# Oracle 的执行计划

## 优化定义

优化是选择最有效的执行计划来执行sql语句的过程,这是在处理任何数据的语句(select,insert,update Or delete)中的一个重要步骤.对Oracle来说,执行这样的语句有许多不同的方法,比如,将随以什么顺序访问那些表或索引的不同而不同.所使用的执行计划可以决定语句能执行的多块.Oracle中称之为优化器(Optimizer )的组件用来选择他认为最有效的

执行计划.

## 概念分析

### 共享SQL语句

**为了不重复解析相同的SQL语句(因为解析操作比较费资源，会导致性能下降)，在第一次解析之后，ORACLE将SQL语句及解析后得到的执行计划存放在内存中。**

**这块位于系统全局区域SGA(system global area)的共享池(shared buffer pool)中的内存可以被所有的数据库用户共享。因此，当你执行一个SQL语句(有时被称为一个游标)时，如果该语句和之前的执行过的某一语句完全相同，并且之前执行的该语句与其执行计划仍然在内存中存在，则ORACLE就不需要再进行分析，直接得到该语句的执行路径。ORACLE的这个功能大大地提高了SQL的执行性能并大大节省了内存的使用。使用这个功能的关键是将执行过的语句尽可能放到内存中，所以这要求有大的共享池(通过设置shared buffer pool参数值)和尽可能的使用绑定变量的方法执行SQL语句。**

### RowId

**rowid是一个伪列，既然是伪列，那么这个列就不是用户定义，而是系统自己给加上的。**

**对每个表都有一个rowid的伪列，但是表中并不物理存储ROWID列的值。不过你可以像使用其它列那样使用它，但是不能删除该列，也不能对该列的值进行修改、插入。一旦一行数据插入数据库，则rowid在该行的生命周期内是唯一的，即即使该行产生行迁移，行的rowid也不会改变**

### 为什么使用Rowid：

**rowid对访问一个表中的给定的行提供了最快的访问方法，通过ROWID可以直接定位到相应的数据块上，然后将其读到内存。我们创建一个索引时，该索引不但存储索引列的值，而且也存储索引值所对应的行的ROWID，这样我们通过索引快速找到相应行的ROWID后，通过该ROWID，就可以迅速将数据查询出来。这也就是我们使用索引查询时，速度比较快的原因。**

**在ORACLE8以前的版本中，ROWID由FILE 、BLOCK、ROW NUMBER构成。随着oracle8中对象概念的扩展，ROWID发生了变化，ROWID由OBJECT、FILE、BLOCK、ROW NUMBER构成。利用DBMS\_ROWID可以将rowid分解成上述的各部分，也可以将上述的各部分组成一个有效的rowid。**

### Recursive SQL概念

**为了执行用户发出的一个sql语句，Oracle必须执行一些额外的语句，我们将这些额外的语句称之为‘recursive calls’或‘recursive SQL statements’。如当一个DDL语句发出后，ORACLE总是隐含的发出一些recursive SQL语句，来修改数据字典信息，以便用户可以成功的执行该DDL语句。当需要的数据字典信息没有在共享内存中时，经常会发生Recursive calls，这些Recursive calls会将数据字典信息从硬盘读入内存中。用户不必关心这些recursive SQL语句的执行情况，在需要的时候，ORACLE会自动的在内部执行这些语句。当然DML语句与SELECT都可能引起recursive SQL。简单的说，我们可以将触发器视为recursive SQL。**

### Row Source(行源)

**用在查询中，由上一操作返回的符合条件的行的集合，即可以是表的全部行数据的集合；也可以是表的部分行数据的集合；也可以为对上2个row source进行连接操作(如join连接)后得到的行数据集合。**

### Predicate(谓词) ：

**一个查询中的WHERE限制条件**

### Driving Table(驱动表)：

**该表又称为外层表(OUTER TABLE)。这个概念用于嵌套与HASH连接中。如果该row source返回较多的行数据，则对所有的后续操作有负面影响。注意此处虽然翻译为驱动表，但实际上翻译为驱动行源(driving row source)更为确切。一般说来，是应用查询的限制条件后，返回较少行源的表作为驱动表，所以如果一个大表在WHERE条件有限制条件(如等值限制)，则该大表作为驱动表也是合适的，所以并不是只有较小的表可以作为驱动表，正确说法应该为应用查询的限制条件后，返回较少行源的表作为驱动表。**

**在执行计划中，应该为靠上的那个row source，后面会给出具体说明。在我们后面的描述中，一般将该表称为连接操作的row source 1。**

### Probed Table(被探查表)：

**该表又称为内层表(INNER TABLE)。在我们从驱动表中得到具体一行的数据后，在该表中寻找符合连接条件的行。所以该表应当为大表(实际上应该为返回较大row source的表)且相应的列上应该有索引。在我们后面的描述中，**

**一般将该表称为连接操作的row source 2。**

### 组合索引(concatenated index)：

**由多个列构成的索引，如create index idx\_emp on emp(col1, col2, col3, ……)，则我们称idx\_emp索引为组合索引。在组合索引中有一个重要的概念：引导列(leading column)，在上面的例子中，col1列为引导列。当我们进行查询时可以使用”where col1 = ? ”，也可以使用”where col1 = ? and col2 = ?”，这样的限制条件都会使用索引，但是”where col2 = ? ”查询就不会使用该索引。所以限制条件中包含先导列时，该限制条件才会使用该组合索引。**

### 可选择性(selectivity)：

**比较一下列中唯一键的数量和表中的行数，就可以判断该列的可选择性。如果该列的”唯一键的数量/表中的行数”的比值越接近1，则该列的可选择性越高，该列就越适合创建索引，同样索引的可选择性也越高。在可选择性高的列上进行查询时，返回的数据就较少，比较适合使用索引查询。**

## 执行计划分析

* **为了执行语句，Oracle可能必须实现许多步骤。这些步骤中的每一步可能是从数据库中物理检索数据行，或者用某种方法准备数据行，供发出语句的用户使用。Oracle用来执行语句的这些步骤的组合被称之为执行计划**
* **执行计划是SQL优化中最为复杂也是最为关键的部分，只有知道了ORACLE在内部到底是如何执行该SQL语句后，才能知道优化器选择的执行计划是否为最优的。执行计划对于DBA来说，就象财务报表对于财务人员一样重要。所以我们面临的问题主要是：**

1. 如何得到执行计划；

2. 如何分析执行计划;

**从而找出影响性能的主要问题。**

* **如何得到执行计划**

显示下面sql语句的执行计划：

SELECT \*

FROM PAGE\_LAYOUT p , PAGE\_LAYOUT\_CONTENT pc

WHERE

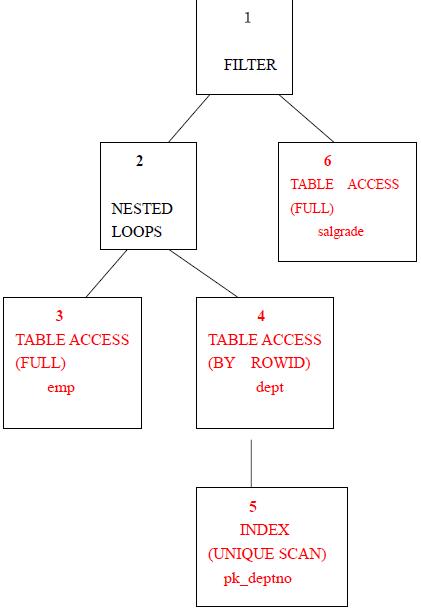
p."SID" = PC.PAGE\_LAYOUT\_SID

AND not EXISTS

(

SELECT \* FROM PAGE\_LAYOUT\_BRAND WHERE p."SID" BETWEEN 8 AND 20

)



上图是:

* + **SELECT ename, job, sal, dname**
  + **FROM emp, dept**
  + **WHERE emp.deptno = derpt.deptno**
  + **AND NOT EXISTS**
  + **( SELECT \***
  + **FROM salgrade**
  + **WHERE emp.sal BETWEEN losal AND hisal );**

的执行计划.

