I 问题说明

1. 对每种存储方式结合本项目说明各自适用于处理什么查询，针对本项目在存储优化中做了什么工作，优化前后的比较结果是怎样的？
2. 如何保证数据质量？哪些情况会影响数据质量？
3. 数据血缘的使用场景有哪些？

II 存储分析

2.1 MySQL

2.1.1 优化措施

1、反范式

如果根据范式对数据表进行切分，一部分查询必然会涉及到多表 join，很可能带来严重的性能影响，因此采取星型模型为逻辑模型优化数据库设计。

2、数据字段

通过对数据字段的类型和内容进行优化，减少数据表中单行记录的存储大小，从而提高单表查询的速度。

(1) 尽量减少单表的字段数，去除不必要的字段。

(2) 避免使用NULL字段，避免NULL字段占用额外索引空间和引起引擎放弃索引采用全表扫描

3、索引

根据数据库索引的原理与结构，合理地为数据表建立索引，提高查询效率。

例如进行根据演员查询电影，查询主要依据是演员名，因此对演员冗余表中的演员名称字段建立了索引。

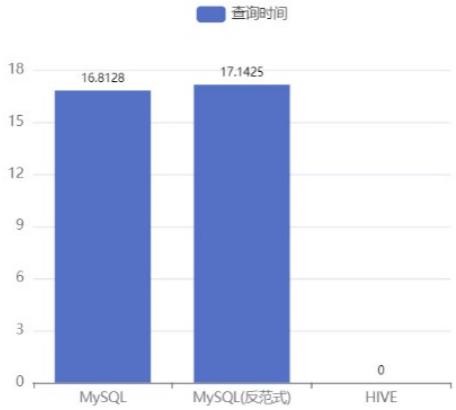
2.1.2 性能对比

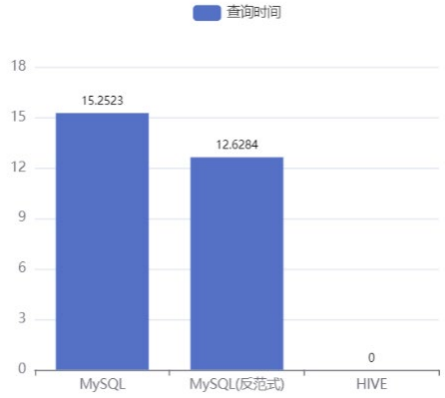
1、单个查询

	遵循范式	反范式
标签查询 电影	60ms	59ms
电影查询 评论	54ms	18ms

在查询单个业务时，我们发现遵循范式的数据库和反范式的数据库性能相差不大。

2、组合查询





在组合条件查询中，左图是条件为一时的对比，右图是条件为三时的对比，可以发现在条件数增多时，反范式数据库相对遵循范式数据库性能好很多。

3、合作查询

	遵循范式	反范式
有起点	20ms	13ms
无起点	8536ms	2402ms

在查询实体关系的业务中，由于反范式需要做的join相对于符合3NF要少，可以发现反范式的数据库速度要快很多。

此外，当不同的查询语句对于无起点的实体关系查询性能的影响也非常显著。我们发现，如果先对应名字，再判断是否合作，通过电影id连接再通过双方名字分组，对比先判断合作，通过电影id连接再通过双方id分组，再对应名字要快很多，我们认为主要是varchar相对于int在查询性能中表现较差。

	先对应名字	先判断 合作
无起点	8536ms	858ms

2.2 Hive

2.2.1 优化措施

Apache Hive支持Apache Hadoop中使用的几种常见的文件格式，如TextFile，RCFile，SequenceFile，AVRO，ORC和Parquet格式。不同的存储格式在内存中有不同的存储方式和性能特点。向 Hive导入数据是默认采用TextFile格式，该格式存

储空间消耗较大，并且压缩的text无法分割和合并。因为可以直接存储，所以加载数据的速度最高，但查询的效率最低。这显然不符合我们数据仓库对查询性能要求高的特点。我们把存储格式更换为ORCFile，以寻求更高的查询性能。ORC文件是一种优化的行列存储相结合的存储方式，数据按行分块，每块按照列存储，其中每个块都有一个索引，数据压缩率非常高，同时能够获得很高的查询性能。

2.2.2 性能对比

查询条件	存储格式	查询时间	结果数目
根据年月查询电影	TextFile	21.755s	128
根据年月查询电影	ORCFile	1.589s	128
根据演员查询电影	TextFile	167.104s	10
根据演员查询电影	ORCFile	137.104s	10

