基于星型模型的混合模型选型原因和比较方案及结论

数据仓库文档

AMAZON MOVIE ANALYZE AND DISPLAY

WAREHOUSE, AUTUMN 2017

BY

1552635 胡嘉鑫
1552672 吴可菲
1552673 陈明曦
1552674 李　源

Tongji University
School of Software Engineering

Contents

1 混合模型选择与建立 **3**
1.1 基本E-R图建立 . . . 3
1.2 星型模型与雪花模型比较 . . . 4
1.2.1 两种模型简介 . . . 4
1.2.2 使用选择 . . . 4
1.2.3 结论与最终的模型 . . . 4
1.3 对星型模型的进一步优化 . . . 5

2 混合模型性能分析 **6**
2.1 硬件环境简介 . . . 6
2.2 查询比较 . . . 7
2.2.1 简单查询 . . . 7
2.2.2 组合查询 . . . 7
2.3 结果分析 . . . 7
2.4 一点其他的比较 . . . 7

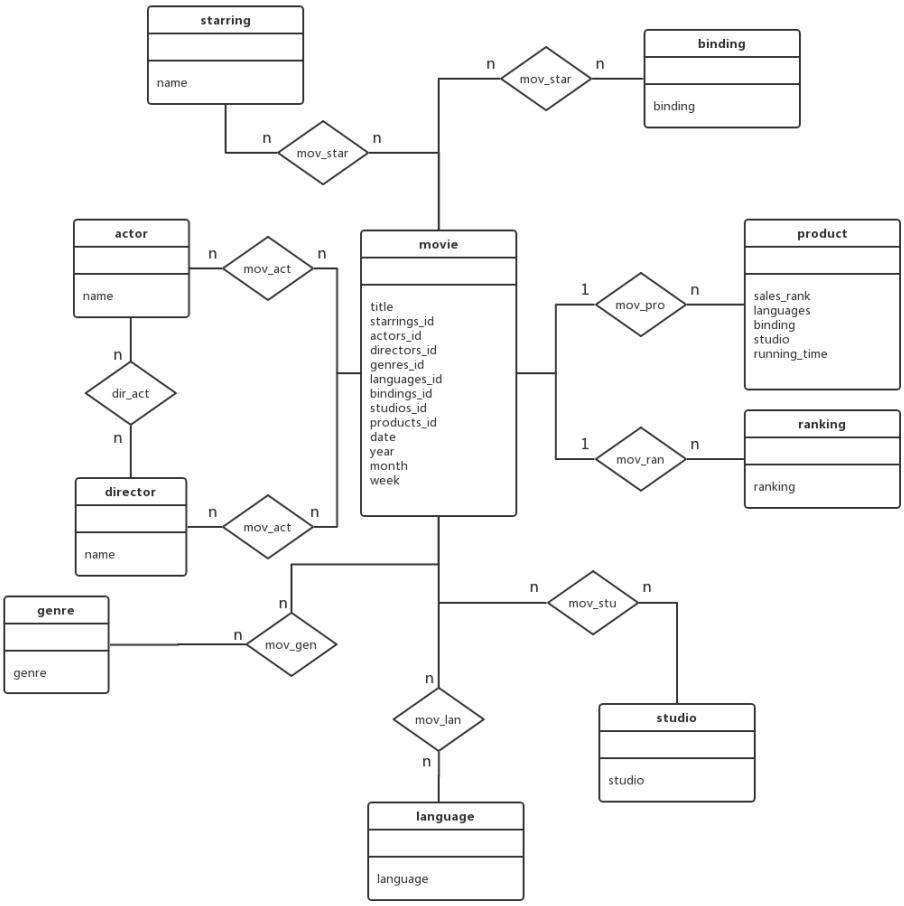
1 混合模型选择与建立

1.1 基本E-R图建立

本次项目我们所需要完成的查询操作一共包括如下各种:

1. 按照时间进行查询及统计（例如XX年有多少电影，XX年XX月有多少电影，XX年XX季度有多少电影，周二新增多少电影等）
2. 按照电影名称进行查询及统计（例如 XX电影共有多少版本等）
3. 按照导演进行查询及统计（例如 XX导演共有多少电影，这个导演喜欢和哪些演员合作，和什么样的演员合作等）
4. 按照演员进行查询及统计（例如 XX演员主演多少电影，XX演员参演多少电影，一起主演的电影，一起参演的电影等）
5. 按照电影类别进行查询及统计（例如 Action电影共有多少，Adventure电影共有多少等）
6. 按照上述条件的组合查询和统计

根据本次项目的数据来源以及所需要完成的查询操作，我们首先建立了最基本的E—R图，我们决定把每一部电影的数据作为事实表的一项，再把电影的派生信息包括上映时间、演员、导演等信息放到维度表中。其最初的E-R结构如下:



但是，如果按照如上的E-R图建立存储模型，显然在查询的时候会存在多次的连接表和事实表的join操作，这些操作会消耗大量的时间，也不符合数据仓库的设计理念。事实上，在多维分析的商业

智能解决方案中，根据事实表和维度表的关系，又可将常见的模型分为星型模型和雪花型模型。在我们的这次项目，我们首先将星型模型和雪花模型进行比较，从中选择一个更加合理的建模方案，并在此基础上，对我们的数据模型进行优化。

1.2 星型模型与雪花模型比较

1.2.1 两种模型简介

当所有维表都直接连接到“ 事实表”上时，整个图解就像星星一样，因此我们常将这样的模型称之为星形模型(Star Schema)。星型架构是一种非正规化的结构，多维数据集的每一个维度都直接与事实表相连接，不存在渐变维度，所以数据有一定的冗余。比如，我们假设现在有一个存储地域的纬度表，存在国家 A 省 B 的城市 C 以及国家 A 省 B 的城市 D 两条记录，那么国家 A 和省 B 的信息分别存储了两次，即存在冗余。

而雪花模型(Snowflake Schema)则是，当有一个或多个维表没有直接连接到事实表上，而是通过其他维表连接到事实表上时，我们可以将其图解比做多个雪花连接在一起。雪花模型是对星型模型的扩展。它对星型模型的维表进一步层次化，原有的各维表可能被扩展为小的事实表，形成一些局部的 ” 层次 ” 区域，这些被分解的表都连接到主维度表而不是事实表。它的优点是：通过最大限度地减少数据存储量以及联合较小的维表来改善查询性能。雪花型结构去除了数据冗余。

1.2.2 使用选择

对于数据仓库中常用到的星形模型(Star Schema)和雪花模型(Snowflake Schema)，我们通常可以从从三个角度来进行讨论、比较

- **数据优化**

雪花模型使用的是规范化数据，也就是说数据在数据库内部是组织好的，以便消除冗余，因此它能够有效地减少数据量。通过引用完整性，其业务层级和维度都将存储在数据模型之中。相比较而言，星形模型实用的是反规范化数据。在星形模型中，维度直接指的是事实表，业务层级不会通过维度之间的参照完整性来部署。星型模型这样的存储方式，能够有效的减少各个表之间的join操作，对于我们这个项目是适合的。

同时，在我们的数据上，是不需要对纬度进行进一步的区分的。那么如果我们将模型设计为雪花模型，是多余的。

- **性能**

雪花模型在维度表、事实表之间的连接很多，因此性能方面会比较低。举个例子，如果我想要知道一部电影的详细信息，雪花模型就会请求许多信息，比如演员、主演、导演等等需要连接起来，然后再与事实表连接。而星形模型的连接就少的多，在这个模型中，如果你需要上述信息，你只要将电影的维度表和事实表连接即可。

