接下来由我们组来进行数据库优化的讲解...通过一周的学习我们数据库优化的结果为

再如下表结构中

Student Class Spec Dept Teacher Course

一亿学生, 一千万班级, 一百万专业, 十万个系, 五百万个教师, 一百万个课程,

Teaching

每个教师至少授课一门,每个课程至少有一个教师教,二千五百万授课信息,

Learn

每个学生至少上一门课,每门课至少有一个学生上,三亿条上课信息,

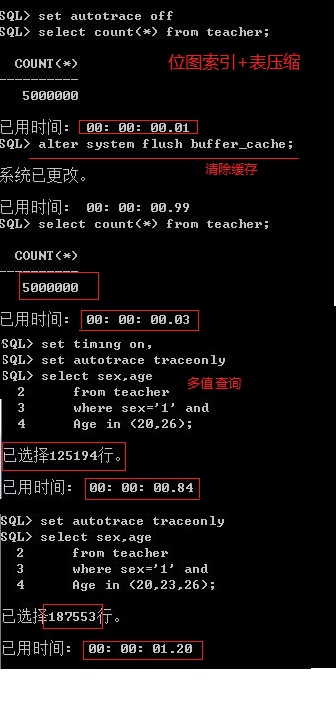
AttendCourse

每个教师至少上课一个班,每个班至少一个教师上课,七千万条数据

500w 查询count<\*>总记录数时间为0.01S 将缓存清除时间为0.03S

多值查询500W中查询出12W 行0.84S

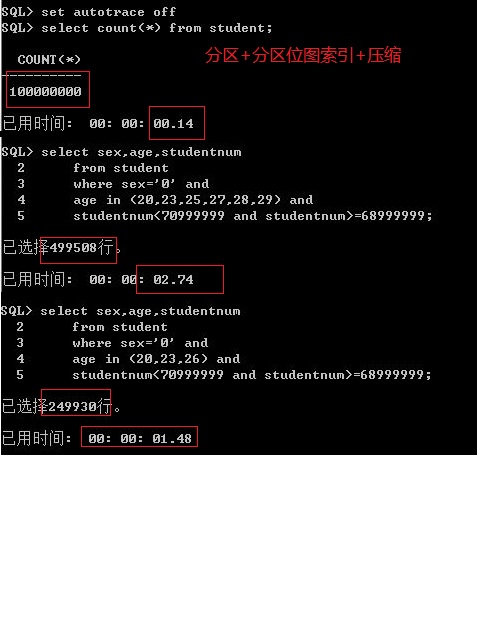
扩大范围（age中多加一条）的多值查询500W中的18W的1.20S



扩值为1亿 count<\*>总记录数0.14S

多值查询1亿中的50W 2.74S

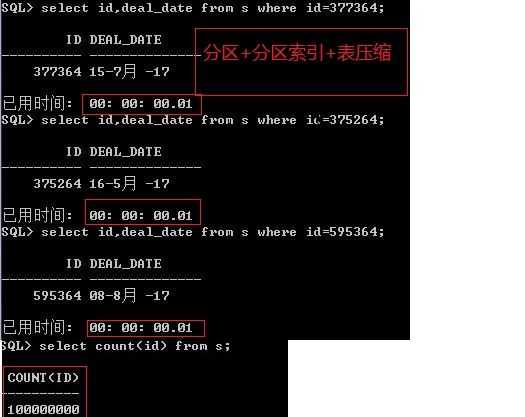
缩小age 1亿中查询出25W 1.48S



单表查询亿级数据的200W行6.39S



单表单值查询 我们换了3个不同的值查询三次均为0.01秒



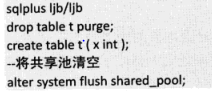
简而言之数据库优化无非就是让你从持久转变为快男的过程，作为四个男人我们在7天中，一次比一次快...

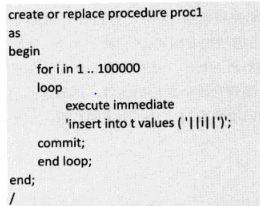
当然了想要快速的查询数据，你首先要有数据，那么数据该怎么插入到数据库中呢

话不过说直接引入第一次优化---在数据库中如何让你插的更快...出张图

这是我们作案解析的小表t 内含10万条数据（数据包括。。。）

当我们执行以下代码时.发现





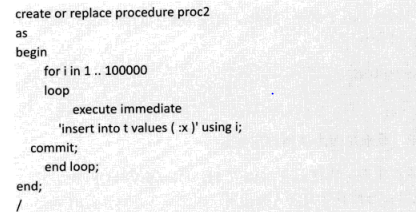


当我们执行一下代码发现，对于我们不做优化的代码而言，插入10万条数据竟然需要42秒 平均2000条/秒

对于我们的亿级数据而言，这个速度简直太慢了，对于我们项目而言可能单单的将表完整的建成就需要14个小时...

