

## 第17章 应用调整

本章要点：

动机

理解优化程序

SQL TRACE与tkprof

理解EXPLAIN PLAN

鉴别典型问题

重写查询

介绍ORACLE 8的新的索引特性

### 17.1 动机

调整应用实际上相当于调整你的应用中的 SQL 语句。假如你的应用有一个图形用户接口 (GUI) 前端, 调整应用仍然相当于调整 SQL 语句。换句话说, 所有 GUI 层的操作可以被映射到具有不同复杂度的一条或多条 SQL 语句中 (SELECT、INSERT、UPDATE、DELETE 等)。总之, 假如可能的话, 你的投资返回值 (ROI) 策略告诉你必须首先调整应用, 可能的话再调整数据库的所有方面。

主观估计表明应用决定着大约 80% 的总系统 (应用加数据库) 性能, 而仅为数据库留下 20% 的系统性能。我同意此观点。然而, 我不太同意该观点中的数据。在我的经历中, 我见过一些系统中数据库影响大部分性能。

对于数据库管理员来说, 这个观点真正的意思是系统性能是相对的、与状况相关的、应用特定的。这听起来很熟悉, 尤其是在关于应用类型的讨论中。然而, 有一条绝对规则永远有效, 我全力提倡数据库管理员应关注第一条调整规则:

“应用应当首先被调整。”

这条规则有如下两条理由: 1) 目前存在的大多数数据库系统把它们的大部分性能归于它们的应用。2) 即使这条规则不适用于你的系统, 作为一个数据库管理员, 你也必须对系统的应用类型进行分类。这意味着你必须检查应用, 具体地说, 检查事务的类型——访问代码和 SQL 代码。从数据库访问的观点来说, 即使你的应用代码尽可能高效地编写, 你仍然必须理解这个程序如何对你的数据库产生影响。

你如何完成检查应用的任务? 假如从代码的行数或分离的程序单元 (模块) 的数量上来说应用相对较小——那么人工检查每一件事情是可能的。相反, 假如应用很大, 那么完全人工检查是不可能的。规则 3——优先次序 (参见第 16 章) 给了你方向。你可以把精力集中在那些消耗了大部分资源和时间的模块或代码段上。

首先, 你试图维护 20% 的应用代码, 这些代码决定了 80% 的系统性能 (假如你有时间和人员, 那么你能够在剩余的代码上工作以榨取余下的性能潜力)。你实际如何做呢? 17.3 节讨论了你所需要的 Oracle 工具。但是首先, 下一节给你了使用那些工具的必要的背景知识——理解

Oracle优化程序以及它是如何工作的。

## 17.2 理解优化程序

优化程序 (optimizer) 是负责优化的一段软件以及一部分 RDBMS, 或者是对一条给定的用于访问数据的 SQL 语句的最有效表达方式。要做到这一点, 优化程序选择一系列访问路径, 该路径为 Oracle 访问数据提供了最快的路线, 然后建立基于那些访问路径的执行计划。访问路径是一条指向数据的物理路线。执行计划是沿着已选定的访问路径的一系列 Oracle 可执行步骤。

Oracle 优化程序中的技术, 如同许多其他的 RDBMS 厂商一样, 在过去几年中已经取得了显著的进展。具有讽刺意义的是, 优化程序仍然不总是能提供访问数据的最佳方法。在较老的版本中, 程序员必须留意如何编写 SQL 语句。SQL 语句里的排序能够强烈影响 SQL 语句的执行, 所以程序员不得不执行许多人工优化。同样, 数据库管理员必须对这种情况有所了解。优化程序越好, 用户、程序员和数据库管理员的优化责任就越小。即使在当前的 RDBMS 优化程序领域, 程序员仍然需要掌握优化技巧并在可能时运用它们。然而, 程序员不必象以前那样对优化程序很内行。数据库管理员仍然应当知道优化程序技术、优化规则以及正确的优化程序用法。

优化程序有两种风格:

基于规则——一个基于规则的优化程序选择的访问路径是基于不变的、RDBMS 厂商特定的排列次序, 其中这些访问路径是到数据的最快方法。尽管这个排列是厂商特定的, 然而在不同厂商的优化程序的输出之间比较和映射类似的访问路径相对比较容易。

基于成本——一个基于成本的优化程序选择的访问路径是基于一些内部存储的数据分布统计数字, 这些统计数字通常放在 RDBMS 数据字典中。一般, 数据库管理员必须周期性地运行一些 RDBMS 命令来维护这些统计数字。

Oracle 7.x 和较高级的优化程序可以在同一个优化程序中既提供基于规则的也提供基于成本的优化能力。Oracle 6.x 优化程序提供基于规则的优化, 但是它没有一个全自动的基于成本的优化。所有这些版本为程序员提供了一个称为提示 (hint) 的替换值。从优化程序不总是遵循该提示的意义上来说, 这不是一个真正的替换值。可以认为一个提示就是对优化程序的一个建议。假如不存在语法错误, 优化程序通常会采用它。一个提示被内置存放, 在 SQL 语句代码中有直接意义。清单 17-1 显示一条 SELECT 语句, 建议优化程序使用正在选择的表中的 INDEX。

清单 17-1 一个具有提示的查询的例子

```
SQL> SELECT /*+ INDEX */ EMPLOYEE_ID  
2> FROM EMPLOYEES  
3> WHERE EMPLOYEE_ID = 503748;
```

注意提示会立刻跟踪 SQL 命令 (SELECT)。提示的注释像任何常规的 SQL\*Plus 注释一样, 以 /\* 开始, 并且马上跟着一个加号 (+), 这样以 /\*+ 形成了一个提示注释的开始。通常以 \*/ 来结束提示注释。还有, 考虑一下, 假如存在不止一条索引的话, 特别是假如由于某些原因多条索引存在于 EMPLOYEE\_ID 上, 优化程序如何知道要使用哪一条索引。在以前的例子中, 仅仅规定了 /\*+ 索引 \*/ 并且假定只有一条索引 (主键) 存在于表中。因此, 对于这样的列表不

存在模糊。在存在模糊的情况下，你需要在提示中包含表和索引名以便给优化程序提供具体信息。提示在接下来一节中将详细讨论。

注意 优化程序仅在每条语句或语句块接受一个提示。优化程序忽略任何附加的提示。

此外，优化程序忽略任何拼写错误的提示。没有被报告的错误，你必须使用 EXPLAIN PLAN来决定是否采纳了你的提示，EXPLAIN PLAN将在17.4节中讨论。