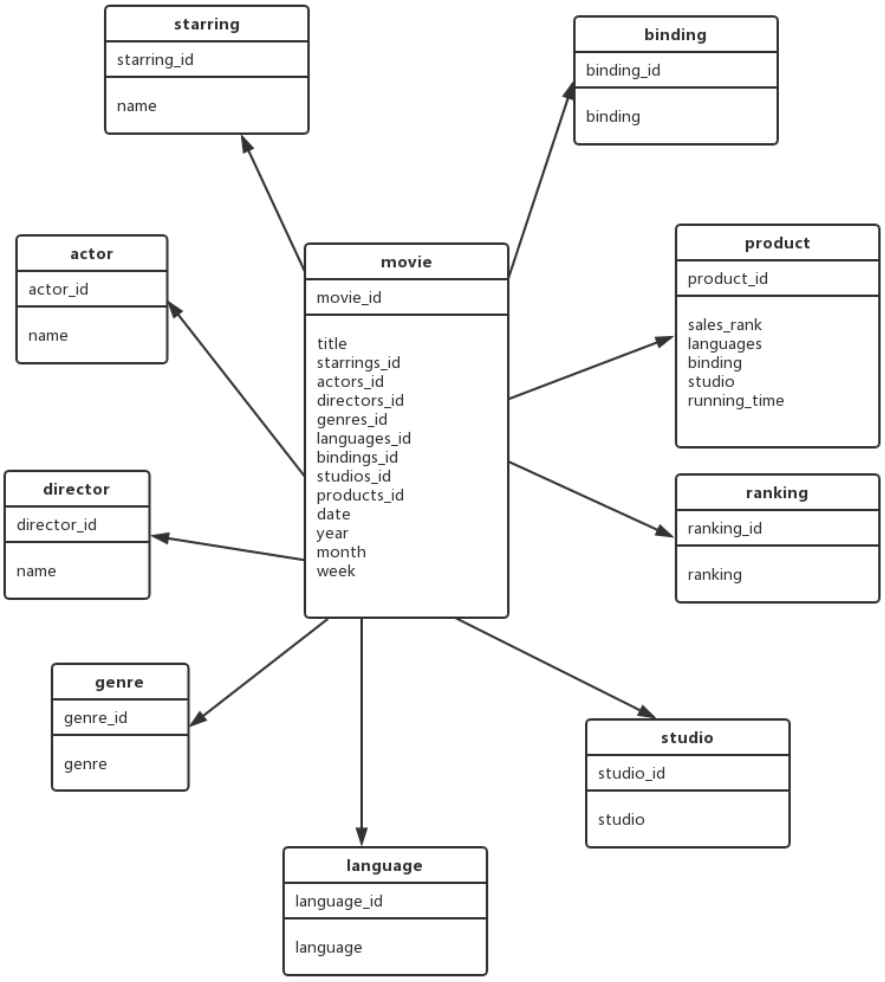
- **ETL**

雪花模型加载数据集市，因此ETL操作在设计上更加复杂，而且由于附属模型的限制，不能并行化。

星形模型加载维度表，不需要再维度之间添加附属模型，因此ETL就相对简单，而且可以实现高度的并行化。

1.2.3 结论与最终的模型

雪花模型使得维度分析更加容易，而星形模型用来做指标分析更适合。在我们的本次项目中，我们所需要经常统计的是与电影相关的各种指标信息，因此我们决定最终使用星型模型作为改进的模型。最终构建的模型如下：



1.3 对星型模型的进一步优化

对于星型模型而言，我们仍然存在一个问题需要解决：导演、主演演员以及参演演员等这些多对多的关系如何存储呢。如果把每一部电影的所有演员以一个JSON数组的形式存储在电影事实表的一项中，就没有办法根据演员来查询了。而如果像最初的E-R图一样，建立桥接表来桥接事实表和维度表，虽然可以满足多对多的关系，可是这样多表查询会对速度产生非常大的影响。

朱宏明老师上课的一句话点醒了我们。“可以为了提高速度而做适当的冗余“。我们可以把所有演员以字符串的形式存储在电影的属性中，同时把某一个演员的演的所有电影以字符串的形式存储在演员维度表的属性中。虽然有比较多的冗余，但数据仓库就是作为这样一个为了查询速度而可以冗余的存在。

其具体在数据库中的存储方法如下面截图:

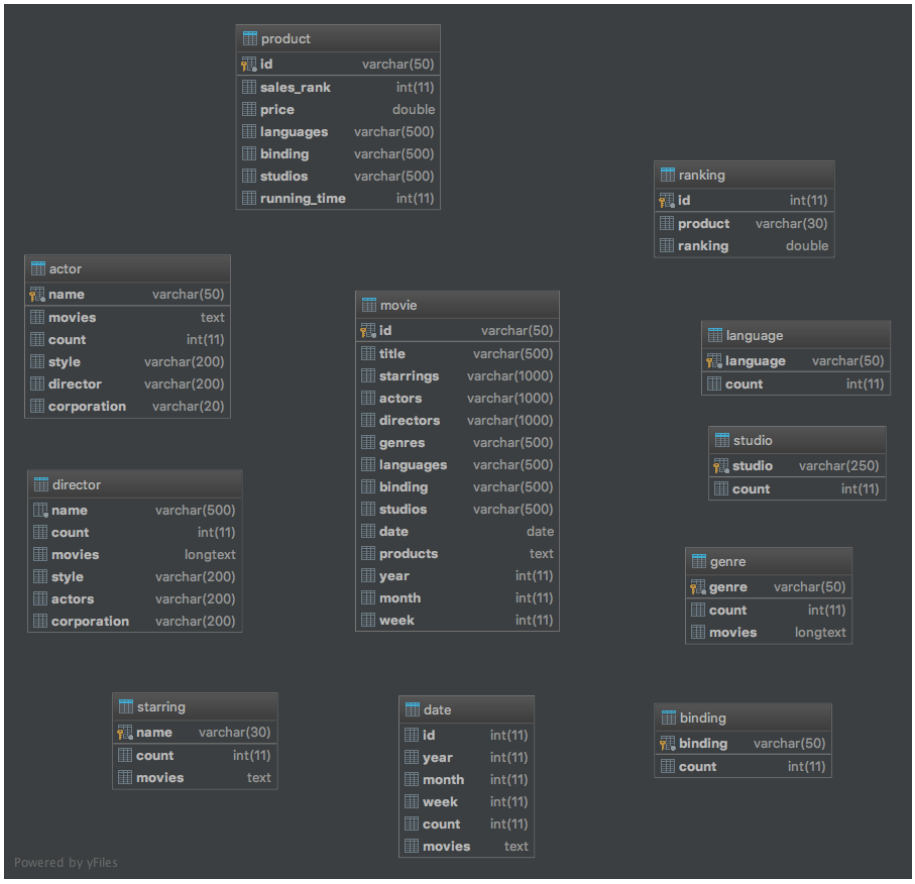
genre	count	movies
ACTION	2	138417,36116
ACTION/ADVENTURE	298	101203,101236,101362,101587,102533,10336,10337,105721,10753,10…

事实上，为了进一步加快查询，我们将许多聚合值也作为表的一个字段存储了起来。比如不同语言、风格等的电影数，某一年某一月周几的上映电影数等等。这样虽然加大了数据冗余，但是极大程度的优化了繁琐的查询操作，利用空间换取时间，我们认为这样的操作在数据仓库中是合理的。

这类数据在数据库中的存储方法如下面截图：

name	movies	count	style	director	corporation
Aaron Ginn-Forsberg	139101,153590	2	Action,None	Jon Bonnell	1
Aaron Goldenberg	158218	1	faith_&_spirituality	Norton Rodriguez	1

我们最终的数据库物理表结构如下：



2 混合模型性能分析

2.1 硬件环境简介

对于本项目，我们分别实现了关系型的MySQL数据库和分布式文件系统型的Hive数据库。其各自的硬件环境如下：

关系型数据库

- OS：Ubuntu 14.04.1 LTS (GNU/Linux 3.13.0-32-generic x86_64)
- Hardware：Core i5 & 2Gb
- Database：MySQL 5.6.25

分布式文件型数据库

- OS：Ubuntu 14.04.5 LTS (GNU/Linux 4.4.0-31-generic i686) x3
- Hardware：Core i5 & 4Gb x1 ; Core i7 & 4Gb x2
- Database：Hadoop 2.8.2 & Hive 2.1.1

2.2 查询比较

我们针对部分简单查询和组合查询，分别对关系型数据库和分布式文件型数据库执行查询十次命令，求平均值，作为时间开销。

2.2.1 简单查询

查询命令	MySQL时间	Hive时间
查询电影名	0.483秒	4.335秒
查询演员名	0.107秒	4.909秒
查询导演名	0.143秒	2.470秒
查询风格	19.56秒	大于1分钟

2.2.2 组合查询

查询命令	MySQL时间	Hive时间
综合时间查询	1.471秒	15.75秒
组合查询	0.181秒	5.215秒
评论打分查询	9.612秒	大于1分钟
风格趋势查询	1.818秒	19.340秒

2.3 结果分析

从上面的时间开销我们可以得到，MySQL的查询速度要明显地快于Hive。这是为什么呢?