不过通过仔细观察研究我们发现共享池中有大量的SQL语句，而SQL\_ID各不相同，每个语句都是解析一次执行一次，也就是说本次插入解析了10万次

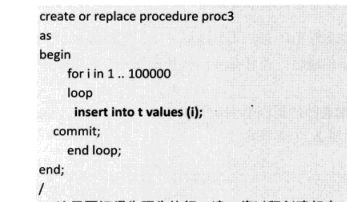
那么怎么优化呢，我们通过绑定变量的方法，将10万条语句改成被HASH成一个SQL\_ID,也就是说解析1次执行10万次，我们再来看结果

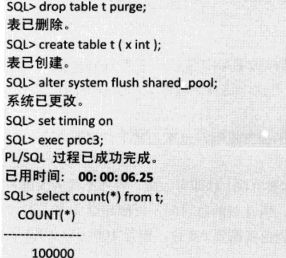




这次我们惊奇的发现时间从42.87秒提升到8.41秒，这真的让我们四个男人第一感受到了快也是一种欣喜。

我们继续看这段代码 如果将动态（execute immediate）改为静态(insert into t values(i))那又会如何-----------------静态改写





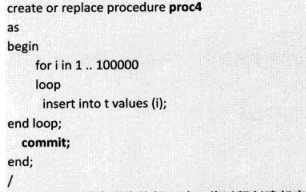
时间进一步缩小到6秒

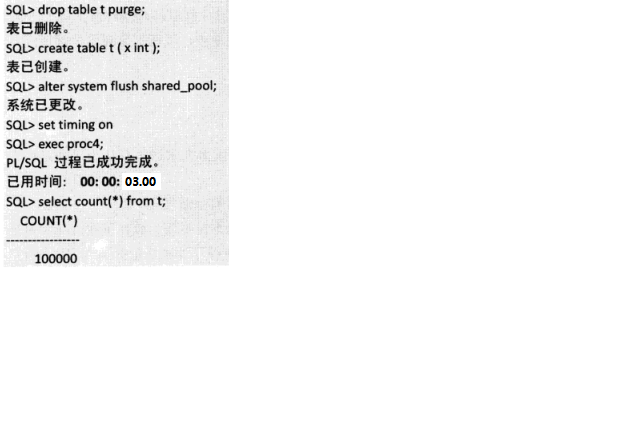
此时单纯的快已经不让我们四个男人感觉到羞耻了，也许在未来的社会快也是一种荣耀

既然已经不要脸了那我们就继续的不要脸下去

当我们进一步的探索，发现commit提交它放在了循环之中，也就是说10万次的插入，需要10万次的提交，如果将commit放在循环外面仅提交一次试一试

将代码改为批量提交





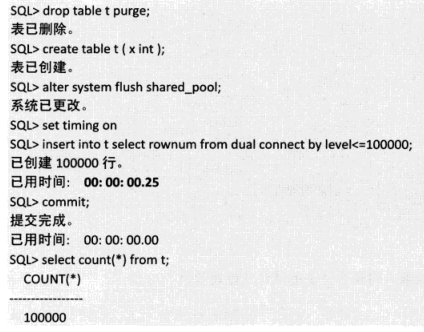
我去，3秒！不愧是我谢兄给我的数据...短小快

好吧，让我们尴尬的继续。

如果我们将语句以集合的写法应用如下

Insert into t select rownum from dual connect by level<=100000;

我们再来看一下结果

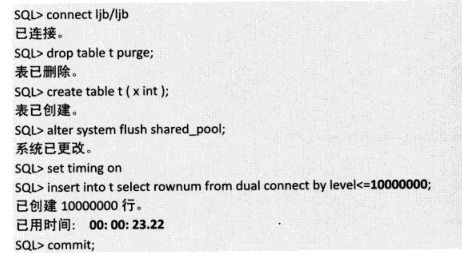


时间进一步优化为0.25秒约为40万/s 与一开始的2000条每秒已经形成巨大提升了。

那么快的脚步还会继续吗？不好意思，我们的谢兄就要恬不知耻的快下去。。。我们继续

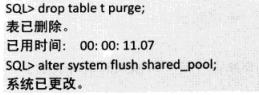
因为刚才的结果已经达到0.几秒了，所以10w的数据已经满足不了我们谢兄不要脸的深度了

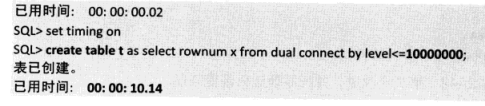
接下来我们扩充100倍，改成千万级数据



发现插入1千万条记录为23秒多大致为43w/s,和10W条记录的40W/s的速度大体接近，

下面我们继续优化，改用create table 的直接路径方式来新建t表，看看结果





速度又提升了两倍只需要10秒便可完成

其实这个速度还是可以继续快下去的

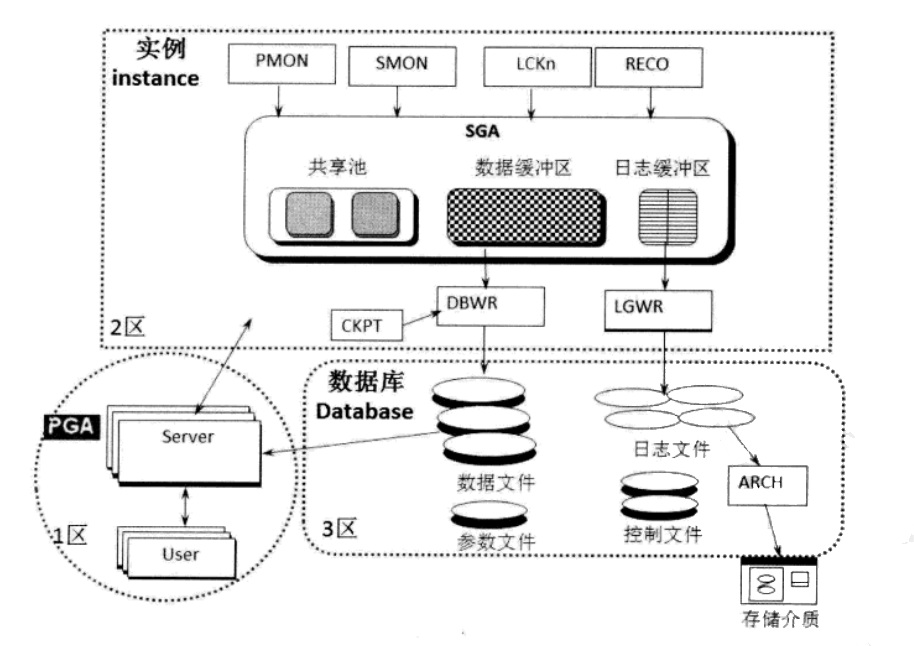
当我们使用并行设置时 我们关闭日志 nologging 并且设置parallel 16表示用到机器的16个CPU，这个再笔记本的收效不是很明显，因为我们的环境时单核的机器

