第 1 章

Oracle 里的优化器

到目前为止，Oracle 数据库是市场占有率最高（接近 50%），使用范围最广的关系型数据库（RDBMS），这意味着有太多太多的系统都是构建在 Oracle 数据库上的。而我们大家都知道，**对于使用关系型数据库的应用系统而言，SQL 语句的好坏会直接影响系统的性能，很多系统性能很差最后发现都是因为 SQL 写得很烂的缘故**。实际上，一条写得很烂的 SQL 语句就能拖垮整个应用，极端情况下，一条写得很烂的 SQL 语句甚至会导致数据库服务器失去响应或者使整个数据库 Hang 住，去 Google 一下吧，这样的例子有很多！

怎样避免在 Oracle 数据库中写出很烂的 SQL？或者说应该如何在 Oracle 数据库中做 SQL 优化？这个问题真的很不好回答，且容我慢慢道来。

对所有的关系型数据库而言，优化器无疑是其中最核心的部分，因为优化器负责解析 SQL，而我们又都是通过 SQL 来访问存储在关系型数据库中的数据的，所以优化器的好坏会直接决定该关系型数据库的强弱。从另外一个方面来说，正是因为优化器负责解析 SQL，所以要想做好 SQL 优化就必须了解优化器，而且最好是能全面、深入的了解，这是做好 SQL 优化基础中的基础。

Oracle 数据库里的优化器以其复杂、强悍而闻名于世，本章会详细介绍与 Oracle 数据库里优化器相关的基础知识，目的是希望通过这一章的介绍，让大家对 Oracle 数据库里的优化器有一个全局、概要性的认识，打好基础，为阅读后续章节扫清障碍。

1.1 什么是 Oracle 里的优化器

优化器（Optimizer）是 Oracle 数据库中内置的一个核心子系统，你也可以把它理解成是 Oracle 数据库中的一个核心模块或者一个核心功能组件。优化器的目的是按照一定的判断原则来得到它认为的目标 SQL 在当前情形下最高效的执行路径（Access Path），也就是说，优化器的目的就是为了得到目标 SQL 的执行计划（关于执行计划，会在“第 2 章 Oracle 里的执行计划”中详细描述）。

依据选择执行计划时所用的判断原则，Oracle 数据库里的优化器又分为 RBO 和 CBO 这两种类型。RBO 是 Rule-Based Optimizer 的缩写，直译过来就是“基于规则的优化器”；相对应的，CBO 是 Cost-Based Optimizer 的缩写，直译过来就是“基于成本的优化器”。

在得到目标 SQL 的执行计划时，RBO 所用的判断原则为一组内置的规则，这些规则是硬编码在 Oracle 数据库的代码中的，RBO 会根据这些规则从目标 SQL 诸多可能的执行路径中选择一条来作为其执行计划；而

CBO 所用的判断原则为成本，CBO 会从目标 SQL 诸多可能的执行路径中选择成本值最小的一条来作为其执行计划，各个执行路径的成本值是根据目标 SQL 语句所涉及的表、索引、列等相关对象的统计信息计算出来的（关于统计信息，会在“第 5 章 Oracle 里的统计信息”中详细描述）。

Oracle 数据库里 SQL 语句的执行过程可以用图 1-1 来表示。

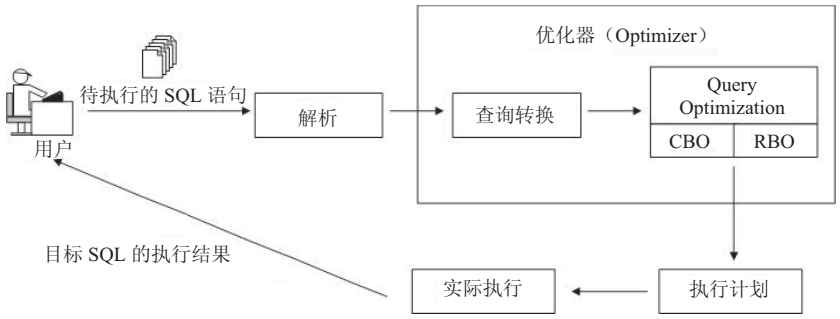

图 1-1 Oracle 数据库里 SQL 语句的执行过程

关于图 1-1，会在“第 4 章 Oracle 里的查询转换”中详细说明，这里只需要知道 Oracle 里优化器的输入是经过解析后（在这个解析过程中，Oracle 会执行对目标 SQL 的语法、语义和权限检查）的目标 SQL，输出是该目标 SQL 的执行计划就好了。

接下来，分别介绍 RBO 和 CBO。

1.1.1 基于规则的优化器

之前已经提到，基于规则的优化器（RBO）通过硬编码在 Oracle 数据库代码中的一系列固定的规则，来决定目标 SQL 的执行计划。具体来说就是这样：Oracle 会在代码里事先给各种类型的执行路径定一个等级，一共有 15 个等级，从等级 1 到等级 15。并且 Oracle 会认为等级值低的执行路径的执行效率会比等级值高的执行效率要高，也就是说在 RBO 的眼里，等级 1 所对应的执行路径的执行效率最高，等级 15 所对应的执行路径的执行效率最低。在决定目标 SQL 的执行计划时，如果可能的执行路径不止一条，则 RBO 就会从该 SQL 诸多可能的执行路径中选择一条等级值最低的执行路径来作为其执行计划。

RBO 是一种适用于 OLTP 类型 SQL 语句的优化器，在这样的前提条件下，大家来猜一猜 RBO 的等级 1 和等级 15 所对应的执行路径分别是什么？

在 Oracle 数据库里，对于 OLTP 类型的 SQL 语句而言，显然通过 ROWID 来访问是效率最高的方式，而通过全表扫描来访问则是效率最低的方式。与之相对应的，RBO 内置的等级 1 所对应的执行路径就是“single row by rowid（通过 rowid 来访问单行数据）”，而等级 15 所对应的执行路径则是“full table scan（全表扫描）”。

RBO 在 Oracle 中由来已久，虽然从 Oracle 10g 开始，RBO 已不再被 Oracle 支持，但 RBO 的相关实现代码并没有从 Oracle 数据库的代码中移除，这意味着即使是在 Oracle 11gR2 中，我们依然可以通过修改优化器模式或使用 RULE Hint 来继续使用 RBO。

和 CBO 相比，RBO 是有其明显缺陷的。在使用 RBO 的情况下，执行计划一旦出了问题，很难对其做调整；另外，如果使用了 RBO，则目标 SQL 的写法，甚至是目标 SQL 中所涉及的各个对象在该 SQL 文本中出现的先后顺序，都可能会影响 RBO 对于该 SQL 执行计划的选择。更糟糕的是，Oracle 数据库中很多很好的特

性、功能均不能在 RBO 下使用，因为它们均不被 RBO 所支持。

只要出现了如下的情形之一（包括但不限于这些情形），那么即便你修改了优化器模式或者使用了 RULE Hint，Oracle 依然不会使用 RBO（而是强制使用 CBO）：

- 目标 SQL 中涉及的对象有 IOT（Index Organized Table）。
- 目标 SQL 中涉及的对象有分区表。
- 使用了并行查询或者并行 DML。
- 使用了星型连接。
- 使用了哈希连接。
- 使用了索引快速全扫描。
- 使用了函数索引。
- ……

在使用 RBO 的情况下，一旦 RBO 选择的执行计划并不是当前情形下最优的执行计划，应该如何对其做调整呢？

这种情况下我们是很难对 RBO 选择的执行计划做调整的，其中非常关键的一个原因就是不能使用 Hint，因为如果在目标 SQL 中使用了 Hint，就意味着自动启用了 CBO，即 Oracle 会以 CBO 来解析含 Hint 的目标 SQL。这里仅有两个例外，就是 RULE Hint 和 DRIVING_SITE Hint，它们可以在 RBO 下使用并且不自动启用 CBO（关于 Oracle 中的 Hint，会在“第 6 章 Oracle 里的 Hint”详细说明）。

那么，是不是在使用 RBO 的情况下就没办法对执行计划做调整了？

当然不是这样，只是这种情况下我们的调整手段会非常有限。其中的一种可行的方法就是等价改写目标 SQL，比如在目标 SQL 的 where 条件中对 NUMBER 或 DATE 类型的列加上 0（如果是 VARCHAR2 或 CHAR 类型，可以加上一个空字符，例如 ||''），这样就可以让原本可以走的索引现在走不了。对于包含多表连接的目标 SQL 而言，这种改变甚至可以影响表连接的顺序，进而就可以实现在使用 RBO 的情况下对该目标 SQL 的执行计划做调整的目的。

之前已经提到：RBO 会从目标 SQL 诸多可能的执行路径中选择一条等级值最低的作为其执行计划，但如果出现了两条或者两条以上等级值相同的执行路径的情况，那么此时 RBO 会如何选择呢？很简单，此时 RBO 会依据目标 SQL 中所涉及的相关对象在数据字典缓存（Data Dictionary Cache）中的缓存顺序和目标 SQL 中所涉及的各个对象在目标 SQL 文本中出现的先后顺序来综合判断。这也就意味着我们还可以通过调整相关对象在数据字典缓存中的缓存顺序，改变目标 SQL 中所涉及的各个对象在该 SQL 文本中出现的先后顺序来调整其执行计划。

我们来看一个在使用 RBO 的情况下对目标 SQL 的执行计划做调整的实例。创建一个测试表 EMP_TEMP：

```
SQL> create table emp_temp as select * from emp;

Table created
```

在表 EMP_TEMP 的列 MGR 和 DEPTNO 上分别创建两个名为 IDX_MGR_TEMP 和 IDX_DEPTNO_TEMP 的索引：

```
SQL> create index idx_mgr_temp on emp_temp(mgr);
```

```
Index created

SQL> create index idx_deptno_temp on emp_temp(deptno);

Index created
```

我们来看一下如下的范例 SQL 1：

```
select * from emp_temp
  where mgr > 100 and deptno > 100;
```

对于范例 SQL 1 而言，其 where 条件中出现了列 MGR 和 DEPTNO，而在列 MGR 和 DEPTNO 上分别存在着索引 IDX_MGR_TEMP 和 IDX_DEPTNO_TEMP。

现在的问题是，如果在启用 RBO 的情形下执行范例 SQL 1，则 Oracle 会选择走上述两个索引中的哪一个？

我们来实际验证一下。在当前 Session 中将优化器模式修改为 RULE，表示在当前 Session 中启用 RBO：

```
SQL> alter session set optimizer_mode='RULE';

Session altered
```

然后执行范例 SQL 1：

```
SQL> set autotrace traceonly explain
SQL> select * from emp_temp where mgr>100 and deptno>100;
执行计划
----------------------------------------------------------
Plan hash value: 1670750536

-----------------------------------------------------------
| Id  | Operation                    | Name            |
-----------------------------------------------------------
|   0 | SELECT STATEMENT             |                 |
|*  1 |  TABLE ACCESS BY INDEX ROWID| EMP_TEMP         |
|*  2 |   INDEX RANGE SCAN           | IDX_DEPTNO_TEMP |
-----------------------------------------------------------

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

   1 - filter("MGR">100)
   2 - access("DEPTNO">100)

Note
-----
   - rule based optimizer used (consider using cbo)
```

注意到 Id = 2 的执行步骤为“INDEX RANGE SCAN | IDX_DEPTNO_TEMP”，Note 部分有关键字“rule based optimizer used (consider using cbo)”，这说明 Oracle 在执行上述范例 SQL 1 时使用的是 RBO，且选择的是走对索引 IDX_DEPTNO_TEMP 的索引范围扫描。

范例 SQL 1 的 where 条件中有“mgr>100”，所以 RBO 实际上是可以选择走列 MGR 上的索引 IDX_MGR_TEMP 的，只不过 RBO 这里并没有选择走该索引，而是选择走列 DEPTNO 上的索引 IDX_DEPTNO_TEMP。

假如我们发现走索引 IDX_DEPTNO_TEMP 不如走索引 IDX_MGR_TEMP 的执行效率高，或者说我们就想让 RBO 选择走索引 IDX_MGR_TEMP，那么应该如何做呢？

之前已经提到过：在使用 RBO 的情况下，可以通过等价改写目标 SQL（加 0 或者空字符串的方式）来调整该 SQL 的执行计划。列 DEPTNO 的类型为 NUMBER，所以我们可以在列 DEPTNO 上加 0，来达到不让 RBO 选择走其上的索引 IDX_DEPTNO_TEMP 的目的。在列 DEPTNO 上加 0 后即形成了如下形式的范例 SQL 2：

```
select * from emp_temp
  where mgr>100 and deptno+0>100;
```

执行范例 SQL 2：

```
SQL> select * from emp_temp where mgr>100 and deptno+0>100;
执行计划
----------------------------------------------------------
Plan hash value: 2973289657

-----------------------------------------------------
| Id  | Operation                    | Name         |
-----------------------------------------------------
|   0 | SELECT STATEMENT             |              |
|*  1 |  TABLE ACCESS BY INDEX ROWID| EMP_TEMP     |
|*  2 |   INDEX RANGE SCAN           | IDX_MGR_TEMP |
-----------------------------------------------------

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

   1 - filter("DEPTNO"+0>100)
   2 - access("MGR">100)

Note
-----
   - rule based optimizer used (consider using cbo)
```

注意，此时 Id = 2 的执行步骤已经从之前的“INDEX RANGE SCAN | IDX_DEPTNO_TEMP”变为了现在的“INDEX RANGE SCAN | IDX_MGR_TEMP”，这说明我们确实迫使 RBO 改变了执行计划，即我们的调整已经生效了。

之前已经提到：如果目标 SQL 出现了有两条或者两条以上的执行路径的等级值相同的情况，我们可以通过调整相关对象在数据字典缓存中的缓存顺序来影响 RBO 对于其执行计划的选择。对于范例 SQL 1 而言，对索引 IDX_DEPTNO_TEMP 走索引范围扫描和对索引 IDX_MGR_TEMP 走索引范围扫描的等级值显然是相同的，所以我们就可以通过调整这两个索引在数据字典缓存中的缓存顺序来改变执行计划。

刚才我们先创建索引 IDX_MGR_TEMP，再创建索引 IDX_DEPTNO_TEMP，所以索引 IDX_MGR_TEMP 和 IDX_DEPTNO_TEMP 在数据字典缓存中的缓存顺序是，先缓存 IDX_MGR_TEMP，再缓存 IDX_DEPTNO_TEMP。这种情形下 RBO 选择的是走对索引 IDX_DEPTNO_TEMP 的索引范围扫描，如果我们现在把索引 IDX_MGR_TEMP 先 Drop 掉再重新创建一次，那么就相当于是先创建索引 IDX_DEPTNO_TEMP，再创建索引 IDX_MGR_TEMP，也就是说此时这两个索引在数据字典缓存中的缓存顺序就刚好颠倒过来了。按照此前介绍的知识，此时 RBO 应该就会选择走对索引 IDX_MGR_TEMP 的索引范围扫描。

现在验证一下：

先 Drop 掉索引 IDX_MGR_TEMP：

```
SQL> drop index idx_mgr_temp;


Index dropped
```

再重新创建上述索引 IDX_MGR_TEMP：

```
SQL> create index idx_mgr_temp on emp_temp(mgr);

Index created
```

然后再次执行范例 SQL 1：

```
SQL> select * from emp_temp where mgr>100 and deptno>100;
执行计划
----------------------------------------------------------
Plan hash value: 2973289657

-----------------------------------------------------
| Id  | Operation                    | Name         |
-----------------------------------------------------
|   0 | SELECT STATEMENT             |              |
|*  1 |  TABLE ACCESS BY INDEX ROWID| EMP_TEMP     |
|*  2 |   INDEX RANGE SCAN           | IDX_MGR_TEMP |
-----------------------------------------------------

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

   1 - filter("DEPTNO">100)
   2 - access("MGR">100)

Note
-----
   - rule based optimizer used (consider using cbo)
```

注意，Id = 2 的执行步骤已经从之前的“INDEX RANGE SCAN | IDX_DEPTNO_TEMP”变为了现在的“INDEX RANGE SCAN | IDX_MGR_TEMP”，说明我们确实迫使 RBO 改变了执行计划，这也说明当目标 SQL 有两条或者两条以上的执行路径的等级值相同时，我们确实可以通过调整相关对象在数据字典缓存中的缓存顺序来影响 RBO 对于其执行计划的选择。

我们之前还提到过：如果目标 SQL 出现了有两条或者两条以上的执行路径的等级值相同的情况，可以通过改变目标 SQL 中所涉及的各个对象在该 SQL 文本中出现的先后顺序来调整该目标 SQL 的执行计划。这通常适用于目标 SQL 中出现了多表连接的情形，在目标 SQL 出现了有两条或者两条以上的执行路径的等级值相同的前提条件下，RBO 会按照从右到左的顺序来决定谁是驱动表，谁是被驱动表，进而会据此来选择执行计划，所以如果我们改变了目标 SQL 中所涉及的各个对象在该 SQL 文本中出现的先后顺序，也就改变了表连接的驱动表和被驱动表，进而就调整了该 SQL 的执行计划。

我们来验证一下上述结论。再创建一个测试表 EMP_TEMP1：

```
SQL> create table emp_temp1 as select * from emp;

Table created
```

我们来看如下的范例 SQL 3：

```
select t1.mgr, t2.deptno
  from emp_temp t1, emp_temp1 t2
 where t1.empno = t2.empno;
```

对于范例 SQL 3 而言，表 EMP_TEMP 和 EMP_TEMP1 唯一的表连接条件为“t1.empno = t2.empno”，而在表 EMP_TEMP 和 EMP_TEMP1 的字段 EMPNO 上均没有任何索引，按照前面介绍的知识，表 EMP_TEMP1 在 SQL 文本中的位置是在表 EMP_TEMP 的右边，所以此时 RBO 会将表 EMP_TEMP1 作为表连接的驱动表，而将表 EMP_TEMP 作为表连接的被驱动表。

执行一下范例 SQL 3：

```
SQL> select t1.mgr,t2.deptno from emp_temp t1,emp_temp1 t2 where t1.empno=t2.empno;
执行计划
----------------------------------------------------------
Plan hash value: 1323777565

-----------------------------------------
| Id  | Operation           | Name      |
-----------------------------------------
|   0 | SELECT STATEMENT    |           |
|   1 |  MERGE JOIN         |           |
|   2 |   SORT JOIN         |           |
|   3 |    TABLE ACCESS FULL| EMP_TEMP1 |
|*  4 |   SORT JOIN         |           |
|   5 |    TABLE ACCESS FULL| EMP_TEMP  |
-----------------------------------------

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

   4 - access("T1"."EMPNO"="T2"."EMPNO")
       filter("T1"."EMPNO"="T2"."EMPNO")

Note
-----
   - rule based optimizer used (consider using cbo)
```

从上面显示的内容可以看出，现在范例 SQL 3 的执行计划走的是排序合并连接，且驱动表确实是表 EMP_TEMP1。

注意，从严格意义上来说，排序合并连接并没有驱动表和被驱动表的概念，这里只是为了方便阐述而人为地给排序合并连接添加了上述概念。

将范例 SQL 3 中的表 EMP_TEMP 和 EMP_TEMP1 在该 SQL 的 SQL 文本中的位置换一下，即形成了如下形式的范例 SQL 4：

```
select t1.mgr, t2.deptno
  from emp_temp1 t2, emp_temp t1
 where t1.empno = t2.empno;
```

按照前面介绍的知识，现在如果再执行范例 SQL 4 的话，那么排序合并连接的驱动表应该会变成表 EMP_TEMP。

我们来验证一下。执行范例 SQL 4：

```
SQL> select t1.mgr,t2.deptno from emp_temp1 t2,emp_temp t1 where t1.empno=t2.empno;
执行计划
----------------------------------------------------------
Plan hash value: 2135683657

-----------------------------------------
| Id  | Operation           | Name      |
-----------------------------------------
|   0 | SELECT STATEMENT    |           |
|   1 |  MERGE JOIN         |           |
|   2 |   SORT JOIN         |           |
|   3 |    TABLE ACCESS FULL| EMP_TEMP  |
|*  4 |   SORT JOIN         |           |
|   5 |    TABLE ACCESS FULL| EMP_TEMP1 |
-----------------------------------------
```

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

   4 - access("T1"."EMPNO"="T2"."EMPNO")
       filter("T1"."EMPNO"="T2"."EMPNO")

Note
-----
   - rule based optimizer used (consider using cbo)
```

从上面显示的内容可以看出，现在范例 SQL 4 的执行计划走的也是排序合并连接，且驱动表确实已经由之前的表 EMP_TEMP1 变为了现在的表 EMP_TEMP。这说明我们确实使 RBO 改变了执行计划，**也说明当目标 SQL 有两条或者两条以上的执行路径的等级值相同时，我们确实可以通过改变目标 SQL 中所涉及的各个对象在该 SQL 文本中出现的先后顺序来影响 RBO 对于其执行计划的选择**。

注意，这种位置的先后顺序对于目标 SQL 执行计划的影响是有前提条件的，那就是仅凭各条执行路径等级值的大小 RBO 难以选择执行计划，也就是说该目标 SQL 一定有两条或者两条以上执行路径的等级值相同。换句话说，**如果 RBO 仅凭各条执行路径等级值的大小就可以选择目标 SQL 的执行计划，那么无论怎么调整相关对象在该 SQL 的 SQL 文本中的位置，对于该 SQL 最终的执行计划都不会有任何影响**。

我们来验证一下上述结论。看看如下的范例 SQL 5：

```
select t1.mgr, t2.deptno
  from emp t1, emp_temp t2
 where t1.empno = t2.empno;
```

对于范例 SQL 5 而言，表 EMP 和 EMP_TEMP 唯一的表连接条件为“t1.empno = t2.empno”。对于表 EMP 而言，列 EMPNO 上存在主键索引 PK_EMP，而对于表 EMP_TEMP 而言，列 EMPNO 上不存在任何索引。所以在使用 RBO 的情况下，范例 SQL 5 的执行路径将不再仅限于排序合并连接（RBO 不支持哈希连接），也就是说 RBO 此时有可能可以仅凭各条执行路径等级值的大小就选择出范例 SQL 5 的执行计划。

执行一下范例 SQL 5：

```
SQL> select t1.mgr,t2.deptno from emp t1,emp_temp t2 where t1.empno=t2.empno;
执行计划
----------------------------------------------------------
Plan hash value: 367190759

-----------------------------------------------
| Id  | Operation                    | Name     |
-----------------------------------------------
|   0 | SELECT STATEMENT             |          |
|   1 |  NESTED LOOPS                |          |
|   2 |   NESTED LOOPS               |          |
|   3 |    TABLE ACCESS FULL         | EMP_TEMP |
|*  4 |    INDEX UNIQUE SCAN         | PK_EMP   |
|   5 |   TABLE ACCESS BY INDEX ROWID| EMP      |
-----------------------------------------------

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

   4 - access("T1"."EMPNO"="T2"."EMPNO")

Note
-----
   - rule based optimizer used (consider using cbo)
```

从上面显示的内容可以看出，现在范例 SQL 5 的执行计划走的是嵌套循环连接，且驱动表是表 EMP_TEMP。

我们将范例 SQL 5 中的表 EMP 和 EMP_TEMP 在该 SQL 的 SQL 文本中的位置换一下，即形成了如下形式的范例 SQL 6：

```
select t1.mgr, t2.deptno
  from emp_temp t2, emp t1
 where t1.empno = t2.empno;
```

然后执行范例 SQL 6：

```
SQL> select t1.mgr,t2.deptno from emp_temp t2,emp t1 where t1.empno=t2.empno;
执行计划
----------------------------------------------------------
Plan hash value: 367190759

-----------------------------------------------
| Id  | Operation                    | Name     |
-----------------------------------------------
|   0 | SELECT STATEMENT             |          |
|   1 |  NESTED LOOPS                |          |
|   2 |   NESTED LOOPS               |          |
|   3 |    TABLE ACCESS FULL         | EMP_TEMP |
|*  4 |    INDEX UNIQUE SCAN         | PK_EMP   |
|   5 |   TABLE ACCESS BY INDEX ROWID| EMP      |
-----------------------------------------------

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

   4 - access("T1"."EMPNO"="T2"."EMPNO")

Note
-----
   - rule based optimizer used (consider using cbo)
```

从上面显示的内容可以看出，现在范例 SQL 6 的执行计划走的还是嵌套循环连接，且驱动表依然是表 EMP_TEMP。这就验证了我们之前提到的观点：如果 RBO 仅凭目标 SQL 各条执行路径等级值的大小就可以选择出执行计划，那么无论怎么调整相关对象在该 SQL 的 SQL 文本中的位置，对于该 SQL 最终的执行计划都不会有任何影响。

1.1.2　基于成本的优化器

我们在 1.1.1 节中已经提到：RBO 是有明显缺陷的，比如 Oracle 数据库中很多很好的功能、特性，RBO 均不支持，RBO 产生的执行计划很难调整等，但这些还不是最要命的，RBO 最大的问题在于它是靠硬编码在 Oracle 数据库代码中的一系列固定的规则来决定目标 SQL 的执行计划的，而并没有考虑目标 SQL 中所涉及的对象的实际数据量、实际数据分布等情况，这样一旦固定的规则并不适用于该 SQL 中所涉及的实际对象时，RBO 根据固定规则产生的执行计划就很可能不是当前情况下的最优执行计划了。

我们来看如下的范例 SQL 7：

