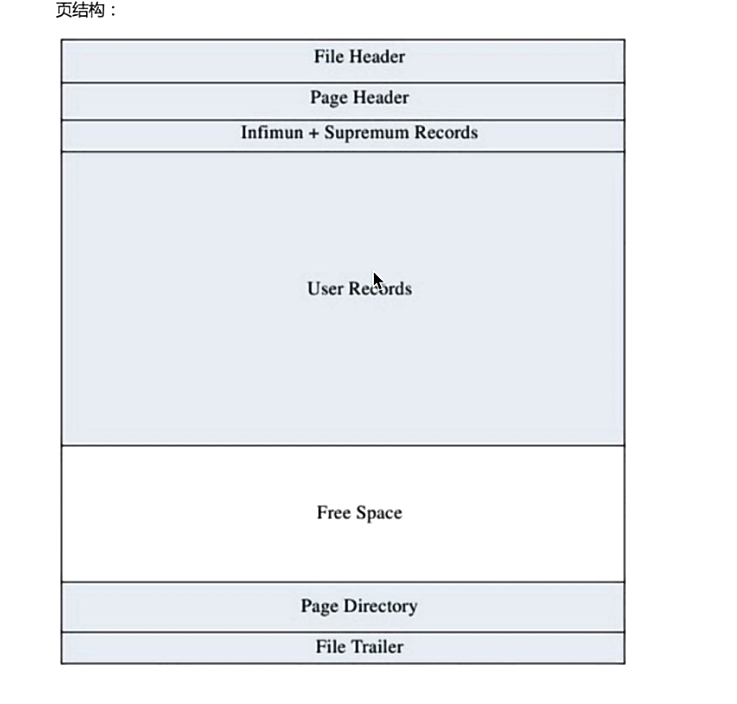
数据库的局部性原理

在程序中一行指令一行代码只需要取一个字节,但是你很有可能用到这个数据相邻的数据

Innodb页



Compact行格式

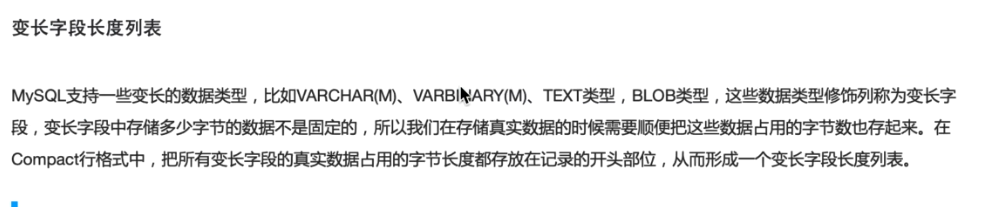


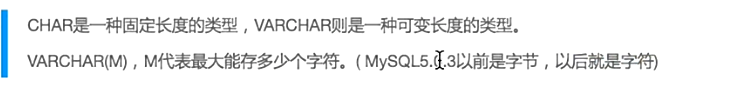
记录变长字段长度列表:当前变长内容所对应的长度,对应的字节,方便统计

存储格式--------

