数据仓库项目报告

1753414 卜滴

1752339 刘一默

1750160 李雨龙

1752289 吕雪飞

目录

1 ETL

1.1 Extract抽取

1.1.1 获取到所有电影信息页面的URL

在800万的评论信息中，截取productid字段中的电影ID，然后与https://www.amazon.com/dp/拼接生成所有电影信息页面的URL，去重之后，大约有25万左右。

1.1.2 编写爬虫程序

1. 由于Python的解释器与进程机制，所以开启多个进程进行爬取的措施是并不能带来很大的性能提升的，利用Python的aiohttp与asyncio库编写异步多协程爬虫程序，能够有效的提升爬取速度。
2. 利用BeautifulSoup库对获取到的html页面进行信息处理，提取所需要的关键字段。
3. 由于Amazon设置了教多的反爬机制，所以采取多种措施，例如设置请求头，设置cookie，使用代理等等。

1.1.3 最终结果

最终获得未进行清洗过的电影信息24万左右，包含字段：

```
ID,title,run_time,IMDB,release_year,dvd_release_date,introduction,genres,director,starring
,supporting_actors,actors,studio,MPAA_rating,subtitles,audio_languages,format,rating,ratin
g_num,rating_detail
```

1.2 Transform转换

1.2.1 清洗爬取的电影数据

由于各种因素，我们爬下来的数据不是很干净，有部分重复数据，所以我们先利用python pandas库中 的数据去重函数，把完全一致的数据给清理掉，剩下的数据作为我们的基础数据。

1.2.2 清洗评论数据

再进行正式的数据清洗之前，我们应该先根据索爬取的电影数据，对给出的用户评论的数据集进行一次 筛选。因为在各种因素的影响下，我们并没有能够将所有数据都爬取下来。所以用户评论数据集中有一 些数据的电影ID是在电影数据中缺失的，这样的数据是无效的，为了保证之后数据外键依赖的正确性， 我们有必要将这些无效数据给清理掉。

1.2.3 清洗算法

为了快速清理，避免嵌套遍历带来的巨大时间消耗，我们采取如下算法对数据进行清理。
首先分别对电影和评论数据按电影ID进行从小到大快速排序，然后两边同时遍历，当电影数据集里的ID 大于评论数据集中的ID时，删除评论数据集中的该条记录，再向下遍历，直到电影数据被遍历完。如果 最后评论数据集内还有剩余，便全部删去。这样能很快的清理并得到有效数据。

1.2.4 电影数据去重清洗

对于电影数据的清洗最重要的就是查重了。能够准确的合并重复的数据项是该项工作的重点，也是难点。

查重判定

在爬下来的数据中，电影的相关信息有很多，怎么在相关信息中找出关键点并作为查重依据是最关键的。

1. **电影名**：首先最容易想到的是电影重名的情况，但是很明显，电影重名不一定意味着是同一部电 影，然而，如果是同一部电影的两条数据，那么他们的标题应该具有很大的相似性。所以我们决定 将电影名作为查重的第一依据，然后在此基础上进行第二步的筛查。

2. **电影发行年**：如果两条数据是指的同一部电影，那么他们的发行年份一定是相同的。这是我们查重 的第二依据。但是，在爬下来的数据中，我们发现有些数据的电影发行年份是缺失的，特别是对于 一些上了年份的电影，这样的现象更为严重。如果说两条记录中有一条发行年份缺失，甚至两条记 录都没有发行年份，我们怎么去判断呢?另外，即使能判断两条记录的发行年份是一样的，那就一 定能认为这两天记录对应的是同一部电影吗?所以我们需要第三个判定的字段来帮助我们更加准确 地做到查重判定。
3. **电影评价数**：为了找到这第三个字段，我们找出来两部名字类似，发行年份一样，但电影ID不同 的电影。我们通过其ID拼接成的URL来访问页面，发现这两条记录指的确实是同一部电影，都是 《复仇者联盟》，只不过其中一部电影的标题上还标注了版本的相关信息。而在浏览网页时，我们 发现两个页面的电影评分以及电影评分数完全一致，不仅如此，在评分细节中的内容也是一样的。 也就是说这大概是亚马逊的两个页面，一个负责产品信息，一个负责电影信息。但评价都是统一 的。这样，我们就找到了我们的第三个字段，评分数量。

以上就是我们查重的基本判定规则，我们在遍历数据的过程中，就按先判断电影名，再判断电影发行年和电影评价数量来判定两条记录是不是对应同一部电影。

合并规则

在找到对应同一部电影的多条记录后，我们需要做的是将多条记录的数据进行整合，而不是单纯的丢弃 记录到只剩下一条记录。整合的意义在于使得我们的数据更加完整有效，能填补某些记录的数据的缺失。

合并的基本规则很简单，是一个查漏补缺的过程。对于简单的字段，我们将选择并保留多项数据中的一 个，而对于一些复杂的字段，譬如电影导演、主演、参演等，我们选择将各个字段的信息提出来，并根据分隔符分隔后做成集合，然后再取并集，这样就能保证我们保留了能获取到的所有记录。而对于ID， 我们都保留要合并的数据组中的第一条数据的ID。

合并记录

对于我们合并的所有数据，都应该将他们的主码，也就是电影ID记录下来，这样做的意义在于能根据合 并的情况来对相应的评论进行同样的合并，而不是简单的舍弃那些被合并的数据的评论。所以

我们在每 一次合并时，将所有被合并的数据记录下来，存在一个npy文件中，用于下一步对评论数据集的合并。

查重算法

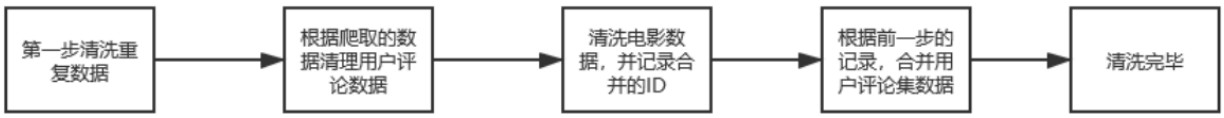
由于初始电影数据量比较大，有足足20多万，再加上算力实在有限，所以我们不可能利用嵌套遍历来查重。因此设计了以下算法:

先将电影数据集按电影title进行从小到大的排序，因为这是我们查重最主要的以据。然后遍历数据集， 对其中的每一个数据，都向下遍历搜索50条数据，并一一做对比，如果找到要合并的数据，便合并数据，并将其ID都记录下来，放入一个NP数组，这样既能做到记录合并情况，也能再之后的遍历中先查 看该数据是否已经被合并，如果已经被合并过，就直接continue，不再进行额外操作，从而减少总体的计算量。

1.2.5 用户评论数据合并

通过读取以上电影数据清洗所得到的npy文件，我们能很方便的得到所有被合并的数据组的电影ID。因 此我们只需要遍历一遍用户评论数据，对于每一条数据判断其电影ID是否在被合并的ID里，如果在其中，便将其ID改为该数据组中的第一个电影ID，也就是在电影数据集中最后得以保留的电影ID。

1.2.6 流程总结

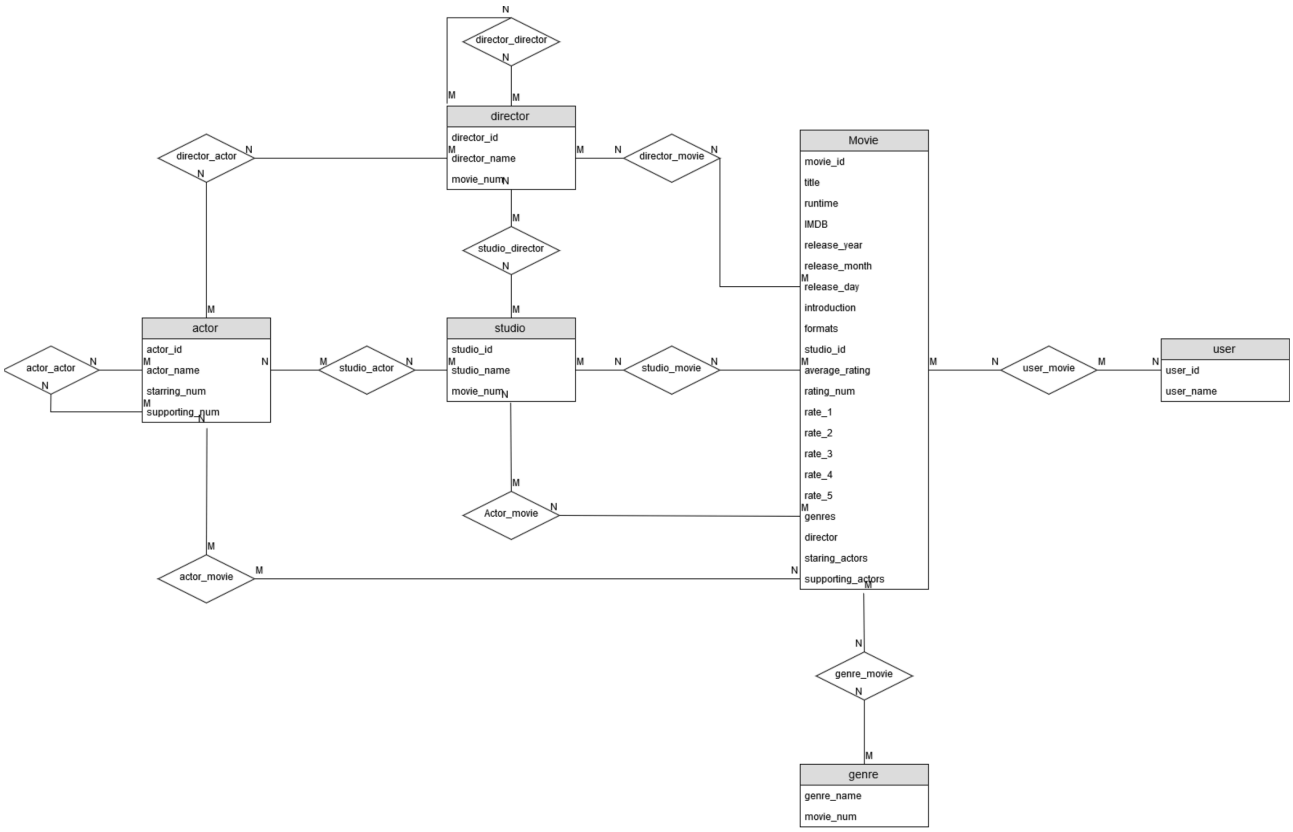


1.3 Load加载

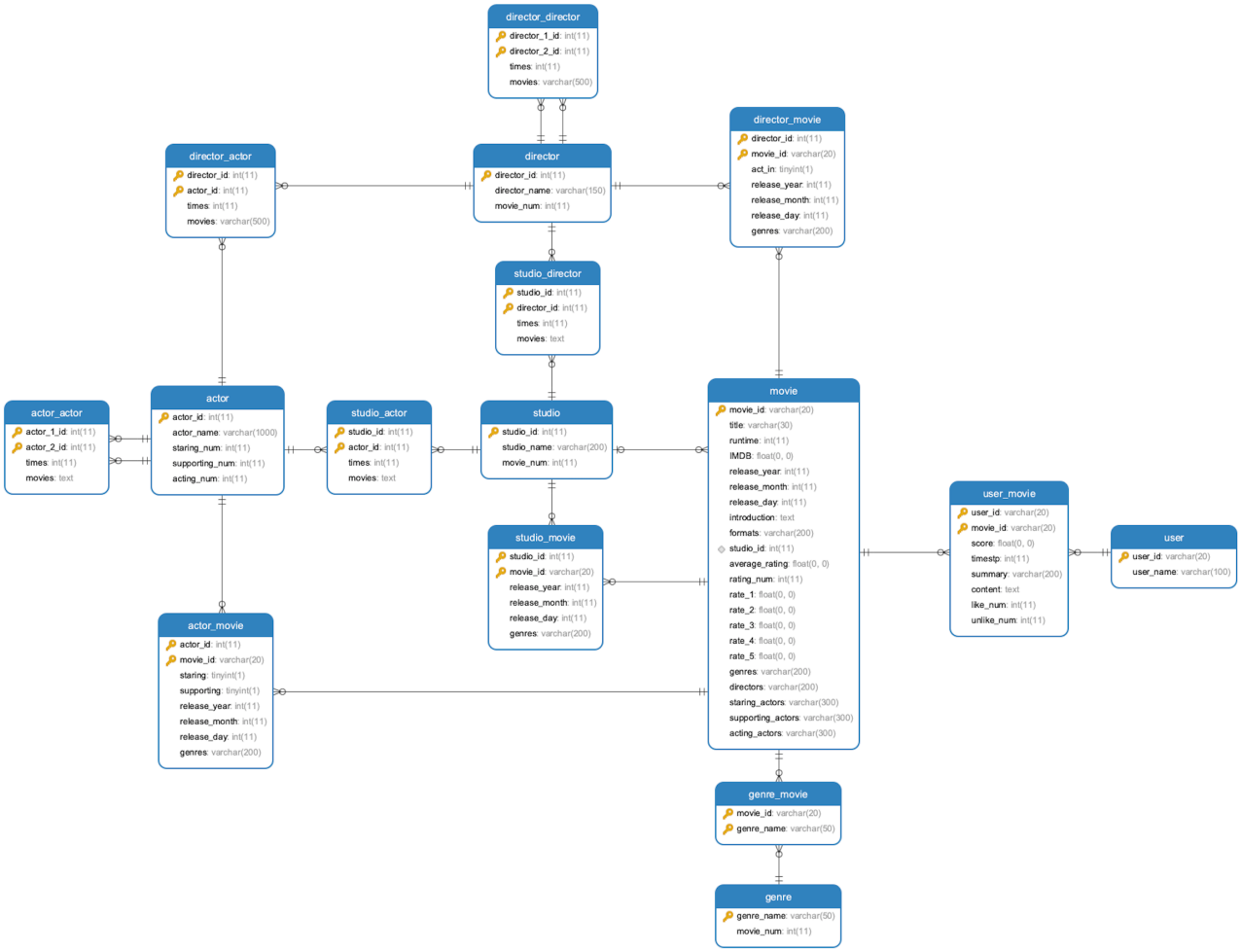
针对不同数据库的不同特性，我们选择了不同的加载方式，具体方法将在下文展开描述。

2 逻辑、物理存储模型

2.1 逻辑存储模型



2.2 物理存储模型


director_director
director_1_id: int(11)
director_2_id: int(11)
times: int(11)
movies: varchar(500)
director_actor
director_id: int(11)
actor_id: int(11)
times: int(11)
movies: varchar(500)
director
director_id: int(11)
director_name: varchar(150)
movie_num: int(11)
director_movie
director_id: int(11)
movie_id: varchar(20)
act_in: tinyint(1)
release_year: int(11)
release_month: int(11)
release_day: int(11)
genres: varchar(200)
studio_director
studio_id: int(11)
director_id: int(11)
times: int(11)
movies: text
actor_actor
actor_1_id: int(11)
actor_2_id: int(11)
times: int(11)
movies: text
actor
actor_id: int(11)
actor_name: varchar(1000)
staring_num: int(11)
supporting_num: int(11)
acting_num: int(11)
studio_actor
studio_id: int(11)
actor_id: int(11)
times: int(11)
movies: text
studio
studio_id: int(11)
studio_name: varchar(200)
movie_num: int(11)
movie
movie_id: varchar(20)
title: varchar(30)
runtime: int(11)
