# 一、数据库设计第一、第二和第三范式

范式是关系数据库理论的基础，也是我们在设计数据库结构过程中所要遵循的规则和指导方法。目前有迹可寻的共有8种范式，依次是：1NF，2NF，3NF，BCNF，4NF，5NF，DKNF，6NF。通常所用到的只是前三个范式，即：第一范式（1NF），第二范式（2NF），第三范式（3NF）。  
**1、第一范式（1NF）：强调的是列的原子性，即列不能够再分成其他几列。**   
 考虑这样一个表：【联系人】（姓名，性别，电话）   
 如果在实际场景中，一个联系人有家庭电话和公司电话，那么这种表结构设计就没有达到 1NF。要符合 1NF 我们只需把列（电话）拆分，即：【联系人】（姓名，性别，家庭电话，公司电话）。1NF 很好辨别，但是 2NF 和 3NF 就容易搞混淆。

**2、第二范式（2NF）：首先是 1NF，另外包含两部分内容，一是表必须有一个主键；二是没有包含在主键中的列必须完全依赖于主键， 而不能只依赖于主键的一部分。** 考虑一个订单明细表：【OrderDetail】（OrderID，ProductID，UnitPrice，Discount，Quantity，ProductName）。   
 因为我们知道在一个订单中可以订购多种产品，所以单单一个 OrderID 是不足以成为主键的，主键应该是（OrderID，ProductID）。显而易见 Discount（折扣），Quantity（数量）完全依赖（取决）于主键（OderID，ProductID），而 UnitPrice，ProductName 只依赖于 ProductID。所以 OrderDetail 表不符合 2NF。

产生的问题：不符合 2NF 的设计容易产生冗余数据。  
解决方法： 可以把【OrderDetail】表拆分为【OrderDetail】（OrderID，ProductID，Discount，Quantity）和【Product】（ProductID，UnitPrice，ProductName）来消除原订单表中UnitPrice，ProductName多次重复的情况。

3、**第三范式（3NF）：首先是 2NF，另外非主键列必须直接依赖于主键，不能存在传递依赖。即不能存在：非主键列 A 依赖于非主键列 B，非主键列 B 依赖于主键的情况。**   
 考虑一个订单表【Order】（OrderID，OrderDate，CustomerID，CustomerName，CustomerAddr，CustomerCity）主键是（OrderID）。   
其中 OrderDate，CustomerID，CustomerName，CustomerAddr，CustomerCity 等非主键列都完全依赖于主键（OrderID），所以符合 2NF。

不过问题是 CustomerName，CustomerAddr，CustomerCity 直接依赖的是 CustomerID（非主键列），而不是直接依赖于主键，它是通过传递才依赖于主键，所以不符合 3NF。   
通过拆分【Order】为【Order】（OrderID，OrderDate，CustomerID）和【Customer】（CustomerID，CustomerName，CustomerAddr，CustomerCity）从而达到 3NF。   
 第二范式和第三范式区分它们的关键点在于，2NF：非主键列是否完全依赖于主键，还是依赖于主键的一部分；3NF：非主键列是直接依赖于主键，还是直接依赖于非主键列。

# 二、关于数据库方面的面试技巧：如何从建表展现自己能力

在面试java web方面的高级程序员时，我一定会问到 jave core，java web（比如Spring MVC，Hibernate等）和数据库相关问题。在数据库方面，对于java 高级程序员而言，不仅需要会基本的增删改查，而且需要具备一定的“优化”方面的技能。

优化是个大话题，可以从索引，建表和SQL 调优（SQL Tuning）方面入手，这个我们来分析下建表时需要注意的优化点。

我一般会问候选人，“你有没有设计过数据表？”，大多数回答是设计过，接着我会比较阴险地问下：“你在设计表时是否用到了三泛式”？很多计算机专业的候选人往往会随口回答“是”。这时我就不细问了，同时给候选人写下如下的评语，“该候选人有基本的数据库操作的技能，会增删改查操作，但缺乏专业的数据表设计的能力”。

