高效 ORACLE 之 索引

关键词：ORACLE 、Effective、高效、B*Tree Index、索引倾斜、索引空间丢失、B*Tree Cluster Index、Reverse key index 、反向键索引、Descending Index、降序索引、函数索引、FBI、 位图索引、Bitmap index、位图联接索引、Bitmap Join Index、Application Domain Index 、应用域索引、CBO、Case Insensitive、选择性索引、选择惟一性、压缩键索引、Compressed Key Index、模式。

测试环境：

ORACLE：Oracle Database 10g Enterprise Edition Release 10.2.0.1.0
操作系统：Windows XP Professional Sp2
硬　　件：IBM R50 笔记本（CPU 1.4Ghz/Mem 512M）

作者信息：

单　　位：信贷及管理信息事业部
作　　者：李杰
电　　话：7663

参考文献：

《Oracle10g Database Concepts Release 2》……...ORACLE
《Oracle Database 10*g*, Release 2New Features》……...ORACLE
《Effective Oracle By Design》……Thomas Kyte
《Cost-Based Oracle Fundamentals》……..Jonathan Lewis
《2 Day + Performance Tuning Guide 10gR2》…… ..ORACLE
《Oracle.High.Performance.Tuning.for.9i.and.10g》……Gavin Powell
《Expert Oracle Database Architecture》…… Thomas Kyte
《Administrator's Guide 10gR2》……..ORACLE

- 降序索引（Descending Index）：降序索引是 B*树索引。它允许数据在索引结构中按“从大到小”的顺序排序。本文下面的内容会说明为什么降序索引比较重要，并说明降序索引是如何工作的。
- 反向键索引（Reverse Key Index）：这也是一种 B*树索引，只不过键中的字节是“反转”的。这对键中存在顺序的连续值（比如序列生成的键值）的索引很有用途。通过“反转”，本来连续的键值，比如 987500、987501、987502，ORACLE 会逻辑的对 205789、105789、005789 等建立索引。这样本来连续的数据就变得相距甚远，对这些索引键值的插入就会分散到多个块（Block）上，从而有效降低大批量数据插入时对同一数据块的‘争用’，避免‘热块’的产生，从而提高效率。

- 位图索引（Bitmap Index）：索引的目的是提供一个指向所给键值对应数据行的指针（RowID）。在普通的索引中，这是通过存储每一个键值对应数据行的 ROWID 来实现的。通常索引条目和数据行之间存在着一种一对一的关系：一个索引条目就指向一行。而对于位图索引，它使用键值的位图来取代普通索引的 ROWID 列表，一个位图索引条目可能会指向多行数据。位图索引适用于高度重复（低基数）且通常只读的数据表。在 OLTP 系统中，由于存在着并发性相关的性能问题，所以不能考虑使用位图索引。而数据以‘只读’为主要特征的 Data Warehouse/MIS/BI/OLAP 应用是位图索引的首选。
- 位图联接索引（Bitmap Join Index）：位图联接索引提供了在表 A 上创建包含表 B 中字段的索引的能力。比如：

  CREATE BITMAP INDEX A_BM_IDX
  ON **A (B.FILED1_NAME)**
  FROM A, B
  WHERE **A.FILED1_CODE=B.FILED1_CODE**;

  这个索引是指向表 A 的，而并不是指向表 B。这是一个全新的概念：能从其它表对某个表属性建立索引。这中索引会改变我们在 Data Warehouse/MIS/BI/OLAP 应用中数据模型设计的方式：一方面保持规范化的数据结构不变，一方面还能得到逆规范化的诸多好处。
- 基于函数的索引（Function-Based Index）：FBI 是 B*树索引或者位图索引，所不同的是，它将一个函数计算得到的结果存贮在索引中，而不是直接存储列数据本身。FBI 可以加快形如 SELECT * FROM A WHERE FUNCTION(T.COL1)＝SOME_VALUE 的查询，因为值 FUNCTION(T.COL1)已经提前计算并存储在索引中。
- 应用域索引（Application Domain Index）：ADI 是你自己构建和存储的索引。ORACLE 通过 ADI 给应用设计人员提供了‘DIY’索引的能力。该索引可以存在 ORACLE 中，也可以存储在 ORACLE 外，你要告诉 ORACLE 索引的选择率如何以及执行的开销有多大。优化器（OPTIMEZER）会根据你提供的信息决定是否使用该索引。

按应用类型分

按照索引的**应用**类型划分，我们还有另外一种索引分类方式：

- 高效读写索引（Efficient as Read-Write Indexes）
  - B*树 索引（B*Tree Index）
  - 基于函数的索引（Function-Based Indexex）
  - 反向键索引（Reverse Key Index）
- 高效只读索引（Efficient as Read-Only Indexes）
  - 位图索引（Bitmap Indexes）

- 位图联接索引（Bitmap Join Indexes）
- 索引组织表（Index-Organized Table）
- B*树索引群集和哈希群集（B*Tree Index Cluster & Hash Cluster）

- 应用域索引（Application Domain Index）：非常特别的一种索引，关于它的阐述超出了本文的范畴，在这里不做详细介绍，有兴趣的话可以参看 ORACLE 相关技术文档去了解和学习。
- 性能调整有关的索引类型（Index relevant to tuning）：
  - 升序或降序索引（Ascending or descending Indexes）
  - 唯一索引（Unique Indexes ）
  - 复合索引（Composite Indexes）
  - 压缩索引（Compression Indexes）
  - 反向键索引（Reverse Key Indexes）
  - 其它（诸如 NOSORT 或者 NULL 空值等）

与性能调整有关的这些索引的使用都具有极强的技巧性，一般来说，这些通常也是我们调优时经常使用的技术。

由上面的描述可以看到，在 ORACLE 中可供选择的索引类型非常丰富，本文的余下部分将通过提供一些技术细节来分别对每种索引深入进行阐述，这包括：如何（How）、何时（When）、何地（Where）使用这些索引。

B*Tree 索引

B*Tree 索引概述

B*Tree 索引是最‘传统’的索引，也是使用最多的一类索引结构。它的目标是尽可能减少 ORACLE 查找数据所花费的时间。它的实现与二叉查找树很类似。不严格的说，如果在一个字符串列上创建一个 B*Tree 索引，那么从概念上讲这个索引的结构可能如下图 1 所示：

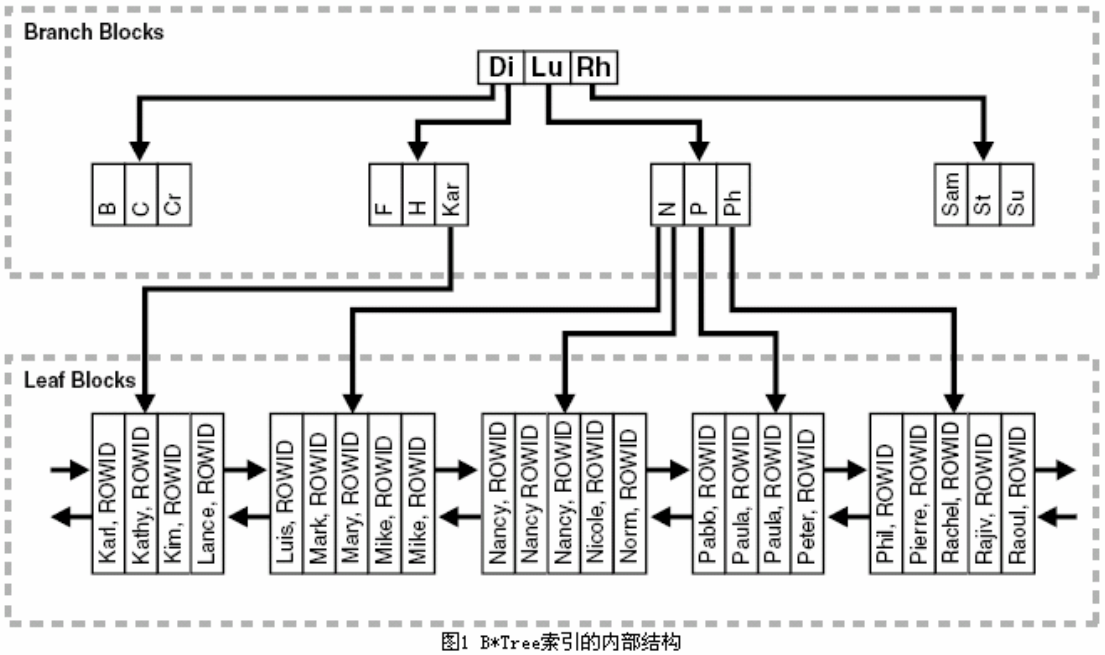

图1 B*Tree索引的内部结构

图中，上面虚线筐内的数据块叫 Branch Block（分支块），它提供到具体的 Leaf Block（页块或者叶子节点，即下面虚线筐内的数据块）数据块的‘导航’，真正的索引数据是存储在 Leaf Block 上的。以查找名为 Karthy 数据行为例：从**根结点开始**（**第一个 I/O**）找到**第二层**中从左侧第二个分支（指针），从该分支开始（**第二个 I/O**）找到**第三层**页块中的最右侧分支（指针），然后再进行**第三个 I/O** 找到该页块，从中找到名为 Karthy 的索引条目，根据里面存储的 ROWID 即可找到名为 Karthy 表记录（这需要另外一个 I/O）。需要特别指出的是，索引的叶子节点实际上是构成一个**双向链表**（如上图所示），所以一旦导航到叶子节点之后（即发现第一个索引键值），执行索引键值的有序扫描就变得很容易，‘索引键值的有序扫描’就是我们经常在执行计划中看到 **Index Range Scan**（索引范围扫描），执行 Index Range Scan 时，ORACLE 不用反复在索引结构中做从根节点到叶子节点的导航，而只是在第一次到达页块后，根据需要在叶子节点中沿双向链表向前或者向后扫描就可以了。所以，如果使用了 B*树索引，ORACLE 要满足诸如以下的谓词条件就相当简单：

```
WHERE  XXX BETWEEN 1000 AND 2000
```

ORACLE 发现第一个最小键值大于或等于 1000 的页子节点后，就会水平的通过双向链表遍历其它页子节点，直到最后发现一个大于 2000 的值为止。整个过程只需从根节点到叶子节点‘导航’一次。

B*树索引结构中，所有的叶子都在同一层上，整个索引结构数据块的层数称之为索引的高度（Height），在图 1 中我们可以看到这个索引的高度是 3。由图 1 可知，无论根据索引查找表中的哪一条记录，所执行的 I/O 此次数都是相同的。即对于诸如如下形式的查询：

```
SELECT BTREE_INDEXED_COL FROM T  WHERE  BTREE_INDEXED_COL=:X
```

无论:X 的值是多少，ORACLE 都将会执行同样的 I/O 次数来找到该值。这就是 B*Tree 索引中的‘B’的含义，即 Balanced（平衡）之意。换句行话说，B*Tree 索引是高度平衡的（Height Balanced）。一个表哪怕有 1000 万条记录，只要其索引高度是 3，那么找到其中一条记录所执行的 I/O 次数，与从具有同样索引高度的 1000 条记录的表中找到一条记录所执行的 I/O 次数是一样的。这就是 B*Tree 索引的魅力所在，也就是因为这一点使它成为一种非常通用的索引，目前几乎所有关系数据库都支持它。

另外，ORACLE 在表示从索引根结点到叶子节点块遍历所涉及的数据块数时使用了两

个稍微不同的技术术语。一个就是前面提到的高度（Height），这是从根节点到叶子结点所需遍历的块数。另一个术语是分支层数（Blevel），它等与 Height-1，即它不把叶子节点层计算在内。下面通过一个例子来加深对这两个术语的理解。

以我在《高效 ORACLE 之 装载》中建立的 BIG_TABLE 测试表为例，该表有 1833792 条记录，并且在 ID 列上存在主键索引(B*树索引)。

```
scott@ORCL> select index_name,blevel,num_rows
  2  from user_indexes
  3  where table_name='BIG_TABLE'
  4  ;
INDEX_NAME                          BLEVEL     NUM_ROWS
------------------------------ ---------- ----------
BIG_TABLE_PK                             2      1833792
```

BLEVEL 为 2，说明索引分支层数为 2，则索引高度为 3，即 ORACLE 找到索引的叶子节点上的索引键值需要执行 3 次 I/O。下面验证一下：

```
scott@ORCL> select max(id),min(id) from big_table;
    MAX(ID)       MIN(ID)
---------- ----------
    1833792              1
scott@ORCL> SET AUTOTRACE TRACEONLY
scott@ORCL> select id from big_table where id = 1;

执行计划
----------------------------------------------------------
Plan hash value: 1688966252
-------------------------------------------------------------------------------------------------------------
| Id  | Operation                    | Name                | Rows  | Bytes | Cost (%CPU)| Time       |
-------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                     |     1 |     6 |     2   (0)| 00:00:01 |
|*  1 |  INDEX UNIQUE SCAN           | BIG_TABLE_PK        |     1 |     6 |     2   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("ID"=1)

统计信息
----------------------------------------------------------
          1  recursive calls
          0  db block gets
          3  consistent gets
          0  physical reads
          0  redo size
```

```
          402  bytes sent via SQL*Net to client
          385  bytes received via SQL*Net from client
            2  SQL*Net roundtrips to/from client
            0  sorts (memory)
            0  sorts (disk)
            1  rows processed

scott@ORCL> select id from big_table where id = 1833792;

执行计划
----------------------------------------------------------
Plan hash value: 1688966252
-------------------------------------------------------------------------------------------------------
| Id  | Operation                    | Name                | Rows  | Bytes | Cost (%CPU) | Time     |
-------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                     |     1 |     6 |     2   (0)| 00:00:01 |
|*  1 |  INDEX UNIQUE SCAN           | BIG_TABLE_PK        |     1 |     6 |     2   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("ID"=1833792)

统计信息
----------------------------------------------------------
          1  recursive calls
          0  db block gets
          3  consistent gets
          0  physical reads
          0  redo size
        405  bytes sent via SQL*Net to client
        385  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed

scott@ORCL>
```

可以看出，无论是查找索引中最小键值 1，还是查找最大键值 1833792，ORACLE 都执行了三次 I/O（**consistent gets = 3**）。因此 B*树索引是一个绝佳的通用索引机制，无论是大表还是小表都很适用，随着底层表数据的增长，获取数据的性能几乎不会恶化。

B*Tree 索引子类介绍

B*Tree 索引有几种子类别，这些子类在解决某些特别的性能问题时，有时会非常有用。

反向键索引

反向键索引是一种能够将索引键‘反转’的 B*Tree 索引，反向索引的结构请参考图 2。

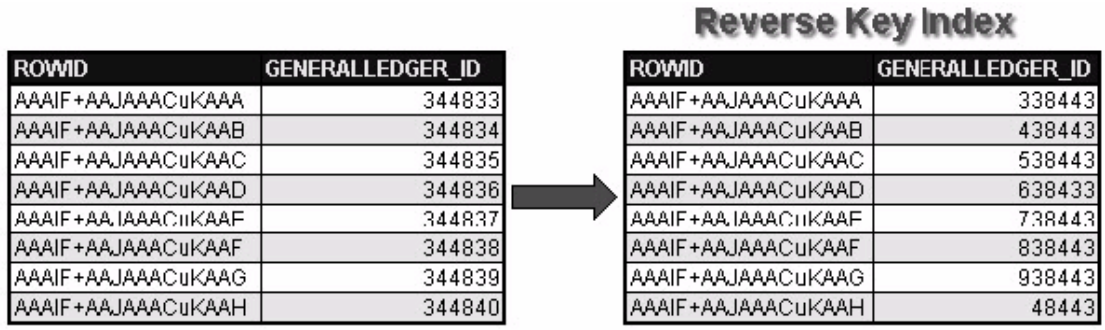

图2 反向键索引示例

从图中可以看出，通过把键‘反转’，本来连续的键值变得非常‘离散’，这样当对基表进行海量数据插入操作时，尤其是在 RAC 环境下，这些本来应该放在一个索引数据块上的连续键值（一般是由序列或者时间戳产生）会被分散到不同的索引块上，从而避免索引块争用的问题。因此，反向键索引可以看作是针对普通 B*Tree 索引的一种优化手段，其实反向键索引就是为了减少“右侧”索引中对索引页块争用而特别设计的，它能有效缓解‘右侧’索引的缓冲区忙等待。

