数据存储设计说明文档

报告要求内容

- 整体存储模式：关系型存储、分布式存储、图数据存储在本项目中是如何协同工作的，在每种类型的存储总分别存了哪些内容。
- 关系型存储逻辑模型：E-R图设计、数据存储模型（start schema或其他）
- 关系型存储物理模型：DDL，存储优化设计，Denormalization分别是如何做的
- 分布式文件系统存储模型（schema定义文件）及优化
- 图数据库存储模型及优化
- 数据表的test case

1. 整体存储模式

在本项目中，采用的关系型数据库是MySQL数据库，分布式存储使用Hive，图数据库使用Neo4j，为了测试各数据库的性能，每个数据库中都对所有数据进行了存储。

查询类型\时间(ms)	Mysql	Hive	Neo4j
某一年某一季度有多少电影	8	2790	38
某一电影有多少版本	12	612	30
某一导演指导多少电影	6	670	22
按人名名称查询合作最多的演员和合作次数	7	1498	15
查询某演员某年参演的电影数	5	2320	26
查询评分在x以上的电影数	818	612	75
查询某年某种类型电影数	409	1804	38

查询哪两个演员合作次数最多	5028	16200	607

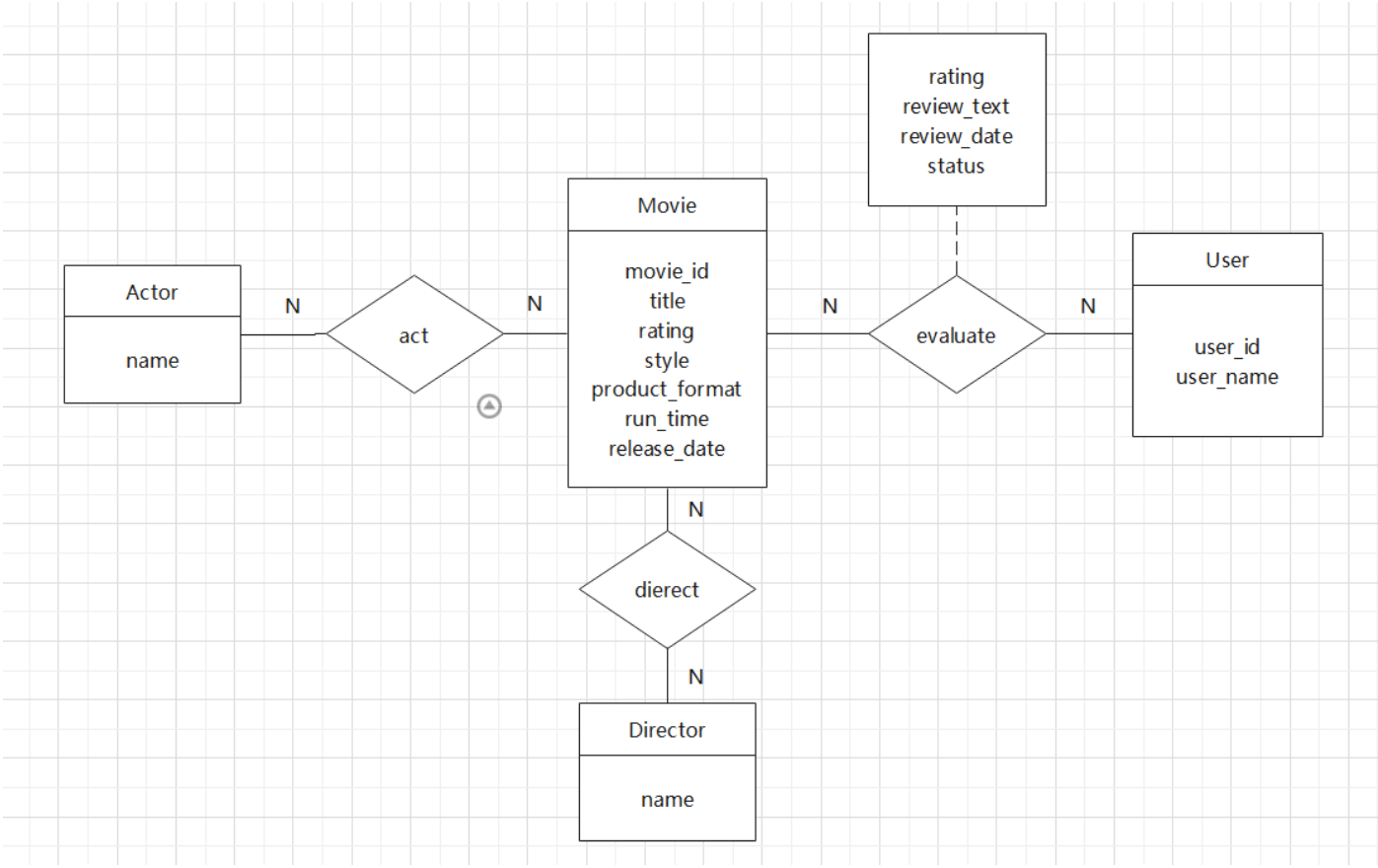
通过测试我们可以看出：对于前5个比较简单的查询，由于在Mysql中进行了统计表、索引等的存储优化(详细内容见第4节)因此Mysql查询的效率最高，而Neo4j也表现出了较为优良的性能，但是Hive的查询性能较差;而对于后3个，比较复杂的查询逻辑，由于Mysql需要涉及联表查询性能会严重下降，而Neo4j ，可以使用图结构的自然伸展特性来设计免索引邻近节点遍历的查询算法，性能更优，但是Hive的性能依旧不佳;第6个查询是一个特例，虽然是比较简单的查询，但是Mysql并没有表现出很好的性能，这是因为设计时没有建立相关的索引，在后续会有关于Mysql索引的说明。

