**SQL编码规范  
（V1.00）**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 文 档 信 息 | | | | | | | |
| **文档名称：** | | **SQL编码规范** | | | | | |
| **电子文档：** | |  | | | | | |
| **版本号：** | | 1.00 | | **密级：** | 保密 | | |
| **文档编号：** | |  | | | | | |
| **编写人：** | | 黄业明 | | **日期：** | 2007-03-27 | | |
| **校对人：** | |  | | **日期：** |  | | |
| **审核人：** | |  | | **日期：** |  | | |
| **批准人：** | |  | | **日期：** |  | | |
| 更 改 记 录 | | | | | | | |
| **更改**  **序号** | **更改原因** | **更改**  **页码** | **更改前版本号** | **更改后版本号** | **更改人** | **生效日期** | **备 注** |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

**目录**

1 注释规范 6

1.1. 一般性注释 6

1.2. 函数文本注释 6

2. 排版格式 7

2.1. 缩进 7

2.2. 换行 8

2.3. 空格 10

2.4. 大小写 10

2.5. 对齐 10

3. 命名规则 11

3.1. 输入变量 11

3.2. 输出变量 11

3.3. 内部变量 11

3.4. 游标命名 11

4. 编码规范 11

4.1. 不等于统一使用"<>" 11

4.2. 使用表的别名 11

4.3. 使用SELECT语句时，必须指出列名 12

4.4. 使用INSERT语句时，必须指定插入的字段名。 12

4.5. 减少子查询的使用 12

4.6. 适当添加索引以提高查询效率 12

4.7. 不要在WHERE字句中对索引列施以函数 12

4.8. 不要使用数据库的类型自动转换功能，使用显式的类型转换 12

4.9. 应使用变量绑定实现SQL语句共享，避免使用硬编码 12

4.10. 用执行计划分析SQL性能 14

附录A：Oracle SQL性能优化 14

A.1 选用适合的ORACLE优化器 14

A.2 访问TABLE的方式 15

A.3 共享SQL语句 15

A.4 选择最有效率的表名顺序(只在基于规则的优化器中有效) 17

A.5 WHERE子句中的连接顺序 19

A.6 SELECT子句中避免使用 ‘ \* ‘ 19

A.7 减少访问数据库的次数 19

A.8 使用DECODE函数来减少处理时间 21

A.9 整合简单,无关联的数据库访问 22

A.10 删除重复记录 23

A.11 用TRUNCATE替代DELETE全表记录 23

A.12 尽量多使用COMMIT 23

A.13 计算记录条数 24

A.14 用Where子句替换HAVING子句 24

A.15 减少对表的查询 24

A.16 通过内部函数提高SQL效率 26

A.17 使用表的别名(Alias) 27

A.18 用EXISTS替代IN 27

A.19 用NOT EXISTS替代NOT IN 28

A.20 用表连接替换EXISTS 29

A.21 用EXISTS替换DISTINCT 30

A.22 识别'低效执行'的SQL语句 31

A.23 使用TKPROF 工具来查询SQL性能状态 31

A.24 用EXPLAIN PLAN 分析SQL语句 32

A.25 用索引提高效率 34

A.26 索引的操作 34

A.27 基础表的选择 36

A.28 多个平等的索引 37

A.29 等式比较和范围比较 38

A.30 不明确的索引等级 39

A.31 强制索引失效 40

A.32 避免在索引列上使用计算 41

A.33 自动选择索引 42

A.34 避免在索引列上使用NOT 42

A.35 用>=替代> 44

A.36 用UNION替换OR (适用于索引列) 44

A.37 用IN来替换OR 48

A.38 避免在索引列上使用IS NULL和IS NOT NULL 49

A.39 总是使用索引的第一个列 50

A.40 ORACLE内部操作 51

A.41 用UNION-ALL 替换UNION ( 如果有可能的话) 51

A.42 使用提示(Hints) 53

A.43 用WHERE替代ORDER BY 54

A.44 避免改变索引列的类型 56

A.45 需要当心的WHERE子句 57

A.46 连接多个扫描 58

A.47 CBO下使用更具选择性的索引 60

A.48 避免使用耗费资源的操作 60

A.49 优化GROUP BY 61

A.50 使用日期 62

A.51 使用显式的游标(CURSORs) 62

A.52 优化EXPORT和IMPORT 62

A.53 分离表和索引 63

# 注释规范

## 一般性注释

### 创建每一数据库对象时都要加上COMMENT ON注释，以说明该对象的功能和用途；建表时，对某些数据列也要加上COMMENT ON注释，以说明该列和/或列取值的含义。

### 注释语法包含两种情况：单行注释、多行注释

单行注释：注释前有两个连字符（--）。

多行注释：符号/\*和\*/之间的内容为注释内容。

## 函数文本注释

### 在每一个块和过程（存储过程、函数、包、触发器、视图等）的开头放置注释

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*name : --函数名

\*function : --函数功能

\*input : --输入参数

\*output : --输出参数

\*author : --作者

\*CreateDate : --创建时间

\*UpdateDate : --函数更改信息（包括作者、时间、更改内容等）

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

CREATE [OR REPLACE] PROCEDURE dfsp\_xxx

…

### 对传入参数的含义进行说明。如果取值范围确定，也一并说明。取值有特定含义的变量（如boolean类型变量），给出每个值的含义。

### 在每一个变量声明的旁边添加注释。说明该变量要用作什么。

通常使用单行注释即可，例如

login\_id VARCHAR2(32) NOT NULL, -- 会员标识

### 对存储过程的任何修改，都需要在注释最后添加修改人、修改日期及修改原因等信息

### 避免在一行代码或表达式的中间插入注释

### 在程序块的结束行右方加注释，以表明程序块结束

### 在块的每个主要部分之前添加注释

在块的每个主要部分之前增加注释，解释下—组语句目的，说明该段语句及算法的目的以及要得到的结果，但不要对其细节进行过多的描述。

### 对程序分支必须书写注释

# 排版格式

## 缩进

### 存储过程中的SQL语句

* 低级别语句在高级别语句后的，缩进4个空格。
* 同一语句不同部分的缩进，如果为子句，则为4个空格，如果与上一行某部分有密切联系的，则缩至与其对齐。

### 对于Pro\*C, Java等代码里的SQL字符串, 每一行字符串不可以空格开头。

## 换行

### 一行最长不能超过80字符

### SELECT/FROM/WHERE/ORDER BY/GROUP BY等子句应独占一行。

### SQL语句中间不允许出现空行

### SELECT子句内容如果只有一项，应与 SELECT 同占一行。

### SELECT子句内容如果多于一项，每一项都应独占一行，并在对应 SELECT的基础上向右缩进8个空格。

### FROM子句内容如果只有一项，应与 FROM同占一行。

### FROM子句内容如果多于一项，每一项都应独占一行，并在对应FROM的基础上向右缩进4个空格。

### WHERE子句内容如果只有一项，应与 WHERE同占一行。

### WHERE子句的条件如果有多项，每一个条件应独占一行，并以AND开头，并在对应WHERE的基础上向右缩进4个空格。

* + - 1. 进1个Tab或者4个字符。

|  |
| --- |
| 示例： |
| //SELECT语句书写的正确示例  SELECT bill\_no,  FROM mft\_list  WHERE manifest\_no =‘000000000000000007’;  SELECT  list.manifest\_no,  list.list\_no,  stat.list\_stat  FROM  mft\_list list,  list\_stat stat  WHERE  list.manifest\_no = stat.manifest\_no  AND stat.stat != 2; |

### （UPDATE）SET子句内容如果有一项，应与 SET同占一行。

### （UPDATE）SET子句内容如果有多项，每一项应独占一行，并在对应SET的基础上向右缩进4个空格。

|  |
| --- |
| 示例： |
| //UPDATE语句书写的正确示例  UPDATE list\_stat  SET  list\_stat = '2',  parent = '0'  WHERE list\_no = 'bill010'; |

### INSERT 子句左/右括号以及每个表字段应独占一行，其中括号无缩进，表字段在对应括号的基础上向右缩进4个空格。

### VALUES子句左/右括号以及每一项的值应独占一行，其中括号无缩进，每一项的值在对应括号的基础上向右缩进4个字符。

|  |
| --- |
| 示例： |
| //INSERT语句书写的正确示例  INSERT INTO list\_stat  (  list\_no,  　 list\_stat,  　 parent,  　 manifest\_no,  div\_flag  )  VALUES  (  'bill020',  '1',  '0',  　 '000000000000007807',  '0'  ); |

## 空格

### SQL内算数运算符、逻辑运算符连接的两个元素之间必须用空格分隔

### 逗号之后必须接一个空格

### 关键字、保留字和左括号之间必须有一个空格

## 大小写

### SQL语句中出现的所有表名、表别名、字段名、序列等数据库对象都应小写

### SQL 语句中出现的系统保留字、内置函数名、SQL保留字、绑定变量等都应大写。

## 对齐

### 变量初始化赋值时，各变量列对齐，赋值号列对齐，被赋值列对齐；

# 变量命名规则

## 输入变量

in + 变量含义的英文单词，单词的首字母大写，单词之间用”\_”分割。

## 输出变量

out + 变量含义的英文单词，单词的首字母大写，单词之间用”\_”分割。

## 内部变量

v + 变量含义的英文单词，单词的首字母大写，单词之间用”\_”分割。

## 游标命名

CURSOR\_表名

# BC命名规范

## 查询类BC

命名规则：cQ[表名缩写]By[查询条件]