在 RAC 环境下，如果两个或更多的实例需要同时修改同一个索引数据块，它们会通过 Interconnect 来回传递这个块来实现共享，interconnect 是两个或者多个机器之间的一条专用网络连接。如果某个索引列用一个序列进行填充（这是非常流行的应用设计方式），那么每当新值插入时，都会试图当前索引结构中‘右侧’的块（因为键值是规律递增的）。如果把这种‘序列’键值反转，本来连续的值就会‘相距甚远’，从而在多个实例同时插入数据时，对索引数据块的写入就会分布在多个数据块上，这样就不会争用最‘右侧’的索引页块，不会产生所有实例一起写同一个数据块引起的缓冲区忙等待事件，从而提高语句执行的效率。

根据反向键索引的特征，很容易发现反向键索引的一个缺点：能用常规索引的地方不一定能用反向键索引。例如，回答如下问题时，COL1 列上的反向键索引就没有用：

```
WHERE COL1 > 98501
```

因为数据是按 REVERSE(COL1)排序并存储在索引中的，所以对与大于、小于等这类区间扫描就不能使用反向键索引。

如果您有兴趣的话，可以自己做试验来证明反向键索引在减少缓冲区忙等待方面的突出贡献。限于篇幅问题，本文不在举例。但是给您提供了如下试验方案仅供参考。

1、创建两张试验表，一个上面建立反向键索引，一个建立普通的索引。为了更能说明问题，请使用序列填充键值；
2、写一个存储过程，循环往两张表中插入 50000 条（或者更多）记录；
3、分别在单用户、5 用户、10 用户下执行并发执行步骤 2 中创建的存储过程。
4、并发执行过程中，启用 ORACLE TRACE 跟踪功能，统计执行中每秒执行的事务数、

占用 CPU 时间、缓冲区忙等待事件出现的次数和等待的时间、总耗时等数据；
5、比较反向键索引和普通索引的各项数据。

下面各表是 Tomas Kyte 在其一本著作中提供的一组测试结果数据，仅供参考：

表 1 利用 PL/SQL 对使用反向键索引进行性能测试：单个用户

统计项	反向键索引	普通索引
事务/秒	38.24	43.45
CPU 时间（秒）	25	22
缓冲区忙等待次数	0	0
缓冲区忙等待时间	0	0
总耗用时间（分钟）	0.42	0.37

表 2 利用 PL/SQL 对使用反向键索引进行性能测试：5 个用户

统计项	反向键索引	普通索引
事务/秒	46.59	49.03
CPU 时间（秒）	77	73
缓冲区忙等待次数	**4267**	**133644**
缓冲区忙等待时间	2	2
总耗用时间（分钟）	0.68	0.65

表 3 利用 PL/SQL 对使用反向键索引进行性能测试：10 个用户

统计项	反向键索引	普通索引
事务/秒	45.90	35.38
CPU 时间（秒）	781	789
缓冲区忙等待次数	**26846**	**456231**
缓冲区忙等待时间	279	1382
总耗用时间（分钟）	3.47	4.5

可以看到，随着并发用户数的增多，反向键索引的优势越来越明显，通过减少缓冲区忙等待的次数，反向键索引使系统的吞吐量大幅提升。

降序索引

ORACLE 8i 中引入的降序索引是用来扩展 B*Tree 索引功能的。降序索引允许以降序（从大到小顺序）存储一列。为了说明降序索引的用法，我们可以做如下试验。

首先，创建实验用表 T_DES_IND，并在其上创建一个复合索引：

```
scott@ORCL> create table t_des_ind as select * from all_objects;
表已创建。
scott@ORCL> create index t_des_ind_idx on
  2   t_des_ind(owner,object_type,object_name)
  3   /
索引已创建。
```

```
scott@ORCL>
```

查询该表，使用 ODERBY DESC 子句：

```
scott@ORCL> select owner,object_type
  2  from t_des_ind
  3  order by owner desc,object_type desc
  4  /

执行计划
----------------------------------------------------------
Plan hash value: 2356375540
-------------------------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name              | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |                   | 49729 |  1359K|   369   (1)| 00:00:05 |
|   1 |  INDEX FULL SCAN DESCENDING| T_DES_IND_IDX | 49729 |  1359K|   369   (1)| 00:00:05 |
-------------------------------------------------------------------------------------------------------------------
```

上面的试验说明：ORACLE 可以实现正常的往前读索引，因为这个执行计划中的最后没有排序（SORT）步骤（ORACLE 直接从后往前扫描索引即获得了降序的查询结果）。因此你可能觉得降序索引并没有用。但是，请考虑如下查询：

```
scott@ORCL> select owner,object_type
  2  from t_des_ind
  3  order by owner desc,object_type ASC
  4  /

执行计划
----------------------------------------------------------
Plan hash value: 3934334124


-------------------------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name              | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT          |                   | 49729 |  1359K|       |   480   (3)| 00:00:06 |
|   1 |  SORT ORDER BY            |                   | 49729 |  1359K|  3544K|   480   (3)| 00:00:06 |
|   2 |   INDEX FAST FULL SCAN| T_DES_IND_IDX | 49729 |  1359K|       |    83   (3)| 00:00:01 |
-------------------------------------------------------------------------------------------------------------------
```

可以看出，执行计划中已经有了 SORT ORDER BY 动作了，即这个查询不能直接从索引中得到排序结果，而只能从索引返回结果后，再对结果进行额外的排序操作，而排序操作的成本一般是较高的，需要消耗大量的 CPU、内存以及是临时表空间。但是，如果我们使用降序索引，结果就会非常不同。

```
scott@ORCL> create index t_des_ind_desidx on t_des_ind(owner desc,object_type asc);
索引已创建。
scott@ORCL> exec dbms_stats.gather_index_stats(user,'T_DES_IND_DESIDX');

PL/SQL 过程已成功完成。
```

```
scott@ORCL> select owner,object_type
  2     from t_des_ind
  3     order by owner desc,object_type ASC
  4      /
执行计划
----------------------------------------------------------
Plan hash value: 3716002730
-------------------------------------------------------------------------------------------------
| Id  | Operation                   | Name                 | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |                      | 49729 |  1359K|   201   (2)| 00:00:03 |
|   1 |  INDEX FULL SCAN | T_DES_IND_DESIDX | 49729 |  1359K|   201   (2)| 00:00:03 |
-------------------------------------------------------------------------------------------------
```

可以看出，通过巧妙的使用降序索引，ORACLE 又能够从索引结构中直接得出排序的查询结果，从而‘节省’了对查询结果的昂贵的 SORT 步骤。再看语句执行消耗的时间，降序索引所用时间（**00:00:03**）仅仅为普通索引（**00:00:06**）的 50%。

从上面的试验得出，降序索引的使用与应用中数据访问的方式是分不开的，也许它的使用面比较‘窄’，但是，如果能巧妙的加以使用，它会令你的数据加工事半功倍。

B*Tree Cluster 索引

请参考职工月报第九十七期中《高效 ORACLE 之 模式》。

索引组织表

请参考职工月报第九十七期中《高效 ORACLE 之 模式》。

B*树索引调整

B*树索引简单易用，其应用也由来已久，而有关 B*树索引导致的性能问题的研究和讨论也源远流长。在长期的应用实践中，广大设计、开发人员已经积累了很多有关 B*树索引性能调整的知识（当然，这里面也有关系数据库厂商的积极参与）。下面分别介绍一下复合 B*树索引的键压缩、‘倾斜’（Skewed）的 B*树索引的重组以及 B*树索引空间丢失的处理，这些都是我们在实际项目中经常碰到的有关 B*树索引性能调整的问题。

压缩键索引

B*树索引的压缩只能应用于复合索引。压缩键复合索引（Compressed Key Index）的基本概念是：把每个索引键条目分解为两个部分：前缀（Prefix）和后缀（Postfix）。如下图 3 所示，前缀是指复合索引中前面具有大量重复值的部分，余下的索引条目中惟一的部分就是后缀。从图 3 的上半部可以看出普通 B*树索引在页块中存储了大量的重复键值，这就浪费

了大量的存储空间，而所谓的压缩键复合索引，则是把前缀中重复的键值移走，而只存储一份，这不仅能节约了大量的空间，而且还能一定程度上提升索引的性能。下图 3 给出了普通复合 B*树索引和压缩键复合 B*树索引的示意结构。

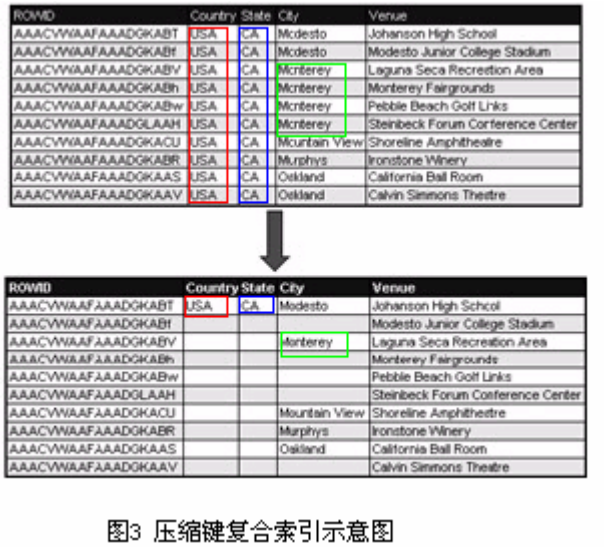

图3 压缩键复合索引示意图

压缩键索引与普通索引对比

下面就通过试验来对比压缩键复合索引与普通索引。

◆ 创建试验用表及普通索引

```
scott@ORCL>create table copress_tab    as
        select owner,object_type,object_id,status,created,object_name from all_objects;
      表已创建。
scott@ORCL> create index nocopress_idx on copress_tab(owner,object_type,object_id);
      索引已创建。
scott@ORCL> set autotrace traceonly
scott@ORCL> select owner,object_type,object_id from copress_tab;
已选择 50136 行。

执行计划
----------------------------------------------------------
Plan hash value: 588488804
-------------------------------------------------------------------------------------------------------
| Id  | Operation                    | Name              | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                   | 50507 |  2022K|    53   (4)| 00:00:01 |
|   1 |  INDEX FAST FULL SCAN| NOCOPRESS_IDX | 50507 |  2022K|    53   (4)| 00:00:01 |
-------------------------------------------------------------------------------------------------------

Note
-----
   - dynamic sampling used for this statement
```

统计信息

250	recursive calls
0	db block gets
3658	consistent gets
212	physical reads
0	redo size
918487	bytes sent via SQL*Net to client
37147	bytes received via SQL*Net from client
3344	SQL*Net roundtrips to/from client
5	sorts (memory)
0	sorts (disk)
50136	rows processed

◆ 分析索引，并查看 index_stats 表

```
scott@ORCL> analyze index nocopress_idx validate structure;
索引已分析
scott@ORCL> select 'nocopress' ,height,lf_blks,
  2         btree_space,opt_cmpr_count,opt_cmpr_pctsave
  3         from index_stats
  4  /
```

'NOCOPRES HEIGHT LF_BLKS BTREE_SPACE OPT_CMPR_COUNT OPT_CMPR_PCTSAVE

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

nocopress 2 212 **1704032** 2 47

◆ 使用 compress 1 子句创建压缩键索引。

```
scott@ORCL> drop index nocopress_idx;
索引已删除。
scott@ORCL> create index copress_idx on copress_tab(owner,object_type,object_id) compute statistics compress 1;
索引已创建。
scott@ORCL> select owner,object_type,object_id from copress_tab;
已选择 50136 行。
执行计划
----------------------------------------------------------
Plan hash value: 1842096743
--------------------------------------------------------------------------------------------
| Id  | Operation                     | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |             | 50507 |  2022K|     44   (5)| 00:00:01 |
```

```
|    1 |   INDEX FAST FULL SCAN| COPRESS_IDX | 50507 |    2022K|      44     (5)| 00:00:01 |
---------------------------------------------------------------------------------------------

Note
-----
    - dynamic sampling used for this statement

统计信息
----------------------------------------------------------
             75  recursive calls
              0  db block gets
           3606  consistent gets
            173  physical reads
              0  redo size
         918487  bytes sent via SQL*Net to client
          37147  bytes received via SQL*Net from client
           3344  SQL*Net roundtrips to/from client
              0  sorts (memory)
              0  sorts (disk)
          50136  rows processed

scott@ORCL> analyze index copress_idx validate structure;
    索引已分析
scott@ORCL> select 'compress1' ,height,lf_blks,
  2   btree_space,opt_cmpr_count,opt_cmpr_pctsave
  3   from index_stats
  4   /
'COMPRESS              HEIGHT           LF_BLKS    BTREE_SPACE    OPT_CMPR_COUNT
OPT_CMPR_PCTSAVE
--------- ---------- ---------- ----------- -------------- ----------------
compress1               2            173        1391340                2                      35
```

上面是使用了 compress 1 子句建立的压缩键索引，已经能够看出查询成本的从 53 降低到 44，大约降低了（53－44）/53×100％=17%。为了进一步进行比较，我们在进行 compress 2 和 Compress 3 的试验。

◆ 使用 compress 2 创建压缩键索引。

```
scott@ORCL> alter index copress_idx rebuild online compute statistics compress 2
  2   /
索引已更改。
scott@ORCL> select owner,object_type,object_id from copress_tab;

已选择 50136 行。

执行计划
----------------------------------------------------------
```

```
Plan hash value: 1842096743

-------------------------------------------------------------------------------------------
| Id  | Operation                    | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |               | 50507 |  2022K|     31   (7)| 00:00:01 |
|   1 |  INDEX FAST FULL SCAN| COPRESS_IDX | 50507 |  2022K|     31   (7)| 00:00:01 |
-------------------------------------------------------------------------------------------

Note
-----
   - dynamic sampling used for this statement

统计信息
----------------------------------------------------------
          89  recursive calls
           0  db block gets
        3552  consistent gets
         112  physical reads
           0  redo size
      918487  bytes sent via SQL*Net to client
       37147  bytes received via SQL*Net from client
        3344  SQL*Net roundtrips to/from client
           0  sorts (memory)
           0  sorts (disk)
       50136  rows processed

scott@ORCL> analyze index copress_idx validate structure;
索引已分析
scott@ORCL> select 'compress2' ,height,lf_blks,
  2          btree_space,opt_cmpr_count,opt_cmpr_pctsave
  3          from index_stats
  4  /

'COMPRESS              HEIGHT           LF_BLKS    BTREE_SPACE    OPT_CMPR_COUNT
OPT_CMPR_PCTSAVE
--------- ---------- ---------- ----------- -------------- ----------------
compress2               2            112          903584                  2                       0

◆   使用 compress 3 创建压缩键索引。
scott@ORCL> alter index copress_idx rebuild online compute statistics compress 3
  2  /
索引已更改。
scott@ORCL> select owner,object_type,object_id from copress_tab;
```

```
已选择 50136 行。

执行计划
----------------------------------------------------------
Plan hash value: 1842096743

--------------------------------------------------------------------------------
| Id  | Operation             | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |               | 50507 |  2022K|    62   (4)| 00:00:01 |
|   1 |  INDEX FAST FULL SCAN| COPRESS_IDX | 50507 |  2022K|    62   (4)| 00:00:01 |
--------------------------------------------------------------------------------

Note
-----
   - dynamic sampling used for this statement

统计信息
----------------------------------------------------------
          89  recursive calls
           0  db block gets
        3679  consistent gets
         253  physical reads
           0  redo size
      918487  bytes sent via SQL*Net to client
       37147  bytes received via SQL*Net from client
        3344  SQL*Net roundtrips to/from client
           0  sorts (memory)
           0  sorts (disk)
       50136  rows processed
scott@ORCL> analyze index copress_idx validate structure;

索引已分析

scott@ORCL> select 'compress3' ,height,lf_blks,
  2          btree_space,opt_cmpr_count,opt_cmpr_pctsave
  3          from index_stats
  4  /

'COMPRESS              HEIGHT           LF_BLKS    BTREE_SPACE    OPT_CMPR_COUNT
OPT_CMPR_PCTSAVE
--------- ---------- ---------- ----------- -------------- ----------------
compress3              2            253        2031020                2                 55
```

