数据存储设计说明文件

本项目以亚马逊电影评论数据集为基础，针对电影信息，构建了一个包含关系型数据库（MySQL）、分布式数据库（Hadoop+Hive）和图数据库（Neo4j）的数据仓库系统，实施了数据治理框架，并进行了反范式化设计。该系统能够高效执行综合条件查询和演员导演合作关系查询，同时支持不同数据库之间的查询性能对比。

1. 整体存储模式

1.1 关系型存储

本项目中，我们使用 MySQL 进行关系型数据库的存储，存储了所有电影相关的信息，包括评论、演员与导演等，以支持所有查询。关系型数据库中，所有表的存储说明如下：

1. **Movie**: 存储电影名称、电影标题、电影评分与电影总评论数
2. **Actor 与 Act**: 存储演员名字与参演电影
3. **Director 与 Direct**: 存储导演名字与导演的电影
4. **Genre**: 存储电影的风格
5. **ReleaseDate**: 存储电影的上映时间相关的信息
6. **Review**: 存储所有电影评论相关的信息
7. **Version**: 存储电影相关的版本、语言等信息

本次项目中，我们通过对物理模型的反规范化、预聚合以及设置冗余存储等操作减少了部分 join 操作，对存储进行了优化。例如为 Movie 表建立冗余字段 `review_num` 以减少与 `Review` 表的联接查询操作，提高了查询效率。
对于电影的上映时间，我们将电影的上映时间拆分为年、月、日、星期等单独的字段，减少查询时对时间的计算次数，从而提高了根据上映时间查询电影的效率。
此外，为方便查询演员与导演间以及演员之间的关系，我们分别建立了记录演员间关系的 `ActorActor` 视图与记录演员与导演之间关系的 `DirectorActor` 视图，简化了对这些关系的查询操作。

1.2 分布式存储

在本项目中，采用 Hive 和 Hadoop 作为分布式数据库，使用1个 Namenode 和1个 Datanode 的伪分布式架构，并以 MapReduce 作为计算引擎，HDFS 作为存储系统。集群的元数据存储使用 PostgreSQL，并配置在 Docker 上，同时尝试集成 SparkSQL 计算框架。

我们的分布式数据库的表结构与关系型数据库相似，但将原本的关系型数据库视图 `actor_actor` 和 `director_actor` 转换为实际的外部表，这样加速了演员之间以及演员与导演之间关系的查询。

所有的分布式数据库表都使用外部表，这对于大数据处理场景是一种优化，因为可以在数据本地进行查询处理，从而减少数据移动的需求。尽管分布式数据库在增加数据时可能表现不佳，但由于本项目主要涉及数据查询，数据增加所带来的问题并不需要考虑。

1.3 图数据存储

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

节点类型（Node Labels）：

1. **Actor（演员）**
   - `actor_uuid`：演员的唯一标识符（UUID）
   - `actor_name`：演员的名字
2. **Director（导演）**
   - `director_uuid`: 导演的唯一标识符（UUID）
   - `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_review_num`: 电影的评论数量

4. **Genre（电影类型）**
   - `genre_uuid`：类型的唯一标识符（UUID）
   - `genre_name`：类型名称

关系类型（Relationship Types）：

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
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
3. **HAS_GENRE（拥有类型）**
   - 连接电影(Movie)和类型(Genre)
   - 表示一部电影属于某种类型或流派

2. 关系型存储模型

本项目以亚马逊电影评论数据集作为数据基础，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，以支持多种类的查询和统计，包括综合条件查询和演员导演之间的关系查询。

