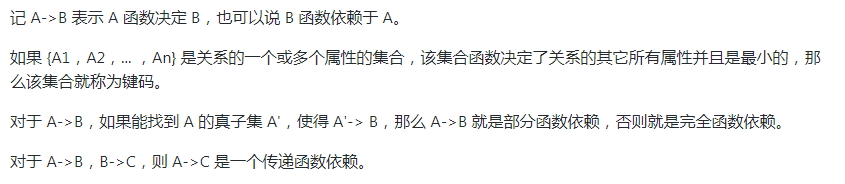
**关系数据库设计理论**

**函数依赖：**

数据依赖是一个关系内部属性与属性之间的一种约束关系。通过属性间值的相等与否体现出来的数据间的相关联系。  
数据依赖包含：函数依赖(FD)和多值依赖(MVD)。

例如：一个学生的关系中，可以有学号，姓名。由于一个学号对应一个学生。因此当学号值确定之后，学生的姓名个系都被唯一确定了。属性间的依赖关系类似于数学的函数f(x)=y；自变量x确定，y就确定了。



**异常：**

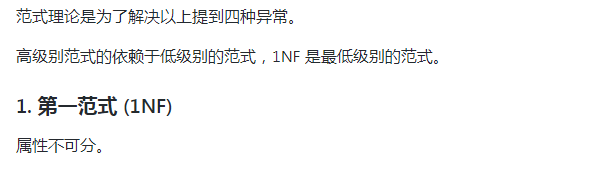
函数依赖的关系模式下存在的异常问题。

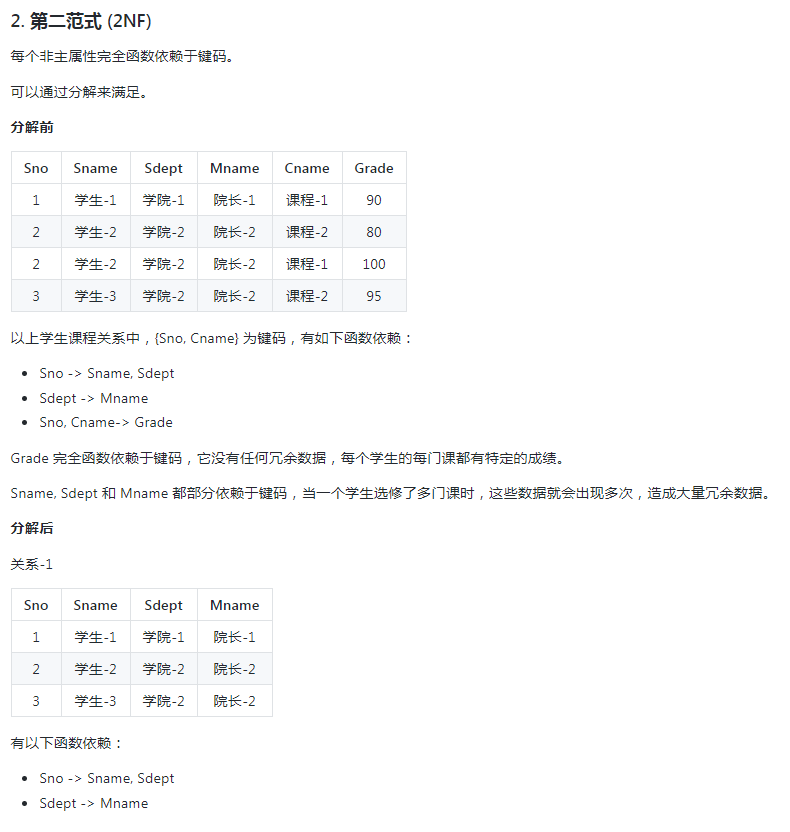


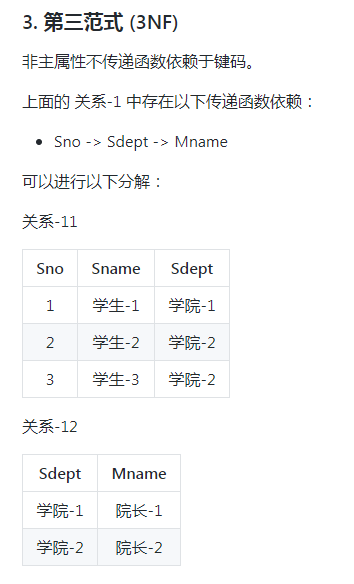
**范式：**

一个低级别的范式的关系模式可以通过模式分解转换成若干个高级别的范式。

5NF∈4NF∈BCNF∈3NF∈2NF∈1NF

、





**E-R图：概念模型**

Entity-Relationship，有三个组成部分：实体、属性、联系。用来进行关系型数据库系统的概念设计。



**数据库管理系统**DBS:**：**

由数据库/数据库管理系统,应用程序,数据库管理员组成的存储,管理,处理和维护数据的系统.数据库管理系统和操作系统一样都是计算机的基础软件.主要功能:  
(1)数据定义  
(2)数据组织/存储和管理  
(3)数据操纵功能  
(4)数据库的事务管理和运行管理  
(5)数据库的建立和维护  
(6)其他

**特点:**

(1)数据结构化,数据之间具有联系.

(2)数据的共享性高,冗余度低且易于扩展;

(3)数据独立性高:物理独立型(应用程序和数据存储独立);逻辑独立(应用程序和数据库逻辑独立)

(4)数据由数据库管理系统统一管理和控制.:

**数据库管理系统的控制功能:**

(1)数据的安全性保护;

(2)数据的完整性检查;

(3)并发控制;

(4)数据库恢复.

**数据模型**：对现实世界数据特征的抽象,是数据库的核心和基础

**概念模型**：又称信息模型:按照用户的观点来对数据和信息建模,主要用于数据库设计.

概念:

(1)实体:客观存在可区分的事物

(2)属性:实体具有的某一个特征

(3)码:唯一标识实体的属性集

(4)实体型:用实体名和属性名集合来抽象和刻画同类实体.例如学生(学号,性别等)

(5)实体集:同一个类型的实体的集合;例如:全体学生

(6)联系:通常指不同实体集之间的联系.具有:一对一,一对多和多对一等多种类型.

**数据模型**：

数据模型构成:

(1)数据结构;

(2)数据操作;

(3)数据的完整性约束.

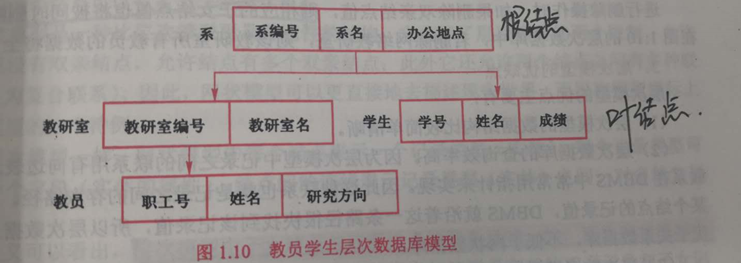
**逻辑模型和物理模型**：物理模型是底层的抽象

逻辑模型:  
(1)层次模型;  
(2)网状模型;  
(3)关系模型  
(4)面向对象数据模型;  
(5)对象关系数据模型  
(6)半结构化数据模型;

**层次模型**：

满足以下两个条件的集合称为层次模型:(树形结构)(1)有且仅有一个结点没有双亲节点,这个节点就是根节点  
(2)根节点以外的其他节点有且仅有一个双亲节点.

层次模型是从父亲节点指向孩子的具有方向的,因此,层次模型数据库系统只能处理一对多的实体联系.任何一个给定的记录值只能按照其层次路径查看,没有一个子女记录值能够脱离双亲记录而存在.一个节点就是一个记录类型,然后一个记录类型包含若干个字段,字段就是描述实体的属性.



进行插入的时候,如果没有相应的双亲节点,就不能插入他的子女节点,就不能插入到该数据库系统中,删除的时候,如果删除了双亲节点,那么子女节点也一并删除了.  
优点:  
(1)数据结构简单清晰  
(2)查询效率高,因为记录之间的关系用有向边表示,用指针实现,性能优于关系数据库,不低于网状数据库.  
(3)提供良好的完整性支持.  
缺点:  
(1)现实中还存在很多非层次的联系  
(2)如果一个节点具有多个双亲,就不行,然后插入和删除具有很多限制  
(3)子女节点必须通过双亲节点查询  
(4)结构严密,区域程序化

**网状模型**：

以下条件:

(1)允许一个以上的节点无双亲

(2)一个节点可以具有多个双亲

优点:

(1)能够很好的描述现实世界

(2)具有良好的性能,存取效率高

缺点:

(1)结构复杂

(2)访问数据的时候需要选择适合的存取路径.加重了编程的负担

**关系模型**：

建立在严格的数学概念基础上.  
关系:一个关系对应一张表(通常意义上的表);  
元祖:表中的一行即为一个元祖  
属性:表中的一列称为属性  
码:表中某个属性组,可以唯一确定一个元祖;  
域:一组具有相同数据类型的值的集合.范围  
分量:元祖中的一个属性值  
关系模式:关系名(属性1,属性2,....)  
关系模式把存取路径对用户隐藏,用户只要指定干什么,不用说明怎么做.  
优点:  
(1)是严格建立在数学概念上的  
(2)关系模型概念单一;数据结构清晰简单  
(3)关系模型存取路径对用户透明,具有高效的数据独立性和安全性.

**数据库模式**：

在数据模型中有"型"和"值"的概念,型是对某一类数据的结构和属性的说明,值是型的一个具体赋值.型是一个通用的模板定义;值是模板赋值后的单个记录.

**模式**:数据库中全体数据的逻辑结构和特征的描述,只涉及型的描述,不涉及值的描述.模式的一个具体值称为模式的一个实例.

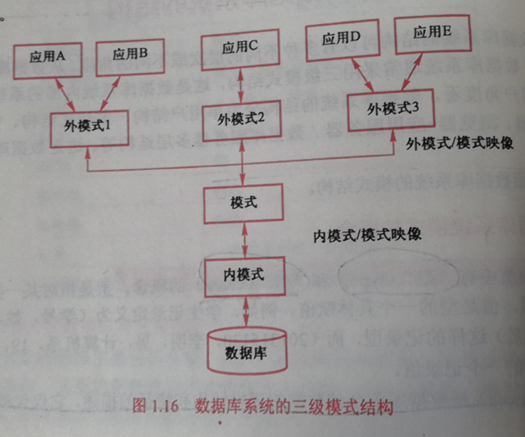
模式是相对稳定的,实例是相对变动的.