为了便于比较，把上面的各组数据放在下表 4 中：

表 4 各种索引下实验数据汇总表

类型	索引高度 HEIGHT	页块数 LF_BLKS	索引空间 BTREE_SPACE	优化压缩数 OPT_CMPR_COUNT	优化压缩空间节省 OPT_CMPR_PCTSAVE	查询成本
Nocopress	2	212	**1704032**	**2**	47	53
Compress 1	2	173	1391340	**2**	35	44
Compress 2	2	112	**903584**	**2**	0	31
Compress 3	2	253	2031020	**2**	55	62

可以看出 compress 1、compress 2、compress 3 索引的大小大约为非压缩索引的

compress 1：1391340/1704032×100% = 81.65%

compress 2：903584/1704032×100% = 53.03%

compress 3：2031020/1704032×100% =119.19%

使用 compress 2 创建的压缩索引几乎使索引空间缩小了一半。

注意上面试验中的 ANALYZE INDEX 语句，对非压缩索引应用该语句会填写 ORACLE 系统表 INDEX_STATS 中 的 OPT_CMPR_COUNT、 OPT_CMPR_PCTSAVE 字段，前者是 ORACLE 建议的优化压缩值，后者是预计的优化压缩能节约的空间百分比。表 4 中的第一行中 OPT_CMPR_COUNT、OPT_CMPR_PCTSAVE 值分别为 2 和 47%，即 ORACLE 建议的压缩数应该是 2(即 compress 2)，而且 ORACLE 预计如使用 compress 2 压缩，大概能节约 47%的空间。即用 compress 2 创建索引后，索引空间大小大致为：

1704032×（1－47%） ＝ **903136.96**

从表 4 第三行中可以看到，使用 compress 2 压缩后，索引空间（BTREE_SPACE）的实际大小为 **903584**，该值与 ANALYZE INDEX 分析的 903136.96 非常接近，可见 ORACLE ANYLYZE INDEX 命令推荐的索引压缩值和预计的索引空间压缩比是非常精确的。

从表 4 的最后一行，我们看到如果不遵从 ANALYZE INDEX 命令的建议，使用了 compress 3 来创建压缩索引，结果是过犹不及：不仅没有起到压缩效果，反而索引占用空间更大了。这是因为第三列并没有重复值可以压缩，第三列值是惟一的（即该索引的后缀）。结果是索引不仅没有压缩，ORACLE 还要额外为每个索引条目增加 4 字节的额外存储。粗略计算一下：

1704032 + 50136×4 = 1904576。 （50136 是表中的记录数）

该值与使用 compress 3 创建的压缩索引大小 2031020 非常接近。微小的差距应该是较大的索引使用了较多的索引数据块，从而需要更多的块头（Block Head）信息。

再看表 4 最后一列，它显示的是上面试验中的各查询成本信息。可以看出，compress 2 索引查询成本为 31，它仅为非压缩索引的

```
        31/53×100%=58.49%。
```

可见，压缩键索引能有效的提高查询的效率。这是因为压缩的索引在每个索引页块上能存储更多的索引数据，从而能有效提高缓冲区高速缓存的使用效率（注意，这里没有说提高缓冲区高速缓存命中率，因为高缓冲区高速缓存命中率并不代表高的使用效率）。

当然，压缩键索引的使用并不是‘免费’的，因为与普通索引相比，它是更加复杂的数据结构，所以 ORACLE 会花更多的时间来处理这个索引结构中的数据，不光在表数据修改期间维护索引时多消耗一些时间，而且查询期间搜索索引也会更花时间。之所以在上面的试验中，我们感觉不到这种查询时间的增长，一方面是因为我们试验用表的数据量太小（只有 50136 条记录），另一方面（也是更为重要地方面），压缩键索引提高了缓冲区高速缓存的使用效率，减少了 I/O，这在很大程度上节约了时间。

那么压缩键索引到底对 DML 操作的效率有多大的影响？下面我将再次通过试验来说明这个问题。

压缩键索引对 DML 语句性能的影响

◆ 丢掉上面试验中创建的压缩索引

```
scott@ORCL> drop index copress_idx;

索引已删除。
```

◆ 创建非压缩索引

```
scott@ORCL> create index nocompress_idx on
copress_tab(owner,object_type,object_id) compute statistics ;

索引已删除。
```

◆ 打开跟踪统计信息，执行 UPDATE 操作

```
scott@ORCL> set autotrace traceonly

scott@ORCL> UPDATE /*+ INDEX(COPRESS_TAB , NOCOMPRESS_IDX) */
  2   COPRESS_TAB
  3   SET STATUS='AAAAAAA',
  4         OBJECT_NAME=OBJECT_NAME ||'AAAAA' WHERE OBJECT_ID =51
  5   /

已更新 1 行。

执行计划
----------------------------------------------------------
```

```
Plan hash value: 2824110062

-----------------------------------------------------------------------------------------------------
| Id  | Operation                    | Name                | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------
|   0 | UPDATE STATEMENT             |                     |     5 |   175 |   223   (2)| 00:00:03 |
|   1 |  UPDATE                      | COPRESS_TAB         |       |       |            |          |
|*  2 |   INDEX FULL SCAN            | NOCOMPRESS_IDX      |     5 |   175 |   216   (2)| 00:00:03 |
-----------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("OBJECT_ID"=51)
       filter("OBJECT_ID"=51)

Note
-----
   - dynamic sampling used for this statement

统计信息
----------------------------------------------------------
        186  recursive calls
          2  db block gets
        341  consistent gets
        212  physical reads
        344  redo size
        924  bytes sent via SQL*Net to client
       1060  bytes received via SQL*Net from client
          6  SQL*Net roundtrips to/from client
          7  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

◆ 更改索引为压缩键索引（限于篇幅，只比较 compress 2 压缩键索引）

```
scott@ORCL> alter index nocompress_idx rename to compress_idx;
索引已更改。
scott@ORCL> alter index compress_idx rebuild online compute statistics compress 2;
索引已更改。
scott@ORCL> UPDATE /*+ INDEX(COPRESS_TAB , COMPRESS_IDX) */
  2  COPRESS_TAB
  3  SET STATUS='AAAAAAA',
  4        OBJECT_NAME=OBJECT_NAME ||'AAAAA' WHERE OBJECT_ID =51
  5  /
```

```
已更新 1 行。

执行计划
----------------------------------------------------------
Plan hash value: 835624928

-------------------------------------------------------------------------------------------------
| Id  | Operation                    | Name                | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | UPDATE STATEMENT             |                     |     5 |   175 |     123   (3)| 00:00:02 |
|   1 |  UPDATE                      | COPRESS_TAB         |       |       |            |          |
|*  2 |   INDEX FULL SCAN            | COMPRESS_IDX        |     5 |   175 |     116   (3)| 00:00:02 |
-------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("OBJECT_ID"=51)
       filter("OBJECT_ID"=51)

Note
-----
   - dynamic sampling used for this statement

统计信息
----------------------------------------------------------
         91  recursive calls
          4  db block gets
        207  consistent gets
        112  physical reads
        656  redo size
        923  bytes sent via SQL*Net to client
       1058  bytes received via SQL*Net from client
          6  SQL*Net roundtrips to/from client
          2  sorts (memory)
          0  sorts (disk)
          1  rows processed
scott@ORCL>
```

◆ INSERT 语句的执行情况对比（限于篇幅，略，但结果大致与 UPDATE 相同）

为了方便对比，我们把上面试验中的统计数据放在表格内，如下表 5 所示：

表 5 压缩键索引对 DML（UPDATE）语句的性能影响。

比较类型	逻辑 I/O	物理 I/O	索引成本	执行时间
普通索引	341	212	216	00:00:03
压缩索引	207	112	116	00:00:02

数据也许令很多人感到出乎意料，压缩键索引并没有大多数人想象中的那么‘糟糕’，相反，它的效率反而比普通索引更快。原因很明显：虽然压缩键索引增加了 CPU 的工作，但是却使减少了逻辑 I/O 和物理 I/O。在上面的试验中，使用了压缩索引的 UPDATE 语句的逻辑 I/O 缩减为普通索引的

207/341×100% ＝ 60.70%，

物理 I/O 缩减为普通索引的

112/212×100% ＝ 52.83%。

正是由于系统 I/O 的减少，才使总的索引成本反而降低，由普通索引的 216 降低为 116，这仅为普通索引成本的

116/216×100% ＝ 53.70%。

所以，试验中使用了压缩索引的 UPDATE 语句执行时间比普通索引执行时间更少也就不足为奇了。通过这个例子，就是想改变一下大多数人在潜意识里对压缩键索引的认识。现在，我们清楚的知道：压缩键索引不仅能起到节约存储空间的作用，而且还能有效提高 DML 操作的效率。但是，需要注意的是，**如果你的系统是一个 CPU 繁忙的系统，那么再使用压缩键索引只能适得其反**，这会减慢你的处理速度（前面曾经说过，压缩键索引的数据结构更加复杂，它会额外增加 CPU 的工作量）。如果你的系统 I/O 比较繁忙，那么使用压缩键索引会加快处理速度。所以，**压缩键使用与否一定要视你的系统实际情况而定，必要的话可以通过一定的测试来帮助你决策**。

压缩键索引小结

根据上面的试验，可以得出如下结论：

- ✓ 压缩索引使得索引变小，因而能加速数据的检索；
- ✓ 一定要考虑复合索引中各列的顺序，把重复值较多的列放在索引前部（前缀）；
- ✓ 压缩键索引的 DML 操作性能可能比非压缩索引‘坏’，但要视系统的实际情况而定，在 I/O 繁忙的系统，压缩键索引可能会给你带来意外的惊喜；
- ✓ 在并发性较高的系统中，压缩键索引会潜在的锁住更多的数据行，从而导致系统的 DML 性能较差，因此建议在高并发性的系统中不要使用压缩键索引。

‘倾斜’的 B*树索引

使用 B*树索引最经常碰到的问题恐怕就是 B*树经常变得‘倾斜’，如下图 4 所示。图中，大约 40%的记录集中在索引的第一个分支(Branch 1)。而分支 4（Branch 4）却只有 33141 个索引条目，大约仅为总索引条目数的 5%。因此，整个 B*树索引的结构变得非常不成比例（即‘倾斜’）。在这种情况，如果我们查找索引键值为 30001 的记录，将会有 40%的物

理空间被检索，而如果查找索引键值为 60001 的索引记录时，仅有约 5%的索引物理空间被检索。这就是所谓的倾斜的 B*树，因为它的某一个分支变得‘太重’了。

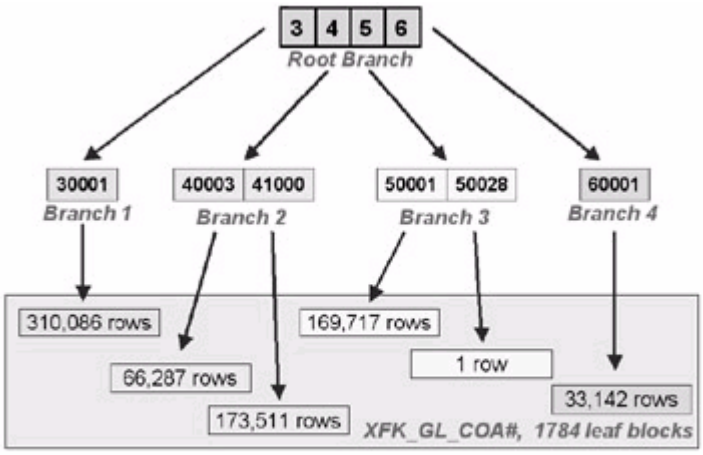

图4 ‘倾斜’B*树索引 示意图

‘倾斜’的的 B*树索引时它失去了 B*树索引原生的平衡性，导致高度‘倾斜’部分数据访问性能变差。因此作为 DBA，要经常观察你的 B*树索引，一旦发现有‘倾斜’现象，就必须进行必要的处理以使其重归平衡。处理‘倾斜’B*树索引的方法就是 ORACLE 提供的 ALTER INDEX 命令，通过 ALTER INDEX 的 REBUILD ONLINE 选项，ORACLE 不仅能重建一个非常‘均匀’的 B*树索引，而且重建过程中还不会中断当前的 DML 操作。

B*树索引空间‘丢失’

B*树索引空间的丢失与 ORACLE 的索引数据块物理存储机制有关。熟悉 ORALCE 物理存储的人都知道 PCTUSED 和 PCTFREE 两个参数，这是控制 ORALCE 数据块物理存储行为的两个重要参数。创建表时，如果使用了 PCTUSED 40 和 PCTFREE 10 参数，当数据块写满 90%时（即 10% FREE），ORACLE 就会把该块从可用自由列表（FREE LIST）中移出，不再往里面写东西，剩余的 10%为将来的 UPDATE 或者 INSERT 操作预留，这能有效防止‘臭名卓著’的行链（ROW CHAINING）的发生。如果由于大量的删除操作，使块中使用空间低于 40%（即 PCTUSED 40）时，ORACLE 又会把该数据块加入自由列表以重新接纳数据。但是，遗憾的是，PCTUSED 参数对于索引对象不适用，即 CREATE INDEX 语句中不能包含 PCTUSED 子句(**注意，ORACLE 官方文档《Oracle Database SQL Reference 10g Release 2》上有个错误，它在第 14 章 59 页上示意 CREATE INDEX 语句是可以包含 PCTUSED 子句的，但是，这是错误的，因为使用 PCTUSED 创建索引时 ORACLE 会报错**)。这意味着什么呢？答案是：除非你使用‘ALTER INDEX index-name REBUILD’命令重建索引，否则，删除的索引空间不被 ORACLE 回收（Reclaimed）再用。这就导致随着时间的推移，由于部分索引数据的删除，索引中会有一部分空间虽然不再被使用，但是这些空间也不再被 ORACLE 回收使用。形象地说，这些空间就‘丢失’了。如果索引中不断有大量地数据被删除，又不断地新增数据，那么长期以来，会导致索引使用的空间不断膨胀，从而导致 B*树索引的效率下降。

那么如何找回这些‘丢失’的空间呢？很简单，重建索引。有经验的 DBA 都会定期的重建数据库中使用的 B*树索引，其原因也就在于此。重建索引应该使用

```
ALTER INDEX index-name REBUILD ONLINE
```

命令。不推荐使用带有COALESCE 和DEALLOCATE UNUSED选项的ALTER INDEX 命令。因为这两个选项**可能**会使情况变得更糟糕。Gavin Powell 在一个咨询项目中曾经做过一次测试，结果发现使用带有COALESCE 和DEALLOCATE UNUSED选项的ALTER INDEX后，不仅没能回收‘丢失’的B*树索引存储空间，反而使索引空间增加了 25%。这里面有ORACLE深层次的原因，对于该原因的解释超出了本文的范畴，因此不在这里做进一步的说明。

注意：对于带有 COALESCE 和 DEALLOCATE UNUSED 子句的 ALTER INDEX 子句**可能**会使索引空间膨胀的问题，应该是暂时现象，我们可以在 ORACLE 每发布一个新版本时，对这个问题进行专题测试，直到发现 ORACLE 解决这个问题时再行使用。

何时使用 B*树索引

一般来说，在如下两种情况下，可以考虑使用 B*树索引：

- 通过索引仅访问表中的一条或者一小部分（占总记录条数的百分比很小）记录时；
- 如果要处理许多行，但仅通过使用索引即可完成处理（不需要访问表）时。

所有规则都有另外，不要盲目的相信所谓的‘经验’。这句话用在何时使用 B*树索引时尤其贴切。专家们只能提供一个一般规则，即一般情况下这样说是正确的。但在实际项目中，我们会碰到各种应用需求，有些应用的实现就有可能需要打破专家的‘魔咒’。比如，在一个交互式 Web 应用中，用户输入检索条件，需要检索一些行，然后再检索一些行，如此继续，最终他可能需要所有的行。按照前面的描述，该应用不应该使用 B*树索引，因为他返回了表的大部分甚至全部数据。但是，需要注意的时，这类 Web 交互应用需要的是快速在线查询响应速度，而不是最优的总体吞吐量。B*树索引具有较快的查询响应速度，因此它完全能满足这个 Web 应用的需求。通过这个应用示例，我主要想说明不要被‘经验、规则’性的东西所苑囿，不要盲从。

使用索引时，注意数据在磁盘上的物理组织对 B*树索引性能的影响。2002 年，IOUG-A 公司的 Wolfgang Bretling 在一篇论文中首次提出了索引成本的计算公式：