* **第３步和第６步分别的从EMP表和SALGRADE表读所有行。**
* **第５步在PK\_DEPTNO索引中查找由步骤３返回的每个DEPTNO值。它找出与DEPT表中相关联的那些行的ROWID。**
* **第４步从DEPT表中检索出ROWID为第５步返回的那些行。**
* **由黑色字框指出的步骤在行源上操作，如做2表之间的关联，排序，或过滤等操作。**
* **第２步实现嵌套的循环操作(相当于C语句中的嵌套循环)，接收从第３步和第４步来的行源，把来自第３步源的每一行与它第４步中相应的行连接在一起，返回结果行到第１步。**
* **第１步完成一个过滤器操作。它接收来自第２步和第６步的行源，消除掉第２步中来的，在第６步有相应行的那些行，并将来自第２步的剩下的行返回给发出语句的用户或应用**

### 实现执行计划的步骤

**执行计划中的步骤不是按照它们编号的顺序来实现的：Oracle首先实现上图树结构图形里作为叶子出现的那些步骤(例如步骤3、5、6)。由每一步返回的行称为它下一步骤的行源。然后Oracle实现父步骤**

**Oracle以下列顺序实现这些步骤**

* **首先，Oracle实现步骤３，并一行一行地将结果行返回给第２步**
* **对第３步返回的每一行，Oracle实现这些步骤**
* **Oracle实现步骤５，并将结果ROWID返回给第４步**

**Oracle实现步骤４，并将结果行返回给第２步。**

**Oracle实现步骤２，将接受来自第３步的一行和来自第４步的一行，并返回给第１步一行。**

**Oracle实现步骤６，如果有结果行的话，将它返回给第１步**

**实现步骤１，如果从步骤６返回行，Oracle将来自第２步的行返回给发出SQL语句的用户。**

## 优化器选择

* **对于以上的操作过程，可以使用first\_rows作为优化目**

**标以便于实现快速响应用户的请求。**

* **有些父步骤在它们被实现之前需要来自子步骤的所有行**

**。对这样的父步骤，直到所有行从子步骤返回之前**

**Oracle不能实现该父步骤。这样的父步骤包括排序，排**

**序一合并的连接，组功能和总计。对于这样的操作，可**

**以用all\_rows作为优化目标，使该中类型的操作耗费的**

**资源最少。**

**有时语句执行时，并不是象上面说的那样一步一步有先有后的进行，而是可能并行运行，如在实际环境中，3、5、4步可能并行运行，以便取得更好的效率。从上面的树型图上，是很难看出各个操作执行的先后顺序，而通过ORACLE生成的另一种形式的执行计划，则可以很容易的看出哪个操作先执行，哪个后执行，这样的执行计划是我们真正需要的，后面会给出详细说明。现在先来看一些预备知识**

## 访问路径—access path

**优化器在形成执行计划时需要做的一个重要选择是如何从数据库查询出需要的数据。对于SQL语句存取的任何表中的任何行，可能存在许多存取路径(存取方法)，通过它们可以定位和查询出需要的数据。优化器选择其中自认为是最优化的路径。**

**在物理层，oracle读取数据，一次读取的最小单位为数据库块(由多个连续的操作系统块组成)，一次读取的最大值由操作系统一次I/O的最大值与multiblock参数共同决定，所以即使只需要一行数据，也是将该行所在的数据库块读入内存。**

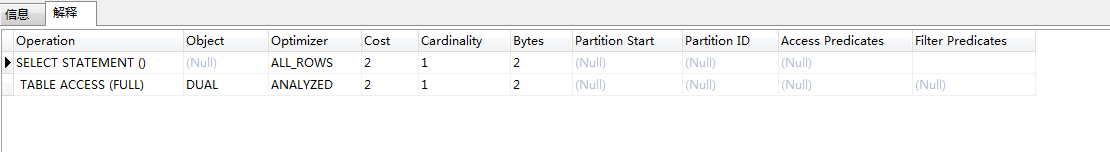
* **逻辑上，oracle用如下存取方法访问数据：**
* **1 全表扫描（Full Table Scans, FTS）**
* **2 通过ROWID的表存取（Table Access by ROWID或rowid lookup）**
* **3 索引扫描（Index Scan或index lookup）**

### 全表扫描

* **为实现全表扫描，Oracle读取表中所有的行，并检查每一行是否满足语句的WHERE限制条件。Oracle顺序读取分配给表的每个数据块，直到读到表的最高水线处(high water mark, HWM，标识表的最后一个数据块)。一个多块读操作可以使一次I/O能读取多块数据块(db\_block\_multiblock\_read\_count参数设定)，而非只读取一个数据块，这极大的减少了I/O总次数，提高了系统的吞吐量，所以利用多块读的方法可以高效实现全表扫描，而且只有在全表扫描的情况下才能使用多块读操作。在这种访问模式下，每个数据块只被读一次。由于HWM标识最后一块被读入的数据，而delete操作不影响HWM值，所以一个表的所有数据被delete后，其全表扫描的时间不会有改善，一般我们需要使用truncate命令来使HWM值归为0。幸运的是oracle 10G后，可以人工收缩HWM的值。**
* **由FTS模式读入的数据被放到高速缓存的Least Recently Used (LRU)列表的尾部，这样可以使其快速交换出内存，从而不使内存重要的数据被交换出内存。**
* **使用FTS的前提：在较大的表上不建议使用全表扫描，除非取出数据的比较多，超过总量的5% -- 10%，或你想使用并行查询功能时。**

#### 例子

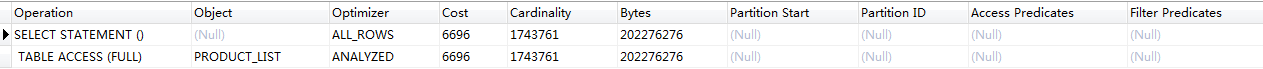
SELECT \* FROM dual;



#### 实验

全表扫描单品表(上品)—大表

SELECT \* FROM PRODUCT\_LIST p

**

耗时:

74 s

### 通过Rowid存取

* **行的ROWID指出了该行所在的数据文件、数据块以及行在该块中的位置，所以通过ROWID来存取数据可以快速定位到目标数据上，是Oracle存取单行数据的最快方法。**
* **为通过ROWID存取表，Oracle 首先要获取被选择行的ROWID，或从语句的WHERE子句中得到，或者通过表的一个或多个索引的索引扫描得到。Oracle然后以得到的ROWID为依据定位每个被选择的行。**
* **此存取方法不会用到多块读操作，一次I/O只能读取一个数据块。我们经常在执行计划中看到该存取方法，如通过索引查询数据。**

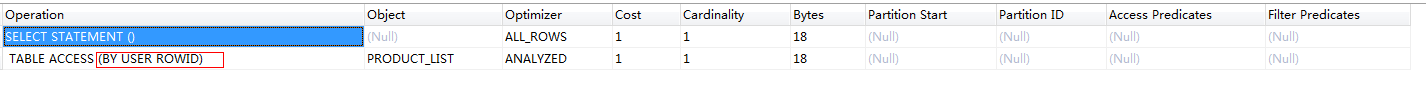
#### 例子

SELECT p."SID",ROWID as r

FROM PRODUCT\_LIST p WHERE

p.ROWID=

'AAAsydAATAALu8WAAh'



耗时:3 ms

### 索引扫描

* **通过index查找到数据对应的rowid值(对于非唯一索引可能返回多个rowid值)，然后根据rowid直接从表中得到具体的数据。**
* **一个rowid唯一的表示一行数据，该行对应的数据块是通过一次i/o得到的，该次i/o只会读取一个数据库块。**
* **在索引中，除了存储每个索引的值外，索引还存储具有此值的行对应的ROWID值。索引扫由2步组成：**