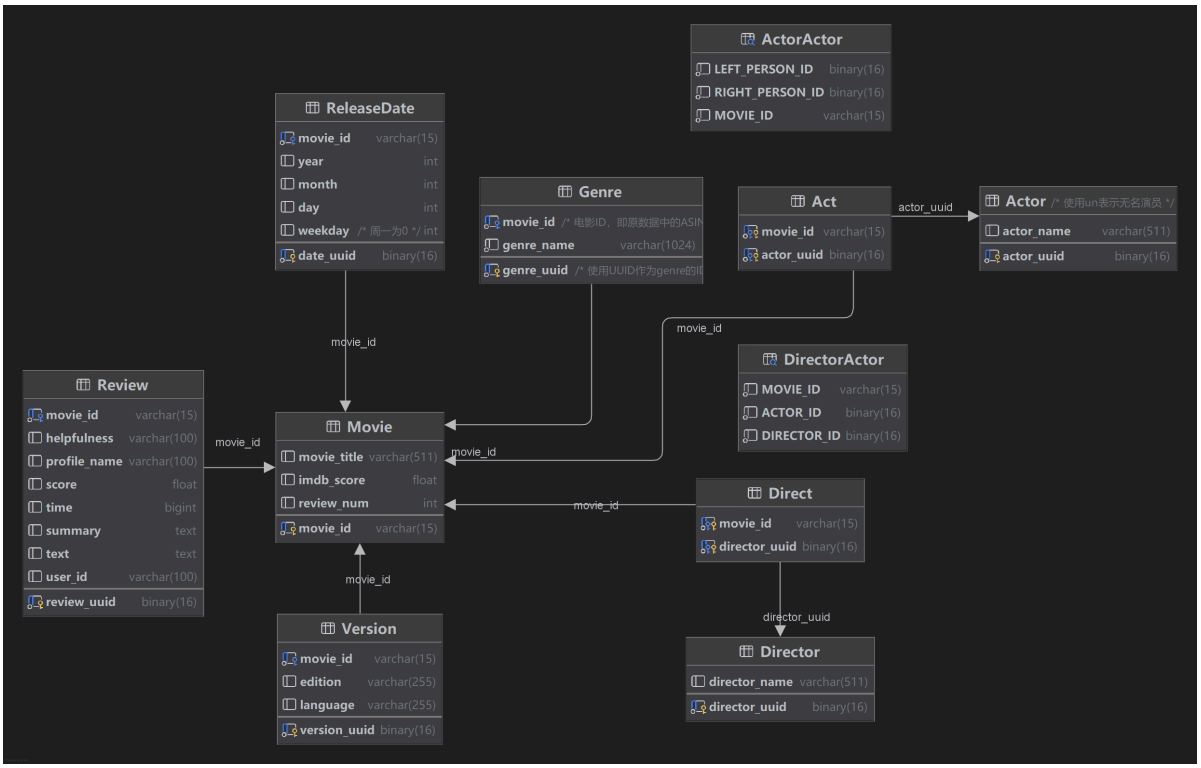
2.1 关系型存储逻辑模型（LDM设计）

2.1.1 数据表结构

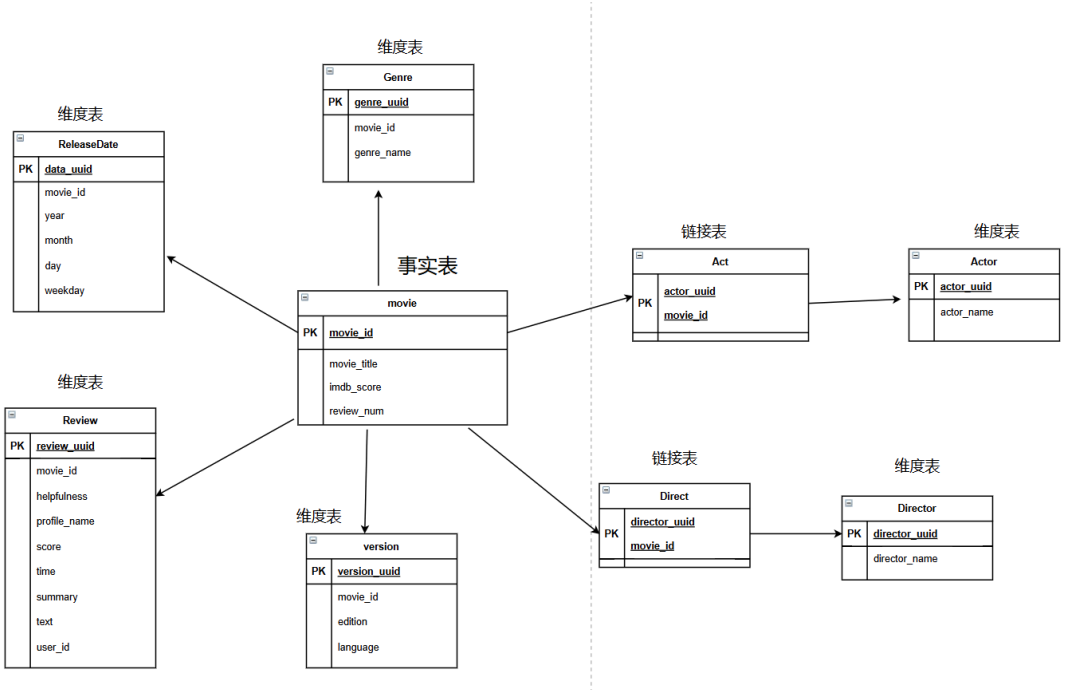
表名	存储内容
Movie	电影id，电影标题，电影评分，电影总评论数
Actor 和 Act	演员的名字及参演的电影
Director 和 Direct	导演的名字及导演的电影
Genre	电影的风格
Release_date	电影上映时间相关的信息
Review	所有评论相关的信息
Version	电影相关的版本、语言信息

表名	引擎	自增	数据长度	Partitioned	描述
Act	InnoDB	0	38M	[ ]	
Actor	InnoDB	0	12M	[ ]	使用un表示无名...
Direct	InnoDB	0	13M	[ ]	
Director	InnoDB	0	1.5M	[ ]	
Genre	InnoDB	0	14M	[ ]	
Movie	InnoDB	0	23M	[ ]	
ReleaseDate	InnoDB	0	19M	[ ]	
Review	InnoDB	0	5.9G	[ ]	
Version	InnoDB	0	21M	[ ]	

2.1.2 E-R图



2.1.3 星型模型图



2.2 关系型存储物理模型（PDM设计）

2.2.1 DDL

```
CREATE DATABASE `dw2024`;

CREATE TABLE `Movie` (
  `movie_id` varchar(15) NOT NULL,
  `movie_title` varchar(511) CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci
DEFAULT NULL,
  `imdb_score` float DEFAULT NULL,
  `review_num` int DEFAULT '0',
  PRIMARY KEY (`movie_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

CREATE TABLE `Review` (
  `review_uuid` binary(16) NOT NULL,
  `movie_id` varchar(15) CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci NOT
NULL,
  `helpfulness` varchar(100) CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci
DEFAULT NULL,
  `profile_name` varchar(100) DEFAULT NULL,
  `score` float DEFAULT NULL,
  `time` bigint DEFAULT NULL,
  `summary` text,
  `text` text,
  `user_id` varchar(100) DEFAULT NULL,
  PRIMARY KEY (`review_uuid`),
  KEY `Review_Movie_FK` (`movie_id`),
  CONSTRAINT `Review_Movie_FK` FOREIGN KEY (`movie_id`) REFERENCES `Movie`
(`movie_id`) ON DELETE CASCADE ON UPDATE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

CREATE TABLE `Version` (
  `version_uuid` binary(16) NOT NULL,
  `movie_id` varchar(15) CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci NOT
NULL,
  `edition` varchar(255) DEFAULT NULL,
  `language` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`version_uuid`),
  KEY `Version_Movie_FK` (`movie_id`),
  CONSTRAINT `Version_Movie_FK` FOREIGN KEY (`movie_id`) REFERENCES `Movie`
(`movie_id`) ON DELETE CASCADE ON UPDATE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

CREATE TABLE `ReleaseDate` (
  `date_uuid` binary(16) NOT NULL,
  `movie_id` varchar(15) CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci NOT
NULL,
  `year` int DEFAULT NULL,
  `month` int DEFAULT NULL,
  `day` int DEFAULT NULL,
  `weekday` int DEFAULT NULL COMMENT '周一为0',
  PRIMARY KEY (`date_uuid`),
  KEY `RleaseDate_Movie_FK` (`movie_id`),
```

```
  CONSTRAINT `RleaseDate_Movie_FK` FOREIGN KEY (`movie_id`) REFERENCES `Movie`
(`movie_id`) ON DELETE CASCADE ON UPDATE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

CREATE TABLE `Genre` (
  `genre_uuid` binary(16) NOT NULL COMMENT '使用UUID作为genre的ID',
  `movie_id` varchar(15) CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci NOT
NULL COMMENT '电影ID，即原数据中的ASIN字段',
  `genre_name` varchar(1024) CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci NOT
NULL,
  PRIMARY KEY (`genre_uuid`),
  KEY `Genre_Movie_FK` (`movie_id`),
  CONSTRAINT `Genre_Movie_FK` FOREIGN KEY (`movie_id`) REFERENCES `Movie`
(`movie_id`) ON DELETE CASCADE ON UPDATE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

CREATE TABLE `Actor` (
  `actor_uuid` binary(16) NOT NULL,
  `actor_name` varchar(511) CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci
DEFAULT NULL,
  PRIMARY KEY (`actor_uuid`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci COMMENT='使用un
表示无名演员';

CREATE TABLE `Act` (
  `movie_id` varchar(15) NOT NULL,
  `actor_uuid` binary(16) NOT NULL,
  PRIMARY KEY (`actor_uuid`,`movie_id`),
  KEY `Act_Movie_FK` (`movie_id`),
  CONSTRAINT `Act_Actor_FK` FOREIGN KEY (`actor_uuid`) REFERENCES `Actor`
(`actor_uuid`) ON DELETE CASCADE ON UPDATE CASCADE,
  CONSTRAINT `Act_Movie_FK` FOREIGN KEY (`movie_id`) REFERENCES `Movie`
(`movie_id`) ON DELETE CASCADE ON UPDATE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

CREATE TABLE `Director` (
  `director_uuid` binary(16) NOT NULL,
  `director_name` varchar(511) CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci
DEFAULT NULL,
  PRIMARY KEY (`director_uuid`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

CREATE TABLE `Direct` (
  `movie_id` varchar(15) NOT NULL,
  `director_uuid` binary(16) NOT NULL,
  PRIMARY KEY (`director_uuid`,`movie_id`),
  KEY `Direct_Movie_FK` (`movie_id`),
  CONSTRAINT `Direct_Director_FK` FOREIGN KEY (`director_uuid`) REFERENCES
`Director` (`director_uuid`) ON DELETE CASCADE ON UPDATE CASCADE,
  CONSTRAINT `Direct_Movie_FK` FOREIGN KEY (`movie_id`) REFERENCES `Movie`
(`movie_id`) ON DELETE CASCADE ON UPDATE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
```