```
COST＝
        BLEVEL+
        Ceiling(LEAF_BLOCKS*EFFECTIVE INDEX SELECTIVITY)+
        Ceiling(clustering_factor* EFFECTIVE TABLE SELECTIVITY)
```

第一行就是前面提到的分支层级，它等于索引高度－1；

第二行表示的是遍历索引页块的的数目；

第三行表示的是访问表块的数目。

注：对公式的详细解释超出了本文范围，本文不在进一步阐述，有兴趣的的话，可以找到 Wolfgang Bretling 的论文研究一下。

在这个公式中，clustering_factor 取值大小非常关键，它是一个度量标准，该值的大小与数据的物理分布非常密切。一般说来，如果数据是跟索引一样进行‘有序’存储的，那么 clustering_factor 的取值就会很小，最小值等于表中的数据块数，此时根据索引全遍历数据时，正好访问了一遍所有的表块（表数据与索引数据存储顺序完美的一致）。clustering_factor 的最大值等于表中的行数，此时根据索引访问表数据时，索引每前进一步，都要发生表数据块的跳转，即相邻的值没有像索引那样有序的存储在一起，因此访问成本就较高。下面通过试验来详细了解一下表数据的物理存储对 B*树索引的影响。

◆ 创建有序物理存储表

```
scott@ORCL> create table t_order (x int,y varchar2(80));
表已创建。
```

```
scott@ORCL> begin
  2  for i in 1 .. 100000
  3  loop
  4       insert into t_order(x,y)
  5       values (i,rpad(dbms_random.random,75,'*'));
  6  end loop;
  7  end;
  8  /
PL/SQL 过程已成功完成。
scott@ORCL> alter table t_order
  2  add constraint t_order_pk
  3  primary key (x);
表已更改。
scott@ORCL> begin
  2  dbms_stats.gather_table_stats(user,'T_ORDER',cascade=>true);
  3  end;
  4  /
PL/SQL 过程已成功完成。
```

T_ORDER 表中，数据是按 X 列的升序物理存储的。

◆ 创建无序(随机)物理存储表

```
scott@ORCL> create table t_noorder
  2  as
  3  select x,y
  4  from t_order
  5  order by y
  6  /
表已创建。
scott@ORCL> alter table t_noorder
  2  add constraint t_noorder_pk
  3  primary key(x);
表已更改。
scott@ORCL> begin
  2  dbms_stats.gather_table_stats(user,'T_NOORDER',cascade=>true);
  3  end;
  4  /
PL/SQL 过程已成功完成。
scott@ORCL>
```

由于初始化 T_NOORDER 表时使用了 ORDER BY Y 子句，所以 T_NOORDER 表中数据不是以 X 的升序进行物理存储的，而是随机存储的，因为 Y 列是使用 ORACLE 随机函数生成的。

◆ 启用 TRACE 跟踪，对比两种表的性能

```
********************************************************************************
select /*+index (t_order, t_order_pk)*/ *
from
```

```
 t_order where x between 20000 and 40000

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.00       0.01          0          0          0           0
Fetch     1335      0.14       0.38         36       2900          0       20001
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total     1337      0.14       0.39         36       2900          0       20001
......
Rows     Row Source Operation
-------  ---------------------------------------------------
  20001  TABLE ACCESS BY INDEX ROWID T_ORDER (cr=2900 pr=36 pw=0 time=284690 us)
  20001   INDEX RANGE SCAN T_ORDER_PK (cr=1375 pr=0 pw=0 time=60061 us)(object id
54499)
......
********************************************************************************
********************************************************************************
select /*+index (t_noorder,t_noorder_pk)*/ *
from
 t_noorder where x between 20000 and 40000

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.01       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch     1335      0.30      13.35       1031      21356          0       20001
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total     1337      0.31      13.36       1031      21356          0       20001
......
Rows     Row Source Operation
-------  ---------------------------------------------------
  20001  TABLE ACCESS BY INDEX ROWID T_NOORDER (cr=21356 pr=1031 pw=0 time=29097686 us)
  20001   INDEX RANGE SCAN T_NOORDER_PK (cr=1375 pr=43 pw=0 time=118076 us)(object id
54501)
......
********************************************************************************
```

◆ 查看两表的 B*树索引信息

```
scott@ORCL> select a.index_name,b.num_rows,b.blocks,a.clustering_factor
  2  from user_indexes a,user_tables b
  3  where index_name in ('T_ORDER_PK','T_NOORDER_PK')
  4          and a.table_name=b.table_name
  5  /
```

```
INDEX_NAME           NUM_ROWS    BLOCKS CLUSTERING_FACTOR
-------------------- ---------- --------- -----------------
T_NOORDER_PK         100000          1219             99924
T_ORDER_PK           100000          1252              1190
scott@ORCL>
```

为了便于比较实验结果，表 6 对这些试验数据做了一个总结：

表 6 物理存储对 B*树索引的影响

表	CPU 时间 cpu	总耗时 elapsed	物理 I/O disk	逻辑 I/O query	CLUSTERING_FACTOR	索引数据块数
t_order	**0.14**	**0.39**	**36**	**2900**	**1190**	1252
t_noorder	**0.31**	**13.36**	**1031**	**21356**	**99924**	1219
有序/无序的百分比	**45.16%**	**2.92%**	**3.49%**	**13.58%**	N/A	N/A

简直难以置信，有序物理存储的的查询时间仅为无序物理存储查询时间的 2.92%，但这一切都在 I/O 数据列得到很好的解释。t_order 的物理和逻辑 I/O 分别仅为 t_noorder 的 3.49%和 13.58%。t_noorder 的物理 I/O 次数竟然达到惊人的 1031 次，而 t_order 的仅为 36 次。原因大致是这样的，因为 t_order 数据的物理存储顺序与索引是一致（有序）的，这时，当进行索引范围扫描时(INDEX RANGE SCAN)，定位一个索引条目后，根据索引条目中 ROWID 找到行数据，索引再前进一步时，对应索引条目的 ROWID 和前一条行记录是在同一个数据块上的，所以 ORACLE 不用再执行一次物理 I/O（因为上次物理 I/O 得到的数据块被缓存）跳转到其它数据块上去取行记录。对于 t_noorder 来说，恰恰相反，当索引再前进一步时，对应索引条目的 ROWID 是在另一个数据块上的，因此 ORACLE 不得不再进行一个物理 I/O 把该块读进来。为了说明 CLUSTERING_FACTOR，这个例子中特意把 t_noorder 所有数据都按 X 列随机杂乱无章的存储，所以索引 t_noorder_pk 索引每前进一步，几乎都会引起一个物理 I/O，这就导致了物理 I/O 的增加，从而影响了查询效率。

可见，表数据的物理存储会严重影响 B*树索引的行为，那么我们该怎么判断表数据是否是有序物理存储的呢？答案就在 CLUSTERING_FACTOR，从试验中可以更深刻的认识到该值实际上就是一个度量标准，用于索引的有序度和表的混乱度之间的比较。ORACLE Reference 手册上也是这样描述 CLUSTERING_FACTOR 含义的：

- 该值越小表示表数据的物理存储越有序，得到了很好的组织，在这种情况下，同一个索引页块中索引条目极可能指向同一个表数据块上的行，此时 B*树索引的性能越好；
- 该值越大表示表数据的物理存储越无序，这种情况下，同一个索引页块上的索引条目不太可能指向同一个表数据块上的行，此时 B*树索引的性能就较差。

如果在实际监控中，发现某索引的 CLUSTERING_FACTOR 较大，那么就意味着 DBA 就要采取行动了，此时 DBA 常犯的错误就是矫枉过正：试图重建所有表来使索引有一个较低的 CLUSTERING_FACTOR 值，但在大多数情况下，DBA 可能会发现他的这个做法原来只是白白浪费时间，因为性能并没有提升。为什么会这样呢？为了说明这个问题，下面再给出一个试验。

```
scott@ORCL> set autotrace traceonly
scott@ORCL> select * from t_order where x between 20000 and 30000;
已选择 10001 行。
```

执行计划

```
----------------------------------------------------------
Plan hash value: 3323386747
--------------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |               | 10007 |   781K|   143   (1)| 00:00:02 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T_ORDER       | 10007 |   781K|   143   (1)| 00:00:02 |
|*  2 |   INDEX RANGE SCAN                  | T_ORDER_PK | 10007 |       |    23   (5)| 00:00:01 |
--------------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("X">=20000 AND "X"<=30000)
```

统计信息

```
----------------------------------------------------------
        312  recursive calls
          0  db block gets
       1502  consistent gets
        183  physical reads
          0  redo size
     947045  bytes sent via SQL*Net to client
       7711  bytes received via SQL*Net from client
        668  SQL*Net roundtrips to/from client
          5  sorts (memory)
          0  sorts (disk)
      10001  rows processed
```

```
scott@ORCL> select * from t_noorder where x between 20000 and 30000;
已选择 10001 行。
```

执行计划

```
----------------------------------------------------------
Plan hash value: 1774554529
-----------------------------------------------------------------------------------------
| Id  | Operation                   | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |                   | 10004 |   781K|   275   (3)| 00:00:04 |
|*  1 |  TABLE ACCESS FULL| T_NOORDER | 10004 |   781K|   275   (3)| 00:00:04 |
-----------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter("X"<=30000 AND "X">=20000)
```

```
统计信息
----------------------------------------------------------
        312  recursive calls
          0  db block gets
       1903  consistent gets
       1197  physical reads
          0  redo size
     907098  bytes sent via SQL*Net to client
       7711  bytes received via SQL*Net from client
        668  SQL*Net roundtrips to/from client
          5  sorts (memory)
          0  sorts (disk)
      10001  rows processed
scott@ORCL>
```

试验中是从 100000 条记录中选取 10%（即 10000 条）的记录。此时 ORACLE 优化器对 t_order 使用了索引范围扫描 **INDEX RANGE SCAN**，而对 t_noorder 表则使用了全表范围扫描 **TABLE ACCESS FULL**。这说明 CLUSTERING_FACTOR 的取值影响了 ORACLE 优化器的选择，当 CLUSTERING_FACTOR 较大时，ORACLE 认为，通过索引选取 10%的记录的成本会比全表扫描还高，因此 ORACLE 选择了全表扫描。下面用试验证明 ORACLE 优化器的选择是非常正确的。下面试验中，我在查询中加入了提示/*+index (t_noorder,t_noorder_pk)*/，这样会强迫 ORACLE 优化器会选择 t_noorder_pk 索引来检索同样的 10%的数据。结果显示，使用了索引后，访问成本大幅增加了，从全表扫描的 **275** 增加到了 **10037** 次。所以，ORACLE 优化器的选择是非常正确的。

```
scott@ORCL> select /*+index (t_noorder,t_noorder_pk)*/ * from  t_noorder where x between 20000 and 30000;
已选择 10001 行。

执行计划
----------------------------------------------------------
Plan hash value: 3672977564
--------------------------------------------------------------------------------------------------------
| Id  | Operation                   | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |               | 10004 |   781K| 10037   (1)| 00:02:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T_NOORDER     | 10004 |   781K| 10037   (1)| 00:02:01 |
|*  2 |   INDEX RANGE SCAN          | T_NOORDER_PK  | 10004 |       |    23   (5)| 00:00:01 |
--------------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("X">=20000 AND "X"<=30000)