IMDB: float(0, 0)
release_year: int(11)
release_month: int(11)
release_day: int(11)
introduction: text
formats: varchar(200)
studio_id: int(11)
average_rating: float(0, 0)
rating_num: int(11)
rate_1: float(0, 0)
rate_2: float(0, 0)
rate_3: float(0, 0)
rate_4: float(0, 0)
rate_5: float(0, 0)
genres: varchar(200)
directors: varchar(200)
staring_actors: varchar(300)
supporting_actors: varchar(300)
acting_actors: varchar(300)
user_movie
user_id: varchar(20)
movie_id: varchar(20)
score: float(0, 0)
timestp: int(11)
summary: varchar(200)
content: text
like_num: int(11)
unlike_num: int(11)
user
user_id: varchar(20)
user_name: varchar(100)
studio_movie
studio_id: int(11)
movie_id: varchar(20)
release_year: int(11)
release_month: int(11)
release_day: int(11)
genres: varchar(200)
actor_movie
actor_id: int(11)
movie_id: varchar(20)
staring: tinyint(1)
supporting: tinyint(1)
release_year: int(11)
release_month: int(11)
release_day: int(11)
genres: varchar(200)
genre_movie
movie_id: varchar(20)
genre_name: varchar(50)
genre
genre_name: varchar(50)
movie_num: int(11)

3 关系型数据库——MySQL

3.1 存储模型优化

在设计关系型数据仓库时，我们首先考虑到的点就是传统的关系型数据库并不适合作为数据仓库，因为 其面向的主要业务是少量数据的插入、修改和删除，而对于相对大量的数据的涌入，存储和查询来说， 传统的关系型模型是很难满足其需求的。所以我们针对模型的建立做了如下优化:

3.1.1 通过冗余提高查询效率

为了高效率地完成相关查询业务，我们将部分关键信息重复存储在不同的表内，例如在电影(movie)表中存储所有的版本信息而没有将版本抽离出来;在电影(movie)表中存储所有的演员和导演姓名信息。除了实体表，在各个关系表中我们也采取了相似的措施，如将电影时间信息存储在电影-演员(actor_movie)关系表中;将电影名存储在工作室-演员(studio_actor)关系表中。
这些措施都是为了在执行某些复杂查询的过程中尽量减少联表查询的需求，从而大大地提高查询效率。