2.2.2 日志表

设计一个如下的 `query_log` 表：

字段名	数据类型	说明
`log_id`	INT (PK, AUTO_INCREMENT)	日志ID，自增
`query_time`	DATETIME	查询执行的时间
`query_text`	TEXT	用户执行的查询语句
`query_params`	TEXT	查询时的参数，以JSON格式存储
`rows_returned`	INT	返回的行数
`query_type`	VARCHAR(50)	查询类型（如：统计，过滤等）
`result_status`	VARCHAR(20)	查询结果状态（成功、失败、超时等）
`error_message`	TEXT	错误信息（若查询失败时）

2.2.3 存储优化设计

a. 星型模型使用

本次项目采用星型模型结构，在数据存储上我们通过引入一定的数据冗余，换取了更好的查询性能。通过围绕事实表 `Movie` 建立一系列的维度表，将 `Movie` 的属性拆分到不同维度表中，简化了查询的复杂度，优化了查询的性能。

b. 字段设置

在 `Review` 表中，我们将记录评论时间信息的 `time` 时间戳字段的类型设计为了 `bigint`，而不是常见的 `datetime` 类型。这一决策主要基于以下考量：

- **平台兼容性**：采用 `bigint` 类型确保了数据能在不同数据库系统与编程语言之间保持一致性与互操作性。
- **空间效率**：相较于 `datetime` 类型，`bigint` 所需的存储空间更小，这对于规模较大的评论表 `Review` 而言尤为重要，能够有效降低存储成本。

而在 `Movie` 表中，我们将评分相关的 `imdb_score` 字段设计为了 `float` 类型，这样的选择主要出于如下考量：

- **节省资源**：相较与 `double` 类型，`float` 所占用的存储空间更小，有助于优化存储资源的利用。
- **精度适配**：考虑到电影评分并不需要较高的数值精度，`float` 类型足以满足需求，也可以避免不必要的复杂性。

c. 冗余存储

在电影表中，我们添加了 `review_num` 评论数量字段，通过添加这一冗余存储，避免在查询最受欢迎的演员组合时与评论表进行联接操作，从而提升查询性能。

此外我们将电影的发布时间单独抽象为一张表，将电影的发布时间拆分为多个字段，而不是直接存储电影发布时间的时间戳，这样在分别根据年、月、日查询电影时，可以减少对数据的处理，简化了操作流程，也提高了查询的性能。

d. 建立索引

除了主键和外键等数据库自动建立的索引外，对常用查询字段建立单列索引和组合索引，以加速查询操作。

- **单列索引**
  - 演员表：为 `演员姓名` 建立单列索引。
  - 导演表：为 `导演姓名` 建立单列索引。
  - 风格表：为 `电影风格` 建立单列索引。
  - 时间表：为 `年份` 建立单列索引。
  - 版本表：为 `电影语言` 和 `电影格式` 分别建立单列索引。
- **组合索引**
  - 时间表：
    - 年和月建立组合索引。
    - 年和季度建立组合索引。
    - 工作日建立单列索引。

e. 建立视图

- **演员合作视图**
  建立演员与演员之间合作的视图，字段包括：
  - 演员1的 `ID`
  - 演员2的 `ID`
  - 合作电影的 `ID`
- **演员与导演合作视图**
  建立演员与导演之间合作的视图，字段包括：
  - 演员的 `ID`
  - 导演的 `ID`
  - 合作电影的 `ID`

2.2.4 PreAggregation 表设计

为了进一步提升查询性能，我们设计了一些 `PreAggregation` 表，将耗时的计算过程放在服务器负载较小时进行。

在 `MovieGenreStatistics` 表中，我们提前汇总了各电影类别的电影总数，以加速查询某类别的电影统计数据。

字段名	数据类型	说明
`genre`	VARCHAR(100)	电影类别
`total_movies`	INT	该类别电影的总数

在 `MovieActorDirectorCollaborations` 表中，我们提前计算了演员与导演之间的合作次数，以便快速查询哪些演员和导演经常合作。

字段名	数据类型	说明
`actor_id`	BINARY(16) (PK)	演员ID
`actor_name`	VARCHAR(511)	演员姓名
`director_id`	BINARY(16) (PK)	导演ID
`director_name`	VARCHAR(511)	导演姓名
`collaborations`	INT	演员与导演合作的次数

