数据仓库课程设计报告

1. 查询简介与优化介绍

1.1 三种查询适合的情况

1.2 查询优化

1.2.1 mysql

1.2.1.1 查询语句层面

- 使用模糊查询，可以匹配关键字进行查询。
- 优化查询语句中条件的排列顺序，将能筛选掉大量数据的条件语句优先度提高。

1.2.1.2 存储结构层面

- 为数据库建立了时间和评分的b+树索引，提高相关查询的效率。
- 数据表冗余，减少跨表查询需求，提高整体查询速度。

1.2.1.3 硬件与数据库层面

无

1.2.2 hive

1.2.1.1 查询语句层面

- 避免使用COUNT(DISTINCT col)类似的语句，能够减少在进行大数据查询的时候发生故障的可能性
- 进行查询语句的列裁剪，将select *的查询拆分为多个属性的查询，避免无效数据的处理
- 减少聚集函数的使用，因为在hive中使用聚集函数是非常耗时间的

1.2.1.2 存储结构层面

存储的时候避免使用很多小文件，这样会导致hive的查询产生瓶颈，导致查询的速度非常慢，因此使用大文件进行直接的查询，在这里采用了一张宽表存储所有数据的形式

1.2.1.3 硬件与数据库层面

分布式文件系统的运行速度同样会收到网络和搭建环境的影响，我们在搭建hadoop集群的时候出于个人设备和经济等问题选择在阿里云上搭建hadoop环境，防止宿主机性能不足所造成的延迟，并且选择搭建伪分布环境来减少节点之间进行网络通信的延迟。

1.2.3 neo4j

1.2.1.1 查询语句层面

- 减少类似于order by，count等关键词，可以将这部分的计算转移到后端或者是前端去完成
- 如果使用精确查找来代替使用了模糊查询的正则匹配的查询，可以提升查询效率
  - match (n:Product) where n.Title='Shelter' return count(n)

    耗费时间数据如下：

    Started streaming 1 records after 1 ms and completed after 73 ms.
  - match (n:Product) where n.Title=~ '.*bShelter*b.' return count(n)

    耗费时间数据如下：

    Started streaming 1 records after 1 ms and completed after 327 ms.

  然而在真实的业务场景中，除了正则匹配的查询，甚至需要更多模糊查询，虽然精确查询可以减少时间开销，但是在某些查询的业务功能要求下是无法使用的。

  不过对于类似于查询电影种类，年份月份等可以精确化的内容，可以使用这种方式减少时间开销

1.2.1.2 存储结构层面

- **拆分节点**

  实现思路大体上是通过拆分Product节点中的属性成为单独的节点，与源节点建立关系，通过这种方式在查询的过程中可以通过查询较少的节点来快速减少查找项。

  需要补充的是，对于年月日的查找，根据整理出来的csv，电影的时间跨度为约为170年，按照一年12月，一月31天计算，有接近6w节点，而清洗过后的数据约为11w，此时创建年月日节点进行查询显然不划算。

  对应类似演员这样的节点，需要在电影中以字符串拼接的形成存储，在查询的过程中需要做正则运算，更加加大了时间开销，这时候这种手法就更加合适了。

  然而，如果做拆分或者增加连接不能够减少查询的节点数量，则对于时间几乎没有提升。比如对于(Actor)-[]-(Title)-[]-(Actor)这样的节点关系，再增加Actor和Actor的直接联系对于时间也没有减少：

  如：

```
match (m:Actor)-[r:Director_Actor]-(n:Director) where m.Actor='Zac Efron' return n,count(n) order by count(n) desc
```

  Started streaming 6 records after 1 ms and completed after 53 ms.

```
match (m:Actor)-[:Title_Actor]-(r:Title)-[:Title_Director]-(n:Director) where m.Actor='Zac Efron' return n
```

  Started streaming 11 records after 1 ms and completed after 50 ms.

  - 时间拆分
  - 月
    - 通过查询月节点来找到对应的product：

```
 match (n:Product)-[:ProductId_Month]-(m:Month) where m.Month=12 return count(n)
```

  时间：Started streaming 1 records after 1 ms and completed after 11 ms.

    - 直接查询：

```
match (n:Product) where n.Month=12 return count(n)
```

Started streaming 1 records after 1 ms and completed after 121 ms.