例如：

cQCustInfoByCustId

cQUserInfoByCustId

对于查询条件是多个的，可选择一个具有代表性的字段作为查询条件，或者新定义一个新的单词

例如根据客户类型、证件类型、证件号码查询客户信息的BC，命名为cQCustInfoByIccid

## 删除类BC

根据主键删除的， 命名规则：cD[表名缩写]

不根据主键删除的， 命名规则：cD[表名缩写]By[删除条件]

例如：

根据用户标识删除用户扩展属性，用户标识是非主键，BC命名为cDUserAddInfoByIdNo

根据主键(用户标识+属性代码)删除的， BC命名为 cDUserAddInfo

## 修改类BC

根据主键修改的， 命名规则：cU[字段信息]Of[表名缩写]

不根据主键修改的， 命名规则：cU[字段信息]Of[表名缩写]By[修改条件]

例如：

根据群成员标识 修改群成员的状态， BC命名为cUStateOfUserGroupMbrInfo

根据群标识修改群成员表中其所有成员的状态，BC命名为cUStateOfUserGroupMbrInfoByGrpId

## 插入类BC

插入本表， 命名规则：cI[表名缩写]

插入历史表：

1、插入本表时，插入历史表， 命名规则：cI[历史表表名缩写]

2、删除、修改本表时，将数据备份到历史

i、根据主键删除修改时备份入历史， 命名规则为：cI[历史表表名缩写]Bak

ii、不根据主键删除修改时备份入历史，命名规则为：cI[历史表表名缩写]BakBy[条件]

例如：

插入用户扩展信息表 BC命名为cIUserAddInfo

插入用户扩展信息表时插用户扩展信息历史表 BC命名为cIUserAddInfoHis

根据(用户标识+属性代码)删除用户扩展信息之前，将预删除记录备份到历史 BC命名为cIUserAddInfoHisBak

根据用户标识删除用户扩展信息之前，将预删除记录备份到历史 BC命名为cIUserAddInfoHisBakByIdNo

## 多表联合查询类BC

命名规则：cGet[获取信息描述]By[条件]

例如：

根据用户标识联合查询用户信息表和客户信息表，获取客户基本信息 BC命名为cGetCustInfoByIdNo

# 编码规范

## 不等于统一使用"<>"

Oracle认为"!="和"<>"是等价的，都代表不等于的意义。为了统一，不等于一律使用"<>"表示。

## 使用表的别名

多表连接时，应为每个表使用别名，别名要简短最好一个字母，且能代表一定意义，所有被引用列要加上表的别名。

## 使用SELECT语句时，必须指出列名

不要使用列的序号或者用“\*”替代所有列名。

## 使用INSERT语句时，必须指定插入的字段名。

## 减少子查询的使用

子查询除了可读性差之外，还在一定程度上影响了SQL运行效率. 请尽量减少子查询的使用，采用其他效率更高、可读性更好的方式替代

## 适当添加索引以提高查询效率

适当添加索引可以大幅度的提高检索速度(具体的优化方法见附录A性能优化部分)

## 不要在WHERE字句中对索引列施以函数

## 不要使用数据库的类型自动转换功能，使用显式的类型转换

## 应使用变量绑定实现SQL语句共享，避免使用硬编码

### 不允许直接拼写SQL语句，而要使用绑定变量。

例如：

例1：此种写法不允许：

init(dynStmt);

sprintf(dynStmt,"INSERT INTO wChg%s(id\_no,total\_date,login\_accept,sm\_code,belong\_code,phone\_no,org\_code,login\_no,op\_code,op\_time,machine\_code,cash\_pay,check\_pay,sim\_fee,machine\_fee,innet\_fee,choice\_fee,other\_fee,hand\_fee,deposit,back\_flag,encrypt\_fee,system\_note,op\_note)VALUES(%ld,TO\_NUMBER(%s),%ld,'%s','%s','%s','%s','%s','%s',TO\_DATE('%s','yyyymmdd hh24:mi:ss'),'zz',0,0,0,0,0,0,0,0,0,'0',0,'%s','%s')",YearMonth,idNo,totalDate,LoginAccept,smCode,belongCode,phoneNo,orgCode,loginNo,opCode,opTime,systemNote,systemNote);

EXEC SQL PREPARE ins\_stmt FROM :dynStmt;

EXEC SQL EXECUTE ins\_stmt ;

例2：可以使用：

init(dynStmt);

sprintf(dynStmt, " INSERT INTO wLoginOpr%s"

"("

"total\_date,login\_accept,op\_code,pay\_type,pay\_money,"

"sm\_code,id\_no,phone\_no,org\_code,loin\_no,op\_time,op\_note,ip\_addr"

")"

"VALUES"

"("

"TO\_NUMBER(:v1), :v2, :v3, :v4, :v5,"

":v6, :v7, :v8, :v9, :v10, TO\_DATE(:v11, 'yyyymmdd hh24:mi:ss'), :v12, :v13"

")",

YearMonth);

EXEC SQL PREPARE prepare1 FROM :dynStmt;

EXEC SQL EXECUTE prepare1 USING :totalDate, :LoginAccept, :opCode, :payType,:handFee,

:smCode,:idNo, :phoneNo, :orgCode,:loginNo,:opTime, :opNote,:ipAddress;

例3：可以使用：

EXEC SQL INSERT INTO dCustMsgAdd

(

id\_no,busi\_type,user\_type,field\_code,field\_order,field\_value,

other\_value

)

VALUES

(

:idNo, :busiType, :userType, :fieldCode, :fieldOrder,:fieldValue,

:otherValue

);

### 执行相同操作的SQL语句必须使用相同名字的绑定变量。

例如：第一组的两个SQL语句，绑定变量是相同的，而第二组中的两个语句绑定变量不同，即使赋于不同的绑定变量相同的值也不能使这两个SQL语句相同，达不到共享SQL语句目的。

a)第一组

select pin , name from people where pin = :blk1.pin;

select pin , name from people where pin = :blk1.pin;

b)第二组

select pin , name from people where pin = :blk1.ot\_ind;

select pin , name from people where pin = :blk1.ov\_ind;

## 用执行计划分析SQL性能

EXPLAIN PLAN是一个很好的分析SQL语句的工具，它可以在不执行SQL的情况下分析语句. 通过分析，我们就可以知道ORACLE是怎样连接表，使用什么方式扫描表（索引扫描或全表扫描），以及使用到的索引名称, 按照从里到外，从上到下的次序解读分析的结果。EXPLAIN PLAN的分析结果是用缩进的格式排列的，最内部的操作将最先被解读，如果两个操作处于同一层中，带有最小操作号的将首先被执行。

目前许多第三方的工具如PLSQL Developer和TOAD等都提供了极其方便的EXPLAIN PLAN工具。

# 附录A：Oracle SQL性能优化

* 1. 选用适合的ORACLE优化器
* ORACLE的优化器共有3种:
  + RULE (基于规则)
  + COST (基于成本)
  + CHOOSE (选择性)

设置缺省的优化器,可以通过对init.ora文件中OPTIMIZER\_MODE参数的各种声明,如 RULE,COST,CHOOSE,ALL\_ROWS,FIRST\_ROWS. 你当然也可以在SQL句级或是会话 (session)级对其进行覆盖.

* + - 为了使用基于成本的优化器(CBO, Cost-Based Optimizer), 你必须经常运行analyze 命令,以增加数据库中的对象统计信息(object statistics)的准确性.

如果数据库的优化器模式设置为选择性(CHOOSE),那么实际的优化器模式将和是否运 行过analyze命令有关. 如果table已经被analyze过, 优化器模式将自动成为CBO, 反 之,数据库将采用RULE形式的优化器.

* + - 在缺省情况下,ORACLE采用CHOOSE优化器, 为了避免那些不必要的全表扫描(full table scan) , 你必须尽量避免使用CHOOSE优化器,而直接采用基于规则或者基于成本的优化器
  1. 访问TABLE的方式

ORACLE 采用两种访问表中记录的方式:

* + - 全表扫描

全表扫描就是顺序地访问表中每条记录. ORACLE采用一次读入多个数据块(database block)的方式优化全表扫描.

* + - 通过ROWID访问表

你可以采用基于ROWID的访问方式情况,提高访问表的效率, ROWID包含了表中记录 的物理位置信息. ORACLE采用索引(INDEX)实现了数据和存放数据的物理位置 (ROWID)之间的联系. 通常索引提供了快速访问ROWID的方法, 因此那些基于索引列 的查询就可以得到性能上的提高.

* 1. 共享SQL语句

为了不重复解析相同的SQL语句,在第一次解析之后, ORACLE将SQL语句存放在内存中.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,如果它

和之前的执行过的语句完全相同, ORACLE就能很快获得已经被解析的语句以及最好的

执行路径.

ORACLE的这个功能大大地提高了SQL的执行性能并节省了内存的使用.

可惜的是ORACLE只对简单的表提供高速缓冲(cache buffering) ,这个功能并不适用于多表连接查询.