先来看下三泛式的概念：在第三范式里，数据不能存在传递关系。

比如有张订单流水表，其中包括（订单编号，商品编号，下订单的会员编号，商品名，商品价格，会员姓名，会员手机，会员地址）这些信息。在这个表里，就存在两个个传递关系。从商品编号能看到商品价格商品名等信息，从下订单的会员编号能看到会员姓名，手机和地址的信息，所以不符合三泛式 。

如果要按经典学院派的三泛式，我们得把这个表拆分成如下3个表。

|  |  |  |
| --- | --- | --- |
| **订单流水表** | 至少包含订单编号、商品编号和下订单的会员编号 | 假设过去1个月有100万条 |
| **商品表** | 至少包含商品编号和商品名 | 假设过去一个月有50万条商品信息 |
| **会员表** | 至少包含会员编号会员手机会员地址 | 假设过去一个月里有10万名会员下过订单 |

这样拆分的好处（也就是三泛式）的好处：那就是没数据冗余。

不拆分：假设之前的订单流水表包括（订单编号，商品编号，下订单的会员编号，商品名，商品价格，会员姓名，会员手机，会员地址），一定也有张商品表和会员表，这样“商品名“就冗余了（出现在订单流水表和商品表里），“会员姓名“等字段也冗余了（同时也出现在会员表里）。这样做，万一我们得修改会员手机，那么就得到两个表里同时修改，增加了工作量不算，而且还增加了出错的风险（万一哪个表忘记修改了，数据会不一致）。

看上去三泛式很美，但是（很多事情就坏在但是之后），万一在一个大型系统里（比如某宝），数据量很大，就如按上表给出的数据量。那么如果我要执行一个非常基本的需求，要列出过去一个月里所有买过Java书籍的会员的邮箱，以便我们发些推荐邮件。这句SQL语句不复杂，但关键是得“关联”，我们可以用订单流水表 left join商品表 on 订单流水表的商品编号 = 商品表的商品编号，在left join 会员表 on 订单流水表的会员编号 = 会员表的会员编号。

关联是要代价的，这里我们就得做三张大表之间做关联，哪怕我再做优化，再利用到数据库系统的优化（比如用尽Oracle里的优化配置），但由于三个表比较大，关联的样本就大了。

这时，如果我们来看下“比较丑”的做法，就一开始把所有字段写到一个表里。

订单流水表 =（订单编号，商品编号，下订单的会员编号，商品名，商品价格，会员姓名，会员手机，会员地址）

那么由于不需要关联，性能就很显著提升。

从这个案例中，大家一定能看到，如果某候选人告诉我设计表时都得遵循三泛式，那么我给出的“没设计过数据表”也没冤枉他。

**那么关于设计数据表方面，大家该怎么展示自己的能力呢？分类讨论。**

第一，如果在设计的时候，已经明确地知道这个系统的数据量不会太大，比如一个中学的图书管理系统，最多有5万条书本的数据，过去一个月里借阅记录不会超过1千条。也就是说，表之间的关联代价不会太高，那么用“三范式”的原则是必需的。毕竟三范式能避免数据冗余带来的更新插入上的“需要同时多表里相同字段”的麻烦。

第二，如果表的数据量很大，如前面举的在线购物网站的例子，我们可能就需要冗余数据。在订单流水表里，同时放入用户邮件地址和商品名的字段。（使用冗余换时间效率）

冗余的后果：在得到“免去连接操作”的好处同时，也得付出相应的代价，比如用户一旦更新了邮件地址，那么我们就需要同时在会员表和订单流水表里修改该字段，这就是冗余带来的后果。

也就是说，我在询问如何设计数据表时，我不在乎你之前设计过哪些表？关键看你在设计表的时候需要考虑哪些因素。

大家不仅需要掌握诸如“连接”和“范式”之类的技术，更应该从业务角度，权衡各种“建表代价”，从而挑选一种最符合本项目的解决方案。

# 三、[数据库事务理解](http://blog.csdn.net/qq_26437925/article/details/50739813)

## 1、事物

事物是并发控制的基本单位。所谓的事物，他是一个操作序列，这些操作要么都执行，要么都不执行，它是一个不可分割的工作单位。事物是数据库维护数据一致性的单位，在每个事物结束时，都能保持数据一致性。