通过资料借鉴 如果将它运行再服务器上，有了16个CPU 大致结果为3秒左右（怎么又是3秒，好尴尬）我们继续 它的素的超过300条/s插入速度

以上就是我们对插入优化的大体讲解，但是对于亿级数据的查询，单单的修改插入远远不够

接下来进入我们本次研究的重点：索引

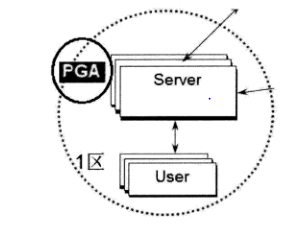
**但是在讲解索引之前我想加一个oracle的物理结构让我们直观的了解到oracle查询数据的底层走动**

****

**我们通过一个小小的例子来对这张物理结构图进行一个小小小小的讲解**

**大家来看这个SQL语句select object\_name from t where object\_id=29;**

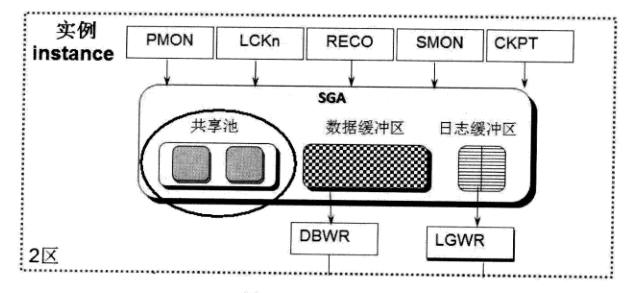
**当我们发出一条上诉SQL指令后，该SQL语句从1区先做准备工作**



**在这里完成了用户连接信息的保存和权限的保存**

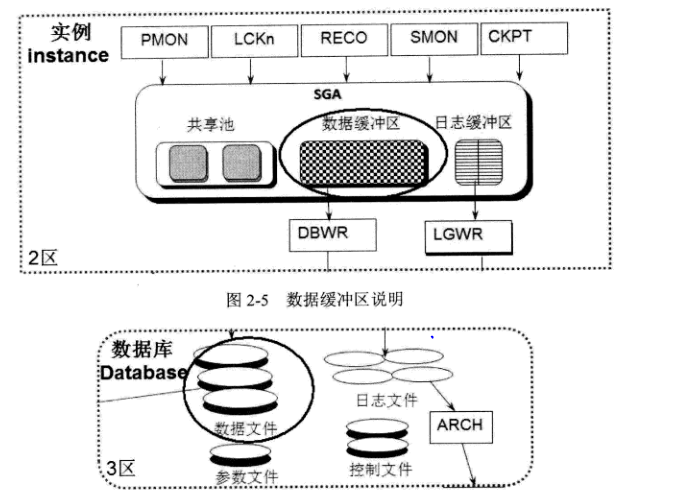
**不要小看这个保存的重要性，只要该SESSION不断开连接，下次系统不用再去硬盘中读取数据，而是直接从PGA内存区中获取**

**此外该SQL还会理解匹配一条唯一的HASH值，接下来该SQL指令进去2区进行处理，首先敲开SGA的共享池的大门，准备登门拜访，如图**

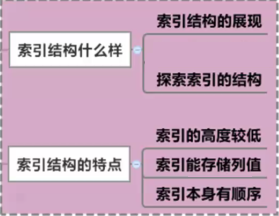


**共享池的大门打开了，该SQL先在房内查询是否什么地方有存储过这个SQL指令的身份证（就是那个唯一的HASH值），如果没有，就会首先查询自己的语句语法是否正确，语义是否正确，是否有权限，一切通过便生成这条语句的身份证，唯一的HASH值就会存储下来了。接下来进行解析。那么解析什么呢？比如select object\_name from t where object\_id=29这个语句，在object\_id列有索引的情况下，是用索引更高效还是全表扫描更高效，Oracle要作出选择。**

**接下来SQL指令会直奔数据缓冲区，数据缓冲区会立即根据ID列的索引从t表中查找object\_id值为29的数据，如果找不到就交给它的下一级去更远的Database的数据文件区中寻找。如果还找不到，那就我真满足不了你了，你爱找谁找谁吧**