```
select * from emp
  where mgr=7902;
```

对于范例 SQL 7 而言，假设在表 EMP 的列 MGR 上事先存在一个名为 IDX_EMP_MGR 的单键值 B 树索引，如果我们使用 RBO，则不管表 EMP 的数据量有多大，也不管列 MGR 的数据分布情况如何，Oracle 在执行范例 SQL 7 时始终会选择走对索引 IDX_EMP_MGR 的索引范围扫描，并回表取得表 EMP 中的记录。Oracle 此时是不会选择全表扫描表 EMP 的，因为对于 RBO 而言，全表扫描的等级值要高于索引范围扫描的等级值。

RBO 的这种选择在表 EMP 的数据量不大，或者虽然表 EMP 的数据量很大，但满足条件“mgr=7902”的记录数很少时是没问题的。如果出现了极端的情况（比如表 EMP 的数据量很大，有 1000 万行记录，且这 1000 万行记录的列 MGR 的值均等于 7902），当出现这种极端情况时，如果使用 RBO，则 RBO 还是会选择走对索引 IDX_EMP_MGR 的索引范围扫描，那就有问题了！因为这相当于要以单块读顺序扫描所有的 1000 万行索引，然后再回表 1000 万次，而这显然是没有使用多块读以全表扫描方式直接扫描表 EMP 的执行效率高的（这里的 1000 万只是一个理论值，实际情况并不完全是这样，因为这里并没有考虑 Index Prefetch 所带来的扫描索引时可能会使用的多块读。不考虑 Index Prefetch 的原因是因为它的存在与否对这里的结论并不会产生本质的影响）。这里 RBO 会选错执行计划就是因为它并没有考虑目标 SQL 中所涉及的对象的实际数据量、实际数据分布等情况，所以 RBO 确实是有先天缺陷的。

为了解决 RBO 的上述先天缺陷，从 Oracle 7 开始，Oracle 就引入了 CBO。之前已经提到过，CBO 在选择目标 SQL 的执行计划时，所用的判断原则为成本，CBO 会从目标 SQL 诸多可能的执行路径中选择一条成本值最小的执行路径来作为其执行计划，各条执行路径的成本值是根据目标 SQL 语句所涉及的表、索引、列等相关对象的统计信息计算出来的。

这里的统计信息是这样的一组数据：它们存储在 Oracle 数据库的数据字典里，且从多个维度描述了 Oracle 数据库里相关对象的实际数据量、实际数据分布等详细信息（关于统计信息，会在“第 5 章 Oracle 里的统计信息”中详细描述）。

这里的成本是指 Oracle 根据相关对象的统计信息计算出来的一个值，它实际上代表了 Oracle 根据相关统计信息估算出来的目标 SQL 的对应执行步骤的 I/O、CPU 和网络资源的消耗量，这也就意味着 Oracle 数据库里的成本实际上就是对执行目标 SQL 所要耗费的 I/O、CPU 和网络资源的一个估算值。

Oracle 在执行目标 SQL 时需要耗费 I/O 和 CPU，这很容易理解，但这里的网络资源消耗是指什么？实际上，这里的网络资源消耗适用于那些使用了 dblink 的分布式目标 SQL，CBO 在解析该类 SQL 时知道在实际执行它们时所需要的数据并不全部在本地数据库中（需要去远程数据库中取数据），所以此时的网络资源消耗就会被 CBO 考虑在内。这里需要注意的是，**Oracle 会把解析这种分布式目标 SQL 所需要考虑的网络资源消耗折算成对等的 I/O 资源消耗，所以实际上你可以认为 Oracle 数据库里的成本仅仅依赖于执行目标 SQL 时所需要耗费的 I/O 和 CPU 资源。另外需要注意的是，在 Oracle 未引入系统统计信息之前，CBO 所计算的成本值实际上全部是基于 I/O 来估算的，只有在 Oracle 引入了系统统计信息之后，CBO 所计算的成本值才真正依赖于目标 SQL 的 I/O 和 CPU 消耗**（关于系统统计信息，会在“第 5 章 Oracle 里的统计信息”中详细描述）。

从上述对 CBO 的介绍中我们可以看出：CBO 会从目标 SQL 诸多可能的执行路径中选择一条成本值最小的执行路径来作为其执行计划，**这也就意味着 CBO 会认为那些消耗系统 I/O 和 CPU 资源最少的执行路径就是当前情况下的最佳选择**。注意，这里的“消耗系统 I/O 和 CPU 资源”（即成本）的计算方法会随着优化器模式的不同而不同，这一点在“1.2.1 优化器的模式”中会详细说明。

CBO 在解析目标 SQL 时，首先会对目标 SQL 执行查询转换（关于查询转换，我们会在“第 4 章 Oracle 里的查询转换”中详细说明）；接下来，CBO 会计算执行完查询转换这一步后得到的等价改写 SQL 的诸多可能的执行路径的成本，然后从上述诸多可能的执行路径中选择成本值最小的一条来作为原目标 SQL 的执行计划；在得到了目标 SQL 的执行计划后，接下来 Oracle 就会根据此执行计划去实际执行该 SQL，并将执行结果返回给用户。这里需要说明的是，Oracle 在对一条执行路径计算成本时，并不一定会从头到尾完整计算完，只要 Oracle 在计算过程中发现算出来的部分成本值已经大于之前保存下来的到目前为止的最小成本值，就会马上中止对当前执行路径成本值的计算，并转而开始计算下一条新的执行路径的成本。这个过程会一直持续下去，

直到目标 SQL 的各个可能的执行路径全部计算完毕或已达到预先定义好的待计算的执行路径数量的阈值。

接下来，介绍与 CBO 相关的一些基本概念。

1.1.2.1　集的势

Cardinality 是 CBO 特有的概念，直译过来就是“集的势”，它是指指定集合所包含的记录数，说白了就是指定结果集的行数。这个指定结果集是与目标 SQL 执行计划的某个具体执行步骤相对应的，也就是说**Cardinality 实际上表示对目标 SQL 的某个具体执行步骤的执行结果所包含记录数的估算。当然，如果是针对整个目标 SQL，那么此时的 Cardinality 就表示对该 SQL 最终执行结果所包含记录数的估算。**

Cardinality 和成本值的估算是息息相关的，因为 Oracle 得到指定结果集所需要耗费的 I/O 资源可以近似看作随着该结果集所包含记录数的递增而递增，所以**某个执行步骤所对应的 Cardinality 的值越大，那么它所对应的成本值往往也就越大，这个执行步骤所在执行路径的总成本值也就会越大。**

1.1.2.2　可选择率

可选择率（Selectivity）也是 CBO 特有的概念，它是指施加指定谓词条件后返回结果集的记录数占未施加任何谓词条件的原始结果集的记录数的比率。

可选择率可以用如下的公式来表示：

$$Selectivity = \frac{施加指定谓词条件后返回结果集的记录数}{未施加任何谓词条件的原始结果集的记录数}$$

从上述计算可选择率的公式可以看出，可选择率的取值范围显然是 0～1，它的值越小，就表明可选择性越好。毫无疑问，可选择率为 1 时的可选择性是最差的。

可选择率和成本值的估算也是息息相关的，因为**可选择率的值越大，就意味着返回结果集的 Cardinality 的值就越大，所以估算出来的成本值也就会越大。**

实际上，CBO 就是用可选择率来估算对应结果集的 Cardinality 的，上述关于可选择率的计算公式等价转换后就可以用来估算 Cardinality 的值。这里我们用“Original Cardinality”来表示未施加任何谓词条件的原始结果集的记录数，用“Computed Cardinality”来表示施加指定谓词条件后返回结果集的记录数，CBO 用来估算 Cardinality 的公式如下：

```
Computed Cardinality = Original Cardinality * Selectivity
```

虽然看起来可选择率的计算公式很简单，但实际上它的具体计算过程还是很复杂的，每一种具体情况都会有不同的计算公式。其中最简单的情况是对目标列做等值查询时可选择率的计算。在目标列上没有直方图且没有 NULL 值的情况下，用目标列做等值查询的可选择率是用如下公式来计算的：

$$Selectivity = \frac{1}{NUM\_DISTINCT}$$

```
//这里的 NUM_DISTINCT 表示目标列的 distinct 值的数量
```

我们现在再回过头来看 1.1.2 节中提到的范例 SQL 7：

```
select * from emp
where mgr=7902;
```

对于范例 SQL 7，我们来看一下 CBO 会如何计算列 MGR 的可选择率和该 SQL 返回结果集的 Cardinality。

先把列 MGR 修改为 NOT NULL：

```
SQL> alter table emp modify (mgr not null);

Table altered
```

然后在列 MGR 上创建一个名为 IDX_EMP_MGR 的单键值 B 树索引：

```
SQL> create index idx_emp_mgr on emp(mgr);

Index created
```

表 EMP 的记录数现在为 13：

```
SQL> select count(*) from emp;
  COUNT(*)
----------
        13
```

列 MGR 的 distinct 值的数量也为 13：

```
SQL> select count(distinct mgr) from emp;
COUNT(DISTINCTMGR)
------------------
                13
```

现在使用 DBMS_STATS 包来对表 EMP、表 EMP 的所有列、表 EMP 上的所有索引收集一下统计信息（注意，这里没有收集直方图统计信息，关于 DBMS_STATS 包的用法，我们会在“第 5 章 Oracle 里的统计信息”中详细说明）：

```
SQL> exec dbms_stats.gather_table_stats(ownname => 'SCOTT',tabname => 'EMP',estimate_percent
=> 100,cascade => true, method_opt=>'for all columns size 1',no_invalidate => false);

PL/SQL procedure successfully completed
```

接着执行范例 SQL 7：

```
SQL> set linesize 800
SQL> set pagesize 900
SQL> set autotrace traceonly
SQL> select * from emp where mgr=7902;
执行计划
----------------------------------------------------------
Plan hash value: 2059184959

-------------------------------------------------------------------------------------------
| Id  | Operation                    | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |             |     1 |    38 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMP         |     1 |    38 |     2   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN           | IDX_EMP_MGR |     1 |       |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------
```

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

   2 - access("MGR"=7902)
```

……省略显示部分内容

从 Oracle 10g 开始，Oracle 在解析目标 SQL 时就会默认使用 CBO。注意到上述执行计划的显示内容中有列 Rows 和列 Cost (%CPU)，这说明 Oracle 在解析范例 SQL 7 时确实使用的是 CBO。这里列 Rows 记录的就是上述执行计划中的每一个执行步骤所对应的 Cardinality 的值，列 Cost (%CPU) 记录的就是上述执行计划中的每一个执行步骤所对应的成本值。

从上面显示的内容可以看出，现在范例 SQL 7 的执行计划走的是对索引 IDX_EMP_MGR 的索引范围扫描。注意，Id = 2 的执行步骤所对应的列 Rows 的值为 1，这说明 CBO 评估出来以驱动查询条件“access("MGR"=7902)”去访问索引 IDX_EMP_MGR 时返回结果集的 Cardinality 的值是 1；另外，Id = 0 的执行步骤所对应的列 Rows 的值也为 1，这说明 CBO 评估出来的范例 SQL 7 的最终执行结果所对应的 Cardinality 的值也是 1。

这两个值 CBO 是如何算出来的呢？

之前提到过：在目标列上没有直方图且没有 NULL 值的情况下，用目标列做等值查询的可选择率的计算公式为 Selectivity = (1 / NUM_DISTINCT)。现在列 MGR 没有 NULL 值也没有直方图统计信息，范例 SQL 7 的 where 条件是针对列 MGR 的等值查询（等值查询条件为“mgr=7902”），而列 MGR 的 distinct 值的数量是 13，所以此时针对列 MGR 做等值查询的可选择率就是 1/13。另外，之前也提到 Cardinality 的计算公式为 Computed Cardinality = Original Cardinality * Selectivity，表 EMP 的记录数为 13，即此时 Original Cardinality 的值为 13，那么根据 Cardinality 的计算公式，上述针对列 MGR 做等值查询的执行步骤所对应的 Cardinality 的值就是 13 * 1/13 = 1，所以这就是 CBO 评估出来以驱动查询条件“access("MGR"=7902)”去访问索引 IDX_EMP_MGR 时返回结果集的 Cardinality 的值为 1 的原因。又因为 where 条件“mgr=7902”是范例 SQL 7 的唯一查询条件，所以范例 SQL 7 的最终执行结果所对应的 Cardinality 的值也会是 1。

我们现在把列 MGR 的值全部修改为 7,902：

```
SQL> update emp set mgr=7902;

13 rows updated

SQL> commit;

Commit complete
```

然后重新收集一下统计信息：

```
SQL> exec dbms_stats.gather_table_stats(ownname => 'SCOTT',tabname => 'EMP',estimate_percent
=> 100,cascade => true, method_opt=>'for all columns size 1',no_invalidate => false);

PL/SQL procedure successfully completed
```

接着重新执行范例 SQL 7：

```
SQL> select * from emp where mgr=7902;
```

```
已选择13行。

执行计划
----------------------------------------------------------
Plan hash value: 2059184959

-------------------------------------------------------------------------------------------
| Id  | Operation                    | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |             |    13 |   494 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMP         |    13 |   494 |     2   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN           | IDX_EMP_MGR |    13 |       |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------

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

   2 - access("MGR"=7902)

……省略显示部分内容
```

从上述显示内容可以看出，现在范例 SQL 7 的执行计划走的依然是对索引 IDX_EMP_MGR 的索引范围扫描，只不过现在 CBO 评估出来以驱动查询条件“access("MGR"=7902)”去访问索引 IDX_EMP_MGR 时返回结果集的 Cardinality 和最终执行结果所对应的 Cardinality 的值均已从之前的 1 变为了现在的 13。

这是很容易理解的。现在表 EMP 总的记录数还是 13，但列 MGR 的 distinct 值的数量已经从之前的 13 变为了 1（即针对列 MGR 做等值查询的可选择率已经从之前的 1/13 变为了 1），所以现在针对列 MGR 做等值查询的执行步骤所对应的 Cardinality 和最终执行结果所对应的 Cardinality 的值就都会是 13 * 1/1 = 13。

我们现在来构造之前在 1.1.2 节中提到的那种极端情况（表 EMP 的数据量为 1000 万行，且这 1000 万行记录的列 MGR 的值均等于 7,902）。注意，这里并不用真正往表 EMP 里插入 1000 万行记录，只需要让 CBO 认为表 EMP 的数据量为 1000 万行就可以了（因为 CBO 计算成本时完全基于目标 SQL 的相关对象的统计信息，所以这里我们只需要改一下表 EMP 和索引 IDX_EMP_MGR 的统计信息，就可以让 CBO 认为表 EMP 的数据量是 1000 万行了）：

使用 DBMS_STATS 包将表 EMP 对应其数据量的统计信息修改为 1000 万：

```
SQL> exec dbms_stats.set_table_stats(ownname => 'SCOTT',tabname => 'EMP',numrows => 10000000,no_invalidate => false);

PL/SQL procedure successfully completed
```

然后再将索引 IDX_EMP_MGR 对应其索引叶子块数量的统计信息修改为 10 万：

```
SQL> exec dbms_stats.set_index_stats(ownname => 'SCOTT',indname => 'IDX_EMP_MGR',numlblks => 100000,no_invalidate => false);

PL/SQL procedure successfully completed
```

再次执行范例 SQL 7：

```
SQL> select * from emp where mgr=7902;
已选择13行。

执行计划
----------------------------------------------------------
Plan hash value: 3956160932
```

```
-------------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |   10M|   362M|   429  (97)| 00:00:05 |
|*  1 |  TABLE ACCESS FULL| EMP  |   10M|   362M|   429  (97)| 00:00:05 |
-------------------------------------------------------------------------------

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

   1 - filter("MGR"=7902)
```

……省略显示部分内容

从上面显示的内容中我们可以看出，范例 SQL 7 的执行计划已经从之前的走对索引 IDX_EMP_MGR 的索引范围扫描变为了现在的对表 EMP 的全表扫描，并且针对列 MGR 做等值查询的执行步骤所对应的 Cardinality 和最终执行结果所对应的 Cardinality 的值已经从之前的 13 变为了现在的“10M”（即 1000 万）。这就契合了我们之前提到的观点：如果出现了上述这种极端的情况，CBO 肯定会选择全表扫描。

这里为什么 Cardinality 的值会变成 1000 万呢？因为表 EMP 的记录数（即 Original Cardinality）在 CBO 的眼里由之前的 13 变为了现在的 1000 万，而 Selectivity 的值还是 1，所以最后 CBO 估算出来的 Cardinality 的值就从之前的 13 变为了现在的 1000 万（这里用到的计算公式还是之前提到的 Computed Cardinality = Original Cardinality * Selectivity）。

现在我们再来看一下在上述这种极端情况下 RBO 的选择。在当前 Session 中将优化器模式修改为 RULE，这表示在当前 Session 中启用 RBO：

```
SQL> alter session set optimizer_mode=rule;


Session altered
```

然后再次执行范例 SQL 7：

```
SQL> select * from emp where mgr=7902;
已选择13行。

执行计划
----------------------------------------------------------
Plan hash value: 2059184959

-------------------------------------------------
| Id  | Operation                   | Name        |
-------------------------------------------------
|   0 | SELECT STATEMENT            |             |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMP         |
|*  2 |   INDEX RANGE SCAN          | IDX_EMP_MGR |
-------------------------------------------------

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

   2 - access("MGR"=7902)

Note
-----
   - rule based optimizer used (consider using cbo)
```

……省略显示部分内容

从上面显示的内容中我们可以看出，范例 SQL 7 的执行计划走的还是对索引 IDX_EMP_MGR 的索引范围扫描，这也契合了我们之前提到的观点：如果出现了上述这种极端的情况，RBO 还是会选择走对索引 IDX_EMP_MGR 的索引范围扫描。

从对范例 SQL 7 的实际执行过程我们可以得到如下结论。

（1）RBO 确实是靠硬编码在 Oracle 数据库代码中的一系列固定的规则来决定目标 SQL 的执行计划的，并没有考虑目标 SQL 中所涉及的对象的实际数据量、实际数据分布等情况。而 CBO 则恰恰相反，CBO 会根据反映目标 SQL 中相关对象的实际数据量、实际数据分布等情况的统计信息来决定其执行计划，这就意味着 CBO 选择的执行计划可能会随着目标 SQL 中所涉及的对象的统计信息的变化而变化。CBO 的这种变化是颠覆性的，这意味着只要统计信息相对准确，则用 CBO 来解析目标 SQL 会比在同等条件下用 RBO 来解析得到正确执行计划的概率要高。

（2）Cardinality 和 Selectivity 的值会直接影响 CBO 对于相关执行步骤成本值的估算，进而影响 CBO 对于目标 SQL 执行计划的选择。

1.1.2.3 可传递性

可传递性（Transitivity）也是 CBO 特有的概念，它是 CBO 在图 1-1 的查询转换中所做的第一件事情，其含义是指 CBO 可能会对原目标 SQL 做简单的等价改写，即在原目标 SQL 中加上根据该 SQL 现有的谓词条件推算出来的新的谓词条件，这么做的目的是提供更多的执行路径给 CBO 做选择，进而增加得到更高效执行计划的可能性。这里需要注意的是，**利用可传递性对目标 SQL 做简单的等价改写仅仅适用于 CBO，RBO 不会做这样的事情**。

在 Oracle 里，可传递性又分为如下这三种情形。

1．简单谓词传递

比如原目标 SQL 中的谓词条件是“t1.c1=t2.c1 and t1.c1=10”，则 CBO 可能会在这个谓词条件中额外地加上“t2.c1=10”，即 CBO 可能会将原谓词条件“t1.c1=t2.c1 and t1.c1=10”修改为“t1.c1=t2.c1 and t1.c1=10 and t2.c1=10”。改写前后的谓词条件显然是等价的，因为如果 t1.c1=t2.c1 且 t1.c1=10，那么我们就可以推算出 t2.c1 也等于 10。

2．连接谓词传递

比如原目标 SQL 中的谓词条件是“t1.c1=t2.c1 and t2.c1=t3.c1”，则 CBO 可能会在这个谓词条件中额外地加上“t1.c1=t3.c1”，即 CBO 可能会将原谓词条件“t1.c1=t2.c1 and t2.c1=t3.c1”修改为“t1.c1=t2.c1 and t2.c1=t3.c1 and t1.c1=t3.c1”，同理，这里改写前后的谓词条件也是等价的。

3．外连接谓词传递

比如原目标 SQL 中的谓词条件是“t1.c1=t2.c1(+) and t1.c1=10”，则 CBO 可能会在这个谓词条件中额外加上“t2.c1(+)=10”，即 CBO 可能会将原谓词条件“t1.c1=t2.c1(+) and t1.c1=10”修改为“t1.c1=t2.c1(+) and t1.c1=10 and t2.c1(+)=10”。关于外连接及上述 SQL 中关键字“(+)”的含义，我们会在“1.2.4.1.2 外连接”中详细描述。

之前已经提到过：Oracle 利用可传递性对目标 SQL 做简单的等价改写的目的是为了提供更多的执行路径给 CBO 做选择，进而增加得到更高效执行计划的可能性。我们现在来看一个 CBO 利用可传递性对目标 SQL 做简单等价改写的实例：

创建两个测试表 T1 和 T2：

```
SQL> create table t1(c1 number,c2 varchar2(10));
```

```
Table created

SQL> create table t2(c1 number,c2 varchar2(10));

Table created
```

在表 T2 的列 C1 上创建一个名为 IDX_T2 的索引：

```
SQL> create index idx_t2 on t2(c1);

Index created
```

往表 T1 和 T2 中各插入一些数据，然后我们来看如下的范例 SQL 8：

```
select t1.c1,t2.c2 from t1, t2
 where t1.c1 = t2.c1 and t1.c1 = 10;
```

上述范例 SQL 8 的 where 条件是“t1.c1 = t2.c1 and t1.c1 = 10”，并没有针对表 T2 的列 C1 的简单谓词条件，所以按道理讲应该是不能走我们刚才在表 T2 的列 C1 上建的索引 IDX_T2 的。

但实际情况是否如此呢？我们来执行一下范例 SQL 8：

```
SQL> select t1.c1,t2.c2 from t1, t2
  2   where t1.c1 = t2.c1 and t1.c1 = 10;
未选定行

执行计划
----------------------------------------------------------
Plan hash value: 2918692618

-------------------------------------------------------------------------------------------
| Id  | Operation                     | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |         |     1 |     9 |    14   (0)| 00:00:01 |
|   1 |  MERGE JOIN CARTESIAN         |         |     1 |     9 |    14   (0)| 00:00:01 |
|*  2 |   TABLE ACCESS FULL           | T1      |     1 |     3 |    13   (0)| 00:00:01 |
|   3 |   BUFFER SORT                 |         |     1 |     6 |     1   (0)| 00:00:01 |
|   4 |    TABLE ACCESS BY INDEX ROWID| T2      |     1 |     6 |     1   (0)| 00:00:01 |
|*  5 |     INDEX RANGE SCAN          | IDX_T2  |     1 |       |     0   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------

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

   2 - filter("T1"."C1"=10)
   5 - access("T2"."C1"=10)

……省略显示部分内容
```

上面显示的内容中 Id = 5 的执行步骤为“INDEX RANGE SCAN | IDX_T2”，这说明 Oracle 现在还是走了对索引 IDX_T2 的索引范围扫描。为什么 Oracle 能够这样做？

注意到 Id = 5 的执行步骤所对应的驱动查询条件为“access("T2"."C1"=10)”，这说明 Oracle 在访问索引 IDX_T2 时用的驱动查询条件是“t2.c1=10”，但这个“t2.c1=10 ”在范例 SQL 8 的原始 SQL 文本中并不存在。这就说明 CBO 此时确实利用可传递性对范例 SQL 8 做了简单等价改写，即 CBO 此时已经将范例 SQL 8 改写成了如下的等价形式：

```
select t1.c1,t2.c2 from t1, t2
  where t1.c1 = t2.c1 and t1.c1 = 10 and t2.c1 = 10;
```

这样做的好处是显而易见的——正是因为上述额外多出来的谓词条件“and t2.c1 = 10”，CBO 在解析范例 SQL 8 时就多出了走索引 IDX_T2 和对应的执行路径这种选择，进而就增加了得到更高效执行计划的可能性。

1.1.2.4 CBO 的局限性

CBO 诞生的初衷是为了解决 RBO 的先天缺陷，并且随着 Oracle 数据库版本的不断进化，CBO 也越来越智能，越来越强悍，但这并不意味着 CBO 就完美无瑕，没有任何缺陷了。这个世界上并没有完美的事情，CBO 同样如此。

实际上，CBO 的缺陷（或者说局限性）至少表现在如下几个方面。

1．CBO 会默认目标 SQL 语句 where 条件中出现的各个列之间是独立的，没有关联关系

CBO 会默认目标 SQL 语句 where 条件中出现的各个列之间是独立的，没有关联关系，并且 CBO 会依据这个前提条件来计算组合可选择率、Cardinality，进而来估算成本并选择执行计划。但这种前提条件并不总是正确的，在实际的应用中，目标 SQL 的各列之间有关联关系的情况实际上并不罕见。在这种各列之间有关联关系的情况下，如果还用之前的计算方法来计算目标 SQL 语句整个 where 条件的组合可选择率，并用它来估算返回结果集的 Cardinality 的话，那么估算结果可能就会和实际结果有较大的偏差，导致 CBO 选错执行计划。

目前可以用来缓解上述问题所带来负面影响的方法是使用动态采样或者多列统计信息，**但动态采样的准确性取决于采样数据的质量和采样数据的数量，而多列统计信息并不适用于多表之间有关联关系的情形，所以这两种解决方法都不能算是完美的解决方案**。关于动态采样和多列统计信息，我们会在的“5.7　动态采样”和“5.8　多列统计信息”中分别予以详细说明。

