数据库范式

# 简介

数据库范式在数据库设计中的地位一直很暧昧，教科书中对于数据库范式倒是都给出了学术性的定义，但实际应用中范式的应用却不甚乐观，这篇文章会用简单的语言和一个简单的数据库DEMO将一个不符合范式的数据库一步步从第一范式实现到第四范式。范式的目标

应用数据库范式可以带来许多好处，但是最重要的好处归结为三点：

* 减少数据冗余（这是最主要的好处，其他好处都是由此而附带的）
* 消除异常（插入异常，更新异常，删除异常）
* 让数据组织的更加和谐…

但剑是双刃的，应用数据库范式同样也会带来弊端，这会在文章后面说到。

# 函数依赖

## 函数依赖的定义

函数依赖定义：设R(U)是属性集U上的关系模式，X、Y是U的子集，r是R的任一具体关系。设t1、t2是关系r中的任意两个元组，如果t1[X]=t2[X]，有t1[Y]=t2[Y]，则称X函数决定Y，或Y函数依赖于X，记作X→Y 。属性集X称为函数依赖的左边(left-hand side)，而Y则称为右边(right-hand side)

非平凡函数依赖： 在关系模式R(U)中，X, Y是U的子集，若X→Y，且不存在X‘⊂X，使X’→Y，则称X→Y是完全函数依赖(full functional dependency），记作：



否则称 X→Y是部分函数依赖（partial functional dependency），记作：



传递函数依赖：在关系模式R(U)中，X, Y是U的子集，若X→Y，Y→Z，并且X不函数依赖于Y ，则称Z传递函数依赖于X。 这里加上条件X 不函数依赖于Y很重要，如果Y→X则X↔Y，X与Y一一对应，实际上处于等价地位，Z就直接函数依赖于X，而不是传递函数依赖于X。

## 函数依赖的逻辑蕴意

关系模式形式化地表示为: R(U, D, dom, F) ，其中R是关系名，是符号化的元组语义；U为组成关系的属性名的集合；D为属性组U中属性所来自的域；dom 为属性和域之间的映像集合；F 为关系中属性间的依赖关系集合。这个关系模式可以简化为一个三元组：R(U, F)。

定义5.4：设F是关系模式R(U, F)的一个函数依赖集，X、Y是U的子集，若对R的每个满足F的关系r，X→Y都成立，则称F逻辑蕴涵X→Y。记作

F|=X→Y，定义5.4表示函数依赖X→Y可由函数依赖集F推导出。

函数依赖集F的闭包：被F逻辑蕴涵的函数依赖的全体构成的集合， (closure)，记作F+。即 F+={X→Y | F|=X→Y}一般F⊆ F+，若F= F+，则称F为函数依赖的完备集。在现实的例子中，实际上不可能指定函数依赖的完备集。

## 键

设F是关系模式R(U, F)的一个函数依赖集，X、Y是U的子集，若

X→U ξ F+ ;不存在X的真子集Y满足Y→U成立，且Y→U ξ F+

则称X是R的一个后候选键。若有多个候选键，则选定其中一个为主键，其他的候选键称为辅键。

## 函数依赖的推理规则

函数依赖XY→Z，XYZ→UV。设有关系模式R(U, F)，U为属性全集，X、Y、Z为U的子集，F是R的函数依赖集。则对R有以下推理规则：

* A1自反律（reflexivity）：若Y⊆X，则X→Y。
* A2增广律（augmentation）：若X→Y，则XZ→YZ。
* A3传递律（transitivity）：若X→Y，Y→Z，则X→Z。

根据Armstrong公理的三条推理规则，可得到三个推论：

* 合并规则(union)：由X→Y，X→Z，则 X→YZ  
  分解规则(decomposition)：由X→YZ，则 X→Y，X→Z
* 伪传递规则(pseudo transitivity)：由 X→Y，YW→Z，则 XW→Z

引理5.1：设Ai ( i=1, 2, …,n )为关系模式R的属性，X→A1A2…An成立的充分必要条件是X→Ai ( i=1, 2, …,n )均成立。

定义5.6：设由关系模式R(U, F)，U为属性全集即U=A1A2…An，X为U的子集，F是U上的一个函数依赖集，则所有基于F能推导出的由X函数决定的属性集合称为属性集X关于函数依赖集F的闭包，记作X+，即