对于如上两张表，我们采用 MySQL 的事件进行更新，在每天凌晨时将执行更新任务，以确保不影响平时业务的执行。

3. 分布式文件系统存储模型及优化

3.1 schema定义文件

```
CREATE DATABASE dw2024;

create external table if not exists dw2024.act
(
    movie_id   string,
    actor_uuid string
)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde'
WITH SERDEPROPERTIES (
    "separatorChar" = ",",
    "quoteChar" = '\"'
)
STORED AS TEXTFILE
LOCATION '/dw2024/Act'
tblproperties ("skip.header.line.count"="1");

create external table if not exists dw2024.actor
(
    actor_uuid string,
    actor_name string
)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde'
WITH SERDEPROPERTIES (
    "separatorChar" = ",",
    "quoteChar" = "\""
)
STORED AS TEXTFILE
LOCATION '/dw2024/Actor/'
tblproperties ("skip.header.line.count"="1");

create external table if not exists dw2024.direct
(
    movie_id      string,
    director_uuid string
)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde'
```

```
WITH SERDEPROPERTIES (
    "separatorChar" = ",",
    "quoteChar" = '\"'
)
STORED AS TEXTFILE
LOCATION '/dw2024/Direct'
tblproperties ("skip.header.line.count"="1");
create external table if not exists dw2024.director
(
    director_uuid string,
    director_name string
)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde'
WITH SERDEPROPERTIES (
    "separatorChar" = ",",
    "quoteChar" = '\"'
)
STORED AS TEXTFILE
LOCATION '/dw2024/Director'
tblproperties ("skip.header.line.count"="1");

create external table if not exists dw2024.genre
(
    genre_uuid string comment '使用UUID作为genre的ID',
    movie_id   string comment '电影ID',
    genre_name string
)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde'
WITH SERDEPROPERTIES (
    "separatorChar" = ",",
    "quoteChar" = '\"'
)
STORED AS TEXTFILE
LOCATION '/dw2024/Genre'
tblproperties ("skip.header.line.count"="1");

create external table if not exists dw2024.movie
(
    movie_id    string,
    movie_title string,
    imdb_score  float,
    review_num  int
)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde'
WITH SERDEPROPERTIES (
    "separatorChar" = ",",
    "quoteChar" = '\"'
)
STORED AS TEXTFILE
LOCATION '/dw2024/Movie'
tblproperties ("skip.header.line.count"="1");

create external table if not exists dw2024.releasedate
(
    date_uuid string,
    movie_id  string,
```

```
    year      int,
    month     int,
    day       int,
    weekday   int comment '周一为0'
)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde'
WITH SERDEPROPERTIES (
    "separatorChar" = ",",
    "quoteChar" = '\"'
)
STORED AS TEXTFILE
LOCATION '/dw2024/ReleaseDate'
tblproperties ("skip.header.line.count"="1");

create external table if not exists dw2024.version
(
    version_uuid string,
    movie_id     string,
    edition      string,
    language     string
)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde'
WITH SERDEPROPERTIES (
    "separatorChar" = ",",
    "quoteChar" = '\"'
)
STORED AS TEXTFILE
LOCATION '/dw2024/Version'
tblproperties ("skip.header.line.count"="1");

CREATE EXTERNAL TABLE IF NOT EXISTS dw2024.actor_actor (
    LEFT_PERSON_ID STRING,
    RIGHT_PERSON_ID STRING,
    MOVIE_ID STRING
)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde'
WITH SERDEPROPERTIES (
    "separatorChar" = ",",
    "quoteChar" = '\"'
)
STORED AS TEXTFILE
LOCATION '/dw2024/ActorActor'
tblproperties ("skip.header.line.count"="1");

CREATE EXTERNAL TABLE IF NOT EXISTS dw2024.director_actor (
    MOVIE_ID STRING,
    ACTOR_ID STRING,
    DIRECTOR_ID STRING
)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde'
WITH SERDEPROPERTIES (
    "separatorChar" = ",",
    "quoteChar" = '\"'
)
STORED AS TEXTFILE
LOCATION '/dw2024/DirectorActor'
```

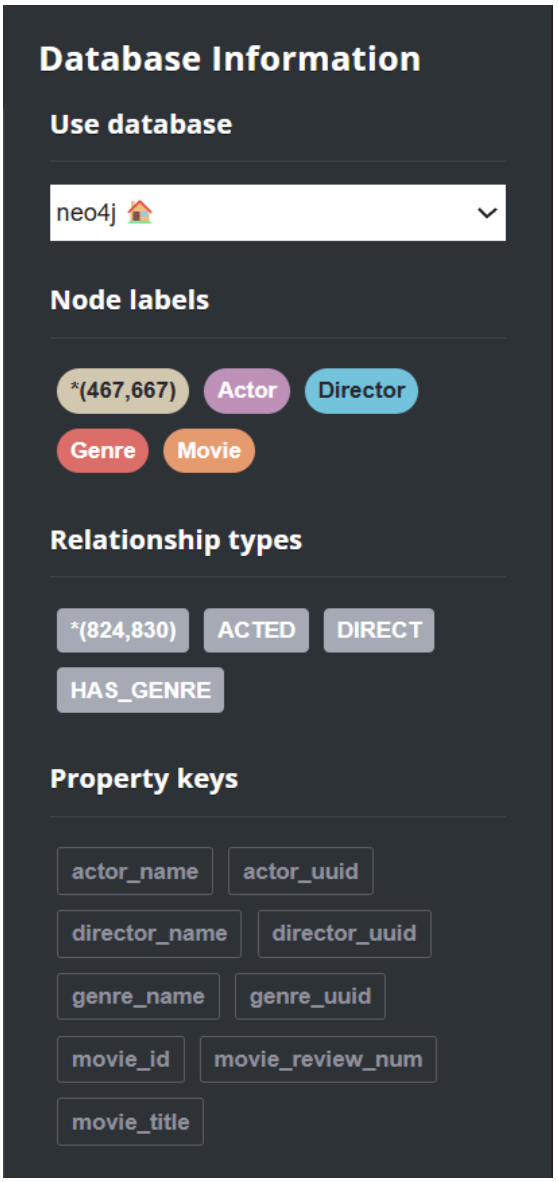
```
tblproperties ("skip.header.line.count"="1");
```

3.2 查询优化

- 将关系型数据库中的视图转为实际表，不仅复用了原有代码，还减少了演员与演员、演员与导演之间的连接操作，从而加速了查询。
- 采用与关系型数据库相同的表结构，通过引入冗余字段避免了复杂的连接操作，并设置了相同的索引以优化查询性能。
- 通过建立外部表来管理存储在外部的数据，数据库仅保存元数据，而实际数据存储在文件系统中。这种方法允许在数据存储位置直接进行查询，减少了数据的传输，提高了查询效率，避免了数据移动可能带来的性能瓶颈。