**数据库三级模式**：三级模式指的是数据库系统由外模式/模式/内模式三级构成.

(1)模式(schema):逻辑模式,是数据库中全体数据的逻辑结构和特征的米搜狐,是所有用户的公共数据视图.是中间层,既不涉及物理存储,有不涉及具体的应用程序.一个数据库只有一个模式.

(2)外模式:子模式/用户模式:数据库用户能够看见和使用的局部数据的逻辑结构和特征的描述,是数据用户的数据视图,是与某一应用有关的数据的逻辑模式.一个数据库可以有多个外模式.外模式可以保证数据库安全性,用户只能看见和访问对应的外模式中的数据,数据库中其余数据是不可见的.

(3)内模式:存储模式,一个数据库只有一个内模式,是数据物理结构和存储方式的描述,是数据在数据库内部的组织方式.



**两层映像**：

(1)外模式-模式映像  
当模式改变的时候,由数据库管理员对各个模式/外模式的映像做出相应的修改,可以使外模式保持不变,应用程序是依据数据的外模式编写的,从而应用程序不用修改,保证数据和程序的逻辑独立性.数据的逻辑独立性  
(2)模式/内模式映像  
当数据库存储结构发生变化的时候,有管理员对模式/内模式映像进行修改,可以使模式保持不变,从而应用程序不用边改,保证数据与程序的物理独立性.简称:数据物理独立性.

**关系数据结构**：关系数据结构很简单:数据中的逻辑结构就是一张扁平的二维表.

(1)域:具有相同数据类型的值的集合:例如:自然数,实数,长度小于25字节的字符串的集合等,都可以是域.

(2)笛卡尔积:域的集合运算.笛卡尔积可以表示成一张二维表.表中的每行对应一个元祖,表中的每一列的值都来自于一个域.

D1\*D2\*D3\*...\*Dn={(d1\*d2\*d3\*...dn)}.每个元素（d1\*d2\*d3\*...dn)为一个n元祖，简称元祖，元素中每个值叫分量。一个域允许的不同的取值的个数成为这个域的基数。

若Di（i=1...n），其基数为mi（i=1..n）；则想乘的基数就是每个的基数想乘。

例如:下面D1,D2,D3是三个域.

(3)关系：D1\*D2\*D3\*...\*Dn的子集叫做域在D1，D2，D3，...，Dn上的关系，表示为R(D1，D2，D3，...，Dn).R表示关系的名字，n是关系的目或者度。

n=1的时候为单元关系

n=2位二元关系。

关系也是一张二维表，表中的每一行对应一个元祖，每一列对应一个域。由于域可以相同，为了加以区分，对列取名叫做属性。

若关系中某一属性组能唯一的标识一个元祖，而其他子集不能，这称该属性组为候选码。

如果一个关系有多个候选码，则选其中一个作为主码；

候选码的各个属性成为主属性，不包含任何候选码的属性称为非主属性，或者非码属性。

最简单的时候只有一个候选码，全是候选码的称为全码。

一般来讲笛卡尔积很多元祖都是没有意义的，因为有些特定的条件设定。

关系有三种类型：基本关系（基表）实际存在的表，查询表（查询结果的表），视图表（基本表和其他视图导成的表，是虚表）。

**关系的6个性质**：

（1）列是同质的，每一列中分量是同一类型的数据，来自同一个域

（2）不同的列可以出自同一个域，称其中每一列为一个属性，不同的属性要给予不同的属性名。

（3）列的顺序无所谓

（4）任意两个元祖的候选码不能取相同的值

（5）行的顺序也无所谓

（6）分量必须是原子值，不可再分割。



**关系操作**：查询，插入，删除，修改；查询操作：选择，投影，并，差，笛卡尔积

**关系的完整性**：

三类完整约束：

（1）实体完整性；

（2）参照完整性；

（3）用户定义完整性；；

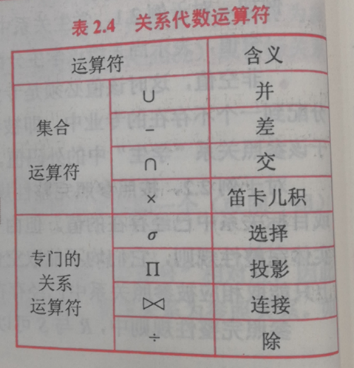
实体完整性和参照完整性是必须满足的，称为：关系的两个不变性。

**实体完整性**：若属性A是关系R的主属性，那么A不能是空值（不知道，不存在或者是无意义）。例如：学生（学号，姓名，班级），学号作为主码的时候，学号不能为空。

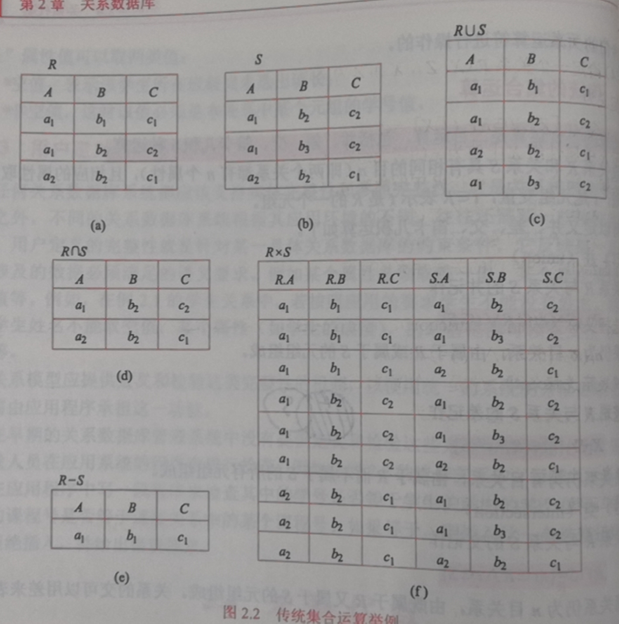
**参照完整性**：设F是关系R的一个或一组属性，但不是关系R的码，Ks是基本关系S的主码。如果F与Ks相对应，则称F是R的外码，并称基本关系R为参照关系，基本关系S为被参照关系，或者目标关系。

**关系代数**：对关系的运算来表达查询。

（1）关系运算符：

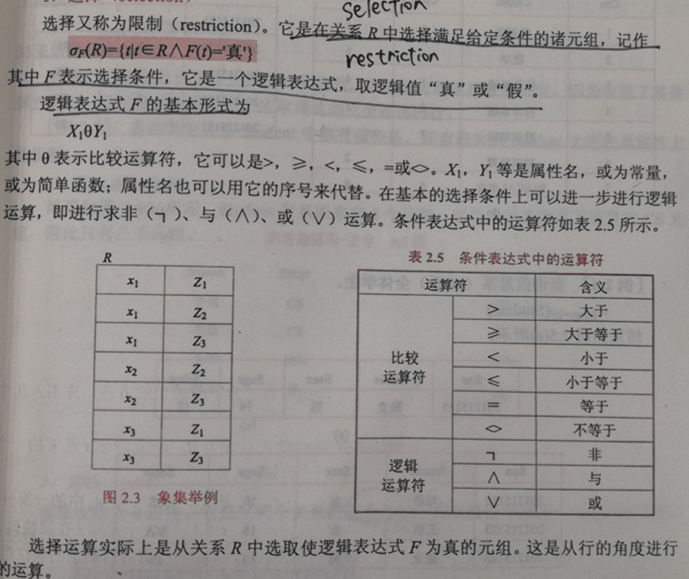


（2）集合运算符：

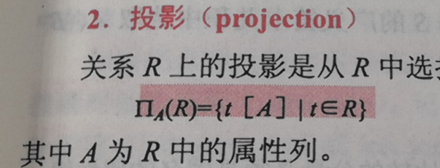


（3）专门的关系运算：选择，投影，连接，除；

1）选择（selection）又称为限制：选择满足某些条件的元祖



2）投影（projection）：是从R中选择若干个属性列组成新的关系

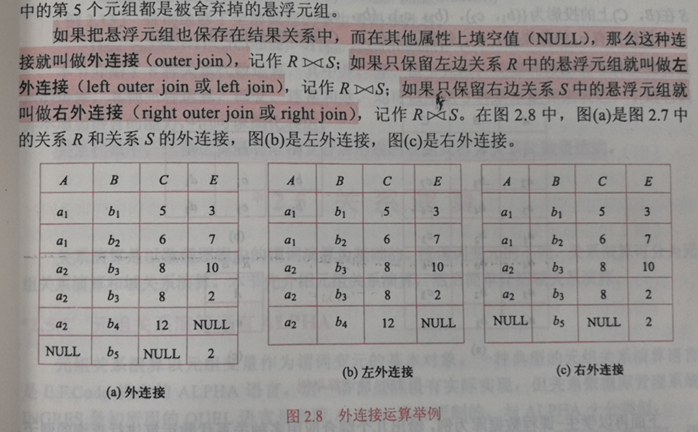


3）连接（join）：从两个关系的笛卡尔积中选取属性间满足一定条件的元祖。

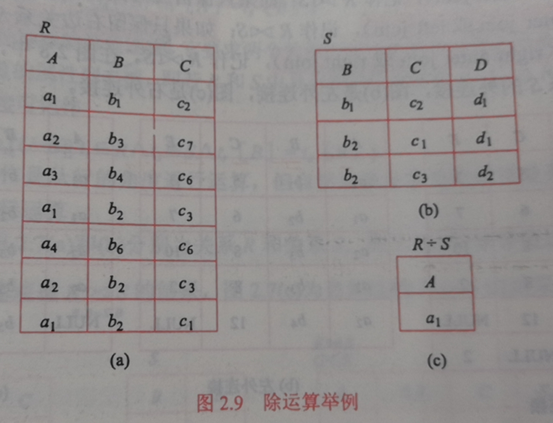


Θ为“=”的连接运算称为等值连接，是从关系R与S 的笛卡尔积中选取A,B属性值相等的那些元祖。

自然连接：是一种特殊的等值连接。要求两个关系中进行比较的分量必须是同名的属性组，并且在结果中把重复的属性列去掉。被舍弃的元祖称为悬浮元祖，



4）除运算（division）：设关系R除以关系S的结果为关系T，则T包含所有在R但不在S中的属性及其值，且T的元祖与S的元祖的所有组合都在R中。



**数据库标准语言SQL**：

SQL(结构化查询语言);

**四大功能:**

(1)定义语言:DDL

(2)查询语言:QL;

(3)操纵语言:DML

(4)控制语言:DCL;

**SQL特点:**

(1)综合统一;

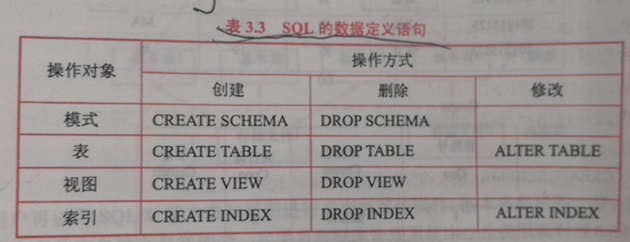
(2)高度非过程化;

(3)面向集合的操作方式;

(4)以同一种语法结构提供多种使用方式;

(5)语言简洁,易学易用.