2．CBO 会假设所有的目标 SQL 都是单独执行的，并且互不干扰

CBO 会假设所有的目标 SQL 都是单独执行、并且是互不干扰的，但实际情况却完全不是这样。我们执行目标 SQL 时所需要访问的索引叶子块、数据块等可能由于之前执行的 SQL 而已经被缓存在 Buffer Cache 中，所以这次执行时也许不需要耗费物理 I/O 去相关的存储上读要访问的索引叶子块、数据块等，而只需要去 Buffer Cache 中读相关的缓存块就可以了。所以，如果此时 CBO 还是按照目标 SQL 是单独执行，不考虑缓存的方式去计算相关成本值的话，就可能会高估走相关索引的成本，进而可能会导致选错执行计划。

3．CBO 对直方图统计信息有诸多限制

CBO 对直方图统计信息的限制体现在如下两个方面。

（1）在 Oracle 12c 之前，Frequency 类型的直方图所对应的 Bucket 的数量不能超过 254，这样如果目标列的 distinct 值的数量超过 254，Oracle 就会使用 Height Balanced 类型的直方图。对于 Height Balanced 类型的直方图而言，因为 Oracle 不会记录所有的 nonpopular value 的值，所以在此情况下 CBO 选错执行计划的概率会比对应的直方图统计信息是 Frequency 类型的情形要高。

（2）在 Oracle 数据库里，如果针对文本型的字段收集直方图统计信息，则 Oracle 只会将该文本型字段的文本值的头 32 字节给取出来（实际上只取头 15 字节）并将其转换成一个浮点数，然后将该浮点数作为上述文本型字段的直方图统计信息存储在数据字典里。这种处理机制的先天缺陷就在于，对于那些超过 32 字节的文本型字段，只要对应记录的文本值的头 32 字节相同，Oracle 在收集直方图统计信息的时候就会认为这些记录该字段的文本值是相同的，即使实际上它们并不相同。这种先天性的缺陷会直接影响 CBO 对相关文本型字段的可选择率及返回结果集的 Cardinality 的估算，进而就可能导致 CBO 选错执行计划。

我们会在第 5 章的“5.5.3 直方图”中对上述两个限制予以详细说明，这里不再赘述。

4．CBO 在解析多表关联的目标 SQL 时，可能会漏选正确的执行计划

在解析多表关联的目标 SQL 时，虽然 CBO 会采取多种手段来避免漏选正确的执行计划，但是这种漏选往往难以完全避免。因为随着多表关联的目标 SQL 所包含表的数量的递增，各表之间可能的连接顺序会呈几何级数增长，即该 SQL 各种可能的执行路径的总数也会随之呈几何级数增长。

假设多表关联的目标 SQL 所包含表的数量为 *n*，则该 SQL 各表之间可能的连接顺序的总数就是 *n*!（*n* 的阶乘）。这意味着包含 10 个表的目标 SQL 各表之间可能的连接顺序总数为 3,628,800，包含 15 个表的目标 SQL 各表之间可能的连接顺序总数为 1,307,674,368,000。

```
SQL> select 10*9*8*7*6*5*4*3*2*1 from dual;

10*9*8*7*6*5*4*3*2*1
--------------------
    3628800
SQL> select 15*14*13*12*11*10*9*8*7*6*5*4*3*2*1 from dual;

15*14*13*12*11*10*9*8*7*6*5*4*
------------------------------
    1307674368000
```

包含 15 个表的多表关联的目标 SQL 在实际的应用系统中并不罕见，显然 CBO 在处理这种类型的目标 SQL 时是不可能遍历其所有可能的情形的，否则解析该 SQL 的时间将会变得不可接受。

在 Oracle 11gR2 中，CBO 在解析这种多表关联的目标 SQL 时，所考虑的各个表连接顺序的总和会受隐含参数_OPTIMIZER_MAX_PERMUTATIONS 的限制，这意味着不管目标 SQL 在理论上有多少种可能的连接顺序，CBO 至多只会考虑其中根据_OPTIMIZER_MAX_PERMUTATIONS 计算出来的有限种可能。这同时也意味着只要该目标 SQL 正确的执行计划并不在上述有限种可能之中，则 CBO 一定会漏选正确的执行计划。

虽然有上述这些局限性，但是瑕不掩瑜，CBO 毫无疑问是当前情形下 Oracle 中解析目标 SQL 的不二选择，并且我们完全有理由相信随着 Oracle 数据库版本不断的进化，CBO 也会越来越完善。

1.2　优化器的基础知识

接下来，介绍一些优化器的基础知识，这些基础知识中的绝大部分内容与优化器的类型是没有关系的，也就是说它们中的绝大部分内容不仅适用于 CBO，同样也适用于 RBO。

1.2.1　优化器的模式

优化器的模式用于决定在 Oracle 中解析目标 SQL 时所用优化器的类型，以及决定当使用 CBO 时计算成本值的侧重点。这里的“侧重点”是指当使用 CBO 来计算目标 SQL 各条执行路径的成本值时，计算成本值的方法会随着优化器模式的不同而不同。

在 Oracle 数据库中，优化器的模式是由参数 OPTIMIZER_MODE 的值来决定的，OPTIMIZER_MODE 的值可能是 RULE、CHOOSE、FIRST_ROWS_n（n = 1, 10, 100, 1000）、FIRST_ROWS 或 ALL_ROWS。

OPTIMIZER_MODE 的各个可能的值的含义为如下所示。

1．RULE

RULE 表示 Oracle 将使用 RBO 来解析目标 SQL，此时目标 SQL 中所涉及的各个对象的统计信息对于 RBO 来说将没有任何作用。

2．CHOOSE

CHOOSE 是 Oracle 9i 中 OPTIMIZER_MODE 的默认值，它表示 Oracle 在解析目标 SQL 时到底是使用 RBO 还是使用 CBO 取决于该 SQL 中所涉及的表对象是否有统计信息。具体来说就是：只要该 SQL 中所涉及的表对象中有一个有统计信息，那么 Oracle 在解析该 SQL 时就会使用 CBO；如果该 SQL 中所涉及的所有表对象均没有统计信息，那么此时 Oracle 就会使用 RBO。

3．FIRST_ROWS_n（n = 1, 10, 100, 1000）

这里 FIRST_ROWS_n（n = 1, 10, 100, 1000）可以是 FIRST_ROWS_1、FIRST_ROWS_10、FIRST_ROWS_100 和 FIRST_ROWS_1000 中的任意一个值，其含义是指当 OPTIMIZER_MODE 的值为 FIRST_ROWS_n（n = 1, 10, 100, 1000）时，Oracle 会使用 CBO 来解析目标 SQL，且此时 CBO 在计算该 SQL 的各条执行路径的成本值时的侧重点在于以最快的响应速度返回头 n（n = 1, 10, 100, 1000）条记录。

我们在 1.1.2 节中提到过：CBO 会从目标 SQL 诸多可能的执行路径中选择一条成本值最小的执行路径来作为其执行计划，这也就意味着 CBO 会认为那些消耗系统 I/O 和 CPU 资源最少的执行路径就是当前情形下的最佳选择。

那么当 OPTIMIZER_MODE 的值为 FIRST_ROWS_n（n = 1, 10, 100, 1000）时，是否意味着 CBO 在选择执行计划时所采用的原则将不再是选择成本值最小的执行路径（即消耗系统 I/O 和 CPU 资源最少的执行路径），而是选择那些能够以最快的响应速度返回头 n（n = 1, 10, 100, 1000）条记录所对应的执行路径？

表面上看确实是这样，但实际上 Oracle 采用了一种变通的办法使得 CBO 在选择执行计划时所采用的总原则（成本值最小）依然没有发生变化。这种变通的办法是什么呢？很简单，**当 OPTIMIZER_MODE 的值为 FIRST_ROWS_n（n = 1, 10, 100, 1000）时，Oracle 会把那些能够以最快的响应速度返回头 n（n = 1, 10, 100, 1000）条记录所对应的执行步骤的成本值修改成一个很小的值（远远小于默认情况下 CBO 对同样执行步骤所计算出的成本值）。这样 Oracle 就既没有违背 CBO 选取执行计划的总原则（成本值最小），同时又兼顾了 FIRST_ROWS_n（n = 1, 10, 100, 1000）的含义。**

4．FIRST_ROWS

FIRST_ROWS 是一个在 Oracle 9i 中就已经过时的参数，它表示 Oracle 在解析目标 SQL 时会联合使用 CBO 和 RBO。这里联合使用 CBO 和 RBO 的含义是指在大多数情况下，FIRST_ROWS 还是会使用 CBO 来解析目标 SQL，且此时 CBO 在计算该 SQL 的各条执行路径的成本值时的侧重点在于以最快的响应速度返回头几条记录（类似于 FIRST_ROWS_n）；但是，当出现了一些特定情况时，FIRST_ROWS 转而会使用 RBO 中的一些内置的规则来选取执行计划而不再考虑成本。比如当 OPTIMIZER_MODE 的值为 FIRST_ROWS 时有一个内置的规则，就是如果 Oracle 发现能用相关的索引来避免排序，则 Oracle 就会选择该索引所对应的执行路径而不

再考虑成本，这显然是不合理的。与之相对应的，在 OPTIMIZER_MODE 的值为 FIRST_ROWS 的情形下，你会发现索引全扫描出现的概率会比之前有所增加，这是因为走索引全扫描能够避免排序的缘故。

5．ALL_ROWS

ALL_ROWS 是 Oracle 10g 以及后续 Oracle 数据库版本中 OPTIMIZER_MODE 的默认值，它表示 Oracle 会使用 CBO 来解析目标 SQL，且此时 CBO 在计算该 SQL 的各条执行路径的成本值时的侧重点在于最佳的吞吐量（即最小的系统 I/O 和 CPU 资源的消耗量）。

之前我们在 1.1.2 节中已经提到过：“消耗系统 I/O 和 CPU 资源”（即成本）的计算方法会随着优化器模式的不同而不同。这里我们怎么来理解成本的计算方法会随着优化器模式的不同而不同？

实际上，成本的计算方法随着优化器模式的不同而不同，主要体现在 ALL_ROWS 和 FIRST_ROWS_*n*（*n* = 1, 10, 100, 1000）对成本值计算方法的影响上。当优化器模式为 ALL_ROWS 时，CBO 计算成本的侧重点在于最佳的吞吐量；而当优化器模式为 FIRST_ROWS_*n*（*n* = 1, 10, 100, 1000）时，CBO 计算成本的侧重点会变为以最快的响应速度返回头 *n*（*n* = 1, 10, 100, 1000）条记录。这意味着同样的执行步骤，在优化器模式为 ALL_ROWS 时和 FIRST_ROWS_*n*（*n* = 1, 10, 100, 1000）时 CBO 分别计算出来的成本值会存在巨大的差异，**这也就意味着优化器的模式对 CBO 计算成本（进而对 CBO 选择执行计划）有着决定性的影响！我们**在“1.3 优化器模式对 CBO 计算成本带来巨大影响的实例”中会介绍一个由于优化器模式的不当设置而导致 CBO 认为全表扫描一个 700 多万行数据的大表的成本值仅为 2，进而直接导致 CBO 选错执行计划的实例。

1.2.2　结果集

结果集（Row Source）是指包含指定执行结果的集合。对于优化器而言（无论是 RBO 还是 CBO），结果集和目标 SQL 执行计划的执行步骤相对应，一个执行步骤所产生的执行结果就是该执行步骤所对应的输出结果集。

对于目标 SQL 的执行计划而言，其中某个执行步骤的输出结果就是该执行步骤所对应的输出结果集，同时，该执行步骤所对应的输出结果集可能就是下一个执行步骤的输入结果集。这样一步一步执行下来，伴随的就是结果集在各个执行步骤之间的传递，等目标 SQL 执行计划的各个执行步骤全部执行完毕后，最后的输出结果集就是该 SQL 最终的执行结果。

对于 RBO 而言，我们在对应的执行计划中看不到对相关执行步骤所对应的结果集的描述，虽然结果集的概念对于 RBO 来说也同样适用。

对于 CBO 而言，对应执行计划中的列（Rows）反映的就是 CBO 对于相关执行步骤所对应输出结果集的记录数（即 Cardinality）的估算值。

我们来看如下使用 CBO 的执行计划范例：

```
已选择13行。

执行计划
----------------------------------------------------------
Plan hash value: 2059184959

-------------------------------------------------------------------------------------------
| Id  | Operation                    | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
```

```
|   0 | SELECT STATEMENT            |             |    13 |   494 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMP         |    13 |   494 |     2   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN          | IDX_EMP_MGR |    13 |       |     1   (0)| 00:00:01 |
---------------------------------------------------------------------------------------------

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

   2 - access("MGR"=7902)
```

……省略显示部分内容

对于上述使用 CBO 的执行计划而言，我们将 Id =1、2 的执行步骤所对应的输出结果集分别记为输出结果集 1 和输出结果集 2。这里 Oracle 会先执行 Id = 2 的执行步骤。注意到上述 Id = 2 的执行步骤所对应的列 Rows 的值为 13，这说明 CBO 对输出结果集 2 的 Cardinality 的估算值为 13。同时，输出结果集 2 又会作为 Id = 1 的执行步骤的输入结果集，注意到上述 Id = 1 的执行步骤所对应的列 Rows 的值也为 13，这说明 CBO 对输出结果集 1 的 Cardinality 的估算值也为 13。同时我们可以看到 Id = 0 的执行步骤为“SELECT STATEMENT”，这说明输出结果集 1 就是上述整个目标 SQL 的最终执行结果。

1.2.3 访问数据的方法

对于优化器而言，它在解析目标 SQL、得到其执行计划时至关重要的一点是决定访问数据的方法，即优化器要决定采用什么样的方式和方法去访问目标 SQL 所需要访问的存储在 Oracle 数据库中的数据。

目标 SQL 所需要访问的数据一般存储在表里，而 Oracle 访问表中数据的方法有两种：一种是直接访问表；另一种是先访问索引，再回表（当然，如果目标 SQL 所要访问的数据只通过访问相关的索引就可以得到，那么此时就不需要再回表了）。

接下来，我们就来分别介绍上述这两种访问表中数据的方法。

1.2.3.1 访问表的方法

Oracle 数据库中直接访问表中数据的方法有两种：一种是全表扫描；另一种是 ROWID 扫描。

1.2.3.1.1 全表扫描

全表扫描是指 Oracle 在访问目标表里的数据时，会从该表所占用的第一个区（EXTENT）的第一个块（BLOCK）开始扫描，一直扫描到该表的高水位线（HWM，High Water Mark），这段范围内所有的数据块 Oracle 都必须读到。当然，Oracle 会对这期间读到的所有数据施加目标 SQL 的 where 条件中指定的过滤条件，最后只返回那些满足过滤条件的数据。

不是说全表扫描不好，事实上 Oracle 在做全表扫描操作时会使用多块读，这在目标表的数据量不大时执行效率是非常高的，但全表扫描最大的问题就在于走全表扫描的目标 SQL 的执行时间会不稳定、不可控，这个执行时间一定会随着目标表数据量的递增而递增。因为随着目标表数据量的递增，它的高水位线会一直不断往上涨，所以全表扫描该表时所需要读取的数据块的数量也会不断增加，这意味着全表扫描该表时所需要耗费的 I/O 资源会随之不断增加，当然完成对该表的全表扫描操作所需要耗费的时间也会随之增加。另外，对于 CBO 而言，所要耗费的 I/O 资源不断增加则意味着全表扫描的成本值也会随着目标表数据量的递增而递增。关于 Oracle 中全表扫描成本值的计算方法，我们会在第 5 章的“5.9 系统统计信息”中详细描述，这里不再赘述。

在 Oracle 中，如果对目标表不停地插入数据，当分配给该表的现有空间不足时高水位线就会向上移动，但如果你用 DELETE 语句从该表删除数据，则高水位线并不会随之往下移动（这在某种程度上契合了“高水位线”的定义，就好比水库的水位，当水库涨水时，水位会往上移，当水库放水后，曾经的最高水位的痕迹还是会清晰可见）。**高水位线的这种特性所带来的副作用是，即使使用 DELETE 语句删光了目标表中的所有数据，高水位线还是会在原来的位置，这意味着全表扫描该表时 Oracle 还是需要扫描该表高水位线下的所有数据块，所以此时对该表的全表扫描操作所耗费的时间与之前相比并不会有明显的改观。**

1.2.3.1.2　ROWID 扫描

ROWID 扫描是指 Oracle 在访问目标表里的数据时，直接通过数据所在的 ROWID 去定位并访问这些数据。ROWID 表示的是 Oracle 中的数据行记录所在的物理存储地址，也就是说 ROWID 实际上是和 Oracle 中数据块里的行记录一一对应的。

既然 ROWID 代表的就是表的数据行所在的物理存储地址，那么当 Oracle 知道待访问的数据行所在的 ROWID 后，自然就可以根据该 ROWID 去直接访问对应表的相关数据行，这就是 ROWID 扫描的含义。

从严格意义上来说，Oracle 中的 ROWID 扫描有两层含义：一种是根据用户在 SQL 语句中输入的 ROWID 的值直接去访问对应的数据行记录；另外一种是先去访问相关的索引，然后根据访问索引后得到的 ROWID 再回表去访问对应的数据行记录。

对 Oracle 中的堆表而言，我们可以通过 Oracle 内置的 ROWID 伪列得到对应行记录所在的 ROWID 的值（注意，这个 ROWID 只是一个伪列，在实际的表块中并不存在该列），然后我们还可以通过 DBMS_ROWID 包中的相关方法（dbms_rowid.rowid_relative_fno、dbms_rowid.rowid_block_number 和 dbms_rowid.rowid_row_number）将上述 ROWID 伪列的值翻译成对应数据行的实际物理存储地址。

我们来看一个使用 ROWID 伪列和 DBMS_ROWID 包的实例。执行如下 SQL，查询表 EMP 中的所有记录：

```
SQL> select empno, ename, rowid, dbms_rowid.rowid_relative_fno(rowid) || '_' ||
dbms_rowid.rowid_block_number(rowid) || '_' || dbms_rowid.rowid_row_number(rowid) location from
emp;

EMPNO     ENAME          ROWID              LOCATION
-------   ---------  ------------------   ----------
 7369     CUIHUA1     AAAR3sAAEAAAACXAAA     4_151_0
 7499     ALLEN       AAAR3sAAEAAAACXAAB     4_151_1
 ……省略显示部分内容
 7902     FORD        AAAR3sAAEAAAACXAAM     4_151_12
 7934     MILLER      AAAR3sAAEAAAACXAAN     4_151_13

13 rows selected
```

从上述显示的内容中我们可以看出，EMPNO 为 7369 的行记录所对应的 ROWID 伪列的值为“AAAR3sAAEAAAACXAAA”，使用 DBMS_ROWID 包对该伪列翻译后的值为“4_151_0”，这表示 EMPNO 为 7369 的行记录实际的物理存储地址位于 4 号文件的第 151 个数据块的第 0 行记录（数据块里数据行记录的记录号从 0 开始算起）。

上述 ROWID 伪列的值是可以直接在 SQL 语句的 where 条件中使用的，这就是 Oracle 中 ROWID 扫描的两层含义中的第一种：根据用户在 SQL 语句中输入的 ROWID 的值直接去访问对应的数据行记录。

现在执行一次如下使用 ROWID 伪列的 SQL：

```
SQL> select empno,ename from emp where rowid='AAAR3sAAEAAAACXAAA';
EMPNO      ENAME
------   ----------
7369       CUIHUA1
```

从上述显示的内容中我们可以看出，Oracle 确实是通过 ROWID 伪列（即 rowid='AAAR3sAAEAAAACXAAA'）直接访问到了 EMPNO 为 7369 的行记录。

1.2.3.2 访问索引的方法

首先需要说明的是，这一节里提到的索引指的是我们在 Oracle 数据库中最常用的 B 树索引，Oracle 数据库里其他类型的索引这里暂不考虑。在详细描述访问索引的方法之前，我们先来看一下 Oracle 数据库里 B 树索引的结构，如图 1-2 所示。

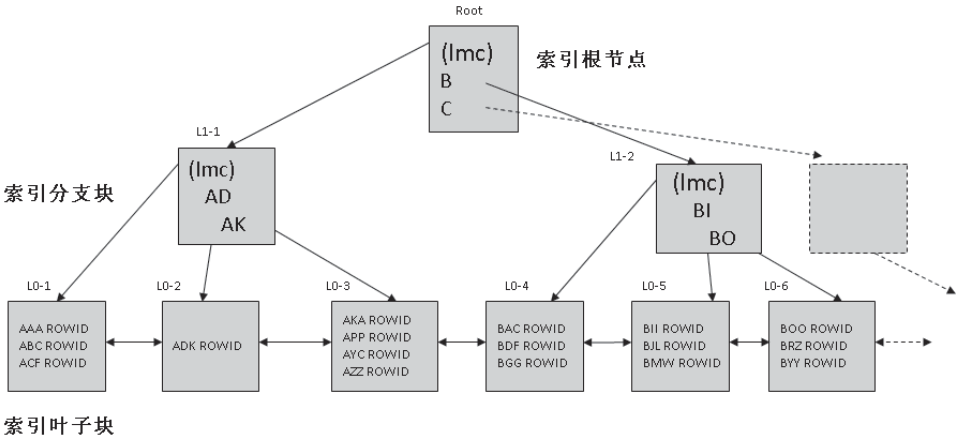

图 1-2　Oracle 数据库中 B 树索引的结构

从图 1-2 中我们可以看出，Oracle 数据库里的 B 树索引就好像一棵倒长的树，它包含两种类型的数据块，一种是索引分支块，另一种是索引叶子块。

索引分支块包含指向相应索引分支块/叶子块的指针和索引键值列（**这里的指针是指相关分支块/叶子块的块地址 RDBA。每个索引分支块都会有两种类型的指针，一种是 lmc，另一种是索引分支块的索引行记录所记录的指针。lmc 是 Left Most Child 的缩写，每个索引分支块都只有一个 lmc，这个 lmc 指向的分支块/叶子块中的所有索引键值列中的最大值一定小于该 lmc 所在索引分支块的所有索引键值列中的最小值；而索引分支块的索引行记录所记录的指针所指向的分支块/叶子块的所有索引键值列中的最小值一定大于或等于该行记录的索引键值列的值**）。这个索引键值列不一定就是完整的被索引键值，它可能只是被索引键值的前缀，只要 Oracle 能通过这些前缀区分相应的索引分支块/叶子块就行，这样 Oracle 就能够既节省索引分支块的存储空间，又可以快速定位其下层的索引分支块/叶子块。索引分支块最上层的那个块就是所谓的索引根节点，在图 1-2 中，根节点就是包含“B C”的那个分支块。在 Oracle 里访问 B 树索引的操作都必须从根节点开始，即都会经历一个从根节点到分支块再到叶子块的过程。

索引叶子块包含被索引键值和用于定位该索引键值所在的数据行在表中实际物理存储位置的 ROWID。对于唯一性 B 树索引而言，ROWID 是存储在索引行的行头，所以此时 Oracle 并不需要额外存储该 ROWID 的长

度。而对于非唯一性 B 树索引而言，ROWID 被当作额外的列与被索引的键值列一起存储，所以此时 Oracle 既要存储 ROWID，同时又要存储其长度，这意味着在同等条件下，唯一性 B 树索引要比非唯一性 B 树索引节省索引叶子块的存储空间。对于非唯一性索引而言，B 树索引的有序性体现在 Oracle 会按照被索引键值和相应的 ROWID 来联合排序。Oracle 里的索引叶子块是左右互联的，即相当于有一个双向指针链表把这些索引叶子块互相连接在了一起。

正是由于上述结构特点，Oracle 数据库中的 B 树索引才具有如下优势。

（1）所有的索引叶子块都在同一层，即它们距离索引根节点的深度是相同的。这也意味着访问索引叶子块的任何一个索引键值所花费的时间几乎相同。

（2）Oracle 会保证所有的 B 树索引都是自平衡的，即不可能出现不同的索引叶子块不处于同一层的现象。

（3）**通过 B 树索引访问表里行记录的效率并不会随着相关表的数据量的递增而显著降低，即通过走索引访问数据的时间是可控的、基本稳定的，这也是走索引和全表扫描的最大区别**。全表扫描最大的劣势就在于其访问时间不可控，不稳定，即全表扫描所花费的时间会随着目标表数据量的递增而递增。

B 树索引的上述结构就决定了在 Oracle 里通过 B 树索引访问数据的过程是先访问相关的 B 树索引，然后根据访问该索引后得到的 ROWID 再回表去访问对应的数据行记录（当然，如果目标 SQL 所要访问的数据通过访问相关的 B 树索引就可以得到，那么就不需要再回表了）。访问相关的 B 树索引和回表都需要消耗 I/O，这意味着在 Oracle 中访问索引的成本由两部分组成：一部分是访问相关的 B 树索引的成本（从根节点定位到相关的分支块，再定位到相关的叶子块，最后对这些叶子块执行扫描操作）；另外一部分是回表的成本（根据得到的 ROWID 再回表去扫描对应的数据行所在的数据块）。关于 Oracle 中访问索引的成本计算方法，在第 5 章“5.4.2 聚簇因子的含义及重要性”中会给出具体的计算公式并予以说明，这里不再赘述。

接下来，我们就来具体介绍 Oracle 中一些常见的访问 B 树索引的方法。

1.2.3.2.1　索引唯一性扫描

索引唯一性扫描（INDEX UNIQUE SCAN）是针对唯一性索引（UNIQUE INDEX）的扫描，它仅仅适用于 where 条件里是等值查询的目标 SQL。因为扫描的对象是唯一性索引，所以索引唯一性扫描的结果至多只会返回一条记录。

