22丨反范式设计：3NF有什么不足，为什么有时候需要反范式设计？

2019-07-31 陈旸

上一篇文章中，我们介绍了数据表设计的三种范式。作为数据库的设计人员，理解范式的设计以及反范式优化是非常有必要的。

为什么这么说呢？了解以下几个方面的内容之后你就明白了。

1. 3NF有什么不足？除了3NF，我们为什么还需要BCNF？
2. 有了范式设计，为什么有时候需要进行反范式设计？
3. 反范式设计适用的场景是什么？又可能存在哪些问题？

BCNF（巴斯范式）

如果数据表的关系模式符合3NF的要求，就不存在问题了吗？我们来看下这张仓库管理关系warehouse_keeper表：

仓库名	管理员	物品名	数量
北京仓	张三	iphone XR	10
北京仓	张三	iphone 7	20
上海仓	李四	iphone 7p	30
上海仓	李四	iphone 8	40

在这个数据表中，一个仓库只有一个管理员，同时一个管理员也只管理一个仓库。我们先来梳理下这些属性之间的依赖关系。

仓库名决定了管理员，管理员也决定了仓库名，同时（仓库名，物品名）的属性集合可以决定数量这个属性。

这样，我们就可以找到数据表的候选键是（管理员，物品名）和（仓库名，物品名），

然后我们从候选键中选择一个作为主键，比如（仓库名，物品名）。

在这里，主属性是包含在任一候选键中的属性，也就是仓库名，管理员和物品名。非主属性是数量这个属性。

如何判断一张表的范式呢？我们需要根据范式的等级，从低到高来进行判断。

首先，数据表每个属性都是原子性的，符合1NF的要求；其次，数据表中非主属性”数量“都与候选键全部依赖，（仓库名，物品名）决定数量，（管理员，物品名）决定数量，因此，数据表符合2NF的要求；最后，数据表中的非主属性，不传递依赖于候选键。因此符合3NF的要求。

既然数据表已经符合了3NF的要求，是不是就不存在问题了呢？我们来看下下面的情况：

1. 增加一个仓库，但是还没有存放任何物品。根据数据表实体完整性的要求，主键不能有空值，因此会出现插入异常；
2. 如果仓库更换了管理员，我们就可能会修改数据表中的多条记录；
3. 如果仓库里的商品都卖空了，那么此时仓库名称和相应的管理员名称也会随之被删除。

你能看到，即便数据表符合3NF的要求，同样可能存在插入，更新和删除数据的异常情况。

这种情况下该怎么解决呢？

首先我们需要确认造成异常的原因：主属性仓库名对于候选键（管理员，物品名）是部分依赖的关系，这样就有可能导致上面的异常情况。人们在3NF的基础上进行了改进，提出了**BCNF，也叫做巴斯-科德范式，它在3NF的基础上消除了主属性对候选键的部分依赖或者传递依赖关**

系。

根据BCNF的要求，我们需要把仓库管理关系warehouse_keeper表拆分成下面这样：

仓库表：（仓库名，管理员）

库存表：（仓库名，物品名，数量）

这样就不存在主属性对于候选键的部分依赖或传递依赖，上面数据表的设计就符合BCNF。

反范式设计

尽管围绕着数据表的设计有很多范式，但事实上，我们在设计数据表的时候却不一定要参照这些标准。

我们在之前已经了解了越高阶的范式得到的数据表越多，数据冗余度越低。但有时候，我们在设计数据表的时候，还需要为了性能和读取效率违反范式化的原则。反范式就是相对范式化而言的，换句话说，就是允许少量的冗余，通过空间来换时间。

如果我们想对查询效率进行优化，有时候反范式优化也是一种优化思路。

比如我们想要查询某个商品的前1000条评论，会涉及到两张表。

商品评论表product_comment，对应的字段名称及含义如下：

字段	comment_id	product_id	comment_text	comment_time	user_id
含义	商品评论ID	商品ID	评论内容	评论时间	用户ID

用户表user，对应的字段名称及含义如下：

字段	user_id	user_name	create_time
含义	用户ID	用户昵称	注册时间

下面，我们就用这两张表模拟一下反范式优化。

实验数据：模拟两张百万量级的数据表

为了更好地进行SQL优化实验，我们需要给用户表和商品评论表随机模拟出百万量级的数据。我们可以通过存储过程来实现模拟数据。

下面是给用户表随机生成100万用户的代码：

```
CREATE DEFINER=`root`@`localhost` PROCEDURE `insert_many_user`(IN start INT(10), IN max_num INT(10))
BEGIN
DECLARE i INT DEFAULT 0;
DECLARE date_start DATETIME DEFAULT ('2017-01-01 00:00:00');
DECLARE date_temp DATETIME;
SET date_temp = date_start;
SET autocommit=0;
REPEAT
SET i=i+1;
SET date_temp = date_add(date_temp, interval RAND()*60 second);
INSERT INTO user(user_id, user_name, create_time)
VALUES((start+i), CONCAT('user_',i), date_temp);
UNTIL i = max_num
END REPEAT;
COMMIT;
END
```

