Mysql

1. 数据库事务
   1. 事务隔离级别

|  |  |  |  |
| --- | --- | --- | --- |
| 事务隔离级别 | 脏读 | 不可重复读 | 幻读 |
| 读未提交（read-uncommitted） | 是 | 是 | 是 |
| 不可重复读（read-committed） | 否 | 是 | 是 |
| 可重复读（repeatable-read） | 否 | 否 | 是 |
| 串行化（serializable） | 否 | 否 | 否 |

* 1. 事务的并发问题
* **脏读：**一个事务读取了另外一个事务没有提交的数据;
* **不可重复读：**事务 A 多次读取同一数据，事务 B 在事务A多次读取的过程中，对数据作了更新并提交，导致事务A多次读取同一数据时，结果 不一致;
* **幻读：**系统管理员A将数据库中所有学生的成绩从具体分数改为ABCDE等级，但是系统管理员B就在这个时候插入了一条具体分数的记录，当系统管理员A改结束后发现还有一条记录没有改过来，就好像发生了幻觉一样，这就叫幻读;

小结：不可重复读的和幻读很容易混淆，不可重复读侧重于修改，幻读侧重于新增或删除。

解决不可重复读的问题只需锁住满足条件的行，解决幻读需要锁表

1. 数据库的锁机制

InnoDB存储引擎既支持**行级锁（row-level locking），也支持表级锁，**但默认情况下是采用**行级锁**。

* **表级锁：**开销小，加锁快；不会出现死锁；锁定粒度大，发生锁冲突的概率最高，并发度最低。
* **行级锁：**开销大，加锁慢；会出现死锁；锁定粒度最小，发生锁冲突的概率最低，并发度也最高。
* **页面锁：**开销和加锁时间界于表锁和行锁之间；会出现死锁；锁定粒度界于表锁和行锁之间，并发度一般

从上述特点可见，很难笼统地说哪种锁更好，只能就具体应用的特点来说哪种锁更合适！仅从锁的角度 来说：

表级锁更适合于以查询为主，只有少量按索引条件更新数据的应用，如Web应用；

而行级锁则更适合于有大量按索引条件并发更新少量不同数据，同时又有 并发查询的应用，如一些在线事务处理（OLTP）系统。

* 1. InnoDB的行锁模式及加锁方法
     1. 两种行级锁

**共享锁（s）：又称读锁。**允许一个事务去读一行，阻止其他事务获得相同数据集的排他锁。若事务T对数据对象A加上S锁，则事务T可以读A但不能修改A，其他事务只能再对A加S锁，而不能加X锁，直到T释放A上的S锁。这保证了其他事务可以读A，但在T释放A上的S锁之前不能对A做任何修改。

**排他锁（Ｘ）：又称写锁。**允许获取排他锁的事务更新数据，阻止其他事务取得相同的数据集共享读锁和排他写锁。若事务T对数据对象A加上X锁，事务T可以读A也可以修改A，其他事务不能再对A加任何锁，直到T释放A上的锁。

对于共享锁很好理解，就是**多个事务**只能读数据不能改数据。

对于排他锁的理解可能就有些差别:

错误的理解：以为排他锁锁住一行数据后，其他事务就不能读取和修改该行数据；

正确的理解：排他锁指的是一个事务在一行数据加上排他锁后，**其他事务不能再在其上加其他的锁**。

* + 1. 加行级锁的方式
* InnoDB引擎默认的修改数据语句：update,delete,insert都会**自动给涉及到的数据加上排他锁;**
* **select语句默认不会加任何锁类型**，如果加排他锁可以使用select …for update语句，加共享锁可以使用select … lock in share mode语句。

所以加过排他锁的数据行在其他事务种是不能修改数据的，也不能通过for update和lock in share mode锁的方式查询数据，但可以直接通过select …from…查询数据，因为**普通查询没有任何锁机制**。

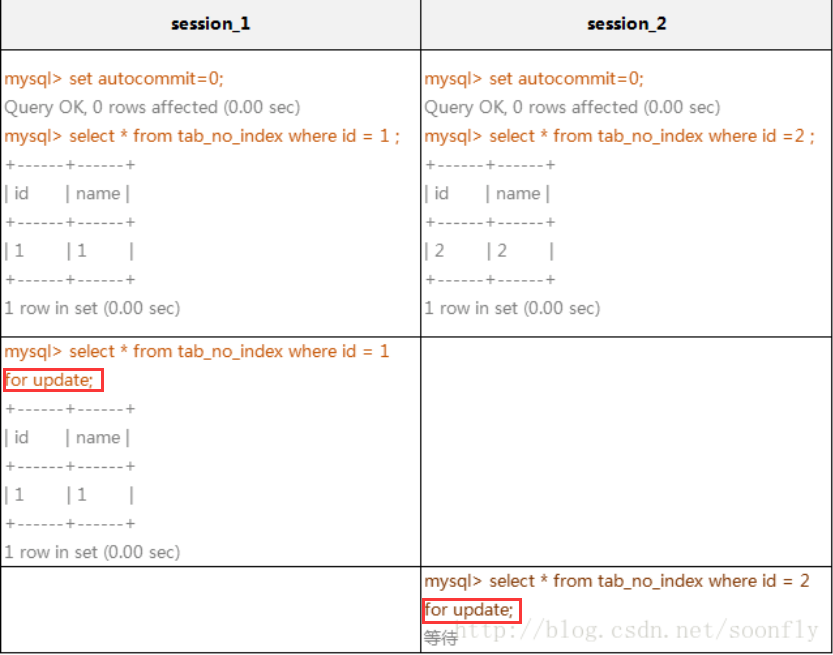
* + 1. 行级锁的实现方式

InnoDB行锁是通过**给索引上的索引项加锁来实现的**，这一点MySQL与Oracle不同，后者是通过在数据块中对相应数据行加锁来实现的。

InnoDB这种行锁实现特点意味着：**只有通过索引条件检索数据，InnoDB才使用行级锁，否则，InnoDB将使用表锁！**

1、在不通过索引条件查询的时候，InnoDB确实使用的是表锁，而不是行锁

没有索引使用表锁



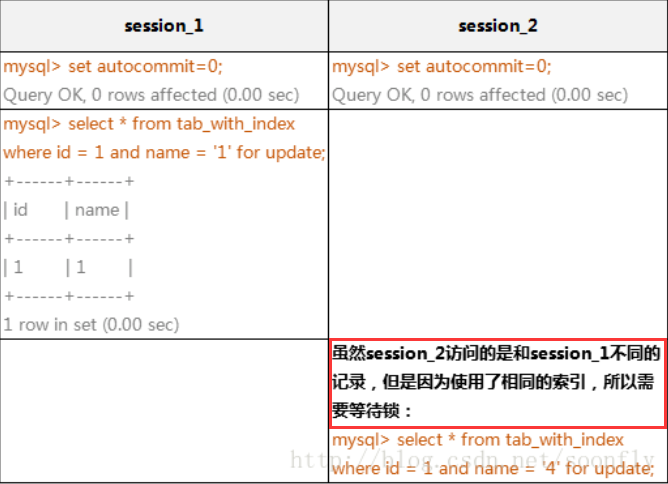
在上面的例子中，看起来session\_1只给一行加了排他锁，但session\_2在请求其他行的排他锁时，却出现了锁等待！原因就是在没有索引的情况下，InnoDB只能使用表锁。