1.2.3.2.2　索引范围扫描

索引范围扫描（INDEX RANGE SCAN）适用于所有类型的 B 树索引，当扫描的对象是唯一性索引时，此时目标 SQL 的 where 条件一定是范围查询（谓词条件为 BETWEEN、<、>等）；当扫描的对象是非唯一性索引时，对目标 SQL 的 where 条件没有限制（可以是等值查询，也可以是范围查询）。索引范围扫描的结果可能会返回多条记录，其实这就是索引范围扫描中“范围”二字的本质含义。

需要注意的是，即使是针对同等条件下的相同 SQL，当目标索引的索引行的数量大于 1 时，索引范围扫描所耗费的逻辑读会多于索引唯一性扫描所耗费的逻辑读。这是因为索引唯一性扫描的扫描结果至多只会返回一条记录，所以 Oracle 明确知道此时只需要访问相关的叶子块一次就可以直接返回了；但对于索引范围扫描而言，因为其扫描结果可能会返回多条记录，同时又因为目标索引的索引行数量大于 1，Oracle 为了确定索引范围扫描的扫描终点，就不得不去多次访问相关的叶子块，所以**在同等条件下，当目标索引的索引行的数量大于 1 时，索引范围扫描所耗费的逻辑读至少会比相应的索引唯一性扫描的逻辑读多 1**。

我们来验证一下上述结论。创建一个测试表 EMP_TEMP：

```
SQL> create table emp_temp as select * from emp;

Table created
```

现在表 EMP_TEMP 中列 EMPNO 的非 NULL 值的数量为 13（这意味着如果在表 EMP_TEMP 的列 EMPNO 上建单键值的 B 树索引，则该索引的索引行的数量一定大于 1）：

```
SQL> select count(empno) from emp_temp;
COUNT(EMPNO)
------------
          13
```

在表 EMP_TEMP 的列 EMPNO 上建一个单键值唯一性 B 树索引 IDX_EMP_TEMP：

```
SQL> create unique index idx_emp_temp on emp_temp(empno);

Index created
```

然后对表 EMP_TEMP 和索引 IDX_EMP_TEMP 收集一下统计信息：

```
SQL> exec dbms_stats.gather_table_stats(ownname => 'SCOTT', tabname => 'EMP_TEMP',
estimate_percent => 100, cascade => true, method_opt=>'for all columns size 1');

PL/SQL procedure successfully completed
```

为了避免 Buffer Cache 和数据字典缓存（Data Dictionary Cache）对逻辑读统计结果的影响，这里我们清空了 Buffer Cache 和数据字典缓存（请注意，这里只是出于测试的目的，请勿随意在生产环境执行下述语句）：

```
SQL> alter system flush shared_pool;  //请勿随意在生产环境执行此语句

System altered

SQL> alter system flush buffer_cache;  //请勿随意在生产环境执行此语句

System altered
```

然后执行如下 SQL：

```
SQL> set autotrace traceonly
SQL> select * from emp_temp where empno=7369;
执行计划
----------------------------------------------------------
Plan hash value: 3451700904
-------------------------------------------------------------------------------------------------
| Id  | Operation                   | Name         | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |              |     1 |    38 |     1   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMP_TEMP     |     1 |    38 |     1   (0)| 00:00:01 |
|*  2 |   INDEX UNIQUE SCAN         | IDX_EMP_TEMP |     1 |       |     0   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("EMPNO"=7369)

统计信息
```

```
------------------------------------------------------------
        459  recursive calls
          0  db block gets
         73  consistent gets
         35  physical reads
          0  redo size
        825  bytes sent via SQL*Net to client
        385  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          6  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

从上述显示内容可以看出，“select * from emp_temp where empno=7369”的执行计划走的是索引唯一性扫描，其耗费的逻辑读为 73。

现在我们 Drop 掉上述唯一性索引 IDX_EMP_TEMP：

```
SQL> drop index idx_emp_temp;

Index dropped
```

然后在表 EMP_TEMP 的列 EMPNO 上创建一个单键值非唯一性同名 B 树索引 IDX_EMP_TEMP：

```
SQL> create index idx_emp_temp on emp_temp(empno);

Index created
```

接着对表 EMP_TEMP 和索引 IDX_EMP_TEMP 收集一下统计信息：

```
SQL> exec dbms_stats.gather_table_stats(ownname => 'SCOTT', tabname => 'EMP_TEMP',
estimate_percent => 100, cascade => true, method_opt=>'for all columns size 1');

PL/SQL procedure successfully completed
```

再次清空 Buffer Cache 和数据字典缓存（请注意，这里只是出于测试的目的，请勿随意在生产环境执行下述语句）：

```
SQL> alter system flush shared_pool;  //请勿随意在生产环境执行此语句

System altered

SQL> alter system flush buffer_cache;  //请勿随意在生产环境执行此语句

System altered
```

最后我们再次执行如下 SQL：

```
SQL> select * from emp_temp where empno=7369;
执行计划
----------------------------------------------------------
Plan hash value: 351331621
--------------------------------------------------------------------------------------------
| Id  | Operation                   | Name         | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |              |     1 |    38 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMP_TEMP     |     1 |    38 |     2   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN          | IDX_EMP_TEMP |     1 |       |     1   (0)| 00:00:01 |
--------------------------------------------------------------------------------------------
```

```
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("EMPNO"=7369)

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

从上述显示内容可以看出，此 SQL 的执行计划已经从之前的索引唯一性扫描变为现在的索引范围扫描，其耗费的逻辑读也从之前的 73 递增到现在的 74，这说明在同等条件下，当目标索引的索引行的数量大于 1 时，索引范围扫描所耗费的逻辑读确实至少会比相应的索引唯一性扫描多 1。

注意，上述测试结果中逻辑读为 73 和 74，这是包含了硬解析时递归调用所耗费的逻辑读，上述 SQL 在软解析/软软解析的情况下不会有这么多的逻辑读。这里容易引起误解，因为这多出来的逻辑读可能来源于递归调用时所耗费的逻辑读（但实际上这里两次递归调用所耗费的逻辑读显然是相同的）。

这里更好的比较方式是不刷新数据字典缓存和 Buffer Cache，然后多执行几次上述 SQL 并取最后几次稳定的执行结果。实际上，这种更好的比较方式所得到的测试结果和上述测试结果是一致的。

关于硬解析和软解析/软软解析，我们会在“第 3 章 Oracle 里的 Cursor 和绑定变量”中详细解释，这里不再赘述。

1.2.3.2.3 索引全扫描

索引全扫描（INDEX FULL SCAN）适用于所有类型的 B 树索引（包括唯一性索引和非唯一性索引）。所谓的“索引全扫描”，就是指要扫描目标索引所有叶子块的所有索引行。这里需要注意的是，索引全扫描需要扫描目标索引的所有叶子块，但这并不意味着需要扫描该索引的所有分支块。在默认情况下，Oracle 在做索引全扫描时只需要通过访问必要的分支块定位到位于该索引最左边的叶子块的第一行索引行，就可以利用该索引叶子块之间的双向指针链表，从左至右依次顺序扫描该索引所有叶子块的所有索引行了。

既然在默认情况下，索引全扫描要从左至右依次顺序扫描目标索引所有叶子块的所有索引行，而索引是有序的，所以**索引全扫描的执行结果也是有序的，并且是按照该索引的索引键值列来排序，这也意味着走索引全扫描能够既达到排序的效果，又同时避免了对该索引的索引键值列的真正排序操作**。

我们来验证上述结论。执行如下 SQL：

```
SQL> set autotrace on
SQL> select empno from emp;
     EMPNO
----------
      7369
      7499
      7521
      7566
      7654
      7698
      7782
      7788
```

```
      7844
      7876
      7900
      7902
      7934
已选择13行。

执行计划
----------------------------------------------------------
Plan hash value: 179099197
-------------------------------------------------------------------------------
| Id  | Operation         | Name   | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |        |    13 |    52 |     1   (0)| 00:00:01 |
|   1 |  INDEX FULL SCAN  | PK_EMP |    13 |    52 |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------
统计信息
----------------------------------------------------------
          0  recursive calls
          0  db block gets
          2  consistent gets
          0  physical reads
          0  redo size
        547  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)
         13  rows processed
```

对于上述 SQL（即 select empno from emp）而言，表 EMP 的列 EMPNO 上存在一个单键值 B 树主键索引 PK_EMP，所以列 EMPNO 的属性一定是 NOT NULL，而该 SQL 的查询列又只有列 EMPNO，所以 Oracle 此时就可以走对主键索引 PK_EMP 的索引全扫描。

从上述显示内容中我们可以看出，该 SQL 的执行计划确实走的是对主键索引 PK_EMP 的索引全扫描，而且执行结果已经按照列 EMPNO 排好序了。注意到上述显示内容中“统计信息”部分的“sorts (memory)”和“sorts (disk)”的值均为 0，这说明虽然上述 SQL 的执行结果已经按照列 EMPNO 排好序了，但实际上 Oracle 在这里并没有执行任何排序操作，所以走索引全扫描确实能够既达到排序的效果又避免对目标索引的索引键值列真正的排序操作。

默认情况下，索引全扫描的扫描结果的有序性就决定了索引全扫描是不能够并行执行的，并且通常情况下索引全扫描使用的是单块读。

通常情况下，索引全扫描是不需要回表的，所以索引全扫描适用于目标 SQL 的查询列全部是目标索引的索引键值列的情形。我们知道，对于 Oracle 数据库中的 B 树索引而言，当所有索引键值列全为 NULL 值时不入索引（即当所有索引键值列全为 NULL 值时，这些 NULL 值不会在 B 树索引中存在），这意味着 **Oracle 中能做索引全扫描的前提条件是目标索引至少有一个索引键值列的属性是 NOT NULL**。这是很显然的事情，如果目标索引的所有索引键值列的属性均为允许 NULL 值，此时如果还走索引全扫描，就会漏掉目标表中那些索引键值列均为 NULL 的记录，即此时走索引全扫描的结果就不准了！Oracle 显然不会允许这种事情发生。

1.2.3.2.4　索引快速全扫描

索引快速全扫描（INDEX FAST FULL SCAN）和索引全扫描极为类似，它也适用于所有类型的 B 树索引（包括唯一性索引和非唯一性索引）。和索引全扫描一样，索引快速全扫描也需要扫描目标索引所有叶子块的所有索引行。

索引快速全扫描与索引全扫描相比有如下三点区别。

（1）索引快速全扫描只适用于 CBO。

（2）索引快速全扫描可以使用多块读，也可以并行执行。

（3）索引快速全扫描的执行结果不一定是有序的。这是因为索引快速全扫描时 Oracle 是根据索引行在磁盘上的物理存储顺序来扫描，而不是根据索引行的逻辑顺序来扫描的，所以扫描结果才不一定有序（对于单个索引叶子块中的索引行而言，其物理存储顺序和逻辑存储顺序一致；但对于物理存储位置相邻的索引叶子块而言，块与块之间索引行的物理存储顺序则不一定在逻辑上有序）。

我们现在来验证“索引快速全扫描的执行结果不一定是有序的”这个结论。创建一个测试表 EMP_TEST：

```
SQL> create table emp_test (empno number,col1 char(2000),col2 char(2000),col3 char(2000));

Table created
```

在表 EMP_TEST 上添加一个复合主键索引 PK_EMP_TEST：

```
SQL> alter table emp_test add constraint pk_emp_test primary key (empno,col1,col2,col3);

Table altered
```

使用如下 SQL 往表 EMP_TEST 中插入一些数据：

```
insert into emp_test select empno,ename,job,'A' from emp;
insert into emp_test select empno,ename,job,'B' from emp;
insert into emp_test select empno,ename,job,'C' from emp;
insert into emp_test select empno,ename,job,'D' from emp;
insert into emp_test select empno,ename,job,'E' from emp;
insert into emp_test select empno,ename,job,'F' from emp;
insert into emp_test select empno,ename,job,'G' from emp;
commit;
```

插入完上述数据后表 EMP_TEST 中有 91 条记录：

```
SQL> select count(*) from emp_test;
  COUNT(*)
----------
        91
```

对表 EMP_TEST 及主键索引 PK_EMP_TEST 收集一下统计信息：

```
SQL> exec dbms_stats.gather_table_stats(ownname => 'SCOTT', tabname => 'EMP_TEST',
estimate_percent => 100, cascade => true, method_opt=>'for all columns size 1');

PL/SQL procedure successfully completed
```

然后执行如下带 Hint 的目标 SQL（这里/*+ index_ffs(emp_test pk_emp_test) */的目的是让 Oracle 走对主键索引 PK_EMP_TEST 的索引快速全扫描）：

```
SQL> select /*+ index_ffs(emp_test pk_emp_test) */empno from emp_test;
```

```
     EMPNO
----------
      7369
      7499
      7521
      7566
      7654
      7698
      7782
      7788
      7844
      7876
      7900
      7902
      7934
      7521
      7369
      7499
      7782
      ……省略显示部分内容
      7788
      7900
      7844

已选择91行。


执行计划
----------------------------------------------------------
Plan hash value: 3550420785
-------------------------------------------------------------------------------------
| Id  | Operation             | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |             |    91 |   364 |   185   (0)| 00:00:02 |
|   1 |  INDEX FAST FULL SCAN | PK_EMP_TEST |    91 |   364 |   185   (0)| 00:00:02 |
-------------------------------------------------------------------------------------

……省略显示部分内容
```

从上述显示内容可以看出，目标 SQL 的执行计划走的确实是对主键索引 PK_EMP_TEST 的索引快速全扫描，但从其执行结果来看，列 EMPNO=7521 的记录在执行结果中的显示顺序位于 EMPNO=7934 的记录之后，说明此 SQL 的执行结果并没有按照主键索引 PK_EMP_TEST 的索引键值前导列 EMPNO 来排序，即索引快速全扫描的执行结果确实不一定是有序的。

1.2.3.2.5　索引跳跃式扫描

索引跳跃式扫描（INDEX SKIP SCAN）适用于所有类型的复合 B 树索引（包括唯一性索引和非唯一性索引），它使那些在 where 条件中没有对目标索引的前导列指定查询条件但同时又对该索引的非前导列指定了查询条件的目标 SQL 依然可以用上该索引，这就像是在扫描该索引时跳过了它的前导列，直接从该索引的非前导列开始扫描一样（实际的执行过程并非如此），这也是索引跳跃式扫描中“跳跃”（SKIP）一词的含义。

为什么在 where 条件中没有对目标索引的前导列指定查询条件但 Oracle 依然可以用上该索引呢？这是因为 Oracle 帮你对该索引的前导列的所有 distinct 值做了遍历。

我们来看一个索引跳跃式扫描的实例。创建一个测试表 EMPLOYEE：

```
SQL> create table employee(gender varchar2(1),employee_id number);


Table created
```

将该表的列 EMPLOYEE_ID 的属性设为 NOT NULL：

```
SQL> alter table employee modify(employee_id not null);
```

```
Table altered
```

创建一个名为 IDX_EMPOLYEE 的复合 B 树索引，其中列 GENDER 是该索引的前导列，列 EMPLOYEE_ID 是该索引的第二列：

```
SQL> create index idx_employee on employee(gender,employee_id);

Index created
```

使用如下 PL/SQL 代码往表 EMPLOYEE 中插入 10,000 条记录，其中 5,000 条记录的列 GENDER 的值为“F”，另外 5,000 条记录的列 GENDER 的值为“M”：

```
begin
for i in 1..5000 loop
insert into employee values ('F',i);
end loop;
commit;
end;
/

begin
for i in 5001..10000 loop
insert into employee values ('M',i);
end loop;
commit;
end;
/
```

然后我们来看如下的范例 SQL 9：

```
select * from employee
  where employee_id = 100;
```

对于范例 SQL 9 而言，其 where 条件是“employee_id = 100”，即它只对复合 B 树索引 IDX_EMPOLYEE 的第二列 EMPLOYEE_ID 指定了查询条件，但并没有对该索引的前导列 GENDER 指定任何查询条件。这种情况下 Oracle 在执行范例 SQL 9 时是否能用上索引 IDX_EMPOLYEE 呢？

我们现在来执行范例 SQL 9：

```
SQL> set autotrace traceonly
SQL> select * from employee where employee_id = 100;
执行计划
----------------------------------------------------------
Plan hash value: 461756150
-------------------------------------------------------------------------------------
| Id  | Operation         | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |               |     1 |     6 |     3   (0)| 00:00:01 |
|*  1 |  INDEX SKIP SCAN  | IDX_EMPLOYEE  |     1 |     6 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("EMPLOYEE_ID"=100)
       filter("EMPLOYEE_ID"=100)
```

……省略显示部分内容

从上述显示内容可以看出，Oracle 在执行范例 SQL 9 时已经用上了索引 IDX_EMPOLYEE，并且其执行计划走的就是对该索引的索引跳跃式扫描。

这里在没有指定前导列的情况下还能用上述索引，就是因为 Oracle 帮我们对该索引的前导列的所有 distinct 值做了遍历。

所谓的对目标索引的所有 distinct 值做遍历，其实际含义相当于对原目标 SQL 做等价改写（即把要用的目标索引的所有前导列的 distinct 值都加进来）。索引 IDX_EMPOLYEE 的前导列 GENDER 的 distinct 值只有“F”和“M”两个值，所以这里能使用索引 IDX_EMPOLYEE 的原因可以简单地理解成是 Oracle 将范例 SQL 9 等价改写成了如下形式：

```
select * from employee where gender = 'F' and employee_id = 100
union all
select * from employee where gender = 'M' and employee_id = 100;
```

从上述分析过程可以看出，**Oracle 中的索引跳跃式扫描仅仅适用于那些目标索引前导列的 distinct 值数量较少、后续非前导列的可选择性又非常好的情形，因为索引跳跃式扫描的执行效率一定会随着目标索引前导列的 distinct 值数量的递增而递减**。

1.2.4　表连接

顾名思义，表连接就是指多个表之间用连接条件连接在一起，使用表连接的目标 SQL 的目的就是从多个表获取存储在这些表中的不同维度的数据。体现在 SQL 语句上，含表连接的目标 SQL 的 from 部分会出现多个表，而这些 SQL 的 where 条件部分则会定义具体的表连接条件。

当优化器解析含表连接的目标 SQL 时，它除了会根据目标 SQL 的 SQL 文本的写法来决定表连接的类型之外，还必须决定如下三件事情才能得到最终的执行计划。

1．表连接顺序

不管目标 SQL 中有多少个表做表连接，Oracle 在实际执行该 SQL 时都只能先两两做表连接，再依次执行这样的两两表连接过程，直到目标 SQL 中所有的表都已连接完毕。所以从严格意义上来说，这里的表连接顺序包含两层含义：一层含义是当两个表做表连接时，优化器需要决定这两个表中谁是驱动表（outer table），谁是被驱动表（inner table）；另外一层含义是当多表（超过两个以上的表）做表连接时，优化器需要决定这些表中谁和谁先做表连接，然后决定这个表连接结果所在的结果集再和剩余表中的哪一个再做表连接，这个两两做表连接的过程会一直持续下去，直到目标 SQL 中所有的表都已连接完为止。

2．表连接方法

在 Oracle 数据库中，两个表之间的表连接方法有排序合并连接、嵌套循环连接、哈希连接和笛卡儿连接这四种，所以优化器在解析含表连接的目标 SQL 时，都需要从上述四种方法中选择一种，作为每一对表两两做表连接时所需要采用的方法。

3．访问单表的方法

对于优化器而言，仅决定表连接顺序和表连接方法是不够的，这还不足以得到目标 SQL 的最终执行计划，

因为优化器在对目标 SQL 中的各个表两两做表连接时，还必须决定如何去获取存储在这些表里的不同维度的数据，即优化器还要决定访问单表的方法。比如在访问某个单表时，是采用全表扫描还是走索引，如果是走索引，应该采用什么样的索引访问方法等。

接下来，我们来介绍与表连接相关的各种基础知识。

1.2.4.1 表连接的类型

通常情况下，我们可以认为 Oracle 数据库中的表连接分为内连接和外连接这两种类型，表连接的类型会直接决定表连接的结果，而目标 SQL 的 SQL 文本的写法又直接决定了表连接的类型。

1.2.4.1.1 内连接

内连接（Inner Join）是指表连接的连接结果只包含那些完全满足连接条件的记录。对于包含表连接的目标 SQL 而言，只要其 where 条件中没有写那些标准 SQL 中定义或者 Oracle 中自定义的表示外连接的关键字（比如标准 SQL 中的 left outer join、right outer join、full outer join，或者 Oracle 中自定义的用来表示外连接的关键字“(+)”），则该 SQL 的连接类型就是内连接。

我们来看一个内连接的实例。创建两个测试表 T1 和 T2：

```
SQL> create table t1(col1 number,col2 varchar2(1));

Table created

SQL> create table t2(col2 varchar2(1),col3 varchar2(2));

Table created
```

使用如下 SQL 往表 T1 和 T2 中各插入三条记录：

```
insert into t1 values(1,'A');
insert into t1 values(2,'B');
insert into t1 values(3,'C');
insert into t2 values('A','A2');
insert into t2 values('B','B2');
insert into t2 values('D','D2');
commit;
```

现在表 T1 和 T2 中的记录为如下所示：

```
SQL> select * from t1;
COL1 COL2
---- ----
   1   A
   2   B
   3   C
SQL> select * from t2;
COL2  COL3
----  ----
 A    A2
```

```
B    B2
D    D2
```

我们来看如下的范例 SQL 10：

```
select t1.col1, t1.col2, t2.col3
  from t1, t2
 where t1.col2 = t2.col2 ;
```

上述范例 SQL 10 中并没有使用之前提到的标准 SQL 中定义或者 Oracle 中自定义的表示外连接的关键字，所以该 SQL 的连接类型为内连接，连接条件为“t2.col2 = t1.col2”。

执行范例 SQL 10：

```
SQL> select t1.col1, t1.col2, t2.col3
  2    from t1, t2
  3   where t1.col2 = t2.col2 ;
COL1 COL2 COL3
---- ---- ----
  1    A    A2
  2    B    B2
```

从范例 SQL 10 的执行结果中我们可以看出，内连接的连接结果确实只包含了那些完全满足连接条件的记录。

范例 SQL 10 中关于内连接的写法实际上是 Oracle 自己的写法，这和标准 SQL 中内连接的写法并不相同。标准 SQL 中内连接的写法是用 JOIN ON 或者 JOIN USING。

其中，JOIN ON 的语法是：

“目标表 1 *join* 目标表 2 *on* (连接条件)”。

JOIN USING 的语法是：

“目标表 1 *join* 目标表 2 *using* (连接列集合)”。

对于使用 JOIN USING 的目标 SQL 而言，如果有多个连接列，则其语法中“(连接列集合)”里的各个连接列之间应使用逗号来分隔。

上述范例 SQL 10 如果换成用 JOIN ON 和 JOIN USING 表示的标准 SQL，则其等价形式分别为如下所示的范例 SQL 11 和范例 SQL 12。

范例 SQL 11：

```
select t1.col1, t1.col2, t2.col3
  from t1
  join t2 on (t1.col2 = t2.col2);
```

范例 SQL 12：

```
select t1.col1, col2, t2.col3
  from t1
  join t2 using (col2);
```

执行范例 SQL 11 和 12：

```
SQL> select t1.col1, t1.col2, t2.col3
  2    from t1
  3    join t2 on (t1.col2 = t2.col2);

COL1 COL2 COL3
---- ---- ----
  1    A    A2
  2    B    B2

SQL> select t1.col1, col2, t2.col3
  2    from t1
  3    join t2 using (col2);

COL1 COL2 COL3
---- ---- ----
  1    A    A2
  2    B    B2
```

从上述显示内容中我们可以看出，使用 JOIN ON 和 JOIN USING 的范例 SQL 11 和 12 的执行结果和原范例 SQL 10 的执行结果是一模一样的。

这里需要注意的是，对于使用 JOIN USING 的标准 SQL 而言，如果连接列同时又出现在查询列中，则该连接列前不能带上表名或者表名的别名（alias），否则 Oracle 会报错（ORA-25154）：

```
SQL> select t1.col1, t1.col2, t2.col3
  2    from t1
  3    join t2 using (col2);

select t1.col1, t1.col2, t2.col3
  from t1
  join t2 using (col2)

ORA-25154: column part of USING clause cannot have qualifier
```

上述 SQL 使用了 JOIN USING，并且连接列 COL2 又恰好是查询列之一，所以 Oracle 会报错，上述 SQL 的第二个查询列应该是“col2”而不应是“t1.col2”。

如果使用标准 SQL 来表示表连接，那么有一种特殊的 JOIN USING，我们称之为 NATURAL JOIN。NATURAL JOIN 是一种特殊的 JOIN USING，其含义是使用 NATURAL JOIN 的表连接的连接列是表连接的两个表所有的同名列。使用 NATURAL JOIN 的语法为：

```
目标表 1 natural join 目标表 2
```

这实际上相当于“目标表 1 *join* 目标表 2 *using* (目标表 1 和目标表 2 的所有同名列集合)”。

对于范例 SQL 10、11、12 中的表 T1 和 T2 而言，它们的同名列为 COL2，所以如下使用了 NATURAL JOIN 的范例 SQL 13 和范例 SQL 10、11、12 实际上是等价的。

范例 SQL 13：

```
select t1.col1, col2, t2.col3
```

```
  from t1 natural join t2;
```

执行范例 SQL 13：

```
SQL> select t1.col1, col2, t2.col3
  2   from t1 natural join t2;
COL1 COL2 COL3
---- ---- ----
  1    A    A2
  2    B    B2
```