### 17.2.1 分级访问路径

Oracle优化程序访问路径按照最快到最慢，顶端到底端的顺序排列，如表 17-1所示。

表17-1 Oracle优化程序访问路径，有序等级（来源：Oracle公司）

等 级	访问路径
1	依据ROWID的单行
2	依据簇连结的单行
3	依据具有唯一键或主键的哈西簇键的单行
4	依据唯一键或主键的单行
5	簇连接
6	哈西簇键
7	索引的簇键
8	组合键
9	单列索引
10	索引列上的限定范围查找
11	索引列上的没有限定范围查找
12	分类合并连结
13	索引列的MAX或MIN
14	索引列上的ORDER BY
15	全表扫描

当Oracle优化程序采用基于规则的策略时，它使用适合给定查询的最快的访问路径。例如，清单 17-2显示了和程序清单 17-1一样的查询，不过没有提示它已经用一个 >WHERE子句修改了。

清单17-2 一个没有提示的查询与一个没有范围限制的查找的例子

```
SQL> SELECT EMPLOYEE_ID
2> FROM EMPLOYEES
3> WHERE EMPLOYEE_ID > 500000;
```

假如Oracle正在使用基于规则的策略并且一个主键索引存在于EMPLOYEES表的EMPLOYEE\_ID上，那么Oracle将对清单 17-2的查询使用访问路径 #11（索引列上的没有范围限制的查找），因为WHERE子句是不受限制的。换句话说，当一条 WHERE子句有大于（>或>=）运算符并且没有相应的小于（<或<=）运算符时，该子句就是不受限制的。反之亦然。直到执行时才知道范围。

假如WHERE子句是受限制的（实际上，它有两个运算符），限定的范围在语法分析时才知道。要使一个不受限制的 WHERE子句变为受限制的，你可以使用直接量最大值（假如你知道它），或者根据你的业务规则至少使用该列的理论最大值。然而，除了你的初始的不受范围限制的查找外，你或许不想使用 MAX运算符——假如你查看等级排序，正在使用的 MAX是

#13, 它把一个不受限制的查找 ( #11 ) 提高到一个受限制的查找 ( #10 )。把MAX运算符用于初始的不受范围限制的查找会降低你的速度。但是使用实际的最大值将获得成功。理解这些在清单17-3中说明的原则:

提前使用一个已知的实际最大值 ( 直接量 )。这在清单17-3的第一条查询中进行了举例说明 ( 678453 )。

假如不知道实际值的话, 提前使用一个已知的理论上的业务最大值 ( 直接量 )。这在清单17-3的第二条查询中进行了举例说明 ( 999999 )。

不要使用最大SQL集合函数, 因为它仅会使事情变慢。这在清单17-3的第三条查询中进行了举例说明。

清单17-3 重写一个不受限制的查找以使它受限制

---

```
SQL> SELECT EMPLOYEE_ID
2> FROM EMPLOYEES
3> WHERE EMPLOYEE_ID > 500000 AND EMPLOYEE_ID <=678453;

SQL> SELECT EMPLOYEE_ID
2> FROM EMPLOYEES
3> WHERE EMPLOYEE_ID > 500000 AND EMPLOYEE_ID <=999999;

SQL> SELECT EMPLOYEE_ID
2> FROM EMPLOYEES
3> WHERE EMPLOYEE_ID > 500000 AND EMPLOYEE_ID <= MAX(EMPLOYEE_ID);
```

---

## 17.2.2 分析查询以提高效率

试图再配置或重写查询以提高它们效率的领域被称为查询重写。查询重写将在本章中的17.5节与17.6节中详细讨论。前面的例子很标准并且容易理解。例如, 要得到一列中一组数值的最大值 ( MAX ), 似乎要花费接近于一个整表扫描所用的时间——不管该列是否被索引。正如你看到的那样, 通过最大值访问路径的等级排序, 情况大致就是这样。然而, 还有不可理解的和更特殊的重写查询方法。

前面已研究了基于规则的优化, 现在来看一下基于成本的优先。先来看一下清单17-4和清单17-5中的两个查询。

清单17-4 同一张表上的两个相似查询的第一个

---

```
SQL> SELECT EMPLOYEE_ID
2> FROM EMPLOYEES
3> WHERE EMPLOYEE_TYPE='VICE PRESIDENT';
```

---

清单17-5 同一张表上的两个相似查询的第二个

---

```
SQL> SELECT EMPLOYEE_ID
2> FROM EMPLOYEES
3> WHERE EMPLOYEE_TYPE='PROGRAMMER';
```

---

现在, 假定你拥有一个相对较大的软件公司, 你有6个副手和总共大约6000个雇员, 这其中有大约2000个程序员。前面的两个查询解释了基于规则和基于成本的查询优化之间的差别以及为什么首选基于成本的优化。假设EMPLOYEE\_TYPE列上有一个非唯一索引, 那么基于

规则的优化为两个查询选择访问路径 #9 (单列索引)。另一方面,假如数据分布是程序员占总行数的 1/3 并且副手仅占总行数的 1/1000——那么为清单 17-4 中的查询选择使用非唯一索引,但是为清单 17-5 中的查询选择整表搜索 (最坏的事件访问路径 #15) 是明智的。这一决定来自对数据分布方面所具有的知识。

假如优化程序必须对一张表中所有行的有效部分进行访问,那么全表扫描实际上比索引搜索更有效。这是因为搜索行索引,然后再检索那个特殊行的过程至少每行需要两个读操作,有时更多——根据索引中独特数值的数量而不同。然而,全表扫描每行仅需要一个读操作。在有许多行时这个次数增加,和清单 17-5 中的查询一样,同与整个表的刚好读出相比,对一张表进行大量访问时索引更慢的原因相当明显。除了读操作的总数以外,全表扫描优于对表的大部分进行检索的索引的另一个主要原因是大部分必要的读操作是连续的或几乎连续的。显而易见,从一个索引中读取,之后从一张表中读取,然后回到该索引,之后回到该表,这样是不可能连续的,不论表和索引是否在同一张磁盘上或者在好的调整方式下展开。索引由于有类似清单 17-5 的查询和数据而使速度慢了下来。

在类似清单 17-4 的情况下,索引明显优于为查询和数据的全表扫描。这是因为你仅需要检索少量的行 (6) 并且你的读操作的总数仅是 12 ( $2 \times 6$ )。相比之下,一个全表扫描必须完成 6000 个读操作 (整个表),因为一个全表扫描不知道单个数据片存储在哪儿,所以它必须检查每一行。实际上,程序清单 17-4 中的查询和数据是一个需要索引的典型例子,特别是在查询经常被执行的情况下。

### 17.2.3 指定优化程序模式

下一个明显的问题是如何指定 Oracle 中的优化程序模式。迄今为止,你已经学习了有关基于规则和基于成本的优化。可以在实例、会话或语句层指定你希望的优化形式。如果要在实例层指定优化形式,把 init.ora 的参数 OPTIMIZER\_MODE 设置为下列值中的一个:

**CHOOSE**——当设置为该值时,如果统计数字可用的话 (已经由数据库管理员运行),优化程序选择基于成本的模式。否则,用基于规则的优化。

**RULE**——优化程序使用基于规则的方法。

**FIRST\_ROWS**——优化程序选择基于成本的方法 (假设统计数字可用的话) 来把反应时间减到最小。也就是说,使第一行尽快出现在屏幕上。假如你有一个高度交互式的、基于屏幕的应用,例如许多 OLTP 和较小的 DSS 系统,那么使用这种模式。

**ALL\_ROWS**——优化程序选择基于成本的模式 (假设统计数字可用的话) 来最小化吞吐量,也就是说,最小化每个时间单元内通过系统的总行数 (每秒钟的事务)。假如你有一个批处理或大型 DSS 系统的话,那么使用这种模式。

要在会话层指定优化形式,发出如下的 DLL 语句:

```
SQL> ALTER SESSION SET OPTIMIZER_GOAL=<value>;
```

该值是以前提及的优化程序模式的一种 (CHOOSE、RULE、FIRST\_ROWS、ALL\_ROWS)。结果仅适用于本会话,因此,必须在一个未来的会话中重新发出这个命令。

要在语句层指定优化形式,像较早前讨论过的那样使用提示。提示可以采用任何一种优化程序模式值 (CHOOSE、RULE、FIRST\_ROWS、ALL\_ROWS),或者是显示于表 17-2 和表 17-3 中的任何一个访问路径。

表17-2 提示的基本访问路径

访问路径	描 述
ROWID	使用用于检索的ROWID搜索
CLUSTER	使用簇键搜索
HASH	使用哈希索引搜索
INDEX	使用索引搜索
INDEX_ASC	使用索引搜索并按升序搜索
INDEX_DESC	使用索引搜索并按降序搜索
AND_EQUAL	使用多个索引并且合并它们的结果
ORDERED	使FROM子句中的表命令成为并运算命令
USE_NL	使用用于表连结的嵌套循环方法
USE_MERGE	使用用于表连结的分类合并方法
FULL	使用全表扫描

表17-3 提示的辅助访问路径（版本7.3和更新的版本）

访问路径	描 述
CACHE	通知Oracle把表作为一个缓冲表对待，在一个完全搜索之后将表的数据块保存在SGA中用于后面的快速访问
HASH_AJ	在一个非连结期间指定要使用的连结类型（Oracle 7.3和更新的版本）
MERGE_AJ	在一个非连结期间指定要使用的连结类型
NO_MERGE	通知Oracle不要将视图的SQL语法与使用连结的查询语法合并
NO_CACHE	把块标记为“最近最少使用的”以便它们可以不久后从SGA中移去
NONPARALLEL	禁用查询的并行操作
ROWID	使用TABLE ACCESS BY ROWID操作
STAR	当解决一个连结时使用一个组合键/开始查询执行路径
USE_CONTACT	迫使WHERE子句中的OR条件组合为UNION ALL
USE_HASH	使用哈希连结

还有，要使用任何一个基于成本的优化模式（FIRST\_ROWS、ALL\_ROWS），数据库管理员必须周期性地运行统计数字以及时反映实际的存储的数据分布。数据库管理员既可以获得一个（简单随机的）样本的所有统计数字也可以获得该样本的某些统计数字。要获得所有的统计数字意味着做一次全表扫描。要获得一个采样意味着要访问少于总行数的某些部分。数据库管理员发出如下的DLL语句以获得所有的统计数字：

```
SQL> ANALYZE TABLE <table_name> COMPUTE STATISTICS;
```

该语句对于相对小的表——少于一百万行来说比较精确。对于大表，数据库管理员或许希望选取一个样本。从统计学上讲，只要样品相对于总行数足够大，那么它就足够精确，不需要获得所有的统计数字。要获得一个样品，数据库管理员应发出如下的DDL语句：

```
SQL> ANALYZE TABLE <table_name> ESTIMATE STATISTICS;
```

不管实际表有多大，这条语句最大可以采样1064行（缺省情况下）。数据库管理员通过发出下面这条语句指定百分率：

```
SQL> ANALYZE TABLE <table_name> ESTIMATE STATISTICS SAMPLE 10 PERCENT;
```

该语句采样了表中总行数的10%，四舍五入到一个整数。例如，该语句从6000行的EMPLOYEES表中采样600行用于估算。数据库管理员也可以通过发出如下的语句指定实际的采样容量：

```
SQL> ANALYZE TABLE <table_name> ESTIMATE STATISTICS SAMPLE 2000 ROWS;
```

很明显，该语句精确地采样了指定表的 2000 行。

**警告** 假如你指定了一个超过 50% 的百分数（或者超过行数的 50% 的数值），那么 ANALYZE 命令便采用全表扫描计算而不是你所指定的百分数（或行数）。

对于 Oracle 的 7.3 版本和更高的版本，ANALYZE 命令有一条附加的 FOR 子句。利用这条子句，你可以命名具体的 FOR COLUMN 列。例如，假如自上一次 ANALYZE 运行以来只有少数列发生了变化或者你想要减少 ANALYZE 的运行时间，那么指定 FOR INDEXED COLUMNS ONLY，它可以不获得非索引列上的统计数字精确地执行要做的内容。

数据库管理员应定期地运行 ANALYZE。然而，对一个应用敏感的东西对另一个应用或许并不敏感。例如，每月从三个批处理系统进行加载的 DSS 仅需要分析（用计算机）那些已经装入的表，然后立即或在它们下一次使用前的某个合理时间跟踪装入值。相比之下，一个高活性的 OLTP 系统——例如一个处理飞行保留信息的系统——几乎需要所有的表，这些表可能每分钟都进行修改，在一个很短的时间间隔，例如一个小时或更短的时间间隔中被分析（用 ESTIMATE）。这些类型的系统或许被映像并且当某个产品拷贝被脱机分析时，其他的产品拷贝可以继续服务，反之亦然。一般而言，当你的表变化很大时，尽可能频繁地或按一定的间隔分析你的表以便 Oracle 优化程序能够为你的应用需求提供最好的服务。如果必要的话，可以交互式地执行分析，但是这些 ANALYZE 操作经常由 DML 事件预定或触发并由非交互性的脚本或存储过程来编写。

#### 17.2.4 理解优化术语

现在，查看一些数据库管理员可以理解并经常使用的术语。直方图（histogram）无非就是一种特殊类型的条形图。当执行 ANALYZE...COMPUTE 或 ANALYZE...ESTIMATE 时，存储该表的列数据分布。一个数据分布就是映射不同值的频率（数量或小计）或百分率的函数。在 Oracle 中，这些数据分布作为数字存储，但是假如你编写一个程序来显示这些数据分布的话，它们可以作为直方图显示。

