数据存储设计说明文件

本项目以亚马逊电影评论数据集作为数据基础，针对电影信息，建立基于关系型数据库（MySQL）、分布式数据库（Hadoop+Hive）和图数据库（Neo4j）的数据仓库系统，构建数据治理体系，并进行反范式化设计，实现了较高效率的综合条件查询与演员导演之间的合作关系查询，此外，还可以进行不同种数据库的查询性能对比。

1. 整体存储模式

1.1 关系型数据库

本项目使用MySQL作为关系型数据库，存储了全部的和电影相关的信息，以支持全部种类的查询，包括综合条件查询和演员导演之间的关系查询。

表名	存储内容
movie	电影名称，电影标题，电影评分，电影运行时间，电影总评论数
actor 和 act	演员的名字及参演的电影
director和direct	导演的名字及导演的电影
genre	存储电影的风格
release_date	存储电影上映时间相关的信息
review	存储所有评论相关的信息
version	存储电影相关的版本、语言、格式信息
review_sentiment	存储某个电影所有评论的情感倾向得分

在本项目中，我们对物理模型进行了反规范化，使用星型模型，减少了一部分join操作。比如，对于release_date表，我们提前处理好了年、月、日、季度和星期五个变量将其存储为单独字段，在根据上映时间查询时便不再需要对综合属性date进行拆分；以及在movie表中添加了冗余字段review_num，从而提高查询速率。同时，为了方便进行演员之间的关系以及演员和导演之间关系的查询，我们建立了记录演员之间关系的视图actor_actor和记录演员和导演之间关系的视图director_actor，方便了查询。

此外，为了加速查询，我们还建立了许多索引。

1.2 分布式数据库

在本项目中，分布式数据库使用Hive+hadoop，是由1个namenode和1个datanode组成的伪分布式架构，并以MR作为计算引擎，HDFS作为存储系统，我们的集群使用postgre进行元数据存储，配置在docker上，并尝试配置了SparkSQL计算框架。

分布式数据库的表结构以及存储的数据和关系型数据库类似，但我们将关系型数据库的视图actor_actor，director_actor处理成了两张实际的外表而不是视图，加速了演员之间的关系以及演员导演之间关系的查询。

我们的分布式数据库的所有表均采用外部表。这对于大数据场景可以是一种优化，因为它允许直接在数据所在位置进行查询处理，减少了数据移动的需要，虽然分布式数据库在增加数据等场景下表现得并不好，但在我们的应用场景中只需要进行数据的查询，因此不必考虑数据的增加带来的问题。

1.3 图数据库

本项目中，我们使用Neo4j进行图数据库的存储。在Neo4j中，实体和关系存储的数据如下：

实体：

1. Actor（演员）
    - actor_id: 演员的唯一标识符（ID），整数类型。
    - actor_name: 演员的名字。
2. Director（导演）
    - director_id: 导演的唯一标识符（ID），整数类型。
    - director_name: 导演的名字。
3. Movie（电影）
    - movie_id: 电影的唯一标识符（ID）。
    - movie_title: 电影的标题。
    - movie_genre: 电影的类型或流派。
    - movie_review_num: 电影的评论数量。

关系：

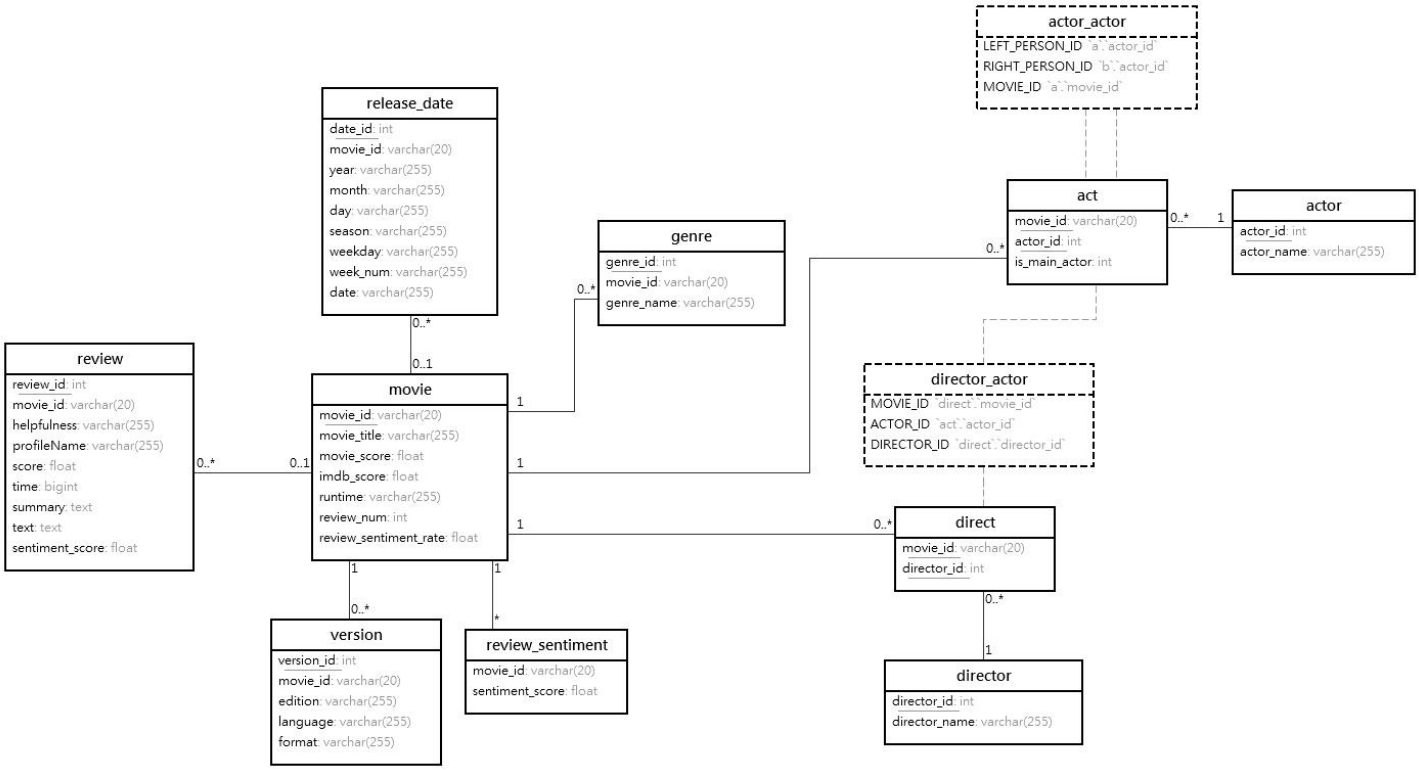
1. ACTED（参演）
    - 连接演员（Actor）和电影（Movie）。
    - 表示一个演员参演了某部电影。
    - 包含属性：
        - is_main_actor: 表示该演员是否是主要演员。
2. DIRECT（执导）