通过以上结果可见，Hive 进行相同的条件组合查询的性能明显低于 Mysql 关系型数据库。

首先，通过查看执行计划，可以看到在 Hive 中进行查询时，但凡涉及到 group by、聚合函数、多表 join 等操作，都会调用 Map Reduce 分布式计算框架进行计算。其次，进一步分析其原因，Hive 本身的特点是在大型数据集上会表现出优越的性能，考虑到我们的项目数据集中，最多的数据集是 700 多万条的用户评论数据，而基本功能的实现都是操作在数据量仅有 25 万余条的电影数据，我们猜测是数据量限制了 Hive 体现其性能优越性。

第三，可以对 Hive 的存储格式进一步探究，发现在 TextFile 和 ORCFile 存储格式下 Hive 的查询性能差异极大，ORCFile 不但数据压缩效果较好，文件存储大小明显更小，而且查询性能更优，甚至查询耗时已经不在一个数量级。

2.3 Neo4j

2.3.1 优化措施

基于Neo4j图数据库的查询性能的优化，我们采用了添加模式索引的方式，在Screenwriter，Actor，Director结点上都加入了索引。

Index Name	Type	Uniqueness	EntityType	LabelsOrTypes	Properties	State
index_2586c141	BTREE	NONUNIQUE	NODE	["Screenwriter"]	["name"]	ONLINE
index_343aff4e	LOOKUP	NONUNIQUE	NODE	[]	[]	ONLINE
index_ad47116a	BTREE	NONUNIQUE	NODE	["Actor"]	["name"]	ONLINE
index_c2eaf96f	BTREE	NONUNIQUE	NODE	["Director"]	["name"]	ONLINE
index_f7700477	LOOKUP	NONUNIQUE	RELATIONSHIP	[]	[]	ONLINE

除此之外，在实际查询中，在正式查询所需数据前，可以先查询所有数据，进行预热，这样可以触发图中所有的点和关系。整个图会被缓存起来。后续查询的速度将会非常快。

2.3.2 性能对比

1、优化前后对比

预热对查询性能的影响

是否预热	是	否
查询时间	877ms	1349ms

索引对查询性能的影响（预热后）

	有索引	无索引
有起点	6ms	52ms
无起点	741ms	877ms

针对以上数据可以看出，两种优化措施的性能优化体现在以下方面：

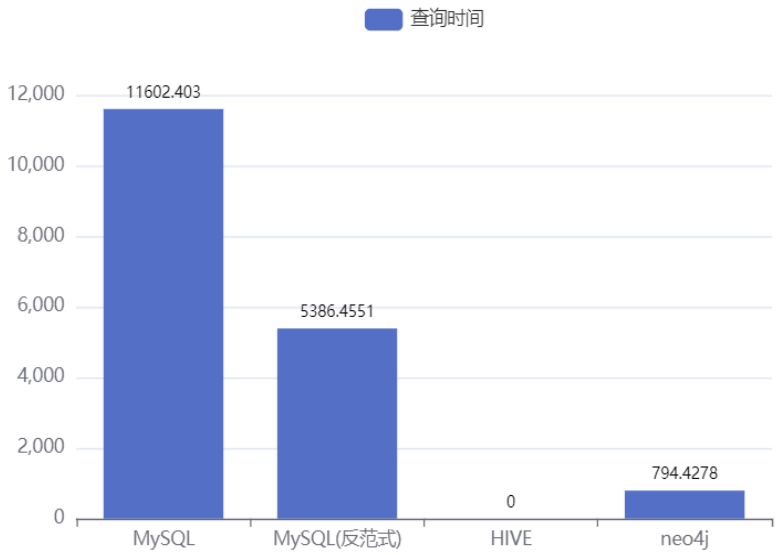
对于添加索引的性能优化措施，其对是否有起点的查询优化效果较为明显，有起点的查询在增加索引后查询速度提升非常快，对于无起点的查询优化效果不大。

对于预热的性能优化措施，其针对两种查询角度，性能优化都比较明显。

因此，进行有起点查询时，使用索引是比较好的方法，其次，对于任何查询，都可以在正式查询前进行预热，提高性能效果较好。

2、MySQL与Neo4j合作关系查询性能对比

(1) 无起点查询演员与导演合作关系：



(2) 有起点查询演员与编剧合作关系：

	MySQL	MySQL（反范式）	Neo4j
第一次	20.6412ms	8.6129ms	676.7394ms
第二次	10067.7199ms	10047.424ms	2720.6706ms
第三次	37.8242ms	38.3078ms	75.4129ms

针对以上查询，可以看出无起点查询时，Neo4j的性能明显好于关系型数据库，因为图数据库找到节点后不需做任何join即可以直接找到与该节点有关系的其他节点，而不论是否反范式的MySQL存储模型或多或少都需要做join操作，浪费了时间。因此，无起点查询时，Neo4j性能更好。