**数据的定义**：包含了：模式定义，表定义，视图，索引定义。



不提供修改模式定义和修改视图定义的操作。用户如果想要修改这些对象，智能先将其删除再重建。SQL也没有提供索引的相关语句。一个数据库中可以建立多个模式，一个模式下通常包含多个表，视图和索引等数据库对象。

**模式的定义与删除**：

（1）定义模式：

CREATE SCHEMA<模式名> AUTHORIZATION<用户名>;

如果没有指定模式名，则<模式名>隐含为<用户名>。

创建模式，必须是在管理员的权限。

CREATE SCHEMA<模式名> AUTHORIZATION<用户名>[<表定义子句>|<视图定义子句>|<授权定义子句>]

（2）删除模式：

DROP SCHEMA<模式名><CASCADE|RESTRICT>；

CASCADE|RESTRICT两者必选其一。

CASCADE（级联）表示删除模式的同时把该模式中所有的数据库对象全部删除；

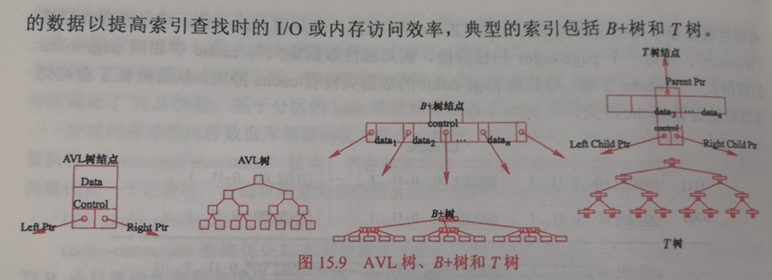
RESTRICT（限制）：表示如果该模式中已经定义了下属的数据库对象（表，视图等），则拒绝该删除语句的执行，只有该模式下没有任何下属对象的时候才删除。

（3）定义表：

CREATE TABLE<表名> （<列名><数据类型>[列完整性约束条件]）；

真正写的时候[] <>括号不用写，标点符号需要写

**数据库的索引**：数据库的索引包括B+树和T树。



B+树是一种动态平衡的多路查找树，B+树节点中存储的数据较多，存储效率高，更新性能较好；B+树层次较少，叶节点存储实际数据，从根节点到任意一个叶节点具有相同路径长度。  
 T树是一种节点中包含多个码值的平衡二叉树，T树具有AVL树的查找性能，又具有与B+树相近的存储效率和更新性能。但是相对于B+树多路查找，T树的二叉树结构增加了树的高度，使得查找次数增加。  
· 在无并发访问的情况下，T树的性能优于B+树，主要原因是在不用加锁解锁的情况下，查找主要的开销是码值的比较。B+树在查找目标码值的过程中需要在每个节点内二分查找；而T树除了最后一个节点需要二分查找，其他节点只需要比较最大项和最小项。  
在有并发控制的时候，由于T树比B+树高，所以更新设计的节点多，需要加锁的节点也多。

**索引的建立与删除**：

表的数据量较大的时候，查询操作比较耗时。建立索引加快查询速度。

常见的索引：

顺序文件上的索引：按照指定的属性值升序或者降序存储的关系，在该属性上建立一个顺序索引文件，索引文件由属性值和相应的元祖指针组成。

B+树索引：将索引组织成B+树的形式，B+树的叶子节点为属性值和相应的元祖指针。B+索引具有动态平衡的优点。

散列索引：建立若干个桶，将索引属性按照其散列函数值映射到相应桶中，桶中存放索引属性值和相应的指针。散列索引查找速度快，不适合范围查找

位图索引：用位向量记录索引属性中可能出现的值，每个位向量对应一个可能值。

**索引的缺点：**占用一定存储空间，表更新的时候，需要维护索引；增加了数据库的负担。

（1）建立索引：

CREATE [UNIQUE][CLUSTER] INDEX<索引名> ON<表名>(<列名>[<次序>][,<列名>[<次序>]]...);

(2)修改索引：

ALTER INDEX <旧索引名> RENAME TO<新索引名>

(3)删除索引

DROP INDEX <索引名>;

真正写的时候[] <>括号不用写，标点符号需要写

**数据查询**：

SQL提供了SELECT函数(不区分大小写)

SELECT [ALL|DISTINCT] <目标列表达式> [,<目标列表达式>]...

FROM<表名或视图名> [,<表名或视图名>]|(<select语句>) [AS] <别名>

[WHERE<条件表达式>]

[GROUP BY<列名1> [HAVING<条件表达式>]]

[GROUP BY<列名2> [ASC|DESC]];

含义如下：根据WHERE子句的条件表达式从FORM子句中指定的基本表和基本图或者派生表中找到满足条件的元祖，再按照SELECT子句中的目标列表表达式选出元祖中属性值形成的结果表。

如果含有GROUP BY子句，将结果按照<列名1> 的值进行分组，该属性列值相等的元祖为一个组。通常会在每组中作用聚集函数。如果GROUP BY子句中带有HAVING短语，则只有满足指定条件的组才输出。

如果还有GROUP BY，则还要将结果按照<列名2> 的值的升序降序排列。

**单表查询**：

(1)**选择表中的若干列**

1)查询指定列:通过SELECT子句的<目标列表达式>中指定要查询的属性列.

SELECT number name

FROM Student

2)查询全部列

方法一:在SELECT关键字后列出所有列名

方法二:如果列的显示顺序与其基表中的顺序相同,可以将<目标列表达式>指定为"\*"。

3)查询经过计算的值

SELECT子句<目标列的表达式>不仅可以是表中的属性列，也可以是表达式

**(2)选择表中的若干元祖**

1)消除取值重复的行;DISTINCT

2)查询满足条件的元祖:

1.确定范围:BETWEEN...AND..;查找某个范围的元祖

SELECT DISTINCT SNO;

FROM SC;

WHERE GRADE<60;

2.确定集合:IN可以用来查找属性值属于指定集合的元祖;

SELECT DISTINCT SNO;

FROM SC;

WHERE SdeptIN('CS','MA','IS');

3.字符匹配:LIKE

[NOT] LIKE ‘<匹配串>’ [ESCAPE'<换码字符>']

%:代表任意长度的字符串a%b表示以a开始的以b结尾的所有字符串

\_:代表任意的单个字符;

SELECT \*;

FROM student;

WHERE Sno LIKE ‘刘%’;

如果要查询的字符串包括了%和\_就需要使用ESCAPE'\'进行通配符进行转义。

WHERE Sno LIKE ‘DB\\_design’ ESCAPE'\';

4.涉及空值的查询IS IS NOT

WHERE Grade IS NULL;//未参加考试的人

WHERE Grade IS NOT NULL;

5.多重条件查询

逻辑运算符AND和OR可以用来连接多个查询条件，AND的优先级高于OR，可以用括号改变优先级

**（3）ORDER BY子句**：

用户用该语句对查询结果按照某一个或者多个属性列的升序(asc)或者降序(DESC)排列。默认为升序。

对于空值，排序时显示的次序由具体系统实现决定。

**（4）聚集函数**：指定了DISTINCT，表示消除重复值；指定ALL表示不消除重复值，默认是ALL；

COUNT(\*) 统计元祖的个数

COUNT([DISTINCT|ALL]<列名>)统计一个列中值的个数

SUM([DISTINCT|ALL]<列名>)计算一个列值的总和（必须是数值型）

AVG([DISTINCT|ALL]<列名>)计算一列的平均值（必须是数值型）

MAX([DISTINCT|ALL]<列名>)求一列中的最大值

MIN([DISTINCT|ALL]<列名>)求一列中的最小值

SELECT COUNT(\*)//不带分号

FROM student;

**(5)GROUP BY子句**

将查询结果按照某一列或者多列的值进行分组，值相等的作为一组

分组后聚集函数将作用于每一个分组，表示每一个组都有一个函数值

例如：求各个课程号及选课人数

SELECT Cno.COUNT(Sno)

FROM SC

GROUP BY Cno;//以分号结尾

WHERE子句不能用作聚集函数作为条件表达式！！

**连接查询**：同时查询涉及两个以上的表，称之为连接查询

（1）等值于非等值连接

连接查询的WHERE子句用来连接两个表的条件称为连接条件和连接谓词，其一般格式为：

[<表名1>.] <列名1> <比较运算符> [<表名2>.] <列名2>

比较运算符主要有=,>,<,<=,>=,!=等

• 此外，还可以是：

[<表名1>.] <列名1> BETWEEN <比较运算符> [<表名2>.] <列名2> AND [<表名3>.] <列名3>

当连接符为=的时候为等值连接，使用其他比较符的时候为非等值连接。

连接谓词中的列名是连接字段，各个连接字段必须是可比的。

一条SQL语句可以同时完成选择和连接查询，这时WHERE子句是由连接谓语和选择谓词组成的复合条件。

（2）自身连接：一个表与自身连接

需要为这个表取一个别名，

（3）外连接：仍然把悬浮数组保存在结果关系中。

（4）多表连接：

**嵌套查询**：

在SQL中，一个SELECT-FROM-WHERE语句称为一个查询快。  
将一个查询块嵌入另外一个查询块的WHERE和HAVING中的查询称之为嵌套查询。  
SELECT SName  
FROM Student  
WHERE Sno IN(SELECT Sno  
FROM SC  
WHERE Cno='2');  
外面的查询叫做外层查询或者父查询，内层的查询块叫做内层查询或者子查询  
1）带有IN谓词的子查询：子查询的结果往往是一个集合，IN是嵌套查询中常用的谓词。  
子查询的条件不依赖于父查询，称为**不相关子查询**。依赖的时候，称之为**相关子查询**或者相关嵌套查询。

**集合查询**：

SELECT语句查询的结果是元祖的集合，所以多个SELECT语句结果可以进行集合操作。集合操作主要包括并操作UNION,交操作INTERSECT和差操作EXCEPT。

**数据更新**：

**插入数据**：

两种：插入一个元祖，插入子查询的结果（一次插入多个元祖）

（1）插入元祖

INSERT

INTO<表名> [(<属性列1>[,<属性列2>]....)]