相同的索引键出现锁冲突

由于MySQL的**行锁是针对索引加的锁**，不是针对记录加的锁，所以虽然是访问不同行的记录，但是如果是使用相同的索引键，是会出现锁冲突的。

应用设计的时候要注意这一点。

在下面的例子中，表tab\_with\_index的id字段有索引，name字段没有索引:

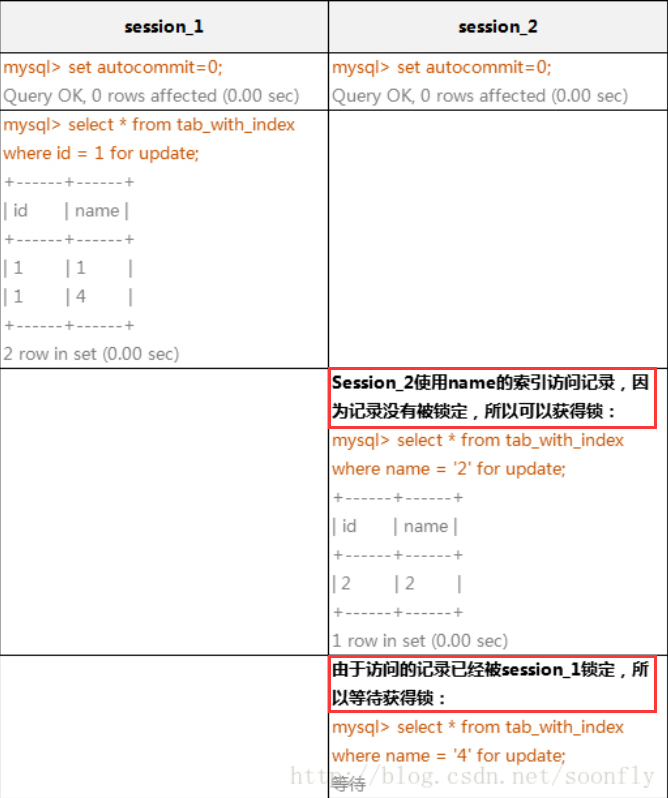


多个索引列分别进行行锁

当表有多个索引的时候，不同的事务可以使用不同的索引锁定不同的行；

不论是使用主键索引、唯一索引或普通索引，InnoDB都会使用行锁来对数据加锁；

如果一个事务通过一个索引锁定了一行，另外一个事务是不能通过其他的索引来锁定相同行的；



索引失效导致行锁变成表锁

另见: [索引不可用的情况](#索引不可用的情况)

在条件中使用了索引字段，但是否使用索引来检索数据是由MySQL通过判断不同执行计划的代价来决 定的，如果MySQL认为全表扫描效率更高，比如对一些很小的表，它就不会使用索引，这种情况下InnoDB将使用表锁，而不是行锁。因此，在分析锁冲突 时，别忘了检查SQL的执行计划，以确认是否真正使用了索引。

比如，在tab\_with\_index表里的name字段有索引，但是name字段是varchar类型的，检索值的数据类型与索引字段不同，虽然MySQL能够进行数据类型转换，但却不会使用索引，从而导致InnoDB使用表锁。通过用explain检查两条SQL的执行计划，我们可以清楚地看到了这一点

* 1. 表级锁（意向锁）

为了允许行锁和表锁共存，实现多粒度锁机制，InnoDB还有两种内部使用的意向锁（Intention Locks），这两种意向锁都是表锁。

**意向共享锁（IS）：**事务打算给数据行共享锁，**事务在给一个数据行加共享锁前必须先取得该表的IS锁**。

**意向排他锁（IX）：**事务打算给数据行加排他锁，**事务在给一个数据行加排他锁前必须先取得该表的IX锁。**

* 1. 间隙锁（Next-Key锁）

概念：

当我们用**范围条件而不是相等条件检索数据**，并请求共享或排他锁时，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（GAP)”，InnoDB也会对这个“间隙”加锁，这种锁机制就是所谓的间隙锁 （Next-Key锁）。

举例：

假如emp表中只有101条记录，其empid的值分别是 1,2,…,100,101，下面的SQL：

Select \* from emp where empid > 100 **for update**;

是一个范围条件的检索，InnoDB不仅会对符合条件的empid值为101的记录加锁，**也会对empid大于101（这些记录并不存在）的“间隙”加锁**。

InnoDB使用间隙锁的目的：

**一方面是为了防止幻读：**以满足相关隔离级别的要求，对于上面的例子，要是不使 用间隙锁，如果其他事务插入了empid大于100的任何记录，那么本事务如果再次执行上述语句，就会发生幻读；

**另外一方面是为了满足其恢复和复制的需 要**：有关其恢复和复制对锁机制的影响，以及不同隔离级别下InnoDB使用间隙锁的情况;

1. 常见问题
   1. 索引不可用的情况

1.**如果条件中有or**，即使其中有条件带索引也不会使用(这也是为什么尽量少用or的原因)

注意：要想使用or，又想让索引生效，只能将or条件中的每个列都加上索引；

2.对于多列索引，不是使用的第一部分，则不会使用索引；

3.like查询是以%开头；

4.如果列类型是字符串，那一定要在条件中将数据使用引号引用起来,否则不使用索引；

5.如果mysql估计使用全表扫描要比使用索引快,则不使用索引

* 1. select 加锁导致的死锁

SELECT ... IN SHARE MODE获得共享锁，主要用在需要数据依存关系时来确认某行记录是否存在，并确保没有人对这个记录进行UPDATE或者DELETE操作。

但是如果**当前事务也需要对该记录进行更新操作**，则很有可能造成死锁，**对于锁定行记录后需要进行更新操作的应用，应该使用SELECT… FOR UPDATE方式获得排他锁**。