存储int时存储大小固定,,4字节

存储varchar 存储不固定,

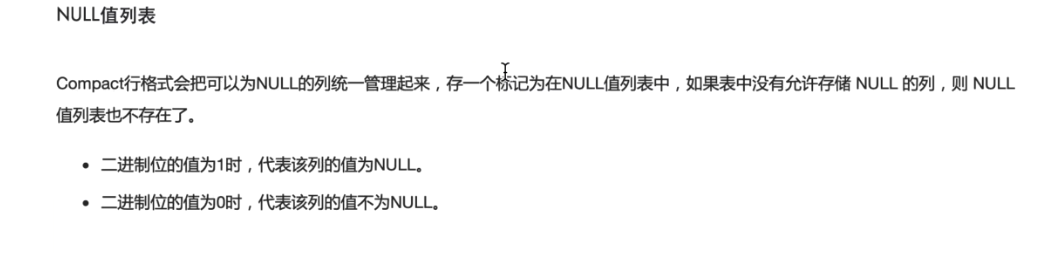




Null标志位 :记录空的位置

存数据时空值会被占用上,

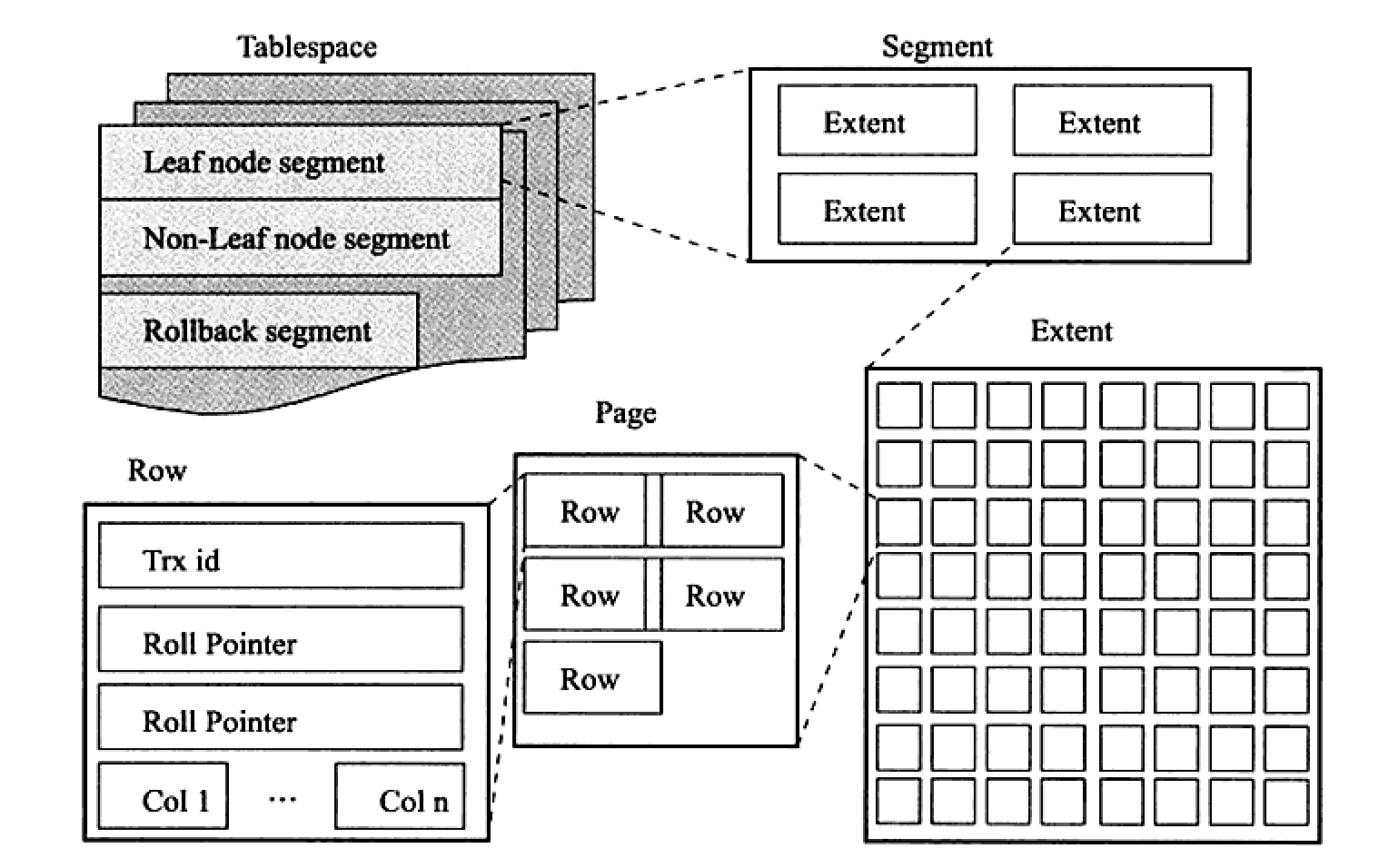




# 表的逻辑存储结构

在InnoDB存储引擎上，表都是根据主键顺序组织存放的，这种存储方式的表称为**索引组织表（index organized table）**，如果一个表没有指定主键，那么会：