数据库管理员必须在init.ora中为这个区域设置合适的参数,当这个内存区域越大,就可以保留更多的语句,当然被共享的可能性也就越大了.

当你向ORACLE 提交一个SQL语句,ORACLE会首先在这块内存中查找相同的语句.

这里需要注明的是,ORACLE对两者采取的是一种严格匹配,要达成共享,SQL语句必须

完全相同(包括空格,换行等).

共享的语句必须满足三个条件:

* + - 字符级的比较: 当前被执行的语句和共享池中的语句必须完全相同

例如:

SELECT \* FROM EMP;

和下列每一个都不同

SELECT \* from EMP;

Select \* From Emp;

SELECT \* FROM EMP;

* + - 两个语句所指的对象必须完全相同

例如:

用户 对象名 如何访问

Jack sal\_limit private synonym

Jack Work\_city public synonym

Jack Plant\_detail public synonym

Jill sal\_limit private synonym

Jill Work\_city public synonym

Jill Plant\_detail table owner

考虑一下下列SQL语句能否在这两个用户(Jack, Jill)之间共享?

* SQL: SELECT MAX(sal\_cap) FROM sal\_limit;
* 能否共享: 不能
* 原因: 每个用户都有一个private synonym - sal\_limit , 它们是不同的对象

* SQL: SELECT COUNT(0) FROM work\_city WHERE sdesc LIKE 'NEW%';
* 能否共享: 不能
* 原因: 两个用户访问相同的对象public synonym - work\_city

* SQL: SELECT a.sdesc,b.location FROM work\_city a , plant\_detail b

WHERE a.city\_id = b.city\_id

* 能否共享: 不能
* 原因: 用户jack 通过public synonym访问plant\_detail 而jill 是表的所有者,对象不同.
  + 两个SQL语句中必须使用相同的名字的绑定变量(bind variables)

例如： 第一组的两个SQL语句是相同的(可以共享),而第二组中的两个语句是不同的 (即使在运行时,赋于不同的绑定变量相同的值)

a)第一组

select pin , name from people where pin = :blk1.pin;

select pin , name from people where pin = :blk1.pin;

b)第二组

select pin , name from people where pin = :blk1.ot\_ind;

select pin , name from people where pin = :blk1.ov\_ind;

* 1. 选择最有效率的表名顺序(只在基于规则的优化器中有效)

ORACLE的解析器按照从右到左的顺序处理FROM子句中的表名.

因此FROM子句中写在最后的表(基础表 driving table)将被最先处理. 在FROM子句中 包含多个表的情况下,你必须选择记录条数最少的表作为基础表.

当ORACLE处理多个表时, 会运用排序及合并的方式连接它们.首先,扫描第一个表 (FROM子句中最后的那个表)并对记录进行派序,然后扫描第二个表(FROM子句中最后第二个表),最后将所有从第二个表中检索出的记录与第一个表中合适记录进行合并.

例如:

表 TAB1 16,384 条记录

表 TAB2 1 条记录

* 选择TAB2作为基础表 (最好的方法)

select count(\*) from tab1,tab2 执行时间0.96秒

* 选择TAB2作为基础表 (不佳的方法)

select count(\*) from tab2,tab1 执行时间26.09秒

* 如果有3个以上的表连接查询, 那就需要选择交叉表(intersection table)作为基础表, 交叉表是指那个被其他表所引用的表.

例如:

EMP表描述了LOCATION表和CATEGORY表的交集.

SELECT \*

FROM LOCATION L ,

CATEGORY C,

EMP E

WHERE E.EMP\_NO BETWEEN 1000 AND 2000

AND E.CAT\_NO = C.CAT\_NO

AND E.LOCN = L.LOCN

将比下列SQL更有效率

SELECT \*

FROM EMP E ,

LOCATION L ,

CATEGORY C

WHERE E.CAT\_NO = C.CAT\_NO

AND E.LOCN = L.LOCN

AND E.EMP\_NO BETWEEN 1000 AND 2000

* 1. WHERE子句中的连接顺序

ORACLE采用自下而上的顺序解析WHERE子句,根据这个原理,表之间的连接必须写在其他WHERE条件之前, 那些可以过滤掉最大数量记录的条件必须写在WHERE子句的末尾.

例如:

* (低效,执行时间156.3秒)

SELECT …

FROM EMP E

WHERE SAL > 50000

AND JOB = ‘MANAGER'

AND 25 < (SELECT COUNT(\*) FROM EMP WHERE MGR=E.EMPNO);

* (高效,执行时间10.6秒)

SELECT …

FROM EMP E

WHERE 25 < (SELECT COUNT(\*) FROM EMP WHERE MGR=E.EMPNO)

AND SAL > 50000

AND JOB = ‘MANAGER';

* 1. SELECT子句中避免使用 ‘ \* ‘

当你想在SELECT子句中列出所有的COLUMN时, 使用动态SQL列引用 ‘\*’ 是一个方便的方法. 不幸的是,这是一个非常低效的方法. 实际上,ORACLE在解析的过程中, 会将‘\*’ 依次转换成所有的列名, 这个工作是通过查询数据字典完成的, 这意味着将耗费更多的时间.

* 1. 减少访问数据库的次数

当执行每条SQL语句时, ORACLE在内部执行了许多工作: 解析SQL语句, 估算索引的利用率, 绑定变量 , 读数据块等等. 由此可见, 减少访问数据库的次数 , 就能实际上减少ORACLE的工作量.

例如,

以下有三种方法可以检索出雇员号等于0342或0291的职员.

* 方法1 (最低效)

SELECT EMP\_NAME , SALARY , GRADE

FROM EMP

WHERE EMP\_NO = 342;

SELECT EMP\_NAME , SALARY , GRADE

FROM EMP

WHERE EMP\_NO = 291;

* 方法2 (次低效)

DECLARE

CURSOR C1 (E\_NO NUMBER) IS

SELECT EMP\_NAME,SALARY,GRADE

FROM EMP

WHERE EMP\_NO = E\_NO;

BEGIN

OPEN C1(342);

FETCH C1 INTO …,..,.. ;

…..

OPEN C1(291);

FETCH C1 INTO …,..,.. ;

CLOSE C1;

END;

* 方法3 (高效)

SELECT A.EMP\_NAME , A.SALARY , A.GRADE,

B.EMP\_NAME , B.SALARY , B.GRADE

FROM EMP A,EMP B

WHERE A.EMP\_NO = 342

OR B.EMP\_NO = 291;

注意:

在SQL\*Plus , SQL\*Forms和Pro\*C中重新设置ARRAYSIZE参数, 可以增加每次数据库访问的检索数据量 ,建议值为200

* 1. 使用DECODE函数来减少处理时间

使用DECODE函数可以避免重复扫描相同记录或重复连接相同的表.

例如:

SELECT COUNT(\*)，SUM(SAL)

FROM　EMP

WHERE DEPT\_NO = 0020

AND ENAME LIKE　‘SMITH%';

SELECT COUNT(\*)，SUM(SAL)

FROM　EMP

WHERE DEPT\_NO = 0030

AND ENAME LIKE　‘SMITH%';

你可以用DECODE函数高效地得到相同结果

SELECT COUNT(DECODE(DEPT\_NO,0020,'X',NULL)) D0020\_COUNT,

COUNT(DECODE(DEPT\_NO,0030,'X',NULL)) D0030\_COUNT,

SUM(DECODE(DEPT\_NO,0020,SAL,NULL)) D0020\_SAL,

SUM(DECODE(DEPT\_NO,0030,SAL,NULL)) D0030\_SAL

FROM EMP

WHERE ENAME LIKE ‘SMITH%';

类似的,DECODE函数也可以运用于GROUP BY 和ORDER BY子句中

* 1. 整合简单,无关联的数据库访问

如果你有几个简单的数据库查询语句,你可以把它们整合到一个查询中(即使它们之间没有关系)

例如:

SELECT NAME

FROM EMP

WHERE EMP\_NO = 1234;

SELECT NAME

FROM DPT

WHERE DPT\_NO = 10 ;

SELECT NAME

FROM CAT

WHERE CAT\_TYPE = ‘RD';

上面的3个查询可以被合并成一个:

SELECT E.NAME , D.NAME , C.NAME

FROM CAT C , DPT D , EMP E,DUAL X

WHERE NVL(‘X',X.DUMMY) = NVL(‘X',E.ROWID(+))

AND NVL(‘X',X.DUMMY) = NVL(‘X',D.ROWID(+))

AND NVL(‘X',X.DUMMY) = NVL(‘X',C.ROWID(+))

AND E.EMP\_NO(+) = 1234

AND D.DEPT\_NO(+) = 10

AND C.CAT\_TYPE(+) = ‘RD';

* 1. 删除重复记录

最高效的删除重复记录方法 (因为使用了ROWID)

DELETE FROM EMP E

WHERE E.ROWID > (SELECT MIN(X.ROWID)

FROM EMP X

WHERE X.EMP\_NO = E.EMP\_NO);

* 1. 用TRUNCATE替代DELETE全表记录

当删除表中的所有记录时,在通常情况下, 回滚段(rollback segments ) 用来存放可以被恢复的信息. 如果你没有COMMIT事务,ORACLE会将数据恢复到删除之前的状态(准确地说是

恢复到执行删除命令之前的状况)

而当运用TRUNCATE时, 回滚段不再存放任何可被恢复的信息.当命令运行后,数据不能被恢复.因此很少的资源被调用,执行时间也会很短.

