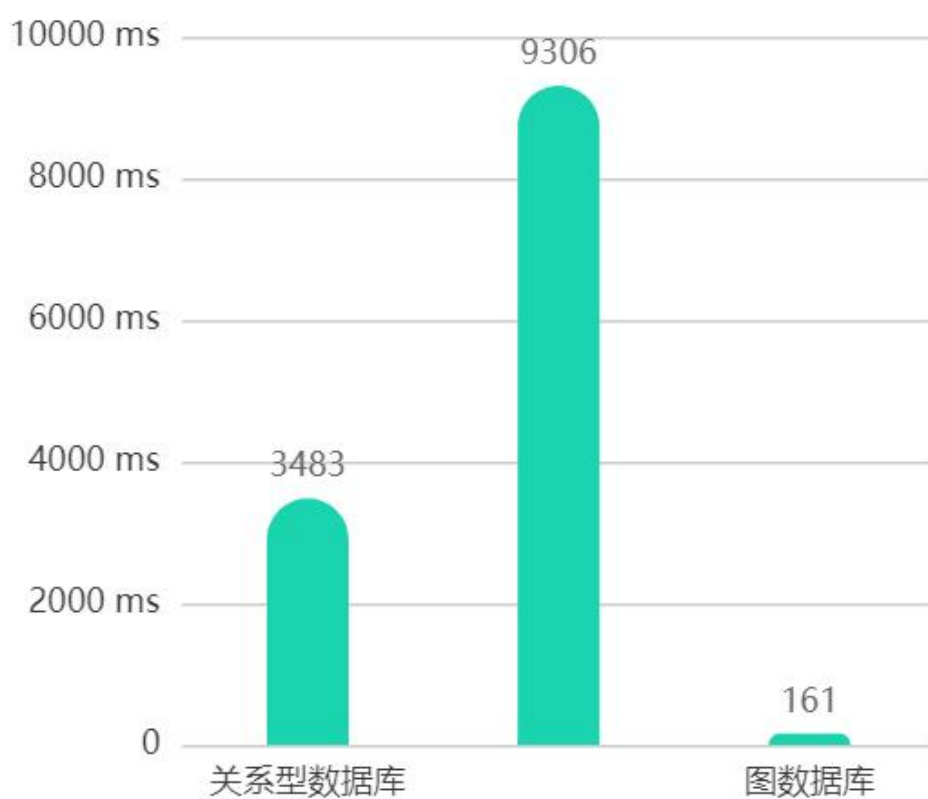
```
sql_df=csv_df.where("cooperate_num">"+cooperate_num);
```

(3) 图数据库

```
MATCH p = (n1:Actor)-[r:Cooperate]->(n2:Actor)
WHERE r.cooperate_num > 25
RETURN p
```

### 6.2.3.2 结果截图

## 性能比较查询



## 7. 前端展示页面

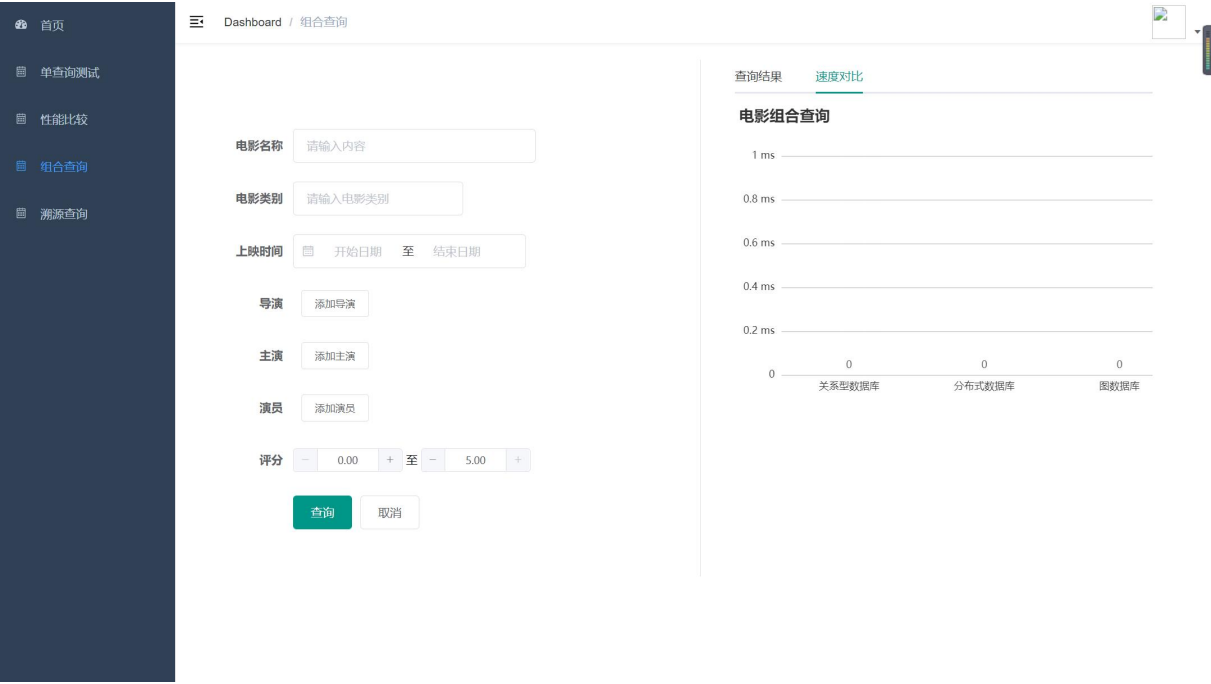
单查询测试：



性能比较：



组合查询：



溯源查询：