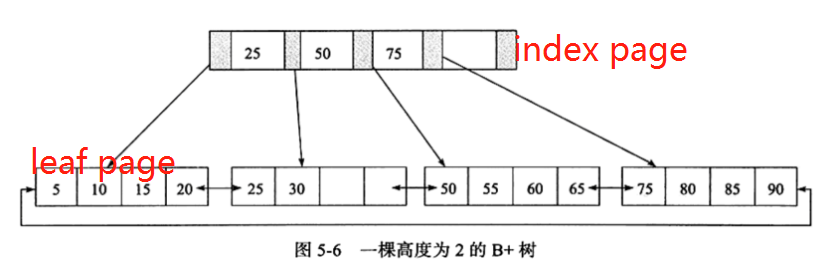
* 1. 范围检索并且锁定记录造成的阻塞

在使用范围条件检索并锁定记录时，InnoDB的间隙锁机制会阻塞符合条件范围内键值的并发插入，这往往会造成严重的锁等待。因此，在实际应用开发中，尤其是并发插入比较多的应用，我们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；

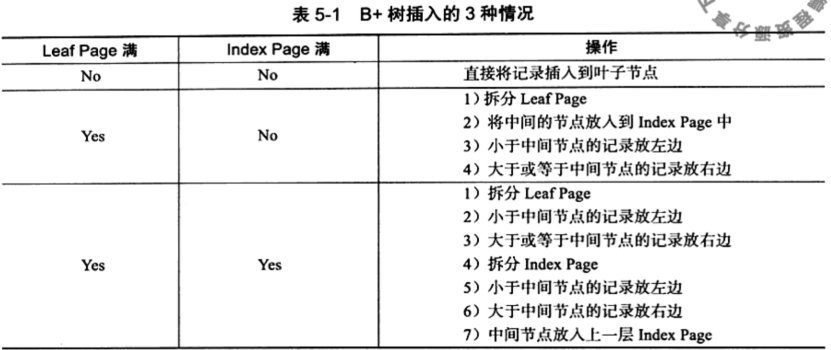
1. 索引实现

* MySQL支持多种索引，包括B+树索引（最为常用）、全文索引、哈希索引等等。
* 数据库中的B+树索引可以分为**聚集索引**和**辅助索引**，但是不管是聚集还是辅助的索引，其内部都是B+树，是高度平衡的，叶子结点存放着所有的数据。
* 聚集索引和辅助索引最大的不同在于，**叶子结点存放的是否是一整行的信息**。
  1. 为什么不使用B树
* 因为B+树的分支结点并不会存储关键字的具体信息，只存储关键字，所以相较于B树也较小，因此一次I/O操作所能够容纳的关键字就多一些，那么读取一个结点的I/O操作次数也就少一些；
* **B+树的所有关键字的具体信息都存储在叶子结点**，通常都会使用链表将叶子结点连接起来，遍历叶子结点就能够获取到所有的数据，也就可以进行区间查询，而B树只有中序遍历才能够获取到所有的数据
  1. B+树

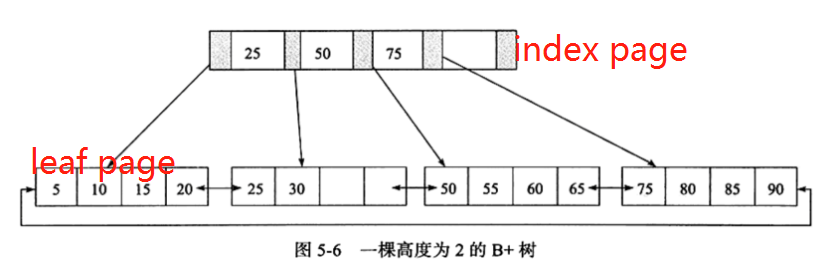
B+树是为磁盘或者其它直接存取辅助设备设计的一种**平衡树**，在B+树中，所有记录结点都是按照键值的大小顺序存**放在同一层的叶子结点上**，由各叶子结点指针进行连接；



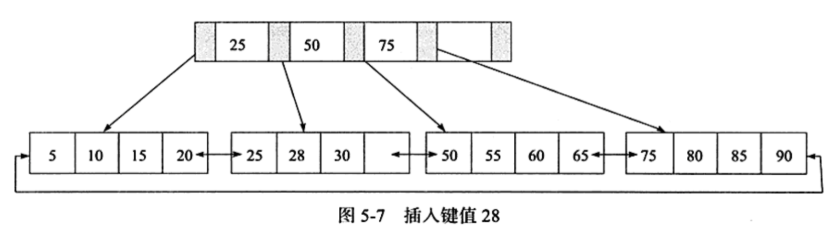
* + 1. 插入操作



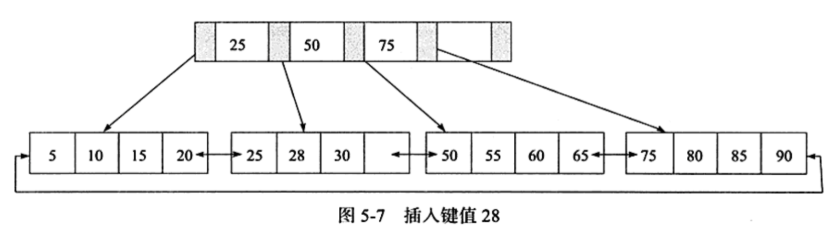
直接插入



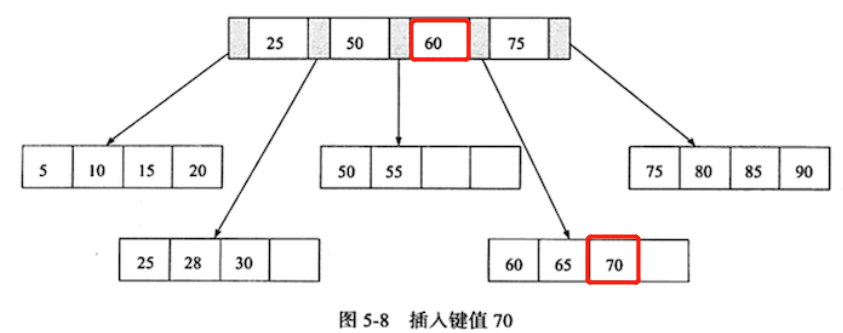
还是根据上面的那副图作为例子，往里面加入28，我们索引到具体的位置，在25-30中间，我们发现当前的情况为**叶子结点不为空，父亲结点也不为空**，查询表格发现我们可以直接将纪录插入到叶子结点中，所以如下：