事实上，对于亚马逊电影数据的分析，虽然更加侧重于实时多维分析技术，即OLAP，对数据进行多角度的模拟和归纳，从而得出数据中所包含的信息和知识。但是由于在我们的本次项目中，除了电影评论表的数据量达到了百万级（并且还是因为我们为了增加数据量而没有求取聚合值），其余的表的数据量通常在十万级，甚至更少。

而在Hive中，对于每一次查询操作会比单机的MySQL多一个MapReduce的操作，即两者的查询时间可以分别为如下两个函数:

$$\text{MySQL时间开销} = \text{查询时间} \tag{1}$$

$$\text{Hive时间开销} = \text{MapReduce时间} + \text{分步查询时间} \tag{2}$$

这里存在一个博弈，即MapReduce时间是Hive的额外开销，但是换来了在表中查询时间的缩短。而对于我们的较小数据量来说，MapReduce的额外开销会远大于其缩短的时间。因此最后的查询结果会存在单机MySQL查询时间反而比Hive更快，还快很多的结果。

在商业级应用当中，百万级的数据，无论侧重OLTP还是OLAP，我们通常会MySql这类的数据库，简洁、高效。而过亿级的数据，侧重OLTP可以继续Mysql，侧重OLAP，就要分场景考虑了。

对于我们本次的应用场景，是一个批处理计算场景：强调批处理，常用于数据挖掘、分析。如果当数据量达到千万级甚至亿万级时候，我们更应该选择Hive&Hadoop搭建数据仓库。而当前的十万级数据量，显然使用MySQL更加方便，而Hive则显得有一些大材小用了。

2.4 一点其他的比较

在我们本次的项目中，MySQL的引擎使用的是InnoDB。事实上，我们可以有另一种选择 – – MyISM。

InnoDB使用的是聚簇索引，将主键组织到一棵B+树中，而行数据就储存在叶子节点上，若使用 ” where id = 14″ 这样的条件查找主键，则按照B+树的检索算法即可查找到对应的叶节点，之后获得行数据。

MyISM使用的是非聚簇索引，非聚簇索引的两棵B+树看上去没什么不同，节点的结构完全一致只是存储的内容不同而已，主键索引B+树的节点存储了主键，辅助键索引B+树存储了辅助键。表数据存

储在独立的地方，这两颗B+树的叶子节点都使用一个地址指向真正的表数据，对于表数据来说，这两个键没有任何差别。由于索引树是独立的，通过辅助键检索无需访问主键的索引树。

看上去聚簇索引的效率明显要低于非聚簇索引，因为每次使用辅助索引检索都要经过两次B+树查找，这不是多此一举吗？其实不然。

由于行数据和叶子节点存储在一起，这样主键和行数据是一起被载入内存的，找到叶子节点就可以立刻将行数据返回了，如果按照主键Id来组织数据，获得数据更快。

辅助索引使用主键作为”指针”而不是使用地址值作为指针的好处是，减少了当出现行移动或者数据页分裂时辅助索引的维护工作，使用主键值当作指针会让辅助索引占用更多的空间，换来的好处是InnoDB在移动行时无须更新辅助索引中的这个”指针”。也就是说行的位置（实现中通过16K的Page来定位，后面会涉及）会随着数据库里数据的修改而发生变化（前面的B+树节点分裂以及Page的分裂），使用聚簇索引就可以保证不管这个主键B+树的节点如何变化，辅助索引树都不受影响。

在我们的本次项目中，由于数据不存在变化，那么MyISM的辅助索引树并不会受到影响，所以MyISM可能是一种比InnoDB更好的选择。