数据存储设计说明文件

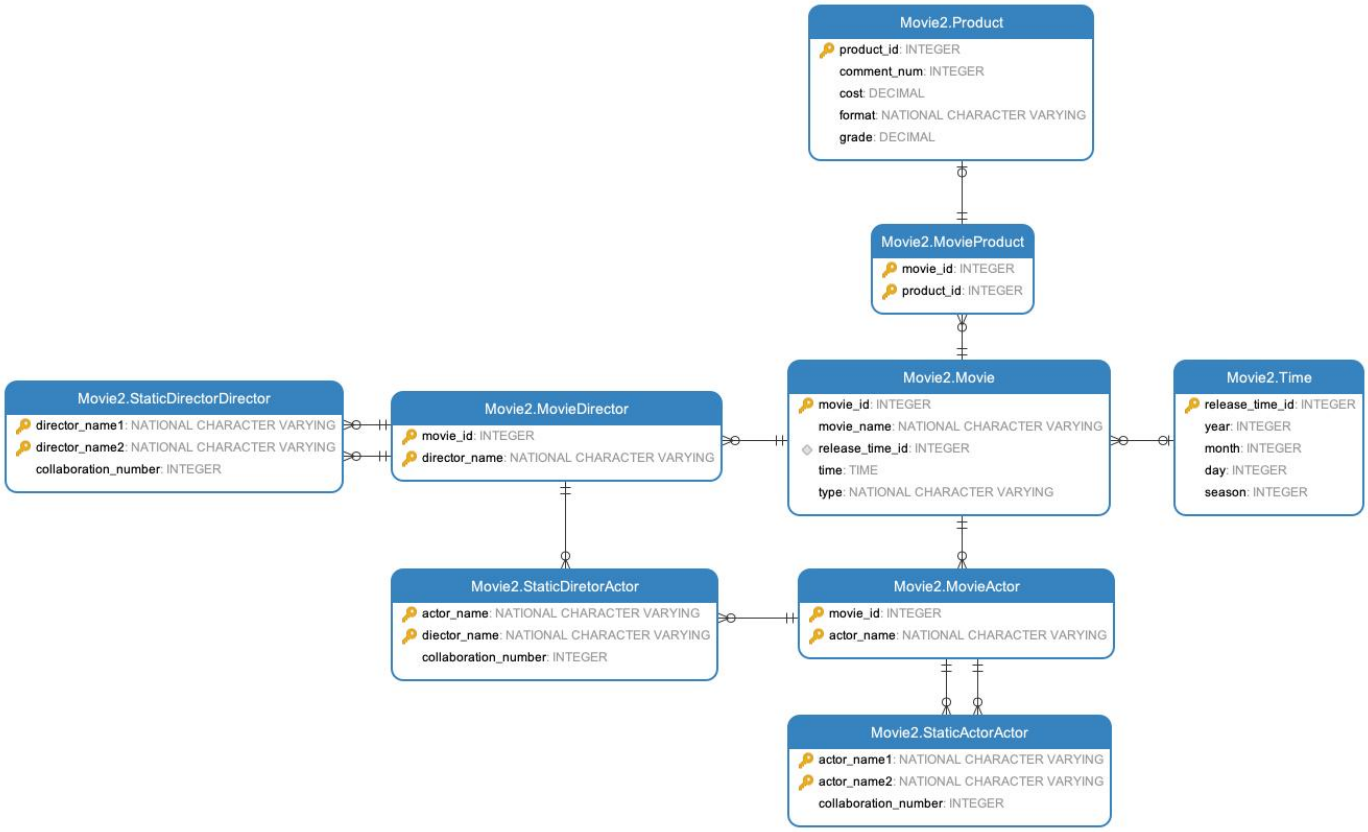
整体存储模式

在我们的项目中，对于所有的查询我们都分三次使用了关系型存储、分布式存储、图数据存储来实现，以对比三种数据存储形式的性能。但对比下来我们发现，图数据库在处理人员关系以及要连接多个表的查询方面的性能显著优于另外两种数据库。所以在真实场景中，推荐使用图数据库进行人员关系的查询。

- **关系型存储（MySQL）**：用于存储结构化的电影数据，如电影详细信息、用户评论、评分等。这些数据由于其结构化特性，适合在关系型数据库中管理。
- **分布式文件系统（Hive）**：用于存储和处理大规模数据集，主要处理批量的、不需要即时响应的数据分析任务，如历史数据分析、大规模数据集的统计和汇总。
- **图数据库（Neo4j）**：专注于存储和查询复杂的关系数据，如电影产品之间的关联、演员与导演之间的合作网络。

