|  |
| --- |
|  |
| 数据仓库期末项目文档 |

|  |
| --- |
| 1751324 贺思奇1752577 杨 宇 1753001 魏敬杰  2019/12/21 |

目录

[1 项目概述 3](#_Toc27912956)

[2 数据获取与处理 3](#_Toc27912957)

[2.1 评论数据 3](#_Toc27912958)

[2.1.1 数据获取 3](#_Toc27912959)

[2.1.2 数据清洗 4](#_Toc27912960)

[2.1.3 数据结果 5](#_Toc27912961)

[2.2 电影数据 5](#_Toc27912962)

[2.2.1 数据获取 5](#_Toc27912963)

[2.2.2 数据清洗 7](#_Toc27912964)

[2.2.3 数据结果 8](#_Toc27912965)

[2.3 数据预处理 10](#_Toc27912966)

[3 数据模型设计 11](#_Toc27912967)

[3.1 数据ER图设计 11](#_Toc27912968)

[3.2 逻辑数据模型 13](#_Toc27912969)

[3.3 物理数据模型（关系型/分布式） 13](#_Toc27912970)

[3.4 物理存储模型（图数据库） 15](#_Toc27912971)

[4 展示界面 16](#_Toc27912972)

[5 示例SQL语句 16](#_Toc27912973)

[5.1 按发布时间查询统计 16](#_Toc27912974)

[5.2 按电影名称进行查询统计 17](#_Toc27912975)

[5.3 按照导演进行查询统计 18](#_Toc27912976)

[5.4 按照演员进行查询统计 19](#_Toc27912977)

[5.5 按照合作关系进行查询统计 19](#_Toc27912978)

[5.6 按照电影类别进行查询统计 20](#_Toc27912979)

[5.7 按照评分进行查询统计 20](#_Toc27912980)

[5.8 组合查询与统计 20](#_Toc27912981)

[6 总结 21](#_Toc27912982)

# 1 项目概述

近年来，电影、DVD、歌剧等影像作品成为了人们生活中不可或缺的休闲消遣之物，自电影于20世纪初成规模时起，在荧幕上出现过的影视作品数不胜数，让想要从浩如烟海的影视作品中筛选出自己需要的内容人们也变得举步维艰。

在数据仓库课程中，我们学习了从获取数据到对数据进行清洗、处理、提炼并将其存放入数据仓库中进行数据分析的知识。

我们的数据来源分两种：第一种是从Amazon网站爬取的近年来的电影数据约20万条，其中包含了电影的ASIN号、名称、演员、导演等信息，以及电影的类型、评分等数据。另一种是以.TXT格式的文本文件作为数据来源的电影评论信息，每条评论对应了一个Amazon用户和一个电影，评论包括了打分和评论内容等信息，共计七百余万条，文本大小超过9G。

对于Amazon网站爬取的信息，我们将其存储在TXT文本文件中，通过ETL和数据清洗的过程，提炼出我们需要的数据并将其以CSV文件格式存储。然后根据具体的业务内容和我们爬取到的信息，设计了数据仓库的ER图和逻辑数据模型。随后根据我们的逻辑需要，对电影数据进行了二次处理，按照逻辑内容将电影内容、演职员的内容和电影与演职员的关系分别生成了CSV文件。

对于电影评论的数据，我们使用Kettle和JAVA脚本两种方式分别对其进行了处理，生成了电影评论的数据表。

我们使用了三种数据库对所有的数据来源进行存储。关系型数据库上我们采用了传统的Oracle数据库，并使用了Neo4j作为图数据库，分布式数据库则采用了布置在Hadoop集群上的Hive数据库。

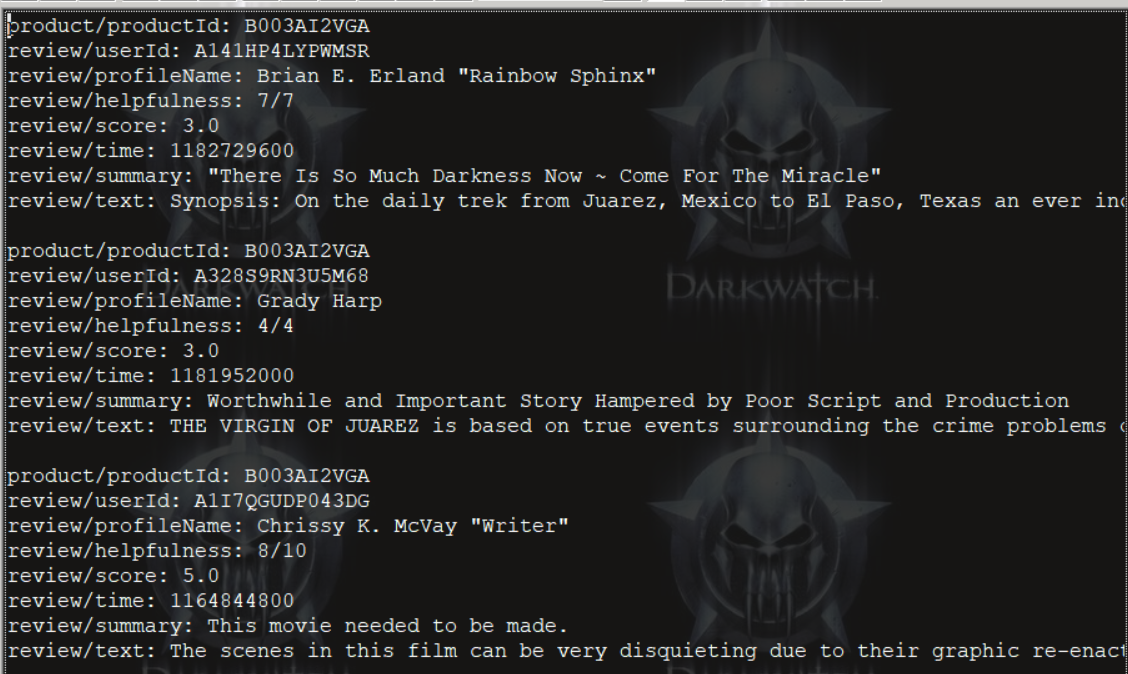
对此，我们编写了一个前端界面来对各个数据库的查询结果进行展示，并编写了一些示例SQL，来对我们的数据内容进行展示性查询。