## 2、事物的四大特性

* **原子性：**原子性是指事务包含的所有操作要么全部成功，要么全部失败回滚。因此事务的操作如果成功就必须要完全应用到数据库，如果操作失败则不能对数据库有任何影响。
* **一致性：**一致性是指事务必须使数据库从一个一致性状态变换到另一个一致性状态，也就是说一个事务执行之前和执行之后都必

须处于一致性状态。

拿转账来说，假设用户A和用户B两者的钱加起来一共是5000，那么不管A和B之间如何转账，转几次账，事务结束后两个用户的钱相加起来应该还得是5000，这就是事务的一致性。

* **隔离型：**隔离性是当多个用户并发访问数据库时，比如操作同一张表时，数据库为每一个用户开启的事务，不能被其他事务的操作所干扰，多个并发事务之间要相互隔离。

1. 两个更新事物同时修改一条数据时（丢失更新）
2. 一个更新事物更新一条数据时，另一个数据读取了还没提交的更新（脏读）【这个人性子急，我都还没弄好，你这么快就催，结果就错了】

3、一个读取事物读取一条数据时另一个更新事物修改了这条数据，这时就会出现不可重现的读取（不可重复读）

我在做，还没做完，你插手干嘛，跟上面的性子急差不多

4、一个读取事物读取时，另一个插入事物插入了一条新数据，这样就可能多读出一条数据，出现幻读（幻读）

* **持久性 ：**持久性是指一个事务一旦被提交了，那么对数据库中的数据的改变就是永久性的，即便是在数据库系统遇到故障的情况下也不会丢失提交事务的操作。

例如我们在使用JDBC操作数据库时，在提交事务方法后，提示用户事务操作完成，当我们程序执行完成直到看到提示后，就可以认定事务以及正确提交，即使这时候数据库出现了问题，也必须要将我们的事务完全执行完成，否则就会造成我们看到提示事务处理完毕，但是数据库因为故障而没有执行事务的重大错误。

以上介绍完事务的四大特性(简称ACID)，现在重点来说明下事务的隔离性，当多个线程都开启事务操作数据库中的数据时，数据库系统要能进行隔离操作，以保证各个线程获取数据的准确性，在介绍数据库提供的各种隔离级别之前，我们先看看如果不考虑事务的隔离性，会发生的几种问题：

**1，脏读**

　脏读是指在一个事务处理过程里读取了另一个未提交的事务中的数据。

当一个事务正在多次修改某个数据，而在这个事务中这多次的修改都还未提交，这时一个并发的事务来访问该数据，就会造成两个事务得到的数据不一致。例如：用户A向用户B转账100元，对应SQL命令如下

update account set money = money + 100 where name = ’B’; (此时A通知B)

update account set money = money - 100 where name = ’A’;

当只执行第一条SQL时，A通知B查看账户，B发现确实钱已到账（此时即发生了脏读），而之后无论第二条SQL是否执行，只要该事务不提交，则所有操作都将回滚，那么当B以后再次查看账户时就会发现钱其实并没有转。

**2、不可重复读**

　不可重复读是指在对于数据库中的某个数据，一个事务范围内多次查询却返回了不同的数据值，这是由于在查询间隔，被另一个事务修改并提交了。

例如事务T1在读取某一数据，而事务T2立马修改了这个数据并且提交事务给数据库，事务T1再次读取该数据就得到了不同的结果，发送了不可重复读。

不可重复读和脏读的区别是，脏读是某一事务读取了另一个事务未提交的脏数据，而不可重复读则是读取了前一事务提交的数据。

在某些情况下，不可重复读并不是问题，比如我们多次查询某个数据当然以最后查询得到的结果为主。但在另一些情况下就有可能发生问题，例如对于同一个数据A和B依次查询就可能不同，A和B就可能打起来了……

**3，幻读**

　幻读是事务非独立执行时发生的一种现象。例如事务T1对一个表中所有的行的某个数据项做了从“1”修改为“2”的操作，这时事务T2又对这个表中插入了一行数据项，而这个数据项的数值还是为“1”并且提交给数据库。而操作事务T1的用户如果再查看刚刚修改的数据，会发现还有一行没有修改，其实这行是从事务T2中添加的，就好像产生幻觉一样，这就是发生了幻读。