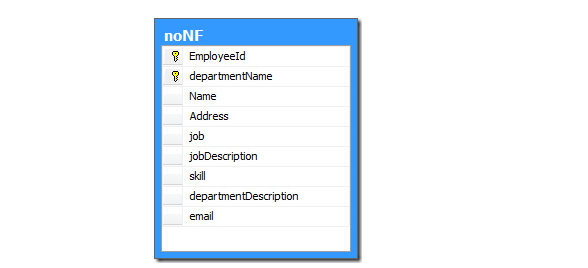
X+={ Ai | X→Ai能由F根据Armstrong公理推导出 }

引理5.2：设F是关系模式R(U, F)的函数依赖集，U为属性全集即U=A1A2…An，X、Y为U的子集，则X→Y能由F根据Armstrong公理导出的充分必要条件是Y⊆ X+

## 函数依赖集的等价、覆盖和最小依赖集

# 什么是范式

简单的说，范式是为了消除重复数据减少冗余数据，从而让数据库内的数据更好的组织，让磁盘空间得到更有效利用的一种标准化标准，满足高等级的范式的先决条件是满足低等级范式。(比如满足2nf一定满足1nf)。让我们先从一个未经范式化的表看起,表如下：

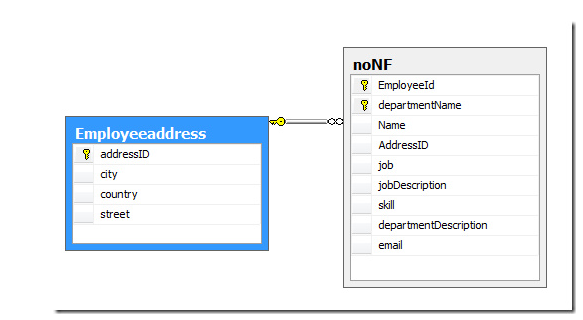


先对表做一个简单说明，employeeId是员工id,departmentName是部门名称，job代表岗位，jobDescription是岗位说明，skill是员工技能，departmentDescription是部门说明，address是员工住址。

## 对表进行第一范式(1NF)

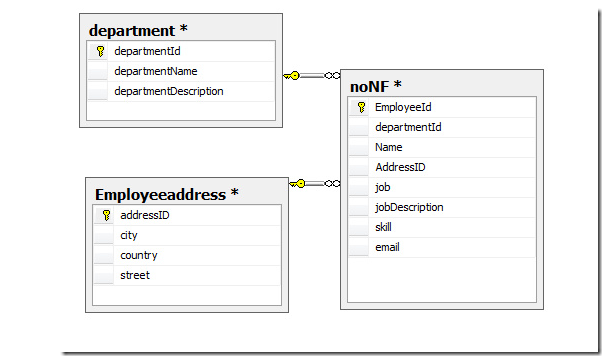
如果一个关系模式R的所有属性都是不可分的基本数据项，则R∈1NF。

简单的说,第一范式就是每一个属性都不可再分。不符合第一范式则不能称为关系数据库。对于上表，不难看出Address是可以再分的，比如”北京市XX路XX小区XX号”，着显然不符合第一范式，对其应用第一范式则需要将此属性分解到另一个表,如下:



## 对表进行第二范式(2NF)

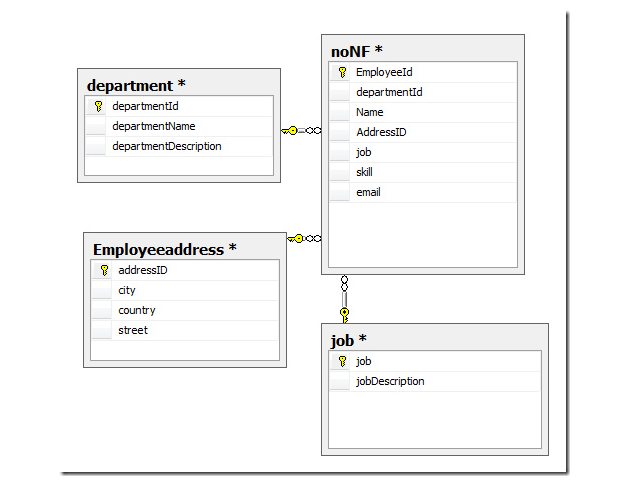
若关系模式R∈1NF，并且每一个非主属性都完全函数依赖于R的键，则R∈2NF。简单的说，是表中的属性必须完全依赖于全部主键，而不是部分主键.所以非主属性完全由主键决定（该实体的属性，而不是其他实体的属性）。这样做的目的是进一步减少插入异常和更新异常。在上表中，departmentDescription是由主键DepartmentName所决定，但却不是由主键EmployeeID决定，所以departmentDescription只依赖于两个主键中的一个，故要departmentDescription对主键是部分依赖，对其应用第二范式如下表：