而Hive表现不佳的原因是：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。

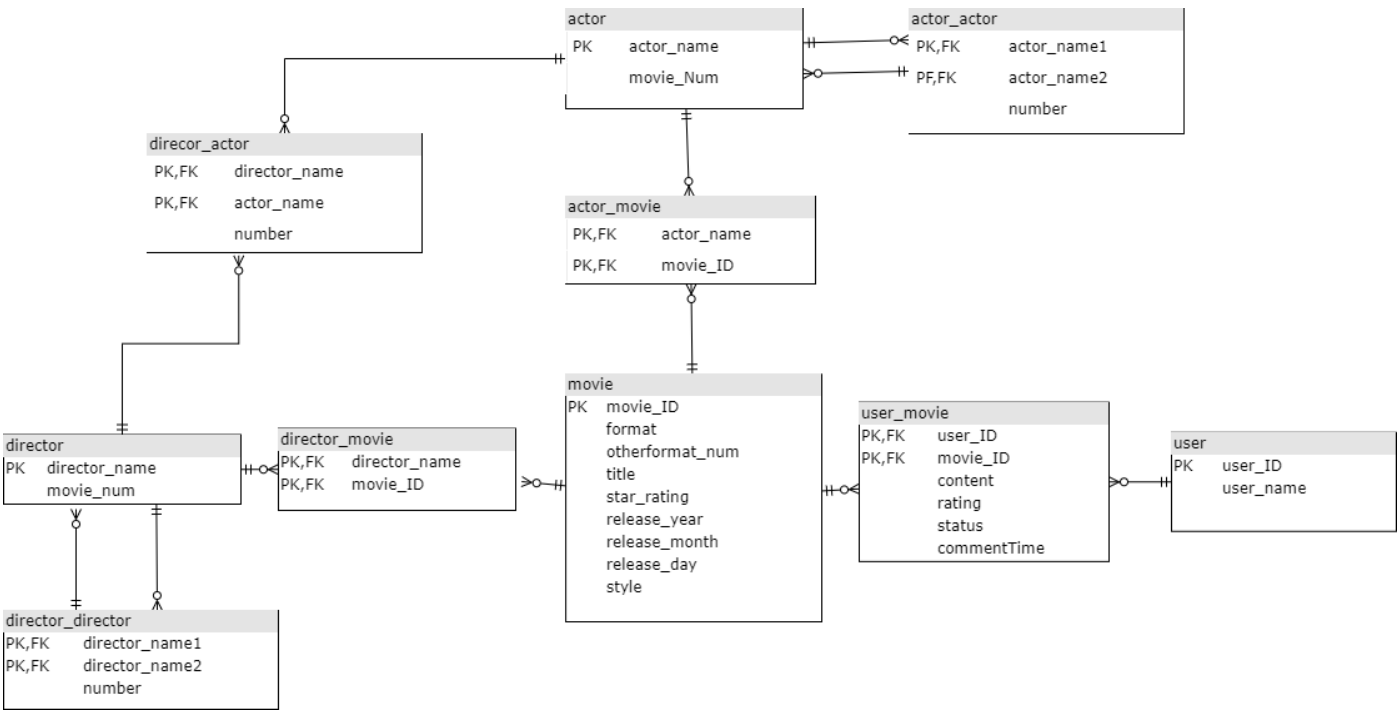
因此，经过上面的分析，可以得出结论：Mysql适用于小数据量的单表查询；Neo4j适用于处理大量复杂的、相互连接的数据，可以规避关系型数据库中的大规模的表连接，比如查询两个演员合作最多的次数等;而Hive更适用于实时性要求不高的场合。如果数据量足够大，那么相对于分析统计海量数据的执行时间，Hive的延迟已经不能成为它的缺点，但是在本项目中，由于数据量并没有那么大，如果从效率的角度来考虑并不适用。当然，由于Hive搭建在分布式文件系统的基础上，所以具有很好的扩展性和容错性。