**(1) 扫描索引得到对应的rowid值。**

**(2) 通过找到的rowid从表中读出具体的数据。**

**每步都是单独的一次I/O，但对于索引，由于经常使用，绝大多数都**

**已经CACHE到内存中，所以第1步的I/O经常是逻辑I/O，即数据可以**

**从内存中得到。但是对于第2步来说，如果表比较大，则其数据不可**

**能全在内存中，则其I/O很有可能是物理I/O，这是一个机械操作，相**

**对逻辑I/O来说，极其费时间。所以如果对大表进行索引扫描，取出**

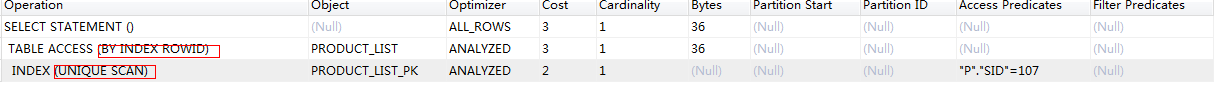
**的数据如果大于总量的5% -- 10%，使用索引扫描会效率下降很多。**

#### 例子

SELECT p."SID",p.PRODUCT\_NAME,p.PRO\_SKU

FROM PRODUCT\_LIST p WHERE

p."SID" = 107



注意:Table Access by rowid 这表面不是通过FTS存取路径访问数据,而是通过index lookup存取路径访问数据的,此例中的rowid是通过索引查找sid列的值得到的,这种方式是Index Unique Scan。

耗时: 2 ms

**根据索引的类型与where限制条件的不同，有4种类型的索引扫描**

#### 索引唯一扫描(index unique scan)

**通过唯一索引查找一个数值经常返回单个ROWID。如果该唯一索引有多个列组成(即组合索引)，则至少要有组合索引的引导列参与到该查询中，如创建一个索引：create index idx\_test on emp(ename, deptno, loc)。则select ename from emp where ename = ‘JACK’ and deptno = ‘DEV’语句可以使用该索引。如果该语句只返回一行，则存取方法称为索引唯一扫描。而select ename from emp where deptno = ‘DEV’语句则不会使用该索引，因为where子句种没有引导列。如果存在UNIQUE 或PRIMARY KEY 约束（它保证了语句只存取单行），Oracle经常实现唯一性扫描。**

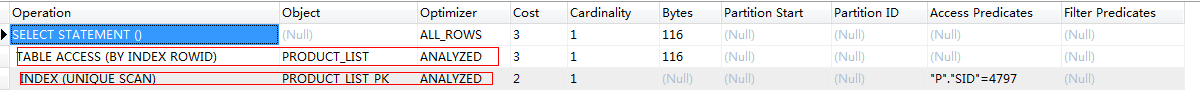
##### 例子

SELECT p.PRODUCT\_NAME

FROM PRODUCT\_LIST p

WHERE p."SID" = 4797

执行计划



耗时: 4 ms

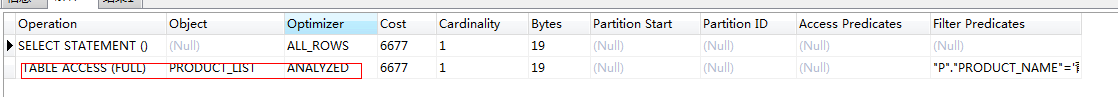
对比:

SELECT p.PRODUCT\_NAME

FROM PRODUCT\_LIST p

WHERE p.PRODUCT\_NAME = '背心084114B31609'

执行计划



耗时: 400 ms

相差100倍.

#### 索引范围扫描(index range scan)

**使用索引存取多行数据，如果索引是组合索引，如索引唯一扫描所示，且select ename from emp where ename = ‘JACK’ and deptno = ‘DEV’语句返回多行数据，虽然该语句还是使用该组合索引进行查询，可此时的存取方法称为索引范围扫描。在唯一索引上使用索引范围扫描的典型情况下是在谓词(where限制条件)中使用了范围操作符(如>、<、<>、>=、<=、between)**

##### 例子

使用索引范围扫描

SELECT p.PRODUCT\_NAME

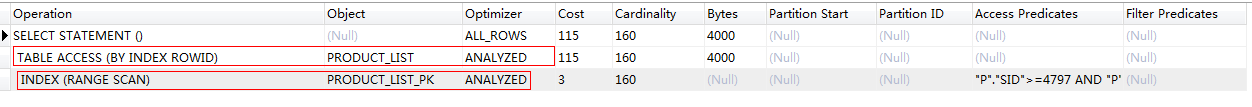
FROM PRODUCT\_LIST p

WHERE

p."SID" BETWEEN 4797 AND 5000

ORDER BY p."SID"

执行计划:



耗时: 4 ms

* **在非唯一索引上，谓词col = 5可能返回多行数据，所以在非唯一索引上都使用索引范围扫描。**
* **使用index range scan的3种情况：**
* **在唯一索引列上使用了range操作符(> < <> >= <= between)**
* **在组合索引上，只使用部分列进行查询，导致查询出多行**
* **在组合索引上，只使用部分列进行查询，导致查询出多行**

#### 索引全扫描(index full scan)

**与全表扫描对应，也有相应的全索引扫描。在某些情况下，可能进行全索引扫描而不是范围扫描，需要注意的是全索引扫描只在CBO模式下才有效。CBO根据统计数值得知进行全索引扫描比进行全表扫描更有效时，才进行全索引扫描，而且此时查询出的数据都必须从索引中可以直接得到**

##### 全索引扫描的例子：

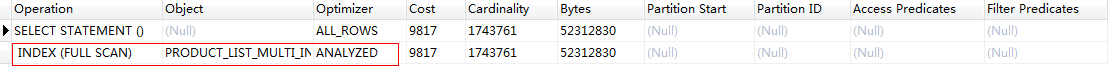
在单品表的sku name字段建立组合索引

SELECT p.PRO\_SKU,p.PRODUCT\_NAME

FROM PRODUCT\_LIST p

ORDER BY p.PRO\_SKU,p.PRODUCT\_NAME

执行计划:



#### 索引快速扫描(index fast full scan)

**扫描索引中的所有的数据块，与 index full scan很类似，但是一个显著的区别就是它不对查询出的数据进行排序，即数据不是以排序顺序被返回。在这种存取方法中，可以使用多块读功能，也可以使用并行读入，以便获得最大吞吐量与缩短执行时间。**

##### 索引快速扫描的例子

Pro\_sku product\_name是组合索引

SELECT p.PRODUCT\_NAME

FROM PRODUCT\_LIST p

执行计划:



## 表间连接

* **Join是一种试图将两个表结合在一起的谓词，一次只能连接2个表，表连接也可以被称为表关联。Join过程的各个步骤经常是串行操作，即使相关的row source可以被并行访问，即可以并行的读取做join连接的两个row source的数据，但是在将表中符合限制条件的数据读入到内存形成row source后，join的其它步骤一般是串行的。有多种方法可以将2个表连接起来，当然每种方法都有自己的优缺点，每种连接类型只有在特定的条件下才会发挥出其最大优势。**
* **row source(表)之间的连接顺序对于查询的效率有非常大的影响。通过首先存取特定的表，即将该表作为驱动表，这样可以先应用某些限制条件，从而得到一个较小的row source，使连接的效率较高，这也就是我们常说的要先执行限制条件的原因。一般是在将表读入内存时，应用where子句中对该表的限制条件。**
* **根据2个row source的连接条件的中操作符的不同，可以将连接分为等值连接(如WHERE A.COL3 = B.COL4)、非等值连接(WHERE A.COL3 > B.COL4)、外连接(WHERE A.COL3 = B.COL4(+))。**
* **下面以等值连接为例进行介绍。在后面的介绍中，都已：**