从上述显示内容中我们可以看出，使用 NATURAL JOIN 的范例 SQL 13 的执行结果和范例 SQL 10、11、12 的执行结果确实是一模一样的。

使用 NATURAL JOIN 的好处是无须在 JOIN USING 中写连接列集合，但其坏处是增加了表连接的执行结果出错的风险，因为两个表之间的同名列不一定在含义上就完全相同（也许它们只是恰好同名而已，其含义是完全不同的，所以也不应该将它们作为连接列），而且即使含义相同，也不一定就需要将它们作为连接列。

我们并不推荐使用 NATURAL JOIN（除非你是刻意为之），虽然它是能让你省去写连接列的麻烦，但不写连接列所付出的代价就是会增加目标 SQL 出错的风险。

1.2.4.1.2　外连接

外连接（Outer Join）是对内连接的一种扩展，它是指表连接的连接结果除了包含那些完全满足连接条件的记录之外还会包含驱动表中所有不满足该连接条件的记录。

标准 SQL 中的外连接分为左连接（Left Outer Join）、右连接（Right Outer Join）和全连接（Full Outer Join）这三种，它们在标准 SQL 中所对应的关键字分别为 left outer join、right outer join 和 full outer join，都可以和 JOIN ON 或 JOIN USING 连用。

左连接的语法为：

```
目标表 1 left outer join 目标表 2 on (连接条件)
```

或

```
目标表 1 left outer join 目标表 2 using (连接列集合)
```

“目标表 1 *left outer join* 目标表 2 *on* (连接条件)”的含义为目标表 1 和目标表 2 按括号中的连接条件来做表连接，位于关键字“*left outer join*”左边的目标表 1 会作为该表连接的驱动表（关键字“*left outer*”即表明位置处于 left 的表就是 outer table，这里的 outer table 就是指驱动表）。此时的连接结果除了包含目标表 1 和目标表 2 中所有满足该连接条件的记录外，还会包含驱动表（即目标表 1）中所有不满足该连接条件的记录，同时，驱动表中所有不满足该连接条件的记录所对应的被驱动表（即目标表 2）中的查询列均会以 NULL 值来填充。

右连接的语法为：

```
目标表 1 right outer join 目标表 2 on (连接条件)
```

或

```
目标表 1 right outer join 目标表 2 using (连接列集合)
```

“目标表 1 *right outer join* 目标表 2 *on* (连接条件)”的含义为目标表 1 和目标表 2 按括号中的连接条件来做表连接，位于关键字“*right outer join*”右边的目标表 2 会作为该表连接的驱动表（“*right outer join*”中的关键字“*right outer*”即表明位置处于 right 的表就是 outer table，即驱动表）。此时的连接结果除了包含目标表 1 和目标表 2 中所有满足该连接条件的记录外，还会包含驱动表（即目标表 2）中所有不满足该连接条件的记录，同时，驱动表中所有不满足该连接条件的记录所对应的被驱动表（即目标表 1）中的查询列均会以 NULL 值来填充。

全连接的语法为：

```
目标表 1 full outer join 目标表 2 on (连接条件)
```

或

```
目标表 1 full outer join 目标表 2 using (连接列集合)
```

“目标表 1 *full outer join* 目标表 2 *on* (连接条件)”的含义为目标表 1 和目标表 2 按括号中的连接条件来做表连接。此时的连接结果除了包含目标表 1 和目标表 2 中所有满足该连接条件的记录外，还会包含目标表 1 和目标表 2 中所有不满足该连接条件的记录，同时，目标表 1 和目标表 2 中所有不满足该连接条件的记录所对应的另外一个表中的查询列均会以 NULL 值来填充。

可以简单地把全连接理解成是先做左连接，再做右连接，最后对左连接和右连接的连接结果做一个 UNION 操作（**注意，虽然可以这么理解，但 Oracle 在实际执行全连接时不会这么做**），即可以把全连接理解成：

```
目标表 1 left outer join 目标表 2 on (连接条件)
union
目标表 1 right outer join 目标表 2 on (连接条件)
```

或

```
目标表 1 left outer join 目标表 2 using (连接列集合)
union
目标表 1 right outer join 目标表 2 using (连接列集合)
```

我们现在来看几个标准 SQL 中外连接的实例。这里还是以“1.2.4.1.1 内连接”中的测试表 T1 和 T2 为例来说明。

现在表 T1 和 T2 中的记录为如下所示：

```
SQL> select * from t1;
COL1 COL2
---- ----
   1   A
   2   B
   3   C
SQL> select * from t2;
COL2 COL3
---- ----
 A    A2
 B    B2
 D    D2
```

我们来看如下的范例 SQL 14：

```
select t1.col1, t1.col2, t2.col3
  from t1 left outer join t2 on (t1.col2 = t2.col2);
```

对于范例 SQL 14 而言，此时的驱动表是位于关键字“*left outer join*”左边的表 T1，连接条件是“t1.col2 = t2.col2”。根据之前我们对左连接的介绍，可以推断出该 SQL 的执行结果除了包含表 T1 中满足上述连接条件的记录之外，还会包含表 T1 中 COL1=3，COL2＝'C'的那条记录，并且其对应的表 T2 中的列 COL3 的值为 NULL。

执行范例 SQL 14：

```
SQL> select t1.col1, t1.col2, t2.col3
  2   from t1 left outer join t2 on (t1.col2 = t2.col2);

COL1 COL2 COL3
---- ---- ----
 1    A    A2
 2    B    B2
 3    C
```

从上述执行结果可以看出，左连接的执行结果确实是除了包含所有满足连接条件的记录外，还包含驱动表中所有不满足该连接条件的记录，同时，驱动表中所有不满足该连接条件的记录所对应的被驱动表中的查询列均以 NULL 值来填充。

我们再来看如下的范例 SQL 15：

```
select t1.col1, t1.col2, t2.col3
  from t1 right outer join t2 on (t1.col2 = t2.col2);
```

对于范例 SQL 15 而言，此时的驱动表是位于关键字“*right outer join*”右边的表 T2，连接条件是“t1.col2 = t2.col2”。根据之前我们对右连接的介绍，可以推断出该 SQL 的执行结果除了包含表 T2 中满足上述连接条件的记录之外，还会包含表 T2 中 COL3＝'D2'的那条记录，并且其对应的表 T1 中的列 COL1 和 COL2 的值均为 NULL。

执行范例 SQL 15：

```
SQL> select t1.col1, t1.col2, t2.col3
  2   from t1 right outer join t2 on (t1.col2 = t2.col2);

COL1 COL2 COL3
---- ---- ----
 1    A    A2
 2    B    B2
           D2
```

从上述执行结果可以看出，和左连接一样，右连接的执行结果也是除了包含所有满足连接条件的记录外，还包含驱动表中所有不满足该连接条件的记录，同时，驱动表中所有不满足该连接条件的记录所对应的被驱动表中的查询列均以 NULL 值来填充。

我们接着来看如下的范例 SQL 16：

```
select t1.col1, t1.col2, t2.col3
  from t1 full outer join t2 on (t1.col2 = t2.col2);
```

根据之前对全连接的介绍，我们知道范例 SQL 16 实际上可以近似看作和如下形式的范例 SQL 17 等价：

```
select t1.col1, t1.col2, t2.col3
  from t1
  left outer join t2 on (t1.col2 = t2.col2)
union
select t1.col1, t1.col2, t2.col3
  from t1
  right outer join t2 on (t1.col2 = t2.col2)
```

我们现在来验证一下，分别执行范例 SQL 16 和范例 SQL 17：

```
SQL> select t1.col1, t1.col2, t2.col3
  2   from t1 full outer join t2 on (t1.col2 = t2.col2);
COL1 COL2 COL3
---- ---- ----
 1    A    A2
 2    B    B2
          D2
 3    C
SQL> select t1.col1, t1.col2, t2.col3
  2   from t1
  3   left outer join t2 on (t1.col2 = t2.col2)
  4 union
  5 select t1.col1, t1.col2, t2.col3
  6   from t1
  7   right outer join t2 on (t1.col2 = t2.col2);
COL1 COL2 COL3
---- ---- ----
 1    A    A2
 2    B    B2
 3    C
          D2
```

我们可以看出，虽然显示顺序并不完全相同（因为 union 会对执行结果排序，而全连接是不需要排序的，所以这里只是近似等价），但范例 SQL 16 和范例 SQL 17 的执行结果实际上还是一模一样的。这说明当目标表 1 和目标表 2 以指定的连接条件做全连接时，全连接的连接结果确实是除了包含目标表 1 和目标表 2 中所有满足该连接条件的记录外，还包含目标表 1 和目标表 2 中所有不满足该连接条件的记录，同时，目标表 1 和目标表 2 中所有不满足该连接条件的记录所对应的另外一个表中的查询列均会以 NULL 值来填充。

我们在“1.2.4.1.1 内连接”和“1.2.4.1.2 外连接”中介绍的范例 SQL 中除了带连接条件外，并没有带其他额外的限制条件。如果目标 SQL 中除了表连接条件之外还带了额外的限制条件，则目标 SQL 中表连接的类型和该额外限制条件在目标 SQL 的 SQL 文本中出现的位置都可能会对最终执行结果产生影响。

我们先来看如下的范例 SQL 18 和范例 SQL 19。

范例 SQL 18：

```
select t1.col1, t1.col2, t2.col3
```

```
  from t1
  join t2 on (t1.col2 = t2.col2 and t1.col1 = 1);
```

范例 SQL 19：

```
select t1.col1, t1.col2, t2.col3
  from t1
  join t2 on (t1.col2 = t2.col2)
 where t1.col1 = 1;
```

对于上述范例 SQL 18 和 19 而言，除了表连接条件“t1.col2 = t2.col2”之外，还多出了一个额外的限制条件“t1.col1 = 1”。对于该限制条件，它在范例 SQL 18 的 SQL 文本中位于 JOIN ON 所对应的括号内，而它在范例 SQL 19 的 SQL 文本中位于 JOIN ON 所对应的括号外，虽然该限制条件在 SQL 文本中所处的位置不同，但因为范例 SQL 18 和 19 的连接类型均为内连接，所以该限制条件的位置并不会影响实际的表连接结果。

我们来验证一下，分别执行范例 SQL 18 和 19：

```
SQL> select t1.col1, t1.col2, t2.col3
  2    from t1
  3    join t2 on (t1.col2 = t2.col2 and t1.col1 = 1);
COL1 COL2 COL3
---- ---- ----
  1    A    A2
SQL> select t1.col1, t1.col2, t2.col3
  2    from t1
  3    join t2 on (t1.col2 = t2.col2)
  4   where t1.col1 = 1;
COL1 COL2 COL3
---- ---- ----
  1    A    A2
```

从上述执行结果可以看出，范例 SQL 18 和 19 的执行结果是一模一样的，这说明对于内连接而言，除了表连接条件之外的额外限制条件在目标 SQL 的 SQL 文本中所处的位置并不会影响该 SQL 的实际执行结果。

我们再来看如下的范例 SQL 20 和范例 SQL 21。

范例 SQL 20：

```
select t1.col1, t1.col2, t2.col3
  from t1
  right outer join t2 on (t1.col2 = t2.col2 and t1.col1 = 1);
```

范例 SQL 21：

```
select t1.col1, t1.col2, t2.col3
  from t1
 right outer join t2 on (t1.col2 = t2.col2)
 where t1.col1 = 1;
```

对于范例 SQL 20 和 21 而言，它们和范例 SQL 18、19 一样，除了表连接条件“t1.col2 = t2.col2”之外，还多出了一个额外的限制条件“t1.col1 = 1”，但由于范例 SQL 20 和 21 的连接类型为外连接，所以现在的情况

就和之前完全不同。具体来说就是这样：对于该限制条件（即 t1.col1 = 1），它在范例 SQL 20 的 SQL 文本中位于 RIGHT OUTER JOIN ON 所对应的括号内，这表示该限制条件会在表 T1 和表 T2 做右连接之前就被应用在表 T1 上，即在范例 SQL 20 中，参与右连接的 T1 中的数据是那些满足条件“t1.col1 = 1”的记录；而该限制条件在范例 SQL 21 的 SQL 文本中位于 RIGHT OUTER JOIN ON 所对应的括号外，这表示该限制条件在表 T1 和表 T2 做完右连接后，才会被应用在表 T1 和表 T2 的连接结果集上，即在范例 SQL 21 中，参与右连接的是表 T1 中的所有数据。

按照右连接的定义，范例 SQL 20 和 21 中的被驱动表是表 T1，所以限制条件“t1.col1 = 1”到底是在右连接之前作用在表 T1 上，还是在右连接之后作用在右连接的连接结果集上，就会对上述 SQL 的最终执行结果产生本质的影响。

我们来验证一下，分别执行范例 SQL 20 和 21：

```
SQL> select t1.col1, t1.col2, t2.col3
  2    from t1
  3    right outer join t2 on (t1.col2 = t2.col2 and t1.col1 = 1);
COL1 COL2 COL3
---- ---- ----
  1    A    A2
            D2
            B2
SQL> select t1.col1, t1.col2, t2.col3
  2    from t1
  3   right outer join t2 on (t1.col2 = t2.col2)
  4   where t1.col1 = 1;
COL1 COL2 COL3
---- ---- ----
  1    A    A2
```

从上述执行结果中我们可以看出，现在范例 SQL 20 和 21 的执行结果就不一样了，这说明**对于外连接而言，除了表连接条件之外的额外限制条件在目标 SQL 的 SQL 文本中所处的位置确实可能会影响该 SQL 的实际执行结果。**

和标准 SQL 里表示外连接的语法不同，Oracle 用自定义的关键字“(+)”来表示外连接。关键字“(+)”的位置在目标 SQL 连接条件中某一个表的连接列的后面，其含义是**关键字“(+)”出现在哪个表的连接列后面，就表明哪个表会以 NULL 值来填充那些不满足连接条件并位于该表中的查询列，此时应该以关键字“(+)”对面的表来作为外连接的驱动表，这里的关键是决定哪个表是驱动表！**

之前介绍过的范例 SQL 14 为如下所示：

```
select t1.col1, t1.col2, t2.col3
  from t1 left outer join t2 on (t1.col2 = t2.col2);
```

范例 SQL 14 是左连接，驱动表是 T1，如果用 Oracle 中自定义的外连接表示方式来等价改写它的话，就是如下所示的范例 SQL 22：

```
select t1.col1, t1.col2, t2.col3
  from t1, t2
 where t1.col2 = t2.col2(+);
```

对于范例 SQL 22 而言，关键字“(+)”出现在表 T2 的连接列 COL2 后面，这就表示表 T2 会以 NULL 值来填充那些不满足连接条件“t1.col2 = t2.col2”并位于表 T2 中的查询列（即列 COL3），此时应该以关键字“(+)”对面的表 T1 来作为外连接的驱动表。

执行范例 SQL 22：

```
SQL> select t1.col1, t1.col2, t2.col3
  2    from t1, t2
  3   where t1.col2 = t2.col2(+);
COL1 COL2 COL3
---- ---- ----
  1    A    A2
  2    B    B2
  3    C
```

从上述执行结果中我们可以看出，范例 SQL 22 和范例 SQL 14 的执行结果确实一模一样。

之前介绍过的范例 SQL 15 为如下所示：

```
select t1.col1, t1.col2, t2.col3
  from t1 right outer join t2 on (t1.col2 = t2.col2);
```

范例 SQL 15 是右连接，驱动表是 T2，如果用 Oracle 中自定义的外连接表示方式来等价改写它的话，就是如下所示的范例 SQL 23：

```
select t1.col1, t1.col2, t2.col3
  from t1, t2
 where t1.col2(+) = t2.col2;
```

对于范例 SQL 23 而言，关键字“(+)”出现在表 T1 的连接列 COL2 后面，这就表示表 T1 会以 NULL 值来填充那些不满足连接条件“t1.col2 = t2.col2”并位于表 T1 中的查询列（即列 COL1 和 COL2），此时应该以关键字“(+)”对面的表 T2 来作为外连接的驱动表。

执行范例 SQL 23：

```
SQL> select t1.col1, t1.col2, t2.col3
  2    from t1, t2
  3   where t1.col2(+) = t2.col2;
COL1 COL2 COL3
---- ---- ----
  1    A    A2
  2    B    B2
            D2
```

从上述执行结果可以看出，范例 SQL 23 和范例 SQL 15 的执行结果确实一模一样。

之前介绍过的范例 SQL 16 为如下所示：

```
select t1.col1, t1.col2, t2.col3
  from t1 full outer join t2 on (t1.col2 = t2.col2);
```

上述范例 SQL 16 是全连接，如果用 Oracle 中自定义的外连接表示方式来等价改写它的话，则其近似等价

改写形式就是如下所示的范例 SQL 24：

```
select t1.col1, t1.col2, t2.col3
  from t1, t2
 where t1.col2 = t2.col2(+)
union
select t1.col1, t1.col2, t2.col3
  from t1, t2
 where t1.col2(+) = t2.col2;
```

范例 SQL 24 是我们杜撰出来的等价改写形式，这里只是为了方便大家理解全连接的原理，Oracle 在实际执行范例 SQL 16 时是断然不会采用范例 SQL 24 这样的等价改写形式的。

在 Oracle 11gR1 之前，Oracle 在执行范例 SQL 16 时实际上是将其等价改写成范例 SQL 25 这样的形式：

```
select t1.col1, t1.col2, t2.col3
  from t1, t2
 where t1.col2 = t2.col2(+)
union all
select null,null,t2.col3
  from t2
 where not exists (select 1 from t1 where t1.col2 = t2.col2);
```

执行范例 SQL 25：

```
SQL> select t1.col1, t1.col2, t2.col3
  2    from t1, t2
  3   where t1.col2 = t2.col2(+)
  4  union all
  5  select null,null,t2.col3
  6    from t2
  7   where not exists (select 1 from t1 where t1.col2 = t2.col2);
COL COL2 COL3
--- ---- ----
 1   A    A2
 2   B    B2
 3   C
          D2
```

从上述执行结果中我们可以看出，范例 SQL 25 和范例 SQL 16 的执行结果确实一模一样。

之前提到过：对于外连接而言，表连接条件之外的额外限制条件在目标 SQL 的 SQL 文本中所处位置的不同可能会影响该 SQL 的实际执行结果。现在的问题是，如果不用标准 SQL，而是用 Oracle 自定义的关键字“(+)”来表示外连接的话，那么应如何体现出标准 SQL 里这种由于额外限制条件所处位置的不同而对实际执行结果所带来的影响？

很简单，Oracle 是通过在额外限制条件的目标列的后面带上同样的关键字“(+)”来体现出上述这种影响的。

之前介绍过的范例 SQL 20 为如下所示：

```
select t1.col1, t1.col2, t2.col3
  from t1
  right outer join t2 on (t1.col2 = t2.col2 and t1.col1 = 1);
```

如果用 Oracle 中自定义的外连接表示方式来等价改写范例 SQL 20 的话，就是如下所示的范例 SQL 26：

```
select t1.col1, t1.col2, t2.col3
  from t1, t2
 where t1.col2(+) = t2.col2
   and t1.col1(+) = 1;
```

对于上述范例 SQL 26 而言，额外的限制条件“t1.col1 = 1”中的目标列 COL1 后被加上了关键字“(+)”，这表示该限制条件会在表 T1 和表 T2 做外连接之前就被应用在表 T1 上，即在范例 SQL 26 中，参与外连接的 T1 中的数据是那些满足条件“t1.col1 = 1”的记录。

执行范例 SQL 26：

```
SQL> select t1.col1, t1.col2, t2.col3
  2    from t1, t2
  3   where t1.col2(+) = t2.col2
  4     and t1.col1(+) = 1;
COL1 COL2 COL3
---- ---- ----
 1    A    A2
           D2
           B2
```

从上述执行结果中我们可以看出，范例 SQL 26 和范例 SQL 20 的执行结果确实一模一样。

之前介绍过的范例 SQL 21 为如下所示：

```
select t1.col1, t1.col2, t2.col3
  from t1
 right outer join t2 on (t1.col2 = t2.col2)
 where t1.col1 = 1;
```

如果用 Oracle 中自定义的外连接表示方式来等价改写上述范例 SQL 21 的话，就是如下所示的范例 SQL 27：

```
select t1.col1, t1.col2, t2.col3
  from t1, t2
 where t1.col2(+) = t2.col2
   and t1.col1 = 1;
```

对于范例 SQL 27 而言，额外的限制条件“t1.col1 = 1”中的目标列 COL1 后并没有关键字“(+)”，这表示该限制条件会在表 T1 和 T2 做完外连接后才会被应用在表 T1 和 T2 的连接结果集上，即在范例 SQL 27 中，参与外连接的是表 T1 中的所有数据。

执行范例 SQL 27：

```
SQL> select t1.col1, t1.col2, t2.col3
  2    from t1, t2
  3   where t1.col2(+) = t2.col2
```

```
  4    and t1.col1 = 1;
COL1 COL2 COL3
---- ---- ----
  1    A    A2
```

从上述执行结果可以看出，范例 SQL 27 和范例 SQL 21 的执行结果确实一模一样。

这里需要注意的是，有些带了额外限制条件的目标 SQL 虽然看起来是外连接，但 Oracle 在实际执行时却可能会选择使用等价的内连接。

我们现在来看一下范例 SQL 26 的执行计划：

```
SQL> set autotrace traceonly
SQL> select t1.col1, t1.col2, t2.col3
  2    from t1, t2
  3   where t1.col2(+) = t2.col2
  4     and t1.col1(+) = 1;
执行计划
----------------------------------------------------------
Plan hash value: 1426054487
------------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     3 |    30 |    27   (4)| 00:00:01 |
|*  1 |  HASH JOIN OUTER   |      |     3 |    30 |    27   (4)| 00:00:01 |
|   2 |   TABLE ACCESS FULL| T2   |     3 |    15 |    13   (0)| 00:00:01 |
|*  3 |   TABLE ACCESS FULL| T1   |     1 |     5 |    13   (0)| 00:00:01 |
------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("T1"."COL2"(+)="T2"."COL2")
   3 - filter("T1"."COL1"(+)=1)
```

……省略显示部分内容

注意，上述显示内容中 Id = 1 的执行步骤 Operation 列的值为“HASH JOIN OUTER”，并且 Predicate Information 部分的谓词条件中均有关键字“(+)”，这说明 Oracle 在执行范例 SQL 26 时确实是在使用哈希外连接。

我们接着来看范例 SQL 27 的执行计划：

```
SQL> select t1.col1, t1.col2, t2.col3
  2    from t1, t2
  3   where t1.col2(+) = t2.col2
  4     and t1.col1 = 1;
执行计划
----------------------------------------------------------
Plan hash value: 1838229974
------------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |    10 |    27   (4)| 00:00:01 |
|*  1 |  HASH JOIN         |      |     1 |    10 |    27   (4)| 00:00:01 |
|*  2 |   TABLE ACCESS FULL| T1   |     1 |     5 |    13   (0)| 00:00:01 |
|   3 |   TABLE ACCESS FULL| T2   |     3 |    15 |    13   (0)| 00:00:01 |
------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("T1"."COL2"="T2"."COL2")
   2 - filter("T1"."COL1"=1)
```

……省略显示部分内容

注意，上述显示内容中 Id = 1 的执行步骤 Operation 列的值为“HASH JOIN”，并且其对应的谓词条件中并没有关键字“(+)”，这说明 Oracle 在执行范例 SQL 27 时并没有像在范例 SQL 26 中那样使用哈希外连接，而是使用哈希连接。

为什么这里 Oracle 不使用哈希外连接而是使用等价的哈希连接？因为对于范例 SQL 27 而言，其外连接条件“t1.col2(+) = t2.col2”中的关键字“(+)”出现在表 T1 的列 COL2 后面。这意味着表 T1 中那些不满足上述外连接条件并位于表 T1 中的查询列（即列 COL1 和 COL2）都会以 NULL 值来填充；同时该 SQL 的额外限制条件“t1.col1 = 1”中并没有关键字“(+)”，这表示该限制条件会在表 T1 和表 T2 做完外连接后才会被应用在表 T1 和表 T2 的连接结果集上，也就是说这个限制条件会把表 T1 中那些已经做完外连接且以 NULL 值来填充的所有记录（即表 T1 中所有列 COL1 值为 NULL 的记录）都给过滤掉。这样一来，范例 SQL 27 当然就可以使用等价的内连接来执行了（因为执行结果中根本就不会出现外连接所特有的以 NULL 值来填充的记录，即执行结果中不会出现 T1.COL1 的值为 NULL 的记录）。

我们在“1.2.4.1.1 内连接”中曾经介绍过 NATURAL JOIN，这里需要说明的是，NATURAL JOIN 不仅适用于内连接，也同样适用于外连接。

之前介绍过的范例 SQL 14 为如下所示：

```
select t1.col1, t1.col2, t2.col3
  from t1 left outer join t2 on (t1.col2 = t2.col2);
```

对于表 T1 和 T2 而言，它们之间的同名列为 COL2，所以上述范例 SQL 14 实际上和如下形式的范例 SQL 28 是等价的：

```
select t1.col1, col2, t2.col3
  from t1 natural left outer join t2;
```

执行范例 SQL 28：

```
SQL> select t1.col1, col2, t2.col3
  2    from t1 natural left outer join t2;
COL1 COL2 COL3
---- ---- ----
  1   A    A2
  2   B    B2
  3   C
```

从上述执行结果中我们可以看出，范例 SQL 28 和范例 SQL 14 的执行结果确实一模一样。

1.2.4.2　表连接的方法

之前已经介绍过，优化器在解析含表连接的目标 SQL 时，当它根据目标 SQL 的 SQL 文本的写法决定表连接的类型之后，接下来要做的事情之一就是决定表连接的方法。

在 Oracle 数据库中，两个表之间的表连接方法有排序合并连接、嵌套循环连接、哈希连接和笛卡儿连接这四种。这四种表连接方法各有其优缺点，也各有其适用场景，接下来，我们就来分别介绍它们。