- 连接导演（Director）和电影（Movie）。
- 表示一个导演执导了某部电影。

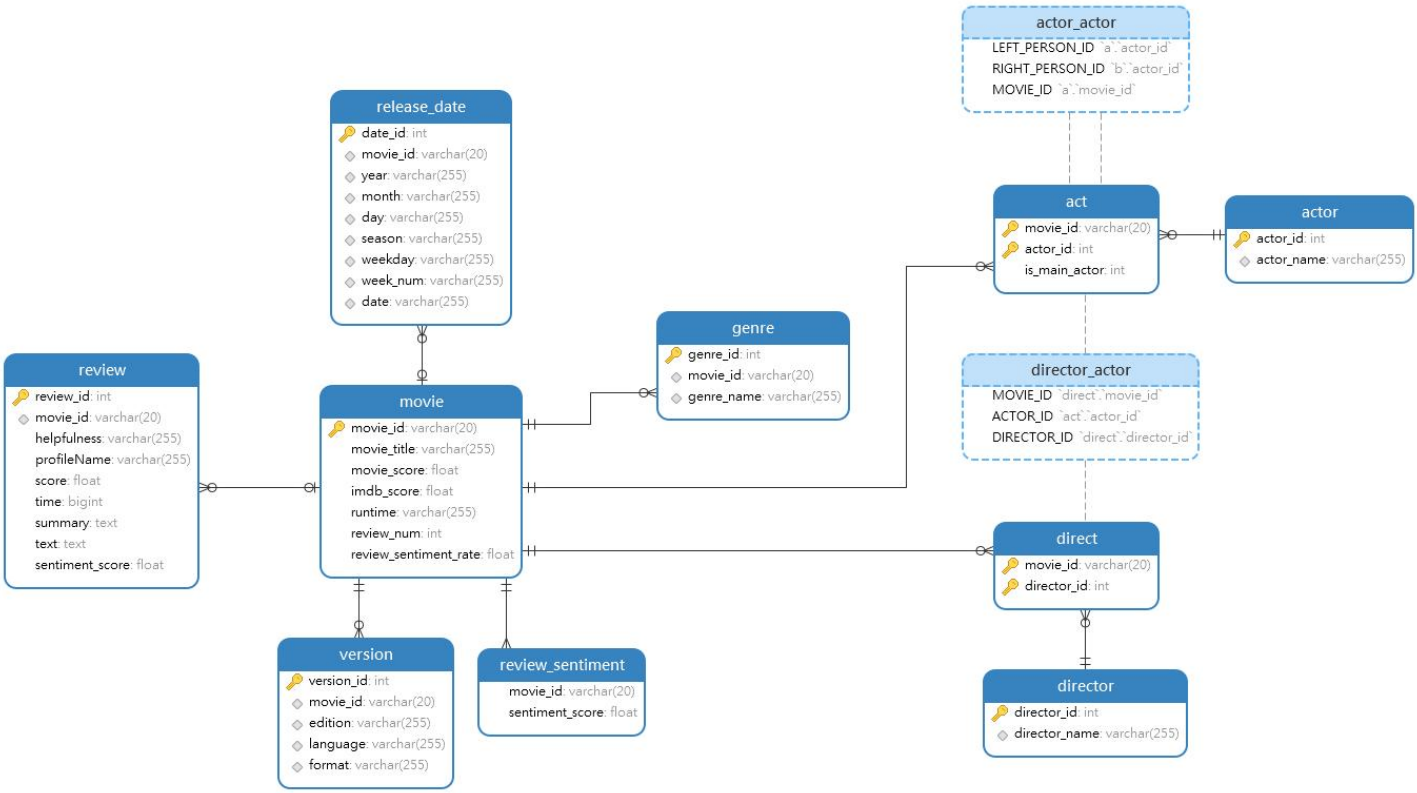
我们仅存储了不同演员之间和导演与演员之间的合作关系，以及演员参演和导演执导的电影，图数据库在我们的项目中仅用于关系的查询和速度的对比，在图数据库中，我们也建立的针对导演和演员名字，电影风格以及电影名称的索引以加速查询。

关系型存储逻辑模型

E-R图设计



数据存储模型



关系型存储物理模型

DDL

```
CREATE TABLE `act`  (
  `movie_id` varchar(20) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci NOT NULL,
  `actor_id` int NOT NULL,
  `is_main_actor` int NULL DEFAULT NULL COMMENT '是否为主演，1为是，0为否',
  PRIMARY KEY (`movie_id`, `actor_id`) USING BTREE,
  INDEX `act_ibfk_1`(`actor_id` ASC) USING BTREE
) ENGINE = InnoDB CHARACTER SET = utf8mb4 COLLATE = utf8mb4_general_ci ROW_FORMAT = DYNAMIC;

CREATE TABLE `actor`  (
  `actor_id` int NOT NULL AUTO_INCREMENT COMMENT '演员id',
  `actor_name` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci NULL DEFAULT NULL COMMENT '演员姓名',
  PRIMARY KEY (`actor_id`) USING BTREE,
  INDEX `actor_name`(`actor_name` ASC) USING BTREE COMMENT '快速查询姓名'
) ENGINE = InnoDB AUTO_INCREMENT = 135486 CHARACTER SET = utf8mb4 COLLATE = utf8mb4_general_ci ROW_FORMAT = DYNAMIC;

CREATE TABLE `default`  (
  `time` bigint NOT NULL,
  `sentiment_score` float NULL DEFAULT NULL
```

```
) ENGINE = InnoDB CHARACTER SET = utf8mb4 COLLATE = utf8mb4_general_ci ROW_FORMAT = Dynamic;

CREATE TABLE `direct`  (
  `movie_id` varchar(20) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci NOT NULL COMMENT '电影id',
  `director_id` int NOT NULL COMMENT '导演id',
  PRIMARY KEY (`movie_id`, `director_id`) USING BTREE,
  INDEX `director_id`(`director_id` ASC) USING BTREE
) ENGINE = InnoDB CHARACTER SET = utf8mb4 COLLATE = utf8mb4_general_ci ROW_FORMAT = DYNAMIC;

CREATE TABLE `director`  (
  `director_id` int NOT NULL AUTO_INCREMENT COMMENT '导演id',
  `director_name` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci NULL DEFAULT NULL COMMENT '导演名',
  PRIMARY KEY (`director_id`) USING BTREE,
  INDEX `director_name`(`director_name` ASC) USING BTREE COMMENT '姓名索引'
) ENGINE = InnoDB AUTO_INCREMENT = 40031 CHARACTER SET = utf8mb4 COLLATE = utf8mb4_general_ci ROW_FORMAT = DYNAMIC;

CREATE TABLE `genre`  (
  `genre_id` int NOT NULL AUTO_INCREMENT COMMENT '主码，自增',
  `movie_id` varchar(20) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci NOT NULL COMMENT '电影id有这个风格',
  `genre_name` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci NULL DEFAULT NULL COMMENT '类别名称',
  PRIMARY KEY (`genre_id`) USING BTREE,
  INDEX `genre_name`(`genre_name` ASC) USING BTREE COMMENT '类别索引',
  INDEX `movie_id`(`movie_id` ASC) USING BTREE
) ENGINE = InnoDB AUTO_INCREMENT = 245448 CHARACTER SET = utf8mb4 COLLATE = utf8mb4_general_ci ROW_FORMAT = DYNAMIC;

CREATE TABLE `movie`  (
  `movie_id` varchar(20) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci NOT NULL COMMENT 'ANSI作为主码',
  `movie_title` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci NULL DEFAULT NULL COMMENT '名称',
  `movie_score` float NULL DEFAULT NULL COMMENT '用户评分',
  `imdb_score` float NULL DEFAULT NULL COMMENT 'imdb评分',
  `runtime` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci NULL DEFAULT NULL COMMENT '时长',
  `review_num` int NOT NULL DEFAULT 0 COMMENT '评论数',
  `review_sentiment_rate` float NULL DEFAULT NULL COMMENT '电影正面评价百分比',
  PRIMARY KEY (`movie_id`) USING BTREE
) ENGINE = InnoDB CHARACTER SET = utf8mb4 COLLATE = utf8mb4_general_ci ROW_FORMAT = DYNAMIC;
```

```
54
55 CREATE TABLE `release_date`  (
56   `date_id` int NOT NULL AUTO_INCREMENT,
57   `movie_id` varchar(20) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci NULL
   DEFAULT NULL COMMENT '该日期对应的电影',
58   `year` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci NULL
   DEFAULT NULL COMMENT '年',
59   `month` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci NULL
   DEFAULT NULL COMMENT '月',
60   `day` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci NULL
   DEFAULT NULL COMMENT '日',
61   `season` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci NULL
   DEFAULT NULL COMMENT '季度',
62   `weekday` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci NULL
   DEFAULT NULL COMMENT '周几',
63   `week_num` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci NULL
    DEFAULT NULL COMMENT '在当前年的第几周',
64   `date` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci NULL
   DEFAULT NULL COMMENT '年月日',
65   PRIMARY KEY (`date_id`) USING BTREE,
66   INDEX `year`(`year` ASC) USING BTREE COMMENT '年索引',
67   INDEX `year_2`(`year` ASC, `month` ASC) USING BTREE COMMENT '年、月组合索引',
68   INDEX `year_3`(`year` ASC, `season` ASC) USING BTREE COMMENT '年、季度组合索
   引',
69   INDEX `weekday`(`weekday` ASC) USING BTREE COMMENT 'weekday单列索引',
70   INDEX `month`(`month` ASC) USING BTREE,
71   INDEX `day`(`day` ASC) USING BTREE,
72   INDEX `season`(`season` ASC) USING BTREE,
73   INDEX `week_num`(`week_num` ASC) USING BTREE,
74   INDEX `date`(`date` ASC) USING BTREE,
75   INDEX `movie_id`(`movie_id` ASC) USING BTREE
76 ) ENGINE = InnoDB AUTO_INCREMENT = 185011 CHARACTER SET = utf8mb4 COLLATE =
   utf8mb4_general_ci ROW_FORMAT = DYNAMIC;
77
78 CREATE TABLE `review`  (
79   `review_id` int NOT NULL COMMENT '主码，自增',
80   `movie_id` varchar(20) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci NULL
   DEFAULT NULL COMMENT '评论的电影',
81   `helpfulness` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci
   NULL DEFAULT NULL COMMENT '有用程度',
82   `profileName` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci
   NULL DEFAULT NULL COMMENT '用户名',
83   `score` float NULL DEFAULT NULL COMMENT '分',
84   `time` bigint NULL DEFAULT NULL COMMENT '时间戳',
85   `summary` text CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci NULL COMMENT
   '总结',
```

```
  `text` text CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci NULL COMMENT '正文',
  `sentiment_score` float NULL DEFAULT 0 COMMENT '情感倾向分数',
  PRIMARY KEY (`review_id`) USING BTREE,
  INDEX `movie_id`(`movie_id` ASC) USING BTREE
) ENGINE = InnoDB CHARACTER SET = utf8mb4 COLLATE = utf8mb4_general_ci ROW_FORMAT = DYNAMIC;

CREATE TABLE `review_sentiment`  (
  `movie_id` varchar(20) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci NULL DEFAULT NULL,
  `sentiment_score` float NULL DEFAULT NULL
) ENGINE = InnoDB CHARACTER SET = utf8mb4 COLLATE = utf8mb4_general_ci ROW_FORMAT = Dynamic;

CREATE TABLE `version`  (
  `version_id` int NOT NULL AUTO_INCREMENT,
  `movie_id` varchar(20) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci NOT NULL COMMENT '该版本对应的电影id',
  `edition` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci NULL DEFAULT NULL COMMENT '本电影单独的版本',
  `language` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci NULL DEFAULT NULL COMMENT '电影语言',
  `format` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci NULL DEFAULT NULL COMMENT '该电影的多个版本',
  PRIMARY KEY (`version_id`) USING BTREE,
  INDEX `edition`(`edition` ASC) USING BTREE COMMENT '版本索引',
  INDEX `language`(`language` ASC) USING BTREE COMMENT '语言索引',
  INDEX `format`(`format` ASC) USING BTREE COMMENT '格式索引',
  INDEX `movie_id`(`movie_id` ASC) USING BTREE
) ENGINE = InnoDB AUTO_INCREMENT = 430093 CHARACTER SET = utf8mb4 COLLATE = utf8mb4_general_ci ROW_FORMAT = DYNAMIC;

ALTER TABLE `act` ADD CONSTRAINT `act_ibfk_1` FOREIGN KEY (`actor_id`) REFERENCES `actor` (`actor_id`) ON DELETE CASCADE ON UPDATE CASCADE;
ALTER TABLE `act` ADD CONSTRAINT `act_ibfk_2` FOREIGN KEY (`movie_id`) REFERENCES `movie` (`movie_id`) ON DELETE CASCADE ON UPDATE CASCADE;
ALTER TABLE `direct` ADD CONSTRAINT `direct_ibfk_2` FOREIGN KEY (`director_id`) REFERENCES `director` (`director_id`) ON DELETE CASCADE ON UPDATE CASCADE;
ALTER TABLE `direct` ADD CONSTRAINT `direct_ibfk_3` FOREIGN KEY (`movie_id`) REFERENCES `movie` (`movie_id`) ON DELETE CASCADE ON UPDATE CASCADE;
ALTER TABLE `genre` ADD CONSTRAINT `genre_ibfk_1` FOREIGN KEY (`movie_id`) REFERENCES `movie` (`movie_id`) ON DELETE CASCADE ON UPDATE CASCADE;
ALTER TABLE `movie` ADD CONSTRAINT `fk_movie_review_sentiment_1` FOREIGN KEY (`review_num`) REFERENCES `review_sentiment` (`movie_id`);
```

```sql
ALTER TABLE `release_date` ADD CONSTRAINT `release_date_ibfk_1` FOREIGN KEY (`movie_id`) REFERENCES `movie` (`movie_id`) ON DELETE CASCADE ON UPDATE CASCADE;
ALTER TABLE `review` ADD CONSTRAINT `review_ibfk_1` FOREIGN KEY (`movie_id`) REFERENCES `movie` (`movie_id`) ON DELETE CASCADE ON UPDATE CASCADE;
ALTER TABLE `version` ADD CONSTRAINT `version_ibfk_1` FOREIGN KEY (`movie_id`) REFERENCES `movie` (`movie_id`) ON DELETE CASCADE ON UPDATE CASCADE;

CREATE ALGORITHM = UNDEFINED DEFINER = `DW2023`@`%` SQL SECURITY DEFINER VIEW `actor_actor` AS select `a`.`actor_id` AS `LEFT_PERSON_ID`,`b`.`actor_id` AS `RIGHT_PERSON_ID`,`a`.`movie_id` AS `MOVIE_ID` from (`act` `a` join `act` `b` on(((`a`.`movie_id` = `b`.`movie_id`) and (`a`.`actor_id` < `b`.`actor_id`))));

CREATE ALGORITHM = UNDEFINED DEFINER = `DW2023`@`%` SQL SECURITY DEFINER VIEW `director_actor` AS select `direct`.`movie_id` AS `MOVIE_ID`,`act`.`actor_id` AS `ACTOR_ID`,`direct`.`director_id` AS `DIRECTOR_ID` from (`act` join `direct` on((`act`.`movie_id` = `direct`.`movie_id`)));