综上，整体存储模式为：Mysql存储使用频率较高的电影发布时间、电影标题、电影评分、演员姓名、演员参演电影总数、导演姓名、导演参演电影总数等信息，以应对小数据量的单表查询；Neo4j存储导演、演员、电影、用户之间的各种关系，以应对类似两个演员合作最多的次数等的较为复杂、数据相互连接的查询；而Hive存储使用频度低的数据量大的数据，比如用户评论的具体内容。

2. 关系型存储逻辑模型



3. 关系型存储物理模型



4. 关系型数据库存储优化设计

传统的关系型数据库本身并不适合作为数据仓库，因此为了处理大量数据的存储和查询，需要对其存储结构进行相应的优化。

4.1 表结构优化

4.1.1 增加冗余的优化

首先通过将部分冗余的信息存储在不同的表中，提高了查询的效率。比如我们在电影表中存储了电影的版本信息：电影的版本和其他版本的数量，而并没有将其作为表的形式抽象出来；在电影和用户的关系表中存储了评论的内容，评分等信息，而并没有将其作为表的形式抽象出来。这是为了尽量减少联表查询的需求，以应对更加复杂的查询条件，提高查询效率。

4.1.2 增加统计属性

对于一些高频的、高复杂度的查询任务，我们选择提前将其计算出来，作为属性冗余的存储在数据库中，虽然增加了冗余，但是大大加快了查询效率。比如演员和演员，演员和导演，导演和导演之间的合作次数，电影版本的数量，每个演员或导演参加的电影总数，这些信息都被提前计算并存储在数据库中，提升了查询效率。

4.2 索引优化

为某些关键的字段添加索引，可以很有效的提高数据查询和导入数据库的时间效率，尤其是数据仓库的项目中，数据量非常庞大，添加索引对效率的增益就非常突出。

举例测试结果如下：

1. 在movie电影实体表中，添加发布日期：release_year,release_month,release_day作为联合索引

查询类型	添加索引前时间(ms)	添加索引后时间(ms)
查询某一年某一月某一日有多少电影	226	2

2. 在actor_actor演员关系表中，添加演员姓名actor_name1，actor_name2作为索引

   在director_actor演员导演关系表中，添加演员姓名actor_name，导演姓名director_name作为索引

查询类型	添加索引前时间(ms)	添加索引后时间(ms)
按人名名称查询合作最多的演员和合作次数	44	5

可见，索引的使用对于该数据仓库的效率的提高作用很大。

5. 分布式文件系统存储模型

5.1 分布式数据库与数据库设计

5.1.1 分布式数据库

搭建全分布式文件系统，节点中共有四台CentOs虚拟机，一台作为NameNode，另外三台作为DataNode。各项配置以及版本信息如下：

Plain Text

```
操作系统 centos7
Hadoop 2.8.3
    Hadoop依赖
        jdk1.8
Hive 3.1.2
    Hive依赖
        mysql:8
        mysql jdbc:8.0.26
```

	A	B	C	D
1	**Node01**	**Node02**	**Node03**	**Node04**
2	Namenode	Secondarynamenode		
3	Resourcemanager			
4	Datanode	Datanode	Datanode	Datanode
5	Hive			
6	Mysql			