3.1.2 通过提前汇总数据提高查询效率

除了将数据冗余存储以外，对于某些高发生率，高复杂度的业务的相关信息，我们选择提前将其查询计 算出来，并冗余地存储在相关表格中。譬如演员间的合作次数，合作的电影，譬如属于某风格(genre)的电影的数量，这些信息都被我们在建表入库之间就计算了出来，并存在表中。这样虽然会增大数据仓库的维护难度，但能使得该数据仓库的查询效率得到极大的提高。

3.2 导入数据优化

在插入数据阶段，由于部分表的数据量十分庞大，想要短时间内一次性插入数据是十分困难的。所以我们采取了以下措施来加速数据的插入。

3.2.1 使用MySQL文件导入语句

我们在插入之前进行了小规模的对比试验，分别比较了用python插入、用kettle工具插入，用workbench提供的文件导入功能插入，以及使用**load data infile**语句进行插入，发现通过执行该语句来导入是最快的方式。而为了该导入能顺利进行，我们对MySQL数据库的一系列参数也进行了修改，这 将在下文中提到并详细解释。

3.2.2 文件分块导入

在多次尝试一次性大量导入数据失败后，我们选择了将文件分块从而实现更快速更安全的数据导入。一次性执行大量数据导入有如下缺点:

首先，导入速度会随着告诉内存中的数据量的增大而大幅度减少，而且这样的减少是非线性的，一次性大量导入的时间可以是可能是分块导入的数十倍!这样的大幅度的效率降低是我们所不能接受的，所以 我们在导入百万级数据时都是采用分块导入，每次快速导入相对少量数据，这样能很快将所有数据全部 导入。

其次，一次性大量导入会导致快速内存中的数据量过大而使得MySQL server宕机，宕机所带来的后果 是无法接受的。因为再一次启动数据库服务时，MySQL会为了维护数据的正确性和安全性自动开始 Rolling Back，如果上次导入的数据量比较庞大，那么回滚过程就会耗费大量时间，而在回滚的过程中，任何增删改的操作都是不被允许的，这样就会浪费大量时间。

综上，我们在面对大量数据时选择分块插入，并只用了一个下午便快速的完成了所有数据的插入。

3.2.3 数据库参数优化

在插入数据的过程中，数据库参数设置不正确会导致插入效率异常的低，这对于百万级的数据量来说是 不可接受的。故针对数据插入，对数据库的参数进行了如下调整:

secure-file-priv=""

修改数据库的该参数，使得MySQL数据库能利用**load data infile**语句来讲CSV文件的数据导入数据库的表中。该语句是从CSV导入数据最快的办法。

max_allowed_packet=2048M

该参数是影响一次性能在高速缓存中存储的数据量。当一次要插入的数据量过大时，会引起内存爆满而 使得MySQL服务器宕机，由此带来的后果十分恶劣，再次启动服务器后MySQL会进行锁表来保证回滚 的进行，而在这个过程中无法对数据库做任何增删操作。而修改只会，该情况出现的机率会变得很小。

sql-mode="ANSI"

修改该参数，MySQL会降低对插入数据检测的严格性，但同时会大大的增加数据插入效率。由于在之前的处理数据过程中，我们都严谨的处理了各个外键依赖和各个字段的合法性，故在插入的时候我们可以大胆的利用该特点来提高数据库插入的效率。同时这样的设置也省去了对空值的处理，讲该处理交 由数据库完成，也减少了之前数据处理的工作量。

3.3 存储优化

3.3.1 索引优化

为某些关键字段添加索引是最为常规且最为有效的方法。故在我们的数据仓库中，许多表中的关键字段 都被加上了索引，从而大幅度提高了查询速度，尤其是在当数据量十分庞大的时候，索引就显得十分有用。相关测试如下表所示:

表格	业务	优化前耗时	优化后耗时
User_Movie	按评分搜索	154.288s	0.359s
User_Movie	按电影ID搜索	2.812s	<0.0005s
User_Movie	按点赞数搜索	157.390s	0.625s

可以看到，在添加索引后，对于大表的查询效率是有极大的提升，而对于小表来说，在一次性获取大量数据时，索引的存在能使得数据fetch的时间大大减少，所以使用索引对于该数据仓库的效率的提高是有极大意义的。

3.3.2 表结构优化

主要用了冗余存储和提前汇总的方式来优化表结构，使得部分查询业务的效率得到大大的提高，该部分 在上文中已经详细讲过，在此不再赘述。

3.4 数据库特点分析

关系型数据库有如下基本特点：

- 原子性：记录之前的版本，允许回滚

- 一致性：事务开始和结束之间的中间状态不会被其他事务看到

- 隔离性：适当的破坏一致性来提升性能与并行度 例如：最终一致~=读未提交。

- 持久性：每一次的事务提交后就会保证不会丢失

可以看到，关系型数据库的特点和数据仓库的要求相性并不高，但是在某些方面，也有较好的表现。

相对于其他两种数据仓库，最适合关系型数据仓库的查询业务是小数据量的单表查询。因为在实验过程中我们发现，在单表上建立索引来说，关系型数据库的查询效率是最高的，特别是当所select的数据量比较小的时候，关系型数据库的表现远远胜于其他两类数据仓库。

然而，当我们所要的数据量十分大，达到百万级时，尽管关系型数据库在索引的帮助下能很快的查询出结果，但需要花费很长的时间去fetch结果。这个现象在关系型数据仓库中很难优化。

4 分布式文件系统存储模型——Hive

4.1 分布式文件系统数据库与数据库设计

4.1.1 分布式文件系统数据库

搭建全分布式文件系统，节点中共有三台Ubuntu虚拟机，一台作为NameNode，另外两台作为DataNode。各项配置以及版本信息如下：

```
操作系统：ubuntu 16.04.2
Hadoop：2.7.7
  hadoop依赖：
    jdk：1.8
Hive：1.2.2
  Hive依赖：
    mysql：5.7.28
    mysql jdbc：5.1.48
Hue：3.11.0
  Hue依赖：
    以上全是
    python：2.7.12
    git：2.7.4
    maven：3.3.9
```