```

存储优化设计及Denormalization

在本项目中，我们对物理模型进行反规范化，并通过建立索引等操作对存储进行优化

1. 星型模型使用

本项目采用星型模型结构，对数据存储有一定的冗余。首先建立一个movie表，然后将movie的属性都

2. 字段设置
    - 在review表中，time字段记录了该评论的时间戳，我们并没有采用 `datetime` 等和时间有关的数据类型来存储时间戳，而是采用了 `bigint` 类型的数据来存储。使用bigint类型存储时间戳可以确保跨不同数据库和编程语言的一致性；相对于 `datetime` 类型， `bigint` 类型通常需要较少的存储空间，这对于像review表这样的行数很多的表来说，可以节省相当的存储空间。在某些情况下，使用 `bigint` 类型进行日期和时间的比较可能比使用 `datetime` 类型更高效。这是因为整数的比较操作通常比日期时间类型的比较要快。
    - 在movie表中，我们使用 `float` 类型的数据来存储movie_score等和评分相关的数据。 `float` 类型的存储空间通常小于 `double` 类型，而电影评分并不需要较高的精度，因此使用 `float` 可以减少存储空间的消耗。
    - 在release_date表中，我们使用varchar类型的数据来存储year, month, day, weekday等字段。这允许存储非标准日期格式或混合格式数据直接以字符串形式存储日期时间等字段，也可以避免不同应用程序间日期格式解析的兼容性问题。对于这些字段，我们也建立了索引，加速了查询。
3. 冗余存储

- 在电影表中存储相关评论数，在查询最受欢迎的演员组合时，可以避免与电影评论表进行join操作，由于电影评论表数据量巨大，所以会节省大量的时间
- 将时间单独抽象成一张表，将时间戳存储为不同数据格式，便于查询时直接进行匹配而不用处理数据，节省查询时间
- 我们对于评论的情感进行了两次优化，第一次将评论的情感单独抽象为一张表，避免根据情感进行查询时与原来的评论表进行join操作，但是由于是百万数量级，直接在电影表中增加一个字段，表示该电影中对应

4. 建立索引

除了主键和外键等数据库自动建立的索引外，对查询操作中需要用到的字段建立单独的单列索引和组合索引，便于快速进行查询操作

- 演员表中为演员姓名建立单列索引
- 导演表中为导演姓名建立单列索引
- 风格表中为电影风格建立单列索引
- 时间表中为年份建立单列索引，为年和月建立组合索引，为年和季度建立组合索引，为工作日建立单列索引
- 版本表中为电影语言建立单列索引，为电影格式建立单列索引

5. 建立视图

- 建立演员和演员之间合作的视图，字段包括演员1的id、演员2的id以及合作的电影id
- 建立演员和导演之间合作的视图，字段包括演员id、导演id以及合作的电影id