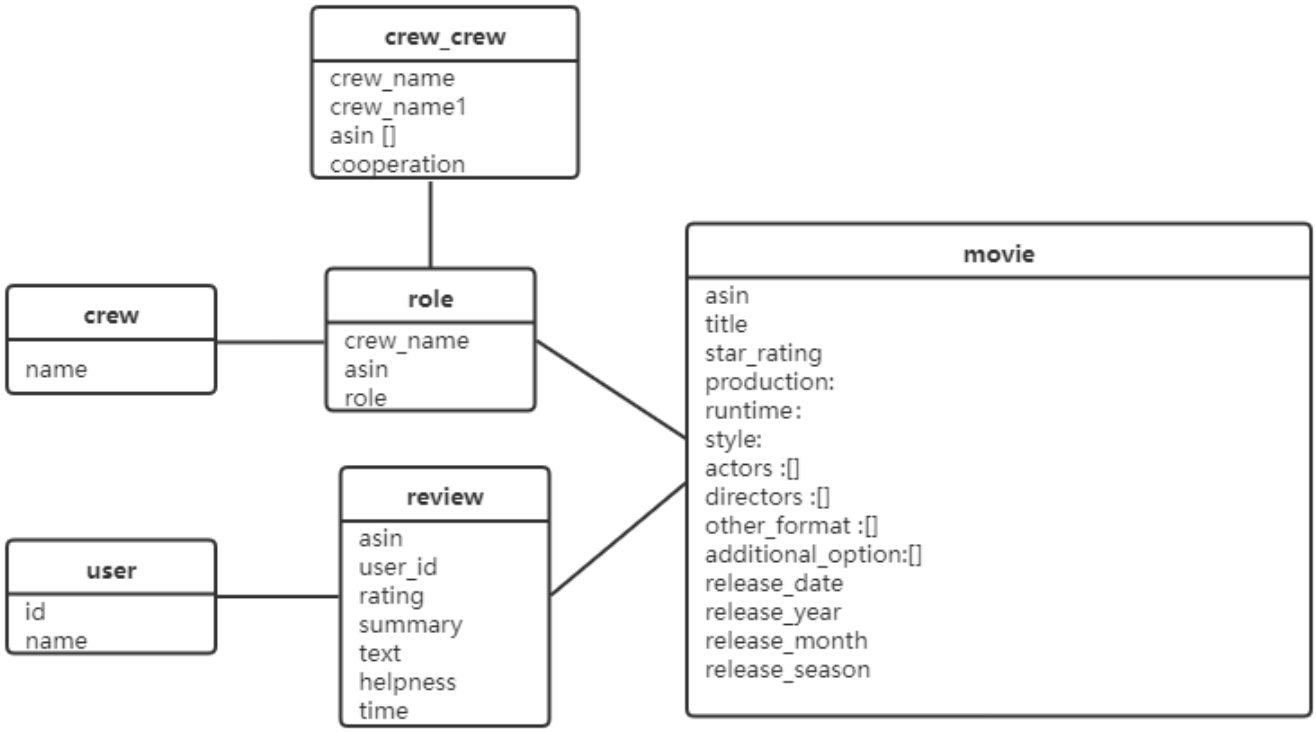
5.1.2数据库设计

项目所有查询围绕着电影展开，并以查询为主要目的，所以为了满足业务需求。我们应该适当设置一些冗余，为此最后生成了6张表来存相关信息。

在设计存储模型时，主要考虑以下几点：

- 演员和导演并不是一个人的固有属性，而是一个人在某部电影里的角色，所以设计了crew表来存储电影中出现的所有演员和导演，爬下来的数据中没有每个人的Id，只能用名字作为去重。
- 考虑到有部分查询涉及年月日和季节，所以需要把数据中的日期拆分成年月日和季节，虽然有一定的冗余，但是在查询时无需拆分日期，可以达到查询优化的目的。

- 在目前的要求中，存在查询合作关系的业务需求，所以建立了一张表crew_crew，它是由crew crew在movie上的联系集，其中cooperation是代表两者的关系，分别是actor_actor，director_actor，director_director。
- 不同于关系数据库，hive中是有数组array这样的结构的，所以对于电影的其他版本或者是电影的演员和导演，我们可以依靠数组来存储。比如在查询一部电影的其他版本数量时，只需要返回数组的长度即可，不需要进行大量的MapReduce，因此大大提高了查询速度。



5.2 数据存储及优化

向Hive中导入数据有五种方式可供选择：

- 本地文件导入到Hive表
- Hive表导入到Hive表;
- HDFS文件导入到Hive表
- 创建表的过程中从其他表导入
- 通过sqoop将mysql库导入到Hive表

首先建立起表结构，csv默认是用逗号来分割。特别注意，对于属性中有数组的表，我们不能设置数组的间隔符是逗号，所以选用了“-”来作为分隔符，并将csv通过docker cp命令上传到namenode01,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csv文件到HDFS文件系统中的速度，导入速度非常可观。

5.2.1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文件由于是列示存储，对于不需要查询整张表的所有数据的sql，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。

新建一个数据库 create database orcdata;然后把原数据库的每张表导入到新的数据库

insert overwrite table orc_database_name.table_name select * from text_database_name.table_name; 命令把Textfile文件格式的表放进ORC文件格式的数据表中。

	A	B
1	**TEXTFILE**	**ORC**
2	39.0M	15.9M

可以看到存储空间压缩了将近百分之40，效果非常好

下面分别执行两条sql来对比性能

查询某电影的所有评论用户数量

SQL

```sql
select count(*) from review where asin='6302782082'
```

	A	B
1	**TEXTFILE**	**ORC**
2	5s 209ms	2s 303ms

查询某电影评分大于3的评论平均分

SQL

```sql
select avg(rating) from review where rating>3 and asin='6302782082'
```

	A	B
1	**TEXTFILE**	**ORC**
2	4s 560ms	4s 767ms

5.2.2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，

执行语句：查询评分等于3的评论信息

```
SQL
1  Select count(*) from review where rating=3
```

	A	B
1	**无索引**	**有索引**
2	7s 670ms	7s 950ms

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

5.2.3分区表优化

Hive在使用条件查询的时候会扫描整张表，消耗很多时间做没必要的工作。若大量的查询是基于某个字段的，那么我们可以对数据进行分区处理。分区表是hive提供的一个特性，利用partition关键字可以把现成的数据表分成多个分区（文件夹）存储分区将表的数据在物理上分成不同的文件夹，以便于在查询时可以精准指定所要读取的分区目录，从而降低读取的数据量。