**SELECT A.COL1, B.COL2**

**FROM A, B**

**WHERE A.COL3 = B.COL4;**

**为例进行说明，假设A表为Row Soruce1，则其对应的连接操作关联列为COL 3；B表为Row Soruce2，则其对应的连接操作关联列为COL 4；**

### 排序 - - 合并连接

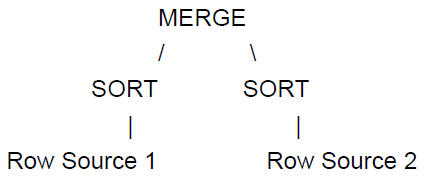
**(Sort Merge Join (SMJ) )**

#### 内部连接过程：

* **首先生成row source1需要的数据，然后对这些数据按照连接操**

**作关联列(如A.col3)进行排序。**

* **随后生成row source2需要的数据，然后对这些数据按照与sort source1对应的连接操作关联列(如B.col4)进行排序。**
* **最后两边已排序的行被放在一起执行合并操作，即将2个row source按照连接条件连接起来。**

****

**如果row source已经在连接关联列上被排序，则该连接操作就不需要再进行sort操作，这样可以大大提高这种连接操作的连接速度，因为排序是个极其费资源的操作， 特别是对于较大的表。 预先排序的row source包括已经被索引的列(如a.col3或b.col4上有索引)或row source已经在前面的步骤中被排序了。尽管合并两个row source的过程是串行的，但是可以并行访问这两个row source(如并行读入数据，并行排序).**

**排序是一个费时、费资源的操作，特别对于大表。基于这个原因，SMJ经常不是一个特别有效的连接方法，但是如果2个row source都已经预先排序，则这种连接方法的效率较高。**

#### SMJ连接的例子

**SQL> explain plan for**

**select /\*+ ordered \*/ e.deptno, d.deptno**

**from emp e, dept d**

**where e.deptno = d.deptno**

**order by e.deptno, d.deptno;**

**Query Plan**

执行计划

**SELECT STATEMENT [CHOOSE] Cost=17**

**MERGE JOIN**

**SORT JOIN**

**TABLE ACCESS FULL EMP [ANALYZED]**

**SORT JOIN**

**TABLE ACCESS FULL DEPT [ANALYZED]**

### 嵌套循环

Nested Loops (NL)

**该连接过程是一个2层嵌套循环，则外层循环的次数越少越好，这也就是我们为什么将小表或返回较小row source的表作为驱动表(用于外层循环)的理论依据。但是这个理论只是一般指导原则，因为遵循这个理论并不能总保证使语句产生的I/O次数最少。有时不遵守这个理论依据，反而会获得更好的效率。如果使用这种方法，决定使用哪个表作为驱动表很重要。有时如果驱动表选择不正确，将会导致语句的性能很差、很差。**

#### 内部连接过程：

Row source1的Row 1 -------------- -- Probe -> Row source 2

Row source1的Row 2 -------------- -- Probe -> Row source 2

Row source1的Row 3 -------------- -- Probe -> Row source 2

…….

Row source1的Row n -------------- -- Probe -> Row source 2

**从内部连接过程来看，需要用row source1中的每一行，去匹配row source2中的所有行，所以此时保持row source1尽可能的小与高效的访问row source2(一般通过索引实现)是影响这个连接效率的关键问题。这只是理论指导原则，目的是使整个连接操作产生最少的物理I/O次数，而且如果遵守这个原则，一般也会使总的物理I/O数最少。但是如果不遵从这个指导原则，反而能用更少的物理I/O实现连接操作，那尽管违反指导原则吧！因为最少的物理I/O次数才是我们应该遵从的真正的指导原则，在后面的具体案例分析中就给出这样的例子。**

* **在上面的连接过程中，我们称Row Source1为驱动表或外部表。Row Source2被称为被探查表或内部表。**
* **在NESTED LOOPS连接中，Oracle读取row source1中的每一行，然后在row sourc2中检查是否有匹配的行，所有被匹配的行都被放到结果集中，然后处理row source1中的下一行。这个过程一直继续，直到row source1中的所有行都被处理。这是从连接操作中可以得到第一个匹配行的最快的方法之一，这种类型的连接可以用在需要快速响应的语句中，以响应速度为主要目标。**
* **如果driving row source(外部表)比较小，并且在inner row source(内部表)上有唯一索引，或有高选择性非唯一索引时，使用这种方法可以得到较好的效率。NESTED LOOPS有其它连接方法没有的的一个优点是：可以先返回已经连接的行，而不必等待所有的连接操作处理完才返回数据，这可以实现快速的响应时间。**
* **如果不使用并行操作，最好的驱动表是那些应用了where 限制条件后，可以返回较少行数据的的表，所以大表也可能称为驱动表，关键看限制条件。对于并行查询，我们经常选择大表作为驱动表，因为大表可以充分利用并行功能。当然，有时对查询使用并行操作并不一定会比查询不使用并行操作效率高，因为最后可能每个表只有很少的行符合限制条件，而且还要看你的硬件配置是否可以支持并行(如是否有多个CPU，多个硬盘控制器)，所以要具体问题具体对待。**

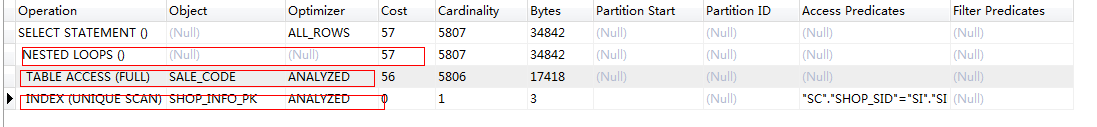
#### NL连接的例子

SELECT /\*+ ordered \*/ SC.SHOP\_SID,SI."SID"

FROM SALE\_CODE sc, SHOP\_INFO si

WHERE SC.SHOP\_SID = SI."SID"

**执行计划**

****

### 哈希连接

Hash Join , HJ

* **理论上来说比NL与SMJ更高效，而且只用在CBO优化器中。**
* **较小的row source被用来构建hash table与bitmap，第2个row source被用来被hansed，并与第一个row source生成的hash table进行匹配，以便进行进一步的连接。Bitmap被用来作为一种比较快的查找方法，来检查在hash table中是否有匹配的行。特别的，当hash table比较大而不能全部容纳在内存中时，这种查找方法更为有用。这种连接方法也有NL连接中所谓的驱动表的概念，被构建为hash table与bitmap的表为驱动表，当被构建的hash table与bitmap能被容纳在内存中时，这种连接方式的效率极高。**
* **要使哈希连接有效，需要设置HASH\_JOIN\_ENABLED=TRUE，缺省情况下该参数为TRUE，另外，不要忘了还要设置hash\_area\_size参数，以使哈希连接高效运行，因为哈希连接会在该参数指定大小的内存中运行，过小的参数会使哈希连接的性能比其他连接方式还要低。**

#### Hash Join的例子

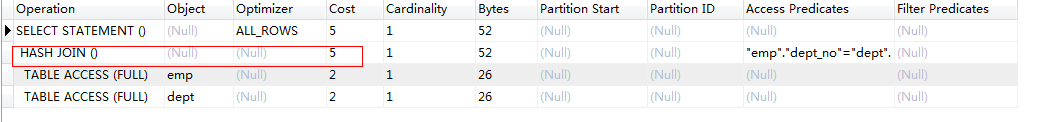
SELECT \*

FROM "emp","dept"

WHERE

"emp"."dept\_no" = "dept"."dept\_no"

执行计划:



### 笛卡儿乘积

Cartesian product

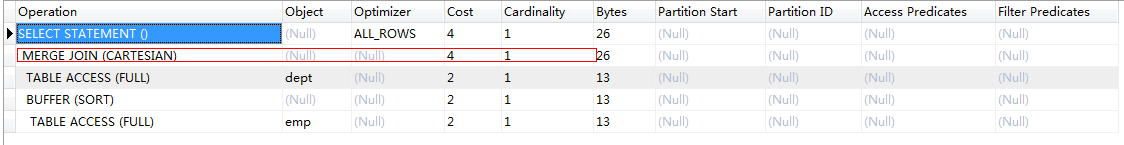
* **当两个row source做连接，但是它们之间没有关联条件时，就会在两个row source中做笛卡儿乘积，这通常由编写代码疏漏造成(即程序员忘了写关联条件)。笛卡尔乘积是一个表的每一行依次与另一个表中的所有行匹配。在特殊情况下我们可以使用笛卡儿乘积，如在星形连接中，除此之外，我们要尽量避免使用笛卡儿乘积。**

#### 笛卡尔乘积例子:

SELECT "emp"."dept\_no","dept"."dept\_no"

FROM "dept","emp"

执行计划：



**CARTESIAN关键字指出了在2个表之间做笛卡尔乘积。假如表emp有n行，dept表有m行，笛卡尔乘积的结果就是得到n \* m行结果。**

### 总结

**在哪种情况下用哪种连接方法比较好**

* **排序 - - 合并连接(Sort Merge Join, SMJ)：** 
  + **对于非等值连接，这种连接方式的效率是比较高的。**
  + **如果在关联的列上都有索引，效果更好。**
  + **对将2个较大的row source做连接，该连接方法比NL连接要好些。**
  + **但是如果sort merge返回的row source过大，则又会导致使用过**

**多的rowid在表中查询数据时，数据库性能下降，因为过多的I/O。**

* **嵌套循环(Nested Loops, NL)：** 
  + **如果driving row source(外部表)比较小，并且在inner row source(内部表)上有唯一索引，或有高选择性非唯一索引时，使用这种方法可以得到较好的效率。**
  + **NESTED LOOPS有其它连接方法没有的的一个优点是：可以先返回已经 连接的行，而不必等待所有的连接操作处理完才返回数据，这可以实现快速的响应时间。**
* **哈希连接(Hash Join, HJ)：** 
  + **此方法是在oracle7后来引入的，使用了比较先进的连接理论，**

**其效率应该好于其它2种连接，但是这种连接只能用在**

**CBO优化器中，而且需要设置合适的hash\_area\_size参数，才能**

**取得较好的性能。**

* + **在2个较大的row source之间连接时会取得相对较好的效率，在一**

**个 row source较小时则能取得更好的效率。**

* + **只能用于等值连接中**

## 如何产生执行计划

**要为一个语句生成执行计划，可以有3种方法：**

* **最简单的办法**

**Sql> set autotrace on**

**Sql> select \* from dual;**

**执行完语句后，会显示explain plan 与 统计信息。**

**这个语句的优点就是它的缺点，这样在用该方法查看执行时间较长的sql语句时，需要等待该语句执行成功后，才返回执行计划，使优化的周期大大增长。**

**如果不想执行语句而只是想得到执行计划可以采用：**

**Sql> set autotrace traceonly**

**这样，就只会列出执行计划，而不会真正的执行语句，大大减少了优化时间。虽然也列出了统计信息，但是因为没有执行语句，所以该统计信息没有用处，**

* **用explain plan命令**

**sqlplus > explain plan set statement\_id =’???’ for select**

**………………**

**此方法时，并不执行sql语句，所以只会列出执行计划，不会列出统计信息，并且执行计划只存在plan\_table中。所以该语句比起set autotrace traceonly可用性要差。需要用下面的命令格式化输出，所以这种方式用的不多：**

**以上2种方法只能为在本会话中正在运行的语句产生执行计划，即我们需要已经知道了哪条语句运行的效率很差，我们是有目的只对这条SQL语句去优化。在很多情况下，我们只会听一个客户抱怨说现在系统运行很慢，而我们不知道是哪个SQL引起的。此时有许多现成的语句可以找出耗费资源比较多的语句，如：**

SELECT ADDRESS,

substr(SQL\_TEXT,1,40) Text,

buffer\_gets,

executions,

buffer\_gets/executions AVG

FROM v$sqlarea

WHERE executions>0

AND buffer\_gets > 100000

ORDER BY 5;

* **用dbms\_system存储过程生成执行计划**

**使用dbms\_system存储过程可以跟踪另一个会话发出的sql语句，并记录所使用的执行计划，而且还提供其它对性能调整有用的信息。这种方法是对SQL进行调整比较有用的方式之一**

## 如何分析执行计划

**通过如下示例进行分析演示：**

### 例1

**假设LARGE\_TABLE是一个较大的表，且username列上没有索引，则**

**运行下面的语句**

**SQL> SELECT \* FROM LARGE\_TABLE where USERNAME = ‘TEST’;**

**Query Plan -----------------------------------------**

**SELECT STATEMENT Optimizer=CHOOSE (Cost=1234 Card=1 Bytes=14)**

**TABLE ACCESS FULL LARGE\_TABLE [:Q65001] [ANALYZED]**

**在这个例子中，TABLE ACCESS FULL LARGE\_TABLE是第一个操作**

**，意思是在LARGE\_TABLE表上做全表扫描。当这个操作完成之后，产**

**生的row source中的数据被送往下一步骤进行处理，在此例中，**

**SELECT STATEMENT 操作是这个查询语句的最后一步。**

**Optimizer=CHOOSE 指明这个查询的optimizer\_mode，即**

**optimizer\_mode 初始化参数指定的值，它并不是指语句执行时真的使用**

**了该优化器。决定该语句使用何种优化器的唯一方法是看后面的cost部**

**分。如果给出的是下面的形式，则表明使用的是CBO优化器，此**

**处的cost表示优化器认为该执行计划的代价:**

**SELECT STATEMENT Optimizer=CHOOSE (Cost=1234 Card=1 Bytes=14)**

**假如执行计划中给出的是类似下面的信息，则表明是使用RBO优化器，因为cost部分的值为空，或者压根就没有cost部分**

**SELECT STATEMENT Optimizer=CHOOSE Cost=**

**SELECT STATEMENT Optimizer=CHOOSE**

**这样我们从Optimizer后面的信息中可以得出执行该语句时到底用了**

**什么样的优化器。特别的，如果Optimizer=ALL\_ROWS|**

**FIRST\_ROWS| FIRST\_ROWS\_n，则使用的是CBO优化器；**

**如果Optimizer=RULE，则使用的是RBO优化器。**

* **cost属性的值是一个在oracle内部用来比较各个执行计划所耗费代价的值，从而使优化器可以选择最好的执行计划。不同语句的cost值不具有可比性，只能对同一个语句的不同执行计划的cost值进行比较。**
* **[:Q65001] 表明该部分查询是以并行方式运行的。里面的数据表示这个操作是由并行查询的一个slave进程处理的，以便该操作可以区别于串行执行的操作。**
* **[ANALYZED] 表明操作中引用的对象被分析过了，在数据字典中有该对象的统计信息可以供CBO使用**