* 1. 尽量多使用COMMIT

只要有可能,在程序中尽量多使用COMMIT, 这样程序的性能得到提高,需求也会因为COMMIT所释放的资源而减少.

COMMIT所释放的资源:

* 回滚段上用于恢复数据的信息.
* 被程序语句获得的锁
* redo log buffer 中的空间
* ORACLE为管理上述3种资源中的内部花费
  1. 计算记录条数

和一般的观点相反, count(\*) 比count(1)稍快 , 当然如果可以通过索引检索,对索引列的计数仍旧是最快的. 例如 COUNT(EMPNO)

(译者按: 在CSDN论坛中,曾经对此有过相当热烈的讨论, 作者的观点并不十分准确,通过实际的测试,上述三种方法并没有显著的性能差别)

* 1. 用Where子句替换HAVING子句

避免使用HAVING子句, HAVING 只会在检索出所有记录之后才对结果集进行过滤. 这个处理需要排序,总计等操作. 如果能通过WHERE子句限制记录的数目,那就能减少这方面的开销.

例如:

* 低效:

SELECT REGION，AVG(LOG\_SIZE)

FROM LOCATION

GROUP BY REGION

HAVING REGION != ‘SYDNEY'

AND REGION != ‘PERTH'

* 高效

SELECT REGION，AVG(LOG\_SIZE)

FROM LOCATION

WHERE REGION != ‘SYDNEY'

AND REGION != ‘PERTH'

GROUP BY REGION

* 1. 减少对表的查询

在含有子查询的SQL语句中,要特别注意减少对表的查询

例如:

* 低效

SELECT TAB\_NAME

FROM TABLES

WHERE TAB\_NAME = ( SELECT TAB\_NAME

FROM TAB\_COLUMNS

WHERE VERSION = 604)

AND　DB\_VER= ( SELECT DB\_VER

FROM TAB\_COLUMNS

WHERE VERSION = 604)

* 高效

SELECT TAB\_NAME

FROM TABLES

WHERE (TAB\_NAME,DB\_VER)

= ( SELECT TAB\_NAME,DB\_VER)

FROM TAB\_COLUMNS

WHERE VERSION = 604)

Update 多个Column 例子:

* 低效:

UPDATE EMP

SET EMP\_CAT = (SELECT MAX(CATEGORY) FROM EMP\_CATEGORIES),

SAL\_RANGE = (SELECT MAX(SAL\_RANGE) FROM EMP\_CATEGORIES)

WHERE EMP\_DEPT = 0020;

* 高效:

UPDATE EMP

SET (EMP\_CAT, SAL\_RANGE)

= (SELECT MAX(CATEGORY) , MAX(SAL\_RANGE)

FROM EMP\_CATEGORIES)

WHERE EMP\_DEPT = 0020;

* 1. 通过内部函数提高SQL效率

SELECT H.EMPNO,E.ENAME,H.HIST\_TYPE,T.TYPE\_DESC,COUNT(\*)

FROM HISTORY\_TYPE T,EMP E,EMP\_HISTORY H

WHERE H.EMPNO = E.EMPNO

AND H.HIST\_TYPE = T.HIST\_TYPE

GROUP BY H.EMPNO,E.ENAME,H.HIST\_TYPE,T.TYPE\_DESC;

通过调用下面的函数可以提高效率.

FUNCTION LOOKUP\_HIST\_TYPE(TYP IN NUMBER) RETURN VARCHAR2

AS

TDESC VARCHAR2(30);

CURSOR C1 IS

SELECT TYPE\_DESC

FROM HISTORY\_TYPE

WHERE HIST\_TYPE = TYP;

BEGIN

OPEN C1;

FETCH C1 INTO TDESC;

CLOSE C1;

RETURN (NVL(TDESC,'?'));

END;

FUNCTION LOOKUP\_EMP(EMP IN NUMBER) RETURN VARCHAR2

AS

ENAME VARCHAR2 (30);

CURSOR C1 IS

SELECT ENAME

FROM EMP

WHERE EMPNO=EMP;

BEGIN

OPEN C1;

FETCH C1 INTO ENAME;

CLOSE C1;

RETURN (NVL(ENAME,'?'));

END;

SELECT H.EMPNO,LOOKUP\_EMP(H.EMPNO),

H.HIST\_TYPE,LOOKUP\_HIST\_TYPE(H.HIST\_TYPE),COUNT(\*)

FROM EMP\_HISTORY H

GROUP BY H.EMPNO , H.HIST\_TYPE;

* 1. 使用表的别名(Alias)

当在SQL语句中连接多个表时, 请使用表的别名并把别名前缀于每个Column上.这样一来,就可以减少解析的时间并减少那些由Column歧义引起的语法错误.

(译者注: Column歧义指的是由于SQL中不同的表具有相同的Column名,当SQL语句中出现这个Column时,SQL解析器无法判断这个Column的归属)

* 1. 用EXISTS替代IN

在许多基于基础表的查询中,为了满足一个条件,往往需要对另一个表进行联接.在这种情况下, 使用EXISTS(或NOT EXISTS)通常将提高查询的效率.

* 低效:

SELECT \*

FROM EMP (基础表)

WHERE EMPNO > 0

AND DEPTNO IN (SELECT DEPTNO

FROM DEPT

WHERE LOC = ‘MELB')

* 高效:

SELECT \*

FROM EMP (基础表)

WHERE EMPNO > 0

AND EXISTS (SELECT ‘X'

FROM DEPT

WHERE DEPT.DEPTNO = EMP.DEPTNO

AND LOC = ‘MELB')

(译者按: 相对来说,用NOT EXISTS替换NOT IN 将更显著地提高效率,下一节中将指出)

* 1. 用NOT EXISTS替代NOT IN

在子查询中,NOT IN子句将执行一个内部的排序和合并. 无论在哪种情况下,NOT IN都是最低效的 (因为它对子查询中的表执行了一个全表遍历). 为了避免使用NOT IN ,我们可以把它改写成外连接(Outer Joins)或NOT EXISTS.

例如:

SELECT …

FROM EMP

WHERE DEPT\_NO NOT IN (SELECT DEPT\_NO

FROM DEPT

WHERE DEPT\_CAT='A');

为了提高效率.改写为:

* (方法一: 高效)

SELECT ….

FROM EMP A,DEPT B

WHERE A.DEPT\_NO = B.DEPT(+)

AND B.DEPT\_NO IS NULL

AND B.DEPT\_CAT(+) = ‘A'

* (方法二: 最高效)

SELECT ….

FROM EMP E

WHERE NOT EXISTS (SELECT ‘X'

FROM DEPT D

WHERE D.DEPT\_NO = E.DEPT\_NO

AND DEPT\_CAT = ‘A');

* 1. 用表连接替换EXISTS

通常来说 , 采用表连接的方式比EXISTS更有效率

SELECT ENAME

FROM EMP E

WHERE EXISTS (SELECT ‘X'

FROM DEPT

WHERE DEPT\_NO = E.DEPT\_NO

AND DEPT\_CAT = ‘A');

(更高效)

SELECT ENAME

FROM DEPT D,EMP E

WHERE E.DEPT\_NO = D.DEPT\_NO

AND DEPT\_CAT = ‘A' ;

(译者按: 在RBO的情况下,前者的执行路径包括FILTER,后者使用NESTED LOOP)

* 1. 用EXISTS替换DISTINCT

当提交一个包含一对多表信息(比如部门表和雇员表)的查询时,避免在SELECT子句中使用DISTINCT. 一般可以考虑用EXIST替换

例如:

* 低效:

SELECT DISTINCT DEPT\_NO,DEPT\_NAME

FROM DEPT D,EMP E

WHERE D.DEPT\_NO = E.DEPT\_NO

* 高效:

SELECT DEPT\_NO,DEPT\_NAME

FROM DEPT D

WHERE EXISTS ( SELECT ‘X'

FROM EMP E

WHERE E.DEPT\_NO = D.DEPT\_NO);

EXISTS 使查询更为迅速,因为RDBMS核心模块将在子查询的条件一旦满足后,立刻返回结果.

* 1. 识别'低效执行'的SQL语句

用下列SQL工具找出低效SQL:

SELECT EXECUTIONS , DISK\_READS, BUFFER\_GETS,

ROUND((BUFFER\_GETS-DISK\_READS)/BUFFER\_GETS,2) Hit\_radio,

ROUND(DISK\_READS/EXECUTIONS,2) Reads\_per\_run,

SQL\_TEXT

FROM V$SQLAREA

WHERE EXECUTIONS>0

AND BUFFER\_GETS > 0

AND (BUFFER\_GETS-DISK\_READS)/BUFFER\_GETS < 0.8

ORDER BY 4 DESC;

(译者按: 虽然目前各种关于SQL优化的图形化工具层出不穷,但是写出自己的SQL工具来解决问题始终是一个最好的方法)

* 1. 使用TKPROF 工具来查询SQL性能状态

SQL trace 工具收集正在执行的SQL的性能状态数据并记录到一个跟踪文件中. 这个跟踪文件提供了许多有用的信息,例如解析次数.执行次数,CPU使用时间等.这些数据将可以用来优化你的系统.