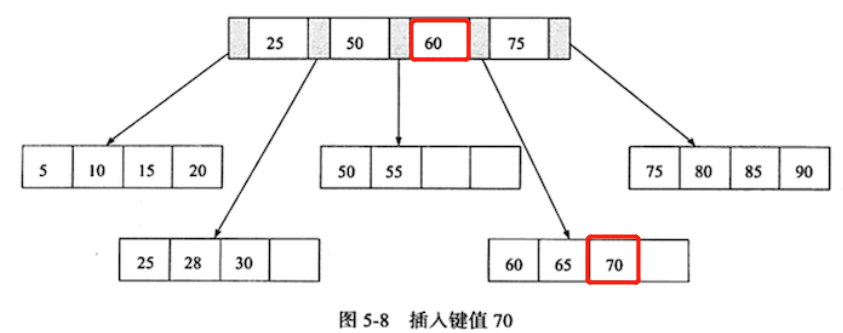
拆分叶子节点



接着往里面插入70，我们发现在**叶子结点已经满了，但是父亲结点并没有满**，所以需要将叶子结点进行分裂，因为70在最右边，所以将中间的数字60向上移动，放到原来父亲结点中75的位置，而75则向后移动一位，得到如下图：



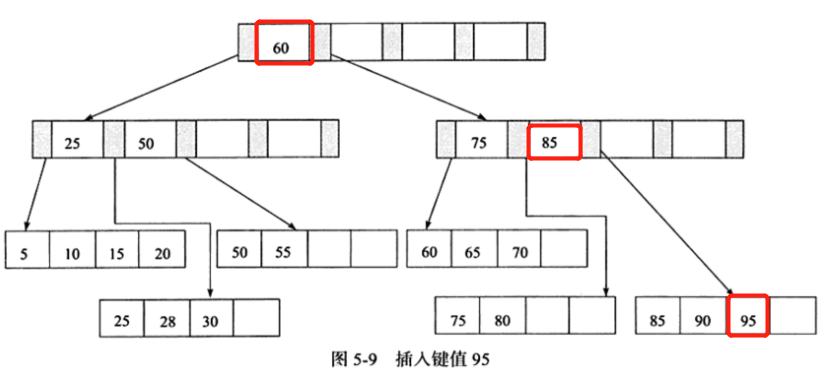
拆分叶子，索引节点



最后插入95，根据图片索引到其插入位置为叶子结点中90的后面，此时我们发现**叶子结点和父亲结点都已经满了**，于是它们两个都需要进行分裂：

1. 首先将叶子结点中的中间位置85向上移动；
2. 将叶子结点分裂成两个叶子结点（左边比85小，右边比85大）；
3. 此时父亲结点也放不下85，于是将中间位置的60向上移动单独成为一个结点；
4. 左子结点中的值比60小，右子结点的值比60大；

具体如下图：

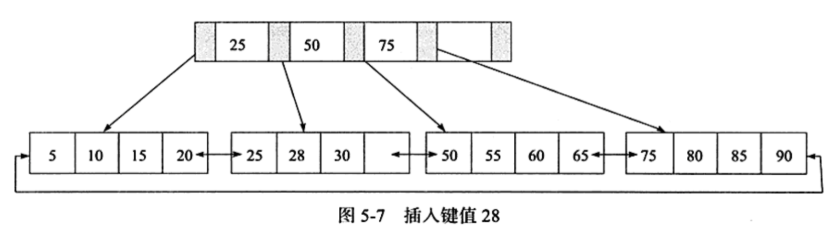


* + 1. 旋转

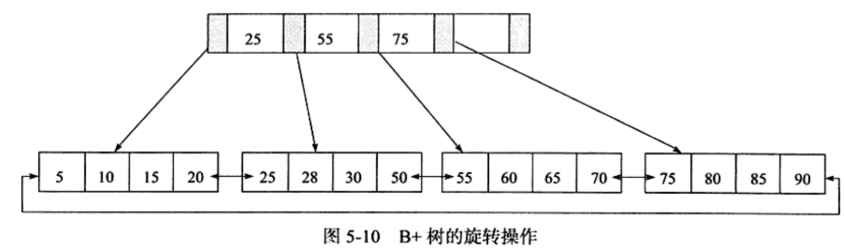
上面的操作**为了保持平衡而进行大量的拆分页操作**，因为B+树主要用于磁盘，而拆分则意味着磁盘的操作，所以要在尽可能的情况下减少拆分的动作。

B+树也提供了旋转的功能：**在当前叶子结点满了，但是兄弟结点没有满的情况下，可以将部分结点移动到兄弟结点中，并且调整父亲结点**。

比如在前面提到的加入70，其图可以为：

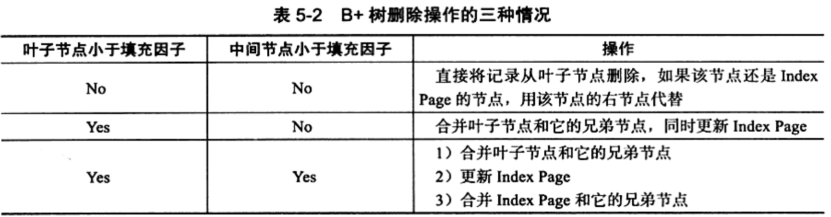


将50移动到了左边的结点，然后调整父亲结点扩大到第一个结点的纪录值，即55

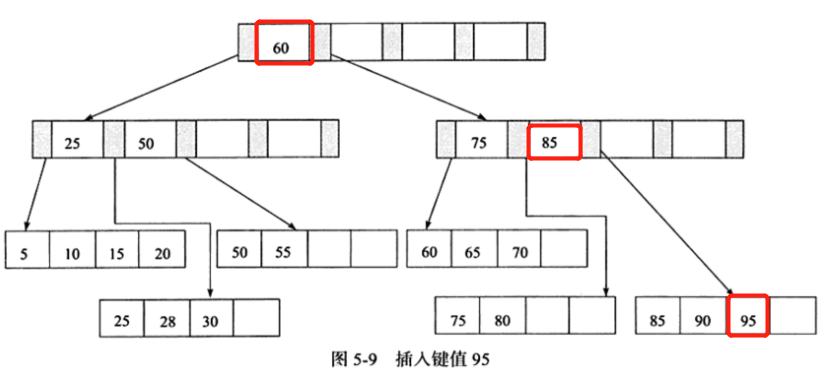


* + 1. 删除操作

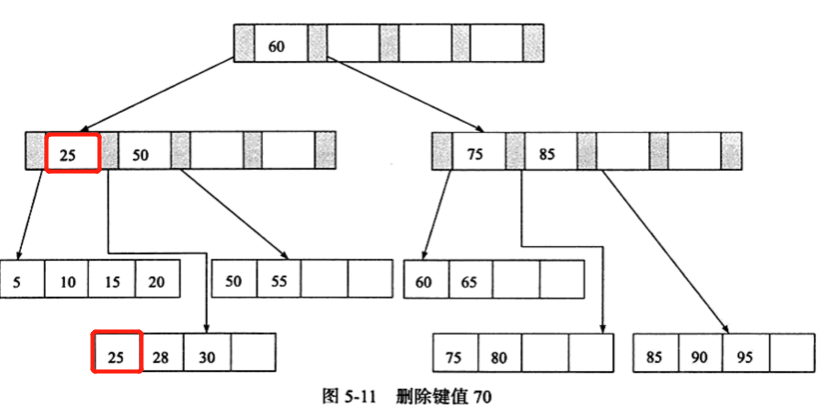
B+树使用**填充因子**来控制树的删除变化，**填充因子最小可设为50%**；同样的，B+树的删除操作需要保证删除后叶子结点中的纪录依然排序，删除分为好几种情况，如下：