选择性（selectivity）有时称为索引的选择性因子，是在一列按总值划分的数据值中总不同值的数量。选择性因子越高越好。除此之外，选择性越高，在那列上创建与使用索引的理由就越充分。通常，不同值越少，意味着选择性越小（这可以讲得通，因为该值是选择性公式的分子）。

通常一个存储性别类型（例如男或女）的列或者只有几种可能值的分类类型（例如婚姻状况，包括独身、已婚、离婚或寡居）的列不具有选择性并且不为索引的使用提供好的基础。一个例外是在这些类型的列中一个或多个不同值几乎不出现或经常被查询的情况下（在 WHERE 子句中频繁使用）。假设婚姻状况用作市长市民表的一列，那么该列数值中独居所占的比例相当小，小于 10%，而独居值经常被查询。因此，索引很适合于这类查询，尽管索引在其他值的查询上不会有太大帮助。此外，在这里位映射索引运行得会很好。决定何时使用索引以及使用何种类型的索引将在本章稍后的 17.5 节中讨论。

与必须从表中取数据相反，优化程序可以从一条可用的索引中读取和返回数据。优化程序优先级被称为索引覆盖（index coverage），尽管偶尔也会使性能变差，但通常会导致好的性能。假定在一张 EMPLOYEE 表的 LAST\_NAME、FIRST\_NAME 与 MIDDLE\_INITIAL 中有一个复合索引，你可以运行如下的查询以找到所有 LAST\_NAME 是 SMITH 的雇员：

```
SQL> SELECT LAST_NAME  
2> FROM EMPLOYEES  
3> WHERE LAST_NAME = 'SMITH';
```

在该例中，这个查询被认为是被索引覆盖的。换句话说，SELECT语句的SELECT子句中需要的所有列被存储（不仅在该表中而且）在该索引中。由于显而易见的原因，优化程序通常更喜欢单独使用索引，而不是同时使用索引和表。当一口井便能满足你的所有需要时，为什么要从两口井中取水？然而，索引覆盖有副作用，这将在17.5节中讨论。

最后，数据库管理员应当知道的关于查询调整的最重要的规则是百分之五规则。这条规则应用于Oracle 7.x中，但是在较老的版本中没有使用。假如从ANALYZE操作中得到可用的统计数字，那么当优化程序会发现期望要返回的行数大于5%时，它采用全表扫描。在Oracle 6.x和5.x中，该百分率分别是15%和20%。尽管实际的百分率不再应用于现代Oracle版本中，但该规则经常被称为20%规则。这是数据库管理员必须知道的重要信息，因为优化程序有时会做出错误的（不是最佳的）选择，根据你的应用的知识，你可能需要忽略它（例如，使用一个提示或重写查询）。

### 17.3 SQL TRACE与tkprof

SQL TRACE工具，连同tkprof格式化程序一起，提供了为单条SQL语句生成和检查性能统计数字的功能。你可以在实例或会话层启用SQL TRACE。然而，你必须在实例或会话层中都启用某些init.ora参数才能使用SQL TRACE。假如还没有设置这些参数，那么请设置如下参数：

```
TIMED_STATISTICS=TRUE  
USER_DUMP_DEST=<directory>  
MAX_DUMP_FILE_SIZE=<size in operating system blocks>
```

很明显，假如你希望SQL TRACE能够收集定时信息的话，那么你应当把TIMED\_STATISTICS设置为TRUE。如果你不想把跟踪文件存储在操作系统特定的缺省位置的话，那么请指定USER\_DUMP\_DEST。此外，MAX\_DUMP\_FILE\_SIZE应被设置得足够大以容纳所有生成的跟踪文件。要在实例层启用SQL TRACE（也就是说，对于所有的会话），设置如下init.ora参数：

```
SQL_TRACE=TRUE
```

所有的会话将产生跟踪文件。除了实例细节外，命名具有会话细节的跟踪文件能够辨别跟踪文件属于哪一个会话并保证在USER\_DUMP\_DEST中有足够的空间以容纳所有的文件。单独运行跟踪文件，可能的话顺序运行它，所以不会有混乱发生。要在会话层启用SQL TRACE，发出如下的DDL语句：

```
SQL> ALTER SESSION SET SQL_TRACE=TRUE;
```

这个SQL TRACE仅产生会话期的统计数字。现在，要能够读取由SQL TRACE创建的信息，你必须使用tkprof。在操作系统层（命令行）运行tkprof程序。执行这项工作的简化语法如下：

```
tkprof <tracefile> <outputfile>
```

一个UNIX系统中的例子如下所示：

```
hostname% tkprof ora_1776.trc ora_1776.out
```

在该例中，tkprof把跟踪文件（ora\_1776.trc）重新格式化到一个可读的、用户定义的输出文件（ora\_1776.out）中。一个类似于下述SQL语句的tkprof输出示例显示在清单17-6中：

```
SELECT * FROM EMPLOYEES WHERE DUTY >= 10;
```

清单17-6 一个已格式化的tkprof输出例子

Call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.17	0.35	0	0	0	0
Execute	1	0.00	0.01	0	0	0	0
Fetch	1	0.06	0.10	13	304	0	3679
Misses in library cache during parse: 1							
Misses in library cache during execute: 1							
Optimizer mode: CHOOSE							
Parsing user id: 8							

上述这些信息意味着什么以及它如何帮助你调整你的应用？正如大部分诊断工具一样，信息被给出，但是没有解释，至少没有自动解释。首先，检查你的输出文件中的柱状统计数字以及它们的含义，如表17-4中所讨论的。

表17-4 tkprof统计数字的描述

统计量	描述
count	语法分析、执行或取数据调用的数量
cpu	以秒计的实际CPU时间（假如语法分析的时间为0，则该语句处于库高速缓存中）
elapsed	以秒计的实际时钟（或背景）时间（总是大于或等于CPU时间）
disk	从数据文件中读取的Oracle数据块；也称为物理读取（physical reads）
query	Oracle缓存读取，用于从回滚缓冲区高速缓存中一致性读取，存储于数据库缓冲区高速缓存中；也称为一致性获取（consistent gets）
current	Oracle缓冲区在当前模式下从数据库缓冲区高速缓存中读取；也称为数据库块获取（db block gets）
rows	由主驱动（或外部）语句访问的行数；对于SELECT语句针对取数据，否则针对执行（DML）