设置SQL TRACE在会话级别: 有效

ALTER SESSION SET SQL\_TRACE TRUE

设置SQL TRACE 在整个数据库有效仿, 你必须将SQL\_TRACE参数在init.ora中设为TRUE, USER\_DUMP\_DEST参数说明了生成跟踪文件的目录

(译者按: 这一节中,作者并没有提到TKPROF的用法, 对SQL TRACE的用法也不够准确, 设置SQL TRACE首先要在init.ora中设定TIMED\_STATISTICS, 这样才能得到那些重要的时间状态. 生成的trace文件是不可读的,所以要用TKPROF工具对其进行转换,TKPROF有许多执行参数. 大家可以参考ORACLE手册来了解具体的配置. )

* 1. 用EXPLAIN PLAN 分析SQL语句

EXPLAIN PLAN 是一个很好的分析SQL语句的工具,它甚至可以在不执行SQL的情况下分析语句. 通过分析,我们就可以知道ORACLE是怎么样连接表,使用什么方式扫描表(索引扫描或全表扫描)以及使用到的索引名称.

你需要按照从里到外,从上到下的次序解读分析的结果. EXPLAIN PLAN分析的结果是用缩进的格式排列的, 最内部的操作将被最先解读, 如果两个操作处于同一层中,带有最小操作号的将被首先执行.

NESTED LOOP是少数不按照上述规则处理的操作, 正确的执行路径是检查对NESTED LOOP提供数据的操作,其中操作号最小的将被最先处理.

译者按:

通过实践, 感到还是用SQLPLUS中的SET TRACE 功能比较方便.

举例:

SQL> list

1 SELECT \*

2 FROM dept, emp

3\* WHERE emp.deptno = dept.deptno

SQL> set autotrace traceonly /\*traceonly 可以不显示执行结果\*/

SQL> /

14 rows selected.

Execution Plan

----------------------------------------------------------

0 SELECT STATEMENT Optimizer=CHOOSE

1 0 NESTED LOOPS

2 1 TABLE ACCESS (FULL) OF 'EMP'

3 1 TABLE ACCESS (BY INDEX ROWID) OF 'DEPT'

4 3 INDEX (UNIQUE SCAN) OF 'PK\_DEPT' (UNIQUE)

Statistics

----------------------------------------------------------

0 recursive calls

2 db block gets

30 consistent gets

0 physical reads

0 redo size

2598 bytes sent via SQL\*Net to client

503 bytes received via SQL\*Net from client

2 SQL\*Net roundtrips to/from client

0 sorts (memory)

0 sorts (disk)

14 rows processed

通过以上分析,可以得出实际的执行步骤是:

1. TABLE ACCESS (FULL) OF 'EMP'

2. INDEX (UNIQUE SCAN) OF 'PK\_DEPT' (UNIQUE)

3. TABLE ACCESS (BY INDEX ROWID) OF 'DEPT'

4. NESTED LOOPS (JOINING 1 AND 3)

注: 目前许多第三方的工具如TOAD和ORACLE本身提供的工具如OMS的SQL Analyze都提供了极其方便的EXPLAIN PLAN工具.也许喜欢图形化界面的朋友们可以选用它们.

* 1. 用索引提高效率

索引是表的一个概念部分,用来提高检索数据的效率. 实际上,ORACLE使用了一个复杂的自平衡B-tree结构. 通常,通过索引查询数据比全表扫描要快. 当ORACLE找出执行查询和Update语句的最佳路径时, ORACLE优化器将使用索引. 同样在联结多个表时使用索引也可以提高效率. 另一个使用索引的好处是,它提供了主键(primary key)的唯一性验证.

除了那些LONG或LONG RAW数据类型, 你可以索引几乎所有的列. 通常, 在大型表中使用索引特别有效. 当然,你也会发现, 在扫描小表时,使用索引同样能提高效率.

虽然使用索引能得到查询效率的提高,但是我们也必须注意到它的代价. 索引需要空间来存储,也需要定期维护, 每当有记录在表中增减或索引列被修改时, 索引本身也会被修改. 这意味着每条记录的INSERT , DELETE , UPDATE将为此多付出4 , 5 次的磁盘I/O . 因为索引需要额外的存储空间和处理,那些不必要的索引反而会使查询反应时间变慢.

注:

定期的重构索引是有必要的.

ALTER INDEX <INDEXNAME> REBUILD <TABLESPACENAME>

* 1. 索引的操作

ORACLE对索引有两种访问模式.

* 索引唯一扫描 ( INDEX UNIQUE SCAN)

大多数情况下, 优化器通过WHERE子句访问INDEX.

例如:

表LODGING有两个索引 : 建立在LODGING列上的唯一性索引LODGING\_PK和建立在MANAGER列上的非唯一性索引LODGING$MANAGER.

SELECT \*

FROM LODGING

WHERE LODGING = ‘ROSE HILL';

在内部 , 上述SQL将被分成两步执行, 首先 , LODGING\_PK 索引将通过索引唯一扫描的方式被访问 , 获得相对应的ROWID, 通过ROWID访问表的方式 执行下一步检索.

如果被检索返回的列包括在INDEX列中,ORACLE将不执行第二步的处理(通过ROWID访问表). 因为检索数据保存在索引中, 单单访问索引就可以完全满足查询结果.

下面SQL只需要INDEX UNIQUE SCAN 操作.

SELECT LODGING

FROM LODGING

WHERE LODGING = ‘ROSE HILL';

* 索引范围查询(INDEX RANGE SCAN)

适用于两种情况:

1. 基于一个范围的检索

2. 基于非唯一性索引的检索

例1:

SELECT LODGING

FROM LODGING

WHERE LODGING LIKE ‘M%';

WHERE子句条件包括一系列值, ORACLE将通过索引范围查询的方式查询LODGING\_PK . 由于索引范围查询将返回一组值, 它的效率就要比索引唯一扫描

例2:

SELECT LODGING

FROM LODGING

WHERE MANAGER = ‘BILL GATES';

这个SQL的执行分两步, LODGING$MANAGER的索引范围查询(得到所有符合条件记录的ROWID) 和下一步同过ROWID访问表得到LODGING列的值. 由于LODGING$MANAGER是一个非唯一性的索引,数据库不能对它执行索引唯一扫描.

由于SQL返回LODGING列,而它并不存在于LODGING$MANAGER索引中, 所以在索引范围查询后会执行一个通过ROWID访问表的操作.

WHERE子句中, 如果索引列所对应的值的第一个字符由通配符(WILDCARD)开始, 索引将不被采用.

SELECT LODGING

FROM LODGING

WHERE MANAGER LIKE ‘％HANMAN';

在这种情况下，ORACLE将使用全表扫描.

* 1. 基础表的选择

基础表(Driving Table)是指被最先访问的表(通常以全表扫描的方式被访问). 根据优化器的不同, SQL语句中基础表的选择是不一样的.

如果你使用的是CBO (COST BASED OPTIMIZER),优化器会检查SQL语句中的每个表的物理大小,索引的状态,然后选用花费最低的执行路径.

如果你用RBO (RULE BASED OPTIMIZER) , 并且所有的连接条件都有索引对应, 在这种情况下, 基础表就是FROM 子句中列在最后的那个表.

举例:

SELECT A.NAME , B.MANAGER

FROM　WORKER A,

LODGING B

WHERE　A.LODGING = B.LODING;

由于LODGING表的LODING列上有一个索引, 而且WORKER表中没有相比较的索引, WORKER表将被作为查询中的基础表.

* 1. 多个平等的索引

当SQL语句的执行路径可以使用分布在多个表上的多个索引时, ORACLE会同时使用多个索引并在运行时对它们的记录进行合并, 检索出仅对全部索引有效的记录.

在ORACLE选择执行路径时,唯一性索引的等级高于非唯一性索引. 然而这个规则只有

当WHERE子句中索引列和常量比较才有效.如果索引列和其他表的索引类相比较. 这种子句在优化器中的等级是非常低的.

如果不同表中两个想同等级的索引将被引用, FROM子句中表的顺序将决定哪个会被率先使用. FROM子句中最后的表的索引将有最高的优先级.

如果相同表中两个想同等级的索引将被引用, WHERE子句中最先被引用的索引将有最高的优先级.

举例:

DEPTNO上有一个非唯一性索引,EMP\_CAT也有一个非唯一性索引.

SELECT ENAME,

FROM EMP

WHERE DEPT\_NO = 20

AND EMP\_CAT = ‘A';

这里,DEPTNO索引将被最先检索,然后同EMP\_CAT索引检索出的记录进行合并. 执行路径如下:

TABLE ACCESS BY ROWID ON EMP

AND-EQUAL

INDEX RANGE SCAN ON DEPT\_IDX

INDEX RANGE SCAN ON CAT\_IDX

* 1. 等式比较和范围比较

当WHERE子句中有索引列, ORACLE不能合并它们,ORACLE将用范围比较.

举例:

DEPTNO上有一个非唯一性索引,EMP\_CAT也有一个非唯一性索引.