我用date_start变量来定义初始的注册时间，时间为2017年1月1日0点0分0秒，然后用date_temp变量计算每个用户的注册时间，新的注册用户与上一个用户注册的时间间隔为60秒内的随机值。然后使用REPEAT ...UNTIL ...END REPEAT循环，对max_num个用户的数据进行计算。在循环前，我们将autocommit设置为0，这样等计算完成再统一插入，执行效率更高。

然后我们来运行call insert_many_user(10000, 1000000);调用存储过程。这里需要通过start和max_num两个参数对初始的user_id和要创建的用户数量进行设置。运行结果：

```
mysql> call insert_many_user(10000, 1000000);
Query OK, 0 rows affected (1 min 37.97 sec)
```

你能看到在MySQL里，创建100万的用户数据用时1分37秒。

接着我们再来给商品评论表product_comment随机生成100万条商品评论。这里我们设置为给某一款商品评论，比如product_id=10001。评论的内容为随机的20个字母。以下是创建随机的100万条商品评论的存储过程：

```
CREATE DEFINER=`root`@`localhost` PROCEDURE `insert_many_product_comments`(IN START INT(10), IN ma
BEGIN
DECLARE i INT DEFAULT 0;
DECLARE date_start DATETIME DEFAULT ('2018-01-01 00:00:00');
DECLARE date_temp DATETIME;
DECLARE comment_text VARCHAR(25);
DECLARE user_id INT;
SET date_temp = date_start;
SET autocommit=0;
REPEAT
SET i=i+1;
SET date_temp = date_add(date_temp, INTERVAL RAND()*60 SECOND);
SET comment_text = substr(MD5(RAND()),1, 20);
SET user_id = FLOOR(RAND()*1000000);
INSERT INTO product_comment(comment_id, product_id, comment_text, comment_time, user_id)
VALUES((START+i), 10001, comment_text, date_temp, user_id);
UNTIL i = max_num
END REPEAT;
COMMIT;
END
```

同样的，我用date_start变量来定义初始的评论时间。这里新的评论时间与上一个评论的时间间隔还是60秒内的随机值，商品评论表中的user_id为随机值。我们使用REPEAT ...UNTIL ...END REPEAT循环，来对max_num个商品评论的数据进行计算。

然后调用存储过程，运行结果如下：

```
mysql> call insert_many_product_comments(10000,1000000);
Query OK, 0 rows affected (2 min 6.98 sec)
```

MySQL一共花了2分7秒完成了商品评论数据的创建。

反范式优化实验对比

如果我们想要查询某个商品ID，比如10001的前1000条评论，需要写成下面这样：

```
SELECT p.comment_text, p.comment_time, u.user_name FROM product_comment AS p
LEFT JOIN user AS u
ON p.user_id = u.user_id
WHERE p.product_id = 10001
ORDER BY p.comment_id DESC LIMIT 1000
```

运行结果（1000条数据行）：

comment_text	comment_time	user_name
462eed7ac6e791292a79	2018-12-14 04:53:25	user_546655
56910cefd01f6d80f0c7	2018-12-14 04:52:35	user_50353
.....	.....	.....
52f6a51769daf701bc68	2018-12-13 20:35:28	user_698675

运行时长为0.395秒，查询效率并不高。

这是因为在实际生活中，我们在显示商品评论的时候，通常会显示这个用户的昵称，而不是用户ID，因此我们还需要关联product_comment和user这两张表来进行查询。当表数据量不大的时候，查询效率还好，但如果表数据量都超过了百万量级，查询效率就会变低。这是因为查询会在product_comment表和user表这两个表上进行聚集索引扫描，然后再嵌套循环，这样一来查询所耗费的时间就有几百毫秒甚至更多。对于网站的响应来说，这已经很慢了，用户体验会非常差。

如果我们想要提升查询的效率，可以允许适当的数据冗余，也就是在商品评论表中增加用户昵称字段，在product_comment数据表的基础上增加user_name字段，就得到了product_comment2数据表。

你可以在百度网盘中下载这三张数据表product_comment、product_comment2和user表，密码为n3l8。

这样一来，只需单表查询就可以得到数据集结果：

```
SELECT comment_text, comment_time, user_name FROM product_comment2 WHERE product_id = 10001 ORD
```