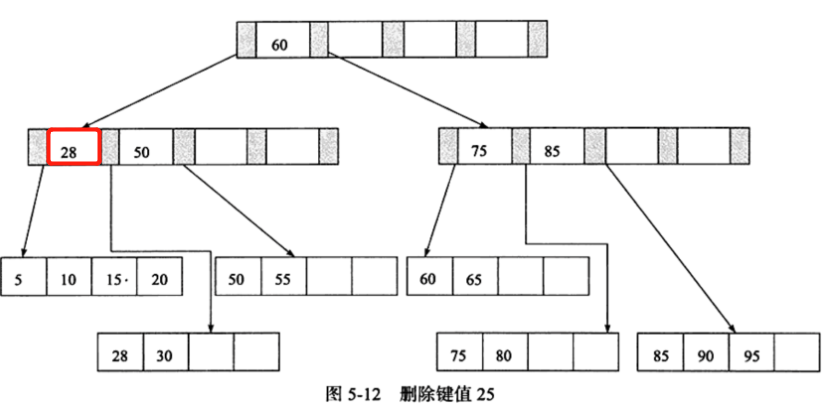
直接删除



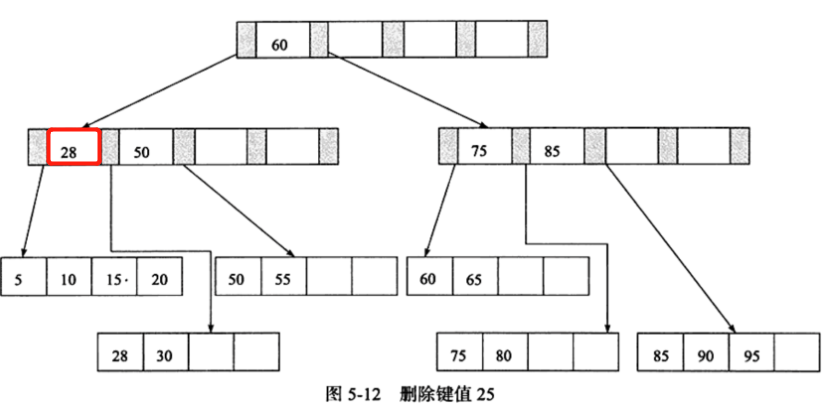
删除70后，发现叶子结点和其父亲结点的值的个数并没有小于一半，所以不需要进行其它的操作。



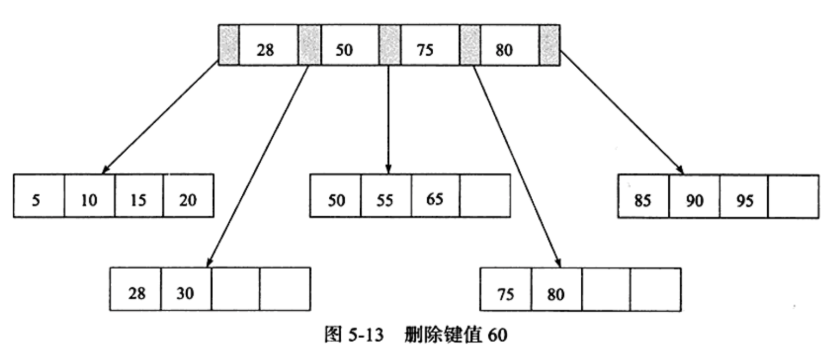
接下来删除25，因为同样是删除表格中的第一种操作，所以直接删除即可，**但是需要注意的是，其父亲结点仍然存在25，所以需要将最左边的28更新到父亲结点中即可**，如图下：



叶节点小于填充因子



接着删除60，我们发现删除60后，**该叶子结点已经小于填充因子**，因此这种情况属于表格中的第三种情况，则需要合并兄弟结点，因为剩余的数字小于75，所以要合并到左边的兄弟结点，同时把根结点的60删除掉，合并它的两个子结点，如图下：



叶节点，父节点都小于填充因子

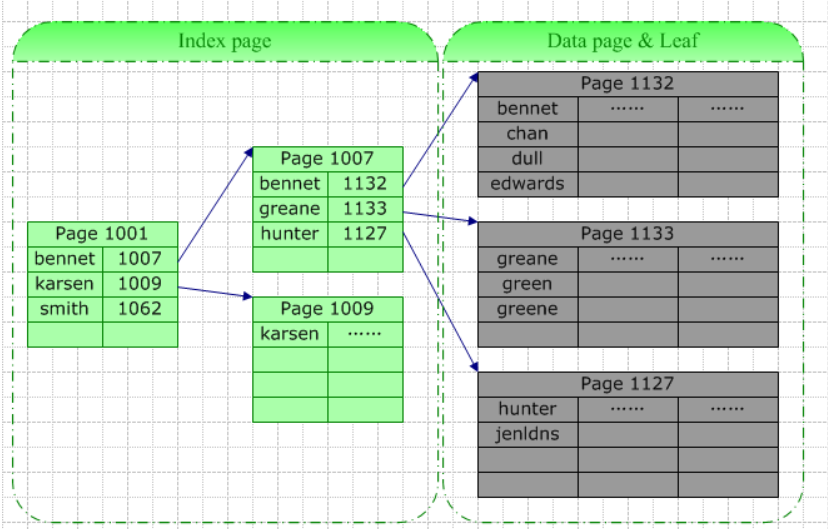
* 1. 索引

数据库中的B+树索引可以分为**聚集索引和辅助索引**，但是不管是聚集索引还是辅助索引，其内部都是B+树的，即高度平衡的，叶子结点存放着所有的数据，它们的不同点在于，叶子结点存放的是否是真实的数据行信息;

* + 1. 聚集索引

InnoDB存储引擎表是**索引组织表**，即表中数据按照主键顺序存放，聚集索引就是按照每张表的主键构造一颗B+树，同时叶子结点存放的即为整张表的行纪录数据（**聚集索引的叶子结点也称为数据页**）。