关系型存储逻辑模型

我们的星型图如下：



数据存储模型采用星型模式，以电影为中心，周围是演员、导演、产品和时间等。

关系型存储物理模型

- 最终数据库中表的分布如下：
  1. **电影（Movie）**：主实体，包含电影ID、电影名称、上映时间ID、时间、类型。
  2. **演员（MovieActor）**：包含电影ID和演员名称，表示电影与演员之间的关系。
  3. **导演（MovieDirector）**：包含电影ID和导演名称，表示电影与导演之间的关系。
  4. **产品（Product）**：包含产品ID、评论数量、成本、格式、评分等。
  5. **电影产品（MovieProduct）**：链接电影和产品，包含电影ID和产品ID。
  6. **时间（Time）**：包含发布时间ID、年、月、日、周、季节、电影数量等。
  7. **演员合作统计（StaticActorActor）**：包含两个演员的名称和他们的合作次数。
  8. **导演和演员合作统计（StaticDirectorActor）**：包含导演和演员名称以及他们的合作次数。
  9. **导演合作统计（StaticDirectorDirector）**：包含两个导演的名称和他们的合作次数。
- **存储优化设计**：
  - **索引优化**：为电影名称、演员名称、导演名称这些频繁查询的字段创建索引。
  - **数据分区**：根据电影的上映时间（年、月、季节）对`Movie`和`Time`表进行分区，提高基于时间的查询效率。
- **Denormalization**：为了提高演员导演之间合作关系查询的性能，建立了三张表：演员合作统计（StaticActorActor）、导演和演员合作统计（StaticDirectorActor）、导演合作统计（StaticDirectorDirector）

通过以上修改，我们的数据存储设计更加贴合项目的实际需求，同时保证了数据查询的高效性和数据整合的准确性。

分布式文件系统存储模型及优化（Hive on Hadoop）

在构建我们的分布式文件系统存储模型时，我们选择了基于Hadoop集群的Hive，这提供了一个强大的平台来处理和分析大规模数据集。以下是我们的配置和优化策略的详细描述：

Hadoop集群配置

- **集群架构**：我们的Hadoop集群配置包括一个主节点（Master Node）和三个数据节点（Data Nodes）。
  - **主节点**：负责管理Hadoop集群的资源分配和调度，运行YARN（资源协调器）和Hive的Metastore服务。
  - **数据节点**：存储数据并执行数据处理任务，每个节点都运行Hadoop的数据节点服务和YARN节点管理器。

Hive Schema定义

- 我们在Hive中创建了与MySQL相似的schema，以保持数据模型的一致性，但进行了适当的调整以适应大数据环境。例如，我们优化了表结构以支持Hive的分布式查询优势。

Hive优化策略

- **列式存储格式**：为了提高查询效率，我们选择了Parquet作为我们的存储格式。Parquet是一种高效的列式存储格式，它能够显著减少I/O操作，加速查询速度，尤其是在分析大规模数据集时。
- **数据分区**：我们根据查询的常见模式对数据进行分区，例如，可以按电影的上映年份或类型进行分区。这样可以减少查询时需要扫描的数据量，从而加快查询速度。
- **数据桶化**：对于某些关键表，我们使用了数据桶化（bucketing）策略，以优化特定类型的查询，如基于演员或导演的查询。通过将数据均匀分布在不同的桶中，我们能够更高效地执行联结和聚合操作。

性能监控和调优

- 我们使用了Hadoop和Hive提供的监控工具来跟踪集群的性能。定期检查资源利用率、查询执行时间和数据吞吐量，确保集群运行在最优状态。
- 我们还实施了定期的维护和调优措施，如调整YARN资源分配、优化Hive的查询计划，以确保我们的数据处理流程既高效又稳定。

通过这些配置和优化措施，我们的Hive on Hadoop解决方案能够高效地处理和分析庞大的数据集，为我们的项目提供了强大的数据处理能力。图数据库存储模型及优化

图数据库存储模型及优化

在Neo4j中，我们针对上述实体和关系构建了图模型如下：

实体（节点）的设计

我们的图数据库模型包含以下实体（节点）：

1. **电影（Movie）**：代表电影本身，节点属性包括 `movie_id`，`movie_name`，`release_year`，`time`，`type`。这些属性提供了电影的基本信息。
2. **产品（Product）**：代表与电影相关的不同产品版本，属性包括 `product_id`，`movie_id`，`Comments`（评论数量），`Cost`，`format`（版本），`Grade`。这表示电影的多样化产品线。
3. **导演（Director）**：代表电影的导演，属性仅包括 `director_name`。
4. **演员（Actor）**：代表参与电影的演员，属性仅包括 `actor_name`。

关系（边）的设计

我们的图数据库模型包含以下类型的关系（边）：

1. **导演关系（DIRECTED）**：表示导演执导某部电影的关系，从导演节点指向电影节点。
2. **演员关系（ACTED_IN）**：表示演员参演某部电影的关系，从演员节点指向电影节点。
3. **电影包含产品关系（INCLUDE）**：表示电影包含了特定产品的关系，从电影节点指向产品节点。
4. **合作关系（COOPERATE）**：表示演员与演员、导演与演员、导演与导演之间的合作关系，这是一种多用途的关系类型，用于连接合作的两个实体。