幻读和不可重复读都是读取了另一条已经提交的事务（这点就脏读不同），所不同的是不可重复读查询的都是同一个数据项，而幻读针对的是一批数据整体（比如数据的个数）。

　　现在来看看MySQL数据库为我们提供的四种隔离级别：

1. Serializable (串行化)： 可避免脏读、不可重复读、幻读的发生。

　　② Repeatable read (可重复读)： 可避免脏读、不可重复读的发生。

　　③ Read committed (读已提交)： 可避免脏读的发生。

　　④ Read uncommitted (读未提交)：最低级别，任何情况都无法保证。

以上四种隔离级别最高的是Serializable级别，最低的是Read uncommitted级别，当然级别越高，执行效率就越低。

像Serializable这样的级别，就是以锁表的方式(类似于Java多线程中的锁)使得其他的线程只能在锁外等待，所以平时选用何种隔离级别应该根据实际情况。

**在MySQL数据库中，支持上面四种隔离级别，默认的为Repeatable read (可重复读)；**

**在Oracle数据库中，只支持Serializable (串行化)级别和Read committed (读已提交)这两种级别，其中默认的为Read committed级别。**

## 3、乐观锁和悲观锁

乐观锁和悲观锁是并发控制主要采取的技术手段。

|  |  |
| --- | --- |
| 悲观锁 | 乐观锁 |
| 顾名思义，就是很悲观，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会block，直到它拿到锁。 | 顾名思义，就是很乐观，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号等机制。 |
| 传统的关系型[数据库](http://lib.csdn.net/base/mysql)里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。 | 乐观锁适用于多读的应用类型，这样可以提高吞吐量。 |
| 悲观锁的流程如下：  1. 在对任意记录进行修改前，先尝试为该记录加上排他锁。 2. 如果加锁失败，说明该记录正在被修改，那么当前查询可能要等待或者抛出异常。 3. 如果成功加锁，那么就可以对记录做修改，事务完成后就会解锁了。   注：其间如果有其他对该记录做修改或加排他锁的操作，都会等待我们解锁或直接抛出异常。 |  |

# 四、数据库索引

**1、索引概念：说白了，索引问题就是一个查找问题。**

数据库索引，是对数据库表统中一个或多个列进行排序的数据结构，目的：以协助快速查询、更新数据库表中数据。

在数据之外，数据库系统还维护着满足特定查找[算法](http://lib.csdn.net/base/datastructure)的数据结构，这些数据结构以某种方式引用（指向）数据，这样就可以在这些数据结构上实现高级查找算法。这种数据结构，就是索引。

**2、实现原理：**

索引的实现通常使用B树及其变种B+树。

**3、优点和缺点**

|  |  |
| --- | --- |
| 优点 | 缺点（**也是不能在每一列建立索引的原因**） |
| 1. 通过创建唯一性索引，可以保证数据库表中每一行数据的唯一性； 2. 可以大大加速数据检索的速度； 3. 可以加快表和表之间的连接，特别是在实现数据的参考完整性方面特别有意义。   。。。。。。。。。。。。。 | 1. 创建和维护索引要耗费时间，这种时间随着数据量的增加而增加； 2. 增加了数据库的存储空间； 3. 在插入和修改数据时要耗费较多的时间（因为索引也要随之变动） |

**4、在创建索引的时候，应该考虑在哪些列上可以创建索引，在哪些列上不能创建索引**

|  |  |
| --- | --- |
| 可以创建索引 | 不应该创建索引 |
| 1. 经常需要搜索的列上，可以加快搜索的速度； 2. 作为主键的列上，强制该列的唯一性和组织表中数据的排列结构； 3. 经常需要和其他表进行连接的列上，这些列主要是一些外键，可以加快连接的速度； 4. 经常需要根据范围进行搜索的列上创建索引，因为索引已经排序，其指定的范围是连续的； 5. 经常需要排序的列上创建索引，因为索引已经排序，这样查询可以利用索引的排序，加快排序查询时间； 6. 经常使用在WHERE子句中的列上面创建索引，加快条件的判断速度。 | 1. 对于那些在查询中很少使用或者参考的列不应该创建索引。 2. 对于那些数据值取值很少的列也不应该增加索引。 3. 对于那些定义为text, image和bit数据类型的列不应该增加索引。 4. 当修改性能远远大于检索性能时，不应该创建索引。 |