### 例2

**假定A、B、C都是不是小表，且在A表上一个组合索引：A(a.col1,a.col2) ，注意a.col1列为索引的引导列。**

**在表做连接时，只能2个表先做连接，然后将连接后的结果作为一个row source，与剩下的表做连接，在上面的例子中，连接顺序为B与A先连接，然后再与C连接：**

**B <---> A <---> C**

**col3=10 col3=5**

* **如果没有执行计划，分析一下，上面的3个表应该拿哪一个作为第一个驱动表？从SQL语句看来，只有B表与C表上有限制条件，所以第一个驱动表应该为这2个表中的一个，到底是哪一个呢？**
* **B表有谓词B.col3 = 10，这样在对B表做全表扫描的时候就将where子句中的限制条件(B.col3 = 10)用上，从而得到一个较小的row source, 所以B表应该作为第一个驱动表。而且这样的话，如果再与A表做关联，可以有效利用A表的索引(因为A表的col1列为leading column)。**
* **当然上面的查询中C表上也有谓词(C.col3 = 5)，有人可能认为C表作为第一个驱动表也能获得较好的性能。让我们再来分析一下：如果C表作为第一个驱动表，则能保证驱动表生成很小的row source，但是看看连接条件A.col2 = C.col2，此时就没有机会利用A表的索引，因为A表的col2列不为leading column，这样nested loop的效率很差，从而导致查询的效率很差。所以对于NL连接选择正确的驱动表很重要。**
* **基于以上原则：上面查询比较好的连接顺序为(B - - > A) - - > C。如果数据库是基于代价的优化器，它会利用计算出的代价来决定合适的驱动表与合适的连接顺序。一般来说，CBO都会选择正确的连接顺序，如果CBO选择了比较差的连接顺序，我们还可以使用ORACLE提供的hints来让CBO采用正确的连接顺序。**

**select /\*+ ordered \*/ A.col4**

**from B,A,C**

**where B.col3 = 10**

**and A.col1 = B.col1**

**and A.col2 = C.col2**

**and C.col3 = 5**

#### 通过执行计划如何判断驱动表：

**在执行计划中，需要知道哪个操作是先执行的，哪个操作是后执行的，这对于判断哪个表为驱动表有用处。判断之前，如果对表的访问是通过rowid，且该rowid的值是从索引扫描中来得，则将该索引扫描先从执行计划中暂时去掉。然后在执行计划剩下的部分中，判断执行顺序的指导原则就是：最右、最上的操作先执行。具体解释如下**

**得到去除妨碍判断的索引扫描后的执行计划：**

**Execution Plan**

**-----------------------------------------------------------------------------------------------**

**0 SELECT STATEMENT Optimizer=CHOOSE**

**1 0 MERGE JOIN**

**2 1 SORT (JOIN)**

**3 2 NESTED LOOPS**

**4 3 TABLE ACCESS (FULL) OF 'B'**

**5 3 TABLE ACCESS (BY INDEX ROWID) OF 'A'**

**7 1 SORT (JOIN)**

**8 7 TABLE ACCESS (FULL) OF 'C'**

**谈论上下关系时，只对连续的、缩进一致的行有效。**

**从这个图中我们可以看到，对于NESTED LOOPS部分，最右、最上的操作是TABLE ACCESS (FULL) OF 'B'，所以这一操作先执行，所以该操作对应的B表为第一个驱动表(外部表)，自然，A表就为内部表了。从图中还可以看出，B与A表做嵌套循环后生成了新的row source ，对该row source进行来排序后，与C表对应的排序了的row source(应用了C.col3 = 5限制条件)进行MSJ连接操作。所以从上面可以得出如下事实：B表先与A表做嵌套循环，然后将生成的row source与C表做排序—合并连接。**

**通过分析上面的执行计划，我们不能说C表一定在B、A表之后才被读取，事实上，B表有可能与C表同时被读入内存，因为将表中的数据读入内存的操作可能为并行的。事实上许多操作可能为交叉进行的，因为ORACLE读取数据时，如果就是需要一行数据也是将该行所在的整个数据块读入内存，而且还有可能为多块读。**

**看执行计划时，我们的关键不是看哪个操作先执行，哪个操作后执行，而是关键看表之间连接的顺序(如得知哪个为驱动表，这需要从操作的顺序进行判断)、使用了何种类型的关联及具体的存取路径(如判断是否利用了索引) 在从执行计划中判断出哪个表为驱动表后，根据我们的知识判断该表作为驱动表(就像上面判断ABC表那样)是否合适，如果不合适，对SQL语句进行更改，使优化器可以选择正确的驱动表。**

## 如何干预执行计划

### 使用hints提示：

**基于代价的优化器在绝大多数情况下它会选择正确的优化器，减轻了DBA的负担。但有时它也聪明反被聪明误，选择了很差的执行计划，使某个语句的执行变得奇慢无比。此时就需要DBA进行人为的干预，告诉优化器使用我们指定的存取路径或连接类型生成执行计划，从而使语句高效的运行。例如: 对于一个特定的语句，执行全表扫描要比执行索引扫描更有效，则我们可以指示优化器使用全表扫描。在ORACLE中，是通过为语句添加hints(提示)来实现干预优化器优化的目的。**

**hints是oracle提供的一种机制，用来告诉优化器按照我们的告诉它的方式生成执行计划。可以用hints来实现：**

* **使用的优化器的类型**
* **基于代价优化器的优化目标，是all\_rows还是first\_rows。**
* **表的访问路径，是全表扫描、索引扫描，还是直接利用rowid**
* **表之间的连接类型**
* **表之间的连接顺序**
* **语句的并行程度**

### 如何使用hints：

**Hints只应用在它们所在sql语句块(statement block，由select、update、delete关键字标识)上，对其它SQL语句或语句的其它部分没有影响。如：对于使用union操作的2个sql语句，如果只在一个sql语句上有hints，则该hints不会影响另一个sql语句。**

**可以使用注释(comment)来为一个语句添加hints，一个语句块只能有一个注释，而且注释只能放在SELECT, UPDATE, or DELETE关键字的后面。**

**使用hints的语法：**

**{DELETE|INSERT|SELECT|UPDATE} /\*+ hint [text] [hint[text]]... \*/**

**or**

**{DELETE|INSERT|SELECT|UPDATE} --+ hint [text] [hint[text]]...**

* **DELETE、INSERT、SELECT和UPDATE是标识一个语句块开始的关键字，包含提示的注释只能出现在这些关键字的后面，否则提示无效。**
* **“+”号表示注释是一个hints，加号必须立即跟在”/\*”后面，中间不能有空格。**
* **hint是下面介绍的具体提示之一，如果包含多个提示，则每个提示之间需要用一个或多个空格隔开。**
* **text 是其它说明hint的注释性文本**
* **如果你没有正确的指定hints，Oracle将忽略该hints，并且不会给出任何错误。**

### 指示优化器的方法与目标的hints：

* **ALL\_ROWS -- 基于代价的优化器，以吞吐量为目标**
* **FIRST\_ROWS(n) -- 基于代价的优化器，以响应时间为目标**
* **CHOOSE -- 根据是否有统计信息，选择不同的优化器**
* **RULE -- 使用基于规则的优化器**

## 总结

# MySQL优化

## 为查询缓存优化你的查询

大多数的MySQL服务器都开启了查询缓存。这是提高性最有效的方法之一，而且这是被MySQL的数据库引擎处理的。当有很多相同的查询被执行了多次的时候，这些查询结果会被放到一个缓存中，这样，后续的相同的查询就不用操作表而直接访问缓存结果了。

这里最主要的问题是，对于程序员来说，这个事情是很容易被忽略的。因为，我们某些查询语句会让MySQL不使用缓存。请看下面的示例：

* 查询缓存不开启的情况

SELECT \*

FROM leave\_message l

WHERE l.leavemsgtime >= CURDATE()

耗时: 75 ms

* 查询缓存开启

SELECT \*

FROM leave\_message l

WHERE l.leavemsgtime >= '2013-12-23 00:00:00'

耗时: 3 ms

上面两条SQL语句的差别就是 CURDATE() ，MySQL的查询缓存对这个函数不起作用。所以，像 NOW() 和 RAND() 或是其它的诸如此类的SQL函数都不会开启查询缓存，因为这些函数的返回是会不定的易变的。所以，你所需要的就是用一个变量来代替MySQL的函数，从而 开启缓存。

## EXPLAIN 你的 SELECT 查询

使用 [EXPLAIN](http://dev.mysql.com/doc/refman/5.0/en/explain.html) 关键字可以让你知道MySQL是如何处理你的SQL语句的。这可以帮你分析你的查询语句或是表结构的性能瓶颈