4.1.2 数据库设计

项目所有需求围绕电影信息展开，并以查询为主要目的，所以在设计数据库表结构时尽量以优化查询为首要目标，因此可以选择进行适量的冗余存储以便查询，结合实体建模与维度建模的方式，最终设计表结构如下。

```
================================================
                TABLES
================================================
actor
actor_actor
actor_movie
director
director_actor
director_director
director_movie
genre
genre_movie
movie
studio
studio_actor
studio_director
studio_movie
user
user_movie

================================================
                ACTOR TABLE
================================================
actor_id        int
actor_name      varchar(50)
staring_num             int
supporting_num          int
acting_num              int

================================================
              ACTOR_ACTOR TABLE
================================================
actor_1_id      int
actor_2_id      int
times           int
movies          varchar(500)
```

```
================================================
              ACTOR_MOVIE TABLE
================================================
actor_id        int
movie_id        varchar(20)
staring         boolean
supporting      boolean
release_year    int
release_month           int
release_day     int
genres          varchar(200)

================================================
              DIRECTOR TABLE
================================================
director_id     int
director_name           varchar(50)
movie_num               int

================================================
            DIRECTOR_ACTOR TABLE
================================================
director_id     int
actor_id        int
times           int
movies          varchar(500)

================================================
          DIRECTOR_DIRECTOR TABLE
================================================
director_1_id   int
director_2_id   int
times           int
movies          varchar(500)
```

DIRECTOR_MOVIE TABLE

director_id	int
movie_id	varchar(20)
act_in	boolean
release_year	int
release_month	int
release_day	int
genres	varchar(200)

GENRE TABLE

genre_name	varchar(50)
movie_num	int

GENRE_MOVIE TABLE

movie_id	varchar(20)
genre_name	varchar(50)

MOVIE TABLE

movie_id	varchar(20)
title	varchar(200)
runtime	int
imdb	float
release_year	int
release_month	int
release_day	int
introduction	string
formats	varchar(200)
studio_id	int
average_rating	float
rating_num	int
rate_5	float
rate_4	float
rate_3	float
rate_2	float
rate_1	float
genres	varchar(200)
directors	varchar(200)
staring_actors	varchar(300)
supporting_actors	varchar(300)
acting_actors	varchar(300)

STUDIO TABLE

studio_id	int
studio_name	varchar(50)
movie_num	int

STUDIO_ACTOR TABLE

studio_id	int
actor_id	int
times	int
movies	varchar(500)

STUDIO_DIRECTOR TABLE

studio_id	int
director_id	int
times	int
movies	varchar(500)

STUDIO_MOVIE TABLE

studio_id	int
movie_id	varchar(20)
release_year	int
release_month	int
release_day	int
genres	varchar(200)

USER TABLE

user_id	varchar(20)
user_name	varchar(100)

USER_MOVIE TABLE

user_id	varchar(20)
movie_id	varchar(20)
score	float
timestp	varchar(50)
summary	varchar(200)
content	string
like_num	int
unlike_num	int

4.2 数据存储

向Hive中导入数据有五种方式可供选择：

1. 本地文件导入到Hive表；
2. Hive表导入到Hive表;
3. HDFS文件导入到Hive表;
4. 创建表的过程中从其他表导入;
5. 通过sqoop将mysql库导入到Hive表；

考虑到整个ETL处理数据的流程，最终我们所处理得到的数据是以csv和txt文件格式存储的，因此选择从HSFS文件系统导入到Hive表中的方法。

首先建立起表结构，同时约定表中个字段之间的分隔符为'\t'，将txt文件通过Hue从本地导入到HDFS文件系统中，再通过Hue将HDFS文件系统中的txt文件导入到各个表中，由于Hive支持Textfile的数据存储格式，所以此种方法导入数据的速度只取决于从本地上传txt文件到HDFS文件系统中的速度，导入速度非常可观。

4.3 存储优化工作

经查阅文献资料发现，Hive提供了多种文件存储的格式，常见的有TextFile，Parquet，ORC，Sequence，RC，AVRO等。其中Textfile文件存储格式未经压缩，会占用大量存储空间，并且查询速度较慢，而ORC文件存储格式的压缩比与查询速度表现都比较良好，所以我们选择以ORC文件格式存储数据，因此将上述以TextFile文件格式存储的数据库转换为ORC存储格式。

新建另一个数据库，设定其中所有表的存储格式为ORC，通过
insert overwrite table orc_database_name.table_name select * from text_database_name.table_name;
命令从Textflie文件格式的表中导入数据到ORC文件格式的表中。

以下是两种存储格式的数据库信息的对比。

TEXTFile

Summary for /user/hive/warehouse/movie_orc.db

Disk space consumed	5.2556 GB
Bytes used	2.6278 GB
Namespace quota	Not available.
Disk space quota	Not available.
Number of directories	19
Number of files	40

ORC

Summary for /user/hive/warehouse/movie.db

Disk space consumed	12.579 GB
Bytes used	6.2897 GB
Namespace quota	Not available.
Disk space quota	Not available.
Number of directories	17
Number of files	16

通过对比可以看出，空间压缩了将近60%，优化效果非常好。

4.4 存储优化

我们主要从四个方面对Hive数据仓库进行了优化工作。

4.4.1 表结构优化

表结构优化思路体现在4.1.3的数据库设计中

4.4.2 文件存储格式优化

查阅文献得知，改变Hive的文件存储格式能够大大提升查询优化速度，优化思路体现在4.2存储优化中，以下给出TEXTFile与ORC两种文件格式存储的数据库的查询速度比较。

查询某电影的所有评论用户

执行语句：SELECT user_id FROM user_movie WHERE movie_id = 'B003AI2VGA'

查询时间：

TEXTFile	ORC
2min 38s 850mse	50s 500mse

查询评分等于x的评论信息

执行语句：SELECT count(*) FROM user_movie WHERE score = '3'