统计信息
```

```
-------------------------------------------------------
          1  recursive calls
          0  db block gets
      10680  consistent gets
          0  physical reads
          0  redo size
     947045  bytes sent via SQL*Net to client
       7711  bytes received via SQL*Net from client
        668  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
      10001  rows processed
scott@ORCL>
```

上面的例子说明在调整前一定要检查优化器的优化选择是否正确，如果优化器并没有错误选择你创建的 B*树索引，而且性能还算可以，那么 DBA 就没有必要重建表，因为重建表的代价实在太大了。限于篇幅问题，不再进行查询 25％以上记录的试验了，如果你做了，你会发现，t_order 和 t_noorder 都会不约而同的选择使用全表扫描 FULL TABLE SCAN。因为本章开始就说过：B*树索引适用于通过索引仅访问表中的一条或者一小部分（占总记录条数的百分比很小）记录的情况。如果你访问 25％（注意这不是阈值）以上的记录，这个比例已经不小，因此 ORACLE 优化器不会选择使用 B*树索引来检索数据了，哪怕这个 B*数索引的聚簇因子 CLUSTERING_FACTOR 再小。此时，DBA 重建表是没有任何意义的，只能是白白的浪费时间。

以上讨论的关键，**索引并不一定是合适的访问方法**，CBO（基于成本的优化器）也许选择不使用索引，如前面的例子，这种选择可能很正确。因此 DBA 在动手调整监控中发现的 CLUSTERING_FACTOR 较小的表时，一定要注意 CBO 的选择，如果 CBO 已经选择了正确的优化访问，你就没有必要去重建表。另外须记住，一般表上只有一个索引能有合适的聚簇因子 CLUSTERING_FACTOR。如果存在多个索引，其它索引上的聚簇因子取值必定会较大，这是因为表数据物理存储只有一种。因此在重建表时肯定是满足了表上的索引 2 却不再满足索引 1 了，你会顾此失彼的。总之，**一切以应用需求为前提，应用中对表数据的主要使用方式决定了 DBA 该如何处理表及其索引**，如果数据的物理聚簇确实很重要，那么可以考虑使用索引组织表（Index Organized Table）、B*树群集（B*Tree Cluster）或者考虑散列群集（Hash Cluster）等。

B*树索引小结

到目前为止，B*树索引是 ORACLE 中最常用的索引，也是我们在项目中最常用的索引结构。人们对 B*树索引的研究和探讨也最为深入，这从众多的 B*树索引变种就能看的出。B*树索引是绝好的通用的索引，在访问时间方面提供了很大的可扩展性，从 1000 和 1000000 条记录中返回一条数据的时间基本是一样的。

什么时候建立索引，在哪些列上建立索引，是模式设计中的必须注意的问题。索引并不一定就意味着更快的访问：在 CBO 占绝对主导地位的 ORACLE 时代，实际上你会发现，许多情况下，如果 ORACLE 使用了索引，反而会使性能下降。这实际上是两个因素的函数，其中一个是通过索引访问表中多少数据（占多大百分比），另一个因素就是数据的物理存储。如果完全使用索引就可以‘回答问题’（不用访问表），那么使用 B*树索引访问大量的行（占

很大百分比）就是有意义的，否则，可能就要确保你使用 B*树索引只是处理整个表中很少一部分数据（只占很小的百分比）。

最后，再强调一次：应该在应用的设计期间考虑索引的设计和实现，而不要事后才想起来（我们经常见到这种情况，不是吗？）用索引来‘救火’，如果在设计期间，对如何访问数据做了精心的计划和考虑，大多数情况下就能很清楚的知道需要什么索引。否则，如果在测试中或者投产后才发现很多索引没有合理创建或者根本没有创建，你将会付出惨重的多倍代价。

位图索引

位图索引简介

位图索引是为数据仓库环境/Ad-hoc 查询环境设计的。了解数据仓库的人都知道，Ad-hoc 查询就是即席查询，即查询是非常具有随意性的，再在系统实现是根本不知道用户要地会作些什么样的查询，因此你不能使用 B*树索引，因为 B*树索引都是专门某个查询进行优化的，它要求 Where 中的谓词以一定的顺序出现才能使用，但是即席查询恰恰满足不了这个要求（Where 中的列的顺序是随机的）。图 5 给出了在 STATE 列上创建位图索引的示意图。图中可以看出，ORACLE 使用一个位图来表示索引，如 STATE=‘CA’的索引条目（图中红色方筐内的部分）中，就是一张位图，位图中仅包含 0 和 1 取值，当某行的 STATE=‘CA’时，该行对应的位就是 1，否则就是 0。从图中可以看出，位图索引的一个索引条目存储指向多行的指针(STATE=‘CA’就指向前四行)，这与 B*树索引结构不同，在 B*树结构中，索引键和表中的行存在着对应关系。在位图索引中，只有很少的索引条目（图 5 中就只有 3 个索引条目），每个索引条目指向很多行，而在传统的 B*树索引中，一个索引条目就指向一行。

ROWID	VENUE	CITY	STATE	COUNTRY
AAAHfVAAJAAAK0KAAA	3com Park	San Francisco	CA	USA
AAAHfVAAJAAAK0KAAB	Agenda Lounge	San Jose	CA	USA
AAAHfVAAJAAAK0KAAC	Altamont Raceway Park	Tracy	CA	USA
AAAHfVAAJAAAK0KAAD	Amador Theatre	Pleasanton	CA	USA
AAAHfVAAJAAAK0KAAQ	Caesars	Lake Tahoe	NV	USA
AAAHfVAAJAAAK0KAAR	Caesars Palace	Las Vegas	NV	USA
AAAHfVAAJAAAK0LAAc	Madison Square Garden	New York	NY	USA
AAAHfVAAJAAAK0LAAb	Yankee Stadium	New York	NY	USA

ROWID	STATE='CA'	STATE='NY'	STATE='NV'
AAAHfVAAJAAAK0KAAA	1	0	0
AAAHfVAAJAAAK0KAAB	1	0	0
AAAHfVAAJAAAK0KAAC	1	0	0
AAAHfVAAJAAAK0KAAD	1	0	0
AAAHfVAAJAAAK0KAAQ	0	1	0

图5 位图索引示意图

使用位图索引时，需要记住如下几个要点：

- ✓ 一般说来，位图对于低基数的单列（single-column indexes with low cardinality）索引比较有效。低基数列是指该列仅有少数不同的值。比如，表示性别、国家、年龄段、联行号、货币、科目等的字段，都是低基数字段的的典型代表。这里需要注意对‘低基’的判断。这是一个相对值，不是绝对值，对于只有 2000 条记录的表来说，2 就是低基数，但是对于有 20000000 条的记录的表来说，20000 也可以认为是低基数，此时在这个具有 20000 个不同值的列上也可以建立位图索引。
- ✓ 位图索引在‘只读’（Read-only）或者以读为主的环境下会比 B*树索引快。‘只读’是数据仓库/MIS/BI 等应用的典型特征。因此 OLTP 系统不适合位图索引，因为 OLTP 系统中的数据通常会由多个频繁的更新。
- ✓ 位图索引最大的性能问题是它的索引机制。它不像 B*树索引那样使用行级锁（Row Level Lock），而是使用块级锁。与 B*树索引（更改操作仅锁定一行）不同的是，如果一个会话更新了位图索引所索引的数据，那么 ORACLE 会把该位图索引条目指向所有的行都锁定。ORACLE 无法锁定一个位图索引条目中的一位；而是会锁定整个位图索引条目。倘若有其它会话也要更改同样的这个索引条目指向的行，就会被‘关在门外’。这样将大大影响 UPDATE 和 DELETE 操作的并发性，因为 DELETE 和 UPDATE 都需要排它锁（Exclusive Locks）。
- ✓ 位图索引不应该被用来创建复合索引。复合索引包含多列，它们的组合一般来说已经是高基数（High Cardinality）的了，不具备使用位图索引的条件。如果应用需要创建那样的一个‘复合索引’，那么可以用多个单列位图索引来代替。在 ORACLE 中，SQL 的 WHERE 子句可以匹配多个单列位图索引（不管这些列在 WHERE 子句中出现的顺序如何）。
- ✓ 不像 B*树索引那样，位图索引可能会溢出（Overflow）。因此它经常应用数据仓库中数据不变化的表中，对于数据仓库中经常要要完全重载（Complete Reload）的表不适应。比如数据仓库的 ODS 层，或者 BUFFER 层的表都不适合用位图索引，而在面向最终用户展现的 DW 层或者我们公司很多项目中提到的 PI 层都有位图索引的用武之地。

注意：传统地，除了 B*树索引之外地任何索引类型都可能会溢出。溢出时会形成‘链’（Chain），这些‘链’连接不能把索引值放入事先预留的索引空间的多个数据块。在谈论表的物理存储时，我们也碰到过表‘链’的问题，它们的危害是一样的，都是增加了 SQL 语句的 I/O 次数，因此对系统的性能非常有害。

和 B*树索引一样，位图索引可能也需要定期的进行重建（Rebuild），然而，对于数据仓库里的大型的数据库表，重建位图索引消耗的大量时间可能会令无法接受。因此在强调一下：位图索引最适合数据仓库/MIS/BI 应用中的大型的、低基数的、只读的‘静态’（不经常变动）表。

下面通过试验，对 DBA 和应用开发人员比较感兴趣的几个问题进行论证。

位图索引比 B*树索引快吗

位图索引比 B*树索引块吗？这可能是很多 DBA 和开发人员比较感兴趣的一个问题，下面就通过试验来回答这个问题。

◆ 建立试验环境

```
scott@ORCL> create table t_bitmap as select * from big_table where owner in ('XDB','ORDSYS',
'PUBLIC');
表已创建。
scott@ORCL> select owner,count(*) from t_bitmap group by owner;
OWNER                          COUNT(*)
------------------------------ ----------
PUBLIC                             900369
ORDSYS                              73350
XDB                                  9585
scott@ORCL> create bitmap index bitmap_idx on t_bitmap(owner);
      索引已创建。
scott@ORCL> create table t_btree as select * from big_table where owner in ('XDB','PUBLIC',);
      表已创建。
scott@ORCL> create index btree_idx on t_btree(owner);
      索引已创建。
```

◆ 分析试验用表

```
scott@ORCL> analyze table t_btree compute statistics;
表已分析。
scott@ORCL> analyze table t_bitmap compute statistics;
表已分析。
```

◆ 分别对两表运行同一个查询，对比性能

```
scott@ORCL> select /*+ INDEX(t_btree,btree_idx) */ * from t_btree
  2   where owner='ORDSYS'
  3   /

已选择 73350 行。

执行计划
----------------------------------------------------------
Plan hash value: 3587339131

-------------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name       | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |            | 41468 |  3766K|    8619   (2)| 00:01:44 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T_BTREE      | 41468 |  3766K|    8619   (2)| 00:01:44 |
|*  2 |   INDEX RANGE SCAN                  | BTREE_IDX | 41468 |       |    1259   (3)| 00:00:16 |
-------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("OWNER"='ORDSYS')
```

```
统计信息
----------------------------------------------------------
          1  recursive calls
          0  db block gets
      11041  consistent gets
        185  physical reads
          0  redo size
    8581851  bytes sent via SQL*Net to client
      54164  bytes received via SQL*Net from client
       4891  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
      73350  rows processed
scott@ORCL> select /*+ INDEX(t_bitmap,bitmap_idx) */ * from t_bitmap
  2  where owner='ORDSYS'
  3  /

已选择 73350 行。

执行计划
----------------------------------------------------------
Plan hash value: 3973342777

-------------------------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name         | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |              | 41468 |  3766K|  1686   (1)| 00:00:21 |
|   1 |  TABLE ACCESS BY INDEX ROWID        | T_BITMAP     | 41468 |  3766K|  1686   (1)| 00:00:21 |
|   2 |   BITMAP CONVERSION TO ROWIDS       |              |       |       |            |          |
|*  3 |    BITMAP INDEX SINGLE VALUE        | BITMAP_IDX   |       |       |            |          |
-------------------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - access("OWNER"='ORDSYS')

统计信息
----------------------------------------------------------
          1  recursive calls
          0  db block gets
       5971  consistent gets
          3  physical reads
          0  redo size
```

```
8581851  bytes sent via SQL*Net to client
   54164  bytes received via SQL*Net from client
    4891  SQL*Net roundtrips to/from client
       0  sorts (memory)
       0  sorts (disk)
   73350  rows processed
```

为了便于比较，把试验数据统计在表格内，如下表 7 所示：

表 7 位图索引与 B*树索引性能对比

表类型	逻辑 I/O Consistent gets	物理 I/O Physical reads	成本 Cost (%CPU)	时间 Time
t_btree	**11041**	**185**	**8619**	**00:01:44**
t_bitmap	**5971**	**3**	**1686**	**00:00:21**

数据说明了速度这个问题，同样的查询，t_btree 使用的时间是 **00:01:44**，而 t_bitmap 仅仅使用了 **00:00:21** 。这是因为 t_bitmap 仅仅进行了 3 次物理 I/O，逻辑 I/O 也只有 5971 次，这分别是 t_btree 的 1.62%和 54.98%。（如果你查看一下索引尺寸的话，会发现位图索引的尺寸比 B*树索引小很多，它只需较少的物理数据块来存储，所以在上面的试验中，t_bitmap 的物理 I/O 较少，这也是位图索引速度较快的一个因素。）

这个试验其实很不公平，其实，B*树索引和 BITMAP 索引根本不存在可比性，因为二者的适用领域不同。这个试验中的索引是建立在低基数列（OWNER）上的，这是 BITMAP 索引的长处，以 BITMAP 之长攻 B*树索引之短，所以这个试验是不公平的。因此，在谈到 BITMAP 和 B*树索引的性能时，一定要加上前提环境。正确的说法前面也已经提到过：**位图索引在‘只读’（Read-only）或者以读为主的低基数环境下会比 B*树索引快**。

何时使用位图索引

前面其实大致已经对‘何时使用位图索引’主题做了说明。本节将进一步详细阐述。数据仓库的 Ad-hoc 查询中，经常会碰到 COUNT 之类的聚合，在这样的环境中，位图索引就特别有用。Ad-hoc 查询的另外一个特征就是 WHERE 语句的里通常存在非常多的逻辑运算，即 AND 或者 OR 操作，这也可以从位图索引中获益。下图 6、图 7、图 8 完整了展示了 ORACLE 中位图索引的存储和使用，通过这个套图基本可以明白：为什么位图索引对复杂的 WHERE 语句中的逻辑运算比较高效。

假如有如下一张表：

图 6 位图索引示例表

CUSTOMER #	MARITAL_ STATUS	REGION	GENDER	INCOME_ LEVEL
101	single	east	male	bracket_1
102	married	central	female	bracket_4
103	married	west	female	bracket_2
104	divorced	west	male	bracket_4
105	single	central	female	bracket_2
106	married	central	female	bracket_3

图6 位图索引示例表

在 REGION 列上建立 BITMAP 索引，图 7 是它的位图索引结构示意图，它包含三个独立的位图，即每个 REGION 取值对应一个位图。

另外，在 MARITAL_STATU、GENDER 和 INCOME_LEVEL 上也分别建立位图索引。

REGION='east'	REGION='central'	REGION='west'
1	0	0
0	1	0
0	0	1
0	0	1
0	1	0
0	1	0

图7 位图索引结构示例

在数据仓库环境中，前端用户会经常发出如下的分析类查询：我们有多少已婚的且住在
中部或者西部的客户？

```
SELECT COUNT(*) FROM CUSTOMER
WHERE MARITAL_STATUS = 'married' AND REGION IN ('central','west');
```

位图索引可以非常高效的处理这种查询，而且访问位图索引即可完成这个查询，不用访问具体的表，图 8 展示了 ORACLE 是如何通过位图索引实现这个查询的。

图 8 位图索引查询原理

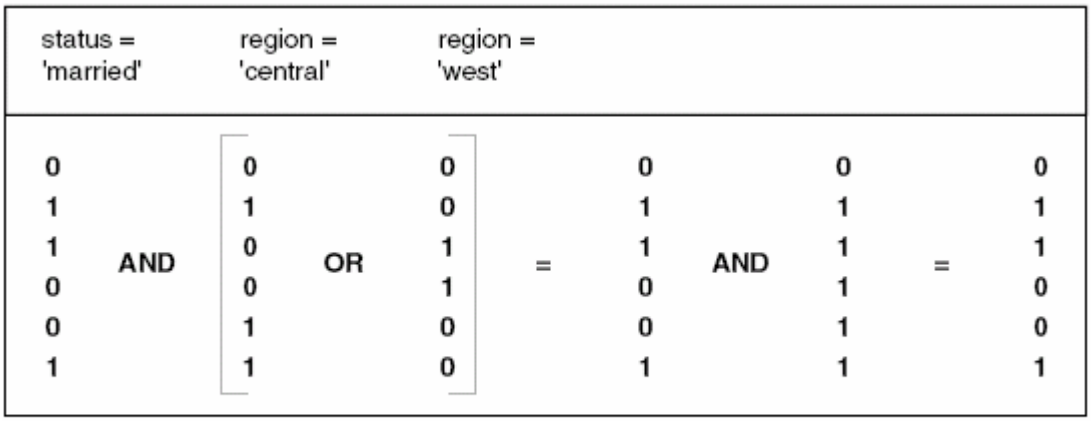

图8 位图索引查询原理

可以看到，ORACLE 是现通过位图进行逻辑运算，然后得到一个满足逻辑条件的位图。上面的查询只是进行一个简单的计数，因此，ORACLE 根本没有必要去访问表，直接数一下位图中的‘1’的个数即可。如果查询的不是简单的 COUNT 信息，而是需要访问具体表中的行，那么 ORACLE 可以直接把这个位图中的位转换成 ROWID，根据该 ROWID 即可访问所需的行数据。

下面具体创建一个数据量稍大的试验用表 TEST_BITMAP。其中有 3 列：GENDER、LOCATION 和 AGE_GROUP。在表中，GENDER 列表示性别，它只有两个取值：M 或 F。LOCATION 表示地区，它的可取值为 1～50。AGE_GROUP 表示年龄段，它的取值范围分别为 18 and under 、19－25、26－30、31－40、41 and over。然后我们会对该表发出如下形式的较为复杂的查询，以再次考察和领略位图索引的性能。

```
Select count(*) from test_bitmap
Where gender='M'
  And location in (1,10,30)
  And age_group='41 and orver';

Select * from test_bitmap
Where   (      (gender='M' and location =20)
            Or (gender='F'   and location=22))
      And   age_group = '18 and under';
```

你会发现，通过传统的 B*树索引机制根本行不通。因为，WHERE 条件中的各列出现的顺序是随机的，没有规律可循，因此，如果想使用 B*树索引，只能需要组合最多多个 B*树索引来优化该查询，但是通过使用位图索引，你会发现一切变得那么简单。请看下面试验。

◆ 创建实验用表

```
scott@ORCL> create table test_bitmap
  2       (gender not null,
  3       location not null,
  4        age_group not null,
  5        data
  6        )
```

```
  7         as
  8         select decode(ceil(dbms_random.value(0,2)),
  9                                 1,'M',
 10                                 2,'F') gender,
 11                     ceil(dbms_random.value(1,50)) location,
 12                 decode(ceil(dbms_random.value(1,5)),
 13                                 1,'18 and under',
 14                                 2,'19-25',
 15                                 3,'26-30',
 16                                 4,'31-40',
 17                                 5,'41 and over') age_group,
 18                     rpad('*',20,'*') data
 19         from big_table
 20         where rownum <=300000;

表已创建。
```

◆ 分别在 GENDER、LOCATION、AGE_GROUP 三个列上创建索引

```
scott@ORCL> create bitmap index gender_bmap_idx on test_bitmap(gender);

索引已创建。

scott@ORCL> create bitmap index location_bmap_idx on test_bitmap(location);