EXPLAIN 的查询结果还会告诉你你的索引主键被如何利用的，你的数据表是如何被搜索和排序的……等等，等等

* Explain 的使用 tid未加索引的情况下:

EXPLAIN SELECT \*

FROM leave\_message l

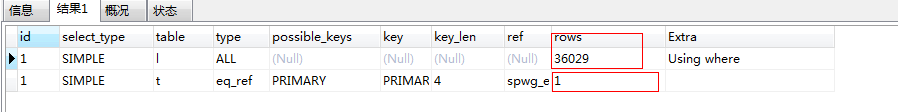
JOIN leavemsgtype t ON (l.tid = t.tid)

WHERE t.tid

BETWEEN 1

AND 10

执行结果:



* **添加索引后:**

**EXPLAIN SELECT \***

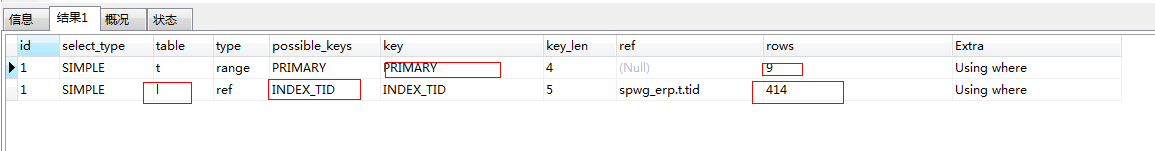
**FROM leave\_message l**

**JOIN leavemsgtype t ON (l.tid = t.tid)**

**WHERE t.tid**

**BETWEEN 1**

**AND 10**



耗时：225 ms

查看rows列可以让我们看到潜在的性能问题.

## 当只要一行数据时使用 LIMIT 1

当你查询表的有些时候，你已经知道结果只会有一条结果，但因为你可能需要去fetch游标，或是你也许会去检查返回的记录数。

在这种情况下，加上 LIMIT 1 可以增加性能。这样一样，MySQL数据库引擎会在找到一条数据后停止搜索，而不是继续往后查少下一条符合记录的数据。

例如:寻找page\_layout表中是否存在title为闪购栏目的数据.

* 没有效率的

SELECT \* FROM page\_layout p

WHERE p.title='闪购栏目'

耗时： 4 ms

* 有效率的

SELECT 1 FROM page\_layout p

WHERE p.title = '闪购栏目' LIMIT 1

耗时: 1 ms

## 为搜索字段建索引

索引并不一定就是给主键或是唯一的字段。如果在你的表中，有某个字段你总要会经常用来做搜索，那么，请为其建立索引吧。

* Name建立索引前

SELECT COUNT(\*)

FROM member m

WHERE m.username LIKE 'y%' /\*SQL\_NO\_CACHE\*/

耗时: 1.3 s

* 建立索引后

SELECT COUNT(\*)

FROM member m

WHERE m.username LIKE 'b%' /\*SQL\_NO\_CACHE\*/;

耗时: 26ms

建立索引前后性能相差: 50倍

另外，你应该也需要知道什么样的搜索是不能使用正常的索引的。例如，当你需要在一篇大的文章中搜索一个词时，如： “WHERE post\_content LIKE ‘%apple%’”，索引可能是没有意义的。你可能需要使用[MySQL全文索引](http://dev.mysql.com/doc/refman/5.1/en/fulltext-search.html) 或是自己做一个索引（比如说：搜索关键词或是Tag什么的）

## 在Join表的时候使用相当类型的例，并将其索引

如果你的应用程序有很多 JOIN 查询，你应该确认两个表中Join的字段是被建过索引的。这样，MySQL内部会启动为你优化Join的SQL语句的机制。

而且，这些被用来Join的字段，应该是相同的类型的。例如：如果你要把 DECIMAL 字段和一个 INT 字段Join在一起，MySQL就无法使用它们的索引。对于那些STRING类型，还需要有相同的字符集才行。（两个表的字符集有可能不一样）

// 在state中查找company

$r = mysql\_query("SELECT company\_name FROM users

    LEFT JOIN companies ON (users.state = companies.state)

    WHERE users.id = $user\_id");

// 两个 state 字段应该是被建过索引的，而且应该是相当的类型，相同的字符集。

## 千万不要 ORDER BY RAND()

想打乱返回的数据行？随机挑一个数据？真不知道谁发明了这种用法，但很多新手很喜欢这样用。但你确不了解这样做有多么可怕的性能问题。

如果你真的想把返回的数据行打乱了，你有N种方法可以达到这个目的。这样使用只让你的数据库的性能呈指数级的下降。这里的问题是：MySQL会不得不去执 行RAND()函数（很耗CPU时间），而且这是为了每一行记录去记行，然后再对其排序。就算是你用了Limit 1也无济于事（因为要排序）

下面的示例是随机挑一条记录：

SELECT \*

FROM member m

ORDER BY RAND() LIMIT 1

千万不要这么做

耗时: 22 s

SELECT \*

FROM member m

LIMIT 10000 , 1

这样会更好:

耗时: 27ms

## 避免 SELECT \*

从数据库里读出越多的数据，那么查询就会变得越慢。并且，如果你的数据库服务器和WEB服务器是两台独立的服务器的话，这还会增加网络传输的负载。

所以，你应该养成一个需要什么就取什么的好的习惯。

* 不推荐

SELECT \* FROM user WHERE user\_id = 1

* 推荐

SELECT username FROM user WHERE user\_id = 1

## 永远为每张表设置一个ID

我们应该为数据库里的每张表都设置一个ID做为其主键，而且最好的是一个INT型的（推荐使用UNSIGNED），并设置上自动增加的AUTO\_INCREMENT标志。

就算是你 users 表有一个主键叫 “email”的字段，你也别让它成为主键。使用 VARCHAR 类型来当主键会使用得性能下降。另外，在你的程序中，你应该使用表的ID来构造你的数据结构。

而且，在MySQL数据引擎下，还有一些操作需要使用主键，在这些情况下，主键的性能和设置变得非常重要，比如，集群，分区……

在这里，只有一个情况是例外，那就是“关联表”的“外键”，也就是说，这个表的主键，通过若干个别的表的主键构成。我们把这个情况叫做“外键”。比如：有 一个“学生表”有学生的ID，有一个“课程表”有课程ID，那么，“成绩表”就是“关联表”了，其关联了学生表和课程表，在成绩表中，学生ID和课程ID 叫“外键”其共同组成主键。

## 使用 ENUM 而不是 VARCHAR

[ENUM](http://dev.mysql.com/doc/refman/5.0/en/enum.html) 类型是非常快和紧凑的。在实际上，其保存的是 TINYINT，但其外表上显示为字符串。这样一来，用这个字段来做一些选项列表变得相当的完美。

如果你有一个字段，比如“性别”，“国家”，“民族”，“状态”或“部门”，你知道这些字段的取值是有限而且固定的，那么，你应该使用 ENUM 而不是 VARCHAR。

MySQL也有一个“建议”（见第十条）告诉你怎么去重新组织你的表结构。当你有一个 VARCHAR 字段时，这个建议会告诉你把其改成 ENUM 类型。使用 PROCEDURE ANALYSE() 你可以得到相关的建议。

## 从 PROCEDURE ANALYSE() 取得建议

[PROCEDURE ANALYSE()](http://dev.mysql.com/doc/refman/5.0/en/procedure-analyse.html) 会让 MySQL 帮你去分析你的字段和其实际的数据，并会给你一些有用的建议。只有表中有实际的数据，这些建议才会变得有用，因为要做一些大的决定是需要有数据作为基础的。

例如，如果你创建了一个 INT 字段作为你的主键，然而并没有太多的数据，那么，PROCEDURE ANALYSE()会建议你把这个字段的类型改成 MEDIUMINT 。或是你使用了一个 VARCHAR 字段，因为数据不多，你可能会得到一个让你把它改成 ENUM 的建议。这些建议，都是可能因为数据不够多，所以决策做得就不够准。