运行结果（1000条数据）：

comment_text	comment_time	user_name
462eed7ac6e791292a79	2018-12-14 04:53:25	user_546655
56910cefd01f6d80f0c7	2018-12-14 04:52:35	user_50353
.....	.....	.....
52f6a51769daf701bc68	2018-12-13 20:35:28	user_698675

优化之后只需要扫描一次聚集索引即可，运行时间为0.039秒，查询时间是之前的1/10。 你能看到，在数据量大的情况下，查询效率会有显著的提升。

反范式存在的问题&适用场景

从上面的例子中可以看出，反范式可以通过空间换时间，提升查询的效率，但是反范式也会带来一些新问题。

在数据量小的情况下，反范式不能体现性能的优势，可能还会让数据库的设计更加复杂。比如采用存储过程来支持数据的更新、删除等额外操作，很容易增加系统的维护成本。

比如用户每次更改昵称的时候，都需要执行存储过程来更新，如果昵称更改频繁，会非常消耗系统资源。

那么反范式优化适用于哪些场景呢？

在现实生活中，我们经常需要一些冗余信息，比如订单中的收货人信息，包括姓名、电话和地址等。每次发生的订单收货信息都属于历史快照，需要进行保存，但用户可以随时修改自己的信息，这时保存这些冗余信息是非常有必要的。

当冗余信息有价值或者能大幅度提高查询效率的时候，我们就可以采取反范式的优化。

此外反范式优化也常用在数据仓库的设计中，因为数据仓库通常存储历史数据，对增删改的实时性要求不强，对历史数据的分析需求强。这时适当允许数据的冗余度，更方便进行数据分析。

我简单总结下数据仓库和数据库在使用上的区别：

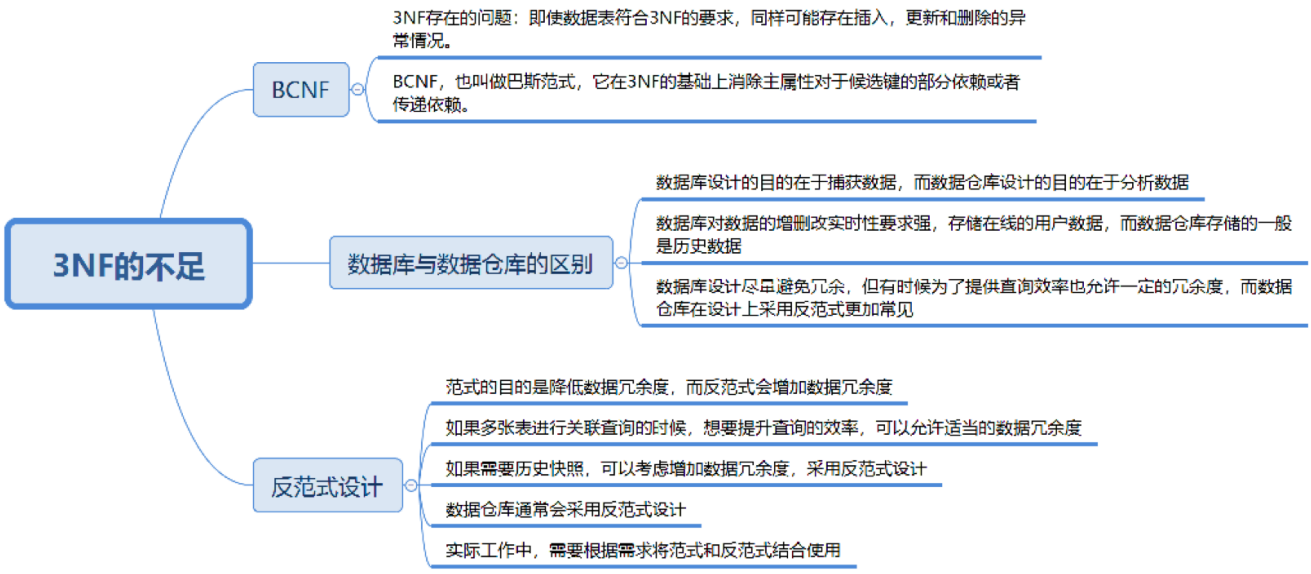
1. 数据库设计的目的在于捕获数据，而数据仓库设计的目的在于分析数据；
2. 数据库对数据的增删改实时性要求强，需要存储在线的用户数据，而数据仓库存储的一般是历史数据；
3. 数据库设计需要尽量避免冗余，但为了提高查询效率也允许一定的冗余度，而数据仓库在设计上更偏向采用反范式设计。