分布式文件系统存储模型及优化

schema定义文件

```
CREATE DATABASE IF NOT EXISTS dw_movie;

CREATE EXTERNAL TABLE IF NOT EXISTS dw_movie.act (
  movie_id STRING,
  actor_id INT,
  is_main_actor INT
)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde'
WITH SERDEPROPERTIES (
  "separatorChar" = ",",
  "quoteChar"     = '\"'
)
STORED AS TEXTFILE
```

```
location '/dw_movie/act'
tblproperties ("skip.header.line.count"="1");

CREATE EXTERNAL TABLE IF NOT EXISTS dw_movie.actor (
  actor_id INT ,
  actor_name STRING
)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde'
WITH SERDEPROPERTIES (
  "separatorChar" = ",",
   "quoteChar"     = '\"'
)
STORED AS TEXTFILE
location '/dw_movie/actor'
tblproperties ("skip.header.line.count"="1");

CREATE EXTERNAL TABLE IF NOT EXISTS dw_movie.direct (
  movie_id STRING ,
  director_id INT
)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde'
WITH SERDEPROPERTIES (
  "separatorChar" = ",",
   "quoteChar"     = '\"'
)
STORED AS TEXTFILE
location '/dw_movie/direct'
tblproperties ("skip.header.line.count"="1");

CREATE EXTERNAL TABLE IF NOT EXISTS dw_movie.director (
  director_id INT,
  director_name STRING
)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde'
WITH SERDEPROPERTIES (
  "separatorChar" = ",",
  "quoteChar"     = '\"'
)
STORED AS TEXTFILE
location '/dw_movie/director'
tblproperties ("skip.header.line.count"="1");

CREATE EXTERNAL TABLE IF NOT EXISTS dw_movie.genre (
  genre_id INT ,
  movie_id STRING ,
  genre_name STRING
)
```

```
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde'
WITH SERDEPROPERTIES (
  "separatorChar" = ",",
  "quoteChar"     = '\"'
)
STORED AS TEXTFILE
location '/dw_movie/genre'
tblproperties ("skip.header.line.count"="1");

CREATE EXTERNAL TABLE IF NOT EXISTS dw_movie.release_date (
  date_id INT ,
  movie_id STRING,
  `year` STRING ,
  `month` STRING ,
  `day` STRING ,
  season STRING ,
  `weekday` STRING ,
  week_num STRING ,
  `date` STRING
)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde'
WITH SERDEPROPERTIES (
  "separatorChar" = ",",
  "quoteChar"     = '\"'
)
STORED AS TEXTFILE
location '/dw_movie/release_date'
tblproperties ("skip.header.line.count"="1");

CREATE EXTERNAL TABLE IF NOT EXISTS dw_movie.review (
  review_id INT ,
  movie_id STRING,
  helpfulness STRING ,
  profileName STRING ,
  score FLOAT ,
  `time` BIGINT ,
  summary STRING ,
  `text` STRING
)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde'
WITH SERDEPROPERTIES (
  "separatorChar" = ",",
  "quoteChar"     = '\"'
)
STORED AS TEXTFILE
location '/dw_movie/review'
tblproperties ("skip.header.line.count"="1");
```

```
108
109 CREATE EXTERNAL TABLE IF NOT EXISTS dw_movie.version (
110   version_id INT ,
111   movie_id STRING,
112   edition STRING ,
113   language STRING ,
114   format STRING
115 )
116 ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde'
117 WITH SERDEPROPERTIES (
118   "separatorChar" = ",",
119   "quoteChar"     = '\"'
120 )
121 STORED AS TEXTFILE
122 location '/dw_movie/version'
123 tblproperties ("skip.header.line.count"="1");
124
125 CREATE EXTERNAL TABLE IF NOT EXISTS dw_movie.movie (
126   movie_id STRING ,
127   movie_title STRING ,
128   movie_score FLOAT ,
129   imdb_score FLOAT ,
130   runtime STRING ,
131   review_num INT
132 )
133 ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde'
134 WITH SERDEPROPERTIES (
135   "separatorChar" = ",",
136   "quoteChar"     = '\"'
137 )
138 STORED AS TEXTFILE
139 location '/dw_movie/movie'
140 tblproperties ("skip.header.line.count"="1");
141
142 CREATE EXTERNAL TABLE IF NOT EXISTS dw_movie.actor_actor (
143   LEFT_PERSON_ID INT ,
144   RIGHT_PERSON_ID INT ,
145   MOVIE_ID STRING
146 )
147 ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde'
148 WITH SERDEPROPERTIES (
149   "separatorChar" = ",",
150   "quoteChar"     = '\"'
151 )
152 STORED AS TEXTFILE
153 location '/dw_movie/actor_actor'
154 tblproperties ("skip.header.line.count"="1");
```

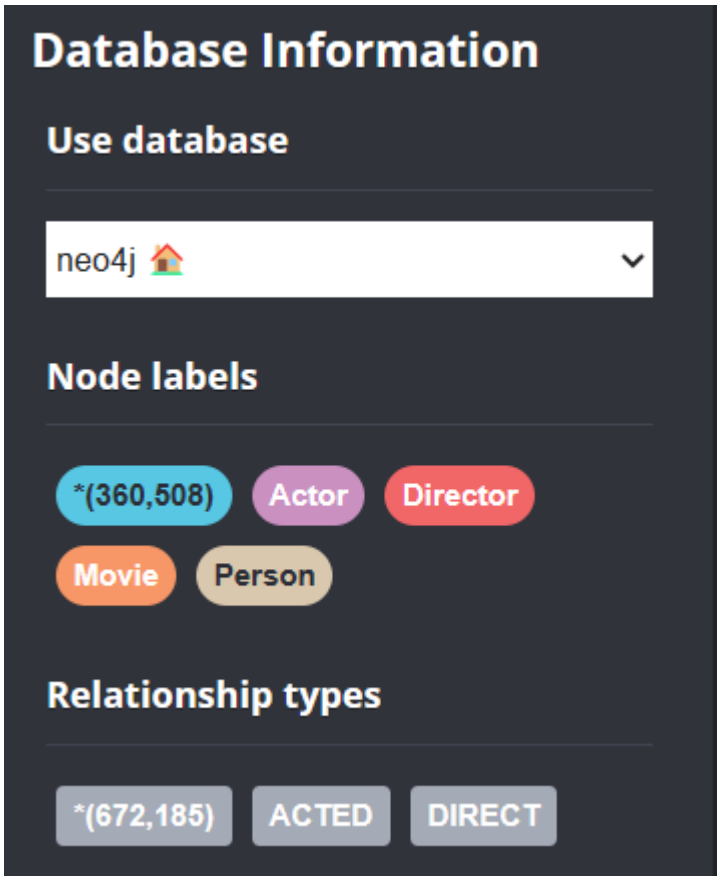
```
155
156 CREATE EXTERNAL TABLE IF NOT EXISTS dw_movie.director_actor (
157   MOVIE_ID STRING ,
158   ACTOR_ID INT ,
159   DIRECTOR_ID INT
160
161 )
162 ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde'
163 WITH SERDEPROPERTIES (
164   "separatorChar" = ",",
165   "quoteChar"     = '\"'
166 )
167 STORED AS TEXTFILE
168 location '/dw_movie/director_actor'
169 tblproperties ("skip.header.line.count"="1");
```

优化

1. 将关系型数据库中的视图存储为实际的表，这样不仅可以复用之前的代码，还可以节省演员和演员以及演员和导演之间的join操作，加快查询速度
2. 建立与关系型数据库相同的数据库表，通过冗余字段避免join操作，并建立了相同的索引以提高查询效率
3. 建立外部表用于管理存储在数据库外部的数据，数据库中只存储元数据，而将实际数据存储到文件系统中，这种方式允许直接在数据所在位置进行查询处理，减少了数据移动的需要，避免其数据移动成为查询瓶颈，从而提升查询效率。