索引已创建。

scott@ORCL> create bitmap index age_group_bmap_idx on test_bitmap(age_group);

索引已创建。
```

◆ 收集表 TEST_BITMAP 的统计信息

```
scott@ORCL> exec dbms_stats.gather_table_stats(user,'TEST_BITMAP',cascade=>true);

PL/SQL 过程已成功完成。
```

◆ 查看各查询下的执行计划

```
scott@ORCL> Select count(*) from test_bitmap
  2         Where gender='M'
  3           And location in (1,10,30)
  4           And age_group='41 and orver'
  5   /

执行计划
----------------------------------------------------------
Plan hash value: 3303620568

-------------------------------------------------------------------------------------------------
```

```
| Id  | Operation                          | Name                  | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                   |                       |     1 |    13 |    20   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE                    |                       |     1 |    13 |            |          |
|   2 |   BITMAP CONVERSION COUNT          |                       |  2316 | 30108 |    20   (0)| 00:00:01 |
|   3 |    BITMAP AND                      |                       |       |       |            |          |
|   4 |     BITMAP OR                      |                       |       |       |            |          |
|*  5 |      BITMAP INDEX SINGLE VALUE     | LOCATION_BMAP_IDX     |       |       |            |          |
|*  6 |      BITMAP INDEX SINGLE VALUE     | LOCATION_BMAP_IDX     |       |       |            |          |
|*  7 |      BITMAP INDEX SINGLE VALUE     | LOCATION_BMAP_IDX     |       |       |            |          |
|*  8 |     BITMAP INDEX SINGLE VALUE      | AGE_GROUP_BMAP_IDX    |       |       |            |          |
|*  9 |     BITMAP INDEX SINGLE VALUE      | GENDER_BMAP_IDX       |       |       |            |          |
-------------------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   5 - access("LOCATION"=1)
   6 - access("LOCATION"=10)
   7 - access("LOCATION"=30)
   8 - access("AGE_GROUP"='41 and orver')
   9 - access("GENDER"='M')
```

这个查询足以展示位图索引的强大能力。ORACLE 看到了 LOCATION IN(1,10,30)条件，知道要读取这 3 个位置上的索引，并在这三个位置对应的位图中对‘位’执行逻辑的 OR 操作。然后得到位图在与 AGE_GROUP ＝‘41 and orver’ AND GENDER＝‘M’的相应位图执行逻辑的 AND 操作，的到一个最终位图，然后再建简单统计一下最终位图中‘1’的个数，就得到了答案，整个查询处理没有涉及到表数据的访问。注意一下查询条件的 AND 和 OR 操作是如何转化为位图的 AND 和 OR 运算的（图中我以蓝色和红色做了高亮加粗显示）。

```
scott@ORCL> Select * from test_bitmap
  2          Where    (       (gender='M'    and location =20)
  3                        Or (gender='F'    and location=22)
  4                     )
  5                  And    age_group = '18 and under'
  6   /

执行计划
----------------------------------------------------------
Plan hash value: 2945505798

-------------------------------------------------------------------------------------------------
| Id  | Operation                   | Name                  | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |                       |  1555 | 51315 |   235   (1)| 00:00:03 |
|   1 |  TABLE ACCESS BY INDEX ROWID| TEST_BITMAP           |  1555 | 51315 |   235   (1)| 00:00:03 |
```

```
|   2 |   BITMAP CONVERSION TO ROWIDS |                          |      |      |            |           |
|   3 |    BITMAP AND                 |                          |      |      |            |           |
|*  4 |     BITMAP INDEX SINGLE VALUE | AGE_GROUP_BMAP_IDX       |      |      |            |           |
|   5 |     BITMAP OR                 |                          |      |      |            |           |
|   6 |      BITMAP AND               |                          |      |      |            |           |
|*  7 |       BITMAP INDEX SINGLE VALUE| LOCATION_BMAP_IDX       |      |      |            |           |
|*  8 |       BITMAP INDEX SINGLE VALUE| GENDER_BMAP_IDX         |      |      |            |           |
|   9 |      BITMAP AND               |                          |      |      |            |           |
|* 10 |       BITMAP INDEX SINGLE VALUE| LOCATION_BMAP_IDX       |      |      |            |           |
|* 11 |       BITMAP INDEX SINGLE VALUE| GENDER_BMAP_IDX         |      |      |            |           |
-------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   4 - access("AGE_GROUP"='18 and under')
   7 - access("LOCATION"=22)
   8 - access("GENDER"='F')
  10 - access("LOCATION"=20)
  11 - access("GENDER"='M')
```

这是一个更加复杂的查询，但是 ORACLE 处理是一样的。计划显示：括号中执行 OR 的两个条件是通过适当的位图的 AND 操作来分别得到，然后再对这些结果执行逻辑 OR 得到一个位图。这个位图再与 年龄段上的 AGE_GROUP＝‘18 and under’位图条目进行 AND，这就得到了最后的位图，位图中的位通过 ORACLE 的 BITMAP CONVERSION TO ROWIDS 操作就直接得到了相应的 ROWID（这步转换非常高效）。在体会一下查询条件的 AND 和 OR 操作是如何转化为位图的 AND 和 OR 运算的（图中我以多种颜色做了高亮加粗显示）。

位图索引还有一个好处：WHERE 条件中各列的无论以什么顺序出现，都不影响 ORACLE 优化器选择使用位图索引（这一点是 B*树索引无法实现的）。限于篇幅，不在这里去试验演示了，有兴趣的话，你可以把上面查询中的条件颠倒一下顺序，看看 ORACLE 优化器是否继续使用位图索引。

综上，何时使用位图索引基本上有了一个答案：在一个数据仓库/MIS/BI 等 OLAP 型应用系统中，合理的使用尽可能多的位图索引是非常有用的，但是在写密集型的 OLTP 型应用中，位图索引是极不适用的。

位图索引小结

如果你还犹豫不决，你可以亲自试一试。向一个表增加位图索引很容易，你可以自己看看位图索引做了些什么，体验一下它的性能。另外，创建位图索引通常比 B*树索引块的多，因为它通常比 B*树索引小很多（索引使用数据块少）。

下面再次重复一下文中提到的一些主要观点以结束本节。

- ✓ ‘低基数’是个相对值，它没有一个量化的大小
- ✓ 位图索引一定要应用在适宜使用的环境中，否则，会导致严重的性能和扩展性问题。如果你实在不知道位图索引是否适合于你的环境，最好的办法就是做试验。

让事实来说话。

- ✓ 尽量不要使用复合位图索引，你可以用多个单列位图索引来代替。（限于篇幅，本文没有对复合位图索引和等价的多个单列索引性能做对比，有兴趣的话，可以自己动手去做，你会发现复合位图索引的性能较差。）

位图联接索引

位图联接索引（Bitmap Join Index）是 ORACLE 9i 中的引入的一个新的索引类型。它本质上是一种位图索引。通常的索引都是使用表本身的列来创建，但是位图联接索引打破了限制，它允许通过联接使用表 B 的列对表 A 建立索引。它的本质是仍然是位图索引，从关系型数据库系统来说，位图联接索引是一种逆规范化的手段。

考虑简单的 SCOTT 用户下的 EMP 表和 DEPT 表，EMP 表由一个外键（DEPTNO）指向 DEPT 表。熟悉数据仓库技术的人都知道，业务人员一般都是在语义层进行查询的。例如他们一般会这样查询：

销售部门有多少人？

谁在市场部门工作？

销售部门中业绩最好的前 N 个人是谁？

他们不会发出这样的查询：

DEPTNO 等于 30 的部门有多少人？

谁在 50 号部门工作？

DEPTNO 等于 30 的部门中，销售业绩最好的前 N 个人是谁？

在数据仓库/MIS/BI 类应用系统中，提供一个使用业务术语的语义层给最终业务用户进行查询是设计和开发人员要实现的一个基本功能。

考虑如下两个查询：

```
Select   count（*）
  From   emp,dept
Where emp.deptno=dept.deptno
And   dept.dname=’SALSE’
/
Select   emp.*
From   emp,dept
Where emp.deptno=dept.deptno
And   dept.dname=’SALSE’
/
```

使用传统的 B*树索引，表 DEPT 和 EMP 都必须被访问才能完成查询，ORACLE 会先根据 DEPT.DNAME 上的索引来查找 SALSE 的数据行，然后使用 EMP.DEPTNO 上的索引来检索匹配的行。但是，如果使用位图联接索引，就不需要这些了。利用位图联接索引，ORACLE 能对表 EMP 建立基于 DEPT.DNAME 列索引，这个索引是指向 EMP 表的，可以这么理解，就相当于在 EMP 上有一个 DNAME 伪例（就像 ROWNUM 一样），当然这一列是根据 emp.deptno=dept.deptno 关联来得到的。这是一个全新的概念，一方面可以保持规范化的数据结构不变，同时通过索引得到了逆范式化的好处。下面就建立这种神奇的位图联接索引。

```
scott@ORCL> create bitmap index emp_bmjoin_dept_idx
  2   on emp(d.dname)
```

```
  3  from emp e,dept d
  4  where e.deptno=d.deptno
  5  /
索引已创建。
scott@ORCL>
```

这个语句看上去和普通的位图索引没有什么两样，但是它却是基于 e.deptno=d.deptno 联接，用 d.dname 列在 EMP 上建立了索引。下面就通过试验来看看位图联接索引在 ORACLE 查询中的应用。由于 DEPT 和 EMP 表都比较小，所以 ORACLE 优化器不会在查询中使用这个位图联接索引，因此，我们需要‘欺骗’ORACLE 优化器，使它觉得 EMP 和 DEPT 表看上去很大。使用 DBMS_STATS 程序包的 SET_TABLE_STATS 函数可以实现这个目的。

```
scott@ORCL> begin
  2  dbms_stats.set_table_stats(user,'EMP',numrows=>1000000,numblks=>300000);
  3  dbms_stats.set_table_stats(user,'DEPT',numrows=>100000,numblks=>30000);
  4  end;
  5  /
PL/SQL 过程已成功完成。
scott@ORCL>
```

现在，发起这个查询：

```
scott@ORCL> select count(*)
  2  from emp,dept
  3  where emp.deptno=dept.deptno
  4  and dept.dname='SALSE'
  5  /

执行计划
----------------------------------------------------------
Plan hash value: 3001258335


-------------------------------------------------------------------------------------------------------------
| Id  | Operation                          | Name                 | Rows  | Bytes | Cost (%CPU )| Time     |
-------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                   |                      |     1 |     3 |     1   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE                    |                      |     1 |     3 |            |          |
|   2 |   BITMAP CONVERSION COUNT          |                      |   333K|   976K|     1   (0)| 00:00:01 |
|*  3 |    BITMAP INDEX SINGLE VALUE | EMP_BMJOIN_DEPT_IDX |       |       |            |          |
-------------------------------------------------------------------------------------------------------------


Predicate Information (identified by operation id):
---------------------------------------------------
   3 - access("EMP"."SYS_NC00009$"='SALSE')
scott@ORCL>
```

可以看到，完成这个特定查询根本，ORACLE 根本没有真正的去访问那 EMP 和 DEPT 表，答案全部来自索引 EMP_BMJOIN_DEPT_IDX 本身。回答这个问题所需的全部信息都能在索引结构中找到。

下面，再发起另外一个查询，注意 ORACLE 是如何避免访问 DEPT 表，仅使用位图联接索引 EMP_BMJOIN_DEPT_IDX 和 EMP 表就能合并我们需要的查询数据的。

```
scott@ORCL> select /*+ index(e,EMP_BMJOIN_DEPT_IDX)*/ e.*
  2  from emp e,dept d
  3  where e.deptno=d.deptno
  4  and d.dname='SALES'
  5  /
执行计划
----------------------------------------------------------
Plan hash value: 3332705302
-------------------------------------------------------------------------------------------------------------
| Id  | Operation                             | Name                 | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                      |                      |   333K|    11M|   268K  (1)| 00:53:48 |
|   1 |  TABLE ACCESS BY INDEX ROWID          | EMP                  |   333K|    11M|   268K  (1)| 00:53:48 |
|   2 |   BITMAP CONVERSION TO ROWIDS         |                      |       |       |            |          |
|*  3 |    BITMAP INDEX SINGLE VALUE          | EMP_BMJOIN_DEPT_IDX  |       |       |            |          |
-------------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   3 - access("E"."SYS_NC00009$"='SALES')
scott@ORCL>
```

如果试验中，ORACLE 没有使用 EMP_BMJOIN_DEPT_IDX 索引，可以如上所示加入 SQL 提示/*+ index(e,EMP_BMJOIN_DEPT_IDX)/来强迫 ORACLE 使用之，以观察位图连接索引的使用。从试验可以看出，实现这个特定的查询，ORACLE 并没有物理访问 DEPT 表。

位图联接索引的使用有一个前提条件：联接条件必须联接到另一个表的主键或者惟一键，前面的例子中，DEPT.DEPTNO 就是 DEPT 上的主键，而且这个主键必须合适，否则就会报如下错误：

```
ORA-25954：missing primary key or unique constraint on dimension
```

总之，位图连接索引是一种新型位图索引，它的使用领域虽然不宽，但是却极具技巧性。如果使用得当，它会使应用的局部性能得到提升。

基于函数的索引

基于函数索引简介

基于函数的索引（Function Based Index）是ORACLE8.1.5中增加的一种索引，现在已经是标准ORACLE版本的一个特性了。在ORACLE 9iR2之前，这还是ORACLE企业版的一个特性。

利用基于函数的索引（下面简称FBI），你可在表字段上创建基于函数或者表达式的索引。FBI能有效提高WHERE 子句中含有函数的SQL语句的性能。它计算函数或者表达式的

值，并把得到的值存储在索引中。然而，当ORACLE在表上处理INSERT或者UPDATE语句的时候，它需要在维护FBI时额外增加了函数计算步骤，因此可能会造成些微的性能下降。在ORACLE中，你可以创建一个B*树FBI或者位图FBI。

创建FBI使用的函数可以是一个算术表达式或者式一个包含PL/SQL、包函数、C调用或者SQL函数的表达式。这个表达式必须是一个‘确定性’的(Deterministic)，即对于给定的相同的输入，该表达式返回的结果总是确定性的。可以在创建函数时使用‘Deterministic’关键字告诉ORACLE：可以相信这个函数，给定相同的输入，不论做多少次函数调用，它肯定返回相同的值。当然，你要确保该函数确实时确定性的（不能在这里欺骗ORACLE），否则，使用这个‘FBI’就会得到与使用全表扫描不同的结果。

ORACLE 不能在LOB、REF或者嵌套表（Nested Table）列上创建FBI，在包含有如上类型的对象类型（Object Type）列上也不能创建FBI。

下面是一个 FBI的例子：

```
CREATE INDEX idx ON table_1 (a + b * (c - 1), a, b);
```

创建上面的索引后，然后ORACLE就可以使用该索引处理如下的查询：

```
SELECT a FROM table_1 WHERE a + b * (c - 1) < 100;
```

基于函数索引的使用

谈到FBI的应用，很多人都会想起它的典型应用：实现与大小写无关的查询。这的确是FBI的一个典型应用，但这绝对不是FBI的全部，在实际应用中，开发人员发现了FBI正变得越来越有用，下面就通过示例来说明FBI的这些应用。

FBI 实现大小写无关查询

FBI 实现大小写无关（Case Insensitive）查询是 FBI 的经典应用，下面举个简单例子演示一下。

◆ 创建示例用表

```
scott@ORCL> create table test_ci
  2  as
  3  select *
  4    from emp
  5  where 1=0
  6  /
表已创建。
scott@ORCL> insert into test_ci(empno,ename,job,mgr,hiredate,sal,comm,deptno)
  2  select rownum empno,
  3         initcap(substr(object_name,1,10)) ename,
  4         substr(object_type,1,9) job,
  5         rownum mgr,
  6         created hiredate,
  7         rownum sal,
  8         rownum comm,
  9         (mod(rownum,4)+1)*10 deptno
```