并实施了以下优化策略：

- **索引创建**：为了加快查询速度，我们在`movie_name`，`actor_name`，`director_name`这些关键属性上建立了索引。
- **关系优化**：我们确保了关系（边）的正确性和高效性。在关系中增加了属性：合作次数，这样可以在查询时直接访问这个属性，不需要count。
- **数据模型优化**：我们反复审视和优化数据模型，确保它既符合业务需求又保持了高效的数据处理能力。精简不必要的属性或调整节点和关系的结构。

通过这些设计和优化，我们的图数据库模型不仅能够准确地反映电影行业的复杂关系，还能高效地支持复杂的查询，如寻找合作网络，查找电影包含产品。

数据表的Test Case

我们设计了针对每种存储系统的测试用例，以确保数据的完整性和查询的准确性。例如，在MySQL中，我们测试了不同类型的查询，如基于电影名称的搜索、基于评分的过滤等；在Hive中，我们测试了大规模数据集的汇总和分析；在Neo4j中，我们测试了复杂的图查询，如寻找频繁合作的演员和导演。

通过这种综合的存储设计，我们确保了数据的完整性、可访问性和高效的查询性能。每种存储方式针对其特定的用例和数据特征进行了优化，从而使整个数据仓库系统在处理各种数据和查询时都能达到最佳性能。

示例1——数据质量控制器：数据治理体系，数据质量

概述

- **目的**：验证 `DataQualityController` 的 `actorByDirector` 方法能够正确地统计和返回数据预处理阶段的空值信息。返回空值字段的数量及其在整体数据中的占比
- **测试范围**：该测试用例将覆盖数据获取、数据处理和结果返回的完整流程。

测试环境

- **软件要求**：包含已部署的后端服务，且所有相关依赖（如 `DataQualityService`）均已正确配置。
- **硬件要求**：标准服务器或开发机器，无特殊硬件要求。

测试前提条件

- 确保 `DataQualityService` 已经获得了预处理数据，并且可以返回统计信息。
- 确保网络连接正常，以便进行API调用。

测试步骤

1. **启动后端服务**：确保包含 `DataQualityController` 的后端服务正在运行。
2. **调用API**：通过HTTP GET请求调用 `/quality` 端点。
3. **检查返回值**：验证返回值是否为 `ResultResponse.success` 类型，并包含预期的数据统计信息。

预期结果

- 返回的 `Result<Object>` 对象应包含一个 `success` 状态，表示请求成功处理。
- 返回的数据应包含预处理阶段的空值统计信息，具体格式根据 `DataQualityService` 的实现而定。

测试用例

1. **正常情况测试**
   - **目的**：验证在正常条件下，API能正确返回数据统计信息。
   - **步骤**：按照上述测试步骤进行。
   - **预期结果**：返回的数据中包含正确的空值统计信息。
2. **异常情况测试**
   - **目的**：验证在 `DataQualityService` 无法提供数据的情况下，API的行为。
   - **步骤**：在 `DataQualityService` 返回空或错误数据的情况下调用API。
   - **预期结果**：API应返回错误信息或空数据的相应处理结果。
3. **性能测试**
   - **目的**：评估在高并发请求下，API的响应时间和稳定性。
   - **步骤**：使用压力测试工具，模拟高并发访问 `/quality` 端点。
   - **预期结果**：API应在可接受的时间内返回结果，并保持稳定运行。

示例代码

```
package com.ssw331.warehousebackend.controller;

import com.ssw331.warehousebackend.Neo4jDTO.serialization.Result;
import com.ssw331.warehousebackend.Neo4jDTO.serialization.ResultResponse;
import com.ssw331.warehousebackend.service.DataQualityService;
import com.ssw331.warehousebackend.service.DirectorActorService;
```

```java
import io.swagger.v3.oas.annotations.Operation;
import io.swagger.v3.oas.annotations.tags.Tag;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

import java.util.ArrayList;
import java.util.List;
import java.util.Map;

@Tag(name = "数据质量")
@RestController
@RequestMapping("/quality")
public class DataQualityController {
    @Autowired
    DataQualityService dataQualityService;
    @Autowired
    private void setDataQualityService(DataQualityService dataQualityService) {
        this.dataQualityService=dataQualityService;
    }

    @Operation(summary = "数据预处理阶段空值统计")
    @RequestMapping(value = "", method = RequestMethod.GET)
    public Result<Object> actorByDirector() {
        List<Long> modelTimes = new ArrayList<>();
        List<String> modelLogs = new ArrayList<>();
        Map<String,Object> data=dataQualityService.getDataStatistics();
        modelTimes.add(0L);
        modelLogs.add("");
        return ResultResponse.success(data, modelTimes, modelLogs);
    }
}
```

示例2——条件查询

测试代码

1. 关系型数据库

```sql
SELECT
    movie.*
FROM
    Movie movie
WHERE
    movie.Type LIKE '%Action%'
```

2. 分布式数据库

```
SELECT
    *
FROM
    movie
WHERE
    Type LIKE '%Action%'
```

3. 图数据库

```
MATCH (movie:Movie)
WHERE movie.Type CONTAINS 'Action'
RETURN movie
```