总结

今天我们讲了BCNF，它是基于3NF进行的改进。你能看到设计范式越高阶，数据表就会越精

细，数据的冗余度也就越少，在一定程度上可以让数据库在内部关联上更好地组织数据。但有时候我们也需要采用反范进行优化，通过空间来换取时间。

范式本身没有优劣之分，只有适用场景不同。没有完美的设计，只有合适的设计，我们在数据表的设计中，还需要根据需求将范式和反范式混合使用。



我们今天举了一个反范式设计的例子，你在工作中是否有做过反范式设计的例子？欢迎你在评论区与我们一起分享，也欢迎把这篇文章分享给你的朋友或者同事，一起交流一下。

精选留言

老毕

7

一言以蔽之：反范式无处不在。

最近正在基于Hadoop建设某国企的数据集市项目（地域性非全网），恰如老师所言，我们就是在遵循反范式的设计。

简要来说，我们把数据加工链路分为四层，从下到上依次为：ODS贴源层、DWD明细层、DWS汇总层和ADS应用层。

多源异构的业务数据被源源不断ETL到ODS贴源层之后，经过清洗、规范、转换、拼接等，生成各类宽表存储在DWD明细层；再根据业务模型设计，以这些宽表为基础，生成各类标准的指标数据存储在DWS汇总层；ADS层则基于DWS层的汇总指标再度组合，计算得出应用层数据，直接面向业务需求。

在这样的系统设计中，反范式不仅体现在“宽表”的设计中，更体现在数据加工链路的完整生命周期中——上层都是对下层的冗余。

2019-07-31

盛

1

个人对于反范式的理解是：它会造成数据的冗余甚至是表与表之间的冗余；不过它最大的好处是减少了许多跨表查询从而大幅减少了查询时间。早期的设计其实一直强调范式化设计，可是当memcache出现后-其实就反向在揭示范式的不足。
互联网行业和传统行业最大的区别是要求相应时间的短暂：这就造成了效率优先，这其实也是为何互联网行业的技术更新和使用走在最前面。曾经经历过设计表的过程中尽力追求范式，可是最终发现带来的问题就是性能的不足；范式其实就是规范，可是完完全全的规范-碰到特殊场景就不能那样使用。10年前接触到非关系型数据库时就引发了这种思考，sql server和mysql的机制和查询特长的不同更加引发了自己对于范式的反思。
其实不同数据库对于范式的操作应当是不同的不同行业对于效率的要求是不同的：我觉得范式与反范式的关系可能有点像现在关系型数据库和非关系型数据库的使用一样，已经不再是单一化，如何让二者合理结合最大发挥数据库的查询效率才是关键-只有最合适的没有最好的；当我们过度的追求标准化时反而会忽视了产品真实的功能者作用，如何充分合理发挥产品性能其实才是我们所追求的。
老师觉得呢：没有最标准的，任何方式都有缺陷，没有最好的只有最合适的；就像Google 的SRE中有句经典的话“没有问题的程序是程序的特殊状态”。

2019-07-31

川杰

0

老师您好，想问个问题；假设我在存储过程中，用到了一个临时表（作用就是保存中间数据以便后续做其他操作），先对临时表进行数据删除操作，然后对临时表进行插入操作。假设现在有两个人A,B同时调用该存储过程，是否存在如下风险，即：A执行存储过程时，正在删除数据，同一时刻，B执行存储过程时，新增数据？

2019-08-01

ABC

0

有一个问题，请问老师，如果一个字段内容存的是:

会员ID@会员名称

这样是不是算违反第一范式？在工作中遇到过类似方式存储的数据，但由于历史数据和牵涉过多的原因，已经无法修改。。

在每次做统计类需求的时候，就会用反范式设计，方便查询，而且速度会很快。

2019-08-01

床头猫

0

老师你好，有个问题，就是我这里有四张表都是1对1关联的，数据量大概四千多万，用left join和分四条sql查，哪个更好一点，oracle数据库，两种方式都会命中索引

2019-07-31

夜路破晓

0

范式与反范式，正如传统与解构，规则与务实，稳定与突破，守成与创新，是阴阳动静的矛盾关系，两者一而二，二而一，即和而不同、求同存异，落脚点是务实，也就是应用场景和业务需求。
所以说，这已经不单是数据库设计的问题，而中国哲学体系在互联网商业中实践指导。
数据库设计提出范式的同时存在反范式的要求，符合否定之否定的螺旋上升轨迹，是数据库也是SQL语言保持强壮生命力而经久不衰的重要原因，是现实生存逻辑的映射。

2019-07-31

全有

0

老师你好，抛开本次课程，问一个现象：
慢查询日志如下：
Query_time: 10.612971 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 0
SET timestamp=1564551836;
commit;

两个问题：
1：Commit 是如何发生的，做啦什么事？
2: 所耗时长为什么会那么久？

2019-07-31

许童童

0

老师你好，问个问题。如果用记表国用户名称字段修改了，那评论表中用户名称是否要跟着改

呢。这个怎么处理？

2019-07-31