# 2 数据获取与处理

## 2.1 评论数据

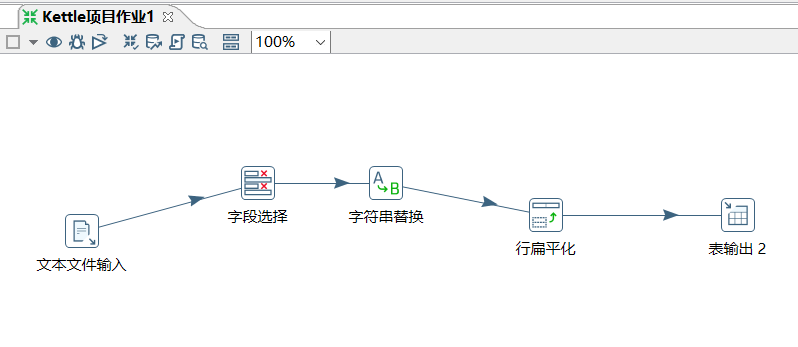
### 2.1.1 数据获取

所有的电影评论数据来自于从课程第二次作业获取的movies.txt文件，该文件大小约9G，包含了Amazon网站中所有的对电影的评论，其中包含了每条评论针对的电影的ASIN、评论用户的ID、评论内容、打分、总结等信息。



### 2.1.2 数据清洗

我们对从文本文件中获取的数据初步分为八个字段，即：productId, userId, profileName, helpfulness, score, time, summary, text进行数据获取。

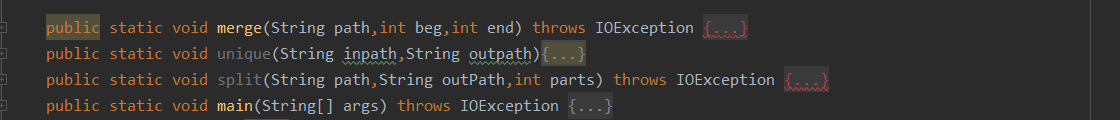


在Kettle的处理中，对数据的清洗分为五个步骤：文本文件输入、字段选择、字符串替换、行扁平化、表输出。

通过文本文件输入，将TXT中数据以换行符作为分隔读入，并利用字段选择将其归于各自的列。随后通过“字符串替换”，将每一行的开头替换为空字符，利用行扁平化将其分至各列，并进行对于数据库的表输出，我们同时还输出了一份CSV文件备用。

在Kettle输出的最后效果中，出现了许多“数据错位”的现象，即某一列的头对上了另一列的内容。因此，我们手动编写JAVA脚本对其进行了数据清洗。

在我们的JAVA脚本中，对该文本文件进行了以下的处理：



评论数据相对规整，有些数据错乱的地方直接删除，最后仅有两行出现了数据错乱的现象，其中为方便数据清洗，将评分的小数形式转变为整数形式进行清理，即将3.5分变为35分，另外将helpness的分子和分母拆开，单独两列进行存储。

### 2.1.3 数据结果

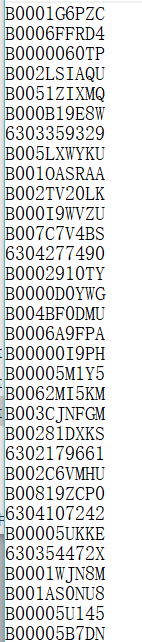
在使用JAVA脚本对TXT文件进行处理后，我们得到了如以下类型的电影评论的原始数据：

（\*贴一下最早的电影评论处理好的CSV的图）

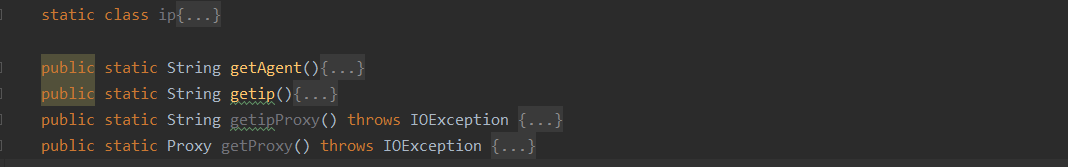
## 2.2 电影数据

### 2.2.1 数据获取

在得到的电影评论的数据集中，我们对每条评论数据的ASIN号进行提取并去重，以此得到这些ASIN号关联的所有电影。



随后，以这些ASIN号作为索引，访问Amazon网站下该电影对应的信息页面，并从中爬取Amazon网站下这些电影的详细信息。因此可以形成每个电影和每个评论对应的连接。我们将这些distinct的ASIN号提取出来并爬取电影内容。



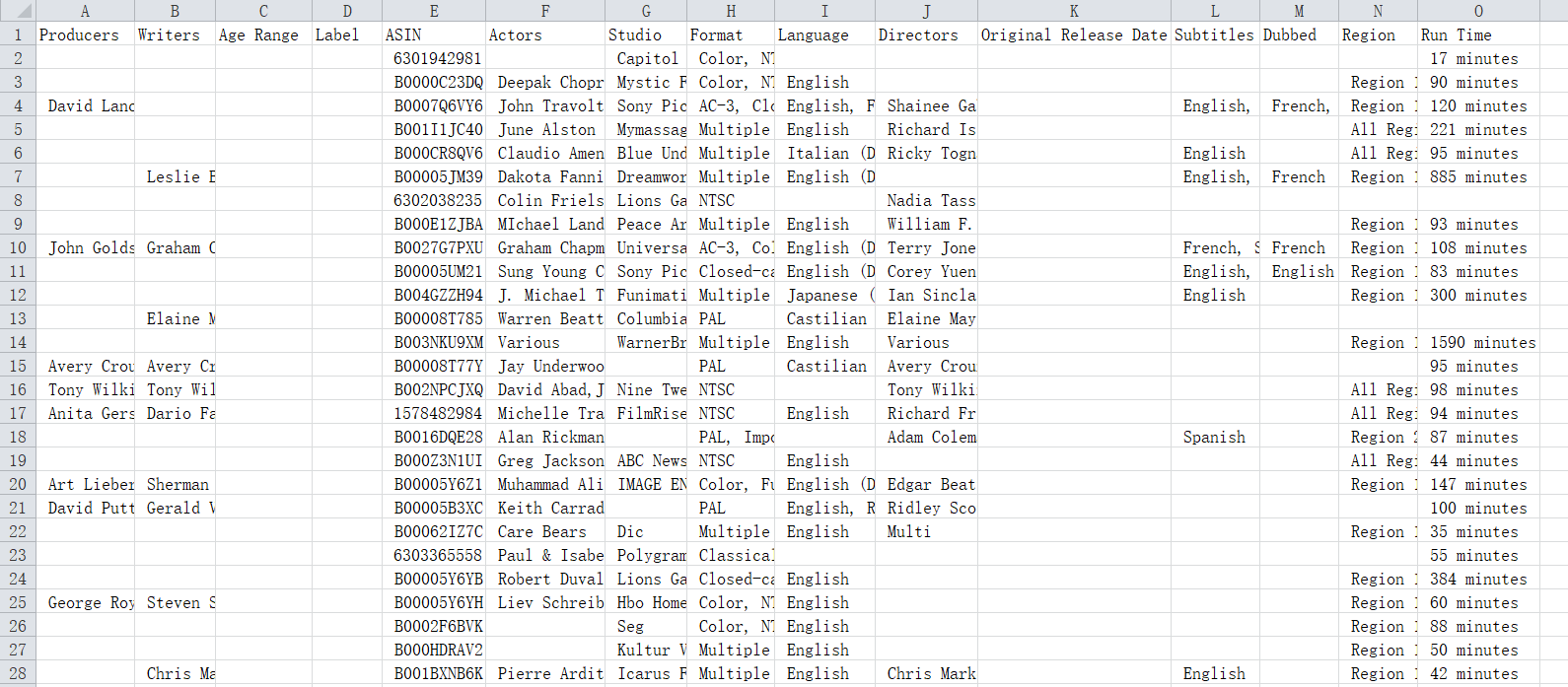
爬虫使用了浏览器代理头和http代理头等相关头部参数来尽量减少被屏蔽的次数。同时，采取了动态时间间隔的处理方式，以二进制指数退避的方式来进行尝试。