聚集索引的这个特性也决定了**索引组织表中数据也是索引的一部分**，和B+树数据结构一样，每个数据页之间都通过一个双向链表来进行链接。



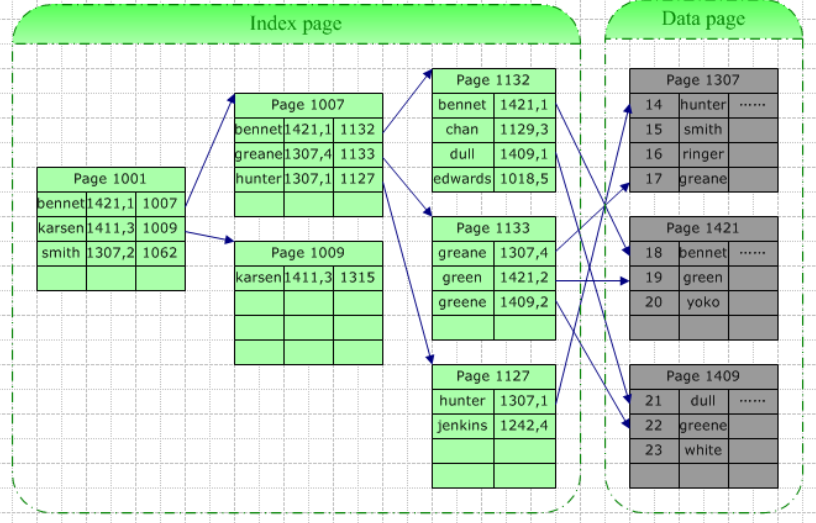
一次查询操作将包含3个索引页的读取:

它对于主键的排序查找和范围查找速度非常快。

叶子结点的数据就是用户所要查询的数据。

另外一个是范围查询，即如果要查找主键某一个范围内的数据，通过叶子结点的上层中间结点就可以得到页的范围，之后直接读取数据页即可。

* + 1. 非聚集索引



一次查询操作将包含3个索引页的读取+1个数据页的读取

* + 1. 两种索引的对比

叶子结点并非数据结点

叶子结点为每一真正的数据行存储一个“键-指针”对

叶子结点中还存储了一个指针偏移量，根据页指针及指针偏移量可以定位到具体的数据行

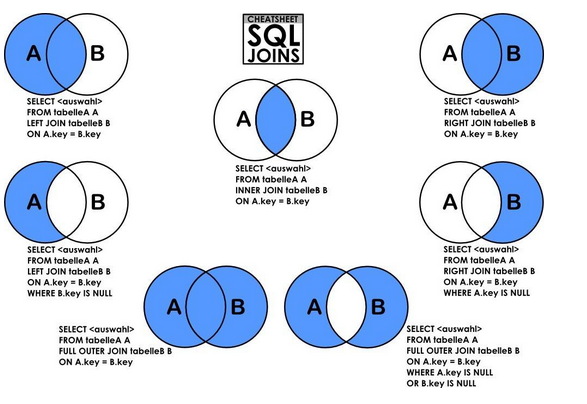
类似的，在除叶结点外的其它索引结点，存储的也是类似的内容，只不过它是指向下一级的索引页的

* 1. 创建索引的规则
* **最左前缀匹配原则**，数据库会一直向右匹配**直到遇到范围查询就停止匹配**，比如a = 1 and b = 2 and c > 3 and d = 4 如果建立(a,b,c,d)顺序的索引，d是用不到索引的；
* =和in可以乱序，比如a = 1 and b = 2 and c = 3 建立(a,b,c)索引可以任意顺序；
* 尽量选择**区分度高的列作为索引**,区分度的公式是count(distinct col)/count(\*)，表示字段不重复的比例，比例越大我们扫描的记录数越少，唯一键的区分度是1，而一些状态、性别字段可能在大数据面前区分度就是0，所以最好别对这些字段创建索引；
* **索引列不能参与计算，保持列“干净”**，因为B+树中存的都是数据表中的字段值，但进行检索的时候，需要把所有元素都应用函数才能比较，显然成本太大；
* **尽量的扩展索引，不要新建索引**。比如表中已经有a的索引，现在要加(a,b)的索引，那么只需要修改原来的索引即可；
* **索引要建立在经常进行select操作的字段上**。这是因为，如果这些列很少用到，那么有无索引并不能明显改变查询速度。相反，由于增加了索引，反而降低了系统的维护速度和增大了空间需求。
  1. 索引使用的优化
* **应尽量避免在 where 子句中对字段进行 null 值判断**，否则将导致引擎放弃使用索引而进行全表扫描；例子：select id from t where num is null
* 应尽量避免在 where 子句中使用!=或<>操作符，否则将引擎放弃使用索引而进行全表扫描；
* **应尽量避免在 where 子句中使用 or 来连接条件**，否则将导致引擎放弃使用索引而进行全表扫描；（可以使用union来代替or）
* **in 和 not in 也要慎用，因为in会使系统无法使用索引**，而只能直接搜索表中的数据，对于连续的数值，能用 between 就不要用 in；
* **尽量避免在索引过的字符数据中，使用非打头字母搜索**，这也使得引擎无法利用索引；比如：SELECT \* FROM T1 WHERE NAME LIKE ‘**%**L%’
* **应尽量避免在 where 子句中对字段进行表达式操作或者函数操作**，都会导致引擎放弃使用索引而进行全表扫描；比如：SELECT \* FROM T1 WHERE SUBSTING(NAME,2,1)=’L’

1. 查询方法
   1. join的用法

**JOIN**的含义就如英文单词“join”一样，连接两张表，大致分为内连接，外连接，右连接，左连接，自然连接。

如下图所示：



示例：

CREATE TABLE t\_blog(

id INT PRIMARY KEY AUTO\_INCREMENT,

title VARCHAR(50),

typeId INT

);

SELECT \* FROM t\_blog;

+----+-------+--------+

| id | title | typeId |

+----+-------+--------+

| 1 | aaa | 1 |

| 2 | bbb | 2 |

| 3 | ccc | 3 |

| 4 | ddd | 4 |

| 5 | eee | 4 |

| 6 | fff | 3 |

| 7 | ggg | 2 |

| 8 | hhh | NULL |

| 9 | iii | NULL |

| 10 | jjj | NULL |

+----+-------+--------+

-- 博客的类别

CREATE TABLE t\_type(

id INT PRIMARY KEY AUTO\_INCREMENT,

name VARCHAR(20)

);

SELECT \* FROM t\_type;

+----+------------+

| id | name |

+----+------------+

| 1 | C++ |

| 2 | C |

| 3 | Java |

| 4 | C# |

| 5 | Javascript |

+----+------------+

* + 1. 笛卡尔积：CROSS JOIN

要理解各种JOIN首先要理解笛卡尔积。

笛卡尔积就是将A表的每一条记录与B表的每一条记录强行拼在一起。所以，如果A表有n条记录，B表有m条记录，笛卡尔积产生的结果就会产生n\*m条记录。

下面的例子，t\_blog有10条记录，t\_type有5条记录，所有他们俩的笛卡尔积有50条记录。

SELECT \* FROM t\_blog CROSS JOIN t\_type;