## 对表进行第三范式(3NF)

关系模式R<U，F> 中若不存在这样的码X、属性组Y及非主属性Z（Z  Y）, 使得X→Y，Y→Z，成立，则称R<U，F> ∈ 3NF。

简单的说，第三范式是为了消除数据库中关键字之间的依赖关系（不存在传递依赖），在上面经过第二范式化的表中，可以看出jobDescription(岗位职责)是由job(岗位)所决定，则jobDescription依赖于job,可以看出这不符合第三范式，对表进行第三范式后的关系图为：



上表中，已经不存在数据库属性互相依赖的问题，所以符合第三范式。

## 对表进行BC范式(BCNF)

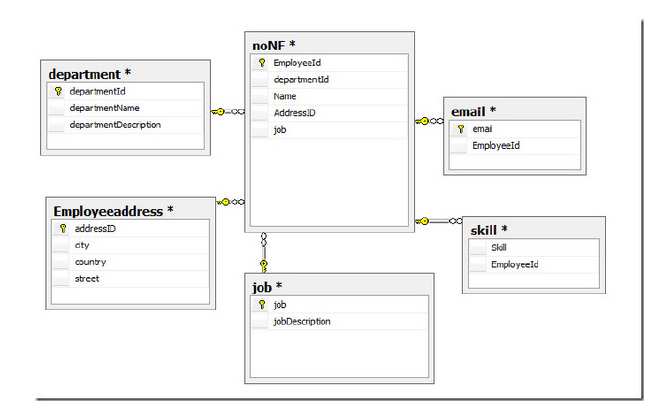
设关系模式R<U，F>∈1NF，如果对于R的每个函数依赖X→Y，若Y不属于X，则X必含有候选键，那么R∈BCNF。

例如关系模式Addr(S,C,Z)，S表示街道名，C表示城市名，Z表示邮政编码。函数依赖集F={（C,S）→Z ,Z→C}其中（C,S）是主键。由于Addr不存在部分依赖和传递依赖，所以满足Addr(S,C,Z) ∈3NF。由于Z→C，Z不包含键，所以不满足BCNF。可以分解为ZC，SZ，这样就满足BCNF。一般属于3NF但不属于BCNF的关系模式不是很多。

## 对表进行第四范式(4NF)

关系模式R<U，F>∈1NF，如果对于R的每个非平凡多值依赖X→→Y（Y  X），X都含有候选码，则R∈4NF。

简单的说，第四范式是消除表中的多值依赖，也就是说可以减少维护数据一致性的工作。对于上面bc范式化的表中，对于员工的skill，两个可能的值是”C#,sql,javascript”和“C#，UML,Ruby”,可以看出，这个数据库属性存在多个值，这就可能造成数据库内容不一致的问题，比如第一个值写的是”C#”,而第二个值写的是”C#.net”,解决办法是将多值属性放入一个新表，则第四范式化后的关系图如下：



而对于skill表则可能的值为:



总结：上面对于数据库范式进行分解的过程中不难看出，应用的范式登记越高，则表越多。表多会带来很多问题：

1 查询时要连接多个表，增加了查询的复杂度

2 查询时需要连接多个表，降低了数据库查询性能

而现在的情况，磁盘空间成本基本可以忽略不计，所以数据冗余所造成的问题也并不是应用数据库范式的理由。因此，并不是应用的范式越高越好，要看实际情况而定。第三范式已经很大程度上减少了数据冗余，并且减少了造成插入异常，更新异常，和删除异常了。我个人观点认为，大多数情况应用到第三范式已经足够，在一定情况下第二范式也是可以的。