比如crew和movie的联系集role这张表，一个crew要么是actor，要么是director。而我们的查询中有很多基于角色来分类的，所以重建一张表partition_role，以role作为区分属性，执行命令insert overwrite table orc_database.role partition(role) select * from role，在物理上把数据库拆分成了两个文件存储

下面来比较性能

查询人员中作为演员参演过电影的数量

```
SQL
1  select count(*) from role where role='actor'
```

	A	B
1	**original**	**Partition**
2	2s 279ms	1s 316ms

查询人员中成为演员的最多次数

SQL

```sql
select count(*) times from role
where role='actor'
group by name
order by times desc
limit 1;
```

	A	B
1	**original**	**Partition**
2	7s 688ms	4s 764ms

可以看到分区之后，由不进行全表扫描，而是针对分区查询，所以效率有很大的提升。

在优化的过程中，由于有大量的按照时间查询电影信息，我尝试把movie按照年份分区，得到了大概40-50份的分区，执行语句**查询12月份发行的电影数量**

SQL

```sql
Select count(*) from movie where release_month=12
```

比较运行时间

original	Partition
4s 360ms	4s 350ms

发现并没有很大的提升，考虑如下两个原因：

- 数据量太小，hive适用于百万级别的数据，在本次实验中，电影的数量只有较少的两万多条，提升效果不显著
- 虽然分区在物理上把文件划分成了多个部分，但逻辑上却产生了很多小文件。也就是整个MapReduce启动的任务数要增加，这样并不利于整个查询速度的提升

5.2.4MapReduce优化

我们都知道hive底层默认使用的是MR引擎，无论是map还是reduce的数量都会极大地影响整个查询的速度，我们可以通过设置一些参数来调整

- 注意到hadoop默认分块大小是128M，对于比较大的数据来说，进行分块无疑可以使得map任务算力均衡，达到比较好的效果；但若一个任务中有很多小文件，如10M，20M等等，这样一个小文件也会被当作一个map任务来执行，而一个map任务启动和初始化的时间远远大于逻辑处理的时间，就会造成很大的资源浪费。所以我们应该合理设置分块的大小来使得资源合理分配。

- Reduce的个数对整个作业的运行性能有很大影响。如果Reduce设置的过大，那么将会产生很多小文件，对NameNode会产生一定的影响，而且整个作业的运行时间未必会减少；如果Reduce设置的过小，那么单个Reduce处理的数据将会加大，很可能会引起OOM异常。实验过程中发现**部分任务**中不管数据量多大，不管有没有调整reduce个数的参数，任务中一直都只有一个reduce任务。经过探究发现，分为以下三种情况
  - 没有group by的汇总，比如下面两种sql语句执行起来是不一样的

SQL

```sql
select asin,count(1) from movie where asin = '2012-07-04' group by asin;'

select count(1) from movie where asin = '2012-07-04';
```

  - 使用了Order by进行排序
  - sql中包含笛卡尔积。

在设置reduce个数的时候需要考虑这两个原则：使大数据量利用合适的reduce数；是单个reduce任务处理合适的数据量；

以下是我为了优化尝试修改的部分参数

	A	B
1	**描述**	**参数设置**
2	设置分块大小	set dfs.block.size=16M
3	设置每个Map最大输入大小	set mapred.max.split.size=16777216
4	开启并发执行	set hive.exec.parallel=true
5	设置并发线程数	set hive.exec.parallel.thread.number=16
6	设置map的任务数	set mapred.map.tasks=4
7	设置reduce的任务数	set mapred.reduce.tasks=4
8	设置每个reduce任务处理的数据量	set hive.exec.reducers.bytes.per.reducer=128M
9	设置每个任务最大的reduce数	set hive.exec.reducers.max=999

优化效果并不显著，推测是本次任务的数量太小

5.3Hive数据仓库特点分析

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。

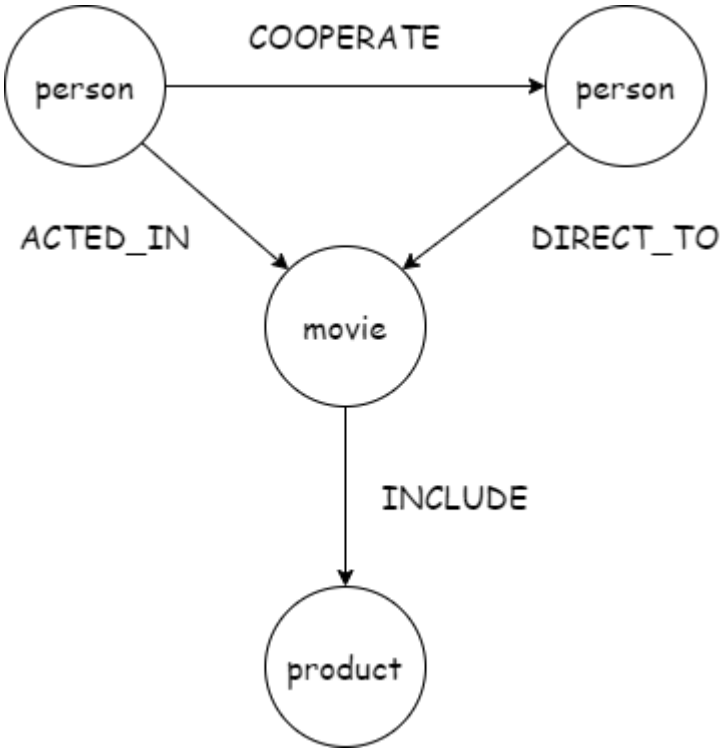
Hive主要用来进行数据分析，对于分析统计海量数据的执行时间而言，Hive的执行延迟已经不能成为它的缺点，MapReduce的算法使得分布式节点的性能数量成为执行时间的主要限制因素。