SELECT ENAME

FROM EMP

WHERE DEPTNO > 20

AND EMP\_CAT = ‘A';

这里只有EMP\_CAT索引被用到,然后所有的记录将逐条与DEPTNO条件进行比较. 执行路径如下:

TABLE ACCESS BY ROWID ON EMP

INDEX RANGE SCAN ON CAT\_IDX

* 1. 不明确的索引等级

当ORACLE无法判断索引的等级高低差别,优化器将只使用一个索引,它就是在WHERE子句中被列在最前面的.

举例:

DEPTNO上有一个非唯一性索引,EMP\_CAT也有一个非唯一性索引.

SELECT ENAME

FROM EMP

WHERE DEPTNO > 20

AND EMP\_CAT > ‘A';

这里, ORACLE只用到了DEPT\_NO索引. 执行路径如下:

TABLE ACCESS BY ROWID ON EMP

INDEX RANGE SCAN ON DEPT\_IDX

注:

我们来试一下以下这种情况:

SQL> select index\_name, uniqueness from user\_indexes where table\_name = 'EMP';

INDEX\_NAME UNIQUENES

------------------------------ ---------

EMPNO UNIQUE

EMPTYPE NONUNIQUE

SQL> select \* from emp where empno >= 2 and emp\_type = 'A' ;

no rows selected

Execution Plan

----------------------------------------------------------

0 SELECT STATEMENT Optimizer=CHOOSE

1 0 TABLE ACCESS (BY INDEX ROWID) OF 'EMP'

2 1 INDEX (RANGE SCAN) OF 'EMPTYPE' (NON-UNIQUE)

虽然EMPNO是唯一性索引,但是由于它所做的是范围比较, 等级要比非唯一性索引的等式比较低!

* 1. 强制索引失效

如果两个或以上索引具有相同的等级,你可以强制命令ORACLE优化器使用其中的一个(通过它,检索出的记录数量少) .

举例:

SELECT ENAME

FROM EMP

WHERE EMPNO = 7935

AND DEPTNO + 0 = 10 /\*DEPTNO上的索引将失效\*/

AND EMP\_TYPE || ‘' = ‘A' /\*EMP\_TYPE上的索引将失效\*/

这是一种相当直接的提高查询效率的办法. 但是你必须谨慎考虑这种策略,一般来说,只有在你希望单独优化几个SQL时才能采用它.

这里有一个例子关于何时采用这种策略,

假设在EMP表的EMP\_TYPE列上有一个非唯一性的索引而EMP\_CLASS上没有索引.

SELECT ENAME

FROM EMP

WHERE EMP\_TYPE = ‘A'

AND EMP\_CLASS = ‘X';

优化器会注意到EMP\_TYPE上的索引并使用它. 这是目前唯一的选择. 如果,一段时间以后, 另一个非唯一性建立在EMP\_CLASS上,优化器必须对两个索引进行选择,在通常情况下,优化器将使用两个索引并在他们的结果集合上执行排序及合并. 然而,如果其中一个索引（EMP\_TYPE）接近于唯一性而另一个索引（EMP\_CLASS）上有几千个重复的值. 排序及合并就会成为一种不必要的负担. 在这种情况下,你希望使优化器屏蔽掉EMP\_CLASS索引.

用下面的方案就可以解决问题.

SELECT ENAME

FROM EMP

WHERE EMP\_TYPE = ‘A'

AND EMP\_CLASS||'' = ‘X';

* 1. 避免在索引列上使用计算

WHERE子句中，如果索引列是函数的一部分．优化器将不使用索引而使用全表扫描．

举例:

* 低效：

SELECT …

FROM DEPT

WHERE SAL \* 12 > 25000;

* 高效:

SELECT …

FROM DEPT

WHERE SAL > 25000/12;

注:

这是一个非常实用的规则，请务必牢记

* 1. 自动选择索引

如果表中有两个以上（包括两个）索引，其中有一个唯一性索引，而其他是非唯一性．

在这种情况下，ORACLE将使用唯一性索引而完全忽略非唯一性索引．

举例:

SELECT ENAME

FROM EMP

WHERE EMPNO = 2326

AND DEPTNO = 20 ;

这里，只有EMPNO上的索引是唯一性的，所以EMPNO索引将用来检索记录．

TABLE ACCESS BY ROWID ON EMP

INDEX UNIQUE SCAN ON EMP\_NO\_IDX

* 1. 避免在索引列上使用NOT

通常，我们要避免在索引列上使用NOT, NOT会产生在和在索引列上使用函数相同的

影响. 当ORACLE”遇到”NOT,他就会停止使用索引转而执行全表扫描.

举例:

* 低效: (这里,不使用索引)

SELECT …

FROM DEPT

WHERE NOT DEPT\_CODE = 0;

* 高效: (这里,使用了索引)

SELECT …

FROM DEPT

WHERE DEPT\_CODE > 0;

需要注意的是,在某些时候, ORACLE优化器会自动将NOT转化成相对应的关系操作符.

NOT > to <=

NOT >= to <

NOT < to >=

NOT <= to >

注:

我做了一些测试:

SQL> select \* from emp where NOT empno > 1;

no rows selected

Execution Plan

----------------------------------------------------------

0 SELECT STATEMENT Optimizer=CHOOSE

1 0 TABLE ACCESS (BY INDEX ROWID) OF 'EMP'

2 1 INDEX (RANGE SCAN) OF 'EMPNO' (UNIQUE)

SQL> select \* from emp where empno <= 1;

no rows selected

Execution Plan

----------------------------------------------------------

0 SELECT STATEMENT Optimizer=CHOOSE

1 0 TABLE ACCESS (BY INDEX ROWID) OF 'EMP'

2 1 INDEX (RANGE SCAN) OF 'EMPNO' (UNIQUE)

两者的效率完全一样，也许这符合作者关于” 在某些时候, ORACLE优化器会自动将NOT转化成相对应的关系操作符” 的观点．

* 1. 用>=替代>

如果DEPTNO上有一个索引,

* 高效:

SELECT \*

FROM EMP

WHERE DEPTNO >=4

* 低效:

SELECT \*

FROM EMP

WHERE DEPTNO >3

两者的区别在于, 前者DBMS将直接跳到第一个DEPT等于4的记录而后者将首先定位到DEPTNO=3的记录并且向前扫描到第一个DEPT大于3的记录.

* 1. 用UNION替换OR (适用于索引列)

通常情况下, 用UNION替换WHERE子句中的OR将会起到较好的效果. 对索引列使用OR将造成全表扫描. 注意, 以上规则只针对多个索引列有效. 如果有column没有被索引, 查询效率可能会因为你没有选择OR而降低.

在下面的例子中, LOC\_ID 和REGION上都建有索引.

* 高效:

SELECT LOC\_ID , LOC\_DESC , REGION

FROM LOCATION

WHERE LOC\_ID = 10

UNION

SELECT LOC\_ID , LOC\_DESC , REGION

FROM LOCATION

WHERE REGION = “MELBOURNE”

* 低效:

SELECT LOC\_ID , LOC\_DESC , REGION

FROM LOCATION

WHERE LOC\_ID = 10 OR REGION = “MELBOURNE”

如果你坚持要用OR, 那就需要返回记录最少的索引列写在最前面.

注意:

WHERE KEY1 = 10 (返回最少记录)

OR KEY2 = 20 (返回最多记录)

ORACLE 内部将以上转换为

WHERE KEY1 = 10 AND

((NOT KEY1 = 10) AND KEY2 = 20)

注:

下面的测试数据仅供参考: (a = 1003 返回一条记录 , b = 1 返回1003条记录)

SQL> select \* from unionvsor /\*1st test\*/

2 where a = 1003 or b = 1;

1003 rows selected.

Execution Plan

----------------------------------------------------------

0 SELECT STATEMENT Optimizer=CHOOSE

1 0 CONCATENATION

2 1 TABLE ACCESS (BY INDEX ROWID) OF 'UNIONVSOR'

3 2 INDEX (RANGE SCAN) OF 'UB' (NON-UNIQUE)

4 1 TABLE ACCESS (BY INDEX ROWID) OF 'UNIONVSOR'

5 4 INDEX (RANGE SCAN) OF 'UA' (NON-UNIQUE)

Statistics

----------------------------------------------------------

0 recursive calls

0 db block gets

144 consistent gets

0 physical reads

0 redo size

63749 bytes sent via SQL\*Net to client

7751 bytes received via SQL\*Net from client

68 SQL\*Net roundtrips to/from client

0 sorts (memory)

0 sorts (disk)

1003 rows processed

SQL> select \* from unionvsor /\*2nd test\*/

2 where b = 1 or a = 1003 ;

1003 rows selected.

Execution Plan

----------------------------------------------------------

0 SELECT STATEMENT Optimizer=CHOOSE

1 0 CONCATENATION

2 1 TABLE ACCESS (BY INDEX ROWID) OF 'UNIONVSOR'

3 2 INDEX (RANGE SCAN) OF 'UA' (NON-UNIQUE)

4 1 TABLE ACCESS (BY INDEX ROWID) OF 'UNIONVSOR'

5 4 INDEX (RANGE SCAN) OF 'UB' (NON-UNIQUE)

Statistics

----------------------------------------------------------

0 recursive calls

0 db block gets

143 consistent gets

0 physical reads

0 redo size

63749 bytes sent via SQL\*Net to client

7751 bytes received via SQL\*Net from client

68 SQL\*Net roundtrips to/from client

0 sorts (memory)

0 sorts (disk)

1003 rows processed

SQL> select \* from unionvsor /\*3rd test\*/

2 where a = 1003

3 union

4 select \* from unionvsor

5 where b = 1;

1003 rows selected.