1.2.4.2.1　排序合并连接

排序合并连接（Sort Merge Join）是一种两个表在做表连接时用排序操作（Sort）和合并操作（Merge）来

得到连接结果集的表连接方法。

如果两个表（这里将它们分别命名为表 T1 和表 T2）在做表连接时使用的是排序合并连接，则 Oracle 会依次顺序执行如下步骤。

（1）首先以目标 SQL 中指定的谓词条件（如果有的话）去访问表 T1，然后对访问结果按照表 T1 中的连接列来排序，排好序后的结果集我们记为结果集 1。

（2）接着以目标 SQL 中指定的谓词条件（如果有的话）去访问表 T2，然后对访问结果按照表 T2 中的连接列来排序，排好序后的结果集我们记为结果集 2。

（3）最后对结果集 1 和结果集 2 执行合并操作，从中取出匹配记录来作为排序合并连接的最终执行结果。

我个人认为这个合并操作的具体执行步骤是这样的：首先遍历结果集 1，即先取出结果集 1 中的第 1 条记录，然后去结果集 2 中按照连接条件判断是否存在匹配记录，然后再取出结果集 1 中的第 2 条记录，按照同样的连接条件再去结果集 2 中判断是否存在匹配的记录，直到最后遍历完结果集 1 中所有的记录。注意，这里去结果集 2 中判断是否存在匹配记录时会存在一个过滤的过程。因为结果集 2 已经按照表 T2 中的连接列排好序了，所以取出结果集 1 中的记录然后去找结果集 2 中的匹配记录时，只需要遍历结果集 2 中满足上述连接条件的那部分数据就可以了（这部分数据在结果集 2 中的存储位置肯定是在一起的），也就是说此时并不需要遍历结果集 2 中所有的记录，并且一旦在结果集 2 中找到匹配记录，就可以把该匹配记录从结果集 2 中删除，或者不删除但记录下其位置（这样下次再从结果集 2 中找匹配记录时就可以从这个位置开始了）。最后，结果集 1 和结果集 2 中所有的匹配结果就是上述排序合并连接的最终执行结果（**注意，这个合并过程的具体执行步骤是我猜的，我不确定是否正确**）。

对于排序合并连接的优缺点及适用场景，我们有如下总结。

- **通常情况下，排序合并连接的执行效率会远不如哈希连接，但前者的使用范围更广，因为哈希连接通常只能用于等值连接条件，而排序合并连接还能用于其他连接条件**（例如<、<=、>、>=）。
- **通常情况下，排序合并连接并不适合 OLTP 类型的系统，其本质原因是因为对于 OLTP 类型的系统而言，排序是非常昂贵的操作，当然，如果能避免排序操作，那么即使是 OLTP 类型的系统，也还是可以使用排序合并连接的**。比如两个表虽然是做排序合并连接，但实际上它们并不需要排序，因为这两个表在各自的连接列上都存在索引。
- 从严格意义上说，排序合并连接并不存在驱动表的概念，虽然我个人认为在执行合并的过程中，实际上还是存在驱动表和被驱动表的。

1.2.4.2.2 嵌套循环连接

嵌套循环连接（Nested Loops Join）是一种两个表在做表连接时依靠两层嵌套循环（分别为外层循环和内层循环）来得到连接结果集的表连接方法。

如果两个表（这里将它们分别命名为表 T1 和表 T2）在做表连接时使用的是嵌套循环连接，则 Oracle 会依次顺序执行如下步骤。

（1）首先，优化器会按照一定的规则来决定表 T1 和 T2 中谁是驱动表、谁是被驱动表。驱动表用于外层循环，被驱动表用于内层循环。这里假设驱动表是 T1，被驱动表是 T2。

（2）接着以目标 SQL 中指定的谓词条件（如果有的话）去访问驱动表 T1，访问驱动表 T1 后得到的结果

集我们记为驱动结果集 1。

（3）然后遍历驱动结果集 1 并同时遍历被驱动表 T2，即先取出驱动结果集 1 中的第 1 条记录，接着遍历被驱动表 T2 并按照连接条件去判断 T2 中是否存在匹配的记录，然后再取出驱动结果集 1 中的第 2 条记录，按照同样的连接条件再去遍历被驱动表 T2 并判断 T2 中是否还存在匹配的记录，直到遍历完驱动结果集 1 中所有的记录为止。这里的外层循环是指遍历驱动结果集 1 所对应的循环，内层循环是指遍历被驱动表 T2 所对应的循环。显然，外层循环所对应的驱动结果集 1 有多少条记录，遍历被驱动表 T2 的内层循环就要做多少次，这就是所谓的“嵌套循环”的含义。

对于嵌套循环连接的优缺点及适用场景，我们有如下总结。

- 从上述嵌套循环连接的具体执行过程可以看出：**如果驱动表所对应的驱动结果集的记录数较少，同时在被驱动表的连接列上又存在唯一性索引（或者在被驱动表的连接列上存在选择性很好的非唯一性索引），那么此时使用嵌套循环连接的执行效率就会非常高；但如果驱动表所对应的驱动结果集的记录数很多，即便在被驱动表的连接列上存在索引，此时使用嵌套循环连接的执行效率也不会高。**
- 只要驱动结果集的记录数较少，那就具备了做嵌套循环连接的前提条件，而驱动结果集是在对驱动表应用了目标 SQL 中指定的谓词条件（如果有的话）后所得到的结果集，**所以大表也可以作为嵌套循环连接的驱动表，关键看目标 SQL 中指定的谓词条件（如果有的话）能否将驱动结果集的数据量降下来。**
- **嵌套循环连接有其他连接方法所没有的一个优点：嵌套循环连接可以实现快速响应，**即它可以第一时间先返回已经连接过且满足连接条件的记录，而不必等待所有的连接操作全部做完后才返回连接结果。虽然排序合并连接和哈希连接也可以先返回已经连接过且满足连接条件的记录，而不必等待所有的连接操作都做完，但是它们并不是第一时间返回，因为排序合并连接要等到排完序后做合并操作时才能开始返回数据，而哈希连接则要等到驱动结果集所对应的 Hash Table 全部建完后才能开始返回数据。

如果 Oracle 使用的是嵌套循环连接，且在被驱动表的连接列上存在索引，那么 Oracle 在访问该索引时通常会使用单块读，这意味着嵌套循环连接的驱动结果集有多少条记录，Oracle 就需要访问该索引多少次。另外，如果目标 SQL 中的查询列并不能全部从被驱动表的相关索引中获得，那么 Oracle 在做完嵌套循环连接后就还需要对被驱动表执行回表操作（即用连接结果集中每一条记录所含的 ROWID 去回表获取被驱动表中的相关查询列）。这个回表操作通常也会使用单块读，这意味着做完嵌套循环连接后的连接结果集有多少条记录，Oracle 就需要回表多少次。

对于这种单块读而言，如果待访问的索引块或数据块不在 Buffer Cache 中，Oracle 就需要耗费物理 I/O 去相应的数据文件中获取。显然，在单块读的数量不降低的情况下，如果能减少这种单块读所需要耗费的物理 I/O 数量，那么嵌套循环连接的执行效率也会随之提高。

为了提高嵌套循环连接的执行效率，**在 Oracle 11g 中，Oracle 引入了向量 I/O（Vector I/O）。在引入向量 I/O 后，Oracle 就可以将原先一批单块读所需要耗费的物理 I/O 组合起来，然后用一个向量 I/O 去批量处理它们，这样就实现了在单块读的数量不降低的情况下减少这些单块读所需要耗费的物理 I/O 数量，也就提高了嵌套循环连接的执行效率。**

向量 I/O 的引入也反映在嵌套循环连接所对应的执行计划上。在 Oracle 11g 中，你会发现明明一次嵌套循环连接就可以处理完毕的 SQL，但其执行计划的显示内容中嵌套循环连接的数量却由之前的一个变为了现在的两个。

这里还是以“1.2.4.1.1 内连接”中的测试表 T1 和 T2 为例来说明。我们在表 T2 的列 COL2 上创建一个名

为 IDX_T2 的索引：

```
SQL> create index idx_t2 on t2(col2);

Index created
```

表 T1、T2 所在 Oracle 数据库的版本为 Oracle 11.2.0.1：

```
SQL> select * from v$version;
BANNER
--------------------------------------------------------------------------------
Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Production
PL/SQL Release 11.2.0.1.0 - Production
CORE 11.2.0.1.0    Production

TNS for 32-bit Windows: Version 11.2.0.1.0 - Production
NLSRTL Version 11.2.0.1.0 - Production
```

之前在“1.2.4.1.1 内连接”中介绍的范例 SQL 10 为如下所示：

```
select t1.col1, t1.col2, t2.col3
  from t1, t2
 where t1.col2 = t2.col2 ;
```

我们现在在范例 SQL 10 中加入让其走嵌套循环连接的 Hint（即/*+ ordered use_nl(t2) */），并实际执行一次：

```
SQL> set autotrace traceonly
SQL> select /*+ ordered use_nl(t2) */t1.col1, t1.col2, t2.col3
  2    from t1, t2
  3   where t1.col2 = t2.col2;
执行计划
----------------------------------------------------------
Plan hash value: 1054738919

-------------------------------------------------------------------------------------------
| Id  | Operation                     | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |         |     3 |    30 |    16   (0)| 00:00:01 |
|   1 |  NESTED LOOPS                 |         |       |       |            |          |
|   2 |   NESTED LOOPS                |         |     3 |    30 |    16   (0)| 00:00:01 |
|   3 |    TABLE ACCESS FULL          | T1      |     3 |    15 |    13   (0)| 00:00:01 |
|*  4 |    INDEX RANGE SCAN           | IDX_T2  |     1 |       |     0   (0)| 00:00:01 |
|   5 |   TABLE ACCESS BY INDEX ROWID| T2      |     1 |     5 |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------

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

   4 - access("T1"."COL2"="T2"."COL2")
```

……省略显示部分内容

注意，上述执行计划的显示内容中关键字“NESTED LOOPS”出现了两次，这说明在 Oracle 11gR2 中执行上述 SQL 时确实用了两次嵌套循环连接。这里第二次嵌套循环连接的被驱动表部分所对应的执行步骤是“TABLE ACCESS BY INDEX ROWID | T2”，这可能是因为 Oracle 想表达此时在通过 ROWID 回表访问表 T2 时是把一批 ROWID 组合起来后通过一个向量 I/O 批量回表，而不是每拿到一个 ROWID 就用一次单块读回表。

我们现在在范例 SQL 10 中加入 OPTIMIZER_FEATURES_ENABLE Hint（即

optimizer_features_enable('9.2.0')）后再次执行：

```
SQL> select /*+ optimizer_features_enable('9.2.0') ordered use_nl(t2) */t1.col1,t1.col2,
t2.col3
  2    from t1, t2
  3   where t1.col2 = t2.col2;
执行计划
----------------------------------------------------------
Plan hash value: 2253255382

-------------------------------------------------------------------------------------------
| Id  | Operation                    | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |         |     3 |    30 |    17   (6)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T2      |     1 |     5 |     2  (50)| 00:00:01 |
|   2 |   NESTED LOOPS               |         |     3 |    30 |    17   (6)| 00:00:01 |
|   3 |    TABLE ACCESS FULL         | T1      |     3 |    15 |    14   (8)| 00:00:01 |
|*  4 |    INDEX RANGE SCAN          | IDX_T2  |     1 |       |     1 (100)| 00:00:01 |
-------------------------------------------------------------------------------------------

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

   4 - access("T1"."COL2"="T2"."COL2")

……省略显示部分内容
```

上述 OPTIMIZER_FEATURES_ENABLE Hint 的作用是让解析 SQL 10 的优化器的版本回退到 Oracle 9iR2，这样我们就可以观察到同一个 SQL 在 Oracle 11g 之前执行嵌套循环连接时的情况。

注意，上述执行计划的显示内容中关键字“NESTED LOOPS”只出现了一次，这说明在 Oracle 9iR2 中执行上述 SQL 时确实只用了一次嵌套循环连接。

1.2.4.2.3　哈希连接

哈希连接（Hash Join）是一种两个表在做表连接时主要依靠哈希运算来得到连接结果集的表连接方法。

在 Oracle 7.3 之前，Oracle 数据库中的常用表连接方法就只有排序合并连接和嵌套循环连接这两种，但这两种方法都各有其明显缺陷。对于排序合并连接，如果两个表在施加了目标 SQL 中指定的谓词条件（如果有的话）后得到的结果集很大且需要排序，则排序合并连接的执行效率一定不高；而对于嵌套循环连接，如果驱动表所对应的驱动结果集的记录数很大，即便在被驱动表的连接列上存在索引，此时使用嵌套循环连接的执行效率也会同样不高。

为了解决排序合并连接和嵌套循环连接在上述情形下执行效率不高的问题，同时也为了给优化器提供一种新的选择，Oracle 在 Oracle 7.3 中引入了哈希连接。从理论上来说，哈希连接的执行效率会比排序合并连接和嵌套循环连接要高，当然，实际情况并不总是这样。

在 Oracle 10g 及其以后的 Oracle 数据库版本中，优化器（实际上是 CBO，因为哈希连接仅适用于 CBO）在解析目标 SQL 时是否考虑哈希连接是受限于隐含参数_HASH_JOIN_ENABLED，而在 Oracle 10g 以前，CBO 在解析目标 SQL 时是否考虑哈希连接则是受限于参数 HASH_JOIN_ENABLED。

_HASH_JOIN_ENABLED 的默认值是 TRUE，表示允许 CBO 在解析目标 SQL 时考虑哈希连接。当然，即使将该参数的值改成了 FALSE，使用 USE_HASH Hint 依然可以让 CBO 在解析目标 SQL 时考虑哈希连接，这说明 USE_HASH Hint 的优先级比参数_HASH_JOIN_ENABLED 的优先级要高。

如果两个表（这里将它们分别命名为表 T1 和表 T2）在做表连接时使用的是哈希连接，则 Oracle 会依次顺序执行如下步骤。

（1）首先 Oracle 会根据参数 HASH_AREA_SIZE、DB_BLOCK_SIZE 和_HASH_MULTIBLOCK_IO_COUNT 的值来决定 Hash Partition 的数量（Hash Partition 是一个逻辑上的概念，它实际上是一组 Hash Bucket 的集合。所有 Hash Partition 的集合就被称为 Hash Table，即一个 Hash Table 由多个 Hash Partition 所组成，而一个 Hash Partition 又是由多个 Hash Bucket 所组成的）。

（2）表 T1 和 T2 在施加了目标 SQL 中指定的谓词条件（如果有的话）后，得到的结果集中数据量较少的那个结果集会被 Oracle 选为哈希连接的驱动结果集，这里我们假设 T1 所对应的结果集的数据量相对较少，记为 S；T2 所对应的结果集的数据量相对较多，记为 B。显然这里 S 是驱动结果集，B 是被驱动结果集。

（3）接着 Oracle 会遍历 S，读取 S 中的每一条记录，并对每一条记录按照该记录在表 T1 中的连接列做哈希运算。这个哈希运算会使用两个内置哈希函数，这两个哈希函数会同时对该连接列计算哈希值，我们把这两个内置哈希函数分别记为 hash_func_1 和 hash_func_2，它们所计算出来的哈希值分别记为 hash_value_1 和 hash_value_2。

（4）然后 Oracle 会按照 hash_value_1 的值把相应的 S 中的对应记录存储在不同 Hash Partition 的不同 Hash Bucket 里，同时和该记录存储在一起的还有该记录用 hash_func_2 计算出来的 hash_value_2。注意，存储在 Hash Bucket 里的记录并不是目标表的完整行记录，只需要存储位于目标 SQL 中与目标表相关的查询列和连接列就足够了。我们把 S 所对应的每一个 Hash Partition 记为 Si。

（5）在构建 Si 的同时，Oracle 会构建一个位图（BITMAP），这个位图用来标记 Si 所包含的每一个 Hash Bucket 是否有记录（即记录数是否大于 0）。

（6）如果 S 的数据量很大，那么在构建 S 所对应的 Hash Table 时，就可能会出现 PGA 的工作区（WORK AREA）被填满的情况。这时候 Oracle 会把工作区中包含记录数最多的 Hash Partition 写到磁盘上（TEMP 表空间）。接着 Oracle 会继续构建 S 所对应的 Hash Table，在继续构建的过程中，如果工作区又满了，则 Oracle 会继续重复上述动作，即挑选包含记录数最多的 Hash Partition 并写回到磁盘上。如果要构建的记录所对应的 Hash Partition 已经事先被 Oracle 写回磁盘，则此时 Oracle 就会去磁盘上更新该 Hash Partition，即把该条记录和 hash_value_2 直接加到这个已经位于磁盘上的 Hash Partition 的相应 Hash Bucket 中。注意，极端情况下可能会出现只有某个 Hash Partition 的部分记录还在内存中，该 Hash Partition 的剩余部分和余下的所有 Hash Partition 都已经被写回到磁盘上。

（7）上述构建 S 所对应的 Hash Table 的过程会一直持续下去，直到遍历完 S 中的所有记录为止。

（8）接着，Oracle 会对所有的 Si 按照它们所包含的记录数来排序，然后把这些已经排好序的 Hash Partition 按顺序依次且尽可能全部放到内存中（PGA 的工作区），当然，如果实在放不下，放不下的那部分 Hash Partition 还是会位于磁盘上。我个人认为这个按照 Si 的记录数来排序的动作不是必须要做的，因为这个排序动作的根本目的就是为了尽可能多地把那些记录数较小的 Hash Partition 保留在内存中，而将那些已经被写回磁盘、记录数较大且现有内存已经放不下的 Hash Partition 保留在磁盘上，显然，如果所有的 Si 本来就都在内存中，也没发生过将 Si 写回到磁盘的操作，这里根本就不需要排序了。

（9）至此 Oracle 已经处理完 S，现在可以开始处理 B 了。

（10）Oracle 会遍历 B，读取 B 中的每一条记录，并按照该记录在表 T2 中的连接列做哈希运算，这个哈

希运算和步骤 3 中的哈希运算是一模一样的，即还是会用步骤 3 中的 hash_func_1 和 hash_func_2，并且也会计算出两个哈希值 hash_value_1 和 hash_value_2。

接着 Oracle 会按照该记录所对应的哈希值 hash_value_1 去 Si 里找匹配的 Hash Bucket；如果能找到匹配的 Hash Bucket，则 Oracle 还会遍历该 Hash Bucket 中的每一条记录，并校验存储于该 Hash Bucket 中的每一条记录的连接列，看是否是真的匹配（**即这里要校验 S 和 B 中的匹配记录所对应的连接列是否真的相等，因为对于哈希运算而言，不同的值经过哈希运算后的结果可能是相同的**）。如果是真的匹配，则上述 hash_value_1 所对应 B 中记录的位于目标 SQL 中的查询列和该 Hash Bucket 中的匹配记录便会组合起来，一起作为满足目标 SQL 连接条件的记录返回。如果找不到匹配的 Hash Bucket，则 Oracle 就会去访问步骤 5 中构建的位图。

如果位图显示该 Hash Bucket 在 Si 中对应的记录数大于 0，则说明该 Hash Bucket 虽然不在内存中，但它已经被写回磁盘，则此时 Oracle 就会按照 hash_value_1 的值把相应 B 中的对应记录也以 Hash Partition 的方式写回到磁盘上，同时和该记录存储在一起的还有该记录用 hash_func_2 计算出来的 hash_value_2 的值。如果位图显示该 Hash Bucket 在 Si 中对应的记录数等于 0，则 Oracle 就无须把上述 hash_value_1 所对应 B 中的记录写回磁盘了，因为这条记录必然不满足目标 SQL 的连接条件。**这个根据位图来决定是否将 hash_value_1 所对应 B 中的记录写回到磁盘的动作就是所谓的“位图过滤”**（Oracle 不一定会启用位图过滤，因为如果所有的 Si 本来就都在内存中，也没发生过将 Si 写回到磁盘的操作，那么这里 Oracle 就不需要启用位图过滤了）。**我们把 B 所对应的每一个 Hash Partition 记为 Bj**。

（11）上述去 Si 中查找匹配 Hash Bucket 和构建 Bj 的过程会一直持续下去，直到遍历完 B 中的所有记录为止。

（12）至此 Oracle 已经处理完所有位于内存中的 Si 和对应的 Bj，现在只剩下位于磁盘上的 Si 和 Bj 还未处理。

（13）因为在构建 Si 和 Bj 时用的是同样的哈希函数 hash_func_1 和 hash_func_2，所以 Oracle 在处理位于磁盘上的 Si 和 Bj 的时候可以放心地配对处理，即只有对应 Hash Partition Number 值相同的 Si 和 Bj 才可能会产生满足连接条件的记录。**这里我们用 Sn 和 Bn 来表示位于磁盘上且对应 Hash Partition Number 值相同的 Si 和 Bj**。

（14）对于每一对 Sn 和 Bn，它们之中记录数较少的会被当作驱动结果集，然后 Oracle 会用这个驱动结果集 Hash Bucket 里记录的 hash_value_2 来构建新的 Hash Table，另外一个记录数较多的会被当作被驱动结果集，然后 Oracle 会用这个被驱动结果集 Hash Bucket 里记录的 hash_value_2 去上述构建的新 Hash Table 中找匹配记录。注意，**对每一对 Sn 和 Bn 而言，Oracle 始终会选择它们中记录数较少的来作为驱动结果集，所以每一对 Sn 和 Bn 的驱动结果集都可能会发生变化，这就是所谓的“动态角色互换”**。

（15）步骤 14 中如果存在匹配记录，则该匹配记录也会作为满足目标 SQL 连接条件的记录返回。

（16）上述处理 Sn 和 Bn 的过程会一直持续下去，直到遍历完所有的 Sn 和 Bn 为止。

对于哈希连接的优缺点及适用场景，我们有如下总结。

- 哈希连接不一定会排序，或者说大多数情况下都不需要排序。
- 哈希连接的驱动表所对应的连接列的可选择性应尽可能好，因为这个可选择性会影响对应 Hash Bucket 中的记录数，而 Hash Bucket 中的记录数又会直接影响从该 Hash Bucket 中查找匹配记录的效率。**如果一个 Hash Bucket 里所包含的记录数过多，则可能会严重降低所对应哈希连接的执行效率，此时典型**

的表现就是该哈希连接执行了很长时间都没有结束，数据库所在数据库服务器上的 CPU 占用率很高，但目标 SQL 所消耗的逻辑读却很低，因为此时大部分时间都耗费在了遍历上述 Hash Bucket 里的所有记录上，而遍历 Hash Bucket 里的记录这个动作发生在 PGA 的工作区里，所以不耗费逻辑读。

- 哈希连接只适用于 CBO，它也只能用于等值连接条件（即使是哈希反连接，Oracle 实际上也是将其转换成了等价的等值连接）。
- 哈希连接很适合于小表和大表之间做表连接且连接结果集的记录数较多的情形，特别是在小表的连接列的可选择性非常好的情况下，这时候哈希连接的执行时间就可以近似看作是和全表扫描那个大表所耗费的时间相当。
- 当两个表做哈希连接时，如果在施加了目标 SQL 中指定的谓词条件（如果有的话）后得到的数据量较小的那个结果集所对应的 Hash Table 能够完全被容纳在内存中（PGA 的工作区），则此时的哈希连接的执行效率会非常高。

我们可以借助于 10104 事件所产生的 trace 文件来观察目标 SQL 在做哈希连接时的大致过程和一些统计信息（比如用了多少个 Hash Partition、多少个 Hash Bucket 以及各个 Hash Bucket 都分别有多少条记录等），10104 事件在实际诊断哈希连接的性能问题时非常有用。

使用 10104 事件观察目标 SQL 做哈希连接的具体过程为：

```
oradebug setmypid
oradebug event 10104 trace name context forever, level 1
set autotrace traceonly
实际执行目标 SQL（必须要实际执行该 SQL，不能用 explain plan for）
oradebug tracefile_name
```