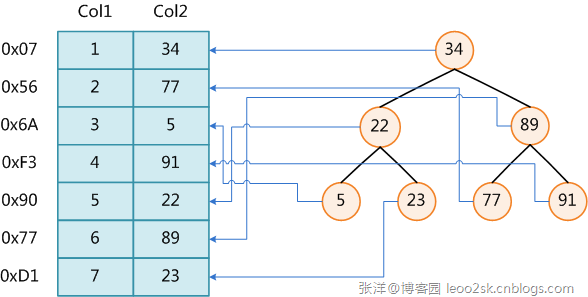
解释：

这是因为，由于这些列的取值很少，例如人事表的性别列，在查询的结果中，结果集的数据行占了表中数据行的很大比例，即需要在表中搜索的数据行的比例很大。增加索引，并不能明显加快检索速度。

这是因为，这些列的数据量要么相当大，要么取值很少。

这是因为，**修改性能和检索性能是互相矛盾的**。当增加索引时，会提高检索性能，但是会降低修改性能。当减少索引时，会提高修改性能，降低检索性能。因此，当修改性能远远大于检索性能时，不应该创建索引。

1. **原理讲解**



上图展示了一种可能的索引方式。左边是数据表，一共有两列七条记录，最左边的是数据记录的物理地址（注意逻辑上相邻的记录在磁盘上也并不是一定物理相邻的）。为了加快Col2的查找，**可以维护一个右边所示的二叉查找树，每个节点分别包含索引键值和一个指向对应数据记录物理地址的指针，这样就可以运用二叉查找在O(log2n)的复杂度内获取到相应数据。**

**5、索引类型**

* 唯一索引：唯一索引是不允许其中任何两行具有相同索引值的索引。

（1）、当现有数据中存在重复的键值时，大多数数据库不允许将新创建的唯一索引与表一起保存。

（2）、数据库还可能防止添加将在表中创建重复键值的新数据。例如，如果在employee表中职员的姓(lname)上创建了唯一索引，则任何两个员工都不能同姓。

* 主键索引：数据库表经常有一列或列组合，其值唯一标识表中的每一行，该列称为表的主键。

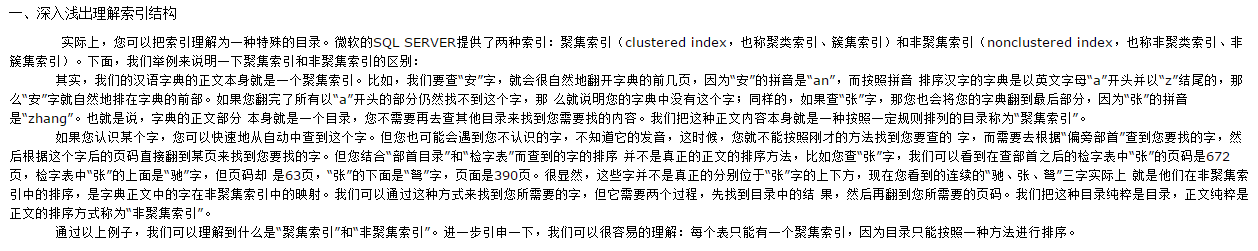
在数据库关系图中为表定义主键将自动创建主键索引，主键索引是唯一索引的特定类型。该索引要求主键中的每个值都唯一。当在查询中使用主键索引时，它还允许对数据的快速访问。

* 聚集索引

在聚集索引中，表中行的物理顺序与键值的逻辑（索引）顺序相同。（把数据移动到连续的存储空间）

一个表只能包含一个聚集索引。

如果某索引不是聚集索引，则表中行的物理顺序与键值的逻辑顺序不匹配。与非聚集索引相比，聚集索引通常提供更快的数据访问速度。



**6、创建索引:**  
　  单一索引:Create Index <Index-Name> On <Table\_Name> (Column\_Name);

　  复合索引: Create Index i\_deptno\_job on emp(deptno,job); —>在emp表的deptno、job列建立索引。

聚集索引：create cluster index Stustname on Student(Sname);

    修改索引：ALTER INDEX emp\_ix REBUILD REVERSE;