VALUES(<常量1>[,<常量2>]...);

属性1的值为常量1....；INTO子句中没有出现的属性列，新元祖再这些列将取空值；需要注意在表定义的时候NOT NULL的属性列不能取NULL。

如果INTO子句中没有指明任何属性列名,则新插入的元祖必须在每个属性列都有值。

字符串常数用单引号。

（2）插入子查询的结果

INSERT

INTO<表名> [(<属性列1>[,<属性列2>]....)]

子查询；

**修改数据**：

UPDATE <表名>

SET<列名>=<表达式> [,<列名>=<表达式>]...

[WHERE <条件>];

(1)修改一个元祖的值

UPDATE Student

SET age=22

WHERE Sno='201215121';

（2）修改多个元祖

UPDATE Student

SET age=age+1;

(3)带子查询的修改:将成绩全部改为0；

UPDATE SC

SET Grade=0;

WHERE Sno IN

(SELECT Sno

FROM Student

WHERE Sdept='CS');

**删除数据**：

DELETE

FROM<表名>

[WHERE <条件>];

delete删除的是表中的数据,表的定义并没有被删除

（1）删除一个元祖的值,删除学号为201215128的记录

DELETE

FROM Student

WHERE Sno=‘201215128’;

（2）删除多个元祖的值：使得SC为空表

DELETE

FROM SC；

（3）带子查询的删除语句:删除所有学生的选课记录

DELETE

FORM SC;

WHERE Sno IN

(SELECT Sno

FROM Student

WHERE Sdept='CS');

**视图**：

视图是从一个表或几个基本表导出的表。是一个虚表。数据库中只存放视图的定义，而不存放视图对应的数据。这些数据还是存在基本表中。所以一旦基本表变化了，视图中查询的数据也变化了。

作用：

（1）简化用户的操作

视图将表与表的连接对用户隐藏，用户只是对一个虚表进行简单的查询。

（2）视图能够使用户多角度看待同一个数据

（3）视图对重构数据库提供了一定程度的逻辑独立性

（4）视图能过对机密数据提供安全保护

（5）表达清晰

**建立视图**：

CREATE VIEW建立视图

CREATE VIEW <视图名> [<列名1>[,<列名2>]]

AS<子查询>

[WITH CHECK OPTION];

子查询是任意的SELECT语句，

WITH CHECK OPTION表示对视图的进行更新，插入，删除操作时要进行更新。

删除视图

DROP VIEW<视图名> [CASCADE]

从数据字典中删除，级联的一起删除。

**视图查询**：

（1）视图消解：将视图对应成基本表，对基本表进行查询  
从数据字典中取出视图的定义，把定义中的子查询和用户查询结合起来，转换成等价的对基本表的查询，然后执行修正了的查询。

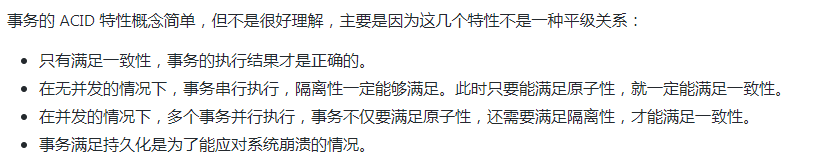
**更新视图**：更新视图最终还是对基本表的更新，和查询一样，对视图的更新也是通过视图消解，转换成对基本表的更新。

**数据库事务**：事务是用户定义的一个数据库操作序列，要么全做，要么不做，是一个不可分割的工作单位。以BEGIN TRANSACTION开始；以 COMMIT 和 ROLLBACK结束

事务是满足**ACID**特性的一组操作,可以通过commit提交一个事务,也可以使用rollback回滚一个事务

ACID:





**AUTOCOMMIT**:mysql采用自动提交的模式。如果不显示使用START TRANSACTION语句开始一个事务,那么每个查询操作都会被当做一个事务自动提交。

**数据库故障**：

（1）事务内部的故障

例如：A转账给B,若果A的账户余额不足，就回滚。这还是程序能够处理的。

事务内部的故障大多是非预期的，是不能由应用程序处理的。

产生这种故障可以强行回滚(UNDO)就像没发生一样

（2）系统故障（软故障）：系统停止运行的任何事件，系统需要重启；

系统重启之后，恢复子系统除了需要撤销未完成的事务之外，还需要重做已经提交了的事务，保证一致性

（3）介质故障（硬故障）

（4）计算机病毒

各种故障造成的影响：一是数据库本身被破坏；二是数据库没有被破坏但是数据不正确。

数据库的恢复：冗余

**数据库恢复技术**：

建立冗余数据：数据转存储和登记日志文件

（1）数据转储

后备副本的概念。

当数据库破坏之后，通过后备副本的重装，重装只能恢复到转储时的状态。

静态转储：转储期间不允许对数据库进行存取，修改，保证一致性。必须等到用户事务结束才进行。

动态转储：转储期间允许更新数据库，但是不能保证数据的正确有效，为此必须把转储期间各个事务对数据的修改活动登记下来，建立日志文件。后援副本就能根据日志将数据正确恢复。

2）转储还可以分为海量和增量：

海量转储是每次转储全部数据库；恢复方便一点

增量转储是每次只转储上一次更新的数据：如果数据库大，事务复杂，该方法方便

3）数据转储的方法：

1动态海量转储

2动态增量转储

3静态海量转储

4静态增量转储

（2）登记日志文件

1）日志文件是用来记录事务对数据库的更新操作的文件。

2）日志文件中的内容：记录单位为日志文件

1.各个事务的开始标记

2.各个事务结束的标记

3，各个事务的所有更新操作

3）每个日志记录的内容：一个日志记录单位

1.事务标识

2.操作的类型

3.操作对象

4.更新前数据的旧值

5.更新后数据的新值

作用：

（1）事务故障恢复和系统故障恢复必须使用日志文件

（2）在动态转储必须建立日志文件，后备文件必须结合日志才能恢复数据

（3）在静态转储可以建立日志文件

**数据库故障恢复**：

**事务故障的恢复**：就是利用日志文件回滚，恢复步骤如下：

（1）反向扫描日志文件，查找该事务的更新操作

（2）对该事务的更新操作执行逆操作，即将日志记录中“更新前的值”写入数据库。如果记录中是插入操作，（因为此时更新前的值为NULL）则相当于做删除操作；若记录是删除操作，则做插入操作；若是修改操作，则用修改前的值代替修改后的值。

（3）继续反向扫描日志文件，查找该事务的其他更新操作，做同样的处理

（4）直到读到该事务开始的标志，事务就恢复完成了

**系统故障恢复**：

一是未完成事务对数据库的更新可能已经写入数据库；二是已经提交的事务对数据库的更新可能还留在缓冲区没来得及写入数据库。因此恢复操作就是要撤销故障发生时未完成的事务，重做已经完成的事务。

（1）正向扫描日志文件，从头开始扫描，找到故障发生前已经提交的事务(有COMMIT记录)，将事务标记入重做对列。同时找出故障发生时上位完成的事务(只有BEGIN TRANSACTION，没有COMMIT记录),将其事务标识记录撤销对列

（2）将撤销对列中的各个事务进行撤销处理；

（3）对重做对列中的各个事务进行重做处理；

**介质故障恢复**

重装数据库，然后重做已完成的事务

（1）装入最新的数据库后备副本，使得数据库恢复到最近一次转储时的一致性状态

（2）装入相应的日志文件副本，重做该事务。

**封锁**：

封锁是实现并发控制的一个非常重要的技术：封锁就是事务T对某个数据对象操作之前，先向系统发出请求，对其加锁。加锁之后事务T就对该数据对象有了一定的控制。在事务T释放他的锁之前，其他事物不能更新此数据对象。

**封锁粒度:**

mysql中有两种:行级锁以及表级锁

应该尽量只锁定需要修改的那部分数据,而不是所有资源.锁定的数据量越少,发生锁争用的可能越小,系统的并发程度越高.

加锁需要资源消耗,锁的各种操作都会增加系统开销.因此封锁粒度越小,系统开销越大

在选择封锁粒度的时候,需要在锁开销和并发程度之间权衡

**并发一致性问题**：在并发的环境下,事务的隔离性很难保证,会导致并发一致性问题:

(1)丢失修改:

指的是一个事务的更新操作被另外一个事务的更新操作替换.一般生活中常遇到:例如A,B两个事务都对一个数据修改,A先提交生效,B之后修改,B的修改覆盖了A的修改

(2)脏读:当前事务可以读取到另外事务未提交的数据,因为未提交,如果不提交或是撤销了,那么就读到了无效数据

(3)不可重复读:一个事务内多次读同一个数据集合.在这个事务还没有结束前,另外一个事务也访问该同一个数据集合并做了修改,由于第二个事务的修改,第一次事务的两次读取数据不一致.第二次读之前其他事物进行了修改。

(4)幻读:其实也属于不可重复读的情况.第二次读之前其他的事务进行增删操作。

产生并发不一致性问题的主要原因是破坏了事务的隔离性,解决办法是通过并发控制来保证隔离性.并发控制可以通过封锁实现,但是封锁需要用户自己控制,相当复杂.数据库管理系统提供了事务的隔离级别,让用户以一种更轻松的方式处理并发一致性问题.