查询时间：

TEXTFile	ORC
3min 26s 410mse	33s 380mse

优化效果十分明显。

4.4.3 索引优化

ORC文件格式内置多种轻量级索引，已经能提供很好的查询性能优化，为验证建立密集索引是否会提升查询性能，设计查询样例：查询评分等于x的评论信息

执行语句：SELECT count(*) FROM user_movie WHERE score = '3'

查询时间：

无索引	有索引
33s 380mse	46s 400mse

发现建立索引反而会降低查询速度，推测可能与ORC文件的压缩方式与存储方式有关。

4.4.4 调节参数优化

在hadoop作业执行过程中，job执行速度更多的是局限于I/O，而不是受制于CPU。如果是这样，通过文件压缩可以提高hadoop性能。

```
set hive.exec.compress.output=true;
set mapred.output.compress=true;
set mapred.output.compression.codec=org.apache.hadoop.io.compress.GzipCodec;
set io.compression.codecs=org.apache.hadoop.io.compress.GzipCodec;
```

4.4.5 SQL查询优化

1. 是否执行MapReduce任务。对于Hive，其会将大部分的SQL查询转换成MapReduce任务，而MapReduce具有高延迟的特性，其JVM的启动过程以及Map和Reduce的过程都会耗费一些时间，所以其具有一个查询时间的下限，所以对于轻量级的数据查询，选择不进行MaoReduce任务会大大优化查询时间。而对于Hive来说简单的SELECT查询是默认不会启动MapReduce任务的，所以对于轻量级数据，尽量采用简单的SELECT完成。
2. 对于Hive的SQL语句，由于其分布式执行MapReduce的特性，所以有很多可以加快查询优化速度的SQL语句设计技巧，例如JOIN时小表放在左边，大表放在右边；避免出现数据倾斜；避免使用count（distinct）操作，先用groupby去重，再用count计算；hive支持in的子查询写法，尽量用left semi join代替；多表join时使用相同的连接键等等。

4.5 Hive数据仓库特点分析

1. Hive是Hadoop上的数据仓库工具，处理的是结构化数据。其定位是数据仓库，适用于实时性要求不高的场合。因为Hive会将大部分的SQL查询转换成MapReduce任务，而MapReduce具有高延迟的特性，其JVM的启动过程以及Map和Reduce的过程都会耗费一些时间，所以其具有一个查询时间的下限，无法很快的应对实时性查询。
2. Hive主要用来进行数据分析，对于分析统计海量数据的执行时间而言，Hive的执行延迟已经不能成为它的缺点，MapReduce的算法使得分布式节点的性能数量成为执行时间的主要限制因素。
3. Hive由于是搭建在分布式文件系统的基础上，所以具有很好的扩展性和容错性。

5 图数据库

5.1 模型设计规范

1. 并不存储Transform阶段为关系型数据库生成的大量冗余的数据（如演员-导演表），而是只存储节点和边有用的信息。这样可以最大利用图数据库对关系查询特化的性能，并且方便后续符合规范的数据的导入。
2. 对于一个节点只设置一个标签（查询时使用多个标签将触发Cypher运行了额外的 hasLabel 过滤器，会降低查询效率）。
3. 对于电影的一些非一对一（如Studio、Genre等）的属性，考虑将其作为节点存储，这样会节省存储空间且大幅提高某些查询模式（如多跳查询）的效率。
4. 其余设计上的细节将在存储优化部分介绍。导入数据的具体语句详见Github仓库：https://github.com/herobrine1010/Repository-for-neo4j-demo/blob/master/insert_data.cypher。以下是图数据库的存储模型：

5.2 优化措施

1. 使用单项关系。

在导入数据并生成关系时仅使用单向关系。在Neo4j中，遍历关系的任何一个方向所需的时间是相同的，即方向可以被完全忽略。本数据集下，所有的逻辑关系用单向关系已经足够可以表示，没有必要同时创建两个方向的关系。

该部分没有实验数据。

2. 通过为单个标签的节点设计与其他节点的不同关系代替设置多个标签

在查询语句的时候，给尾节点定义标签会强制Cypher使用标签查询。设置不同的关系可以避免匹配过多的标签，从而更方便快速的遍历整个图，提高整体性能。

考虑如下的查询：查找同时导演过和演出过某部电影的人的总数。

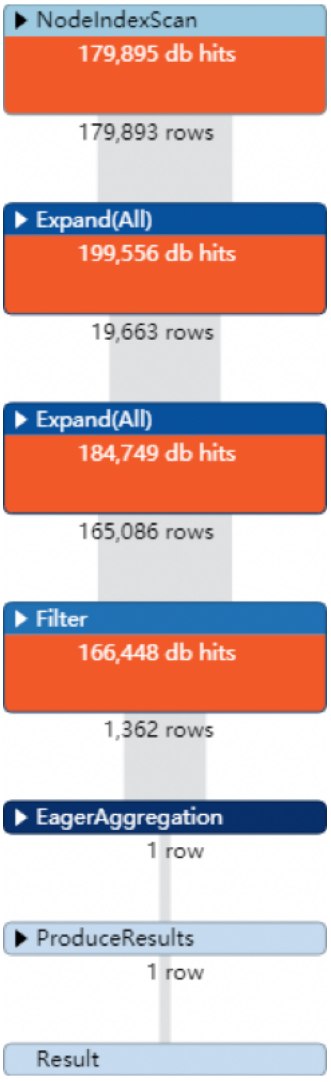
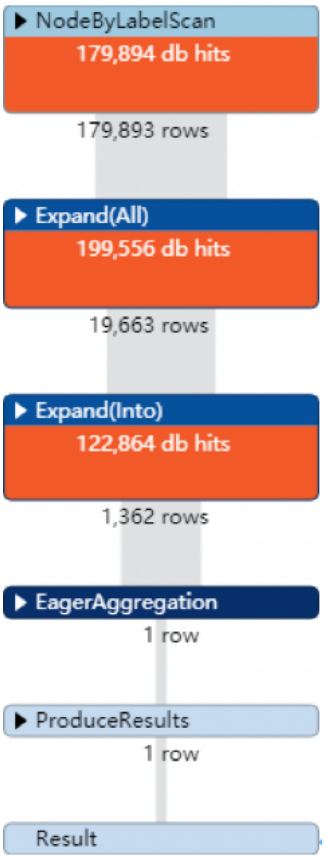
优化前，在数据导入阶段为同时具有导演-电影和演出-电影的人赋予Director和Actor标签。由于无法确定实际意义上的”同一个人”，只能通过两个节点的名字进行匹配（ETL阶段已约定相同名字的认为是同一个人）。