* 首先判断是否存在唯一非空索引，如果有则为主键。
* 否则InnoDB会自动创建一个6个字节大小的指针。

所有的数据都被逻辑的存放在一个空间里，成为**表空间**。表空间又由段（Segment）、区（extent）、页（page）组成。  


* **表空间**  
  表空间分为独立表空间和共享表空间。如果启用了独立表空间，那么会存放每张表的数据、索引和插入缓冲Bitmap页，而其他数据，如回滚信息，插入缓冲索引页、系统事务信息、二次写缓冲仍然放在原来的共享表空间中。
* **段**  
  常见的段有数据段、索引段、回滚段等。InnoDB是索引组织（index organized）的，因此数据段即为B+树的叶子节点，索引段即为非叶节点；回滚段比较特殊，以后会介绍到。
* **区**  
  由连续页组成，每个区都是1MB。
* **页**  
  默认大小16KB，在InnoDB 1.2.x开始，可以是4K、8K、16K，设置完成后所有页的大小都相同。

# Mysql基础

## 语句练习

1去重 ----distinct ---dei si ting kei de

SELECT DISTINCT roles from role\_permission

索引:帮助mysql高效查询的数据结构

## Innodb存储引擎----总结

1.InnoDB是事务型数据库的首选引擎，支持事务安全表（ACID）

事务的ACID属性：即原子性、一致性、隔离性、持久性

                            a.原子性：原子性也就是说这组语句要么全部执行，要么全部不执行，如果事务执行到一半出现错误，数据库就要回滚到事务开始执行的地方。原子性是指事务是一个**不可再分割的工作单元**，事务中的操作要么都发生，要么都不发生。

                             实现：主要是基于MySQ日志系统的redo和undo机制。事务是一组SQL语句，里面有选择，查询、删除等功能。每条语句执行会有一个节点。例如，删除语句执行后，在事务中有个记录保存下来，这个记录中储存了我们什么时候做了什么事。如果出错了，就会回滚到原来的位置，redo里面已经存储了我做过什么事了，然后逆向执行一遍就可以了。

                                b.一致性：事务开始前和结束后，数据库的完整性约束没有被破坏。（eg:比如A向B转账，不可能A扣了钱，B却没有收到）

                                c.隔离性：同一时间，只允许一个事务请求同一数据，不同的事务之间彼此没有任何干扰(由并发事务所做的修改必须与其他事务所做的修改隔离,事务查看数据所处的状态,要么是并发事务修改它之前的状态,要么是另一个并发事务修改它之后的状态,事务不会查看中间状态的数据)；

                                如果不考虑隔离性则会出现几个问题：

                                 i、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）；（读取了另一个事务未提交的脏数据）

                                 ii、不可重复读：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；（读取了前一个事务提交的数据，查询的都是同一个数据项）

                                 iii、虚读（幻读）：指一个线程中的事务读取到了另一个线程中提交的数据,,,,,,是事务非独立执行时发生的一种现象（eg: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）；（读取了前一个事务提交的数据，针对一批数据整体）

                                d.持久性：事务完成后，事务对数据库的所有更新将被保存到数据库，不能回滚 事务完成后,他对于系统的影响是永久性的,这个修改即使出现系统故障也会一直保持一致.

2.InnoDB是mySQL默认的存储引擎，默认的隔离级别是RR(可重复读)的隔离级别下更近一步，通过多版本并发控制（MVCC）解决不可重复读问题，加上间隙锁（也就是并发控制）解决幻读问题。因此InnoDB的RR隔离级别其实实现了串行化级别的效果，而保留了比较好的并发性能。

MySQL数据库为我们提供的四种隔离级别：

a、Serializable（串行化）：可避免脏读、不可重复读、幻读的发生；

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

c、Read committed（读已提交）：可避免脏读的发生；

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

从a----d隔离级别由高到低，级别越高，执行效率越低

3.InnoDB支持行级锁。行级锁可以最大程度的支持并发，行级锁是由存储引擎层实现的。

锁：锁的主要作用是管理共享资源的并发访问，用于实现事务的隔离性

        类型：共享锁（读锁）、独占锁（写锁)

        MySQL锁的力度：表级锁（开销小、并发性低），通常在服务器层实现

                                    行级锁（开销大、并发性高），只会在存储引擎层面进行实现

4、InnoDB是为处理巨大数据量的最大性能设计。它的CPU效率可能是任何

基于磁盘的关系型数据库引擎所不能匹敌的(处理大量数据效率快)

5、InnoDB存储引擎完全与MySQL服务器整合，InnoDB存储引擎为在主内存中缓存数据和索引而维持它自己的缓冲池。InnoDB将它的表和索引在一个逻辑表空间中，表空间可以包含数个文件（或原始磁盘文件）；