• 悲观锁

正如其名，它指的是对数据被外界（包括本系统当前的其他事务，以及来自外部系统的事务处理）修改持保守态度，因此，在整个数据处理过程中，将数据处 于锁定状态。悲观锁的实现，往往依靠数据库提供的锁机制（也只有数据库层提供的锁机制才能真正保证数据访问的排他性，否则，即使在本系统中实现了加锁机 制，也无法保证外部系统不会修改数据）。

在悲观锁的情况下，为了保证事务的隔离性，就需要一致性锁定读。读取数据时给加锁，其它事务无法修改这些数据。修改删除数据时也要加锁，其它事务无法读取这些数据。

• 乐观锁

相对悲观锁而言，乐观锁机制采取了更加宽松的加锁机制。悲观锁大多数情况下依靠数据库的锁机制实现，以保证操作最大程度的独占性。但随之而来的就是数据库性能的大量开销，特别是对长事务而言，这样的开销往往无法承受。

而乐观锁机制在一定程度上解决了这个问题。乐观锁，大多是基于数据版本（ Version ）记录机制实现。何谓数据版本？即为数据增加一个版本标识，在基于数据库表的版本解决方案中，一般是通过为数据库表增加一个 “version” 字段来实现。读取出数据时，将此版本号一同读出，之后更新时，对此版本号加一。此时，将提交数据的版本数据与数据库表对应记录的当前版本信息进行比对，如 果提交的数据版本号大于数据库表当前版本号，则予以更新，否则认为是过期数据。

**隔离级别**：



**封锁的类型**：

(1)读写锁:

1)互斥锁（排他锁）:简写为x锁,又称为写锁

2)共享锁:简称为S锁,又称为读锁

规定:加什么锁表示要进行什么操作!!!

1)一个事务对数据对象A加了X锁,这个事务就可以对A进行读取和更新.加锁期间其他事务不能对A加任何锁.

2)一个事务对数据对象A加了S锁,可以对S进行读取操作,但是不能进行更新操作,加锁期间其他事物能对A加S锁,但是不能加X锁.

读-读能共存

读写不能共存

写写不能共存.

**封锁的协议**：

在运用X锁和S锁这两种基本的锁对数据对象加锁的时候，需要遵循一些规则，申请，持续时间，何时释放等规则，这些规则称为封锁协议。

（1）一级封锁协议：

事务T在修改数据R之前必须对其加X锁，直到事务T结束才释放。事务结束包括正常结束（commit）和非正常结束（rollback）。一级封锁可以解决丢失修改的问题，并且保证事务T是可恢复的。一级封锁中，如果仅仅是读数据是不对其修改的，是不需要加锁的，所以不能避免不可重复读，幻读和脏读。

（2）二级封锁协议：

在一级封锁的基础上增加了事务T在读取数据R之前必须先对其加上S锁，读完之后即可释放S锁（不一定是事务结束）。

二级协议除了可以防止丢失修改之外，还可以避免脏读。由于读完之后立即释放了S锁，但是不能保证可重复读。

（3）在一级封锁的基础上增加了事务T在读取数据R之前必须先对其加上S锁，直到事务结束释放S锁。可以避免丢失修改，脏读和不可重复读。

**活锁和死锁**：

（1）活锁

如果事务T1封锁了数据R,事务T2这个时候请求封锁R，自然进入等待；整个时候T3又请求封锁R,当事务T1结束之后，系统首先批准T3封锁数据R，如果这个时候T4又来了；T3结束之后T4又封锁了R。这样T2可能就永远的等待。这就是所谓的活锁。

避免活锁最直接的办法就是先来先服务的策略。

（2）死锁

如果T1封锁了数据R1，T2封锁了数据R2；然后T1又请求封锁R2，T2又请求封锁R1。这个时候互相进入等待的状态，都在等待对方释放锁。进入死锁状态。

两类解决方式：一是预防；二是允许发生，发生之后解除

1）死锁的预防

1.一次封锁法：要求每个事物必须一次将所有要使用的数据全部加锁，否则就不能继续执行。上面例子T1必须先对R1，R2加锁，T1结束之后T2再执行。

扩大了封锁范围，降低了系统的并发度；数据库中数据是不断变化的，原来不要求封锁的数据可能会变成封锁对象，将事务在执行过程中可能封锁的数据对象全部加锁，进一步降低了并发度。

2.顺序封锁法：预先对数据对象规定一个封锁顺序，所有事务按照这个顺序封锁。例如在B树索引中，规定封锁的顺序是从根节点开始，然后是下一集的子节点，逐级封锁。

数据库中封锁的数据对象极多，并且随时会有数据的更新，要维护封锁顺序很难。事务的封锁请求可以随着事务的执行动态决定，很难实现确定一个事务要封锁的对象，很难得到一个顺序。

综上，预先预防的方法不太适合，所以都是诊断解除的方法。

2）死锁的诊断和解除

1.诊断的方法一般采用超时法和等待图法：

超时法：一个事务等待的时间超过了规定的时间，就认为发生了死锁。两个缺点：时间太短容易误判；时间太长不能及时发现死锁

等待图法：是一个有向图；如果图中出现两个事务互相等待就是死锁。

2.解除：选择一个处理死锁代价最小的事务，将其撤销，释放该事务所有的锁，使得其他事务执行。

**并发调度的可串行性**：

（1）可串行化调度

多个事务的并发执行是正确的，当且仅当其结果与按照某一次序串行执行这些事物的结果相同，则就叫可串行化调度。

可串行化调度是并发事务的正确调度准则，按照这个准则规定，一个给定的并发调度，当且仅当他是可串行化的，才认为是正确调度。

例如：

事务T1：读B；A=B+1;写回A;

事务T2：读A:B=A+1;写回B.

假设，初始A=2，B=2;按照T1-T2的顺序的话，A=3，B=4；按照T2->T1的顺序：B=3，A=4。如果这两个事务并发执行，结果和上面两个结果有一个一样就是可串行化调度的，和上面结果都不一样就是不可串行化调度。

（2）判断可串行化调度的充分条件

冲突操作是指不同的事务对同一个数据的读写操作和写写操作；

若一个调度的冲突可串行化，则一定是可串行化调度。冲突可串行化调度是可串行化调度的充分条件，不是必要条件。

**两段封锁协议（可串行化）**：

为了保证并发调度的正确性，采用一定的手段保证调度的可串行化。采用两段锁协议（2PL）协议。  
2PL协议：所有事务必须分两个阶段对数据项加锁和解锁  
（1）对任何数据进行读、写操作之前，首先要申请并且获得对该数据的封锁。  
（2）在释放一个锁之后，事务不再申请和获得其他封锁。  
第一个阶段获得封锁，称为扩展阶段，可以申请得到任何类型的锁，但是不能释放锁；第二个阶段是释放封锁，称为收缩阶段，事务可以释放任何数据的任何锁，但是不能申请任何锁。

**封锁粒度**：

封锁的粒度就是指的封锁对象的大小。

封锁粒度与系统的并发度和并发控制的开销密切相关。

封锁的粒度越大，数据库所能封锁的数据单元越少，并发度越小，系统的开销就小；反之。

多粒度封锁：

首先定义多粒度树；几级粒树就是树的深度是几。

多粒度封锁协议：读一个节点加锁意味着这个节点的所有后裔节点也被加了同样的锁。因此，一个数据对象可能被显示加锁和隐式加锁。

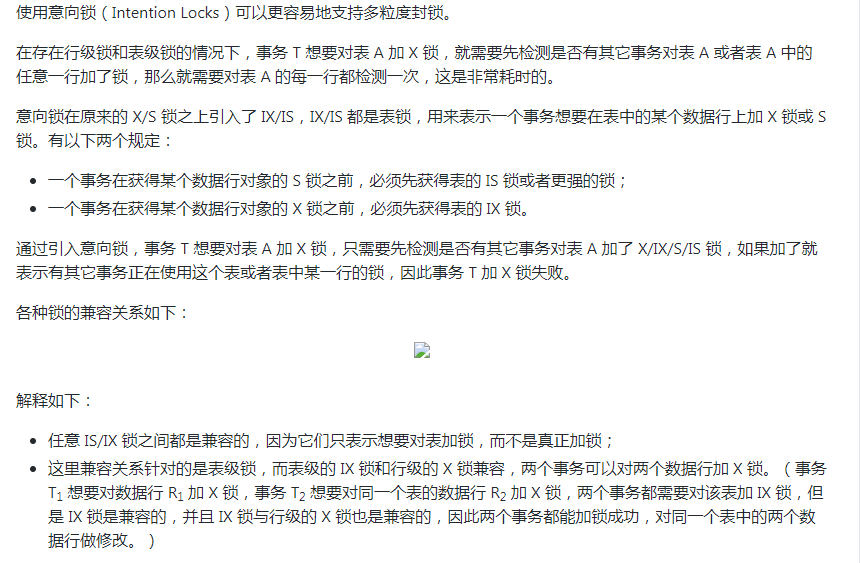
显示加锁：事务直接加在数据对象上的锁

隐式加锁：该数据没有被独立加锁，由于上级加锁而被迫加锁。

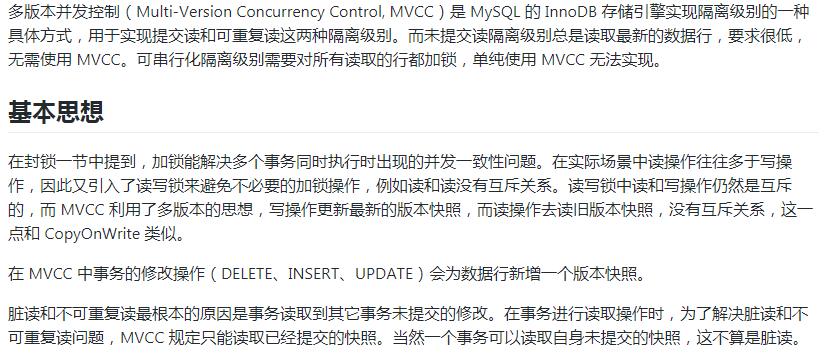
一般对一个数据加锁，系统要检查该数据对象上有无显示封锁与之冲突，再检查其所有上级节点，看本事务的显示封锁是否与该数据对象的隐式封锁冲突，还要检查其所有的下级节点，看他们的显示封锁是否与本数据的隐式 封锁冲突。显然，效率极低。

为此，产生一种新的锁：意向锁：

意向锁：如果对一个节点加意向锁，则说明该节点下层的节点正在被加锁，对任一节点加锁，必须对他的上层节点加上意向锁。



**多版本并发控制**：



MySQL、Oracle 等数据库系统的大多数事务型存储引擎实现的都不是简单的行级锁。 基于提高并发性能的考虑，一般都同时实现了 MVCC。MVCC 是通过在每行记录的后面保 存两个隐藏的列来实现的，这两个列一个保存了行的创建时间，一个保存了行的过期时间 （删除时间）。当然存储的不是实际的时间值，而是系统的版本号。每开始一个新的事务，系统版本号都会自动递增。事务开始时刻的系统版本号会作为事务的版本号，用来和查询到 的每行记录的版本号进行比较。 保存额外的系统版本号，使大多数读操作都可以不用加锁，MVCC 使得读取数据操作 很简单，性能好，而且保证只会读取到符合标准的行，不足之处是每行都需要额外的存储空 间和更多的检查工作，有一些额外的维护工作。