以下是爬取到的原始数据内容：



### 2.2.2 数据清洗

在获取到电影数据后，我们使用自己编写的JAVA脚本对其进行数据的清洗。首先是提取我们获取到的所有字段，并将每个字段分隔开，按每部电影一行，每个字段一列的方式将获取的文本文件输出为CSV文件。



这是我们第一次得到的CSV文件，其中充满了许多无用的字段、或是几乎都为空值的没有获取到的字段。因此，我们重新编写了清洗脚本，按照我们在逻辑数据模型设计中所列出的字段来获取数据，并丢弃了那些基本为空的无用信息，使得我们的最终数据趋向于我们逻辑数据模型的格式。

我们的电影数据ETL脚本主要进行了以下几方面的工作：

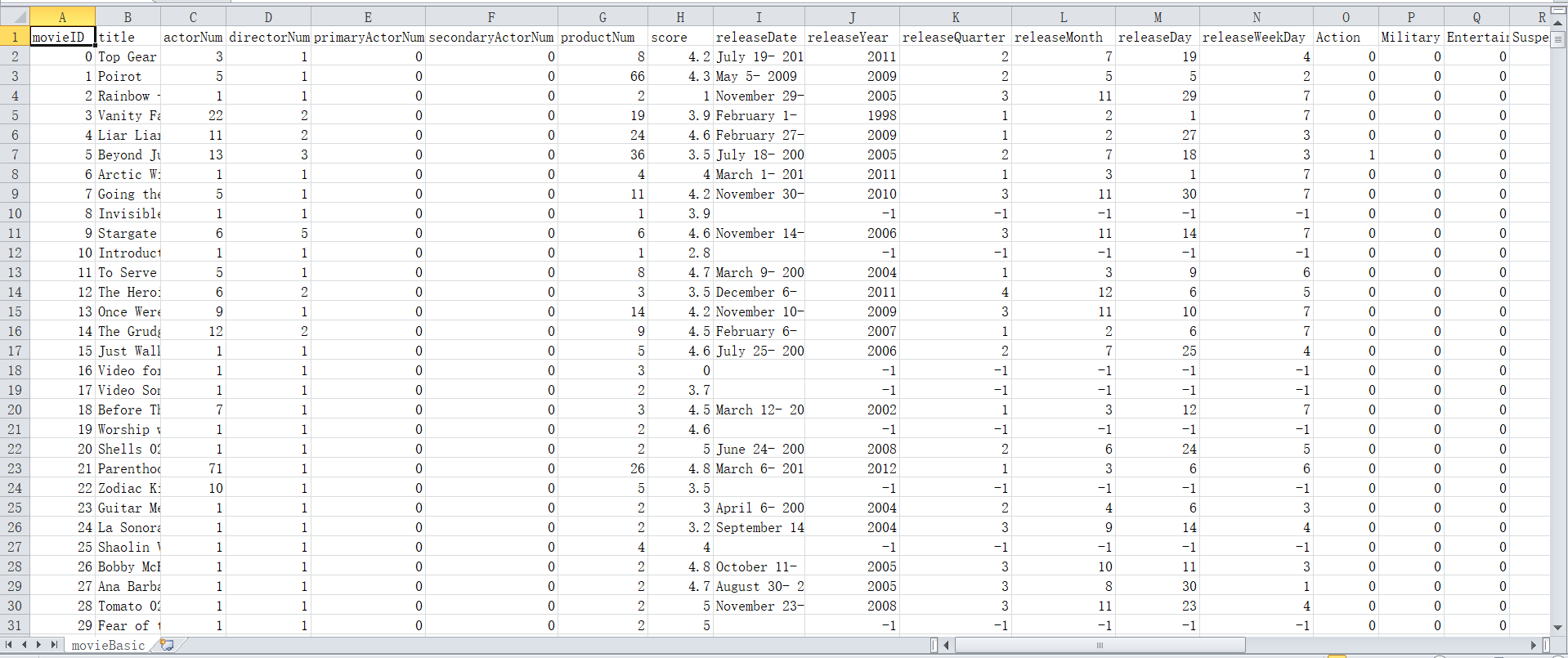
对于人名方面，我们将爬取数据中的所有乱码（即所有空格、短横线、字母、数字、点以外的字符）替换为空字符。去除了首尾的空格，并对长度不大于3的内容进行过滤筛选，最后进行去重，得到所有人名集合。

我们脚本的核心内容，是将每一部ASIN对应的电影进行相似度辨识，将可能是一系列的电影归于一条Movie。同样地，我们首先将所有乱码替换为为空字符，去除首尾空格并进行短长度的过滤。此外，对于每一部电影名，我们去除了各种括号以及其中的描述文字，以消除不同版本间的差异。然后我们获取了每部电影的导演列表和演员列表，通过“枚举+编辑距离”来统计其导演相似率与演员相似率。我们还获取了电影时长，通过电影的相似率与绝对时长差以及电影名称的相似率（编辑距离）来判断两部电影是否是同一系列的电影，并统计出电影与产品的关系。