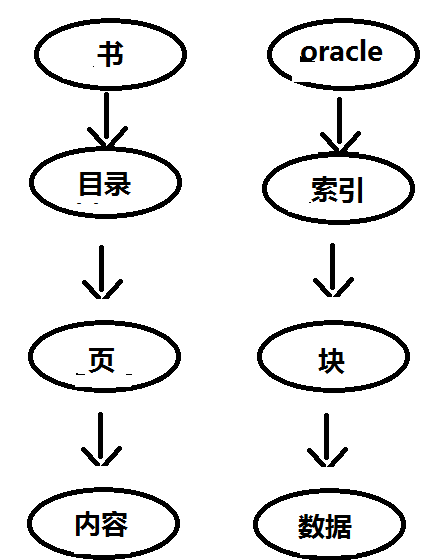


我们通过以下几个模块来对索引进行具体解析

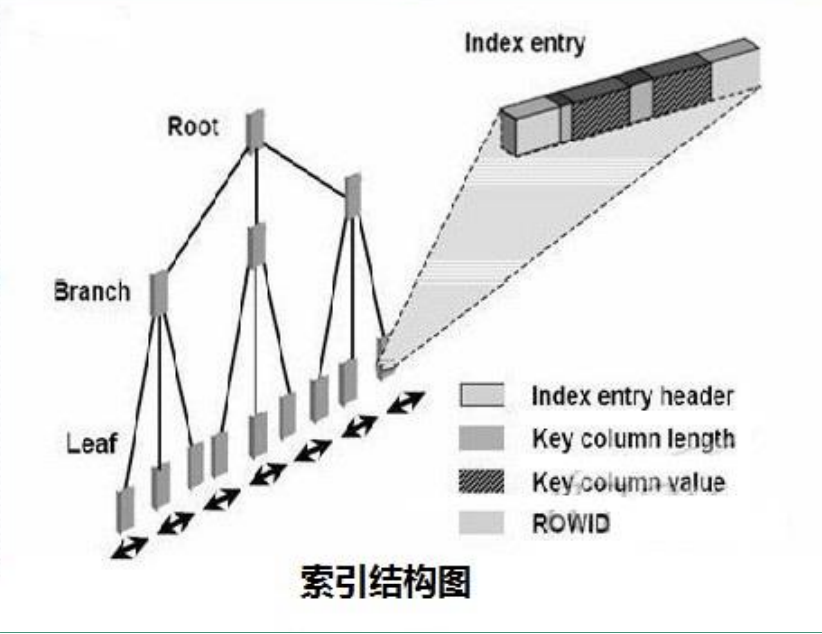


首先我们来探析一下索引究竟是什么？

我们形象的比喻把ORACLE当作一本书，里面的数据组成了书的每一页（在oracle中把书中的页称为块（block）），而书的目录就是索引，我们通过目录可以快速的查找的书中想要找的那一页然后获得那一页中的内容，oracle也一样，通过索引来获得我们块（block）,在块中找到我们的数据