效率相差越10倍

- 年
  - 通过查询年节点来找到对应的product：

```
match (n:Product)-[:ProductId_Year]-(m:Year) where m.Year=2014 return count(n)
```

Started streaming 1 records in less than 1 ms and completed after 2 ms.

  - 直接查询：

```
match (n:Product) where n.Year=2014 return count(n)
```

Started streaming 1 records after 1 ms and completed after 59 ms.

- 年月组合查询
  - 通过查询年月节点来找到对应的product：

```
match (n:Product)-[:ProductId_Yearmonth]-(m:Yearmonth) where m.Yearmonth="2014.2" return count(n)
```

Started streaming 1 records after 1 ms and completed after 2 ms.

  - 直接查询

```
match (n:Product) where n.Year=2014 and n.Month=2 return count(n)
```

Started streaming 1 records after 1 ms and completed after 77 ms.

- 这样的手段可以用于其他的节点的拆分，比如title节点中分离出对应的演员，查询某个演员参演过哪些电影的过程中，可以使用同样的手段。
- 同时由于演员在电影中是使用字符串存储多个演员，所以在直接查询的过程中需要遍历电影的过程中还需要做正则运算，加大了开销。
  - 使用演员查询电影：
    - 优化查询

```
match (:Product)-[:Productid_Title]-(:Title)-[:Title_Actor]-(m:Actor) where m.Actor='George Macready' return count(m)
```

Started streaming 1 records after 1 ms and completed after 66 ms.

    - 直接查询

```
match (m:Product) where m.Actor=~ '.\bGeorge Macready\b.' return count(m)
```

Started streaming 1 records after 1 ms and completed after 497 ms.

- **建立索引**
  - 建立索引与拆分节点一样是通过冗余来获取速度的提升的手段，在这里建立的是B+树索引，通过建立索引来比较优化前后性能，同时对比拆分节点的情况，发现两者的开销类似。但是对比拆分节点的做法，建立索引不需要维护这么多表，将工作转移给了数据库，可以专心处理业务逻辑，对于通过建立索引能够解决的问题，使用索引是很好的方案，对于演员这些索引无法解决的问题，拆分节点更有优势

针对title节点的Title属性建立索引：

create index on :Title(Title)

match (n:Title) where n.Title=”Voodoo Dawn” return n

索引前后数据为

Started streaming 1 records after 1 ms and completed after 28 ms.

Started streaming 1 records after 1 ms and completed after 2 ms.

将索引与拆分节点对比：

- 未优化时间情况：
  - Started streaming 1 records in less than 1 ms and completed after 71 ms.
- 拆分节点情况：
  - Started streaming 1 records after 1 ms and completed after 7 ms.
- 建立索引情况：
  - Started streaming 1 records after 2 ms and completed after 6 ms.

1.2.1.3 硬件与数据库层面

- 预热加载
  - 对于同一个查询

```
match (n:Product) where n.Title='Doomsday' return n
```

使用

```
MATCH (n)
OPTIONAL MATCH (n)-[r]->()
RETURN count(n.prop) + count(r.prop);
```

预热前后的查询时间为407ms和93ms

- APOC

使用neo4j插件完成类似的预热加载操作

使用命令：

```
call apoc.warmup.run(true)
```

时间开销为275 ms，效果不如直接遍历节点和属性好

- 调整内存堆大小

内存堆初始大小和最大大小从512m提高到2048m，对于查询：

```
match (m:Actor)-[:Actor_Actor]-(n:Actor)-[:Director_Actor]-(k:Director)
where m.Actor='Eric Kot' return count(k)
```

时间从60ms减少到45ms

- 调整页面缓存

通过负优化来推断提高页面缓存能够提高查询效率

dbms.memory.pagecache.size=10g改为：

dbms.memory.pagecache.size=1m

查询：

```
match (m:Actor)-[:Actor_Actor]-(n:Actor)  return count(n)
```

时间从713 ms增加到9026ms，反过来证明提高页面缓存能够提升查询效率

- 固态硬盘和机械硬盘的情况没有区别，虽然在独写速度上两种硬盘有快慢之分，但是对于这个项目来说数据量还没有达到瓶颈，所以没有区别

1.2.4 前端角度可以做的工作

从用户角度来看，查询的速度快慢与否只取决于发起请求到拿到数据的过程的时间跨度，如果请求的数据量很大，那么服务器的带宽也是制约查询速度的瓶颈之一。那么可以将部分的工作从数据库以及后端转嫁到前端完成。