4. 图数据库存储模型及优化

4.1 存储模型



4.2 cypher建表

```
// 创建节点
// 1.在Actor节点上创建约束
CREATE CONSTRAINT FOR (a:Actor) REQUIRE a.actor_uuid IS UNIQUE;
// 导入Actor数据
LOAD CSV WITH HEADERS FROM 'file:///Actor.csv' AS row
MERGE (a:Actor {actor_uuid: row.actor_uuid})
ON CREATE SET a.actor_name = row.actor_name;

// 2.在Director节点上创建约束
CREATE CONSTRAINT FOR (d:Director) REQUIRE d.director_uuid IS UNIQUE;
//导入Director数据
LOAD CSV WITH HEADERS FROM 'file:///Director.csv' AS row
CREATE(:Director {director_uuid: row.director_uuid,
director_name:row.director_name});

// 3.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.movie_id, movie_title: row.movie_title,
movie_review_num: row.review_num});

// 4.在Genre节点上创建约束
CREATE CONSTRAINT FOR (g:Genre) REQUIRE g.genre_uuid IS UNIQUE;
//导入Genre数据
LOAD CSV WITH HEADERS FROM 'file:///Genre.csv' AS row
CREATE(:Genre {genre_uuid: row.genre_uuid, genre_name: row.genre_name});

// 建立关系
// 1.导入演员参演关系
LOAD CSV WITH HEADERS FROM 'file:///Act.csv' AS row
MATCH (a:Actor {actor_uuid: row.actor_uuid})
WITH a, row
MATCH (m:Movie {movie_id: row.movie_id})
CREATE (a)-[:ACTED]->(m);

// 2.导入导演执导关系
LOAD CSV WITH HEADERS FROM 'file:///Direct.csv' AS row
MATCH (a:Director {director_uuid:row.director_uuid})
WITH a, row
MATCH (m:Movie {movie_id: row.movie_id})
CREATE (a)-[:DIRECT]->(m);

// 3.导入电影风格关系
LOAD CSV WITH HEADERS FROM 'file:///Genre.csv' AS row
MATCH (m:Movie {movie_id: row.movie_id})
WITH m, row
MATCH (g:Genre {genre_uuid: row.genre_uuid})
MERGE (m)-[:HAS_GENRE]->(g);
```

4.3 查询优化

4.3.1 存储字段选择

- 在数据库中只存储**4类节点**（演员、导演、电影、电影类型）以及**3种关系**（导演与电影间的执导关系、演员与电影之间的参演关系、电影类型的拥有关系），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。

4.3.2 建立索引

- 对查询时4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

5. 数据表的Test Case

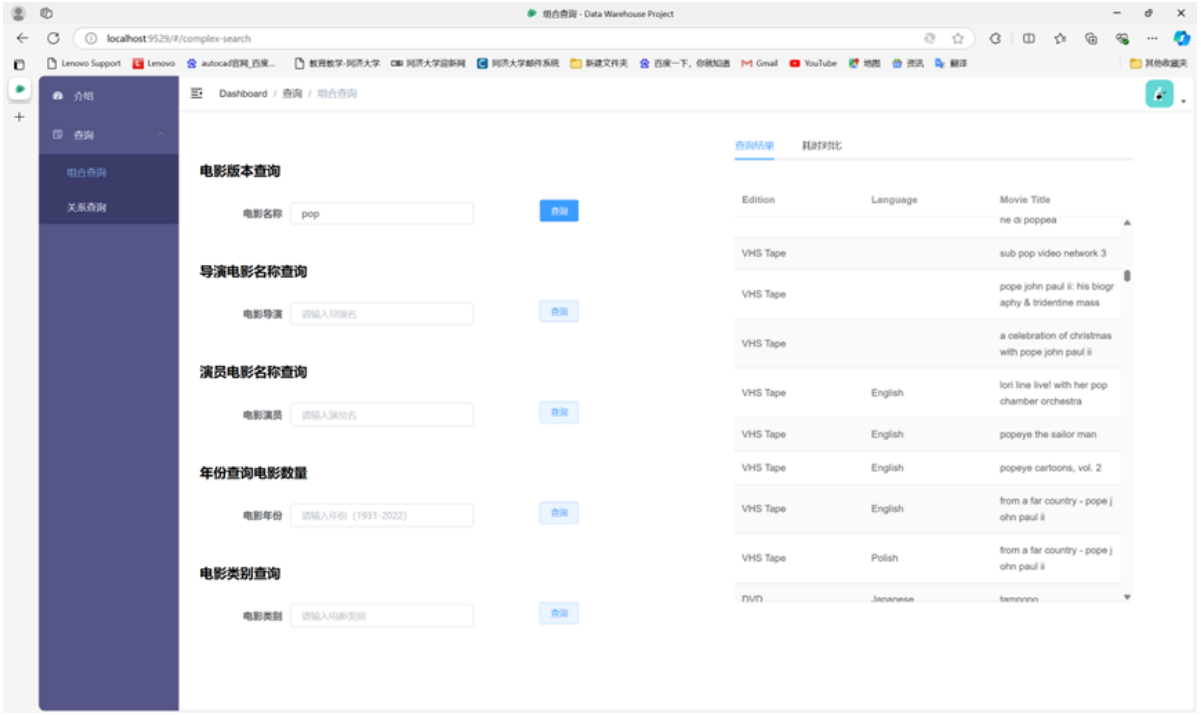
5.1 组合查询

我们可以针对下面的筛选条件对电影进行查询:

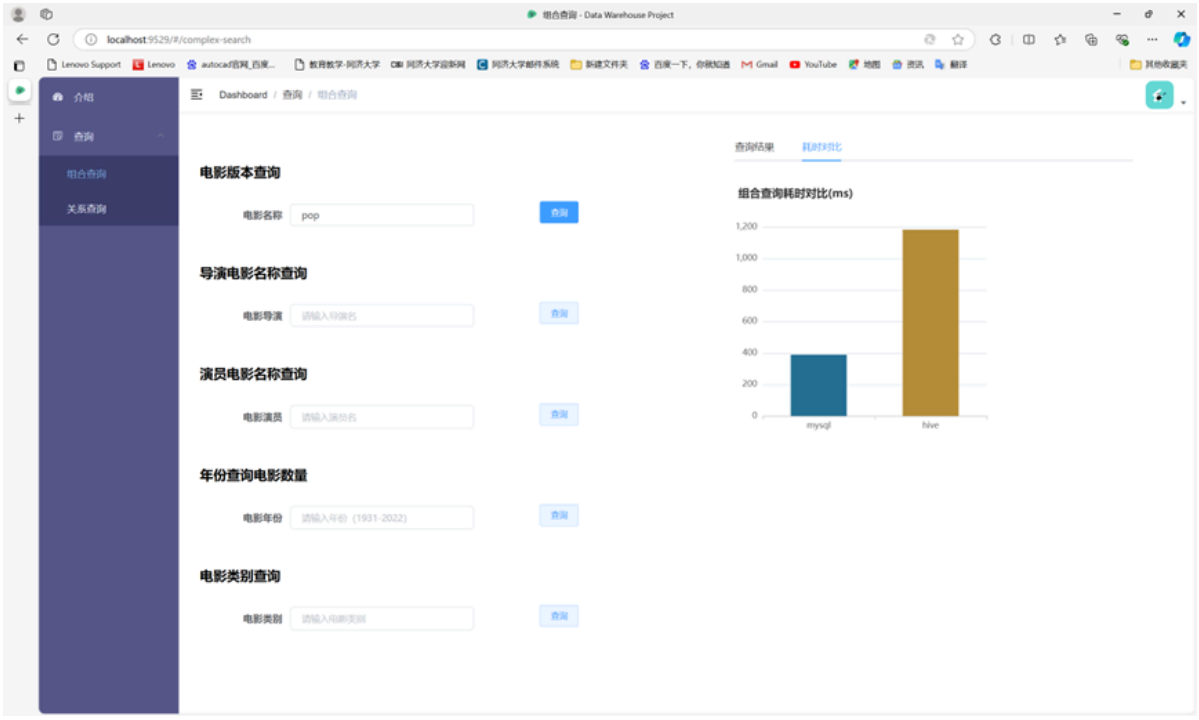
5.1.1 按电影名称查找版本

输入电影的名称，查询该电影的版本。返回电影名称，版本和语言，在右侧查询列表中以表格形式显示。

如输入pop：



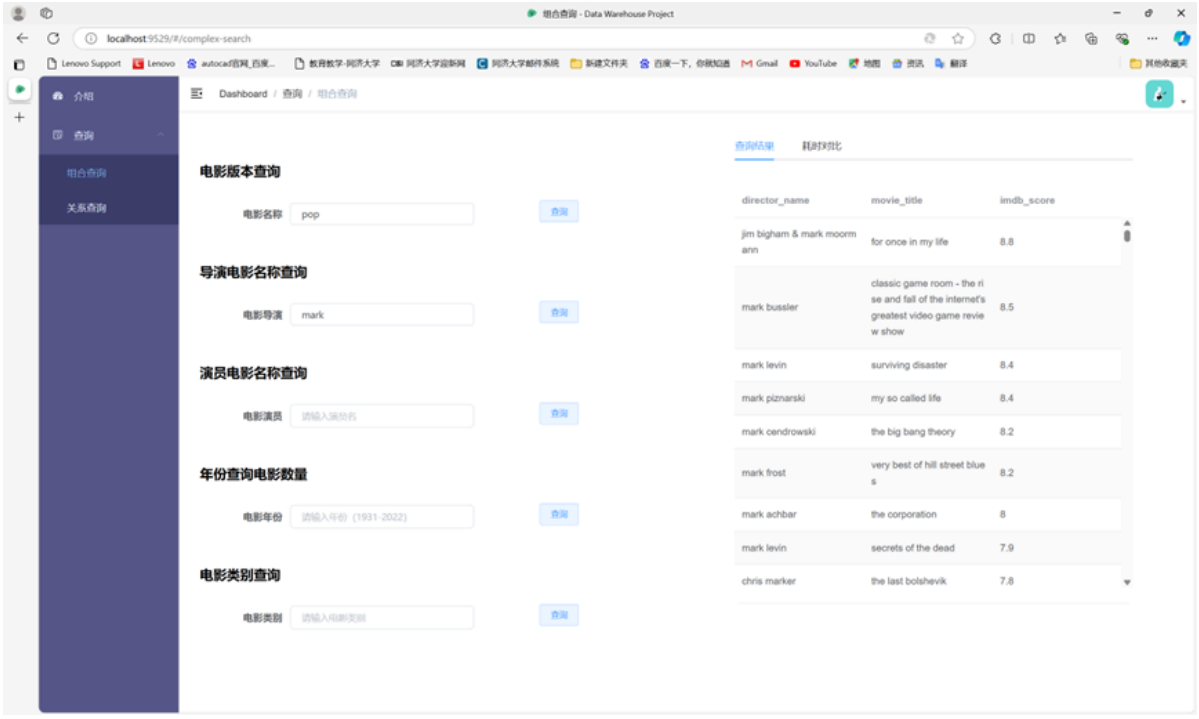
也可以选择耗时对比查看 MySQL 数据库与分布式数据库速度对比。



5.1.2 按照电影导演或演员姓名查询参演电影

输入导演或演员的姓名，查询该导演执导的电影\该演员参演的电影。返回电影的id、电影标题，电影评分和浏览次数，在右侧查询结果列表中以表格形式显示，也可以选择耗时对比查看 MySQL 数据库与分布式数据库速度对比。