SELECT \* FROM t\_blog INNER JOIN t\_type;

SELECT \* FROM t\_blog,t\_type;

SELECT \* FROM t\_blog NATURE JOIN t\_type;

select \* from t\_blog NATURA join t\_type;

+----+-------+--------+----+------------+

| id | title | typeId | id | name |

+----+-------+--------+----+------------+

| 1 | aaa | 1 | 1 | C++ |

| 1 | aaa | 1 | 2 | C |

| 1 | aaa | 1 | 3 | Java |

| 1 | aaa | 1 | 4 | C# |

| 1 | aaa | 1 | 5 | Javascript |

| 2 | bbb | 2 | 1 | C++ |

| 2 | bbb | 2 | 2 | C |

| 2 | bbb | 2 | 3 | Java |

| 2 | bbb | 2 | 4 | C# |

| 2 | bbb | 2 | 5 | Javascript |

| 3 | ccc | 3 | 1 | C++ |

| 3 | ccc | 3 | 2 | C |

| 3 | ccc | 3 | 3 | Java |

| 3 | ccc | 3 | 4 | C# |

| 3 | ccc | 3 | 5 | Javascript |

| 4 | ddd | 4 | 1 | C++ |

| 4 | ddd | 4 | 2 | C |

| 4 | ddd | 4 | 3 | Java |

| 4 | ddd | 4 | 4 | C# |

| 4 | ddd | 4 | 5 | Javascript |

| 5 | eee | 4 | 1 | C++ |

| 5 | eee | 4 | 2 | C |

| 5 | eee | 4 | 3 | Java |

| 5 | eee | 4 | 4 | C# |

| 5 | eee | 4 | 5 | Javascript |

| 6 | fff | 3 | 1 | C++ |

| 6 | fff | 3 | 2 | C |

| 6 | fff | 3 | 3 | Java |

| 6 | fff | 3 | 4 | C# |

| 6 | fff | 3 | 5 | Javascript |

| 7 | ggg | 2 | 1 | C++ |

| 7 | ggg | 2 | 2 | C |

| 7 | ggg | 2 | 3 | Java |

| 7 | ggg | 2 | 4 | C# |

| 7 | ggg | 2 | 5 | Javascript |

| 8 | hhh | NULL | 1 | C++ |

| 8 | hhh | NULL | 2 | C |

| 8 | hhh | NULL | 3 | Java |

| 8 | hhh | NULL | 4 | C# |

| 8 | hhh | NULL | 5 | Javascript |

| 9 | iii | NULL | 1 | C++ |

| 9 | iii | NULL | 2 | C |

| 9 | iii | NULL | 3 | Java |

| 9 | iii | NULL | 4 | C# |

| 9 | iii | NULL | 5 | Javascript |

| 10 | jjj | NULL | 1 | C++ |

| 10 | jjj | NULL | 2 | C |

| 10 | jjj | NULL | 3 | Java |

| 10 | jjj | NULL | 4 | C# |

| 10 | jjj | NULL | 5 | Javascript |

+----+-------+--------+----+------------+

* + 1. 内连接：INNER JOIN

**1、内连接INNER JOIN是最常用的连接操作。**

从数学的角度讲就是求两个表的交集；

从笛卡尔积的角度讲就是从笛卡尔积中挑出ON子句条件成立的记录。

**2、有INNER JOIN，WHERE（等值连接），STRAIGHT\_JOIN,JOIN(省略INNER)四种写法。**

**3、示例：**

SELECT \* FROM t\_blog **INNER JOIN** t\_type **ON** t\_blog.typeId=t\_type.id;

SELECT \* FROM t\_blog,t\_type **WHERE** t\_blog.typeId=t\_type.id;

SELECT \* FROM t\_blog **STRAIGHT\_JOIN** t\_type ON t\_blog.typeId=t\_type.id; --注意STRIGHT\_JOIN有个下划线

SELECT \* FROM t\_blog **JOIN** t\_type **ON** t\_blog.typeId=t\_type.id;

+----+-------+--------+----+------+

| id | title | typeId | id | name |

+----+-------+--------+----+------+

| 1 | aaa | 1 | 1 | C++ |

| 2 | bbb | 2 | 2 | C |

| 7 | ggg | 2 | 2 | C |

| 3 | ccc | 3 | 3 | Java |

| 6 | fff | 3 | 3 | Java |

| 4 | ddd | 4 | 4 | C# |

| 5 | eee | 4 | 4 | C# |

+----+-------+--------+----+------+

* + 1. 左连接：LEFT JOIN

1、左连接LEFT JOIN的含义就是:求两个表的交集外加左表剩下的数据。

依旧从笛卡尔积的角度讲:就是先从笛卡尔积中挑出ON子句条件成立的记录，然后加上左表中剩余的记录（见最后三条）

2、示例：

SELECT \* FROM t\_blog LEFT JOIN t\_type ON t\_blog.typeId=t\_type.id;

+----+-------+--------+------+------+

| id | title | typeId | id | name |

+----+-------+--------+------+------+

| 1 | aaa | 1 | 1 | C++ |

| 2 | bbb | 2 | 2 | C |

| 7 | ggg | 2 | 2 | C |

| 3 | ccc | 3 | 3 | Java |

| 6 | fff | 3 | 3 | Java |

| 4 | ddd | 4 | 4 | C# |

| 5 | eee | 4 | 4 | C# |

| 8 | hhh | NULL | NULL | NULL |

| 9 | iii | NULL | NULL | NULL |

| 10 | jjj | NULL | NULL | NULL |

+----+-------+--------+------+------+

* + 1. 右连接：RIGHT JOIN

1、右连接RIGHT JOIN就是求两个表的交集外加右表剩下的数据

从笛卡尔积的角度描述，右连接就是:从笛卡尔积中挑出ON子句条件成立的记录，然后加上右表中剩余的记录（见最后一条）。

2、示例：

SELECT \* FROM t\_blog **RIGHT JOIN** t\_type ON t\_blog.typeId=t\_type.id;