注意到逻辑读取的数量等于 query+current。你如何解释这一点？一名应用开发者、程序员或数据库管理员的目标是要做如下事情：

1) 尽可能地减少逻辑读取的数量。确保你正好且仅仅访问你需要访问的东西。换句话说，不要访问表格中的不必访问表或行。尽可能地采用简化操作。例如，假设一张 EMPLOYEES 表在DEPT\_ID上包含一个指向DEPARTMENTS表和一个称为DEPT\_NAMES的查询表的外键。当使用如下的语句检索一个 EMPLOYEES行时，在给定DEPT\_ID的情况下，现在假设一条通用的查询来检索DEPT\_NAME：

```
SQL> SELECT E.LAST_NAME, E.FIRST_NAME, N.DEPT_NAME
2> FROM EMPLOYEES E, DEPARTMENTS D, DEPT_NAMES N
3> WHERE E.DEPT_ID = D.DEPT_ID
4> AND D.DEPT_ID = N.DEPT_ID;
```

上述语句产生了预想的结果，但是没有必要对 DEPARTMENTS表进行访问。假如正确的完整性和有关的约束条件被定义和维护的话，这就是真实的。特别是，DEPT\_NAMES中的DEPT\_ID对于DEPARTMENTS表来说也是一个外键。因此，如果DEPT\_ID不首先存在于DEPARTMENTS表中的话，你决不会有存在于DEPT\_NAMES表中的DEPT\_ID。为了减少不必要的访问，该查询可以简化为如下所示：

```
SQL> SELECT E.LAST_NAME, E.FIRST_NAME, N.DEPT_NAME
2> FROM EMPLOYEES E, DEPT_NAMES N
3> WHERE E.DEPT_ID = N.DEPT_ID;
```

2) 尽可能地把物理读/逻辑读的比率减小到接近零。首先, 确保实现第一个目标——减少总逻辑读取。其次, 确保用最有效方式访问你需要访问的东西。使用迄今为止你已在本章中学到的知识。其余小节将为你提供更多的例子和方案。第三, 假如你已经完成了前两个工作, 那么在你的SGA中增加数据库缓冲区, 如果需要的话购买更多的内存。这将在第18章中进一步讨论。一般而言, 对于大部分的语句, 你的比率应该小于10%。在SELECT语句的tkprof输出示例中, 比值是13/304, 或大约0.04。这是一个好的比值。事实上, 这相当于96% (100-4) 的数据库高速缓存的命中率。

3) 尽可能地把逻辑读/行减小到接近零。对于每一次逻辑读取, 你希望能够读取尽可能多的行。因此, 比率越小, 你的读取越有效。这是块效率 (block efficiency) 的一个方面, 将在第19章中讨论。很明显, 假如比值太高了, 例如 1/8 (0.125) 或更大, 或许应增加db\_block\_size的值, 因为它相对于大的行容量来说似乎是太低了。在以前的tkprof示例中, 该比值较好 (13/879), 大约是.015, 或小于1/67。这意味着对于每15个数据块读取, 大约有1000行被检索, 或者说每个数据块大约67行 (假如你正在使用一个16K的Oracle db\_block\_size的话, 每行大约246个字节)。

除了刚刚提到的简化示例外, tkprof还提供了几个其他的命令行选项。最有用的在表17-5中讨论。解释执行计划将在随后的17.4节中更为详细地讨论。

表17-5 一些tkprof命令行选项

选 项	描 述
explain=user/password	使你作为一个用户 (user) 运行EXPLAIN PLAN; 该选项极为有用, 因为它为跟踪文件中的每一条语句提供了执行计划
record=file	使你能够指定一个文件 (file), 跟踪文件的语句保存在该文件中; 它还用来保存用于单个语句测试的脚本
sys=NO	抑制通过系统用户回忆的循环调用; 这很有用, 因为即使你可以通过缺省值查看这些调用, 它们也不能真正地被修改以有助于性能的提高——不考虑它们, 简化你必须过滤的输出量

要在屏幕上显示所有的选项, 在命令行键入 tkprof help=YES。

**警告** explain tkprof命令行选项在tkprof运行时刻提供了计划且并不基于SQL TRACE操作, 因此这两组输出或许不一致。

以下对使用SQL TRACE与tkprof进行了总结:

- 1) 当你有多个并发应用并且希望查明你的最大的资源密集型的应用时, 主要使用这些工具。把那些对于你的优先调整尝试不太有效的应用作为目标。
- 2) 在init.ora中或用ALTER SESSION打开SQL TRACE。
- 3) 运行你的应用。
- 4) 关闭SQL TRACE (假如你不打算不久后重新使用它)。
- 5) 运行tkprof, 把你的跟踪文件格式化到可读的输出文件中。
- 6) 对结果进行解释。
- 7) 按照提示调整你的应用或数据库。这经常意味着要重写查询。
- 8) 返回到步骤2, 然后重复, 直到你对步骤6的结果满意为止。

## 17.4 理解EXPLAIN PLAN

如果许多语句按一定的应用顺序运行，尽管 SQL TRACE 工具和tkprof程序可以提供有用的统计数字和执行计划信息，然而 EXPLAIN PLAN不必运行语句便可以一次为一条语句给出执行计划信息。这在分析单个问题查询中非常有用，尤其是如果该查询是运行时间很长的查询。当你可以通过使用 EXPLAIN PLAN立即得到它时，为什么要运行一条查询并通过 SQL TRACE或tkprof等待执行计划输出？因此，Oracle也提供这个非常有用的工具。

你如何使用 EXPLAIN PLAN？首先，运行脚本以创建存储 EXPLAIN PLAN输出的 PLAN\_TABLE。尽管目录和文件名或许随着操作系统平台的不同而变化，但这在 UNIX操作系统中通常是 \$ORACLE\_HOME/rdbms<version>/admin/utxlplan.sql。在NT上的位置是类似的。在你运行该脚本后，应当在你的模式下创建了 PLAN\_TABLE（假设你有正确的特权）。

其次，在你想要生成执行计划信息的语句上运行 EXPLAIN PLAN，例如用如下的查询：

```
SQL> EXPLAIN PLAN
      2> SET STATEMENT_ID='STMT_1'
      3> FOR
      4> SELECT EMPLOYEE_ID
      5> FROM EMPLOYEES
      6> WHERE EMPLOYEE_ID=243218;
```

使用如下的 SELECT 语句从 PLAN\_TABLE 中得到你想要的输出：

```
SQL> SELECT OPERATION, OPTIONS, OBJECT_NAME, ID, PARENT_ID, POSITION
      2> FROM PLAN_TABLE
      3> WHERE STATEMENT_ID='STMT_1'
      4> ORDER BY ID;
```