而有起点查询时，我们发现了一个奇怪的现象，第一次是将后端与MySQL数据库部署到一台机器上运行，查询前已进行过预热，第二次是将后端与Neo4j数据库部署到一台机器上运行，并且查询前未进行过预热，第三次是在第二次的基础上进行预热过后的结果，可以看出，后端访问同一台机器上的数据库时间是明显要比访问另一台机器上的数据库的时间短的，第一次Neo4j时间较长的原因可能是后端访问不在同一台机器上的数据库连接时间较长，第二次三种时间都很大的原因可能是首次连接数据库花费了较多时间，第三次可能是MySQL底层优化使得对于查询做了缓存，为了验证这一猜想，我们使用同一条件组合查询多次，发现MySQL数据库越到后面的查询就会越快，于是猜测或许是MySQL的底层优化对查询做了缓存。

从数据库的角度来说，有起点查询时间Neo4j和MySQL的查询速度是不相上下的，都是个位数毫秒级别。因此有起点查询两个数据库性能都比较好。

2.4 总结

MySQL适合查询实体属性，反范式的关系型数据库在组合查询条件多时性能较好，Neo4j适合查询实体关系，Hive适合查询数据量庞大近乎百万级时的实体属性。

III 数据质量

3.1 数据爬取

3.1.1 提取所有asin

从评论文本中提取asin时，为了确保提取的正确性，首先把所有的asin提取到一个链表里，通过比较评论内JSON和链表的长度，如果相同，则说明asin提取正常，比较后再通过集合去重。

3.1.2 爬取网页

在网页爬取中，为了保证数据质量，主要采取了以下措施：

为了区分是否是正常网页，我们在爬虫过程去检查了页面某个组件的ID，若网页是反爬的验证码或连接不正常导致的错误时，该ID为'Sorry! Something went wrong!'或'Robot Check'。若ID不正常，则视作该网页未爬到，会在后续执行中继续爬取。

为了防止网页未渲染完成就爬取的情况，我们检查了“Product Details”组件是否存在（因为主要提取该部分的数据），若该组件存在则视作网页渲染完毕。否则后续继续爬取。

在爬取完成后，为了检查数据质量，我们抽样了100份数据，打开爬取得到的html文件，发现都是正常数据。

3.2 数据清洗

3.2.1 数据解析

为了保证数据解析时的数据正确性，我们在数据解析完成后进行抽样检查，抽样了100份结果，对比其原始数据，未发现关于逻辑的异常，只有提取内容伴随了“\\xa0”等文本不规范情况，将在下一部分处理。

3.2.2 信息规范化

1、删除干扰信息

由于标题信息的不标准携带"DVD"、“VHS”等旁支信息，以及网页格式影响提

取内容会伴随“\\xa0”等符号，我们使用正则表达式对解析后的数据进行了规范化处理，比如删除标题中的伴随信息，将电影时长统一为分钟为单位。此外，对于一些不合法的值，比如演员、导演等名字中的“Various”，“-”，“--”，进行了删除处理。

2、缺失值处理

因为数据中可能存在一些缺失值，所以我们需要对缺失值进行合理补充来让数据更适应于业务。比如针对电影发行时间的缺失，我们首先使用IMDB电影数据库的日期来填充，填充后仍存在缺失的数据，则采取使用最早的评论时间来进行第二层填充。

3、正确性保证

我们采取了抽样检查的方式进行正确性检验，最终处理结果数量一致且处理符合预先逻辑。

3.2.3 过滤非电影信息

1、Amazon label & IMDB

在过滤非电影信息时，我们考虑到亚马逊网站标签的不规范性，所以对于亚马逊网站标签中明确存在“Movie”或“Film”，且不是“Film History”，我们认定为电影，其余的部分，我们查询电影名是否存在于IMDB的电影数据来判断。而考虑到同名不同一的情况，我们进一步考虑导演信息，作为鉴别依据。

2、正确性保证

我们采取了抽样检查的方式进行正确性检验，发现结果数据满足Amazon label为电影或确实存在于IMDB。

3.2.4 过滤评论信息

在过滤评论信息完成后，我们使用2.1.1的程序脚本提取了过滤得到评论的asin，与电影的asin数量对比，若相等，则说明评论过滤正确。

3.2.5 导入数据库

1、MySQL && Hive

针对以上两个数据库，将数据导入时使用了同一个数据源文件，通过导入向导并且导入过程未发现外键异常或其他异常，因此这两个数据库的数据一定是相同的。

2、Neo4j

对于图数据库，其数据源是导入MySQL和Hive数据的总和（即演员，电影，导演，编剧信息在一张表上），使用python脚本导入，导入过程未发现异常，在导入完成后，通过对比Neo4j与MySQL的导演，演员，电影，编剧数量，发现全部