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。

6. 图数据库存储模型

6.1 存储模型

图数据库使用Neo4j存储，设计有三种不同的label，分别是电影movie，人员person，商品product，设计四种联系，分别是参演电影的ACTED_IN，指导电影的DIRECT_TO，人员之间合作的COOPERATE，电影包含不同商品类型的INCLUDE



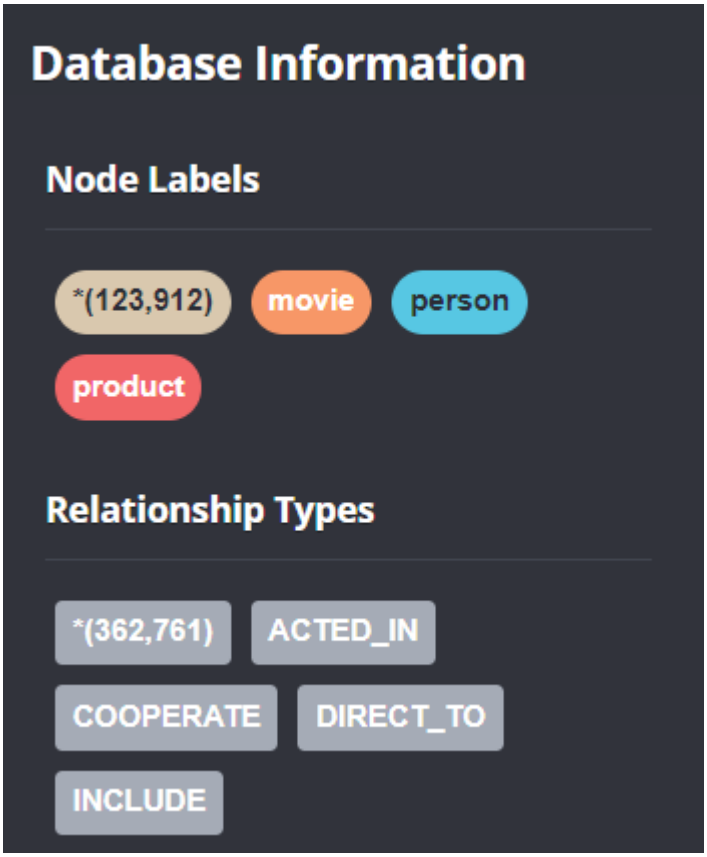
各个字段的属性以及label示例如下：

```
{
  "identity": 4,
  "labels": [
    "movie"
  ],
  "properties": {
"month": 10,
"year": 2010,
"rating": "3.5",
"format": "DVD",
"asin": "B003X3BY8S",
"style": "Drama",
"title": "Fade to Black",
"day": 5
  }
}
```

```
{
  "identity": 23168,
  "labels": [
    "person"
  ],
  "properties": {
"name": "Mike McFarland",
"direct_num": 12,
"act_num": 0
  }
}
```

```
{
  "identity": 510,
  "labels": [
    "product"
  ],
  "properties": {
"asin": "B00MP2FVIC"
  }
}
```


最终有人员结点46860个，电影结点23158个，商品结点53894个，共有123912个结点，362761个联系。



在设计存储模型时，主要需要考虑以下几点：

（1） 图数据库不需要存储在ETL中为了提高关系型数据库的查询效率，而人为添加的冗余表

（2） 图数据库主要存储电影、人员和商品的信息，为这三种实体考虑设计他们之间的联系

（3）对于Neo4j来说，并不推荐使用无向边来建立联系，所有的关系在创建的时候都必须是有向的，但是在查询的时候neo4j可以使用双向的关系进行查询，这正是我们想要的

（4）由于一个人既可以是导演又可以是演员，有两种方案进行存储：一是为相同的结点赋予多个label；二是不区分演员和导演结点，而是通过与电影结点的不同关系来区分一个人是否指导或者参演某部电影。在本项目中我们采用了后者，这种方式可以避免Neo4j查询lebel时效率降低。