删除索引：drop index pk\_dept;

# 五、存储过程以及优缺点

1、存储过程就是一些预编译的SQL语句。直白的理解：存储过程可以说是一个记录集，他是由一些预编译SQL语句组成的代码块，这些预编译的SQL语句代码像一个方法一样实现一些功能，然后在给这个代码块取一个名字，在用到的时候调用它。

|  |  |
| --- | --- |
| 优点 | 缺点 |
| 存储过程是一个预编译的代码块，所以执行效率比较高 | 调试麻烦 |
| 调用一个名字可以代替大量SQL语句，可以降低网络通信量，提高通信效率 | 移植问题 |
| 在一定程度上提高数据安全，防止SQL注入 | 重新编译问题、维护麻烦 |

# 十、SQL语法

## 1、简单说一下drop、delete 和 truncate的区别

* 相同点：

1. truncate和不带where子句的delete、以及drop都会删除表内的数据。

* 不同点：

1. truncate 和 delete 只删除数据不删除表的结构(定义)；

drop 语句将删除表的结构以及被依赖的约束(constrain)、触发器(trigger)、索引(index)；

1. delete 语句是数据库操作语言(dml)， 不会自动提交，事务提交之后才生效；如果有相应的 trigger，执行的时候将被触发；

truncate、drop 是数据库定义语言(ddl)，操作立即生效，不能回滚，操作不触发 trigger。

3.速度，一般来说: drop> truncate > delete

5.安全性：小心使用 drop 和 truncate，尤其没有备份的时候.否则哭都来不及

6、使用上：

（1）、想删除部分数据行用 delete，注意带上where子句；  
（2）、想删除表,当然用 drop  
（3）、想保留表而将所有数据删除，如果和事务无关，用truncate即可。

如果和事务有关,或者想触发trigger,还是用delete。

（4）、如果是整理表内部的碎片，可以用truncate跟上reuse stroage，再重新导入/插入数据。

数据库优化

1、**选择最有效率的表名顺序:** 在FROM子句中包含多个表的情况下,因为执行顺序为从后往前, 所以数据量较少的表尽量放后。

2、 **WHERE子句中的连接顺序：因为**自下而上的顺序解析WHERE子句,所以表之间的连接必须写在其他WHERE条件之前, 那些可以过滤掉最大数量记录的条件必须写在WHERE子句的末尾.

3、 **SELECT子句中避免使用 ‘ \* ‘：** 在解析过程中会将'\*' 依次转换成列名, 这个工作是通过查询数据字典完成的, 这意味着将耗费更多的时间。

4、 **减少访问数据库的次数：** 在工作中的经验就是for循环中尽量不能放sql语句，在for之前一次性查出，在循环组装数据。

5、 **用Where子句替换HAVING子句。**

6、 **减少对表的查询：**在含有子查询的SQL语句中,要特别注意减少对表的查询.例子：

7、 **通过内部函数提高SQL效率：**复杂的SQL往往牺牲了执行效率. 能够掌握上面的运用函数解决问题。

8、 **使用表的别名(Alias)**

9、 **用EXISTS替代IN、用NOT EXISTS替代NOT IN。**

在子查询中,NOT IN子句将执行一个内部的排序和合并. 无论在哪种情况下,NOT IN都是最低效的：因为它对子查询中的表执行了一个全表遍历。

10、 **用索引提高效率：**

11、 **用EXISTS替换DISTINCT：**   
当提交一个包含一对多表信息(比如部门表和雇员表)的查询时,避免在SELECT子句中使用DISTINCT. 一般可以考虑用EXIST替换, EXISTS 使查询更为迅速,因为RDBMS核心模块将在子查询的条件一旦满足后,立刻返回结果.

12、 **在java代码中尽量少用连接符“＋”连接字符串！**

13、 **避免在索引列上使用NOT ：通常**NOT会产生在和在索引列上使用函数相同的影响. 当ORACLE”遇到”NOT,他就会停止使用索引转而执行全表扫描.

14、 **避免在索引列上使用计算．** WHERE子句中，如果索引列是函数的一部分．优化器将不使用索引而使用全表扫描

15、 **用>=替代>**

16、用IN来替换OR

<http://www.jb51.net/article/19024.htm>