一个典型的 10104 事件所产生的 trace 文件内容如下所示：

```
kxhfInit(): enter
kxhfInit(): exit
*** RowSrcId: 1 HASH JOIN STATISTICS (INITIALIZATION) ***
Join Type: INNER join
Original hash-area size: 3642760
Memory for slot table: 2826240
Calculated overhead for partitions and row/slot managers: 816520
Hash-join fanout: 8
Number of partitions: 8
Number of slots: 23
Multiblock IO: 15
Block size(KB): 8
Cluster (slot) size(KB): 120
Minimum number of bytes per block: 8160
Bit vector memory allocation(KB): 128
Per partition bit vector length(KB): 16
……省略显示部分内容
  Slot table resized: old=23 wanted=12 got=12 unload=0
*** RowSrcId: 1 HASH JOIN RESIZE BUILD (PHASE 1) ***
Total number of partitions: 8
Number of partitions which could fit in memory: 8
```

```
    Number of partitions left in memory: 8
    Total number of slots in in-memory partitions: 8
    kxhfResize(enter): resize to 14 slots (numAlloc=8, max=12)
    kxhfResize(exit): resized to 14 slots (numAlloc=8, max=14)
      set work area size to: 2215K (14 slots)
    *** RowSrcId: 1 HASH JOIN BUILD HASH TABLE (PHASE 1) ***
    Total number of partitions: 8
    Number of partitions left in memory: 8
    Total number of rows in in-memory partitions: 1000
       (used as preliminary number of buckets in hash table)
    Estimated max # of build rows that can fit in avail memory: 79800
    ### Partition Distribution ###
    Partition:0    rows:120      clusters:1     slots:1     kept=1
    Partition:1    rows:122      clusters:1     slots:1     kept=1
    ……省略显示部分内容
    Partition:6    rows:118      clusters:1     slots:1     kept=1
    Partition:7    rows:137      clusters:1     slots:1     kept=1
    *** (continued) HASH JOIN BUILD HASH TABLE (PHASE 1) ***
    Revised number of hash buckets (after flushing): 1000
    Allocating new hash table.
    *** (continued) HASH JOIN BUILD HASH TABLE (PHASE 1) ***
    Requested size of hash table: 256
    Actual size of hash table: 256
    Number of buckets: 2048
    Match bit vector allocated: FALSE
    *** (continued) HASH JOIN BUILD HASH TABLE (PHASE 1) ***
    Total number of rows (may have changed): 1000
    Number of in-memory partitions (may have changed): 8
    Final number of hash buckets: 2048
    Size (in bytes) of hash table: 8192
    qerhjBuildHashTable(): done hash-table on partition=7, index=0 last_slot#=3 rows=137
total_rows=137
    qerhjBuildHashTable(): done hash-table on partition=6, index=1 last_slot#=4 rows=118
total_rows=255
    ……省略显示部分内容
    qerhjBuildHashTable(): done hash-table on partition=1, index=6 last_slot#=2 rows=122
total_rows=880
    qerhjBuildHashTable(): done hash-table on partition=0, index=7 last_slot#=5 rows=120
total_rows=1000
    kxhfIterate(end_iterate): numAlloc=8, maxSlots=14
    *** (continued) HASH JOIN BUILD HASH TABLE (PHASE 1) ***
    ### Hash table ###
    # NOTE: The calculated number of rows in non-empty buckets may be smaller
    #      than the true number.
    Number of buckets with   0 rows:      1249
    Number of buckets with   1 rows:       626
```

```
Number of buckets with   2 rows:       149
Number of buckets with   3 rows:        21
Number of buckets with   4 rows:         3
Number of buckets with   5 rows:         0
……省略显示部分内容
Number of buckets with between  90 and  99 rows:         0
Number of buckets with 100 or more rows:         0
### Hash table overall statistics ###
Total buckets: 2048 Empty buckets: 1249 Non-empty buckets: 799
Total number of rows: 1000
Maximum number of rows in a bucket: 4
Average number of rows in non-empty buckets: 1.251564
Disabled bitmap filtering: filtered rows=0 minimum required=50 out of=1000
qerhjFetch: max probe row length (mpl=0)
*** RowSrcId: 1, qerhjFreeSpace(): free hash-join memory
kxhfRemoveChunk: remove chunk 0 from slot table
```

注意上述显示内容中用粗体标出的部分，如“Number of in-memory partitions (may have changed): 8”、“Final number of hash buckets: 2048”、“Total buckets: 2048 Empty buckets: 1249 Non-empty buckets: 799”、“Total number of rows: 1000”、“Maximum number of rows in a bucket: 4”、“Disabled bitmap filtering: filtered rows=0 minimum required=50 out of=1000”等，这说明上述哈希连接驱动结果集的记录数为 1,000，共有 8 个 Hash Partition、2,048 个 Hash Bucket，这 2,048 个 Hash Bucket 中有 1,249 个是空的（即没有记录），799 个有记录，包含记录数最多的那个 Hash Bucket 所含记录的数量为 4，以及上述哈希连接并没有启用位图过滤。

1.2.4.2.4 笛卡儿连接

笛卡儿连接（Cross Join）又称为笛卡儿乘积（Cartesian Product），它是一种两个表在做表连接时没有任何连接条件的表连接方法。

如果两个表（这里将它们分别命名为表 T1 和表 T2）在做表连接时使用的是笛卡儿连接，则 Oracle 会依次顺序执行如下步骤。

（1）首先以目标 SQL 中指定的谓词条件（如果有的话）访问表 T1，此时得到的结果集我们记为结果集 1，这里假设结果集 1 的记录数为 m。

（2）接着以目标 SQL 中指定的谓词条件（如果有的话）访问表 T2，此时得到的结果集我们记为结果集 2，这里假设结果集 2 的记录数为 n。

（3）最后对结果集 1 和结果集 2 执行合并操作，从中取出匹配记录来作为笛卡儿连接的最终执行结果。这里的特殊之处在于对于笛卡儿连接而言，因为没有表连接条件，所以在对结果集 1 和结果集 2 执行合并操作时，对于结果集 1 中的任意一条记录，结果集 2 中的所有记录都满足条件，即它们都会是匹配记录，所以上述笛卡儿连接的连接结果的记录数就是 m 和 n 的乘积（即 $m \times n$）。

从上述笛卡儿连接的执行过程我们可以看出，**笛卡儿连接实际上是一种特殊的“合并连接”，这里的“合并连接”和排序合并连接类似，只不过笛卡儿连接不需要排序，并且在执行合并操作时没有连接条件而已**。关于这一点，实际上可以从笛卡儿连接所对应的执行计划中看出些端倪。

这里还是以“1.2.4.1.1 内连接”中的测试表 T1 和 T2 为例来说明。表 T1 和 T2 还是和原来一样，各有 3 条记录，我们执行如下不带连接条件的 SQL：

```
SQL> set autotrace on
SQL> select t1.col1,t2.col3 from t1,t2;

  COL1   COL3
------- --------
     1    A2
     2    A2
     3    A2
     1    B2
     2    B2
     3    B2
     1    D2
     2    D2
     3    D2

已选择9行。

执行计划
----------------------------------------------------------
Plan hash value: 1323614827

-------------------------------------------------------------------------------
| Id  | Operation            | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |      |     9 |    54 |    46   (0)| 00:00:01 |
|   1 |  MERGE JOIN CARTESIAN|      |     9 |    54 |    46   (0)| 00:00:01 |
|   2 |   TABLE ACCESS FULL  | T2   |     3 |     9 |    13   (0)| 00:00:01 |
|   3 |   BUFFER SORT        |      |     3 |     9 |    33   (0)| 00:00:01 |
|   4 |    TABLE ACCESS FULL | T1   |     3 |     9 |    11   (0)| 00:00:01 |
-------------------------------------------------------------------------------

……省略显示部分内容
```

上述 SQL 的执行结果包含 9 条记录，这刚好就是表 T1 和表 T2 记录数的乘积（9＝3×3）。

上述执行计划和排序合并连接所对应的执行计划非常相似，并且 Id ＝ 1 的 Operation 列的值为“MERGE JOIN CARTESIAN”，只是在排序合并连接的合并部分所对应的关键字“MERGE JOIN”后添加了一个单词“CARTESIAN”，所以我们才说笛卡儿连接实际上就是一种特殊的“合并连接”。

标准 SQL 用关键字“CROSS JOIN”来表示笛卡儿连接，这里我们用标准 SQL 的方式来改写上述 SQL 并再执行一次：

```
SQL> select t1.col1,t2.col3 from t1 cross join t2;

  COL1   COL3
------- --------
     1    A2
     2    A2
     3    A2
     1    B2
     2    B2
     3    B2
     1    D2
     2    D2
     3    D2

已选择9行。

执行计划
----------------------------------------------------------
Plan hash value: 1323614827

-------------------------------------------------------------------------------
| Id  | Operation            | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
```

```
|   0 | SELECT STATEMENT      |      |     9 |    54 |    46   (0)| 00:00:01 |
|   1 |  MERGE JOIN CARTESIAN|      |     9 |    54 |    46   (0)| 00:00:01 |
|   2 |   TABLE ACCESS FULL   | T2   |     3 |     9 |    13   (0)| 00:00:01 |
|   3 |   BUFFER SORT         |      |     3 |     9 |    33   (0)| 00:00:01 |
|   4 |    TABLE ACCESS FULL | T1   |     3 |     9 |    11   (0)| 00:00:01 |
-------------------------------------------------------------------------------
```

……省略显示部分内容

从上述显示内容可以看出，用标准 SQL 改写后其执行结果和执行计划与原先的确实是一模一样的。

对于笛卡儿连接的优缺点及适用场景，我们有如下总结。

（1）笛卡儿连接的出现通常是由于目标 SQL 中漏写了表连接条件，所以笛卡儿连接一般是不好的，除非刻意这样做（比如有些情况下可以利用笛卡儿连接来减少对目标 SQL 中大表的全表扫描次数）。

（2）有时候出现笛卡儿连接是因为在目标 SQL 中使用了 ORDERED Hint，同时在该 SQL 的 SQL 文本中位置相邻的两个表之间又没有直接的关联条件。

（3）有时候出现笛卡儿连接是因为目标 SQL 中相关表的统计信息不准。比如三个表 T1、T2、T3 做表连接，T1 和 T2 的连接条件为 T1.ID1=T2.ID1，T2 和 T3 的连接条件为 T2.ID2=T3.ID2，同时在表 T2 的连接列 ID1 和 ID2 上存在一个包含这两个连接列的组合索引。如果表 T1 和 T3 的统计信息不准，导致 Oracle 认为表 T1 和 T3 都只有很少量的记录（比如都只有 1 条记录），则此时 Oracle 很可能会选择先对表 T1 和 T3 做笛卡儿连接，然后再和表 T2 做表连接。因为 Oracle 认为表 T1 和 T3 做笛卡儿连接后连接结果集的 Cardinality 的值是 1，并且连接结果中会同时包含列 ID1 和列 ID2，这意味着此时 Oracle 就可以利用表 T2 中的上述组合索引了。这种笛卡儿连接通常是有问题的，还是拿这个例子来说，如果表 T1 和表 T3 的实际记录数并不都是 1，而全部是 1000，那么此时表 T1 和表 T3 做笛卡儿连接的连接结果集的 Cardinality 的值就将是 100 万，显然这种情况下如果还是按照笛卡儿连接的方式来执行的话，则该 SQL 的执行效率就会受到严重影响。

1.2.4.3 反连接

反连接（Anti Join）是一种特殊的连接类型，与内连接和外连接不同，Oracle 数据库里并没有相关的关键字可以在 SQL 文本中专门表示反连接，所以这里把它单独拿出来说明。

为了方便说明反连接的含义，我们用“t1.x *anti*= t2.y”来表示表 T1 和 T2 做反连接，且 T1 是驱动表，T2 是被驱动表，反连接条件为 t1.x=t2.y。这里“t1.x *anti*= t2.y”的含义是只要表 T2 中有满足条件 t1.x=t2.y 的记录存在，则表 T1 中满足条件 t1.x=t2.y 的记录就会被丢弃，最后返回的记录就是表 T1 中那些不满足条件 t1.x=t2.y 的记录。

当做子查询展开时，Oracle 经常会把那些外部 where 条件为 NOT EXISTS、NOT IN 或<> ALL 的子查询转换成对应的反连接（关于子查询展开，会在第 4 章的“4.2 子查询展开”中详细说明，这里不再赘述）。

我们来看如下的范例 SQL 29、30 和 31（这里还是以“1.2.4.1.1 内连接”中的测试表 T1 和 T2 为例来说明）。

范例 SQL 29：

```
select * from t1
 where col2 not in (select col2 from t2);
```

范例 SQL 30：

```
select * from t1
 where col2 <> all (select col2 from t2);
```

范例 SQL 31：

```
select * from t1
 where not exists (select 1 from t2 where col2 = t1.col2);
```

现在表 T1 和 T2 在各自的连接列 COL2 上均没有 NULL 值，在这种情况下范例 SQL 29、30、31 实际上是等价的，我们来执行它们：

```
SQL> set autotrace on
SQL> select * from t1
  2   where col2 not in (select col2 from t2);

      COL1 C
---------- -
         3 C

执行计划
----------------------------------------------------------
Plan hash value: 1275484728

-------------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |     7 |    27   (4)| 00:00:01 |
|*  1 |  HASH JOIN ANTI NA |      |     1 |     7 |    27   (4)| 00:00:01 |
|   2 |   TABLE ACCESS FULL| T1   |     3 |    15 |    13   (0)| 00:00:01 |
|   3 |   TABLE ACCESS FULL| T2   |     3 |     6 |    13   (0)| 00:00:01 |
-------------------------------------------------------------------------------

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

   1 - access("COL2"="COL2")

……省略显示部分内容

SQL> select * from t1
  2   where col2 <> all (select col2 from t2);

      COL1 C
---------- -
         3 C

执行计划
----------------------------------------------------------
Plan hash value: 1275484728

-------------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |     7 |    27   (4)| 00:00:01 |
|*  1 |  HASH JOIN ANTI NA |      |     1 |     7 |    27   (4)| 00:00:01 |
|   2 |   TABLE ACCESS FULL| T1   |     3 |    15 |    13   (0)| 00:00:01 |
|   3 |   TABLE ACCESS FULL| T2   |     3 |     6 |    13   (0)| 00:00:01 |
-------------------------------------------------------------------------------

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

   1 - access("COL2"="COL2")

……省略显示部分内容

SQL> select * from t1
  2   where not exists (select 1 from t2 where col2 = t1.col2);
```

```
      COL1 C
---------- -
         3 C

执行计划
----------------------------------------------------------
Plan hash value: 2706079091

-------------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |     7 |    27   (4)| 00:00:01 |
|*  1 |  HASH JOIN ANTI    |      |     1 |     7 |    27   (4)| 00:00:01 |
|   2 |   TABLE ACCESS FULL| T1   |     3 |    15 |    13   (0)| 00:00:01 |
|   3 |   TABLE ACCESS FULL| T2   |     3 |     6 |    13   (0)| 00:00:01 |
-------------------------------------------------------------------------------

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

   1 - access("COL2"="T1"."COL2")

……省略显示部分内容
```

上述三个范例 SQL 的执行结果是一样的，范例 SQL 29、30 和范例 SQL 31 的执行计划中，Id = 1 的执行步骤的列 Operation 的值分别为“HASH JOIN ANTI NA”和“HASH JOIN ANTI”，虽然不是完全一样，但它们都有关键字“ANTI”，这就说明 Oracle 在执行上述三个范例 SQL 时确实是在用反连接，即 Oracle 在执行时实际上是将它们转换成了如下的等价反连接形式：

```
select t1.* from t1,t2
 where t1.col2 anti= t2.col2
```

这里表 T1、T2 在各自的连接列 COL2 上没有 NULL 值，所以此时这三个范例 SQL 是等价的，但如果连接列 COL2 上有 NULL 值，则它们就不完全等价了。这种 NULL 值所带来的影响又细分为两种情况。

1．表 T1 的连接列 COL2 上出现了 NULL 值。

往表 T1 里插入 1 条列 COL2 为 NULL 的记录：

```
SQL> insert into t1 values(4,null);

1 row inserted

SQL> commit;

Commit complete
```

现在表 T1 中的记录为如下所示：

```
SQL> select * from t1;

 COL1 COL2
----- ----
    1  A
    2  B
    3  C
    4
```

分别执行范例 SQL 29、30、31：

```
SQL> select * from t1 where col2 not in (select col2 from t2);
 COL1  COL2
------ ----
    3    C
SQL> select * from t1 where col2 <> all (select col2 from t2);
 COL1  COL2
------ ----
    3    C
SQL> select * from t1 where not exists (select 1 from t2 where col2 = t1.col2);
 COL1 COL2
----- ----
   4
   3   C
```

2．表 T2 的连接列 COL2 上出现了 NULL 值

先删除表 T1 里那条列 COL2 为 NULL 的记录：

```
SQL> delete from t1 where col1=4;

1 row deleted

SQL> commit;

Commit complete
```

然后往表 T2 里插入 1 条列 COL2 为 NULL 的记录：

```
SQL> insert into t2 values(null,'E2');

1 row inserted

SQL> commit;

Commit complete
```

现在表 T2 中的记录为如下所示：

```
SQL> select * from t2;
COL2 COL3
---- ----
  A   A2
  B   B2
  D   D2
      E2
```

再次执行范例 SQL 29、30、31：

```
SQL> select * from t1 where col2 not in (select col2 from t2);
COL1   COL2
------ ----
```

```
SQL> select * from t1 where col2 <> all (select col2 from t2);
 COL1 COL2
----- ----
SQL> select * from t1 where not exists (select 1 from t2 where col2 = t1.col2);
 COL1 COL2
----- ----
    3  C
```

最后删除表 T2 里那条列 COL2 为 NULL 的记录：

```
SQL> delete from t2 where col3='E2';

1 row deleted

SQL> commit;

Commit complete
```

从上述测试中我们可以得出如下结论。

（1）表 T1、T2 在各自的连接列 COL2 上一旦有了 NULL 值，则范例 SQL 29、30、31 就不完全等价了。

（2）NOT IN 和<> ALL 对 NULL 值敏感，这意味着 NOT IN 后面的子查询或者常量集合一旦有 NULL 值出现，则整个 SQL 的执行结果就会为 NULL，即此时的执行结果将不包含任何记录。

（3）NOT EXISTS 对 NULL 值不敏感，这意味着 NULL 值对 NOT EXISTS 的执行结果不会有什么影响。

正是因为 NOT IN 和<> ALL 对 NULL 值敏感，所以一旦相关的连接列上出现了 NULL 值，此时 Oracle 如果还按照通常的反连接的处理逻辑来处理，得到的结果就不对了。

为了解决 NOT IN 和<> ALL 对 NULL 值敏感的问题，Oracle 推出了改良的反连接，这种反连接能够处理 NULL 值，Oracle 称其为 Null-Aware Anti Join。上述范例 SQL 29、30 的执行计划中，Id = 1 的执行步骤的列 Operation 的值为“HASH JOIN ANTI NA”，关键字“NA”就是 Null-Aware 的缩写。Oracle 就是想告诉我们：这里采用的不是普通的哈希反连接，而是改良后的、能够处理 NULL 值的哈希反连接。

在 Oracle 11gR2 中，Oracle 是否启用 Null-Aware Anti Join 受隐含参数 _OPTIMIZER_NULL_AWARE_ANTIJOIN 控制，其默认值为 TRUE，表示启用 Null-Aware Anti Join。

如果我们把_OPTIMIZER_NULL_AWARE_ANTIJOIN 的值修改为 FALSE，则 Oracle 就不能再用 Null-Aware Anti Join 了，而又因为 NOT IN 对 NULL 值敏感，所以 Oracle 此时也不能用普通的反连接。

我们来验证一下。在当前 Session 中将_OPTIMIZER_NULL_AWARE_ANTIJOIN 的值修改为 FALSE：

```
SQL> alter session set "_optimizer_null_aware_antijoin" = false;

Session altered
```

然后再次执行范例 SQL 29：

```
SQL> select * from t1
  2  where col2 not in (select col2 from t2);
```

```
      COL1 C
---------- -
         3 C

执行计划
----------------------------------------------------------
Plan hash value: 895956251

-------------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     2 |    10 |    33   (0)| 00:00:01 |
|*  1 |  FILTER            |      |       |       |            |          |
|   2 |   TABLE ACCESS FULL| T1   |     3 |    15 |    13   (0)| 00:00:01 |
|*  3 |   TABLE ACCESS FULL| T2   |     1 |     2 |    13   (0)| 00:00:01 |
-------------------------------------------------------------------------------

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

   1 - filter( NOT EXISTS (SELECT 0 FROM "T2" "T2" WHERE
              LNNVL("COL2"<>:B1)))
   3 - filter(LNNVL("COL2"<>:B1))
```

……省略显示部分内容

从上述显示内容中可以看到，当我们把_OPTIMIZER_NULL_AWARE_ANTIJOIN 的值修改为 FALSE 后，Oracle 果然没有走反连接（当然也不能走）。

这里 Oracle 选择了走 FILTER 类型的执行计划，FILTER 类型的执行计划实际上是一种改良的嵌套循环连接，我们会在第 2 章的“2.5.5.4　FILTER”中详细介绍它，这里不再赘述。

1.2.4.4　半连接

半连接（Semi Join）是一种特殊的连接类型，与反连接一样，Oracle 数据库里也没有相关的关键字可以在 SQL 文本中专门表示半连接，所以这里也把它单独拿出来说明。

为了方便说明半连接的含义，这里我们用“t1.x *semi*= t2.y”来表示表 T1 和表 T2 做半连接，且 T1 是驱动表，T2 是被驱动表，半连接条件为 t1.x=t2.y。这里“t1.x *semi*= t2.y”的含义是只要在表 T2 中找到一条记录满足 t1.x=t2.y，则马上停止搜索表 T2，并直接返回表 T1 中满足条件 t1.x=t2.y 的记录。也就是说，表 T2 中满足半连接条件 t1.x=t2.y 的记录即使有多条，表 T1 中也只会返回第一条满足条件的记录。所以**半连接和普通的内连接不同，半连接实际上会去重**。

当做子查询展开时，Oracle 经常会把那些外部 where 条件为 EXISTS、IN 或= ANY 的子查询转换为对应的半连接（关于子查询展开，会在第 4 章的“4.2　子查询展开”中详细说明，这里不再赘述）。

我们来看如下 3 个范例 SQL（这里还是以“1.2.4.1.1　内连接”中的测试表 T1 和 T2 为例来说明）。

范例 SQL 32：

```
select * from t1
 where col2 in (select col2 from t2);
```

范例 SQL 33：

```
select * from t1
 where col2 = any (select col2 from t2);
```

范例 SQL 34：

```
select * from t1
 where exists (select 1 from t2 where col2 = t1.col2);
```

上述范例 SQL 32、33、34 实际上是等价的，我们现在来执行它们：

```
SQL> set autotrace on
SQL> select * from t1
  2   where col2 in (select col2 from t2);

      COL1 C
---------- -
         1 A
         2 B

执行计划
----------------------------------------------------------
Plan hash value: 1713220790

------------------------------------------------------------------------------
| Id  | Operation           | Name | Rows  | Bytes | Cost (%CPU)| Time     |
------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |      |     3 |    21 |    27   (4)| 00:00:01 |
|*  1 |  HASH JOIN SEMI     |      |     3 |    21 |    27   (4)| 00:00:01 |
|   2 |   TABLE ACCESS FULL| T1   |     3 |    15 |    13   (0)| 00:00:01 |
|   3 |   TABLE ACCESS FULL| T2   |     3 |     6 |    13   (0)| 00:00:01 |
------------------------------------------------------------------------------

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

   1 - access("COL2"="COL2")
```

……省略显示部分内容

```
SQL> select * from t1
  2    where col2 = any (select col2 from t2);

      COL1 C
---------- -
         1 A
         2 B

执行计划
----------------------------------------------------------
Plan hash value: 1713220790

------------------------------------------------------------------------------
| Id  | Operation           | Name | Rows  | Bytes | Cost (%CPU)| Time     |
------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |      |     3 |    21 |    27   (4)| 00:00:01 |
|*  1 |  HASH JOIN SEMI     |      |     3 |    21 |    27   (4)| 00:00:01 |
|   2 |   TABLE ACCESS FULL| T1   |     3 |    15 |    13   (0)| 00:00:01 |
|   3 |   TABLE ACCESS FULL| T2   |     3 |     6 |    13   (0)| 00:00:01 |
------------------------------------------------------------------------------

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

   1 - access("COL2"="COL2")
```

……省略显示部分内容

```
SQL> select * from t1
  2    where exists (select 1 from t2 where col2 = t1.col2);

      COL1 C
---------- -
         1 A
         2 B

执行计划
```

```
-----------------------------------------------------------
Plan hash value: 1713220790

-------------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     3 |    21 |    27   (4)| 00:00:01 |
|*  1 |  HASH JOIN SEMI    |      |     3 |    21 |    27   (4)| 00:00:01 |
|   2 |   TABLE ACCESS FULL| T1   |     3 |    15 |    13   (0)| 00:00:01 |
|   3 |   TABLE ACCESS FULL| T2   |     3 |     6 |    13   (0)| 00:00:01 |
-------------------------------------------------------------------------------

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

   1 - access("COL2"="T1"."COL2")
```

……省略显示部分内容

注意，上述三个范例 SQL 的执行结果是一样的，而且它们的执行计划的显示内容中均有关键字“HASH JOIN SEMI”，其中的关键字“SEMI”就说明 Oracle 在执行这三个范例 SQL 时确实是在用半连接，即 Oracle 实际上是将它们转换成了如下的等价半连接形式：

```
select t1.* from t1,t2
 where t1.col2 semi= t2.col2
```

1.2.4.5　星型连接

星型连接（Star Join）通常用于数据仓库类型的应用，它是一种单个事实表（Fact Table）和多个维度表（Dimension Table）之间的连接。从严格意义上来说，星型连接既不是一种额外的连接类型，也不是一种额外的连接方法，只是它有其自身很明显的、有别于其他连接类型的特征，所以这里我们把它单独拿出来说明。

星型连接的各维度表之间没有直接的关联条件，其事实表和各维度表之间是基于事实表的外键列和对应维度表的主键列之间的连接，并且通常在事实表的外键列上还会存在对应的位图索引。

我们来看关于星型连接的一个示意图，如图 1-3 所示。

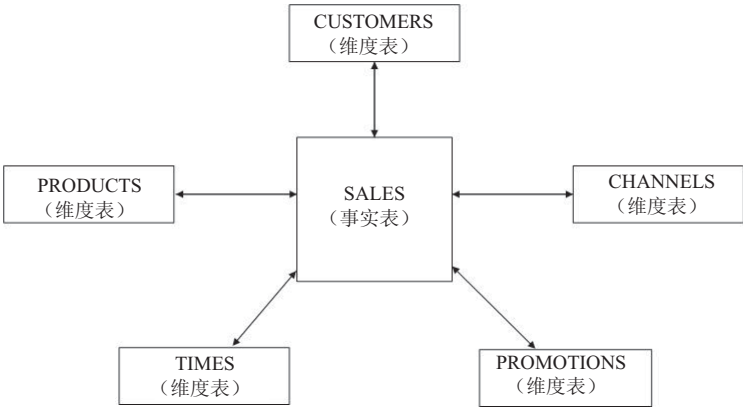

图 1-3　Oracle 数据库中的星型连接原型

图 1-3 所示的就是一个典型的星型连接原型，其中表 SALES 和表 CUSTOMERS、PRODUCTS、TIMES、PROMOTIONS、CHANNELS 之间通过外键列和主键列关联，表 SALES 是事实表（它的数据量可能会非常大），剩下的表都是维度表（它们的数据量和 SALES 相比会小很多），而且它们之间没有直接的关联关系。