图数据库存储模型及优化

存储模型



cypher建表

```
// 在Actor节点上创建约束
CREATE CONSTRAINT FOR (a:Actor) REQUIRE a.actor_id IS UNIQUE;

// 导入Actor数据
LOAD CSV WITH HEADERS FROM 'file:///actor.csv' AS row
CREATE (:Actor {actor_id: toInteger(row.actor_id), actor_name:
row.actor_name});
// 在Director节点上创建约束
CREATE CONSTRAINT FOR (d:Director) REQUIRE d.director_id IS UNIQUE;

// 导入Director数据
LOAD CSV WITH HEADERS FROM 'file:///director.csv' AS row
CREATE (:Director {director_id: toInteger(row.director_id), director_name:
row.director_name});
// 在Movie节点上创建约束
CREATE CONSTRAINT FOR (m:Movie) REQUIRE m.movie_id IS UNIQUE;

// 导入Movie数据
LOAD CSV WITH HEADERS FROM 'file:///movie.csv' AS row
CREATE (:Movie {movie_id: row.ASIN, movie_title: row.Title, movie_genre:
row.Genres, movie_review_num: row.review_num});

// 建立关系
```

```
// 导入演员参演关系
LOAD CSV WITH HEADERS FROM 'file:///act.csv' AS row
MATCH (a:Actor {actor_id: toInteger(row.actor_id)}), (m:Movie {movie_id:
row.movie_id})
CREATE (a)-[:ACTED {is_main_actor: row.is_main_actor}]->(m);
// 导入导演执导关系
LOAD CSV WITH HEADERS FROM 'file:///direct.csv' AS row
MATCH (a:Director {director_id: toInteger(row.director_id)}), (m:Movie
{movie_id: row.movie_id})
CREATE (a)-[:DIRECT]->(m);
```

优化

1. 存储字段选择
   - 在数据库中只存储三类节点(演员、导演、电影)以及两种关系(导演与电影间的执导关系和演员与电影之间的参演关系)，便于查询时快速进行计算
   - 在数据库中电影节点中存储电影相关的评论数，便于统计
2. 建立索引
   - 对查询时三类节点中将会用到的字段建立索引，分别是演员名、演员名、电影名、电影风格

数据表的test case

对电影的组合查询

我们可以针对下面的筛选条件对电影进行查询：

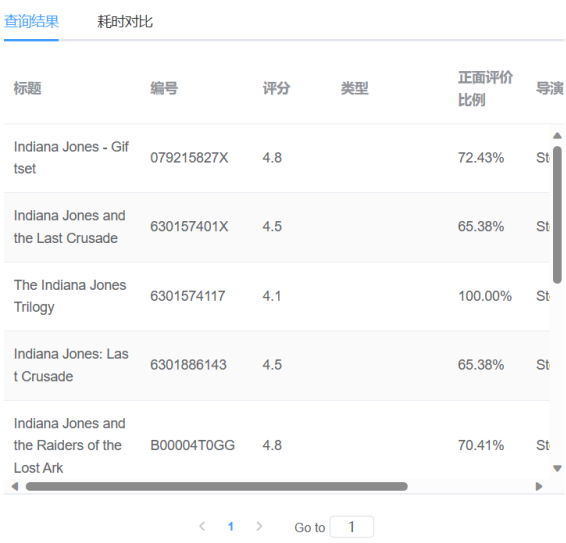
电影标题	title
上映日期	year、month、day、season、weekday
导演	director
演员	actor
风格	genre
评分	min_score、max_score

在查询时上述字段参数都**可空**，表示不作为筛选条件。

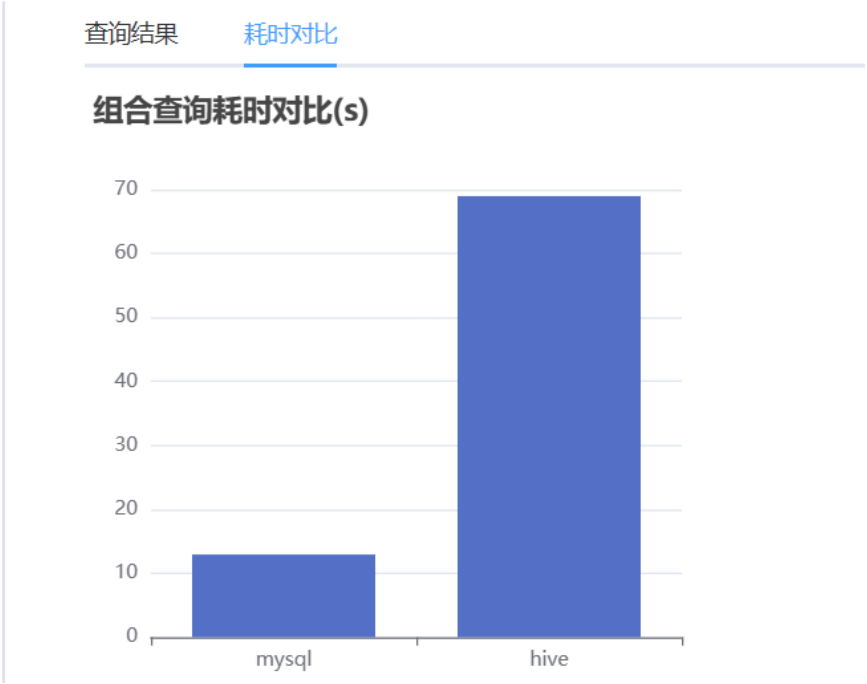
我们可以根据上述筛选条件，筛选出符合用户输入的筛选条件的电影的下列字段：

编号	asin
电影标题	title
电影版本数	edition
电影版本	format
电影风格	genre
电影评分	score
上映日期	date
导演	director_name
演员	actor_name
时长	runtime

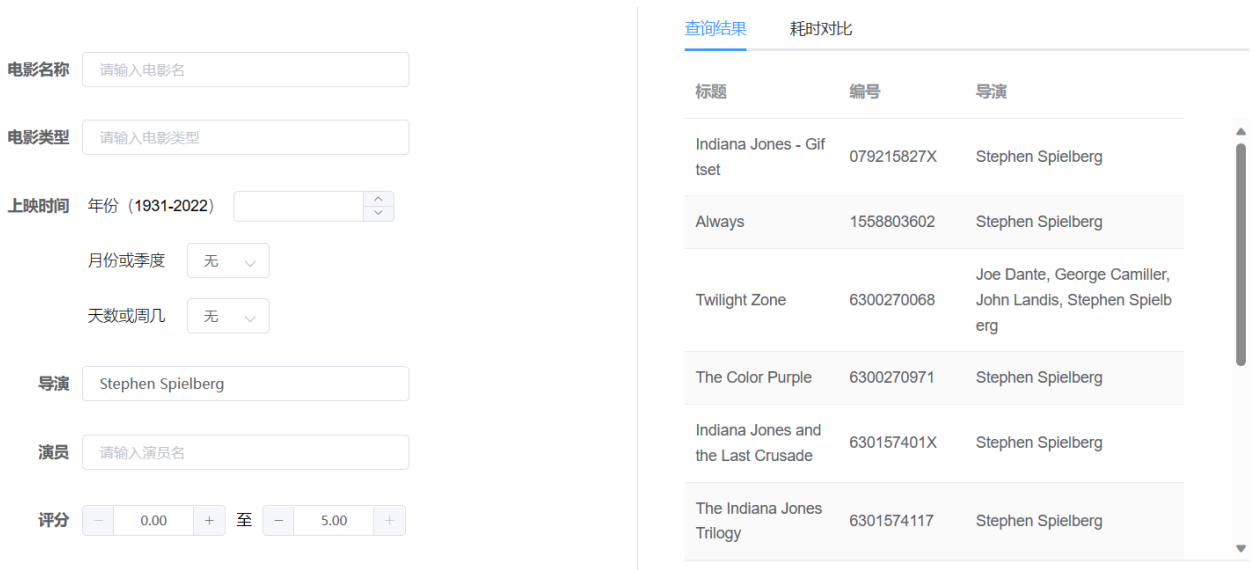
1. 组合查询示例一：查询导演为Stephen Spilberg，电影名称含Indiana，正面评价比例大于40%的电影，查询结果如下：