一定要注意，这些只是建议，只有当你的表里的数据越来越多时，这些建议才会变得准确。一定要记住，你才是最终做决定的人。

## 尽可能的使用 NOT NULL

除非你有一个很特别的原因去使用 NULL 值，你应该总是让你的字段保持 NOT NULL。这看起来好像有点争议，请往下看。

首先，问问你自己“Empty”和“NULL”有多大的区别（如果是INT，那就是0和NULL）？如果你觉得它们之间没有什么区别，那么你就不要使用NULL。（你知道吗？在 Oracle 里，NULL 和 Empty 的字符串是一样的！)

不要以为 NULL 不需要空间，其需要额外的空间，并且，在你进行比较的时候，你的程序会更复杂。 当然，这里并不是说你就不能使用NULL了，现实情况是很复杂的，依然会有些情况下，你需要使用NULL值。

## Prepared Statements

Prepared Statements很像存储过程，是一种运行在后台的SQL语句集合，我们可以从使用 prepared statements 获得很多好处，无论是性能问题还是安全问题。

Prepared Statements 可以检查一些你绑定好的变量，这样可以保护你的程序不会受到“SQL注入式”攻击。当然，你也可以手动地检查你的这些变量，然而，手动的检查容易出问题， 而且很经常会被程序员忘了。当我们使用一些framework或是ORM的时候，这样的问题会好一些。

在性能方面，当一个相同的查询被使用多次的时候，这会为你带来可观的性能优势。你可以给这些Prepared Statements定义一些参数，而MySQL只会解析一次。

虽然最新版本的MySQL在传输Prepared Statements是使用二进制形势，所以这会使得网络传输非常有效率。

当然，也有一些情况下，我们需要避免使用Prepared Statements，因为其不支持查询缓存。但据说版本5.1后支持了。

## 无缓冲的查询

正常的情况下，当你在当你在你的脚本中执行一个SQL语句的时候，你的程序会停在那里直到没这个SQL语句返回，然后你的程序再往下继续执行。你可以使用无缓冲查询来改变这个行为。

## 把IP地址存成 UNSIGNED INT

很多程序员都会创建一个 VARCHAR(15) 字段来存放字符串形式的IP而不是整形的IP。如果你用整形来存放，只需要4个字节，并且你可以有定长的字段。而且，这会为你带来查询上的优势，尤其是当 你需要使用这样的WHERE条件：IP between ip1 and ip2。

我们必需要使用UNSIGNED INT，因为 IP地址会使用整个32位的无符号整形。

## 固定长度的表会更快

如果表中的所有字段都是“固定长度”的，整个表会被认为是 [“static” 或 “fixed-length”](http://dev.mysql.com/doc/refman/5.1/en/static-format.html)。 例如，表中没有如下类型的字段： VARCHAR，TEXT，BLOB。只要你包括了其中一个这些字段，那么这个表就不是“固定长度静态表”了，这样，MySQL 引擎会用另一种方法来处理。

固定长度的表会提高性能，因为MySQL搜寻得会更快一些，因为这些固定的长度是很容易计算下一个数据的偏移量的，所以读取的自然也会很快。而如果字段不是定长的，那么，每一次要找下一条的话，需要程序找到主键。

并且，固定长度的表也更容易被缓存和重建。不过，唯一的副作用是，固定长度的字段会浪费一些空间，因为定长的字段无论你用不用，他都是要分配那么多的空间。

使用“垂直分割”技术（见下一条），你可以分割你的表成为两个一个是定长的，一个则是不定长的。

## 垂直分割

“垂直分割”是一种把数据库中的表按列变成几张表的方法，这样可以降低表的复杂度和字段的数目，从而达到优化的目的。（以前，在银行做过项目，见过一张表有100多个字段，很恐怖）

* **示例一**在Users表中有一个字段是家庭地址，这个字段是可选字段，相比起，而且你在数据库操作的时候除了个人信息外，你并不需要经常读取或是改写这个字段。那 么，为什么不把他放到另外一张表中呢？ 这样会让你的表有更好的性能，大家想想是不是，大量的时候，我对于用户表来说，只有用户ID，用户名，口令，用户角色等会被经常使用。小一点的表总是会有 好的性能。
* **示例二**： 你有一个叫 “last\_login” 的字段，它会在每次用户登录时被更新。但是，每次更新时会导致该表的查询缓存被清空。所以，你可以把这个字段放到另一个表中，这样就不会影响你对用户ID，用户名，用户角色的不停地读取了，因为查询缓存会帮你增加很多性能。

另外，你需要注意的是，这些被分出去的字段所形成的表，你不会经常性地去Join他们，不然的话，这样的性能会比不分割时还要差，而且，会是极数级的下降。

## 拆分大的 DELETE 或 INSERT 语句

如果你需要在一个在线的网站上去执行一个大的 DELETE 或 INSERT 查询，你需要非常小心，要避免你的操作让你的整个网站停止相应。因为这两个操作是会锁表的，表一锁住了，别的操作都进不来了。

Apache 会有很多的子进程或线程。所以，其工作起来相当有效率，而我们的服务器也不希望有太多的子进程，线程和数据库链接，这是极大的占服务器资源的事情，尤其是内存。

如果你把你的表锁上一段时间，比如30秒钟，那么对于一个有很高访问量的站点来说，这30秒所积累的访问进程/线程，数据库链接，打开的文件数，可能不仅仅会让你泊WEB服务Crash，还可能会让你的整台服务器马上掛了。

所以，如果你有一个大的处理，你定你一定把其拆分，使用 LIMIT 条件是一个好的方法。

while (1) {

    //每次只做1000条

    mysql\_query("DELETE FROM logs WHERE log\_date <= '2009-11-01' LIMIT 1000");

    if (mysql\_affected\_rows() == 0) {

        // 没得可删了，退出！

        break;

    }

    // 每次都要休息一会儿

    usleep(50000);

}

## 越小的列会越快

对于大多数的数据库引擎来说，硬盘操作可能是最重大的瓶颈。所以，把你的数据变得紧凑会对这种情况非常有帮助，因为这减少了对硬盘的访问。

如果一个表只会有几列罢了（比如说字典表，配置表），那么，我们就没有理由使用 INT 来做主键，使用 MEDIUMINT, SMALLINT 或是更小的 TINYINT 会更经济一些。如果你不需要记录时间，使用 DATE 要比 DATETIME 好得多。

## 选择正确的存储引擎

在 MySQL 中有两个存储引擎 MyISAM 和 InnoDB，每个引擎都有利有弊。酷壳以前文章《[MySQL: InnoDB 还是 MyISAM?](http://coolshell.cn/articles/652.html)》讨论和这个事情。

MyISAM 适合于一些需要大量查询的应用，但其对于有大量写操作并不是很好。甚至你只是需要update一个字段，整个表都会被锁起来，而别的进程，就算是读进程都 无法操作直到读操作完成。另外，MyISAM 对于 SELECT COUNT(\*) 这类的计算是超快无比的。

InnoDB 的趋势会是一个非常复杂的存储引擎，对于一些小的应用，它会比 MyISAM 还慢。他是它支持“行锁” ，于是在写操作比较多的时候，会更优秀。并且，他还支持更多的高级应用，比如：事务。

## 使用一个对象关系映射器（Object Relational Mapper）

使用 ORM (Object Relational Mapper)，你能够获得可靠的性能增涨。一个ORM可以做的所有事情，也能被手动的编写出来。但是，这需要一个高级专家。

ORM 的最重要的是“Lazy Loading”，也就是说，只有在需要的去取值的时候才会去真正的去做。但你也需要小心这种机制的副作用，因为这很有可能会因为要去创建很多很多小的查询反而会降低性能。

ORM 还可以把你的SQL语句打包成一个事务，这会比单独执行他们快得多得多。