输出结果如下所示：

Operation	Options	OBJECT_NAME	ID	PARENT_ID	Position
SELECT STATEMENT			0	2	
TABLE ACCESS	BY ROWID	EMPLOYEES	1	0	1
INDEX	RANGE SCAN	EMPL_IDX	2	1	1

来源：Oracle公司

假如你喜欢的话，你也可以生成一个树型输出。可以使用如下的 SQL 代码得到一个缩进的文本图表示：

```
SQL> SELECT LPAD(' ', 2*LEVEL-1) || OPERATION || ' ' || OPTIONS || ' ' ||
OBJECT_NAME || ' ' || DECODE(ID, 0, 'COST= ' || POSITION) "QUERY PLAN"
      2> FROM PLAN_TABLE
      3> START WITH ID=0 AND STATEMENT_ID='STMT_1'
      4> CONNECT BY PRIOR ID=PARENT_ID;
```

输出结果如下所示：

```
Query Plan
-----
SELECT STATEMENT          COST=2
  TABLE ACCESS BY ROWID EMPLOYEES
    INDEX RANGE SCAN EMPL_IDX
```

这对于产生大的执行计划的大型查询来说常常很有用。就像父子关系一样，缩进表示了执行树中的另一个下属层次。前述的示例包含 3 个层次，成本等于 2。最重要的事情是证实，即你期望优化程序使用索引并且它使用了。假如它没有使用，你需要检查它没有使用的各种

原因，例如返回的行的数量、索引不具有高度选择性等等。努力去改变这种状况；然后返回到EXPLAIN PLAN并且查看是否你已经解决了该问题。在决定是否应继续使用优化程序时，EXPLAIN PLAN的重复使用是EXPLAIN PLAN最值得注意的用法。

下面对使用EXPLAIN PLAN进行了概述：

1) 当在你的应用中仅有一个或几个查询时，当已经把主要的性能问题孤立到一个或几个查询时，当你的查询通常顺序运行时或者当你不能提供重复执行时间用于等待每条 SQL TRACE/tkprof的运行时，使用EXPLAIN PLAN代替SQL TRACE或tkprof。

2) 运行\$ORACLE\_HOME/rdbms<version>/admin/utxlplan.sql。

3) 把EXPLAIN PLAN用于你的当前查询。

4) 从结果PLAN\_TABLE中选择文本或树输出。

5) 解释结果。验证你对优化程序性能的期望值。

6) 按照提示调整查询。这经常意味着重写查询。

7) 返回到步骤3并且重复，直到你对步骤5的结果满意为止。

## 17.5 鉴别典型问题

大部分应用调整问题的根源归结于不正确地使用索引。在 17.5.1 节中，你将浏览一下正确使用索引的一些规则。在 17.5.2 节中，你将考虑一些典型的应用问题，这些问题有的直接或间接地与索引有关，有的与索引无关。

### 17.5.1 正确使用索引

一般，大部分应用调整问题发生在决定是否使用索引的时候。随着时间的过去，索引既可能使用不足又可能使用过度。这是不同专业知识的数据库管理员之间的两个极端情况。你对索引所做的最好的事情就是必须正确地使用它。下面是正确使用索引的提示列表：

频繁用于WHERE子句中的索引列。

频繁用于连结表的索引列。

具有高度选择性的索引列（接近1）。

为具有低选择性（接近0）的列使用位映射索引。

除了上述的提示外，通过使用索引，你可以做一些别的事情以使你的数据库更有效。

例如，对具有高度选择性（接近1）并且几乎总是通过点（point）查询来访问的列使用哈希索引，点查询也称为精确匹配（exact match）查询。有具有相等比较关系的查询（例如WHERE x=y或WHERE s='abc'）为几乎所有其他列使用普通索引（B\*树），这些列趋向于采用实际的受限制的或不受限制的范围（搜索）查询率来访问。还有具有非相等比较关系的查询，例如WHERE x > y 或WHERE s LIKE c%。

不要对那些必须经常修改的列做索引。还有，设法抑制对具有高插入与删除操作的表中的任何列做索引。这两种情况中的任何一个都会引起 B\*树结构的连续重组并最终会由于索引碎片而导致性能变差。主键和外键例外。

对主键和外键做索引。不管你是否创建了一个索引，Oracle在已定义的关键上创建一个唯一索引。然而，缺省情况下Oracle不在外键上创建一个索引，除非你明确地用一条CREATE或ALTER命令执行这项操作。之所以这样做是因为当子表通过DML访问时不锁定父表可以增强系统性能。

在适当的时候，增加辅助列以形成复合索引。这利用了优化程序的索引覆盖功能，索引覆盖在较早前讨论过。Oracle不像Sybase，它使用一个更受限制的索引覆盖形式。Oracle强加约束以便列只有在它们形成那个（复合）索引的主要列的时候才能够从索引中检索。假设一张表在a+b+c列上有一个复合索引。优化程序通过使用a、a+b或a+b+c而不是b、c、b+c或a+c把该索引用于一个WHERE子句。对于Oracle来说，索引覆盖仅能用于复合索引的左侧排序子集（前缀）。

不要对那些总是用于具有函数或操作符（MIN或MAX操作符除外）的WHERE子句中的列做索引。在这种情况下，优化程序不使用索引。换句话说，创建一个导出列或重写该查询，使用提示或技巧绕过这个操作。使用EXPLAIN PLAN验证你的结果。

不要对那些主要用于否定或NULL比较的列做索引。Oracle优化程序不使用WHERE子句中基于如下比较操作的索引：

IS NULL

IS NOT NULL

!=

做为选择，积极地使用=或IN操作符重写查询。如果使用NULL比较，你或许需要考虑使用缺省值。

小心使用视图和复杂子查询的索引。如果给定各种层次的视图和复杂子查询语句，很难预计将要发生的事情。你必须反复使用EXPLAIN PLAN以确定这些查询正在做什么。假如你不能使你的查询按照要求运转，考虑通过把子查询压缩到一个较大的查询中或把它们分裂为多个连续的子查询的方式来重写它们。还有，考虑放弃视图或者实现它们。

最后，不要忘记适当地做索引（参见以前的步骤），不要为了做索引而做索引。我曾经见过表中不含索引的数据库，并且我也见过表中在每件东西上都有索引的数据库。没有索引一般不是好事情，因为你总应该有一个主键索引。在每件东西上都有索引一般也没益处，因为这经常是不良的需求分析、不良的数据库设计或不良的应用设计的症状。