索引结构的展现：



Root 根

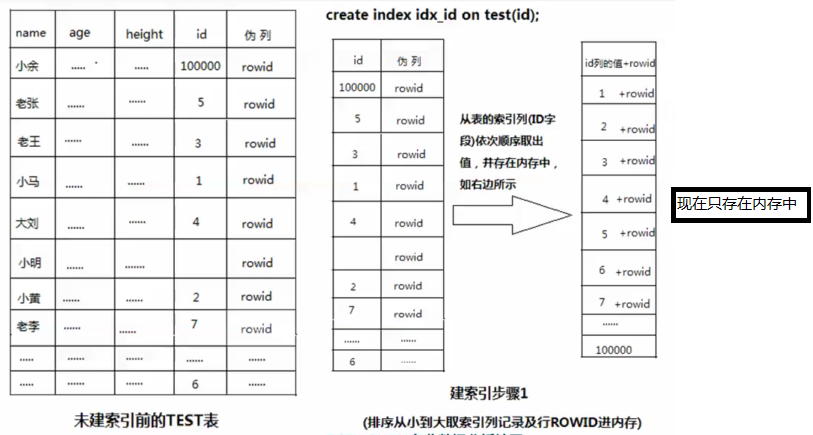
Branch 茎

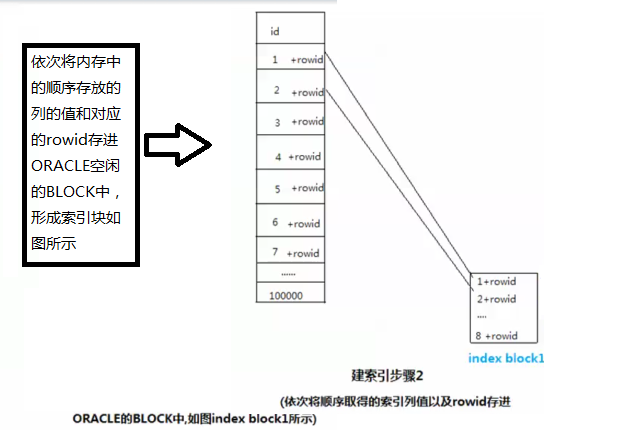
Leaf 叶

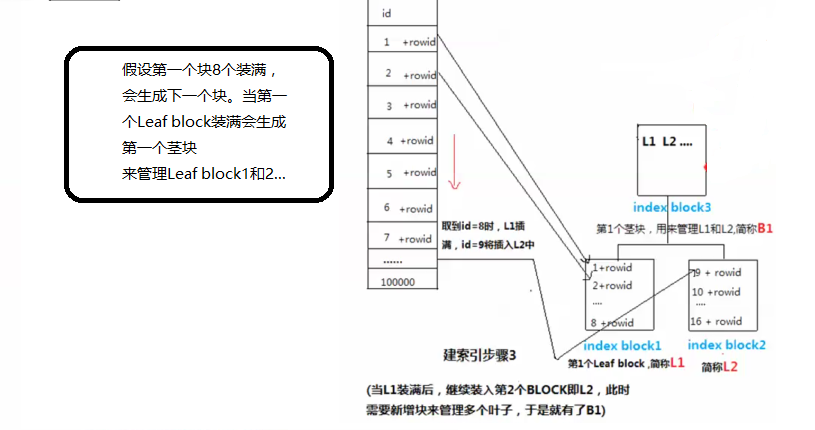
Key column value 列的值

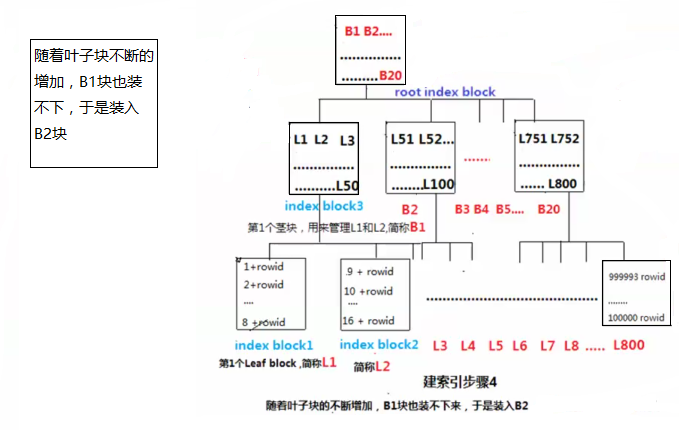
Key column length 列的长度

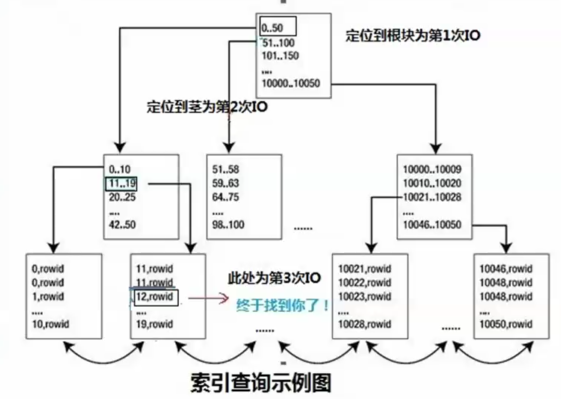
探索索引的结构











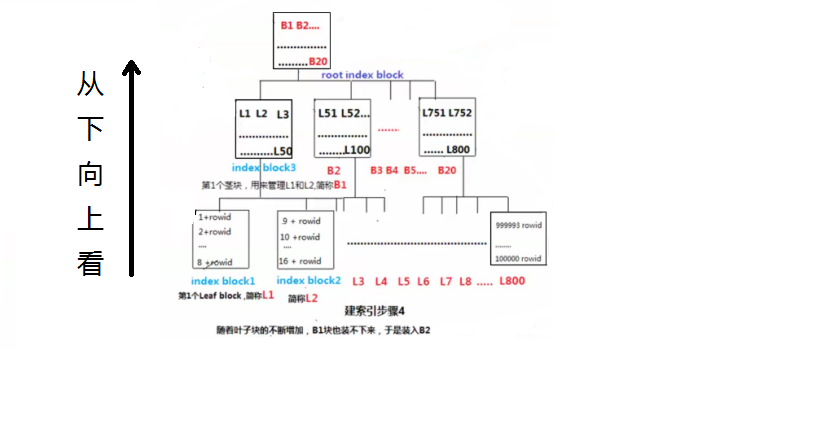
以上使我们通过原理来解析索引的优点，单单的原理介绍是展示不了它的强大的，让我们通过索引结构的三大特性引出各个小案例让我们进一步体会。。。