6、InnoDB支持外键完整性约束，存储表中的数据时，每张表的存储都按照主键顺序存放，如果没有显示在表定义时指定主键。InnoDB会为每一行生成一个6字节的ROWID,并以此作为主键

7、InnoDB被用在众多需要高性能的大型数据库站点上

8、InnoDB中不保存表的行数（eg:select count(\*)from table时，InnoDB需要扫描一遍整个表来计算有多少行）；清空整个表时，InnoDB是一行一行的删除，效率非常慢；

9.InnoDB不创建目录，使用InnoDB时，MySQL将在MySQL数据目录下创建一个名为ibdata1的10MB大小的自动扩展数据文件，以及两个名为ib\_logfile0和ib\_logfile1的5MB大小的日志文件

二、InnoDB引擎的底层实现

InnoDB的存储文件有两个，后缀名分别是 .frm和 .idb；其中 .frm是表的定义文件， .idb是表的数据文件。

1、InnoDB引擎采用B+Tree结构来作为索引结构

B-Tree（平衡多路查找树）：为磁盘等外存储设备设计的一种平衡查找树

系统从磁盘读取数据到内存时是以磁盘块位基本单位的，位于同一磁盘块中的数据会被一次性读取出来，而不是按需读取。

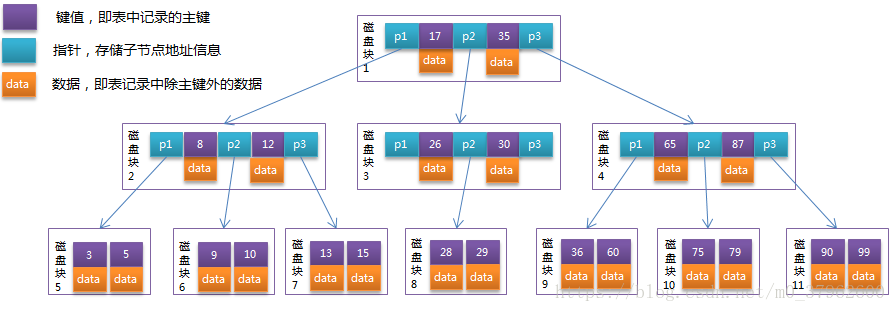
InnoDB存储引擎使用页作为数据读取单位，页是其磁盘管理的最小单位，默认page大小是16k.

系统的一个磁盘块的存储空间往往没有那么大，因此InnoDB每次申请磁盘空间时都会是若干地址连续磁盘块来达到页的大小16KB。

InnoDB在把磁盘数据读入到磁盘时会以页为基本单位，在查询数据时，如果一个页中的每条数据都能助于定位数据记录的位置，这将会减少磁盘I/O的次数，提高查询效率。

B-Tree结构的数据可以让系统高效的找到数据所在的磁盘块

B-Tree中的每个节点根据实际情况可以包含大量的关键字信息和分支，例：



每个节点占用一个盘块的磁盘空间，一个节点上有两个升序排序的关键字和三个指向子树根节点的指针，指针存储的是子节点所在磁盘块的地址。

以根节点为例，关键字为17和35，P1指针指向的子树的数据范围小于17，P2指针指向的子树的数据范围为17----35，P3指针指向的子树的数据范围大于35；

模拟查找关键字29的过程：

a.根据根节点找到磁盘块1，读入内存。【磁盘I/O操作第一次】

b.比较关键字29在区间（17,35），找到磁盘块1的指针P2;

c.根据P2指针找到磁盘块3，读入内存。【磁盘I/O操作第二次】

d.比较关键字29在区间（26,30），找到磁盘块3的指针P2；

e.根据P2指针找到磁盘块8，读入内存。【磁盘I/O操作第三次】

f.在磁盘块8中的关键字列表中找到关键字29.