时间戳方法是给每一个事务盖上一个时标，按照这个时间戳来解决事务的冲突操作。

版本指的是数据库中数据对象的一个快照，记录数据对象某个时刻的状态。

主要是随着计算机存储设备的价格降低，可以考虑为数据库系统的数据对象保留多个版本，提高系统并发操作程度。

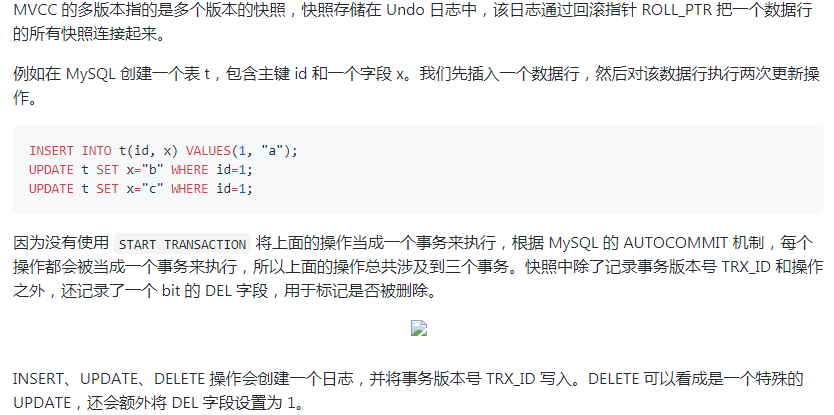
例如：有一个数据对象A有两个事务，事务T1是写事务，事务T2是读事务。假定先启动T1事务，后启动T2事务。按照传统的封锁协议，T2必须等待T1执行完，然后才能读，实际上还是串行执行的。如果在T1准备写A时不是等待，而是为A生成一个新的版本，那么T2可以在新版本上进行操作。只是在T2准备提交的时候要检查一下T1是否已经完成，如果T1已经完成，T2就可以放心提交；如果T1没有完成，那么T2必须等待T1完成，保证事务的可串行化，又提高了事务的执行并行度。

在多版本机制中，每一个write(q)操作都创建一个q的新版本，每一个版本qi用于版本的值和创建该版本的事务的时间戳和成功读取qi事务的最大时间戳。创建该版本的事务的时间戳表示在数据项q上成功执行write（q）操作的所有事务中最大时间戳；成功读取事务的最大时间戳表示数据项q上成功执行read（q）操作的所有事务中最大的时间戳。

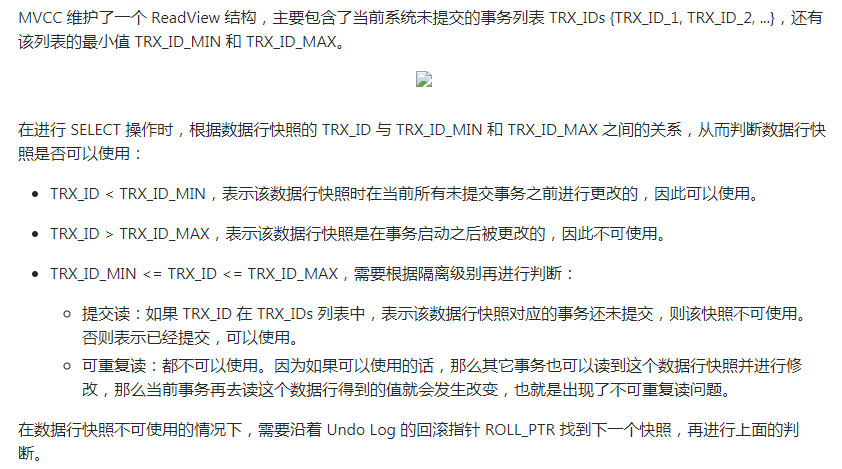
多版本并发利用物理存储上的多版本维护数据的一致性，意味着当检索数据库的时候，每个事务都看到一个数据的一段时间的快照，而不管正在处理的数据的当前的状态。

多版本并发控制和封锁机制相比，主要的好处是消除了数据库中数据对象读和写操作的冲突，有效的提高了系统性能。但是也会产生大量无效的版本，而且在事务结束时刻，其所影响的元组有效性不能马上确定，就为保存事务执行过程中的状态提出了难题。

**Undo日志**：



**Readview**：



**快照读和当前读**：



**NEXT-KEY LOCKS**：



**改进型多版本并发控制：**

区分事务类型为只读事务和更新事务。对于只读事务发生冲突的可能性很小，可以采用多版本时间戳。对于更新事务，采用保守的两阶段封锁机制。混合MV2PL协议。

除了传统的读锁和写锁，引进一个新的封锁类型，称为验证锁(c锁)。

**数据存储**：

数据库的数据存储：  
（1）行存储模型，列存储模型，混合模型  
 1.行存储模型：元祖是连续存放的，适合事务处理中一次更新多个属性的操作，能够保证对多个属性的操作产生最小的内存访问；但是对于只涉及表中相对较少属性的分析处理时，即使该查询仅仅涉及元祖的某个或者某些属性，其他属性也会被同时读入到缓存中，降低了缓存的利用率。  
 2.列存储模型：将关系进行垂直划分，相同属性的数据连续存储。当访问特定属性时，只读入所需要的属性所在的分片，所以节省内存带宽，并且具有较高的数据访问的局部性。并且具有较高的数据访问局部性，可以减少缓存失效，提高数据访问效率；同时列存储将相同类型的数据集中存储，能够更好的对数据进行压缩减少内存宽带消耗。但是，如果查询所需要的属性过多，列存储需要连接多个划分来满足查询需求，导致性能下降，特别是元祖重构的时候需要较多的连接操作，代价较高。  
3.混合数据存储：将同一元祖所有的属性值存储在一页内，在页内对元祖进行垂直划分。由于元组在页内进行垂直划分，所以模型具有较好的数据空间局部性，可以优化缓存性能，由于同一元组的值存储在同一页内，所以元组的重构代价少。  
  
 关系数据库系统利用缓存区缓存数据，当数据存取层需要读取数据的时候，数据存储子系统首先到系统缓存区中查找。只有当缓冲区中不存在该数据时才真正的从外存中读入该数据所在的页面。当数据存取层写回一元组到数据库中时，存储子系统并不把他立即写回外存，仅把该元组所在的缓冲区页面作为一个标志，表示可以释放。只有当该用户事务结束或缓冲区满了，才按一定的淘汰策略进行释放。减少页面的交换次数，提高存取效率。  
缓冲区主要的管理算法：淘汰算法，查找算法  
FIFO LRU；  
顺序查找，折半查找，hash查找。

**数据库的存储过程**：

SQL 语句需要先编译然后执行，而存储过程 Stored Procedure 是一组为了完成特点功能的 SQL 语句集，经编译后存储在数据库中，用户通过制定存储过程的名字并给定参数（如果该存储过程带有参数）来调用执行它。

存储过程是可编程的函数，在数据库中创建并保存，可以由 SQL 语句和控制结构组成。当想要在不同的应用程序或平台上执行相同的函数或者封装特定的功能时，存储过程非常有用。数据库中的存储过程可以看做是对编程中面向对象方法的模拟，允许控制数据的访问。

**存储过程的优点**：

1、 增强 SQL 语言的功能和灵活性：存储过程可以用控制语句编写，有很强的灵活性，

可以完成复杂的判断和较复杂的运算。

2、 标准组件式编程：存储过程被创建后，可以在程序中多次调用，不需要重新编写该

存储过程的 SQL 语句。数据库维护人员修改存储过程的内容并不会影响服务器的

程序源码。

3、 较快的执行速度：如果某一操作包含大量的 Transaction-SQL 代码或分别被多次执

行，那么存储过程要比批处理的执行速度快。因为存储过程是预编译的，在首次运

行一个存储过程时查询，优化器对其进行分析优化，并且给出最终被存储在系统表

中的执行计划。而批处理的语句每次执行都要重新进行编译和优化，速度较慢。

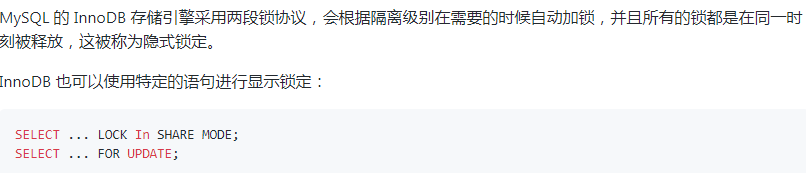
4、 减少网络流量：针对同一个数据库对象的操作，存储过程只需要传输调用语句名，

存储过程的逻辑代码在数据库中，传输数据量小。

5、 安全机制：通过对执行某一存储过程的权限进行限制，能够实现对相应的数据的访

问权限的限制，避免了非授权用户对数据的访问，保证了数据的安全性。

**Mysql隐式和显示锁定**：



**数据库通信加密和HTTPS一样**。

**Varchar与char的区别**：

Varchar:不定长，效率低

char：定长，效率高

**数据库的三大范式**：

第一范式：当关系模式R的所有属性都不能再分解为更基本的数据单位时，称R是满足第一范式，即属性不可分。

第二范式：如果关系模式R满足第一范式，并且R得所有非主属性都完全依赖于R的每一个候选关键属性，称R满足第二范式

第三范式：设R是一个满足第一范式条件的关系模式，X是R的任意属性集，如果X非传递依赖于R的任意一个候选关键字，称R满足第三范式，即非主属性不传递依赖于键码。

**mysql的端口号，如何修改**：

查看端口号：使用命令show global variables like 'port';查看端口号 ，mysql的默认端口是3306。（补充：sqlserver默认端口号为：1433；oracle默认端口号为：1521；DB2默认端口号为：5000；PostgreSQL默认端口号为：5432）

修改端口号：编辑/etc/my.cnf文件，早期版本有可能是my.conf文件名，增加端口参数，并且设定端口，注意该端口未被使用，保存退出。

**索引的优缺点**：

优点:

(1)通过索引,可以保证数据库表中每一行数据的唯一性

(2)可以加快数据检索速度

(3)可以加速表与表之间的连接

(4)使用分组和排序子句进行数据检索的时候,同样可以显著减少查询和排序的时间

(5)在查询的过程中,优化隐藏起,提高系统性能

缺点:

(1)创建和维护索引需要时间,随着数据量的增加而增加

(2)索引需要占据物理内存

(3)对表中数据进行增删操作的时候,索引也需要动态维护,降低了数据的维护速度

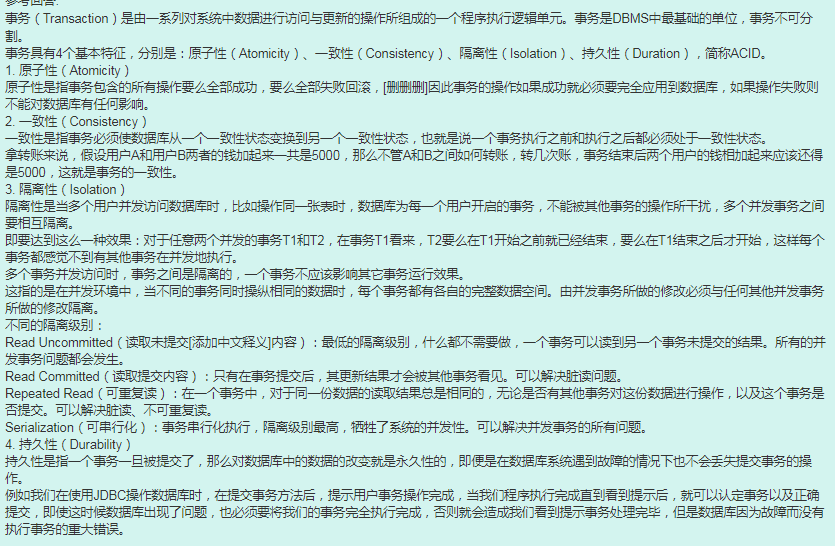
**数据库的索引**：索引是对数据库表中一列或者多列的值进行排序的一种结构,使用索引可以快速访问数据库表中的特定信息。索引的目的就是加快检索表中数据的方法.

**数据库事务**：单个逻辑工作单元执行的一系列操作,要么完全的执行,要么完全的不执行.

事务处理可以确保非事务性单元内的所有操作都成功完成,否则不会永久更新面向数据的资源.通过将一组相关操作组合成一个要么全部成功要么全部失败的单元,可以简化错误恢复并且能使得应用程序更加可靠。

**数据库事务的隔离性**：同一时间,只允许一个事务请求同一个数据,不同事务之间没有干扰.

**事务的基本特征**：



**mysql四种隔离状态**：



**mysql的MVCC机制**：

MVCC是一种多版本并发控制机制，是MySQL的InnoDB存储引擎实现隔离级别的一种具体方式，用于实现提交读和可重复读这两种隔离级别。MVCC是通过保存数据在某个时间点的快照来实现该机制，其在每行记录后面保存两个隐藏的列，分别保存这个行的创建版本号和删除版本号，然后Innodb的MVCC使用到的快照存储在Undo日志中，该日志通过回滚指针把一个数据行所有快照连接起来。

**优化SQL的方法**：

（1）通过索引建立查询进行优化

（2）对查询进行优化，尽量避免全表扫描

**mongodb和redis的区别**：

内存管理机制上：Redis 数据全部存在内存，定期写入磁盘，当内存不够时，可以选择指定的 LRU 算法删除数据。MongoDB 数据存在内存，由 linux系统 mmap 实现，当内存不够时，只将热点数据放入内存，其他数据存在磁盘。支持的数据结构上：Redis 支持的数据结构丰富，包括hash、set、list等。

MongoDB 数据结构比较单一，但是支持丰富的数据表达，索引，最类似关系型数据库，支持的查询语言非常丰富。

**redis机制的实现**：

Redis服务器是一个事件驱动程序，服务器需要处理以下两类事件：文件事件（服务器对套接字操作的抽象）和时间事件（服务器对定时操作的抽象）。Redis的定时机制就是借助时间事件实现的。一个时间事件主要由以下三个属性组成：id：时间事件标识号；when：记录时间事件的到达时间；timeProc：时间事件处理器，当时间事件到达时，服务器就会调用相应的处理器来处理时间。一个时间事件根据时间事件处理器的返回值来判断是定时事件还是周期性事件

一个时间事件主要由以下三个属性组成：id：时间事件标识号；when：记录时间事件的到达时间；timeProc：时间事件处理器，当时间事件到达时，服务器就会调用相应的处理器来处理时间。一个时间事件根据时间事件处理器的返回值来判断是定时事件还是周期性事件。

**redis是单线程的为什么这么高效**：

虽然Redis文件事件处理器以单线程方式运行，但是通过使用I/O多路复用程序来监听多个套接字，文件事件处理器既实现了高性能的网络通信模型，又可以很好地与Redis服务器中其他同样以单线程运行的模块进行对接，这保持了Redis内部单线程设计的简单性。

**redis数据类型，底层如何实现**：

1）字符串：整数值、embstr编码的简单动态字符串、简单动态字符串（SDS)2）列表：压缩列表、双端链表

3）哈希：压缩列表、字典

4）集合：整数集合、字典

5）有序集合：压缩列表、跳跃表和字典

**redis的reshash怎么实现**：

因为redis是单线程，当K很多时，如果一次性将键值对全部rehash，庞大的计算量会影响服务器性能，甚至可能会导致服务器在一段时间内停止服务。不可能一步完成整个rehash操作，所以redis是分多次、渐进式的rehash。渐进性哈希分为两种：1）操作redis时，额外做一步rehash

对redis做读取、插入、删除等操作时，会把位于table[dict->rehashidx]位置的链表移动到新的dictht中，然后把rehashidx做加一操作，移动到后面一个槽位。

2）后台定时任务调用rehash

后台定时任务rehash调用链，同时可以通过server.hz控制rehash调用频率

**redis和memcache的区别**：

1）数据类型 ：redis数据类型丰富，支持set liset等类型；memcache支持简单数据类型，需要客户端自己处理复杂对象2）持久性：redis支持数据落地持久化存储；memcache不支持数据持久存储。)

3）分布式存储：redis支持master-slave复制模式；memcache可以使用一致性hash做分布式。

4）value大小不同：memcache是一个内存缓存，key的长度小于250字符，单个item存储要小于1M，不适合虚拟机使用

5）数据一致性不同：redis使用的是单线程模型，保证了数据按顺序提交；memcache需要使用cas保证数据一致性。CAS（Check and Set）是一个确保并发一致性的机制，属于“乐观锁”范畴；原理很简单：拿版本号，操作，对比版本号，如果一致就操作，不一致就放弃任何操作

6）cpu利用：redis单线程模型只能使用一个cpu，可以开启多个redis进程

**mysql引擎及区别**：

在Mysql数据库中，常用的引擎为Innodb和MyIASM,其中Innodb是一个事务型的存储引擎，有行级锁定和外键约束，提供了对数据库ACID事物的支持，实现了SQL标准的四种隔离级别，即读未提交，不可重复读，可重复读以及串行,其涉及目标就是处理大数据容量的数据库系统。而MyIASM引擎是Mysql默认的引擎，不提供数据库事务的支持，也不支持行级锁和外键，因此当写操作时需要锁定整个表，效率较低。不过其保存了表的行数，当金星select count(\*)form table时，可直接读取已经保存的值，不需要进行全表扫描。因此当表的读操作远多于写操作，并且不需要事务支持时，可以优先选择MyIASM

**数据库视图的作用**：

视图是从一个或多个表导出的虚拟的表，其内容由查询定义，具有普通表的结构，但是不实现数据存储。对于视图的修改：单表视图一般用于查询和修改，会改变基本表的数据；多表视图一般用于查询，不会改变基本表的数据

（1）简化操作，把常用的数据定义为视图

（2）安全性，用户只能查询和修改能看到的数据

（3）逻辑上独立型，屏蔽真实表的结构带来的影响

缺点：

（1）性能差：

（2）修改限制

**脏读幻读**：

脏读：无效数据读出，一个事务读取另外一个事务还没有提交的数据，办法：把数据库的事务隔离级别调整到read\_committed

不可重复读：指的是同一个事务内，两个相同的查询返回了不同的结果。办法：把数据库的事务隔离级别调整为REPEATABLE\_READ

幻读：事务在执行读取操作的时候，需要统计两次数据的总量，前一次查询之后，此时事务 B执行新增的操作并且提交之后，这时事务再次读取的时候，和前一次不一样，平白无故。

脏读、不可重复读、幻读的级别高低是：脏读 < 不可重复读 < 幻读。所以，设置、最高级别SERIALIZABLE\_READ就不用设置REPEATABLE\_READ和READ\_COMMITTED了。

**数据库的锁**：

按照锁的粒度划分：表级锁，行级锁，页级锁

按照锁的级别划分：共享锁，排它锁，意向锁

按加锁的方式划分：隐式加锁，显示加锁

按照使用的方式划分：悲观锁、乐观锁

**行级锁：**

行级锁分为共享锁和排他锁。行级锁是 MySQL 中锁定粒度最细的锁。InnoDB 引擎支 持行级锁和表级锁，只有在通过索引条件检索数据的时候，才使用行级锁，否则使用表级锁。 行级锁开销大，加锁慢，锁定粒度最小，发生锁冲突的概率最低，并发度高。

**表级锁：**

表级锁分为表共享锁和表独占锁，表级锁开销小，加锁快，锁定粒度大，发生锁冲突最 高，并发度最低。

**页级锁：**

页级锁是 MYSQL 中锁定粒度介于行级锁和表级锁中间的一种锁。表级锁速度快，但冲 突多，行级冲突少，但速度慢，所以取折中的页级锁，一次锁定相邻的一组记录。BDB 支持 页级锁，开销和加锁时间介于表锁和行锁之间，会出现死锁。锁定粒度界于表锁和行锁之间， 并发一般。

**共享锁/读锁：**

共享锁是针对同一份数据，多个读操作可以同时进行，简单来说即读加锁，不能写并且 可并行读；

**排它锁/写锁:**

排它锁针对写操作，假如当前写操作没有完成，那么它会阻断其它的写锁和读锁，即写 加锁，其它读写都阻塞

**意向锁：**

当事务 A 对一行数据加读锁，而这时候事务 B 又要对这个行数据所在的表加写锁，这 是存在冲突，如果系统逐行检查存在的行锁效率很低，所以存在了意向锁。当事务 A 申请 行锁额的时候，数据库会自动先开始申请表的意向锁，如果事务 B 发现表中有意向锁时，就 会阻塞等到锁释放。