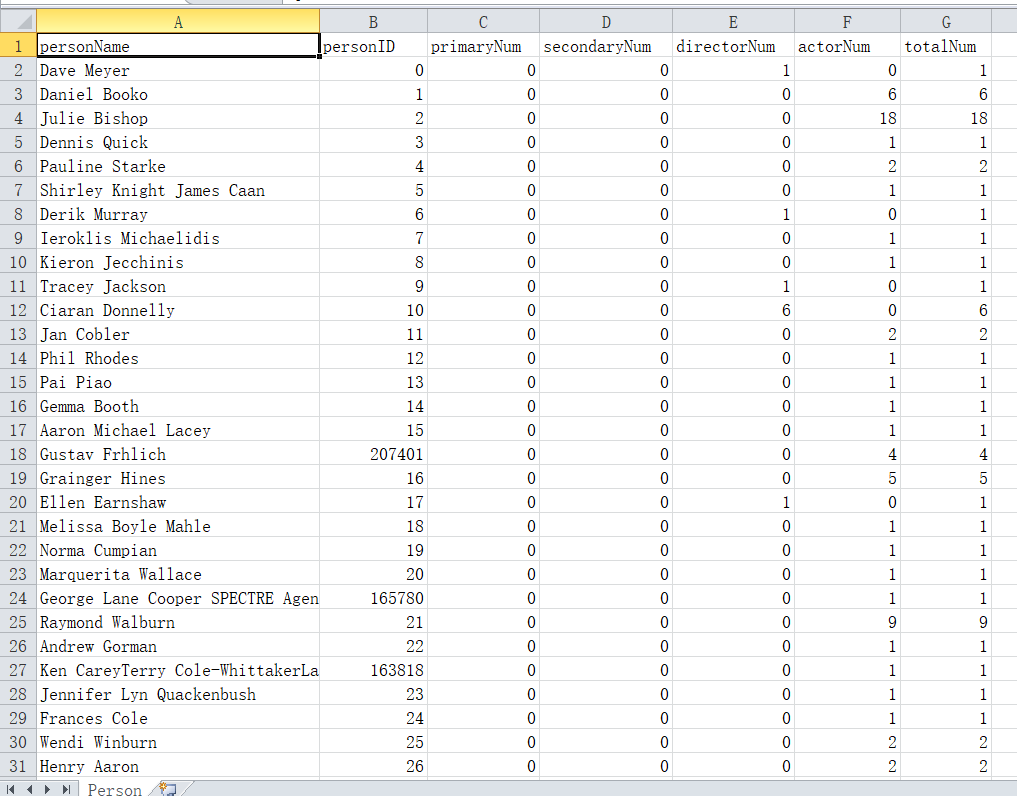
### 2.2.3 数据结果

我们将实际得到的数据根据逻辑数据模型的需求分为以下几个种类：电影基本信息MovieBasic、电影演职人员信息Person、电影与演职人员关系PersonMovieRelation、同类电影数据Product。其中，我们还根据从评论结果数据中得到的每个评论的打分，通过以ASIN号分组的方式统计出每部电影的评分并进行汇总，也放入了MovieBasic中。

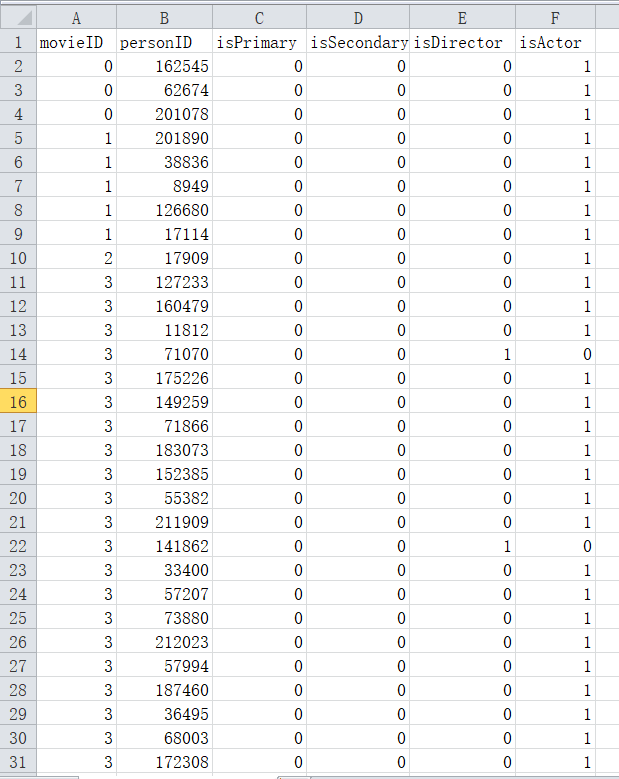
MovieBasic.csv:



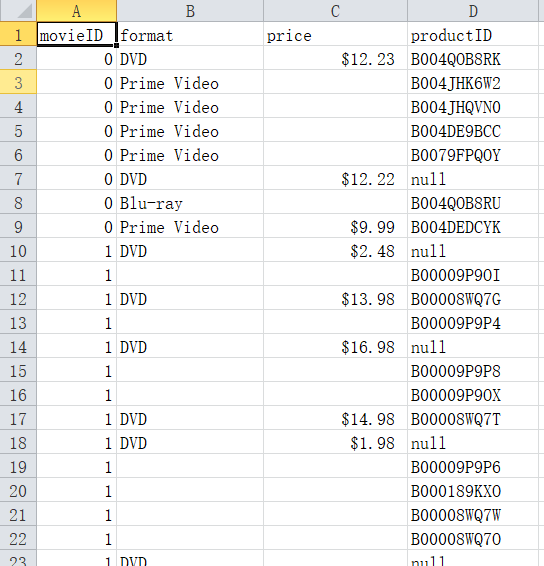
Person.csv:



PersonMovieRelation.csv:



Product.csv:



## 2.3 数据预处理

为了方便进行业务有关的查询，我们将一些具有多个值的列单独抽取了出来，并以提高查询性能的方式进行存储。

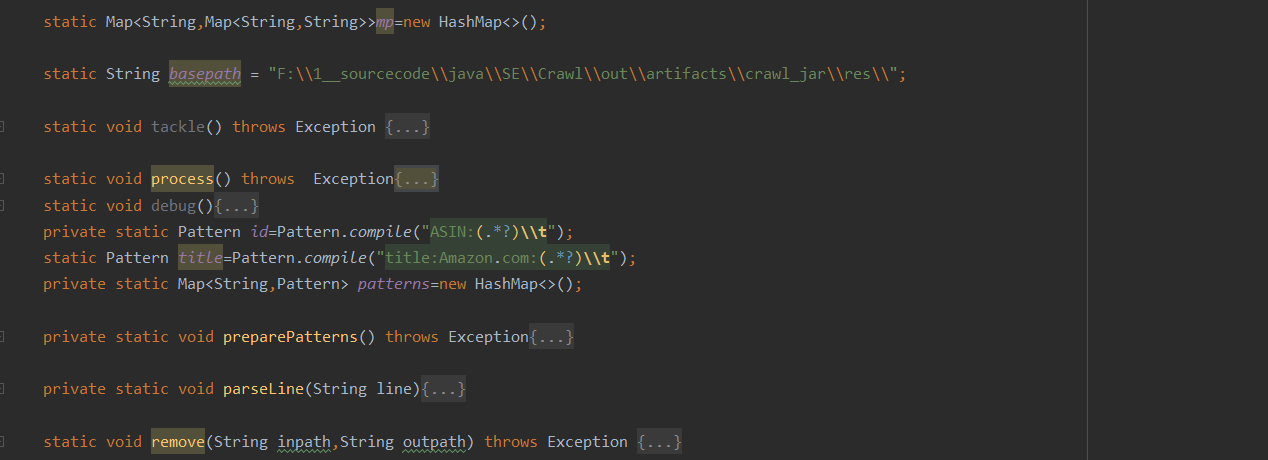
例如：我们对每个电影的类型进行了提取，以Amazon上提供的电影分类为标准，我们将其分为了共31种分类。根据其类型字段中的多种属性，我们以逗号为分隔符提取出了其电影分类，并在其具有的分类属性上标为1，其不具有的属性则为0。