```
  10   from all_objects
  11   where rownum <10000
  12   /
已创建 9999 行。
scott@ORCL>
```

◆ 在 ENAME 列上创建 UPPER(ENAME)函数索引，然后分析表

```
scott@ORCL> create index upper_idx on test_ci(upper(ename));
索引已创建。
scott@ORCL> begin
  2   dbms_stats.gather_table_stats(user,TEST_CI',cascade=>true);
  3   end;
  4   /
PL/SQL 过程已成功完成。
scott@ORCL>
```

◆ 执行大小写不敏感的查询

```
scott@ORCL> select *
  2   from test_ci
  3   where upper(ename) = 'KING'
  4   /

执行计划
----------------------------------------------------------
Plan hash value: 1557836545
-------------------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name          | Rows  | Bytes | Cost (%CPU)| Time       |
-------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |               |     2 |    90 |     2   (0)| 00:00:01   |
|   1 |  TABLE ACCESS BY INDEX ROWID| TEST_CI       |     2 |    90 |     2   (0)| 00:00:01   |
|*  2 |   INDEX RANGE SCAN                  | UPPER_IDX     |     2 |       |     1   (0)| 00:00:01   |
-------------------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access(UPPER("ENAME")='KING')
scott@ORCL>
```

这样，就在语句执行中使用了基于函数索引 UPPER_IDX，查询性能得到了一定的提升。想象在 ORACLE 引入 FBI 特性之前，TEST_CI 中的每一行都要扫描，并在把 ENAME 改为大写后与‘KING’进行比较，效率是非常低的。现在，利用 UPPDER（ENAME）上的索引，仅仅通过简单的一个索引范围扫描（**INDEX RANGE SCAN**）就可以获得相应的 ROWID，并按 ROWID 来访问表数据。这是相当快的。

FBI 实现选择性地索引

FBI 实现选择性地索引，是一种打破常规的做法，它的使用或许 FBI 的创造者也没有预

料到，不过只要想一下就会知道它们是很自然的。

‘打破常规’是一种流行的说法，它表示不受限于传统的知识，采用实践的方法，探索以前从未考虑过的问题。用你可供使用的工具考虑新的、不寻常的、创造性的方法。ORACLE 有许多特性和功能，也具有许多文献资料，它们能很清晰的向你阐述 ORACLE 是如何工作的。但文献资料永远不会提供我们所面临的特定问题的创造性解决方案。提出创造性解决方案是 DBA 或者应用程序设计、开发人员自己的任务。我们要以某种独特的、不一定能事先预料到结果的方式把所有线索拼凑起来，并最终能够解决应用种所面临的挑战性问题。’

Thomas Kyte

下面就分举例说明如何使用 FBI 打破常规，来解决应用中经常碰到的有选择性的进行索引问题。这种打破常规的做法是基于的 ORACLE 文献资料上记载的两个事实的：

- ✧ B*树索引决不能有完全为 NULL 的项。如果在 100 万上的表上利用在每行中都为 NULL 的列创建索引，那么这个索引是空的。
- ✧ FBI 可以组合出复杂的逻辑。

我们经常听到这样的问题，“怎样才能索引表中的某些行？”，“能不能只索引满足特定 WHERE 子句的少数记录但不对表中所有行进行索引？”考虑有一个很大的表，其中有一个 NOT NULL 列，名为 PROCESS_FLAG，它有两个取值‘Y’或‘N’，默认值为‘N’。PROCESS_FLAG 列指示该条记录是否处理过。其中**大量**记录都是‘Y’，只有很**少数**的记录取值为‘N’。‘N’指示这些记录是还未被处理的，你想找一种索引，它只索引 PROCEE_FLAG ＝‘N’的记录，这样不仅能减小索引的尺寸，而且也能提高索引的效率。

多数人头脑中马上想到位图索引，原因是这是个低基数的列。但问题是，我们要解决的这个问题发生是一个 OLTP 环境中，即该表上会有大量的插入、更新、删除操作。位图索引会严重恶化这些操作的性能。因此，必须寻找其它解决方法。

这里 FBI 就有了用武之地。通过使用 FBI，我们可以只对满足某种条件的列进行索引。通过使用 FBI，而且基于 B*树索引中不会有完全 NULL 的项这个事实，我们可以如下创建索引，就可以达到目的：

```
Create index selective_idx on table_name(
      Case when <some criteria is met> then ‘Y’
      Else NULL
       End)
```

下面举一个例子来详细说明一下。

◆ 创建一个据具有许多行的表

```
scott@ORCL> create table test_slc
  2  as
  3  select 'Y' process_flag,a.*
  4  from all_objects a;
表已创建。
scott@ORCL>
```

◆ 为了编程简单，创建一个视图，它有助于隐藏索引列的复杂性

```
scott@ORCL> create or replace view v_test_slc
  2  as
  3  select t.*,
  4            case when process_flag = 'N' then 'N'
  5                 else NULL
```

```
  6              end process_flag_indexed
  7   from test_slc t
  8   /
视图已创建。
scott@ORCL>
```

这种使用视图隐藏索引列复杂性的方法对大多数 FBI 工作良好。开发者或者最终用户不需要确切知道使用什么函数，他们只要使用视图中的一列即可，该列中有函数，而且 ORACLE 优化器也清楚的知道这一点。如果逻辑改变，只需要重建索引并更新视图，所有应用程序立即适应。因此这个方法在我们的应用中是非常值得借鉴的。

◆ 现在用相同的函数创建 FBI，并分析索引，看索引中当前有多少行

```
scott@ORCL> create index test_slc_idx on
  2   test_slc(case when process_flag='N' then 'N'
  3                 else NULL
  4            end)
  5   /
索引已创建。
scott@ORCL> create index test_slc_idx on
  2   test_slc(case when process_flag='N' then 'N'
  3                 else NULL
  4            end)
  5   /
索引已创建。
scott@ORCL> analyze index test_slc_idx   validate structure;
索引已分析
scott@ORCL> select name,del_lf_rows,lf_rows,lf_blks
  2   from index_stats
  3   /
NAME                           DEL_LF_ROWS      LF_ROWS      LF_BLKS
------------------------------ -----------      ----------   ----------
TEST_SLC_IDX                             0            0            1
scott@ORCL>
```

可以看到索引中什么都没有，0 个页行（LF_ROWS），没有删除页行（DEL_LF_ROWS）。

◆ 下面更新 TEST_SLC，设置其中 100 行的 PROCESS_FLAG 为‘N’

```
scott@ORCL> update test_slc set process_flag='N'
  2   where rownum <= 100
  3   /
已更新 100 行。
scott@ORCL> analyze index test_slc_idx   validate structure;
索引已分析
scott@ORCL> select name,del_lf_rows,lf_rows,lf_blks
  2   from index_stats
  3   /
NAME                           DEL_LF_ROWS      LF_ROWS      LF_BLKS
------------------------------ ----------- ---------- ----------
```

```
TEST_SLC_IDX                                    0          100          1
scott@ORCL> select count(*) from test_slc;
  COUNT(*)
  ----------
     50149
scott@ORCL>
```

可以看到索引有了 100 项，我们知道 TEST_SLC 表具有 **50149** 行，但是 PROCESS_FLAG 上的函数索引 TEST_SLC_IDX 却非常小，只有 **100** 行。这就达到了只对表中满足条件（PROCESS_FLAG=‘N’）的**少量数据**行进行索引的目的。

◆ 为了使用 FBI，必须使用 CBO，分析 TEST_SLC 以使下面处理使用 CBO。

```
scott@ORCL> analyze table test_slc compute statistics
  2  for table
  3  for all indexes
  4  for all indexed columns
  5  /
表已分析。
scott@ORCL>
```

◆ 模拟某种处理，访问第一个未处理记录，并更新其标记为‘Y’。进行两次处理。

```
scott@ORCL> column rowid new_val r
scott@ORCL> select rowid,object_name
  2  from v_test_slc
  3  where process_flag_indexed = 'N'
  4  and rownum = 1
  5  /
ROWID                  OBJECT_NAME
------------------ ------------------------------
AAANULAAEAAA39UAAA ICOL$
scott@ORCL> update v_test_slc
  2  set process_flag='Y'
  3  where rowid='&R'
  4  /
原值       3: where rowid='&R'
新值       3: where rowid='AAANULAAEAAA39UAAA'
已更新 1 行。
scott@ORCL>set autotrace on
scott@ORCL> select rowid,object_name
  2  from v_test_slc
  3  where process_flag_indexed = 'N'
  4  and rownum = 1
  5  /
ROWID                      OBJECT_NAME
------------------         ------------------------------
AAANULAAEAAA39UAAB I_USER1
```

```
执行计划
----------------------------------------------------------
Plan hash value: 3681546225
-------------------------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |               |     1 |    32 |     2   (0)| 00:00:01 |
|*  1 |  COUNT STOPKEY                      |               |       |       |            |          |
|   2 |   TABLE ACCESS BY INDEX ROWID| TEST_SLC       |     1 |    32 |     2   (0)| 00:00:01 |
|*  3 |    INDEX RANGE SCAN                 | TEST_SLC_IDX  |    99 |       |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter(ROWNUM=1)
   3 - access(CASE  WHEN "PROCESS_FLAG"='N' THEN 'N' ELSE NULL END ='N')

统计信息
----------------------------------------------------------
        128  recursive calls
          0  db block gets
         29  consistent gets
          0  physical reads
          0  redo size
        488  bytes sent via SQL*Net to client
        385  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          8  sorts (memory)
          0  sorts (disk)
          1  rows processed
scott@ORCL> update v_test_slc
  2  set process_flag = 'Y'
  3  where rowid='&R'
  4  /
原值       3: where rowid='&R'
新值       3: where rowid='AAANULAAEAAA39UAAB'
已更新 1 行。
scott@ORCL> analyze index test_slc_idx  validate structure;
索引已分析
scott@ORCL> set autotrace off
scott@ORCL> select name,del_lf_rows,lf_rows,lf_blks
  2  from index_stats
  3  /
NAME                           DEL_LF_ROWS    LF_ROWS    LF_BLKS
------------------------------ ----------- ---------- ----------
```

```
TEST_SLC_IDX                    2          100          1
scott@ORCL>
```

可见对 V_TEST_SLC 的检索成功的使用了表 TEST_SLC 上的函数索引 **TEST_SLC_IDX**。如果没有建立 **TEST_SLC_IDX** 索引，ORACLE 会使用在 TEST_SLC.PROCESS_FLAGL 列上建立的普通 B*树索引，为了仅从表中取出一条记录，它将会扫描数万条索引项，完成这个操作的逻辑 I/O（**consistent gets**）就不再是上面示例中 **29** 了，而是会以数万记，这与使用 TEST_SLC_IDX 索引从 100 个索引项中取出一条记录相比，它的速度会慢出很多。

再从上面的示例中观察索引 TEST_SLC_IDX 的状况，在从索引中删除（把原来索引的 PROCESS_FLAG 为‘N’的两条记录更新成‘Y’）两个索引项后之后，这个索引仍然保持一定的稳定状态，具有两个可被后继插入重用的删除项，索引将保持较小，索引状态良好。

以上示例巧妙利用了 FBI，打破常规地只对表中满足一定条件**少量数据（较小比例的数据）**进行索引，这样就可以有效减小索引的尺寸，而且可以极大提高使用这些索引的处理效率。

下面将再次巧用 FBI，实现选择唯一性（可以理解成某种完整性约束）。

FBI 实现选择唯一性

实现‘选择唯一性’，是另一个挑战性问题。下面就通过一个实际应用中的例子来说明FBI是如何解决这种问题的。（这个例子上在有关ORACLE的书籍和杂志上广为流传。）

‘我有一个表PROJECT(PROJECT_ID NUMBER PRIMARY KEY,TEAMID NUMBER,JOB VARCHAR2(100),STATUS VARCHAR2(8))。STATUS 列可能具有值 ACTIVE、INACTIVE，前者表示项目是活动的，后者表示项目是归档的。要求执行一个惟一约束规则：活动的项目必须有个惟一名，而不活动的项目无此要求’。也就是说只有一个活动的“项目X”，但是如果你愿意，你可以有很多名X的不活动项目。完成此任务的最好方法是什么？

开发人员了解到这个需求，可能首先想到的是使用触发器。但是熟悉ORACLE并发控制和多版本内容的人都知道，这种简单的触发器实现在多用户环境下是不可行的。如果两个人想同时创建一个活动项目X，他们都会成功的。这里需要将X的创建串行化，但是对此惟一的做法是锁住整个项目表（这样做并发性就不太好了），或者使用FBI，让数据库为我们做这个工作。

确实可以使用FBI来解决这类惟一性约束问题。这同样基于如下一条事实：

✧ B*树索引决不能有完全为 NULL 的项。如果在 100 万上的表上利用在每行中都为 NULL 的列创建索引，那么这个索引是空的。

可以如下创建一个UNIQUE索引：

```
Create unique index active_project_must_be_unique
On project (case    when status='ACTIVE' then name end);
```

这就行了，ORACLE使用了FBI，当状态列STATUS=‘ACTIVE’，时，NAME 列上将建立惟一索引，如果试图创建同名的活动项目时，ORACLE就会检测得到违反这个惟一索引，而且这根本不会影响对这个表的并发访问。只对活动项目进行索引的结果也很好，它只消耗了尽可能少的存储空间。

基于函数索引的一些问题

FBI 的性能问题

谈到 FBI，很多人会想到它的性能问题，的确，使用 FBI 确实会存在一定的性能开销。但是，这不仅仅是 FBI 的问题，任何类型的索引都会引起性能的开销，比如，任何类型的索引都会影响插入的性能，这是勿庸置疑的。但是为什么应用设计中还是使用索引呢？这是因为索引虽然一定程度上恶化了数据插入的性能，但同时却带来了查询性能的显著提高。一条数据可能只插入你的数据库 1 次，但是却可能被查询若干次。因此如果能提高查询性能的，那么牺牲一点插入的性能还是值得的。下面就通过示例来具体了解一下 FBI 的性能。

◆ 示例仍使用大小写无关查询中创建的表 TEST_CI

```
scott@ORCL> create table test_ci
  2  as
  3  select *
  4    from emp
  5  where 1=0
  6  /
表已创建。
scott@ORCL> insert into test_ci(empno,ename,job,mgr,hiredate,sal,comm,deptno)
  2  select rownum empno,
  3             initcap(substr(object_name,1,10)) ename,
  4             substr(object_type,1,9) job,
  5             rownum mgr,
  6             created hiredate,
  7             rownum sal,
  8             rownum comm,
  9             (mod(rownum,4)+1)*10 deptno
 10  from all_objects
 11  where rownum <10000
 12  /
已创建 9999 行。
scott@ORCL>
```

◆ 创建一个较为复杂的用户自定义函数 MY_FUNC 用来创建 FBI，为了监控 MY_FUNC 在示例语句中执行了多少次，再创建一个全局包变量作为一个计数器。

```
scott@ORCL> create or replace package counter
  2  as
  3             cnt number default 0;
  4  end;
  5  /
程序包已创建。
scott@ORCL> create or replace function
  2  my_func1(p_string in varchar2 ) return varchar2
  3  deterministic
```

```
  4  as
  5      l_return_string varchar2(6) default substr(p_string,1,1);
  6      l_char varchar2(1);
  7      l_last_digit number default 0;
  8      type vcarray is table of varchar2(10) index by binary_integer;
  9      l_code_table vcarray;
 10  begin
 11      counter.cnt:=counter.cnt+1;
 12      l_code_table(1):='BPFV';
 13      l_code_table(2):='CSKGJQXZ';
 14      l_code_table(3):='DT';
 15      l_code_table(4):='L';
 16      l_code_table(5):='MN';
 17      l_code_table(6):='R';
 18      for i in 1..length(p_string)
 19      loop
 20          exit when (length(l_return_string)=6);
 21          l_char:=upper(substr(p_string,i,1));
 22          for j in 1..l_code_table.count
 23          loop
 24          if (instr(l_code_table(j),l_char)>0 and j <> l_last_digit)
 25          then
 26              l_return_string:=l_return_string||to_char(j,'fm9');
 27              l_last_digit:=j;
 28          end if;
 29          end loop;
 30      end loop;
 31      return rpad(l_return_string,6,'0');
 32  end;
 33  /
函数已创建。
scott@ORCL>
```

注意在这个函数中，这里就使用了关键字 **deterministic**，这就声明了：这个函数在给定相同的输入时，无论被调用多少次，它总会返回完全相同的输出。必须是声明为 **deterministic** 的函数才能被用来建立 FBI。

没有必要看懂这个函数，觉得它足够复杂就行。下面就用该函数来创建 FBI。

◆ 在创建 FBI 前，先看看没有 FBI 索引的情况