那么索引结构究竟有那三大特性，在这里

**索引结构的三大特性**



索引的三大特性：高度较低，存储列值，本身有序



**与脚本一起讲解逻辑读的概念**

**逻辑读在Oracle调优中有四个好处：**

**（1）逻辑读是受制于CPU能力的操作，因而，很好的反映了CPU的使用情况。**

**（2）逻辑读可能导致物理读，因而，通过减少逻辑读的数量，很可能会降低I/O操作次数。**

**（3）逻辑读是受制于串行的操作，既然经常要考虑多用户负载的优化，最小化逻辑读将有利于避免扩展性问题。**

**（4）逻辑读的数量可以通过SQL跟踪文件和动态性能视图在SQL语句以及执行计划级别获得。**

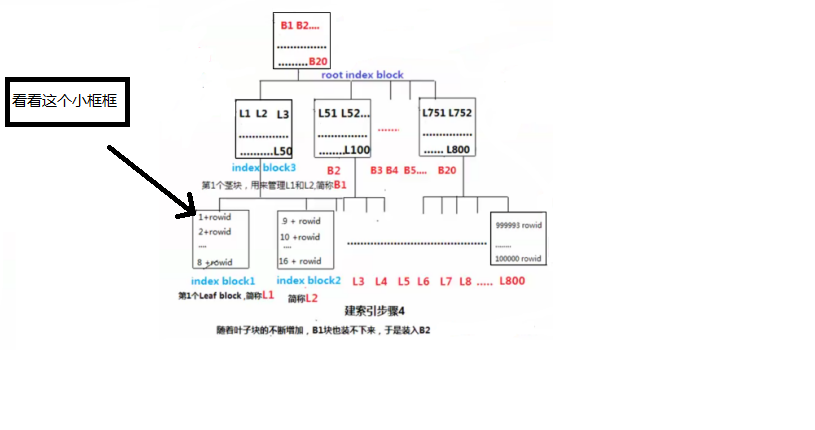
**数据块是oracle最基本的读写单位,但用户所需要的数据,并不是整个块,而是块中的行,或列.当用户发出SQL语句时,此语句被解析执行完毕，就开始了数据的抓取阶段,在此阶段,服务器进程会先将行所在的数据块从数据文件中读入buffer cache,这个过程叫做物理读.物理读,每读取一个块,就算一次物理读.当块被送进buffer cache后,并不能立即将块传给用户,因为用户所需要的并不是整个块,而是块中的行.从buffer cache的块中读取行的过程,就是逻辑读.为了完成一次逻辑读,服务器进程先要在hash表中查找块所在的buffer cache 链.找到之后,需要在这个链上加一个cache buffer chains 闩,加闩成功之后,就在这个链中寻找指定的块,并在块上加一个pin锁.并释放cache buffer chains闩.然后就可以访问块中的行了.服务器进程不会将块中所有满足条件的行一次取出,而是根据你的抓取命令,每次取一定数量的行.这些行取出之后,会经由PGA传给客户端用户.行一旦从buffer cache中取出,会话要释放掉在块上所加的PIN.本次逻辑读就算结束.如果还要再抓取块中剩余的行,服务器进程要再次申请获得cache bufffer链闩.再次在块上加PIN.这就算是另外一次逻辑读咯.也就是说,服务器进程每申请一次cache buffer链闩,就是一次逻辑读.而每次逻辑读所读取的行的数量,可以在抓取命令中进行设置.  
逻辑读和Cache buffer chains闩关系密切，TOM曾有文章提到，进程每申请一次Cache buffer chains闩，就是一次逻辑读。但是，逻辑读并不等同于Cache buffer chains闩，每次逻辑读，在9i中至少需要获得两Cache buffer chains闩。逻辑读是指在Hash表中定位块的这个过程。**

脚本一

 **打开（引入逻辑读）**

脚本二

 **打开**



索引特性2存储列值

 **粗略**

脚本一索引特性之存列值优化count

索引是由列值和rowId组成的

这个脚本要说明一点注意事项，索引不能存储空值的所以会出现在建立索引后不走索引的情况

所以我们要加上is not null 就可以解决了

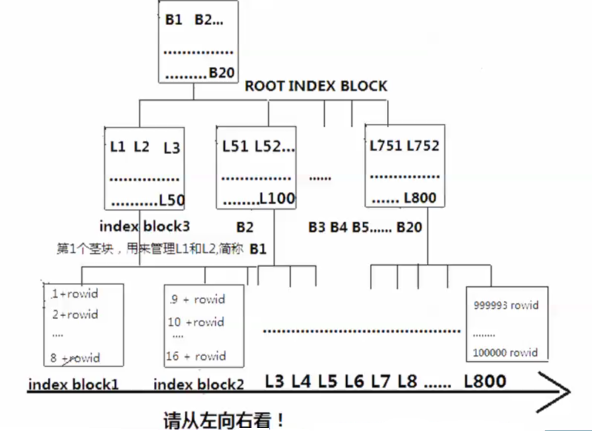
如果是主键就无需定义列是否允许为空了

无论如何都走索引

脚本二索引特性之存列值优化sum avg

**粗略**

索引特性3 索引本身有序



1--8 9--16...从小到大 从左到右

脚本1索引特性之有序优化order by

**粗略**

因为索引本身就排序了 所以当执行order by时oracel会发生查询转换，忽略了这个排序

索引也会提升速度

把\*改成object\_id

多了一步 TABLE ACCESS BY INDEX ROWID 在表中找object\_id列以外字段一同展现出来

object\_id 直接找 不需要找表中的其它字段

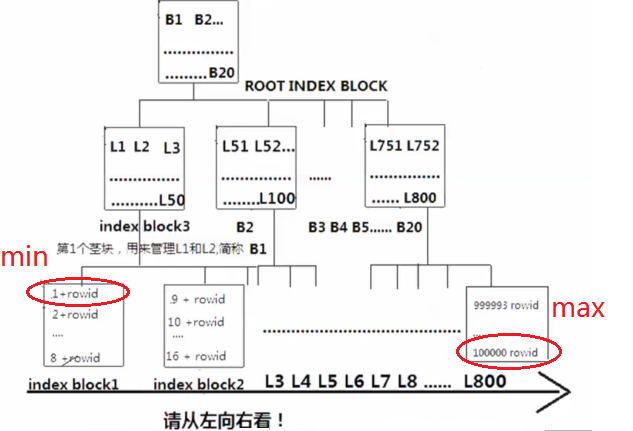
CPU 1000+ 到177

假如我们需要表中object\_id 会走索引 因为又可以避免排序，又不需要回表

脚本2 索引特性之有序和存储列值共同优化max

 **打开** **粗略**

最小的还是最小的 最大的还是最大的



聚合因子

**粗略**

第一张表colocated 有序的 增加主键啦 1---100000 收集统计信息

第二张表是从一张表来的 但是是无需的 顺序非常乱

两者性能

第一个为2900buffers 理解为逻辑读

第二个为2万+buffers 逻辑度如此巨大

所以说索引的顺序和表的顺序关联非常大

COLOCATED\_PK 有1190不再一起

DISORGANIZED\_PK 99899不再一起

**拓展：**

组合索引

适合在单独chauxn返回记录很多，组合查询后记录很少：

回表 \* from t 必然要回表

Object\_id from t 必然不要回表

取两个字段 如果建的是objec\_id的索引 查不到name的信息 所以在查name的时候必然回表 如果建联合索引 会更小 因为查询角度不同 所以不建议使用 更新程度太大。。。也会造成一系列问题 ，会把表撑大等等原因

**粗略**

组合索引经要素!