在提取出电影的发行时间后，我们还通过自己编写的脚本，计算出该电影的发行年、季度、月、日和星期几的信息，并以数字的方式存储在对应字段中。

通过对每部电影的导演及演员名单中进行字段分隔分析，我们也计算出了每部电影具有的导演与演员的数量。考虑到演员分为主演和配角，我们还设置了主演数和配角数的字段，但因为我们在爬取数据过程中并没有得到相应的数据，因此这些字段的值都为默认值。

在获取电影的过程中，考虑到Amazon中对于同一部电影的不同版本甚至不同型号是以不同的ASIN号存储的，即我们认为的“同一部电影”可能具有多个不同的ASIN号，因此我们在进行数据处理时，通过一些关键字的识别将我们认为可能是同一部电影的一系列影像作品都归于一个movieID，并赋予一个统一的title。与此同时我们使用了另外一个CSV文件来记录每个movieID下包含了哪些ASIN号对应的影视作品。换言之，MovieBasic.csv文件下记录的每个movieID都是一个系列的作品，而Product.csv中则具体记录了该系列的作品对应了哪些ASIN号。

以下是我们进行数据预处理的脚本示意：



# 3 数据模型设计

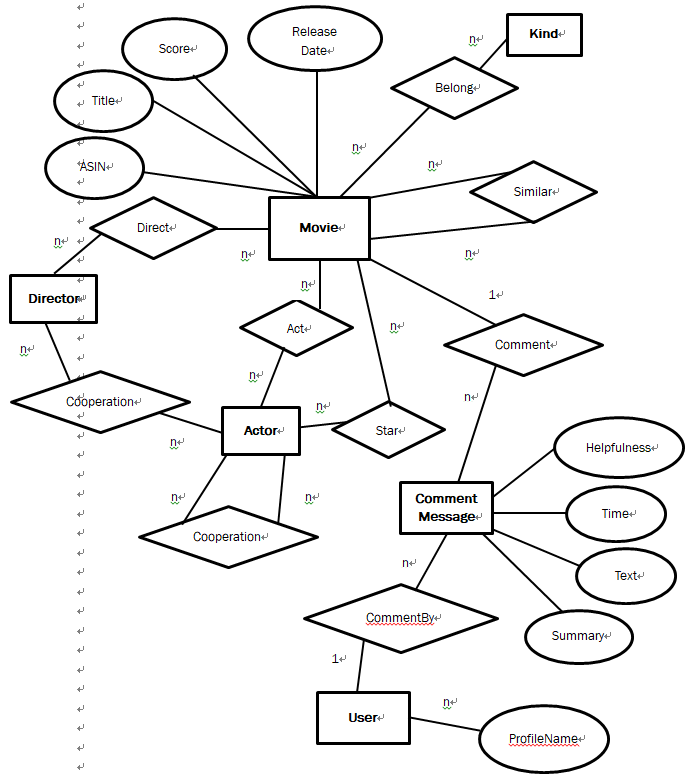
## 3.1 数据ER图设计

我们按照所得数据的逻辑关系，将数据分为多个实体集，并根据它们间的逻辑联系建立了联系集。ER图如下。

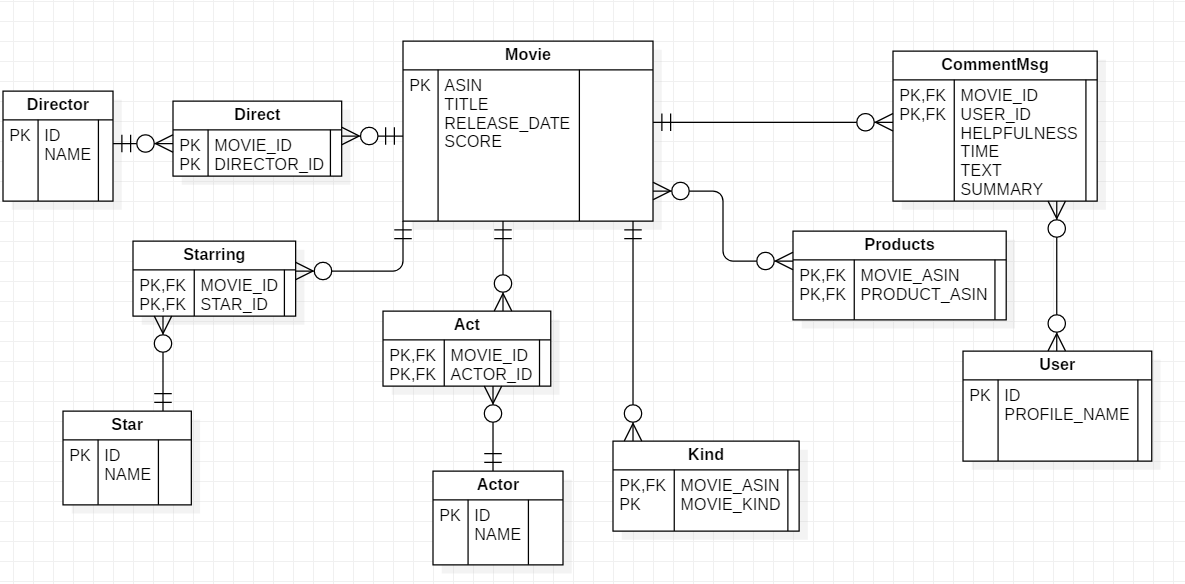
我们首先将“电影”设为一个实体集Movie，实体集中的每个实体对应的是一部具体的电影。该实体集中包含了电影的各种属性，包括其ASIN号、电影名、发行日期、评分等。其中评分是应当从评论数据中统计而来。该电影与另一个实体集Kind具有一个名为Belong的联系，Kind中的每个实体即为一种电影类型。每部电影还与自己具有一个Similar联系，该联系指明了两部可能是同一系列的电影。