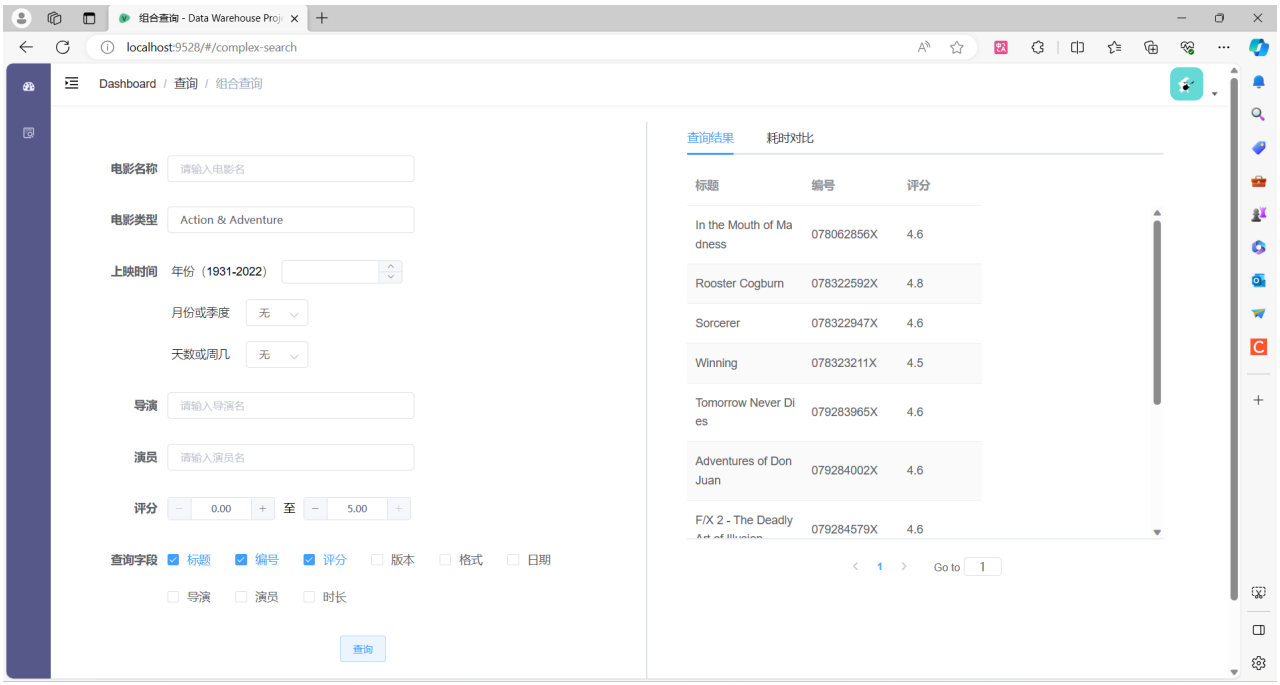
耗时对比如下：



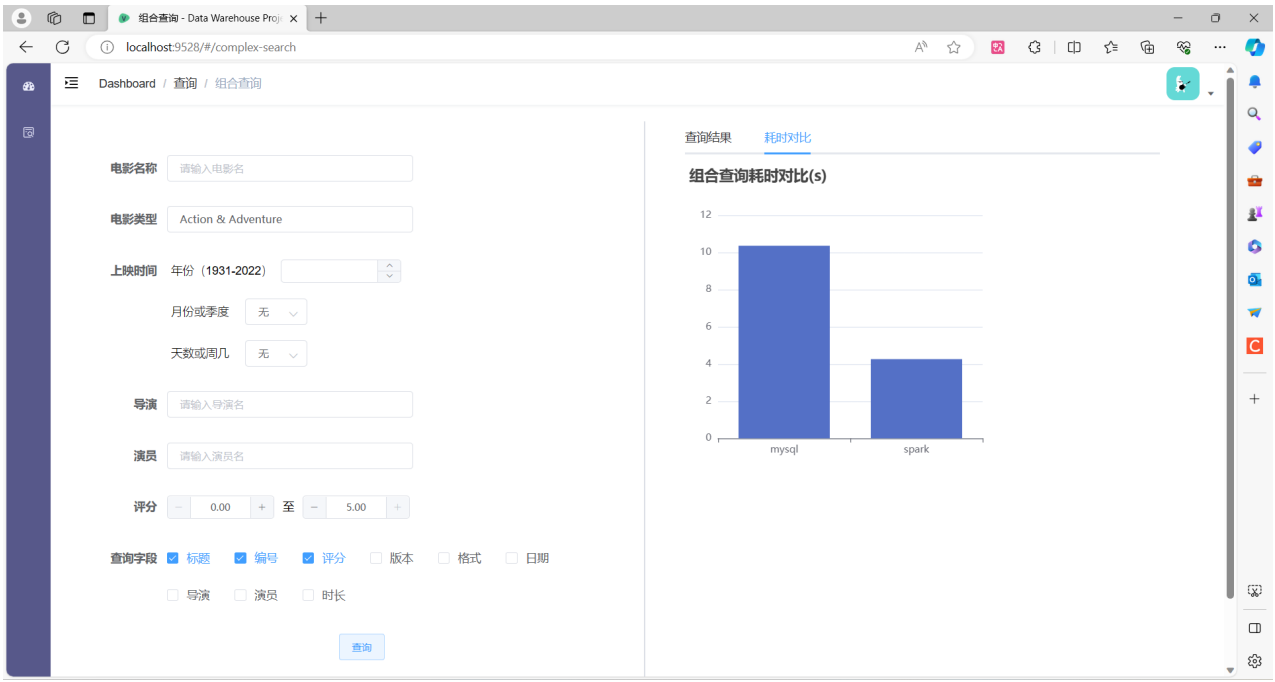
2. 按照导演进行查询及统计，比如查询所有Stephen Spielberg导演的电影：



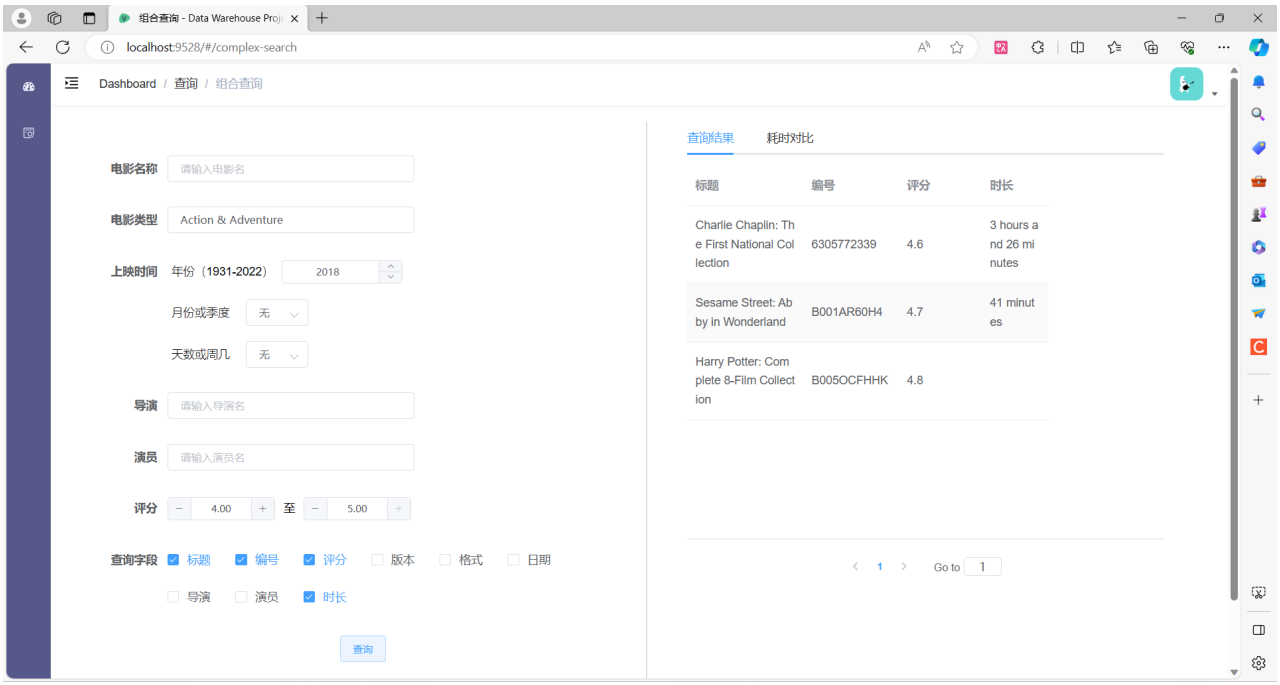
3. 对于较简单的查询，分布式数据库的表现明显好于关系型数据库，比如查询所有电影类型为Action & Adventure类型的电影：