/\* 1.适用在单独查询返回记录很多，组合查询后忽然返回记录很少的情况:

比如where 学历=硕士以上 返回不少的记录

比如where 职业=收银员 同样返回不少的记录

于是无论哪个条件查询做索引，都不合适。

可是，如果学历为硕士以上，同时职业又是收银员的，返回的就少之又少了。

于是联合索引就可以这么开始建了。

\*/

--2.仅等值无范围查询时，组合索引顺序不影响性能（比如where col1=xxx and col2=xxx，无论COL1+COL2组合还是COL2+COL1组合）

组合索引顺序不影响性能

结论

1. 适用在单独查询返回很多，组合查询返回很少
2. 组合查询的组合顺序，要全面考虑单列查询情况
3. 仅等值无范围查询时，组合索引顺序不影响性能
4. 组合索引最佳顺序一般是将列等值查询的列置前
5. 注意组合索引与组合条件中关于IN的优化

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

位图索引

用在某个列的值非常的少如性别，

**粗略****粗略**

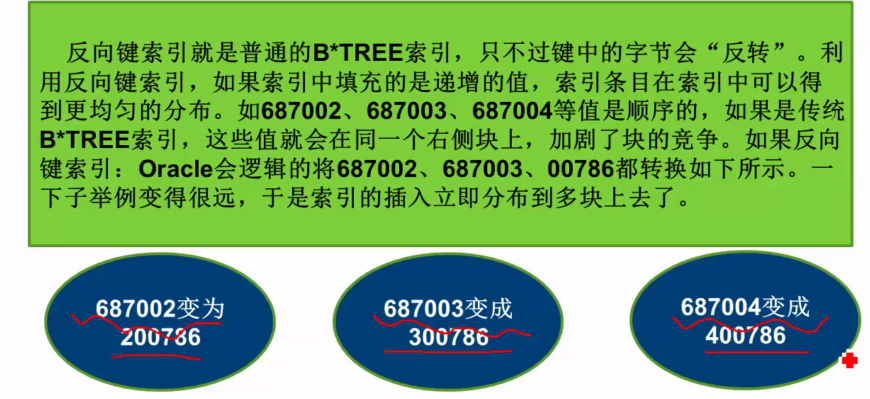
修改过于频繁不适合，值特别多不适合

函数索引

基于函数的索引时将一个函数计算得到的结果存储再行的列中，而不是存储列数据本省。可以把基于函数的索引看作一个虚拟列上的索引（这个列不是物理的存储在表中）。

反向键索引

反向键索引就是普通的B\*TREE索引，只不过键中的字节会“反转”。利用反向键索引，如果索引中填充的是递增的值，索引条目在索引中可以得到更均匀的分布。如687002，687003，687004等值是顺序的，如果是传统的B\*TREE索引，这些值就会再同一个右侧块上，加剧了块的竞争。如果反向键索引：Oracle会逻辑的将687002，00786都转换如下所示。一下子举例变得很远，于是索引的插入立即分布到多块上去了



1. B树索引(B-Tree index)  
                 适用范围：1）OLTP系统中 **略或粗略**

                                  2）主键、唯一性约束

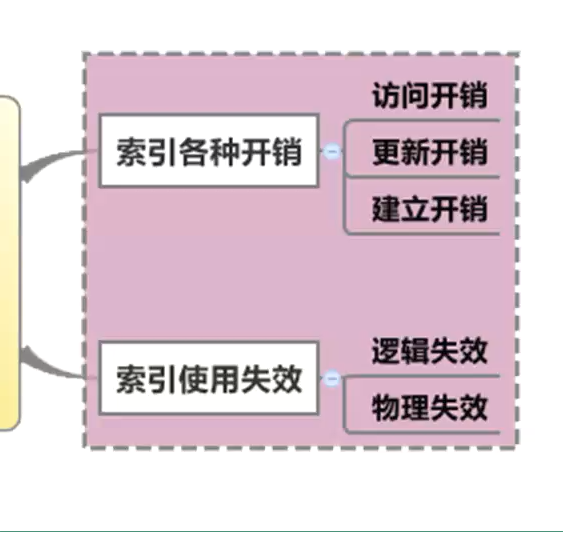
                                  3）重复度非常低的字段  
               不适用范围：重复度非常高的字段  
               特性：1）执行效率非常高  
                          2）查询时间基本不受数据量的影响  
                          3）如果重复度非常高，使用B-Tree索引还没有全盘扫描效率高

1. 位图索引(Bitmap index)  
                适用范围：1）OLAP系统中  
                                  2）主要是重复数据导致的查询性能低的问题，用来弥补B-Tree性能低的问题  
                                  3）使用OR\AND这类逻辑操作  
                不适用范围：大量DML操作、重复率较低的字段

              特性：1）适合在索引字段间进行OR\AND这类逻辑运算  
                         2）位图索引占用的空间比BTree索引用的少的多  
                         3）位图索引不适用于大量的DML操作（除select外），因为每个DML操作对应的位图索引键值对应多条数据，修改时键值会被上锁，如果此时其他用户需要更改数据，就会造成阻塞。