+------+-------+--------+----+------------+

| id | title | typeId | id | name |

+------+-------+--------+----+------------+

| 1 | aaa | 1 | 1 | C++ |

| 2 | bbb | 2 | 2 | C |

| 3 | ccc | 3 | 3 | Java |

| 4 | ddd | 4 | 4 | C# |

| 5 | eee | 4 | 4 | C# |

| 6 | fff | 3 | 3 | Java |

| 7 | ggg | 2 | 2 | C |

| NULL | NULL | NULL | 5 | Javascript |

+------+-------+--------+----+------------+

* + 1. 外连接：OUTER JOIN

1、外连接就是求两个集合的并集。

从笛卡尔积的角度讲就是：从笛卡尔积中挑出ON子句条件成立的记录，然后加上左表中剩余的记录，最后加上右表中剩余的记录。

2、另外MySQL不支持OUTER JOIN，但是我们可以对左连接和右连接的结果做UNION操作来实现。

3、示例：

SELECT \* FROM t\_blog LEFT JOIN t\_type ON t\_blog.typeId=t\_type.id

**UNION**

SELECT \* FROM t\_blog RIGHT JOIN t\_type ON t\_blog.typeId=t\_type.id;

+------+-------+--------+------+------------+

| id | title | typeId | id | name |

+------+-------+--------+------+------------+

| 1 | aaa | 1 | 1 | C++ |

| 2 | bbb | 2 | 2 | C |

| 7 | ggg | 2 | 2 | C |

| 3 | ccc | 3 | 3 | Java |

| 6 | fff | 3 | 3 | Java |

| 4 | ddd | 4 | 4 | C# |

| 5 | eee | 4 | 4 | C# |

| 8 | hhh | NULL | NULL | NULL |

| 9 | iii | NULL | NULL | NULL |

| 10 | jjj | NULL | NULL | NULL |

| NULL | NULL | NULL | 5 | Javascript |

+------+-------+--------+------+------------+

* 1. 联合查询

在数据库中，UNION和UNION ALL关键字都是**将两个结果集合并为一个**，但这两者从使用和效率上来说都有所不同。

* + 1. Union

**UNION**在进行表链接后会筛选掉重复的记录，所以在表链接后会对所产生的结果集进行排序运算，删除重复的记录再返回结果。

实际大部分应用中是不会产生重复的记录，最常见的是过程表与历史表UNION。

如：

select \* from gc\_dfys union select \* from ls\_jg\_dfys

这个SQL在运行时先取出两个表的结果，再用排序空间进行排序删除重复的记录，最后返回结果集，如果表数据量大的话可能会导致用磁盘进行排序。

结合group by

1、如果您想使用ORDER BY或LIMIT子句来对全部UNION结果进行分类或限制，则应对单个地SELECT语句加圆括号，并把ORDER BY或LIMIT放到最后一个的后面：

(SELECT a FROM tbl\_name WHERE a=10 AND B=1)

UNION

(SELECT a FROM tbl\_name WHERE a=11 AND B=2)

ORDER BY a LIMIT 10;

2、如果你还想group by，而且还有条件，那么：

select userid from (select userid from testa union all select userid from testb) t group by userid having count(userid) = 2;

注意：在union的括号后面必须有个别名，否则会报错

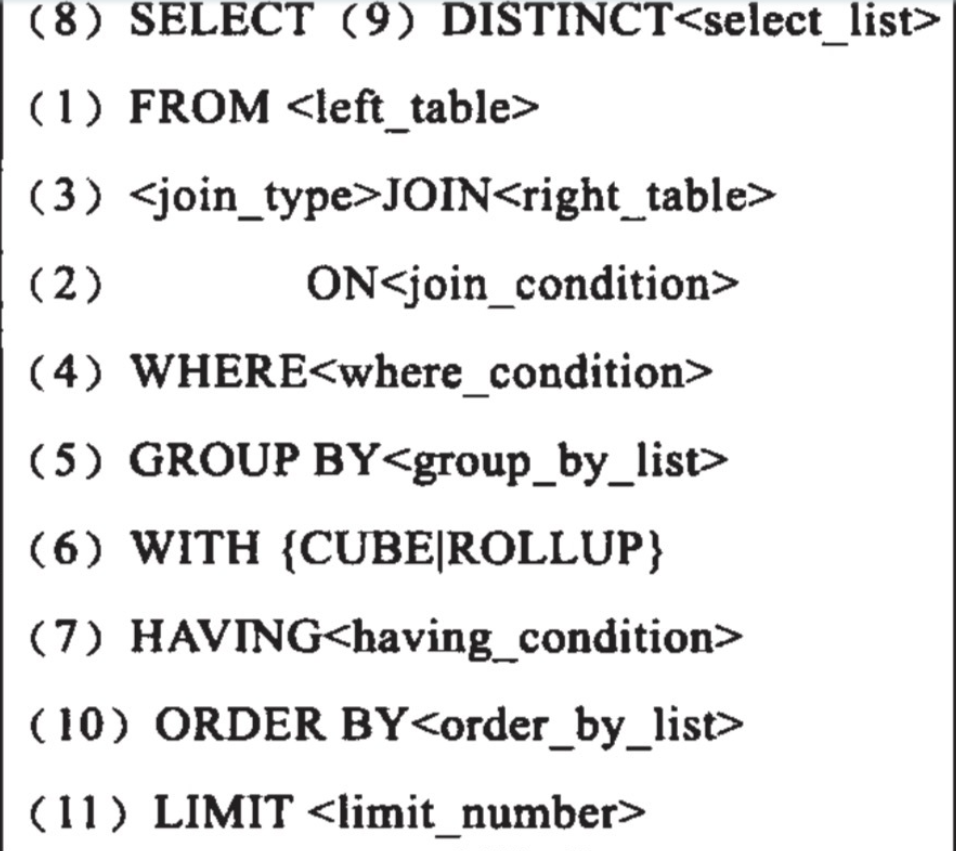
* + 1. union all

**UNION ALL**只是简单的将两个结果合并后就返回。这样，如果返回的两个结果集中有重复的数据，那么返回的结果集就会包含重复的数据了。

从效率上说，UNION ALL 要比UNION快很多，所以，如果可以确认合并的两个结果集中不包含重复的数据的话，那么就使用UNION ALL，如下：

select \* from gc\_dfys union all select \* from ls\_jg\_dfys

1. 执行顺序



(1) from：对左表left-table和右表right-table执行笛卡尔积(a\*b)，形成虚拟表VT1;

(2) on: 对虚拟表VT1根据on条件进行筛选，只有符合条件的记录才会插入到虚拟表VT2中;

(3) join: 指定out join会将未匹配行添加到VT2产生VT3,若有多张表，则会重复(1)~(3);

(4) where: 对VT3进行条件过滤，形成VT4, where条件是**从左向右执行的**;

(5) group by: 对VT4进行分组操作得到VT5;

(6) cube | rollup: 对VT5进行cube | rollup操作得到VT6;

(7) having: 对**VT6进行过滤得到VT7**;

(8) select: 执行选择操作得到VT8，本人看来VT7和VT8应该是一样的;

(9) distinct: **对VT8进行去重，得到VT9**;

(10) order by: 对VT9进行排序，得到VT10;

(11) limit: 对记录进行截取，得到VT11返回给用户。