MySQL的InnoDB存储引擎在设计时是将根节点常驻内存的，因此力求达到树的深度不超过3，也就是I/O不需要超过三次；

分析上面的结果，发现需要三次磁盘I/O操作，和三次内存查找操作。由于内存中的关键字是一个有序表结构，可以利用二分法查找提高效率；而三次磁盘I/O操作时影响整个B-Tree查找效率的决定因素。

B+Tree

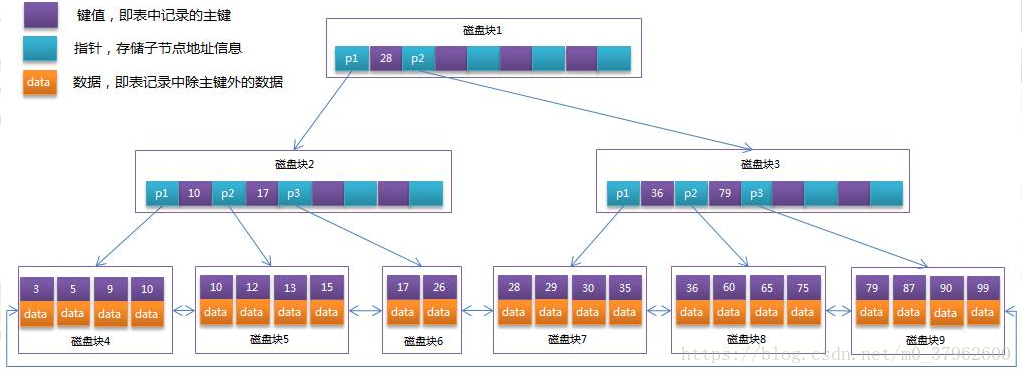
B+Tree是在B-Tree基础上的一种优化，使其更适合实现外存储索引结构，B-Tree中每个节点中有key，也有data，而每一页的存储空间是有限的，如果data数据较大时将会导致每个节点（即一个页）能存储的key的数量很小。当存储的数据量很大时同样会导致B-Tree的深度较大，增大查询时的磁盘I/O次数，进而影响查询效率。

在B+Tree中所有数据记录节点都是按照键值大小顺序存放在同一层的叶子节点上，而非叶子节点上只存储key值信息，这样可以大大加大每个节点存储的key值数量，降低B+Tree的高度；

B+Tree在B-Tree的基础上有两点变化：（1）数据是存在叶子节点中的

                                                          （2）数据节点之间是有指针指向的

由于B+Tree的非叶子节点只存储键值信息，假设每个磁盘块能存储4个键值及指针信息，则变成B+Tree后其结构如下图所示：



通常在B+Tree上有两个头指针，一个指向根节点，另一个指向关键字最小的叶子节点，而且所有叶子节点（即数据节点）之间是一种链式环结构。

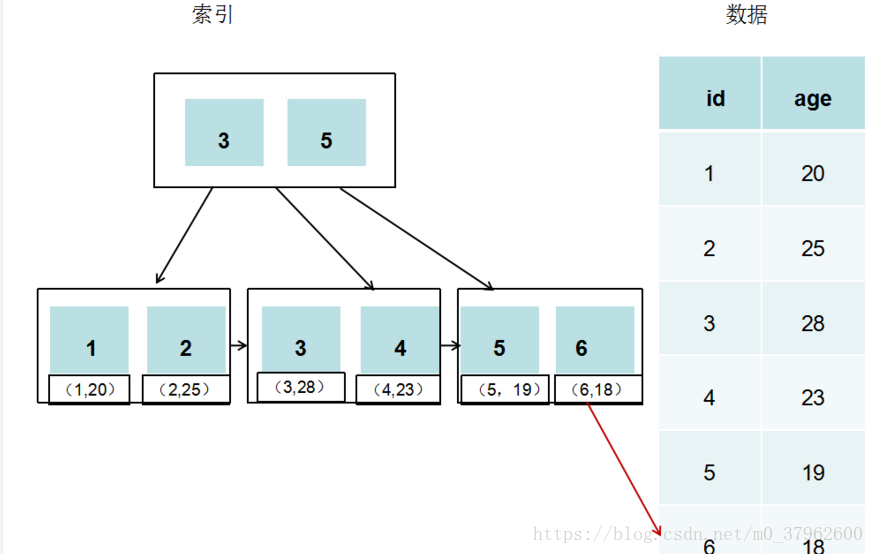
因此可以对B+Tree进行两种查找运算，一种是对于主键的范围查找和分页查找，另一种是从根节点开始，进行随机查找。

InnoDB中的B+Tree

InnoDB是以ID为索引的数据存储

采用InnoDB引擎的数据存储文件有两个，一个定义文件，一个是数据文件。

InnoDB通过B+Tree结构对ID建索引，然后在叶子节点中存储记录



若建立索引的字段不是主键ID，则对该字段建索引，然后在叶子节点中存储的是该记录的主键，然后通过主键索引找到对应记录