```
scott@ORCL> set timing on
scott@ORCL> set autotrace on explain
scott@ORCL> select ename,hiredate
  2  from test_ci
  3  where my_func1(ename)=my_func1('Kings')
  4  /
ENAME      HIREDATE
```

```
---------- --------------
Ku$_Chunk_ 30-8 月  -05
Ku$_Chunk_ 30-8 月  -05
Ku$_Chunk_ 30-8 月  -05
Ku$_Chunk_ 30-8 月  -05
已用时间:   00: 00: 01.51
执行计划
----------------------------------------------------------
Plan hash value: 2340976419
-------------------------------------------------------------------------------------------------
| Id  | Operation                   | Name       | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |            |   100 |  1900 |     33   (46)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL          | TEST_CI    |   100 |  1900 |     33   (46)| 00:00:01 |
-------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("MY_FUNC1"("ENAME")="MY_FUNC1"('Kings'))
scott@ORCL> set autotrace off
scott@ORCL> set timing off
scott@ORCL> set serveroutput on
scott@ORCL> exec dbms_output.out_line(counter.cnt);
9977
PL/SQL 过程已成功完成。
```

可以看到这个查询用时 **00: 00: 01.51**，而且执行了全表扫描 **TABLE ACCESS FULL**。函数 MY_FUNC1 被调用了 **9977** 次。

◆ 对 MY_FUNC1 创建索引，看看查询性能会有如何变化。

```
scott@ORCL> rem reset our counter
scott@ORCL> exec counter.cnt:=0
PL/SQL 过程已成功完成。
scott@ORCL> set timing on
scott@ORCL> set autotrace on explain
scott@ORCL> select ename,hiredate
  2  from test_ci
  3  where substr(my_func1(ename),1,6) = my_func1('Kings')
  4  /
ENAME          HIREDATE
---------- --------------
Ku$_Chunk_ 30-8 月  -05
Ku$_Chunk_ 30-8 月  -05
Ku$_Chunk_ 30-8 月  -05
Ku$_Chunk_ 30-8 月  -05

已用时间:   00: 00: 00.04
```

```
执行计划
----------------------------------------------------------
Plan hash value: 400330232
-------------------------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |               |   100 |  1900 |    11   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| TEST_CI       |   100 |  1900 |    11   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN                  | MY_FUNC1_IDX |    40 |       |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access(SUBSTR("SCOTT"."MY_FUNC1"("ENAME"),1,6)="MY_FUNC1"('Kings'))
scott@ORCL> set autotrace off
scott@ORCL> set timing off
scott@ORCL> set serveroutput on
scott@ORCL> exec dbms_output.put_line(counter.cnt);
1
PL/SQL 过程已成功完成。
```

可以看到，使用 FBI 后，查询使用了 MY_FUNC1_IDX 索引上的索引范围扫描 **INDEX RANGE SCAN**，查询使用时间降到 **00: 00: 00.04**，仅为原来的 **0.04/1.51** × 100%＝**2.65%**；MY_FUNC1 函数仅仅被执行了一次。

◆ 在看看使用索引前后的 INSERT 性能

```
scott@ORCL> alter session set sql_trace=true;
会话已更改。
scott@ORCL> alter session set events '10046 trace name context forever,level 12';
会话已更改。
scott@ORCL> insert into test_ci(empno,ename,job,mgr,hiredate,sal,comm,deptno)
  2       select rownum empno,
  3              initcap(substr(object_name,1,10)) ename,
  4              substr(object_type,1,9) job,
  5              rownum mgr,
  6              created hiredate,
  7              rownum sal,
  8              rownum comm,
  9              (mod(rownum,4)+1)*10 deptno
 10      from all_objects
 11      where rownum <10000
 12      /

已创建9999行。
－－查看生成的跟踪文件，得到如下信息
********************************************************************************
```

```
call     count       cpu    elapsed       disk      query    current
rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.14       0.14          0          0          0
0
Execute      1      2.05       3.10          8      16910      15350
9999
Fetch        0      0.00       0.00          0          0          0
0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        2      2.19       3.24          8      16910      15350        9999

Misses in library cache during parse: 1
Optimizer mode: ALL_ROWS
Parsing user id: 54
——此时，丢掉索引MY_FUNC1_IDX，再看看INSERT 性能
scott@ORCL> drop index MY_FUNC1_IDX;
索引已删除。
scott@ORCL> truncate table test_ci;
表被截断。
scott@ORCL> alter session set sql_trace=true;
会话已更改。
scott@ORCL> alter session set events '10046 trace name context forever,level 12';
会话已更改。
scott@ORCL> insert into test_ci(empno,ename,job,mgr,hiredate,sal,comm,deptno)
  2      select rownum empno,
  3             initcap(substr(object_name,1,10)) ename,
  4             substr(object_type,1,9) job,
  5             rownum mgr,
  6             created hiredate,
  7             rownum sal,
  8             rownum comm,
  9             (mod(rownum,4)+1)*10 deptno
 10     from all_objects
 11     where rownum <10000
 12     /
已创建9999行。
——再次查看生成的跟踪文件，得到如下信息
********************************************************************************
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.05       0.25          0          0          0           0
Execute      1      0.57       1.15          5      16747       8911        9999
Fetch        0      0.00       0.00          0          0          0           0
```

```
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        2      0.62       1.41          5      16747       8911        9999

Misses in library cache during parse: 1
Optimizer mode: ALL_ROWS
Parsing user id: 54
```

没有索引，INSERT 时，ORACLE 少了一大块成本开销，因此速度极快，总共才用时 1.41，仅为使用 FBI 时的 1.41/3.24×100%＝43.52% 。

从上面的示例可以看出：如果不使用 FBI，INSERT 操作的速度确实能提高 1 倍多，但是如果使用 FBI，查询性能却提高了 50 倍。因此，我个人觉得，使用 FBI 还是极具性价比的。

FBI 和 CASE 警告

某些 ORACLE 版本有一个 BUG，其中 FBI 使用的函数会以某种方式被重写，以致于索引无法被透明使用。例如，下面的 CASE 语句

```
Case when X = 'N' then 'N' end
```

会被 ORACLE 悄悄重写成以下更高效的语句：

```
Case X  when 'N' then 'N' end
```

但是这个函数与我们创建时使用的那个函数不再匹配，所以查询无法使用此函数。请看下面的示例：

```
scott@ORCL> create table t_case (x int);
表已创建。
scott@ORCL> create index t_case_idx on t_case(case when x=42 then 1 end);
索引已创建。
scott@ORCL> set autotrace traceonly explain
scott@ORCL> select /*+ index(t_case t_case_idx)*/ *
  2  from t_case
  3  where (case when x=42 then 1 end)=1;
执行计划
----------------------------------------------------------
Plan hash value: 1479674813
-------------------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |               |     1 |    13 |     1   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T_CASE        |     1 |    13 |     1   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN                  | T_CASE_IDX |     1 |       |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access(CASE "X" WHEN 42 THEN 1 END =1)
Note
```

```
      -----
         - dynamic sampling used for this statement
scott@ORCL> set autotrace off
scott@ORCL> select column_expression
  2  from user_ind_expressions
  3  where index_name='T_CASE_IDX'
  4  /
COLUMN_EXPRESSION
--------------------------------------------------------------------------------
CASE "X" WHEN 42 THEN 1 END
scott@ORCL>
```

这个示例是我在 ORACLE 10gR2 里执行的，所以一切 OK，ORACLE 虽把‘**case when x=42 then 1 end**’ 语句重写成了‘**CASE "X" WHEN 42 THEN 1 END**’（通过查询系统视图 user_ind_expressions 的 column_expression 列可知。）但是我们在查询中如示例中那样使用‘where (**case when x=42 then 1 end**)=1’子句时，仍然可以使用 **T_CASE_IDX** 索引。但是再 ORACLE 10.1.0.3 版本中执行上面的示例会发现使用了‘where (**case when x=42 then 1 end**)=1’子句的查询无法使用 **T_CASE_IDX** 索引，这是 ORACLE 的 bug，这个 bug 在 ORACLE 的 10.1.0.4 中得到了修复。

如果你正在使用以前版本的 ORACLE，你可能会碰巧碰到这个问题，对此的解决的办法有：

- 使用 DECODE 来取代 CASE，因为 DECODE 不会被重写；
- 使用类似‘**CASE X WHEN 42 THEN 1 END**’的优化语法，防患未然。

倘若，ORACLE 优化器仍然没有使用你的 FBI，那么就可以检查 user_ind_expressions 视图，验证函数使用的方法是否正确，使用要与 user_ind_expressions. column_expression 列保持一致。

FBI 和 ORA-01743 警告

对于FBI，我们会发现有这样一个奇怪的现象，如果你在内置函数TO_DATE上创建一个索引，某些情况下并不能成功创建，例如：

```
scott@ORCL> create table t_to_date(year varchar2(4));
表已创建。
scott@ORCL> create index t_to_date_idx on t_to_date(to_date(year,'YYYY'));
create index t_to_date_idx on t_to_date(to_date(year,'YYYY'))
                                                        *
第 1 行出现错误:
ORA-01743: 仅能编制纯函数的索引
```

这看上去很奇怪，因为有时候使用TO_DATE又可以成功创建FBI，例如：

```
scott@ORCL> create index t_to_date_idx on t_to_date(to_date('01'||year,'MMYYYY'));
索引已创建。
```

相关错误信息也很含糊，让人摸不着头脑：

```
scott@ORCL>!oerr ora 1743
01743, 00000, "only pure functions can be indexed"
```

```
// *Cause: The indexed function uses SYSDATE or the user environment.
// *Action: PL/SQL functions must be pure (RNDS, RNPS, WNDS, WNPS).  SQL
//          expressions must not use SYSDATE, USER, USERENV(), or anything
//          else dependent on the session state.  NLS-dependent functions
//          are OK.
```

上例中并没有使用SYSDATE，也没有使用用户环境。这里并没有使用PL/SQL函数，而且没有任何与会话状态有关的方面。问题只是在于我们使用的格式：‘YYYY’。给定完全相同的输入，这种格式可能返回不同的答案，这取决于调用函数时输入的月份。例如对于11月的某个时间：

```
scott@ORCL> select to_char(to_date('2007','YYYY'),'DDMMYYYY') from dual;
TO_CHAR(
--------
01112007
```

改变系统日期为 2007年12月的一天，再次运行上述查询：

```
scott@ORCL> select to_char(to_date('2007','YYYY'),'DDMMYYYY') from dual;
TO_CHAR(
--------
01122007
```

因此TO_DATE函数使用‘YYYY’格式时，不具备创建FBI的先决条件：**Deterministic**，即确定性。这真是无法创建索引的原因。因此要记住：要在一个FBI中使用TO_DATE，必须使用一种无歧义的确定性日期格式。

基于函数索引小结

基于函数的索引已经很成熟，它很容易实现和使用。使用这种索引能提前计算并存储复杂的值，因此可以用来加快现有应用的查询速度，而且不用修改应用中的任何逻辑或查询。使用FBI还可以有很多巧妙的应用，如：FBI实现大小写无关的查询、FBI实现选择性地索引、FBI实现选择唯一性等等。这些巧妙的打破常规的应用，极大的提高了FBI的应用价值。

FBI和其它所有的索引一样，会影响插入和更新的性能。必须考虑这一点对你是否重要。如果你总是插入数据，而不经常查询数据，FBI可能对你并不适用。另一方面，要记住，一般数据都是一次插入一行，查询可能会有数千次。所以一次性插入时的性能下降（最终用户跟本感觉不到）能换来无数次的查询速度的提高（最终用户是有切身体会的）。一般说来，这种情况下利远大于弊。所以，强烈推荐在应用中战术性的使用FBI。

应用域索引

ORACLE 称应用域索引（Application Domain Index,，下文简称 ADI）为可扩展索引（Extensible Indexing）。利用它可以创建用户自定义的索引结构，此索引结果就像 ORACLE 所提供的索引结构那样工作。如果有人用你创建的索引类型发布一条 CREATE INDEX 语句，ORACLE 将以你所希望的存储格式生成索引。也就是说 ADI 提供了一种索引‘DIY’能力，它完全可以实现第三方索引技术。

目前这种技术在影像、GPS 等领域应用颇多。ORACLE 自己的文本索引（Text Indexing）

也是 ADI 应用的典型例子，这个索引用于对大量的文本提供关键字搜索。

由于这种技术在我们公司开发的系统中鲜有应用而且也确实使用的机会不多，限于篇幅，本文不再在这里进行详细阐述。如果有兴趣的话，可以通过 ORACLE 的相关手册去了解。

小结

在本文中，我们介绍了 ORACLE 提供的各种类型的索引。首先讨论了最传统的 B*树索引，并讲了它的几个子类型：反向键索引、降序索引等等。而且还花时间讨论了这些索引的使用场合。

其次介绍了位图索引及其变种位图联接索引。知道了这种类型的索引比较适合数据仓库/MIS/BI 环境中基数较低的列，而且了解到不适宜创建复合位图索引。

接着又介绍了基于函数索引，它实际上是一种 B*树索引或者位图索引。文中谈到了基于函数索引的几个典型应用：实现大小写无关查询、实现选择性的索引、实现选择惟一等。这些应用中有些是 ORACLE 中打破常规的应用。该主题的最后又谈到了使用 FBI 中碰到的问题，如性能问题、有些版本中由于 BUG 而导致的 CASE 问题以及使用 TO_DATE 时可能会碰到的 ORA-01743 问题。

最后简略了分析了一个特殊的应用域索引，由于我们公司在开发中几乎用不到，文中并没有深入讲解。只要大致明白它是一种 DIY 索引类型就行，通过 ADI，ORACLE 向世人展示了它具有提供可扩展索引（Extensible Indexing）的能力。

索引设计是 ORACLE 模式设计的一个重要组成部分（本文本来就应是《高效 ORACLE 之 模式》一文中的一章，但是当时限于篇幅，把这一章从中摘出单独成文）。要做好索引的模式设计，设计人员必须充分了解应用中的**数据特征**及其**使用方式**。数据的特征可以从战略角度来决定你使用的索引类型（B*树索引、位图索引、应用域索引）。例如，如果你面临的是数据仓库/MIS/BI 类应用(OLAP)，那么你的数据特征可能是以‘只读’为主，那么战略上就应该多考虑使用位图索引；如果你面临的是具有大量高并发性插入、更新操作的 OLTP 系统，那么 B*树索引应该是你主要考虑的索引类型。数据的使用方式可以从战术角度来决定你应该使用哪类索引子类型。例如，如果你的语句中有降序排列需求，那么就可以使用 B*树的降序索引；如果你处在 RAC 的高并发性 DML 环境下，那么就可以考虑使用 B*树的反向键索引；如果你在 WHERE 子句中多引用函数，那么请考虑 B*树/位图函数索引；如果你的用户要你实现一个类似语义层的检索，那么位图联接索引可能是你的较好选择。等等……

下面，总结一下本人在应用中进行索引设计的一般步骤，仅供参考：

- **决定是否需要使用索引**
- **决定使用何种类型的索引**
- **决定是否使用某种索引子类型**
- **以与生产相当规模的数据测试你的索引，看它能否真正的起到作用**
- **循环上述步骤，直到项目结束**

注：有时候，系统投产后出现了性能问题，可能需要在生产环境上设计新索引来‘救火’。此时更应遵守上述步骤，尤其要注意其中的第四步，一定要在准生产环境上进行充分测试。