演员和导演各为一个实体集，人员之间有合作关系形成的联系集，演员/导演与电影之间也具有联系，演员还有主演和参演的区别。

从文本文件中提取出的电影评论信息中我们抽取了两个实体，一个为电影评论实体，另一个则为评论用户的实体。一部电影可以有多个评论，一个用户也可以进行多次评论。

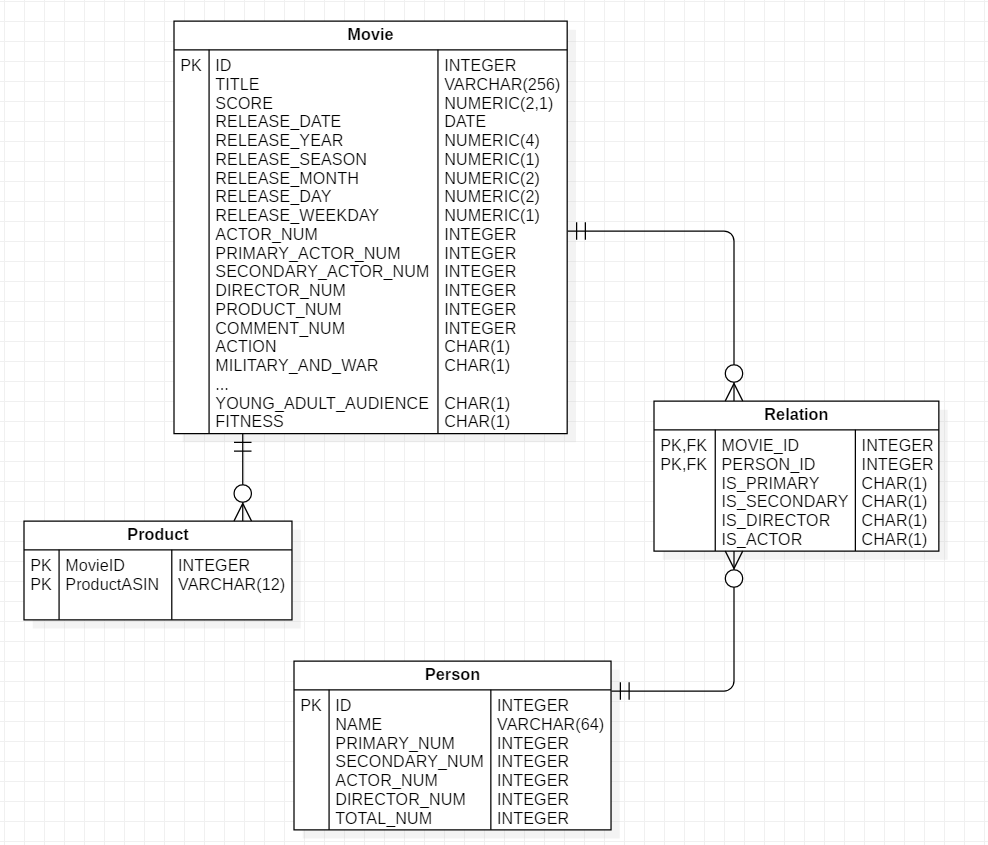


## 3.2 逻辑数据模型



在数据仓库的逻辑数据模型设计中，我们将同系列的多个电影用Products表作为其联系的存储。因为每部电影可能有多个类型，故用一个Kind表来存储每部电影的类型。而考虑到每部电影可以有多位演职员，每位演职员也可以参与多部电影，因此除了Star、Actor、Director三种表，还添加了Direct、Starring、Act三个关系表来存储这三种多对多关系。

## 3.3 物理数据模型（关系型/分布式）



在关系型和分布式数据库的物理数据模型中，为了提升查询的性能，便于添加索引，我们通过增加数据冗余的方式来简化了表关系的设定。

我们将Movie表的每一行设定为一系列电影，为其赋予自增的主键ID，每一行中除了包含电影系列名Title、评分Score、发行时间等基础属性，还添加了一些冗余属性如发行年、月、日、星期的信息进行存储，便于在查询时直接得到数据。对于演员、主演、参演、导演的数量，也分别进行了直接的存储。因此，在进行统计查询时，可直接获取总值，而不需要遍历整个表进行计算。

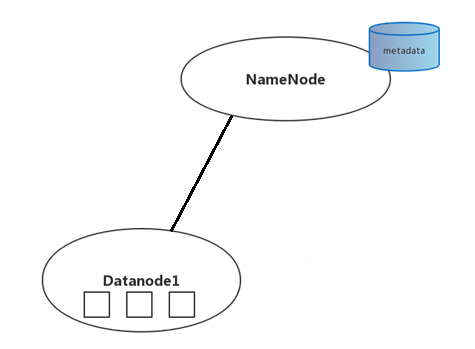
另外，考虑到每一次查询电影种类时，都需要对Kind表进行join操作，开销巨大，又因为我们的电影种类其实并非很多，因此，我们把每一个种类都作为一个属性字段放入了Movie表中，在上图中以Action, Military\_And\_War, ……, Yound\_Adult\_Audience, Fitness等列表示。每个列的值为1则代表该系列电影为该种类，为0则相反。在表示种类的每一列上（以及表示了月、日、星期的列）我们都设置了位图索引，以增加查询速度。

每系列的电影对应的具体的ASIN号的影视作品被存储在Product表中，与Movie表形成一对多的关系。

对于演职人员的存储，为了避免对不同类型人员的查询带来复杂的语句和“自导自演”情况带来的数据冗余，我们将所有类型的演职人员统一存储为Person表，赋予自增的主键ID，并添加了其执导、参演、主演、配角的电影数，避免了临时聚集查询带来的高消耗。

所有演职人员和某系列电影的关系都存储在Relation表中，以电影ID和人员ID作为复合主键。每个关系中会注明其是否是导演/演员/主演/配角，因此对于担任了多个职务的人员，也可以通过一行结果表示。

而对于每条评论，由于没有针对具体用户的查询，为了节省评论表与用户表的join带来的时间消耗，我们引入了一定的数据冗余，将评论与用户重新存回一张表中。我们将原评论数据中的Helpfulness字段理解为“认为此评测有用的用户数/对该评测作出评价的用户数”，即通常论坛中的点“赞”与“踩”的对比。我们将两个数据分开存储，即HELPFUL\_NUM与HELPFUL\_TOTAL（每个评论的打分已被我们统计至Movie表中）。



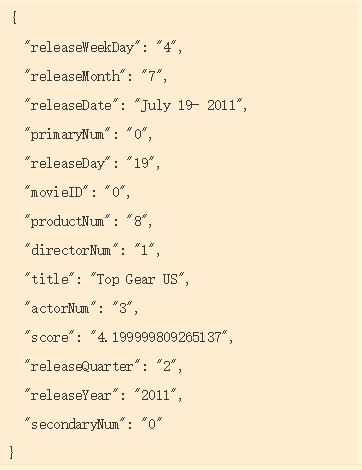
在分布式的存储上，因为我们自身机器内存的限制，我们只采用了一个数据节点，通过部署在虚拟机上的方式完成分布式节点的搭建。分布式中，我们使用了Hive数据库，并通过文本方式将已有的数据进行导入。