既然已经讨论了主要的索引准则，你可以查看一些在应用调整中发现的更为典型的问题。

### 17.5.2 应用调整的典型问题

几乎在所有的应用调整中遇到的典型问题都涉及到某些类型的低效率的SQL编码或不适当地使用索引。本节对这些问题中的其中几个进行讨论。

有一个称为索引覆盖异常（index coverage anomaly）的问题。一个关于索引覆盖的有趣的事情可能会发生。当它发生时，那么通过此查询返回的行数比较大，因而全表扫描可能会更加有效。然而，假如ANALYZE没有运行，优化程序借助于基于规则的优化并选择索引访问路径。如果数据库管理员没有保持最新的统计数字，那么优化程序可能选择不良的访问路径，尽管它本身并没有缺陷。

可能发生的另一个问题被称为不必要的多次通过或多回路。请看如下查询：

```
SQL> SELECT A FROM T WHERE A > 9;
```

```
SQL> SELECT B/A FROM T WHERE B < 3 AND A > 9;
```

很明显，这可以通过表T在一个表中完成：

```
SQL> SELECT A , B/A FROM T WHERE B < 3 AND A > 9;
```

使用不必要的函数或NULL比较是另一个可能发生的问题。我们已经讨论过如何正确使用

索引，优化程序不会在这些情形下使用索引，所以不会做如下的事情：

```
SQL> SELECT A FROM T WHERE X < MAX(X);
```

正如较早前讨论过的一样，你希望使用直接量并且以如下形式限制范围搜索：

```
SQL> SELECT A FROM T WHERE X >= 0 AND X < 999999;
```

还有，如果可能的话，把 NULL 列转化为一个缺省条件：

使用

```
SQL> SELECT A FROM T WHERE X > 0;
```

而不是

```
SQL> SELECT A FROM T WHERE X != 0;
```

或者

```
SQL> SELECT A FROM T WHERE X IS NOT NULL;
```

如果是有意的话，后两条语句都没有在 X 列上使用索引。

不要忘记对外键做索引。当子表正被插入、修改或删除时，这样做会引起子表锁定父表，反之亦然。假如你对所有的外键做索引，对子表的修改不会锁定父表并且对父表的修改仅得到一个行级锁定。

当表能够以某种方式推导出时，不要从该表中选取信息。涉及到数据库的一个主要应用编程规则如下：

注意 在数据库应用中对存储进行计算是一件好事情。

这是事实，因为大部分数据库应用趋向于受输入/输出的限制。例如，假设你正在计算基于单一固定税率（或者少数平齐的税率）的销售税。税表通常是静态查询表，利率在一个给定的税期（例如，一年）中通常是不变的。下面的示例展示了如何使用一个函数，而不必访问一个已存储的推导列。

不要使用下述语句：

```
SQL> SELECT (P.COST + (P.COST * T.RATE)) "COST WITH TAX"  
2> FROM PRICES P, TAXES T  
3> WHERE P.PART_ID = T.PART_ID
```

使用如下语句：

```
SQL> SELECT (COST * 1.06) "COST WITH TAX"  
2> FROM PRICES;
```

这说明原始设计中有一些问题，你或许不需要 TAXES 表。然而，把该问题搁置一边，你当然不必访问税率列（RATE）才够用那个乘数。这是事实，因为你正在依赖一个常量（简单的、硬编码的直接量替换值）。如果开发者在这一业务领域有充足的知识并且如果知识量相对较小且所有开发者都很熟悉的话，那么这可以很好地运行。总之，该例子表明了一个简单的函数如何替代一个表访问。请注意，硬编码技术被认为是不良的程序设计，但是你的目的是取得好的应用性能。这两者经常有矛盾，最后这个例子也不例外。你必须决定你的所有目标中哪一个是最重要的目标。

不要触发不必要的全表扫描，引起累积百分数问题。当你不需要时为什么要对一张表进行扫描？通常，你必须寻找会引发一个全表扫描的查询，换句话说，查询不使用一个会有意或可能以另一种方式引起一个全表扫描的索引。还有，不要忘记 5% 规则。假如优化程序希望从查询中返回的值超过 5%（在版本 7.x 或 7.x 以上），就要求助于全表扫描。使用如下语句，在

合并多个 WHERE 条件总计该数量时可能发生全表扫描：

```
SQL> SELECT A FROM T WHERE X = 1 AND Y = 2 AND Z = 3;
```

如果统计数字显示  $x=1$  有 3%， $Y=2$  有 2%， $Z=3$  有 2%，优化程序会发现累积百分数达到了 7% 并选择全表扫描。你也可以把这两个查询拆开，尤其是对于大表而言，这样就可以在同样的结果下取得较好的性能：

```
SQL> SELECT A FROM T WHERE X = 1;  
SQL> SELECT A FROM T WHERE Y = 2 AND Z = 3;
```

优化程序在 X、Y、Z 上使用索引，因为第一条查询仅显示 3%，第二条查询仅为 4%。

小心地使用 DISTINCT、ORDER BY 与 UNION。这些操作会创建临时表并增加排序的附加系统开销。作为一个数据库管理员，除非你有无限的资源，否则你几乎不能容忍这些事情同时发生。许多这些同时发生的操作与同时发出许多 CREATE INDEX 语句有相同的性能消耗。假如你必须使用 DISTINCT 的话，那么就频繁地使用它并且存储输出值用于将来的使用。假如你必须使用 ORDER BY，设法在那一列上设置一个索引。假如你必须使用 UNION，那么就使用 UNION ALL（不消除副本）。只要你已定义并启用所有的主键，就可以在大多数情况下使用后一种解决方案。

**注意** 在任何应用调整中，因为应用前端或许用其他 SQL 语句编写，所以研究已翻译的 SQL 语句。由于各种原因，ODBC 驱动程序翻译不总是生成优化的查询。注意这些情况并打开 ODBC 跟踪文件以便你可以看到已翻译的 SQL 语句，它们或许正阻止索引的使用。

## 17.6 重写查询

在前几节中已重写了大量查询。现在，把注意力集中在重写查询的几个示例中，在这些示例中你会看到如何能够真正地调整 SQL，使之具有高效率。除了已经讨论过的许多技术外，最好的两个技术如下：

使用集合操作符。

使用布尔转换。

重写查询的领域高度数字化，然而突出了实用。当你听到重写查询时，就意味着重新编写它们以加快速度，而不是调整它们。后者遵循第一次正确得到它的法则，然后对程序设计加以维护。你所感兴趣的是原始速度，经常可以牺牲可读性和可维护性。因此，在投入这一领域的冒险以前就必须做出决定。

### 17.6.1 使用集合操作符

你已经看到 UNION 可以降低性能。你应更感兴趣的是使用 MINUS 或 INTERSECT。它们实际上有助于性能的提高。观察如下的查询：

```
SQL> SELECT ACCOUNT_ID  
2> FROM JOBS_COMPLETED  
3> WHERE ACCOUNTS_ID NOT IN (ACCOUNTS_PAID);
```