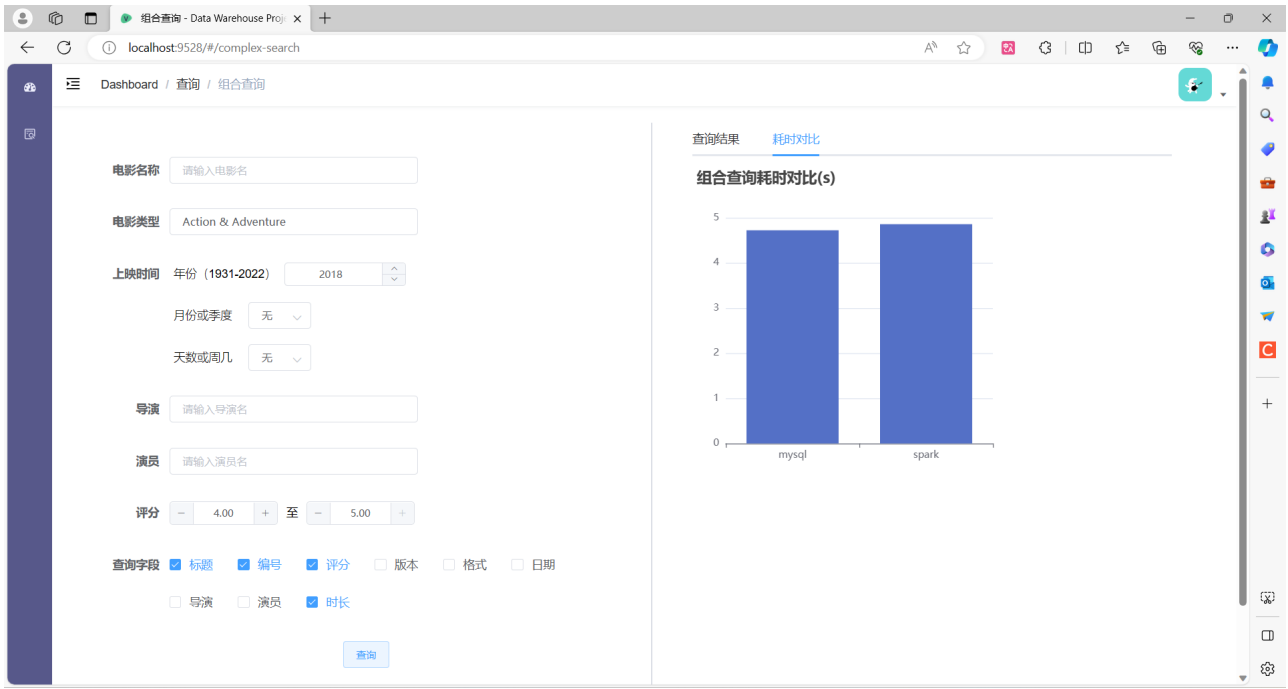
查询时间对比如下：



4. 组合查询示例二：查询2018年评分4-5分，电影类型为Action & Adventure类型的电影，查询结果如下：



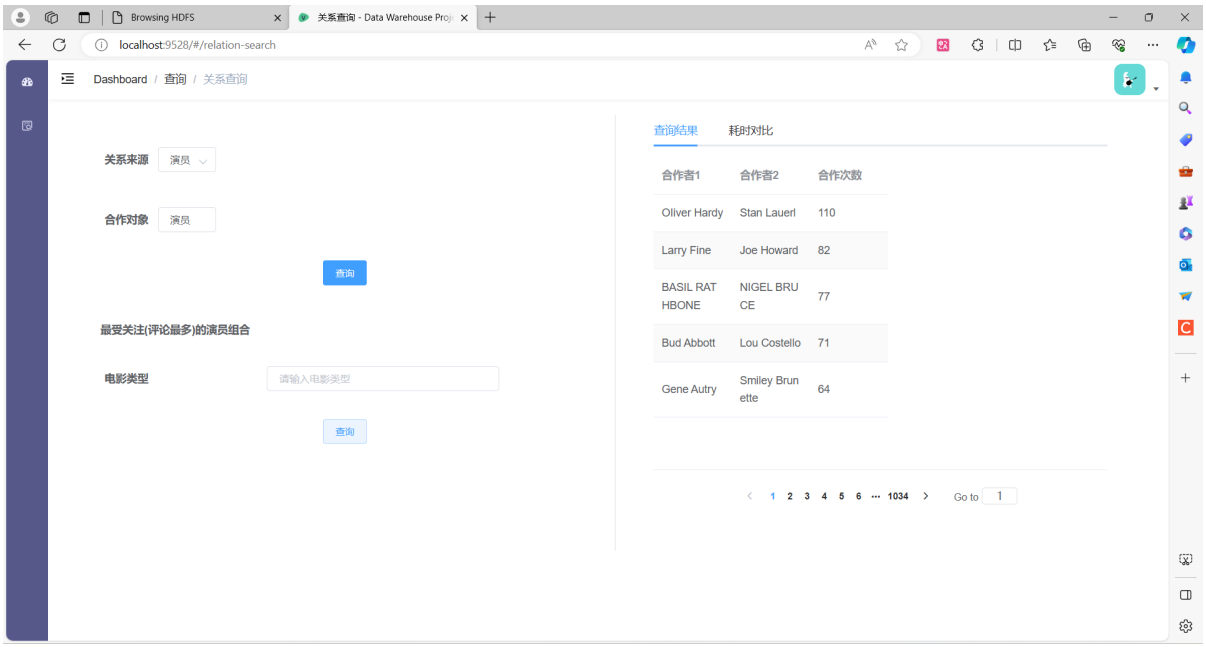
查询时间对比如下：



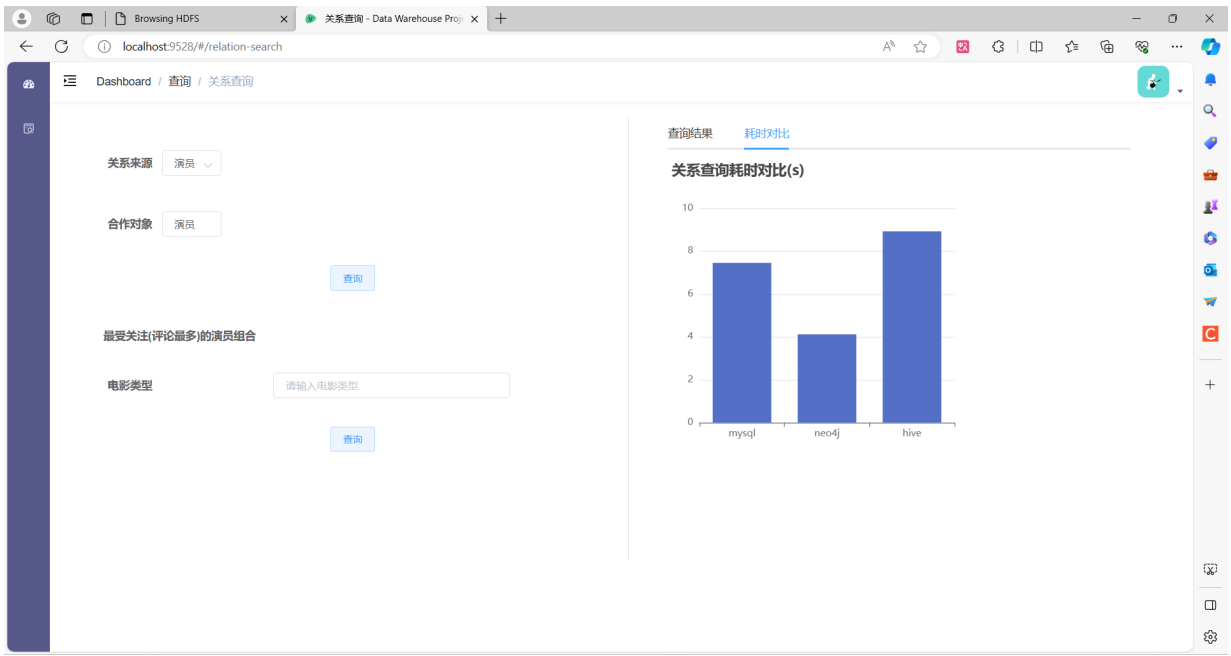
我们发现，当查询字段需要跨过多张表时，当前分布式数据库的表现并不如mysql，可能是由于分布式数据库的join操作查询效率太低。