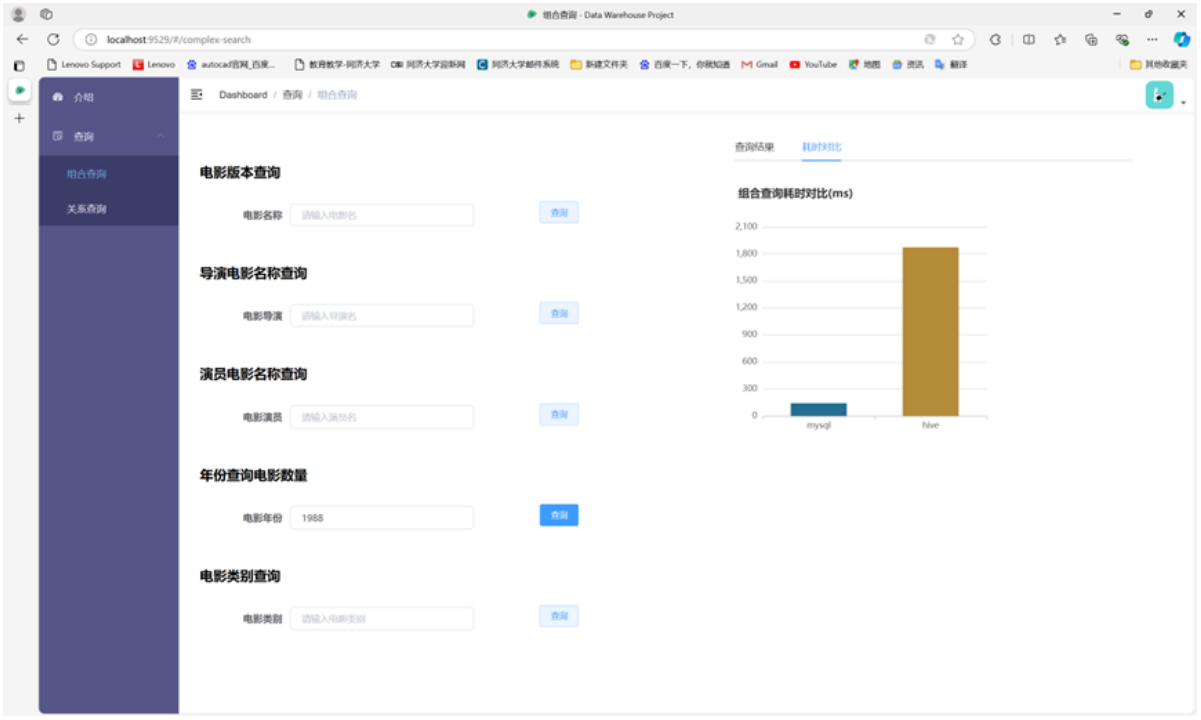
如输入mark：



5.1.3 查询某个年份的电影数量

输入年份，查询该年份发行的电影数量，在右侧查询结果上显示，也可以选择耗时对比查看 MySQL 数据库与分布式数据库速度对比。

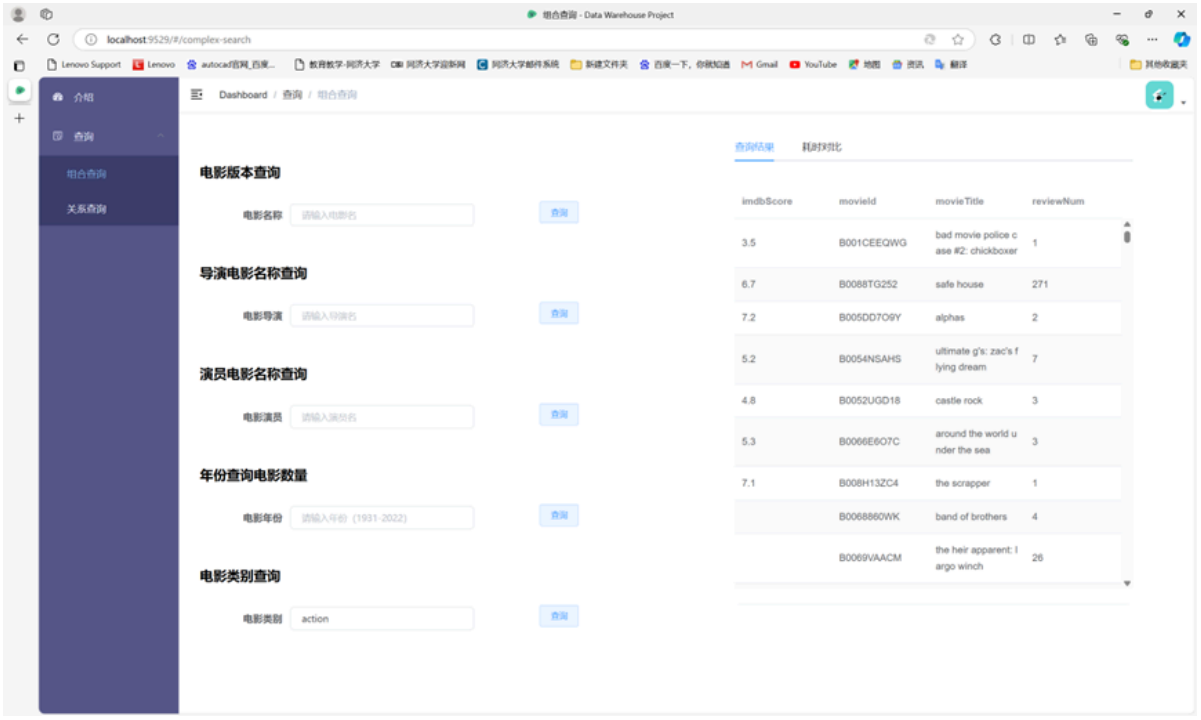
如输入1988：



5.1.4 查询某个类别的电影

输入电影类别，查询该类别的电影。返回电影id和标题，电影评分和浏览数量。在右侧查询结果上显示，也可以选择耗时对比查看 MySQL 数据库与分布式数据库速度对比。

如输入action

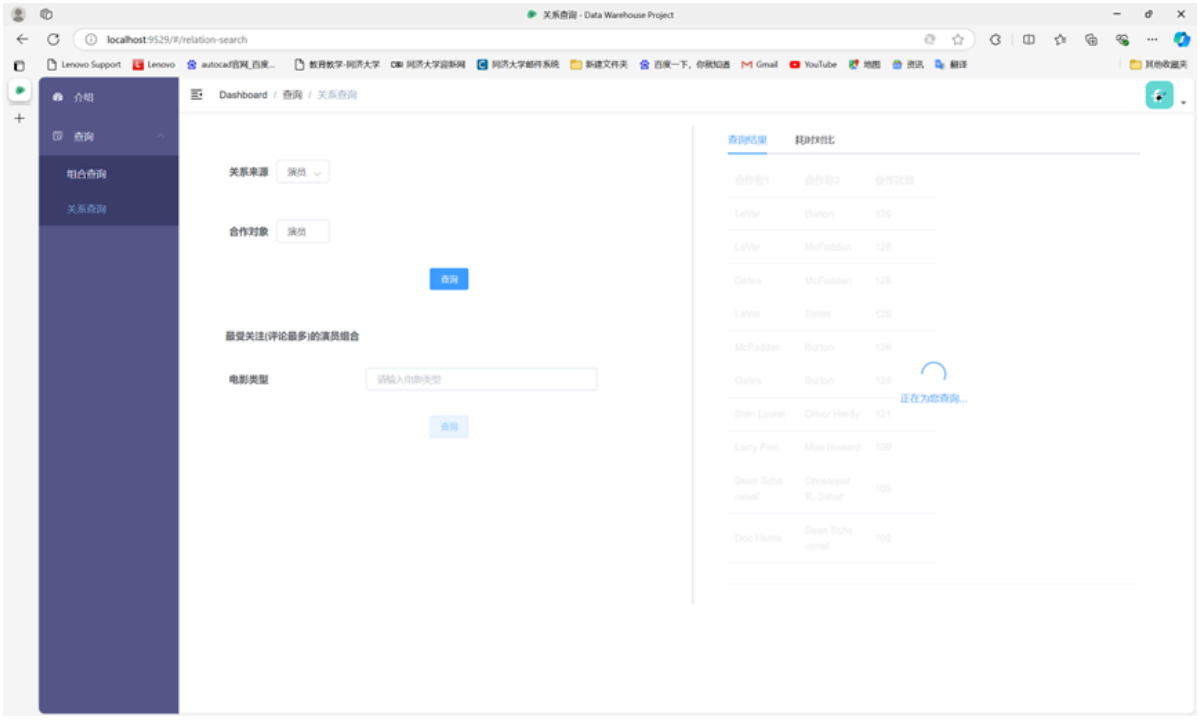


5.2 关系查询

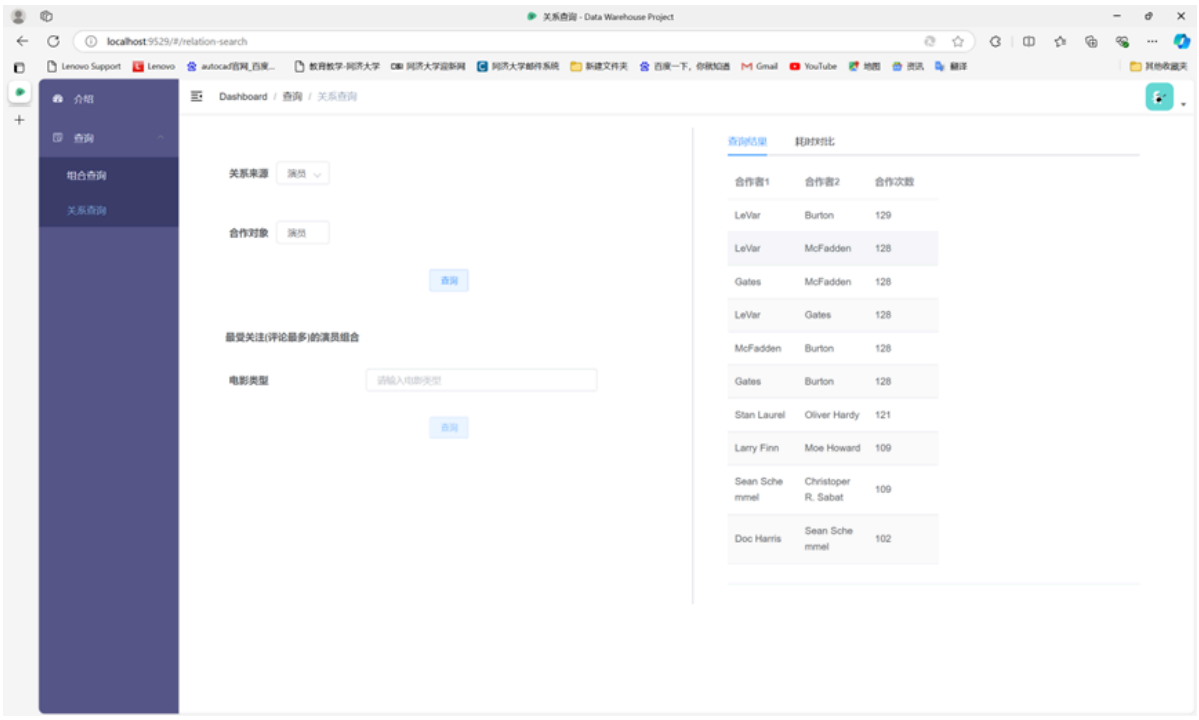
5.2.1查询演员导演合作关系