**间隙锁(next-locks)多版本并发**：

当使用范围条件而不是相等条件检索数据，并请求共享或排它锁时，InnoDB 会给符合 条件的已有数据记录的索引项加锁；对于键值在条件范围内但并不存在的记录叫做“间隙”， InnoDB 也会对这个间隙加锁，这种锁机制就是所谓的间隙锁。InnoDB 使用间隙锁一方面是 为了防止幻读；另一方面是为了满足恢复和复制的需求。显然，在使用范围条件检索并锁定 记录时，InnoDB 这种加锁机制会阻塞符合条件范围内键值的并发插入，往往造成严重的锁 等待，因此在实际应用开发过程中，尤其是并发插入较多的应用，要尽量优化业务逻辑，尽 量使用相等条件来访问更新数据，避免使用范围条件。

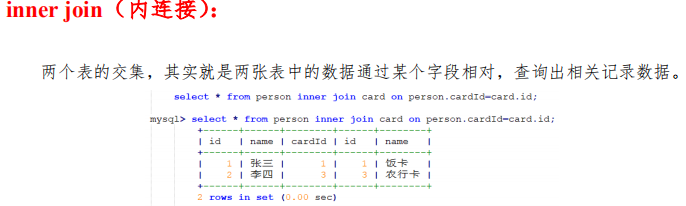
**悲观锁：**

悲观锁认为被它保护的数据是极其不安全的，每时每刻都有可能变动，一个事务拿到悲 观锁后（可以理解为一个用户），其他任何事务都不能对该数据进行修改，只能等待锁被释 放才可以执行。

**乐观锁**：

乐观锁的“乐观情绪”体现在认为数据的变动不会太频繁。因此，它允许多个事务同时对 数据进行变动。 当事务从数据库中提取一个数据时，会同时提取其时间戳，在事务提交时， 对比之前提取的时间戳与现在数据库中该数据的时间戳是否一样，如果一样就认为这段时间 没有其它事务操作数据，可以提交。如果不一样，则交给用户处理。 对于 UPDATE、DELETE、INSERT 语句会自动给涉及数据集加排它锁；对于普通的 SELECT 语句不会加任何锁。

**内连接外链接**：



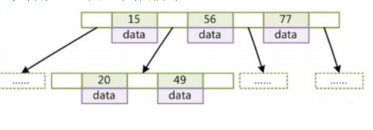


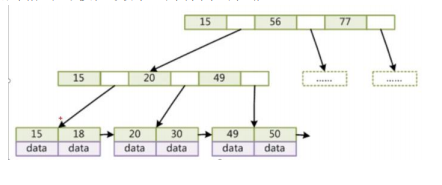




**B树与B+树：**

**B树**：

1、 叶子节点具有相同的深度，叶子节点的指针为空   
2、 所有索引元素不重复   
3、 节点中的数据索引从左到右递增排列  
  
**B+树：**  
1、 非叶子节点不存储 data，只存储索引，则可以放更多的索引（因为一次读取的空 间大小相同，而 B 树的索引还包含 data 的内容，所以放的索引小）   
2、 叶子节点包含所有的索引字段，查询性能稳定  
3、 叶子节点用指针连接，提高区间访问的性能



**聚集索引与非聚集索引**：

（索引和数据存在一起就是聚集索引，分开存储就是非聚集索引, 聚集索引的叶子节点包含完整的数据记录）

聚集索引（聚簇索引）：数据行的物理顺序与列值（一般是主键的那一列）的逻辑顺序相同，一个表中只能拥有一个聚集索引。

非聚集索引：该索引中索引的逻辑顺序与磁盘上行的物理存储顺序不同，一个表中可以拥有多个非聚集索引。

**Innodb 索引文件（聚集索引）**：

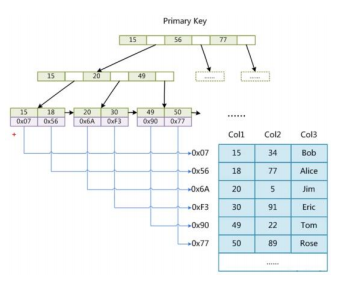
表数据文件本身就是按照 B+Tree 组织一个索引结构文件

\*.frm: 存储的表的结构

\*ibd: 存储的是索引和数据，就是一个 B+树，数据和索引是存在一起的，而不是地址指针

**MyISAM 索引文件和数据文件是分离的（非聚集索引）**：

\*.frm : 存储的是表的结构，字段名之类的   
\*.MYD：存储的是数据 Data   
\*.MYI：存储的是索引   
查找过程：select \* from test where col==49 先在表结构中查询 col 在不在 frm 文件中，找到后在 MYI 文件中查找 col=49 的索引，索引 的叶子节点挂载着 data 数据的磁盘地址指针，根据这个指针在 MYD 文件中找出数据。



**为什么 InnoDB（聚集）表必须有主键，并且推荐使用整形的自增主键**？

innoDB 的底层实现就是默认有主键的，如果出现没有主键的情况，系统首先会选一个适合作为主键的列作为默认主键，如果没有合适的，就会自动生成一列来作为索引。

主键使用整形而不使用 UUID 是因为整形所占的空间小，而且在大小比较的过程中计算开销较小。使用自增主键是当主键不是自增时，就可能向前面已经有序的数据做一个插入操作，而这个插入操作，可能会让 B+树分叉，改变层数，重新平衡等大量操作。而自增有序会顺序插入。

**垂直拆分**：

**垂直分库**：当一个数据库中表中的数据量过大时，将其中不同的表建立不同的数据库存

放，这样有的业务只依赖其中几张表就可以把这几张表分出来做成相应服务。

**垂直分表：**适用于字段比较多的表中，例如可以将常用的 20%字段和不常用的 80%字段分开，然后使用主键进行联合。垂直分表可解决跨页问题。跨页问题是一行数据过大，存储空间超过一页，而需要用到的也就其中几个常用字段，垂直分表可以减少磁盘 IO 次数，

读取跟多的数据到内存。

**垂直拆分的优缺点：**

优点：

1、跟随业务进行分割，和微服务概念相似，方便解耦之后的管理及扩展。

2、高并发的场景下，垂直拆分使用多台服务器的 CPU、I/O、内存能提升性能，同时，对单机数据库连接数、一些资源限制也得到了提升。

3、能实现冷热数据的分离。

缺点：1、部分业务表无法 join，应用层需要很大的改造，只能通过聚合的方式实现。增加了开发难度。

2、当单库中的表的数据量增大的时候依然没有得到有效解决。

3、分布式事务也是难点。

**水平拆分**：

**库内分表**：例如 Order 表中的数据达到 50000 万行，对读写效率影响非常大。所以考虑按照订单编号进行 range 划分，分为五段区间存储在不同的表中。

**分库分表**：上面的库内分表虽然提高了读写效率，但是这些表仍然在一个数据库内，CPU内存 IO 都还存在限制。所以在库内分表的基础上，将分表挪动到不同的主机和数据库上。缺陷是当不根据订单编号查询时，就不知道要查询的内容在哪个库，就需要广播查询，每个库都检索，系统要开的线程较多。

**优点**：

1、水平扩展能够无限扩展，不存在某个库某个表过大的情况。

2、能够较好的应对高并发，同时将热点数据打散。

3、应用侧的改动较小，不需要根据业务来拆分。

**缺点**：

1、需要增加一层路由计算，不带分片键的查询会产生广播

2、跨库 join 的性能较差

3、需要处理分布式事务的一致性问题

物理表逻辑表：物理表就是实际划分的存储到不同库或者表里面的表，而逻辑表则是被划分的表在应用层看来仍然是一个整个的表。

Nginx：高性能的 Web 服务器。

**为什么非主键索引结构叶子节点存储的是主键值？**

一致性

当数据库表进行 DML 操作时，同一行记录的页地址会发生改变，因非主键索引保存的

是主键的值，无需进行更改。

节省存储空间

Innodb 数据本身就已经汇聚到主键索引所在的 B+树上了， 如果普通索引还继续再保

存一份数据，就会导致有多少索引就要存多少份数据。

**红黑树**：

**二叉查找树、平衡树与红黑树的关系？**

虽然平衡树解决了二叉查找树退化为近似链表的缺点，能够把查找时间控制在 O(logn)，

不过却不是最佳的，因为平衡树要求每个节点的左子树和右子树的高度差至多等于 1，这个

要求实在是太严了，导致每次进行插入/删除节点的时候，几乎都会破坏平衡树的第二个规

则，进而我们都需要通过左旋和右旋来进行调整，使之再次成为一颗符合要求的平衡树。

显然，如果在那种插入、删除很频繁的场景中，平衡树需要频繁着进行调整，这会使平

衡树的性能大打折扣，为了解决这个问题，于是有了红黑树。

与平衡树不同的是，红黑树在插入、删除等操作，不会像平衡树那样，频繁着破坏红黑

树的规则，所以不需要频繁着调整，这也是我们为什么大多数情况下使用红黑树的原因。但

单单在查找方面的效率的话，平衡树比红黑树快。

所以，我们也可以说，红黑树是一种不大严格的平衡树。也可以说是一个折中发方案。

平衡树是为了解决二叉查找树退化为链表的情况，而红黑树是为了解决平衡树在插入、

删除等操作需要频繁调整的情况。

**红黑树的性质：**

1) 每个结点不是红色就是黑色。

2) 不可能有连在一起的红色结点。

3) 根节点都是黑色 root。

4) 每个红色结点的两个子结点都是黑色。叶子节点都是黑色：出度为 0.

**红黑树的三种变换：**

1) 颜色变换

2) 左旋

3) 右旋

**红黑树的变换规则：（所有插入的点默认为红色**）

**1)变颜色的情况，**当前结点的父亲是红色，且它的爷爷结点的另一个子结点也是红色（叔

叔结点）。

1- 把父结点变为黑色

2- 把叔叔结点变为黑色

3- 把爷爷结点变为红色

4- 把指针定义到爷爷结点设为当前要操作的结点

**2) 左旋**：当前父结点是红色，叔叔是黑色的时候，且当前的结点是右子树，以父结点进

行左旋。把指针定位到刚才的父结点。

**3)右旋：**当前父结点是红色，叔叔是黑色的时候，且当前的结点是左子树。右旋 。

1- 把父结点变为黑色

2- 把爷爷结点变为红色

3- 以爷爷结点右旋