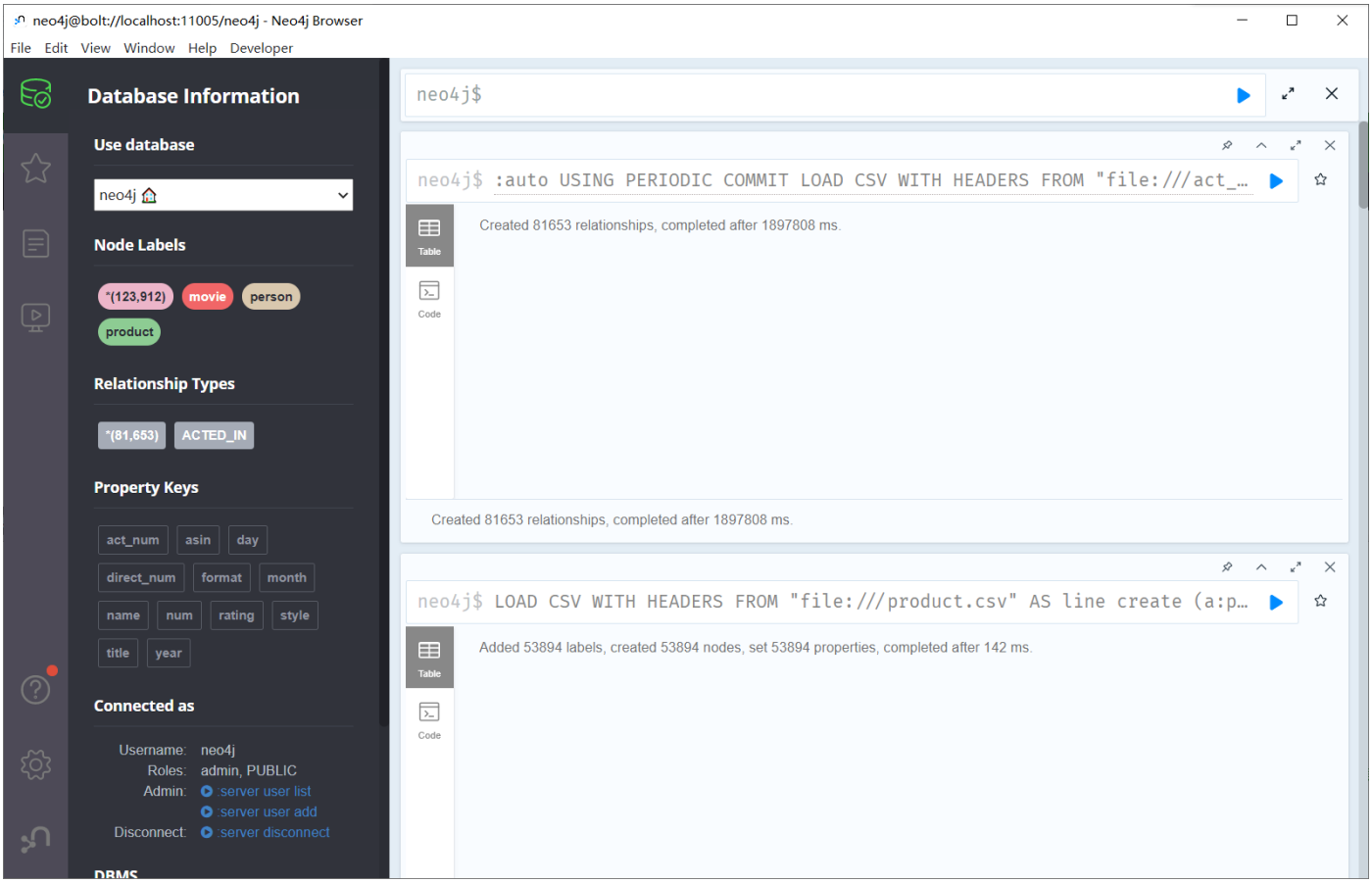
6.2 查询优化

对于图数据库，较为重要和有效的查询优化方法是建立索引， 对于movie的结点，以asin建立索引，对于person结点，以name建立索引：

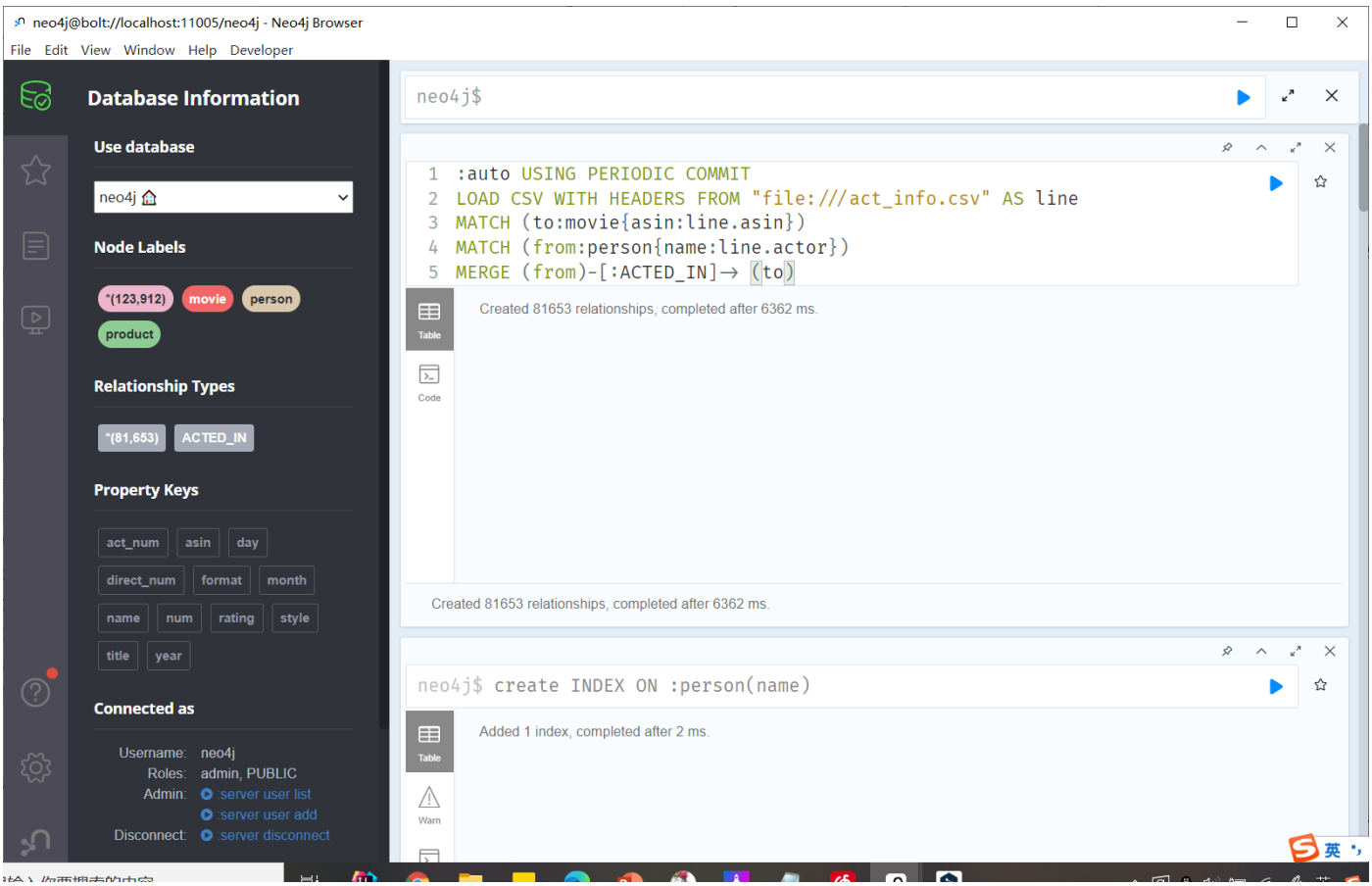
CREATE INDEX ON :movie(asin)

CREATE INDEX ON :person(name)

在做person和movie的联系ACTED_IN的建立时，需要匹配符合要求的演员和电影，如果不添加索引，即相当于需要对人员和电影整张表进行笛卡尔积，效率显然会很慢，需要花费1897808ms，即接近20分钟才能完成这个工作：



建立索引优化后，查询并建立联系的效率显著提升，只需要6362ms就可以完成这项工作：



对于这种两个万级数据的匹配工作，使用索引进行查询优化使效率提高了近300倍，可以看到使用索引对较大数据量查询工作的优化效果是十分明显的。对于不同的查询任务，建立所需要的的对应索引即可。

7. 数据表的test case

按照时间进行查询及统计

1. 某一年有多少电影
2. 某一年某一月有多少电影
3. 某一年某一季度有多少电影
4. 某一年某一月某一日有多少电影

按照电影名称进行查询及统计

5. 某一电影有多少版本

按照导演进行查询及统计

6. 某一导演指导多少电影

按照演员进行查询及统计

7. 某一演员参演多少电影

按照演员和导演的关系进行查询及统计

8. 按人名名称查询合作最多的演员和合作次数
9. 按人名名称查询合作最多的导演和合作次数

按照电影类别进行查询及统计

10. 按照电影类别查询电影总数

按照用户评价进行查询及统计

11. 查询评分在x以上的电影数
12. 查询存在评分大于x的评论的电影数

按照上述条件的组合查询和统计

13. 某演员某年参演的电影数
14. 某年某种类型电影数
15. 哪两个演员合作次数最多