它可以重写为如下形式：

```
SQL> SELECT ACCOUNT_ID FROM JOBS_COMPLETED  
2> MINUS  
3> SELECT ACCOUNT_ID FROM ACCOUNTS_PAID;
```

这两个查询都给出了相同的操作结果：它们返回那些已经完成了他们的工作但还没有得

到付款的客户。然而，使用 EXPLAIN PLAN 可能会发现第一次的总逻辑读取比第二次的更大。在 Oracle 中，使用 MINUS 运算符是非常有效的。

除了 MINUS 是一个非对称（单向）运算符而 INTERSECT 是对称（双向）运算符外，INTERSECT 像是对 MINUS 的补充。也就是说，对称运算符在两个方向上的工作是相同的。UNION 是对称运算符。如果你交换在以前的示例中使用 MINUS 所选取的表，你会得到一个不同的结果。你将得到所有的已得到付款但还没有完成他们的工作的客户（这或许不是一般的商业惯例）。对于另外一个例子，要查找所有已完成他们的工作并且已得到付款的客户，使用如下语句：

```
SQL> SELECT ACCOUNT_ID FROM JOBS_COMPLETED
2> INTERSECT
3> SELECT ACCOUNT_ID FROM ACCOUNTS_PAID;
```

### 17.6.2 使用布尔转换

布尔表达式是一个求值结果为 TRUE 或 FALSE 的表达式。一条 SQL 语句的 WHERE 子句便是布尔表达式的一个例子。你可以利用这个事实，提供一个将 WHERE 子句（布尔表达式）转化为数值的函数，例如用 1 表示 TRUE，用 0 表示 FALSE。这样做有何用？如果给出四种类型的婚姻状况（S=单身，M=已婚，D=离婚，W=丧失配偶），考虑如下用于获得不同的税率的查询：

```
SQL> SELECT SINGLE_TAX "TAX" FROM TAXES WHERE STATUS='S';
SQL> SELECT MARRIED_TAX "TAX" FROM TAXES WHERE STATUS='M';
SQL> SELECT DIVORCED_TAX "TAX" FROM TAXES WHERE STATUS='D';
SQL> SELECT WIDOWED_TAX "TAX" FROM TAXES WHERE STATUS='W';
```

还是先把数据库设计问题搁置一边，因为这或许是一个不良设计的表，你可以发现为了得到需要的信息，必须执行 TAXES 表的四个全表扫描。这是一个好的示例，因为你正在处理两个以上的不同值。你可以做得更好吗？首先，你需要一个布尔转换。换句话说，你需要一个函数以便在 STATUS='S' 时返回 SINGLE\_TAX，在 STATUS='M' 时返回 MARRIED\_TAX，诸如此类。Oracle 恰好提供了这样一个函数——DECODE。假如你仔细地研究了下述查询，你会看到它完全替代了以前的四个查询并提供相同的操作结果，然而它仅需要一个表扫描。

```
SQL> SELECT DECODE(STATUS, 'S', 1, 0)*SINGLE_TAX +
2> DECODE(STATUS, 'M', 1, 0)*MARRIED_TAX +
3> DECODE(STATUS, 'D', 1, 0)*DIVORCED_TAX +
4> DECODE(STATUS, 'W', 1, 0)*WIDOWED_TAX
5> "TAX"
6> FROM TAXES;
```

这明显地提高了效率。然而，正如你所料，可读性与可维护性受到了破坏。除此之外，使用 DECODE 函数不总能发挥作用。你或许要使用其他的转换函数。在本例中，你需要的是一个相关数组，DECODE 仅用于提供那种类型的函数功能。显而易见，困难的工作是检查原始查询以提出一个给你相等结果的函数。

## 17.7 介绍 Oracle 8 的新的索引特性

几个关于索引的新特性已经添加到 Oracle 8 中，毫无疑问，这些新特性在将来的应用开发中会大有用处。这些新特性将在下面讨论。

### 17.7.1 使用索引分区

Oracle 8 使你能够按分区键将索引分区（物理划分）。这意味着按组成索引的同一列划分

该索引。分区可以存储在分开的表空间中，这些表空间具有各自的存储参数。因此，分区是“子表”。下面便是一个例子：

```
SQL8> CREATE INDEX EMPL_IDX ON EMPLOYEES(EMPLOYEE_ID)
2> PARTITION BY RANGE (EMPLOYEE_ID)
3> (PARTITION ip1 VALUES LESS THAN (499999)
4> TABLESPACE empl_idx1,
5> PARTITION ip2 VALUES LESS THAN (1000000)
6> TABLESPACE empl_idx2);
```

### 17.7.2 使用相等分区的本地索引

在Oracle 8中，假如你创建了一个本地索引，Oracle会自动地对它进行同等分区。也就是说，它使用相同的分区键、相同的分区数以及分区界限，这与它引用的分区表相同。本地索引是一种索引，在该索引中一个分区中的所有索引键指向一个表分区中的所有数据。这是一对一映射。全局索引是另一种索引，在该索引中一对一映射并不适用。这确保了表与它的索引被同等分区。除了类似于数据条的较高的可用性外，本地索引的相等分区能够使优化程序知道分区。下面便是一个例子：

```
SQL8> CREATE INDEX EMPL_IDX ON EMPLOYEES(EMPLOYEE_ID)
2> LOCAL
3> (PARTITION ip1 TABLESPACE empl_idx1,
4> PARTITION ip2 TABLESPACE empl_idx2);
```

### 17.7.3 使用知道分区的优化程序

正如刚刚提到的那样，假如你创建相等分区的本地索引，优化程序可以通过使用分区知识生成查询计划。因此，它可以并行执行一些操作。

### 17.7.4 使用唯一索引表

在Oracle 8中，你现在可以创建一张唯一索引表。它也称为一个在位（in-place）的索引。本质上，该表被物理排序，而不是使用B\*树的逻辑排序。把逻辑读取从B\*树移动到表中能够给性能带来明显的提高，因为数据和索引是相同的。再有，通常把主键作为索引列。一个例子如下：

```
SQL8> CREATE TABLE EMPLOYEES
2> (EMPLOYEE_ID NUMBER(6) CONSTRAINT empl_pk PRIMARY KEY,
3> <column, column, ...>)
4> ORGANIZATION INDEX TABLESPACE empl_dat1;
```

ORGANIZATION INDEX子句告诉Oracle 8这是一个唯一索引表。

### 17.7.5 使用反向键值索引

反向键值（reverse key）索引是一种索引类型，在该索引中单个列字节的顺序被反向（不是列序）。已经证明反向键值索引在提高Oracle并行服务器（OPS）性能方面非常有效。在CREATE INDEX中使用REVERSE关键字创建一个用于Oracle并行服务器的索引。