关系之间关系的查询

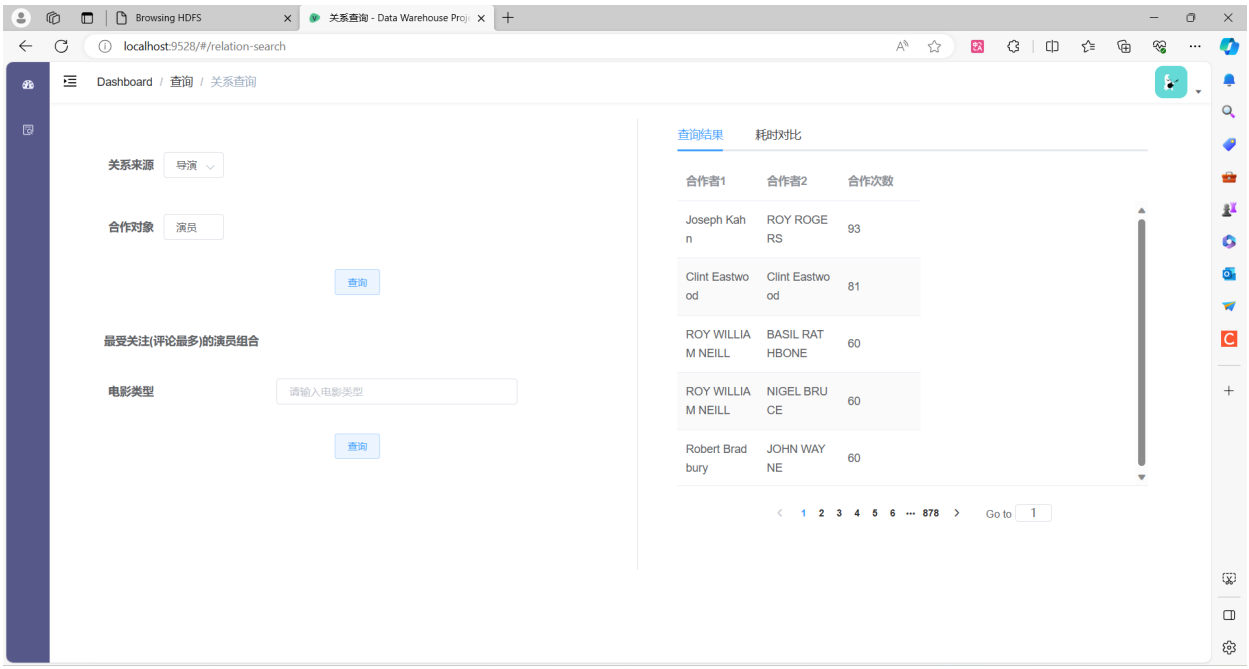
1. 查询合作次数较多的演员，查询结果如下：



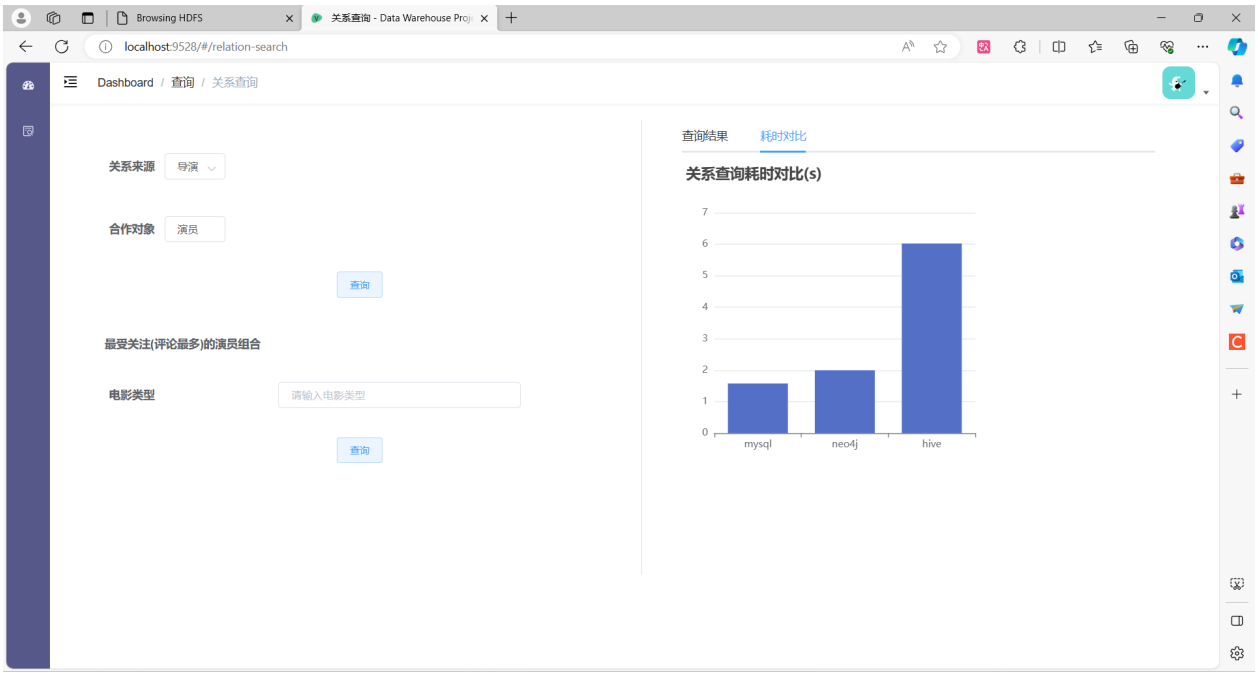
查询时间对比如下，可以看出neo4j确实更适合这种演员关系间的查询，mysql关系型数据库次之，分布式数据库由于需要较多的join操作，因此耗时较长。



2. 查询合作次数较多的演员和导演，查询结果如下：

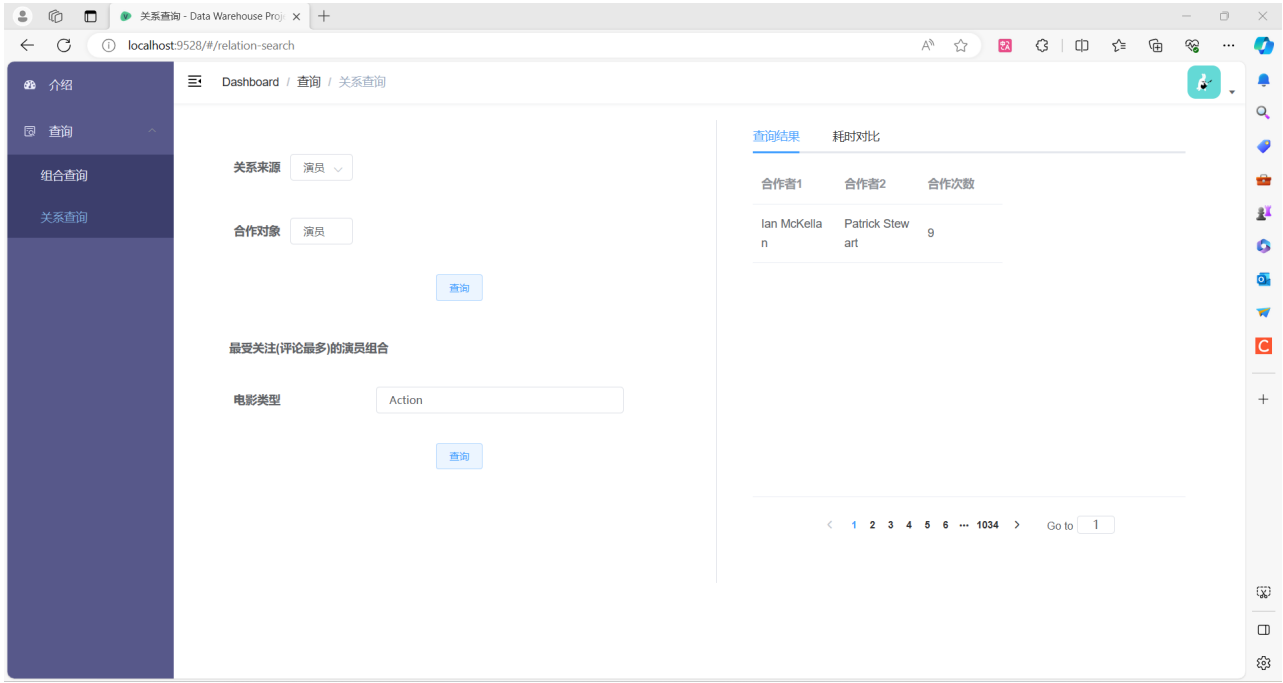


查询时间对比如下，结论基本一致，neo4j更适合这种演员关系间的查询，mysql关系型数据库次之，分布式数据库需要较多的join操作耗时较长。



3. 某类型的电影，最受关注（评论最多）的演员组合（2人）是什么？

查询结果如下：



耗时对比结论与之前相同，不再赘述。