相等，则数据导入是正确的。

3.3 总结

在保障数据质量方面，我们通过第三方权威数据库或合理假设，对于异常数据都进行了规范与完善，同时在预处理的每一步，我们都进行了正确性检验，以保证处理的正确性。

在电影数据处理过程中，我们认为影响数据质量的因素主要有：数据的完整性、唯一性、准确性。Amazon的很多影视数据都存在缺失导演、演员等信息的情况，此外电影名称的同名引起了唯一性的，同时在处理时的人为疏忽也会导致数据质量的下降。

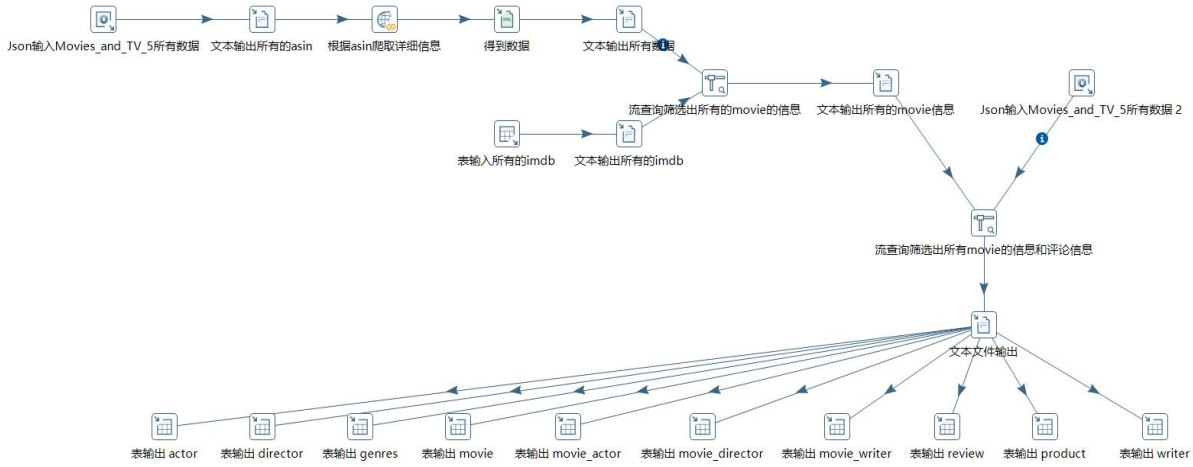
溯源查询：

在预处理过程中，我们在60175条影视数据中筛选出18935条电影数据。

在ETL和数据预处理中，我们找到12部哈利波特系列的电影，总共有12个版本，12个网页。由于大部分哈利波特系列的电影都是作为合集存在，因此，哈利波特第一部我们没有合并网页。

IV 数据血缘

4.1 数据血缘简图



4.2 使用场景

4.2.1 数据解析

在数据的处理过程中，从数据源头到最终的数据生成，每个环节都可能会导致我们出现数据质量的问题。比如我们数据源本身数据质量不高，在后续的处理环节中如果没有进行数据质量的检测和处理，那么这个数据信息最终流转到我们的

目标表，它的数据质量也是不高的。也有可能在某个环节的数据处理中，我们对数据进行了一些不恰当的处理，导致后续环节的数据质量变得糟糕。因此，对于数据的血缘关系，我们要确保每个环节都要注意数据质量的检测和处理，那么我们后续数据才会有优良的基因，即有很高的数据质量。

就本次项目而言，我们在最后建表时发现，有一些电影是没有相应的review信息的，而这个数据异常我们就可以根据我们的数据血缘图进行定位，我们建表的数据来源是流查询筛选出的所有movie的信息和评论信息，该数据是从json输入的所有Movies_and_TV_5和文本输出的所有movie信息来的,经过确定之后，发现异常来源是文本输出的所有asin信息，而文本输出的所有的movie信息来源于流查询的筛选，进一步确定，我们发现是在进行流查询的文本输出的所有imdb出现问题，有一些数据的丢失，在再一次筛选之后得到正确数据，将丢失的四千多条评论信息加入。

对于数据血缘的利用还有很多，如通过收集调度任务的开始结束时间，了解关键任务ETL链路的时间瓶颈，再根据JOB任务的执行情况，定位到性能瓶颈，以及通过对表和字段的下游使用频次进行一些仓库优化等等，不过在本次项目中，数据血缘更多使用在确定数据异常上。