Execution Plan

----------------------------------------------------------

0 SELECT STATEMENT Optimizer=CHOOSE

1 0 SORT (UNIQUE)

2 1 UNION-ALL

3 2 TABLE ACCESS (BY INDEX ROWID) OF 'UNIONVSOR'

4 3 INDEX (RANGE SCAN) OF 'UA' (NON-UNIQUE)

5 2 TABLE ACCESS (BY INDEX ROWID) OF 'UNIONVSOR'

6 5 INDEX (RANGE SCAN) OF 'UB' (NON-UNIQUE)

Statistics

----------------------------------------------------------

0 recursive calls

0 db block gets

10 consistent gets

0 physical reads

0 redo size

63735 bytes sent via SQL\*Net to client

7751 bytes received via SQL\*Net from client

68 SQL\*Net roundtrips to/from client

1 sorts (memory)

0 sorts (disk)

1003 rows processed

用UNION的效果可以从consistent gets和 SQL\*NET的数据交换量的减少看出

* 1. 用IN来替换OR

下面的查询可以被更有效率的语句替换:

* 低效:

SELECT….

FROM LOCATION

WHERE LOC\_ID = 10

OR LOC\_ID = 20

OR LOC\_ID = 30

* 高效

SELECT…

FROM LOCATION

WHERE LOC\_IN IN (10,20,30);

注:

这是一条简单易记的规则，但是实际的执行效果还须检验，在ORACLE8i下，两者的执行路径似乎是相同的．

* 1. 避免在索引列上使用IS NULL和IS NOT NULL

避免在索引中使用任何可以为空的列，ORACLE将无法使用该索引 ．对于单列索引，如果列包含空值，索引中将不存在此记录. 对于复合索引，如果每个列都为空，索引中同样不存在此记录.　如果至少有一个列不为空，则记录存在于索引中．

举例:

如果唯一性索引建立在表的A列和B列上, 并且表中存在一条记录的A,B值为(123,null) , ORACLE将不接受下一条具有相同A,B值（123,null）的记录(插入).

然而如果 所有的索引列都为空，ORACLE将认为整个键值为空而空不等于空. 因此你可以插入1000 条具有相同键值的记录,当然它们都是空!

因为空值不存在于索引列中,所以WHERE子句中对索引列进行空值比较将使ORACLE停用该索引.

举例:

* 低效: (索引失效)

SELECT …

FROM DEPARTMENT

WHERE DEPT\_CODE IS NOT NULL;

* 高效: (索引有效)

SELECT …

FROM DEPARTMENT

WHERE DEPT\_CODE >=0;

* 1. 总是使用索引的第一个列

如果索引是建立在多个列上, 只有在它的第一个列(leading column)被where子句引用时,优化器才会选择使用该索引.

注:

这也是一条简单而重要的规则. 见以下实例.

SQL> create table multiindexusage ( inda number , indb number , descr varchar2(10));

Table created.

SQL> create index multindex on multiindexusage(inda,indb);

Index created.

SQL> set autotrace traceonly

SQL> select \* from multiindexusage where inda = 1;

Execution Plan

----------------------------------------------------------

0 SELECT STATEMENT Optimizer=CHOOSE

1 0 TABLE ACCESS (BY INDEX ROWID) OF 'MULTIINDEXUSAGE'

2 1 INDEX (RANGE SCAN) OF 'MULTINDEX' (NON-UNIQUE)

SQL> select \* from multiindexusage where indb = 1;

Execution Plan

----------------------------------------------------------

0 SELECT STATEMENT Optimizer=CHOOSE

1 0 TABLE ACCESS (FULL) OF 'MULTIINDEXUSAGE'

很明显, 当仅引用索引的第二个列时,优化器使用了全表扫描而忽略了索引

* 1. ORACLE内部操作

当执行查询时,ORACLE采用了内部的操作. 下表显示了几种重要的内部操作.

ORACLE Clause

内部操作

* ORDER BY

ORT ORDER BY

* UNION

NION-ALL

* MINUS

INUS

* INTERSECT

NTERSECT

* DISTINCT,MINUS,INTERSECT,UNION

ORT UNIQUE

* MIN,MAX,COUNT

ORT AGGREGATE

* GROUP BY

ORT GROUP BY

* ROWNUM

OUNT or COUNT STOPKEY

* Queries involving Joins

ORT JOIN,MERGE JOIN,NESTED LOOPS

* CONNECT BY

ONNECT BY

* 1. 用UNION-ALL 替换UNION ( 如果有可能的话)

当SQL语句需要UNION两个查询结果集合时,这两个结果集合会以UNION-ALL的方式被合并, 然后在输出最终结果前进行排序.

如果用UNION ALL替代UNION, 这样排序就不是必要了. 效率就会因此得到提高.

举例:

* 低效：

SELECT ACCT\_NUM, BALANCE\_AMT

FROM DEBIT\_TRANSACTIONS

WHERE TRAN\_DATE = '31-DEC-95'

UNION

SELECT ACCT\_NUM, BALANCE\_AMT

FROM DEBIT\_TRANSACTIONS

WHERE TRAN\_DATE = '31-DEC-95'

* 高效:

SELECT ACCT\_NUM, BALANCE\_AMT

FROM DEBIT\_TRANSACTIONS

WHERE TRAN\_DATE = '31-DEC-95'

UNION ALL

SELECT ACCT\_NUM, BALANCE\_AMT

FROM DEBIT\_TRANSACTIONS

WHERE TRAN\_DATE = '31-DEC-95'

注:

需要注意的是，UNION ALL 将重复输出两个结果集合中相同记录. 因此各位还是

要从业务需求分析使用UNION ALL的可行性.

UNION 将对结果集合排序,这个操作会使用到SORT\_AREA\_SIZE这块内存. 对于这

块内存的优化也是相当重要的. 下面的SQL可以用来查询排序的消耗量

Select substr(name,1,25) "Sort Area Name",

substr(value,1,15) "Value"

from v$sysstat

where name like 'sort%'

* 1. 使用提示(Hints)

对于表的访问,可以使用两种Hints.

FULL 和 ROWID

* FULL hint 告诉ORACLE使用全表扫描的方式访问指定表.

例如:

SELECT /\*+ FULL(EMP) \*/ \*

FROM EMP

WHERE EMPNO = 7893;

* ROWID hint 告诉ORACLE使用TABLE ACCESS BY ROWID的操作访问表.

通常, 你需要采用TABLE ACCESS BY ROWID的方式特别是当访问大表的时候, 使用这种方式, 你需要知道ROIWD的值或者使用索引.

如果一个大表没有被设定为缓存(CACHED)表而你希望它的数据在查询结束是仍然停留

在SGA中,你就可以使用CACHE hint 来告诉优化器把数据保留在SGA中. 通常CACHE hint 和 FULL hint 一起使用.

例如:

SELECT /\*+ FULL(WORKER) CACHE(WORKER)\*/ \*

FROM WORK;

索引hint 告诉ORACLE使用基于索引的扫描方式. 你不必说明具体的索引名称

例如:

SELECT /\*+ INDEX(LODGING) \*/ LODGING

FROM LODGING

WHERE MANAGER = ‘BILL GATES';

在不使用hint的情况下, 以上的查询应该也会使用索引,然而,如果该索引的重复值过多而你的优化器是CBO, 优化器就可能忽略索引. 在这种情况下, 你可以用INDEX hint强制ORACLE使用该索引.

ORACLE hints 还包括ALL\_ROWS, FIRST\_ROWS, RULE,USE\_NL, USE\_MERGE, USE\_HASH 等等.

注:

使用hint, 表示我们对ORACLE优化器缺省的执行路径不满意,需要手工修改.

这是一个很有技巧性的工作. 我建议只针对特定的,少数的SQL进行hint的优化.

对ORACLE的优化器还是要有信心(特别是CBO)

* 1. 用WHERE替代ORDER BY

ORDER BY 子句只在两种严格的条件下使用索引.

* ORDER BY中所有的列必须包含在相同的索引中并保持在索引中的排列顺序.
* ORDER BY中所有的列必须定义为非空.
* WHERE子句使用的索引和ORDER BY子句中所使用的索引不能并列.