在左侧可以选择关系来源和合作对象。关系来源可选演员或导演，合作对象只能选择演员。选择结束后点击查询。

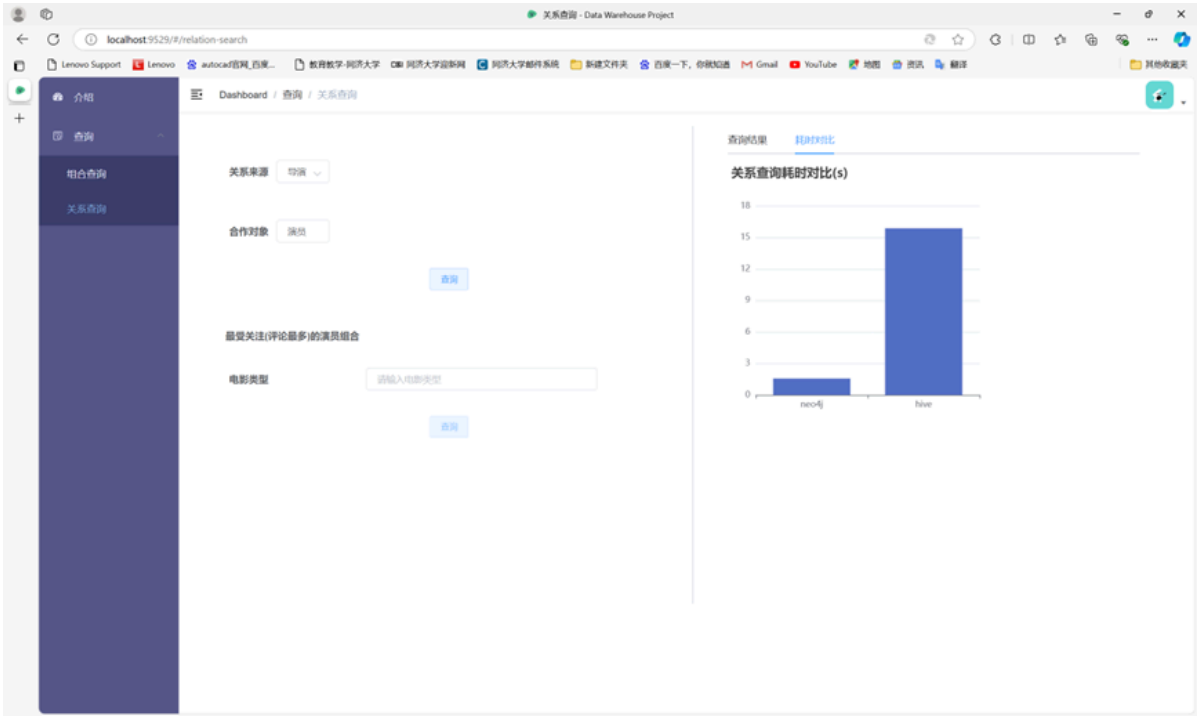
由于查询很费时，设置查询中页面：



查询两个演员合作的次数，右边会显示合作者1和合作者2以及合作次数记录。也可以选择耗时对比查看图数据库与分布式数据库速度对比。



选择导演和演员，右侧显示导演、演员、合作次数。也可以选择耗时对比查看图数据库与分布式数据库速度对比。



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，分布式数据库需要较多的join操作耗时较长。

5.2.2 最受关注演员组合

用户可以选择输入电影类型，查询该类型的电影中最受关注（评论最多）的演员组合，右侧显示合作者1、合作者2、合作次数。也可以选择耗时对比查看图数据库与分布式数据库速度对比。