查询语句:

MATCH (p1:Director)-[r:DIRECTS]->(m)<-[r2:ACTS_IN]-(p2:Actor) WHERE p1.director_name=p2.actor_name RETURN COUNT(*)

优化后，在数据导入阶段先导入导演-电影关系（同时CREATE Person节点），再导入演员-电影关系（同时MERGE Person节点）。这样，查询语句可以写成:

MATCH (p:Person)-[r:DIRECTS]->(m)<-[r:ACTS_IN]-(p) RETURN COUNT(*)

	优化前	优化后
查询时间（ms）	1908	482
查询方案		

3. 采用多个关系代替一个关系上的属性

考虑如下的查询：查找10个同时主演过并且参演过一些电影的人。

优化前，设计一个关系ACTS_IN代表演员对电影的演出关系，并用ACTS_IN的变量role表示演员在电影的角色。由于生数据中该关系主演和参演是完全组合（即一个演员同时可以是或不是参演或主演）关系，创建时使用了0,1,2,3来分别表示每种情况。

查询语句：

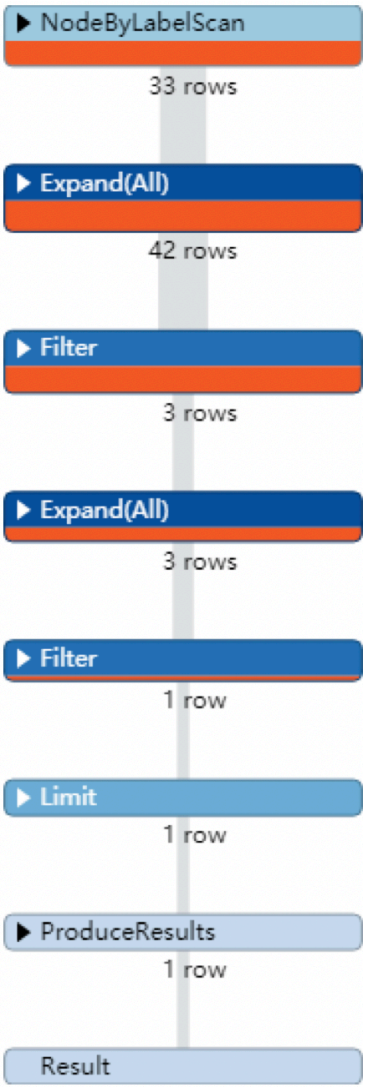
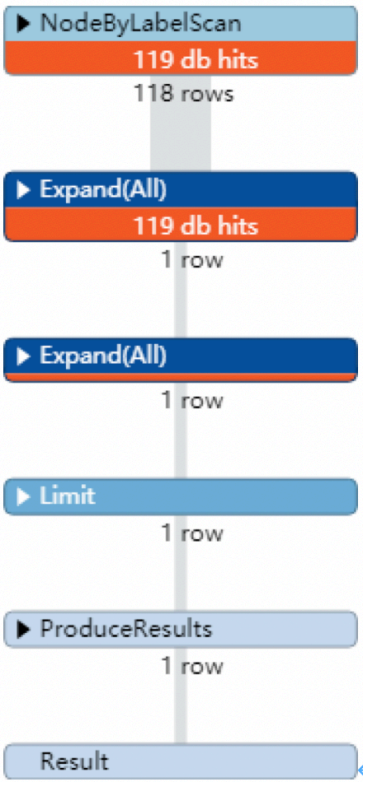
EXPLAIN MATCH (m1)<-[ACTS_IN{role:2}]-(p:Person)-[r2:ACTS_IN{role:1}]->(m2) RETURN p LIMIT 10

优化后，设计两个关系STARS和SUPPORTS分别代表演员对电影的主演和参演关系（基于上面的优化已考虑将主演和参演设计为同一节点、同一标签）。

针对目标查询需求，写出查询语句：

EXPLAIN MATCH (m1)<-[r1:STARS]-(p:Person)-[r2:SUPPORTS]->(m2) RETURN p LIMIT 10

在查询时，该方案跳过了对属性的查询，省略了filter阶段。

	优化前	优化后
查询时间（ms）	39	*
查询方案		

（* 在最终的模型中该优化措施并没有被实施，因为需要体现同时主演和参演的逻辑关系）

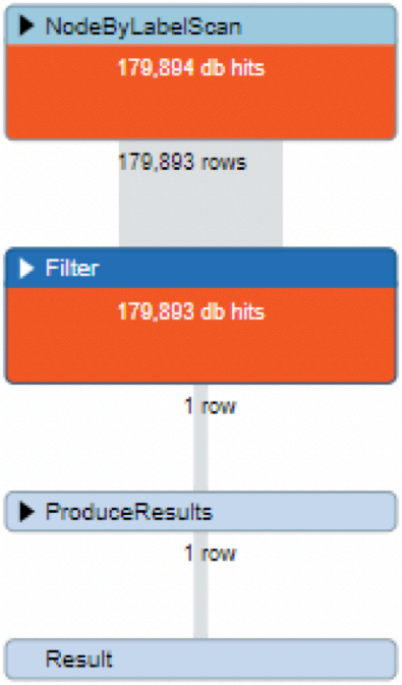
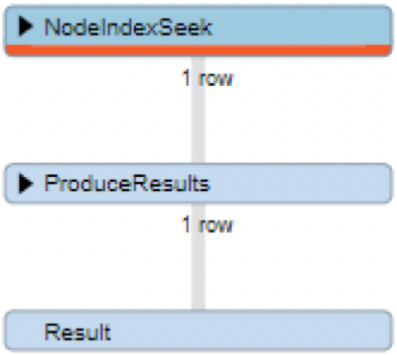
4. 建立索引