例如:

表DEPT包含以下列:

DEPT\_CODE PK NOT NULL

DEPT\_DESC NOT NULL

DEPT\_TYPE NULL

非唯一性的索引(DEPT\_TYPE)

* 低效: (索引不被使用)

SELECT DEPT\_CODE

FROM DEPT

ORDER BY DEPT\_TYPE

EXPLAIN PLAN:

SORT ORDER BY

TABLE ACCESS FULL

* 高效: (使用索引)

SELECT DEPT\_CODE

FROM DEPT

WHERE DEPT\_TYPE > 0

EXPLAIN PLAN:

TABLE ACCESS BY ROWID ON EMP

INDEX RANGE SCAN ON DEPT\_IDX

注:

ORDER BY 也能使用索引! 这的确是个容易被忽视的知识点. 我们来验证一下:

SQL> select \* from emp order by empno;

Execution Plan

----------------------------------------------------------

0 SELECT STATEMENT Optimizer=CHOOSE

1 0 TABLE ACCESS (BY INDEX ROWID) OF 'EMP'

2 1 INDEX (FULL SCAN) OF 'EMPNO' (UNIQUE)

* 1. 避免改变索引列的类型

当比较不同数据类型的数据时, ORACLE自动对列进行简单的类型转换.

假设 EMPNO是一个数值类型的索引列.

SELECT …

FROM EMP

WHERE EMPNO = ‘123'

实际上,经过ORACLE类型转换, 语句转化为:

SELECT …

FROM EMP

WHERE EMPNO = TO\_NUMBER(‘123')

幸运的是,类型转换没有发生在索引列上,索引的用途没有被改变.

现在,假设EMP\_TYPE是一个字符类型的索引列.

SELECT …

FROM EMP

WHERE EMP\_TYPE = 123

这个语句被ORACLE转换为:

SELECT …

FROM EMP

WHERE TO\_NUMBER(EMP\_TYPE)=123

因为内部发生的类型转换, 这个索引将不会被用到!

注:

为了避免ORACLE对你的SQL进行隐式的类型转换, 最好把类型转换用显式表现出来. 注意当字符和数值比较时, ORACLE会优先转换数值类型到字符类型.

* 1. 需要当心的WHERE子句

某些SELECT 语句中的WHERE子句不使用索引. 这里有一些例子.

在下面的例子里, ‘!=' 将不使用索引. 记住, 索引只能告诉你什么存在于表中, 而不能告诉你什么不存在于表中.

* 不使用索引:

SELECT ACCOUNT\_NAME

FROM TRANSACTION

WHERE AMOUNT !=0;

* 使用索引:

SELECT ACCOUNT\_NAME

FROM TRANSACTION

WHERE AMOUNT >0;

下面的例子中, ‘||'是字符连接函数. 就象其他函数那样, 停用了索引.

* 不使用索引:

SELECT ACCOUNT\_NAME,AMOUNT

FROM TRANSACTION

WHERE ACCOUNT\_NAME||ACCOUNT\_TYPE='AMEXA';

* 使用索引:

SELECT ACCOUNT\_NAME,AMOUNT

FROM TRANSACTION

WHERE ACCOUNT\_NAME = ‘AMEX'

AND ACCOUNT\_TYPE=' A';

下面的例子中, ‘+'是数学函数. 就象其他数学函数那样, 停用了索引.

* 不使用索引:

SELECT ACCOUNT\_NAME, AMOUNT

FROM TRANSACTION

WHERE AMOUNT + 3000 >5000;

* 使用索引:

SELECT ACCOUNT\_NAME, AMOUNT

FROM TRANSACTION

WHERE AMOUNT > 2000 ;

下面的例子中,相同的索引列不能互相比较,这将会启用全表扫描.

* 不使用索引:

SELECT ACCOUNT\_NAME, AMOUNT

FROM TRANSACTION

WHERE ACCOUNT\_NAME = NVL(:ACC\_NAME,ACCOUNT\_NAME);

* 使用索引:

SELECT ACCOUNT\_NAME, AMOUNT

FROM TRANSACTION

WHERE ACCOUNT\_NAME LIKE NVL(:ACC\_NAME,'%');

注:

如果一定要对使用函数的列启用索引, ORACLE新的功能: 基于函数的索引(Function-Based Index) 也许是一个较好的方案.

CREATE INDEX EMP\_I ON EMP (UPPER(ename)); /\*建立基于函数的索引\*/

SELECT \* FROM emp WHERE UPPER(ename) = ‘BLACKSNAIL'; /\*将使用索引\*/

* 1. 连接多个扫描

如果你对一个列和一组有限的值进行比较, 优化器可能执行多次扫描并对结果进行合并连接.

举例:

SELECT \*

FROM LODGING

WHERE MANAGER IN (‘BILL GATES','KEN MULLER');

优化器可能将它转换成以下形式

SELECT \*

FROM LODGING

WHERE MANAGER = ‘BILL GATES'

OR MANAGER = 'KEN MULLER';

当选择执行路径时, 优化器可能对每个条件采用LODGING$MANAGER上的索引范围扫描. 返回的ROWID用来访问LODGING表的记录 (通过TABLE ACCESS BY ROWID 的方式). 最后两组记录以连接(CONCATENATION)的形式被组合成一个单一的集合.

Explain Plan :

SELECT STATEMENT Optimizer=CHOOSE

CONCATENATION

TABLE ACCESS (BY INDEX ROWID) OF LODGING

INDEX (RANGE SCAN ) OF LODGING$MANAGER (NON-UNIQUE)

TABLE ACCESS (BY INDEX ROWID) OF LODGING

INDEX (RANGE SCAN ) OF LODGING$MANAGER (NON-UNIQUE)

注:

本节和第37节似乎有矛盾之处.

* 1. CBO下使用更具选择性的索引

基于成本的优化器(CBO, Cost-Based Optimizer)对索引的选择性进行判断来决定索引的使用是否能提高效率.

如果索引有很高的选择性, 那就是说对于每个不重复的索引键值,只对应数量很少的记录.

比如, 表中共有100条记录而其中有80个不重复的索引键值. 这个索引的选择性就是80/100 = 0.8 . 选择性越高, 通过索引键值检索出的记录就越少.

如果索引的选择性很低, 检索数据就需要大量的索引范围查询操作和ROWID 访问表的

操作. 也许会比全表扫描的效率更低.

注

下列经验请参阅:

a. 如果检索数据量超过30%的表中记录数.使用索引将没有显著的效率提高.

b. 在特定情况下, 使用索引也许会比全表扫描慢, 但这是同一个数量级上的区别. 而通常情况下,使用索引比全表扫描要块几倍乃至几千倍!

* 1. 避免使用耗费资源的操作

带有DISTINCT,UNION,MINUS,INTERSECT,ORDER BY的SQL语句会启动SQL引擎

执行耗费资源的排序(SORT)功能. DISTINCT需要一次排序操作, 而其他的至少需要执行两次排序.

例如,一个UNION查询,其中每个查询都带有GROUP BY子句, GROUP BY会触发嵌入排序(NESTED SORT) ; 这样, 每个查询需要执行一次排序, 然后在执行UNION时, 又一个唯一排序(SORT UNIQUE)操作被执行而且它只能在前面的嵌入排序结束后才能开始执行. 嵌入的排序的深度会大大影响查询的效率.

通常, 带有UNION, MINUS , INTERSECT的SQL语句都可以用其他方式重写.

注

如果你的数据库的SORT\_AREA\_SIZE调配得好, 使用UNION , MINUS, INTERSECT也是可以考虑的, 毕竟它们的可读性很强

* 1. 优化GROUP BY

提高GROUP BY 语句的效率, 可以通过将不需要的记录在GROUP BY 之前过滤掉.下面两个查询返回相同结果但第二个明显就快了许多.

* 低效:

SELECT JOB , AVG(SAL)

FROM EMP

GROUP JOB

HAVING JOB = ‘PRESIDENT'

OR JOB = ‘MANAGER'

* 高效:

SELECT JOB , AVG(SAL)

FROM EMP

WHERE JOB = ‘PRESIDENT'

OR JOB = ‘MANAGER'

GROUP JOB

注:

本节和8.14节相同. 可略过.

* 1. 使用日期

当使用日期是,需要注意如果有超过5位小数加到日期上, 这个日期会进到下一天!

例如:

* 1.

SELECT TO\_DATE(‘01-JAN-93'+.99999)

FROM DUAL;

Returns:

'01-JAN-93 23:59:59'

* 2.

SELECT TO\_DATE(‘01-JAN-93'+.999999)

FROM DUAL;

Returns:

'02-JAN-93 00:00:00'

* 1. 使用显式的游标(CURSORs)

使用隐式的游标,将会执行两次操作. 第一次检索记录, 第二次检查TOO MANY ROWS 这个exception . 而显式游标不执行第二次操作.

* 1. 优化EXPORT和IMPORT

使用较大的BUFFER(比如10MB , 10,240,000)可以提高EXPORT和IMPORT的速度.

ORACLE将尽可能地获取你所指定的内存大小,即使在内存不满足,也不会报错.这个值至少要和表中最大的列相当,否则列值会被截断.

注:

可以肯定的是, 增加BUFFER会大大提高EXPORT , IMPORT的效率. (曾经碰到过一个CASE, 增加BUFFER后,IMPORT/EXPORT快了10倍!)

“这个值至少要和表中最大的列相当,否则列值会被截断. “ 其中最大的列是指最大的记录大小.

* 1. 分离表和索引

总是将你的表和索引建立在不同的表空间内(TABLESPACES). 决不要将不属于ORACLE内部系统的对象存放到SYSTEM表空间里. 同时,确保数据表空间和索引表空间置于不同的硬盘上.

注：

本附录主要参考了**<<ORACLE SQL性能优化系列>>**