## 3.4 物理存储模型（图数据库）

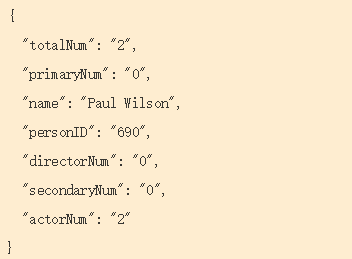
在图数据库的建模过程中，受制于我们自身的计算机处理能力的限制，我们将设计目标指向空间与效率的平衡。Neo4j数据库中，关系都是有向关系，为了节省空间，我们将双向关系转化为了有向关系，且双向关系只进行一次存储（如A-B只存储为A->B或A<-B）。

以下是我们节点结构的示例图：

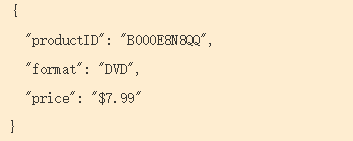
Movie：



Person：

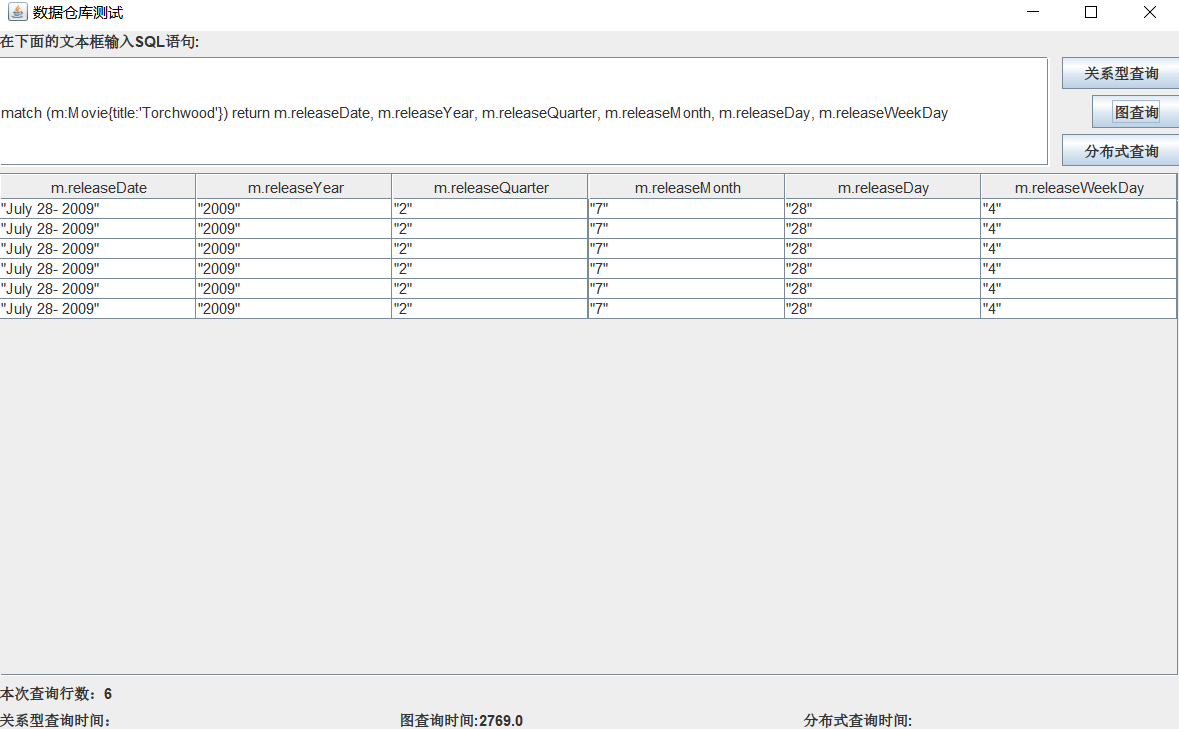


Product：



此外，在图数据库中还有五种关系：ActorIn表示了电影和参演演员的联系，ActorWith表示演员与演员的关系，DirectorIn表示电影与导演的关系，DirectorTo表示演员与导演的关系，而Include表示了同系列不同版本电影间的关系。

# 4 展示界面



我们的UI采用了JFrame框架，提供了一个SQL输入的查询框，在框中输入任意SQL查询语句，并在右侧按钮选择执行三种数据库查询中的一种，下方会显示查询结果的列名以及表单。在最下方会显示该次查询得到的行数与消耗的时间，三种数据库的时间会一起显示，以进行时间对比。

# 5 示例SQL语句

## 5.1 按发布时间查询统计

·查询电影《Torchwood》的发行时间：

关系型/分布式：

select releaseDate, releaseYear, releaseQuarter, releaseMonth, releaseDay, releaseWeekDay from DBMovie where title='Torchwood';

图数据库：

match (m:Movie{title:'Torchwood'}) return m.releaseDate, m.releaseYear, m.releaseQuarter, m.releaseMonth, m.releaseDay, m.releaseWeekDay

·查询2019年发行的所有电影名字和评分并按评分排序：

关系型/分布式：

select title, score from dbMovie where releaseYear=2019 order by score desc;

图数据库：

match(m:Movie{releaseYear:'2019'}) return m.title,m.score order by m.score

·查询2010年1月发行的所有电影名字和评分并按评分排序：

关系型/分布式：

select title, score from dbMovie where releaseYear=2010 and releaseMonth=1 order by score desc;

图数据库：

match(m:Movie{releaseYear:'2010',releaseMonth:'1'}) return m.title,m.score order by m.score desc

·查询每一年第二季度的电影数量并按数量排序：

关系型/分布式：

select releaseYear, count(MOVIEID) as MOVIE\_COUNT from dbMovie where releaseQuarter=2 group by releaseYear order by MOVIE\_COUNT desc;

图数据库：

match(m:Movie{releaseQuarter:'2'}) return m.releaseYear,count(\*) order by count(\*) desc

·查询所有周二的电影和评分，并按发布时间排序：

关系型/分布式：

select releaseDate, title, score from dbMovie where releaseWeekday=2 order by releaseYear desc, releaseMonth desc, releaseDay desc;

图数据库：

match (m:Movie{releaseWeekDay:'2'}) return m.releaseDate,m.title,m.score order by m.releaseYear desc,m.releaseMonth desc,m.releaseDay desc

## 5.2 按电影名称进行查询统计

·查询电影《The Mummy》系列的评分：

关系型/分布式：

select score from dbMovie where title='The Mummy';

图数据库：

match (m:Movie{title:'The Mummy'}) return m.score

·查询电影《The Mummy》的系列电影有多少个版本：

关系型/分布式：

select productNum from dbMovie where title='The Mummy';