1. 全文索引(Text index)  
                适用范围：1）模糊查询  
                                  2）数据库空间种族，因为全文索引比较占用空间  
                                  3）语言文字范畴  
                不适用范围：全文索引占用大量的空间，不适合在硬盘资源紧张的库中（原数据的1.5倍）  
                                      BUG比较多，维护困难  
                        特性：1）适用于不变的数据，因为DML的新数据，不会在全文索引中包含  
                                   2）由于全文索引不是基于事物的，一旦有DML，就需要重新构建这个全文索引：alter index t\_idx rebuild parmeters('sync');

索引的危害

**粗略**

1.索引越多插入明显慢得多

2.无序插入索引影响更惊人

3.修改删除与插入略有差别

4.要增加IO和调整索引所计算的量。索引占用存储空间，随着table数据里增长，索引数据量也会增长，带来存储空间的消耗

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**分区(还没有修改)**

**-Hash分区（散列分区）**

    散列分区为通过指定分区编号来均匀分布数据的一种分区类型，因为通过在I/O设备上进行散列分区，使得这些分区大小一致。

一、好处  
  1、对于分区本生不需要定期的进行分区加入（范围分区和LIST分区需要定期的对新加入的值新建分区）  
  2、可以消除访问热点块及索引热点块，由于索引是排序后的结构，对于一列自增的列加入范围分区，可能对索引的高位块进行频繁的数据插入，导致频繁的写入和分裂  
     对于这样的索引如果加入散列分区索引即可消除。  
二、注意  
  1、分区不能太多，典型的大约1000个分区，那么在分区触发（谓词导致索引范围扫描）的并行访问操作时可能更慢，因为有非常多额外的分区维护操作（我曾经遇到过1个1844个HASH分区，导致的并行性能问题）  
  2、散列索引在维护方面可能和范围和LIST分区有一些区别。  
  详细见：

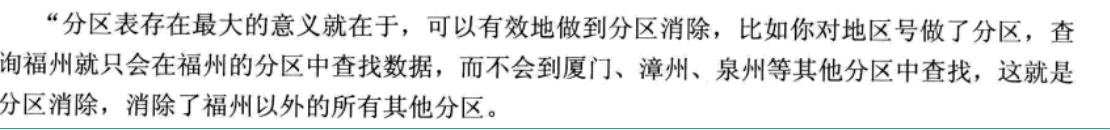
列表分区：

列表分区明确指定了根据某字段的某个具体值进行分区，而不是像范围分区那样根据字段的值范围来划分的。

范围分区：



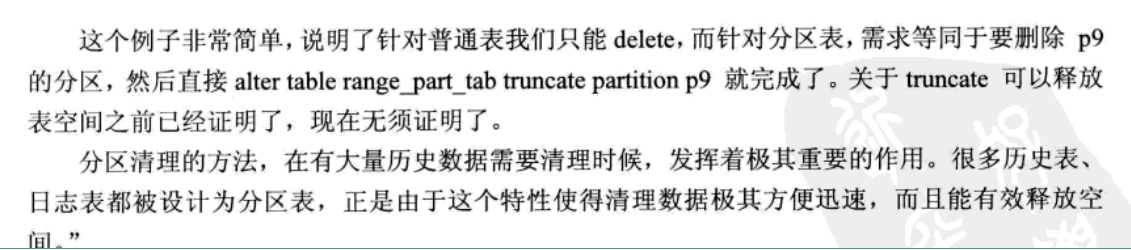
各种分区之间的优势

访问路径减少



分区truncate

delete无法释放空间，而truncate却有效释放空间，针对普通表，truncate往往不能轻易使用，delete针对某些条件的局部记录删除，而truncate显然不能戴上条件，无法做到局部删除





分区数据：

分区交换，可以实现普通表和分区表的某个分区之间数据的相互交换，他们之间的交换非常快，基本上瞬间完成，实际上在oracle内部的数据字典做的一些小动作而已，不过要两张表的字段都相同

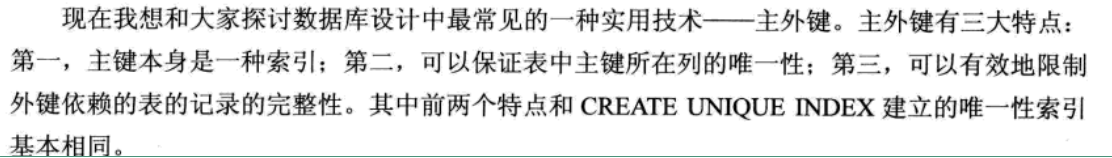


分区切割

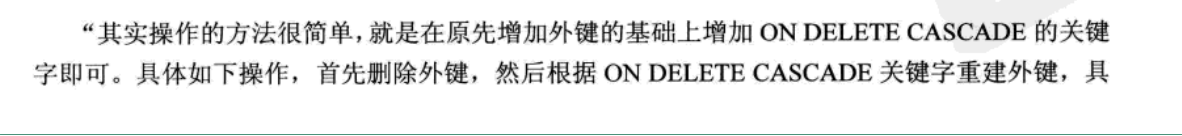


表设计：

外键 业务方面的







压缩索引

 oracle 索引压缩（key compression）是oracle 9i 中引入的一项新特性。该特性可以压缩索引或者索引组织表中的重复键值，从而节省存储空间。非分区的unique 索引和non-unique（至少两列）索引都能够被压缩。bitmap 索引不能够进行压缩。



分区索引

全局索引，局部索引



通过以上列举出的的20多个脚本案例的揉合呈现出我开篇所讲的最终的结果

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

---建索引,联合索引

---select 查询语句定死

---oracle的解析顺序，查询和where时，小表放后面。

---多个筛选where语句，id放前面。先筛选出小表。

---join小表在前

---建小表