部分数据直接缓存到web端，通过后端返回的数据渲染前端页面，减轻服务器压力，能够更好地处理高并发的请求，同时前端在获得请求数据之后会摒弃不需要的数据从而加快前端页面渲染。

2. 数据质量

- 为了确保数据质量，我们首先得确保数据来源的正确，在本项目中，对于普通的amazon页面，其中的数据有一部分是本身就有丢失，还有一部分是在爬虫的过程中没有爬取到detail，我们发现来自黑色页面的数据更加准确，更加格式化，其中的电影类型也更加标准。
- 但是在观察网页的过程中发现，大部分网页时能够通过商品选项相互连接的，对于这些页面，我们将他们认为是同一个电影，放入同一个集合中，等到把所有的页面全部解析完，得到一个并查集，那么，对于一个集合中的电影，可以认为他们拥有相同的导演，演员，类型（除了商品上架时间等与电影无关的信息）。根据上一条中我们可以知道，黑色的电影页面的信息更加标准，同时也是可以被连接到的，所以我们可以使用这种方法来补充数据。
- 为了保证数据的统一，我们需要确保数据使用相同的语言，我们在数据清洗的过程中发现在本应为英语的数据中出现了其他语种的数据，这会使得我们的数据查询结果错误，或者查询程序的运行过程中出现问题，为了解决该问题我们调用了百度翻译api，将这些其他语种的数据翻译成英语，保证数据的语言统一。但是由于百度翻译的200w字符的限制，我们无法无限制的使用，所以主要针对的是电影的类型进行翻译，将法语或其他语言的类型转化成英语进行统计
- 在合并电影演员、导演、类型的过程中，基于第二次作业的经验，我们检查的这些数据，发现其中有着相当数量拼写错误，于是我们采用了距离编辑算法来消除这些误差确保演员的唯一性，但是编辑距离的选取是基于直觉的，所以在这方面无法完全保证数据的质量。
- 在筛选电影页面的过程中，我们意识到直觉的选择评判规则是有问题的，于是使用词频统计来计算在电影和类型中出现的高频词，然后对于其中频率较高的词语做直接的人工审核来判断有这个单词的是否是电影，以及这个词语是否是与电影相关的信息，如VHS等与电影无关但是不能认为不是电影的我们直接删除关键词，对于类似于Season这种表示电视剧或者动漫的季的词语，我们直接删除product。同时对于错误词和高频词的维护的工作我们在项目过程中持续经行来保证结果的正确性。
- 然后对于处理过的电影名合并，对于有相同导演和电影名的电影我们认为是一部电影据继续合并。在这次合并完成之后，还有数据项确实的product，我们直接删除，然后进行数据库的导入工作

3. 数据血缘

1. 定位异常数据。在底层数据产生波动的情况下，使用血缘分析配合数据质量进行定位，方便知道异常数据出现在哪。
2. 数据仓库优化。对表和字段的使用频率进行统计，找到使用较多的表和字段，判断其是否存在重复计算和数据浪费的情况，并以此重新设计表和字段。
3. 提升ETL链路效率。通过对小数据量的样本数据清洗任务的时间和结果统计分析，寻找ETL链路的时间瓶颈，确认各个数据清洗任务的优先级，在对源数据进行数据清晰的过程中按照分析结果安排顺序，从而提升ETL链路效率。
4. 对于最终得到的结果的csv文件，我们发现其中有一项有着异常多的导演和演员，我们根据他的id返回查看商品id的并查集，发现他本身并没有这这么多的id（96个），所以我们意识到问题所在不是在并查集的过程中，而是在第二次电影合并的过程中，错误地将某一个集合和另一个集合合并，导致拥有相同导演和电影名，导致了连锁的集合的合并，产生

了一个非常异常的数据。通过数据血缘的关系，我们很快地定位了出现问题的算法的位置。

5. 同时，在数据下游，有时候会出现一些难以理解的数据，比如单独的括号引号或者换行符，这时候通过product id定位上一级的数据，检查是否是上一级出现的问题，如果不是，则追踪到在上一集的product id，检查到底是数据源本身有问题还是算法出现了问题。