图数据库：

match (m:Movie{title:'The Mummy'}) return m.productNum

·查询电影《The Mummy》系列电影的所有ASIN号：

关系型/分布式：

select productID from dbMovie, dbProduct where dbProduct.movieID=dbMovie.movieID and productID!='null' and title='The Mummy';

图数据库：

match (m:Movie{title:'The Mummy'})-[r:Include]->(p:Product) return p.productID

·查询电影《The Mummy》系列的所有演员：

关系型/分布式：

select name from dbMovie,dbMoviePerson,dbMoviePersonRelation where dbMovie.movieID=dbMoviePersonRelation.movieID and dbMoviePerson.personID=dbMoviePersonRelation.personID and title='The Mummy' and isActor=1;

图数据库：

match (p:Person)-[:ActorIn]->(m:Movie{title:'The Mummy'}) return p.name

·查询电影《The Mummy》系列的所有主演（无对应数据）：

关系型/分布式：

select name from dbMovie,dbMoviePerson,dbMoviePersonRelation where dbMovie.movieID=dbMoviePersonRelation.movieID and dbMoviePerson.personID=dbMoviePersonRelation.personID and title='The Mummy' and isPrimary=1;

## 5.3 按照导演进行查询统计

·查询导演“Happy Traum”执导过的所有电影，并按评分排序：

关系型/分布式：

select title, score from dbMovie,dbMoviePerson,dbMoviePersonRelation where dbMovie.movieID=dbMoviePersonRelation.movieID and dbMoviePerson.personID=dbMoviePersonRelation.personID and name='Happy Traum' and isDirector=1 order by score desc;

图数据库：

match (p:Person{name:'Happy Traum'})-[:DirctorIn]->(m:Movie) return m.title,m.score order by m.score desc

## 5.4 按照演员进行查询统计

·查询演员“Jackie Chan”主演的电影数量（无对应数据）：

关系型/分布式：

select primaryNum from dbMoviePerson where name='Jackie Chan';

图数据库：

match (p:Person{name:'Jackie Chan'}) return p.primaryNum

·查询演员“Jackie Chan”参演的所有电影，并按评分排序：

关系型/分布式：

select title, score from dbMovie,dbMoviePerson,dbMoviePersonRelation where dbMovie.movieID=dbMoviePersonRelation.movieID and dbMoviePerson.personID=dbMoviePersonRelation.personID and name='Jackie Chan' and isActor=1 order by score desc;

图数据库：

match (p:Person{name:'Happy Traum'})-[:ActorIn]->(m:Movie) return m.title,m.score order by m.score desc

·查询“Jackie Chan”自导自演的所有电影，优先显示最近电影：

关系型/分布式：

select title, releaseDate from dbMovie,dbMoviePerson,dbMoviePersonRelation where dbMovie.movieID=dbMoviePersonRelation.movieID and dbMoviePerson.personID=dbMoviePersonRelation.personID and name='Jackie Chan' and isActor=1 and isDirector=1 order by releaseYear desc, releaseMonth desc, releaseDay desc;

图数据库：

match (p:Person{name:'Jackie Chan'})-[:ActorIn]->(m:Movie)<-[:DirctorIn]-(p) return m.title,m.releaseDate order by m.releaseYear desc,m.releaseMonth desc,m.releaseDay desc

## 5.5 按照合作关系进行查询统计

经常合作的演员有哪些，按合作数排序

match (p1:RPerson)-[r1:RActorWith]->(p2:RPerson) with p1,p2,count(r1)as c1 match (p2)-[r2:RActorWith]->(p1) return p1.name,p2.name,c1+count(r2) limit 100

经常合作的导演和演员有哪些，按合作数排序

match (p1:Person)-[r:Director]->(p2:Person) return p1.name,p2.name,count(r) order by count(r) desc

## 5.6 按照电影类别进行查询统计

·查询有多少类型为“Comedy”的电影，并按评分排序：

关系型/分布式：

select title, score from dbMovie where Comedy=1 order by score desc;

·查询共有多少既是“Action”又是“Horror”的电影：

关系型/分布式：

select count(title) from dbMovie where Action=1 and Horror=1;

## 5.7 按照评分进行查询统计

·查询最近的评分大于4.5的电影与发布时间：

关系型/分布式：

select title, score, releaseDate from dbMovie where score>=4.5 order by releaseYear desc, releaseMonth desc, releaseDay desc;

图数据库：

match (m:Movie) where ToFloat(m.score)>=4.5 return m.title,m.score,m.releaseDate order by m.releaseYear desc,m.releaseMonth desc,m.releaseDay desc

·查询Drama电影中评分最高的电影与发布时间：

关系型/分布式：

select title, score, releaseDate from dbMovie where Drama=1 order by score desc;

## 5.8 组合查询与统计

·查询2010年以来“Jackie Chan”执导的所有Action电影，并按评分排序：

关系型/分布式：

select title, score, releaseDate from dbMovie,dbMoviePerson,dbMoviePersonRelation where dbMovie.movieID=dbMoviePersonRelation.movieID and dbMoviePerson.personID=dbMoviePersonRelation.personID and name='Jackie Chan' and Action=1 and releaseYear>=2010 and isDirector=1 order by score desc;

图数据库：

match (p:Person{name:'Jackie Chan'})-[DirctorIn]->(m:Movie) where ToInteger(m.releaseYear)>=2010 return m.title,m.score,m.releaseDate order by m.score desc

·查询所有具有自导自演影片的人物，并通过这样的影片数排序：

关系型/分布式：

select name, count(\*) as amount from dbMovie,dbMoviePerson,dbMoviePersonRelation where dbMovie.movieID=dbMoviePersonRelation.movieID and dbMoviePerson.personID=dbMoviePersonRelation.personID and isActor=1 and isDirector=1 group by name order by amount desc;

图数据库：

match (p:Person)-[r:DirctorIn]->(m:Movie)<-[r2:ActorIn]-(p) return p.name,count(\*) order by count(\*) desc

# 6 总结

本项目中，我们通过Snap的文本文件与Amazon网站爬取的电影与评论的数据，建立了基于关系型数据仓库、分布式文件型数据仓库系统以及图数据库的数据仓库系统并进行了数据比对。

我们完整地进行了数据的获取、数据的ETL和预处理，并根据数据特点和我们具体的业务需求绘制了数据ER图，建立了数据的逻辑模型与物理存储模型。此外，我们制作了展现数据的JFrame窗口，将查询内容和执行时间分别以图表和数值的方式显示在了图形界面上。

针对每种存储方式，我们进行了特定的数据结构设计，针对具体的业务需求来设计数据表，通过引入适当冗余的方式来增加数据的查询速度。通过三种不同数据仓库系统的结合，我们实现了一个在多种业务需求条件下均能够较好工作的电影数据仓库系统。