和关系型数据库相似，图数据库在执行一条Cyhper语句的时候，默认的方式是根据搜索条件按标签进行扫描，遇到匹配条件的就加入搜索结果集合。如果对某一字段增加索引，查询时就会先去索引列表中一次定位到特定值的行数，大大减少遍历匹配的行数，所以能明显增加查询的速度。对于Neo4j，即使不指定查询使用的索引，查询时也会自动对索引使用索引。

1. 考虑如下查询：按查找某个演员（或导演），返回节点。可以先写出查询语句：

   MATCH (p:Person{person_name:'Rex Ingram'}) return p

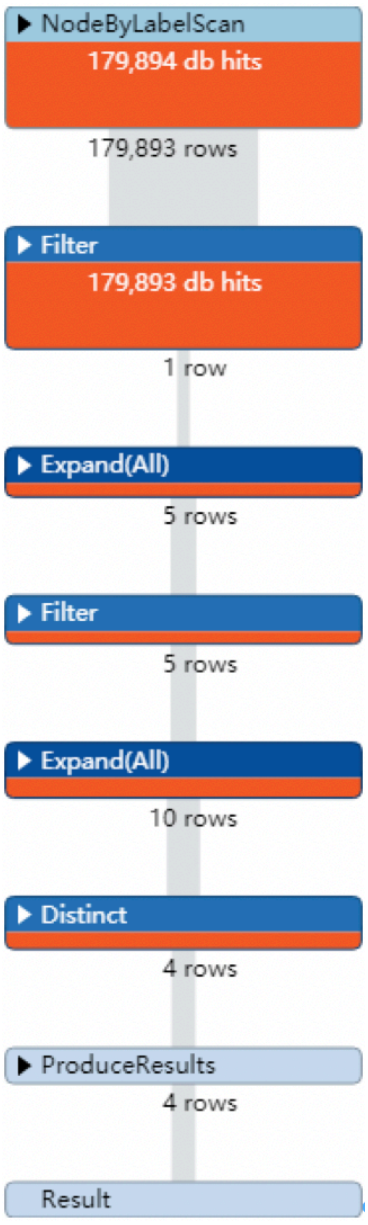
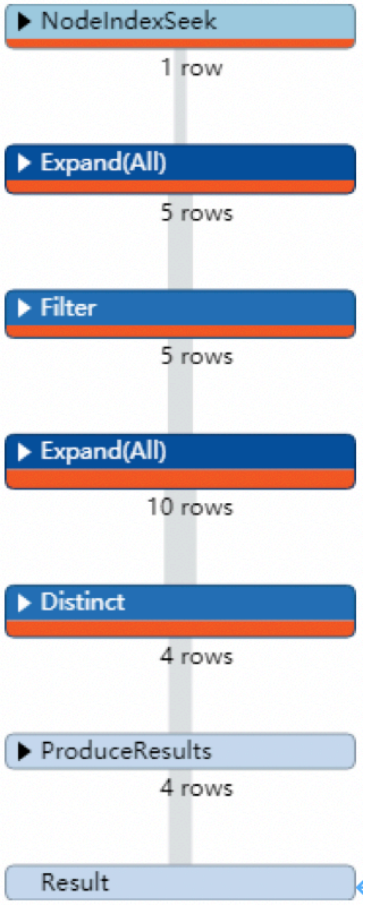
   直接使用CREATE INDEX ON :Person(person_name)创建索引进行优化。

	优化前	优化后
查询时间（ms）	393	1
查询方案		

2. 考虑如下查询：查找某个导演导演过的所有电影的所有风格。查询语句：

PROFILE MATCH (p:Person{person_name:'Adam Green'})
-[r:DIRECTS]->(m:Movie)-[r2:IS_GENRE]->(g) return distinct g.genre_name

直接使用CREATE INDEX ON :Person(person_name)创建索引进行优化。

	优化前	优化后
查询时间（ms）	327	15
查询方案		

5.3 数据库特点分析

图数据库在业务上适用于处理大量复杂的、相互连接的、低结构化的数据，它可以规避关系型数据库中的大规模的表连接；且本身查询邻近节点的机制并不受数据量的影响，其查询性能总是比较高。在本项目中，该存储模型非常适用于查询多节点间的关系（尤其是多跳关系），如查询两个导演之间多跳关系的最短路径等；同时因为不依赖冗余存储，在一些复杂查询中可以更高效率地进行关系型数据库需要联表的查询。

6 三种数据库常用场景查询时间对比分析

查询条件	关系型数据库（ms）	分布式数据库（ms）	图数据库（ms）
某一年有多少电影（2000）	47	318	24
某一年某一月有多少电影（2000-03）	234	311	337
某一年某一季度有多少电影（2000-01——2000-03）	225	474	66
某一天有多少电影（2000-05-03）	265	273	328
按电影名称进行查询（TheColdWar）	16	355	3
按导演名称查询电影数（Adrian Grunberg）	<0.1	323	442
按演员名称查询主演电影数（Marlon Brando）	<0.1	241	6
按演员名称查询参演电影数（Marlon Brando）	<0.1	314	2
按演员名称查询经常合作的演员（Marlon Brando）	<0.1	59,715	182
按导演名称查询经常合作的演员（Marlon Brando）	<0.1	52,749	15
按电影类别查询电影数（Action）	<0.1	221	3
查询评分X分以上的电影数（X=3）	16	303	1,993
查询用户评价中有正面评价的电影	4,276	122,649	50,414

7 团队分工

卜滴（25%）：数据仓库系统、数据爬取、数据清洗

刘一默（25%）：关系型数据库、数据爬取、数据清洗

吕雪飞（25%）：分布式数据库、数据爬取、数据清洗

李雨龙（25%）：图数据库、数据爬取、数据清洗