为什么称之为星型连接呢？因为事实表和各维度表之间的表连接看起来就像是一颗五角星，在图 1-3 中加

入连接线，就可以形成如图 1-4 所示这样的五角星。

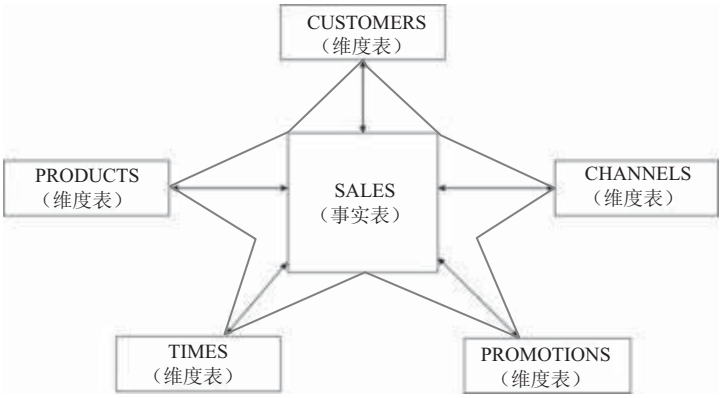

图 1-4　Oracle 数据库中的星型连接

关于星型连接和其所对应的星型转换实例，我们会在第 4 章的“4.4 星型转换”中详细说明，这里不再赘述。

1.3　优化器模式对 CBO 计算成本带来巨大影响的实例

我们在“1.2.1 优化器的模式”中曾经详细介绍过 Oracle 数据库中各种优化器模式的含义，现在来看一个由于优化器模式的设置不当而导致 CBO 认为全表扫描一个 700 多万条数据的大表的成本值仅为 2，进而直接导致 CBO 选错执行计划的实例。

2010 年 12 月 6 日，某公司 Call Center 系统坐席登录非常慢，严重影响了该公司的日常工作。据了解，此前坐席登录慢的问题曾多次不间断出现，但未找到根本原因，因此需要尽快做出处理，恢复 Call Center 系统坐席的正常登录，并找出造成坐席登录不间断慢的根本原因，以从根本上解决该问题。

当我们赶到现场后，该问题仍在持续，没有丝毫好转的迹象，不过这对于排错人员而言是好事情，要的就是这样。我们做的第一件事情就是登录上上述 Call Center 系统后台数据库所在的数据库服务器，并使用 TOPAS 命令查看当时的系统资源状况，TOPAS 的输出结果如下所示：

```
    Topas Monitor for host:    XXXXXXXX          EVENTS/QUEUES    FILE/TTY
Mon Dec  6 10:47:51 2010   Interval:  2       Cswitch   25489  Readch   179.8M
                                           Syscall  213.4K  Writech   37.9M
Kernel    8.2   |###                       | Reads    31196  Rawin        0
User     24.1   |#######                   | Writes     309  Ttyout     750
Wait      1.1   |#                         | Forks        6  Igets        0
Idle     66.5   |####################      | Execs        5  Namei      545
Physc =  6.90                 %Entc=  34.5   Runqueue    9.5  Dirblk       0
                                           Waitqueue   1.0
Network  KBPS   I-Pack  O-Pack   KB-In  KB-Out
en2     32.5K   11.9K   22.7K 8827.1    23.9K PAGING          MEMORY
en3     197.8   173.0   142.0    79.7   118.1  Faults    3697  Real,MB   81920
lo0      64.3     4.0     4.0    32.2    32.2  Steals   10803  % Comp    32.2
```

```
                                          PgspIn       0  % Noncomp  67.7
Disk    Busy%     KBPS     TPS KB-Read KB-Writ PgspOut      0  % Client   67.7
hdisk23  98.0    37.7K  338.5    37.7K    0.0   PageIn    9648
hdisk12  64.0     9.2K  332.5     9.2K    0.0   PageOut      0  PAGING SPACE
hdisk21  67.0     9.1K  295.5     9.1K    0.0   Sios      9728  Size,MB   98304
hdisk19  82.5     8.9K  303.0     8.9K    0.0                   % Used     0.0
hdisk17  85.5     8.9K  309.5     8.9K    0.0   NFS (calls/sec)  % Free    100.0
hdisk20  52.0     8.8K  317.0     8.8K    0.0   ServerV2      0
hdisk18  74.0     8.8K  332.5     8.8K    0.0   ClientV2      0   Press:
hdisk13  64.5     8.7K  299.0     8.7K    0.0   ServerV3      0   "h" for help

    Name          PID  CPU%  PgSp Owner
    oracle     3092682   4.0   4.7 oracle
oracle     4345856   3.4   4.7 oracle
oracle     2732154   2.8   4.7 oracle
……省略显示部分内容
    oracle     3096808   0.3  14.7 oracle
```

从上述 TOPAS 的显示结果可以看出，现在的状况是磁盘较繁忙，en2 的网卡流量也较大，但 USER CPU 的使用率并不算高，整个系统 CPU 的 IDLE 值也接近 67%。

从整个系统的资源使用状况来看，并无明显异常，但为什么 Call Center 系统坐席登录会非常慢呢？

一般来说，系统慢分为两种情况：一种是执行什么操作都慢（这通常说明该系统的资源使用状况出了问题，或者所有的人都在等待什么东西）；另外一种是执行某种操作慢，但执行其他操作的速度还可以（这通常说明执行慢的那种操作所对应的 SQL 出了问题）。

现在 Call Center 系统只是在坐席登录的时候特别慢，但一旦登录进去后，执行各种操作的速度还可以，所以这是属于上述第二种情况，也就是说我们只需要把所有坐席登录时都需要执行且执行速度非常慢的 SQL 抓出来，并对其做调整，就应该可以解决上述问题了。

于是我们采集了 Call Center 系统后台数据库在 2010 年 12 月 6 日上午 9 点至 10 点的 AWR 报告，该 AWR 报告中“SQL ordered by Elapsed Time”的内容如下所示：

Elapsed Time (s)	Executions	Elap per Exec (s)	% Total DB Time	SQL Id	SQL Text
20,666	**216**	**95.67**	**82.41**	**49jn33ac20q8s**	**SELECT T18.CONFLICT_ID, ...**
1,428	4,491	0.32	5.69	21s56fnnm38ts	SELECT T1.CONFLICT_ID, ...
					……省略显示部分内容
51	18	2.86	0.21	dgtsn81r8uhv1	SELECT T48.CONFLICT_ID, ...
47	192	0.25	0.19	g4yu3f28adt2q	BEGIN INSERT INTO SIEBEL....

可以看出，SQL ID 为 49jn33ac20q8s 的 SQL 是在上述采样时间段执行时间最长的 SQL，它在采样时间段一共执行了 216 次，单次平均执行时间达到了 95.67 s，其执行时间占了整个 DB Time 的 82.41%。

通过咨询相关的运维人员，得知该 SQL 正是每次坐席登录都会调用的 SQL，因此我们有理由相信，正是因为该 SQL 的单次执行时间过长（单次平均执行时间为 95.67 s）而导致了 Call Center 系统坐席登录缓慢。事

实上，如果每次登录的平均等待时间都需要 95.67 s 的话，确实会让人觉得非常非常慢。

既然已经抓到了待调整的目标 SQL，我们现在就来看一看该 SQL。这是一个 18 张表做表连接的 SQL，格式化后的 SQL 文本如下所示：

```
SELECT T18.CONFLICT_ID,
       T18.LAST_UPD,
       T18.CREATED,
       ……省略显示部分内容
       T17.ROW_ID,
       T11.EMAIL_BODY
  FROM SIEBEL.S_ACT_EMP    T1,
       SIEBEL.S_EVT_MKTG   T2,
       SIEBEL.S_CONTACT    T3,
       SIEBEL.S_CONTACT    T4,
       SIEBEL.S_EXP_RPT    T5,
       SIEBEL.S_PARTY      T6,
       SIEBEL.S_PROJ       T7,
       SIEBEL.S_PROD_DEFECT T8,
       SIEBEL.S_PROJITEM   T9,
       SIEBEL.S_EVT_ACT_SS  T10,
       SIEBEL.S_EVT_MAIL   T11,
       SIEBEL.S_USER       T12,
       SIEBEL.S_EVT_CAL    T13,
       SIEBEL.S_ORG_EXT    T14,
       SIEBEL.S_TMSHT_LINE  T15,
       SIEBEL.S_OPTY       T16,
       SIEBEL.S_PARTY      T17,
       SIEBEL.S_EVT_ACT    T18
 WHERE T18.TARGET_PER_ID = T4.PAR_ROW_ID(+)
   AND T18.PR_EXP_RPT_ID = T5.ROW_ID(+)
   AND T18.SRA_DEFECT_ID = T8.ROW_ID(+)
   AND T18.TARGET_OU_ID = T14.PAR_ROW_ID(+)
   AND T18.OPTY_ID = T16.ROW_ID(+)
   AND T18.PROJ_ID = T7.ROW_ID(+)
   AND T18.PROJ_ITEM_ID = T9.ROW_ID(+)
   AND T18.PR_TMSHT_LINE_ID = T15.ROW_ID(+)
   AND T18.ROW_ID = T2.PAR_ROW_ID(+)
   AND T18.ROW_ID = T11.PAR_ROW_ID(+)
   AND T18.ROW_ID = T13.PAR_ROW_ID(+)
   AND T18.ROW_ID = T10.PAR_ROW_ID(+)
   AND T1.EMP_ID = :1
   AND T18.ROW_ID = T1.ACTIVITY_ID
   AND T1.EMP_ID = T6.ROW_ID
   AND T1.EMP_ID = T12.PAR_ROW_ID(+)
   AND T18.TARGET_PER_ID = T17.ROW_ID(+)
   AND T18.TARGET_PER_ID = T3.PAR_ROW_ID(+)
```

```
    AND ((T18.APPT_REPT_REPL_CD IS NULL) AND
       (T1.ACT_TEMPLATE_FLG != 'Y' AND T1.ACT_TEMPLATE_FLG != 'P' OR
       T1.ACT_TEMPLATE_FLG IS NULL))
```

接下来我们对上述 SQL 做一个相同采样时间段的 AWR SQL Report，从这份 AWR SQL Report 中可以看出如下内容。

（1）该 SQL 有四份执行计划，坐席登录非常慢时 CBO 选择的是 Plan Hash Value 为 4128147724 所对应的执行计划。

#	Plan Hash Value	Total Elapsed Time(ms)	Executions	1st Capture Snap ID	Last Capture Snap ID
1	**4128147724**	**20,665,673**	**216**	**15399**	**15399**
2	3875831895	0	0	15399	15399
3	3512509353	0	0	15399	15399
4	2583579266	0	0	15399	15399

（2）走 Plan Hash Value 为 4128147724 所对应的执行计划时，平均执行时间为 95.67s，逻辑读高达 7,178,737 次，但每次执行平均只返回 3.17 行记录。

Stat Name	Statement Total	Per Execution	% Snap Total
Elapsed Time (ms)	**20,665,673**	**95,674.41**	**82.41**
CPU Time (ms)	13,100,244	60,649.28	85.06
Executions	216		
Buffer Gets	**1,550,607,319**	**7,178,737.59**	**93.74**
Disk Reads	16,807,055	77,810.44	98.81
Parse Calls	108	0.50	0.04
Rows	**684**	**3.17**	
User I/O Wait Time (ms)	3,492,891		
Cluster Wait Time (ms)	4,188,285		
Application Wait Time (ms)	0		
Concurrency Wait Time (ms)	13,761		
Invalidations	0		
Version Count	5		
Sharable Mem(KB)	435		

（3）上述 Plan Hash Value 为 4128147724 所对应的执行计划如下所示：

```
Plan 1(PHV: 4128147724)
-----------------------
Execution Plan
----------------------------------------------------------------------------------------------------------------------
| Id  | Operation                                | Name             | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                         |                  |       |       |    42 (100)|          |
|   1 |  NESTED LOOPS OUTER                      |                  |    10 | 25020 |    42   (0)| 00:00:01 |
|   2 |   NESTED LOOPS OUTER                     |                  |    10 | 24880 |    41   (0)| 00:00:01 |
|   3 |    NESTED LOOPS OUTER                    |                  |    10 | 24620 |    38   (0)| 00:00:01 |
|   4 |     NESTED LOOPS OUTER                   |                  |    10 | 24500 |    37   (0)| 00:00:01 |
|   5 |      NESTED LOOPS OUTER                  |                  |    10 | 23770 |    31   (0)| 00:00:01 |
|   6 |       NESTED LOOPS OUTER                 |                  |    10 | 22830 |    25   (0)| 00:00:01 |
|   7 |        NESTED LOOPS OUTER                |                  |    10 | 22080 |    24   (0)| 00:00:01 |
|   8 |         NESTED LOOPS OUTER               |                  |    10 | 21390 |    18   (0)| 00:00:01 |
|   9 |          NESTED LOOPS                    |                  |    10 | 21230 |    15   (0)| 00:00:01 |
|  10 |           NESTED LOOPS OUTER             |                  |    10 | 20880 |    12   (0)| 00:00:01 |
|  11 |            NESTED LOOPS OUTER            |                  |    10 | 16630 |     9   (0)| 00:00:01 |
```

```
|  12 |              NESTED LOOPS OUTER              |                |    10 | 15990 |     8   (0)| 00:00:01 |
|  13 |               NESTED LOOPS OUTER             |                |    10 | 11990 |     7   (0)| 00:00:01 |
|  14 |                NESTED LOOPS OUTER            |                |    10 | 10370 |     6   (0)| 00:00:01 |
|  15 |                 NESTED LOOPS OUTER           |                |    10 |  9690 |     5   (0)| 00:00:01 |
|  16 |                  NESTED LOOPS OUTER          |                |    10 |  9050 |     4   (0)| 00:00:01 |
|  17 |                   NESTED LOOPS               |                |    10 |  6290 |     3   (0)| 00:00:01 |
|  18 |                    INDEX UNIQUE SCAN         | S_PARTY_P1     |     1 |    12 |     1   (0)| 00:00:01 |
|  19 |                    TABLE ACCESS FULL         | S_EVT_ACT      |    10 |  6170 |     2   (0)| 00:00:01 |
|  20 |                   TABLE ACCESS BY INDEX ROWID| S_OPTY         |     1 |   276 |     1   (0)| 00:00:01 |
|  21 |                    INDEX UNIQUE SCAN         | S_OPTY_P1      |     1 |       |     1   (0)| 00:00:01 |
|  22 |                  TABLE ACCESS BY INDEX ROWID | S_TMSHT_LINE   |     1 |    64 |     1   (0)| 00:00:01 |
|  23 |                   INDEX UNIQUE SCAN          | S_TMSHT_LINE_P1|     1 |       |     1   (0)| 00:00:01 |
|  24 |                 TABLE ACCESS BY INDEX ROWID  | S_PROJITEM     |     1 |    68 |     1   (0)| 00:00:01 |
|  25 |                  INDEX UNIQUE SCAN           | S_PROJITEM_P1  |     1 |       |     1   (0)| 00:00:01 |
|  26 |                TABLE ACCESS BY INDEX ROWID   | S_PROD_DEFECT  |     1 |   162 |     1   (0)| 00:00:01 |
|  27 |                 INDEX UNIQUE SCAN            | S_PROD_DEFECT_P1|    1 |       |     1   (0)| 00:00:01 |
|  28 |               TABLE ACCESS BY INDEX ROWID    | S_PROJ         |     1 |   400 |     1   (0)| 00:00:01 |
|  29 |                INDEX UNIQUE SCAN             | S_PROJ_P1      |     1 |       |     1   (0)| 00:00:01 |
|  30 |              TABLE ACCESS BY INDEX ROWID     | S_EXP_RPT      |     1 |    64 |     1   (0)| 00:00:01 |
|  31 |               INDEX UNIQUE SCAN             | S_EXP_RPT_P1   |     1 |       |     1   (0)| 00:00:01 |
|  32 |             TABLE ACCESS BY INDEX ROWID      | S_EVT_ACT_SS   |     1 |   425 |     1   (0)| 00:00:01 |
|  33 |              INDEX RANGE SCAN                | S_EVT_ACT_SS_U1|     1 |       |     1   (0)| 00:00:01 |
|  34 |            TABLE ACCESS BY INDEX ROWID       | S_ACT_EMP      |     1 |    35 |     1   (0)| 00:00:01 |
|  35 |             INDEX RANGE SCAN                 | S_ACT_EMP_F1   |     1 |       |     1   (0)| 00:00:01 |
|  36 |           TABLE ACCESS BY INDEX ROWID        | S_USER         |     1 |    16 |     1   (0)| 00:00:01 |
|  37 |            INDEX UNIQUE SCAN                 | S_USER_U2      |     1 |       |     1   (0)| 00:00:01 |
|  38 |          TABLE ACCESS BY INDEX ROWID         | S_EVT_MAIL     |     1 |    69 |     1   (0)| 00:00:01 |
|  39 |           INDEX RANGE SCAN                   | S_EVT_MAIL_U1  |     1 |       |     1   (0)| 00:00:01 |
|  40 |         TABLE ACCESS BY INDEX ROWID          | S_ORG_EXT      |     1 |    75 |     1   (0)| 00:00:01 |
|  41 |          INDEX UNIQUE SCAN                   | S_ORG_EXT_U3   |     1 |       |     1   (0)| 00:00:01 |
|  42 |        TABLE ACCESS BY INDEX ROWID           | S_EVT_MKTG     |     1 |    94 |     1   (0)| 00:00:01 |
|  43 |         INDEX RANGE SCAN                     | S_EVT_MKTG_U1  |     1 |       |     1   (0)| 00:00:01 |
|  44 |       TABLE ACCESS BY INDEX ROWID            | S_EVT_CAL      |     1 |    73 |     1   (0)| 00:00:01 |
|  45 |        INDEX RANGE SCAN                      | S_EVT_CAL_U1   |     1 |       |     1   (0)| 00:00:01 |
|  46 |      INDEX UNIQUE SCAN                       | S_PARTY_P1     |     1 |    12 |     1   (0)| 00:00:01 |
|  47 |     TABLE ACCESS BY INDEX ROWID              | S_CONTACT      |     1 |    26 |     1   (0)| 00:00:01 |
|  48 |      INDEX UNIQUE SCAN                       | S_CONTACT_U2   |     1 |       |     1   (0)| 00:00:01 |
|  49 |    TABLE ACCESS BY INDEX ROWID               | S_CONTACT      |     1 |    14 |     1   (0)| 00:00:01 |
|  50 |     INDEX UNIQUE SCAN                        | S_CONTACT_U2   |     1 |       |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------------
```

可以看到上述执行计划总的成本值为 42，Id = 19 的执行步骤是对表 S_EVT_ACT 的全表扫描。

（4）该 SQL 的其他三份执行计划如下所示：

```
Plan 2(PHV: 3875831895)
------------------------
Execution Plan
-----------------------------------------------------------------------------------------------------------
| Id  | Operation                                    | Name             | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                             |                  |       |       |    41 (100)|          |
|   1 |  NESTED LOOPS OUTER                          |                  |    10 | 24960 |    41   (0)| 00:00:01 |
|   2 |   NESTED LOOPS OUTER                         |                  |    10 | 24830 |    40   (0)| 00:00:01 |
|   3 |    NESTED LOOPS OUTER                        |                  |    10 | 24590 |    37   (0)| 00:00:01 |
|   4 |     NESTED LOOPS OUTER                       |                  |    10 | 24480 |    36   (0)| 00:00:01 |
|   5 |      NESTED LOOPS OUTER                      |                  |    10 | 23750 |    30   (0)| 00:00:01 |
|   6 |       NESTED LOOPS OUTER                     |                  |    10 | 22820 |    24   (0)| 00:00:01 |
|   7 |        NESTED LOOPS OUTER                    |                  |    10 | 22080 |    23   (0)| 00:00:01 |
|   8 |         NESTED LOOPS OUTER                   |                  |    10 | 21410 |    17   (0)| 00:00:01 |
|   9 |          NESTED LOOPS OUTER                  |                  |    10 | 17160 |    14   (0)| 00:00:01 |
|  10 |           NESTED LOOPS OUTER                 |                  |    10 | 16520 |    13   (0)| 00:00:01 |
|  11 |            NESTED LOOPS OUTER                |                  |    10 | 12520 |    12   (0)| 00:00:01 |
|  12 |             NESTED LOOPS OUTER               |                  |    10 | 10900 |    11   (0)| 00:00:01 |
|  13 |              NESTED LOOPS OUTER              |                  |    10 | 10220 |    10   (0)| 00:00:01 |
|  14 |               NESTED LOOPS OUTER             |                  |    10 |  9580 |     9   (0)| 00:00:01 |
|  15 |                NESTED LOOPS                  |                  |    10 |  6820 |     8   (0)| 00:00:01 |
|  16 |                 NESTED LOOPS OUTER           |                  |    10 |   620 |     5   (0)| 00:00:01 |
|  17 |                  NESTED LOOPS                |                  |    10 |   460 |     2   (0)| 00:00:01 |
|  18 |                   INDEX UNIQUE SCAN          | S_PARTY_P1       |     1 |    11 |     1   (0)| 00:00:01 |
|  19 |                   TABLE ACCESS BY INDEX ROWID| S_ACT_EMP        |    10 |   350 |     1   (0)| 00:00:01 |
|  20 |                    INDEX RANGE SCAN          | S_ACT_EMP_M6     |     1 |       |     1   (0)| 00:00:01 |
|  21 |                  TABLE ACCESS BY INDEX ROWID | S_USER           |     1 |    16 |     1   (0)| 00:00:01 |
|  22 |                   INDEX UNIQUE SCAN          | S_USER_U2        |     1 |       |     1   (0)| 00:00:01 |
|  23 |                 TABLE ACCESS BY INDEX ROWID  | S_EVT_ACT        |     1 |   620 |     1   (0)| 00:00:01 |
|  24 |                  INDEX UNIQUE SCAN           | S_EVT_ACT_P1     |     1 |       |     1   (0)| 00:00:01 |
|  25 |                TABLE ACCESS BY INDEX ROWID   | S_OPTY           |     1 |   276 |     1   (0)| 00:00:01 |
……省略显示部分内容
|  50 |   TABLE ACCESS BY INDEX ROWID                | S_CONTACT        |     1 |    13 |     1   (0)| 00:00:01 |
|  51 |    INDEX UNIQUE SCAN                         | S_CONTACT_U2     |     1 |       |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------------
```

```
Plan 3(PHV: 3512509353)
-----------------------
Execution Plan
----------------------------------------------------------------------------------------------------------
| Id  | Operation                                | Name             | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                         |                  |       |       |    43 (100)|          |
|   1 |  NESTED LOOPS OUTER                      |                  |    10 | 25020 |    43   (0)| 00:00:01 |
|   2 |   NESTED LOOPS OUTER                     |                  |    10 | 24900 |    42   (0)| 00:00:01 |
|   3 |    NESTED LOOPS OUTER                    |                  |    10 | 24170 |    36   (0)| 00:00:01 |
|   4 |     NESTED LOOPS OUTER                   |                  |    11 | 26301 |    35   (0)| 00:00:01 |
|   5 |      NESTED LOOPS OUTER                  |                  |    11 | 26147 |    32   (0)| 00:00:01 |
|   6 |       NESTED LOOPS OUTER                 |                  |    11 | 25113 |    26   (0)| 00:00:01 |
|   7 |        NESTED LOOPS OUTER                |                  |    11 | 24288 |    25   (0)| 00:00:01 |
|   8 |         NESTED LOOPS OUTER               |                  |    12 | 25668 |    18   (0)| 00:00:01 |
|   9 |          NESTED LOOPS OUTER              |                  |    12 | 22356 |    17   (0)| 00:00:01 |
|  10 |           NESTED LOOPS OUTER             |                  |    12 | 21588 |    16   (0)| 00:00:01 |
|  11 |            NESTED LOOPS OUTER            |                  |    12 | 16488 |    13   (0)| 00:00:01 |
|  12 |             NESTED LOOPS OUTER           |                  |    12 | 15672 |    12   (0)| 00:00:01 |
|  13 |              NESTED LOOPS OUTER          |                  |    12 | 13728 |    11   (0)| 00:00:01 |
|  14 |               NESTED LOOPS OUTER         |                  |    12 |  8928 |    10   (0)| 00:00:01 |
|  15 |                NESTED LOOPS              |                  |    12 |  8160 |     9   (0)| 00:00:01 |
|  16 |                 NESTED LOOPS OUTER       |                  |    12 |   768 |     5   (0)| 00:00:01 |
|  17 |                  NESTED LOOPS            |                  |    12 |   564 |     2   (0)| 00:00:01 |
|  18 |                   INDEX UNIQUE SCAN      | S_PARTY_P1       |     1 |    12 |     1   (0)| 00:00:01 |
|  19 |                   TABLE ACCESS BY INDEX ROWID| S_ACT_EMP     |    11 |   385 |     1   (0)| 00:00:01 |
|  20 |                    INDEX RANGE SCAN      | S_ACT_EMP_M6     |     1 |       |     1   (0)| 00:00:01 |
|  21 |                  TABLE ACCESS BY INDEX ROWID | S_USER         |     1 |    17 |     1   (0)| 00:00:01 |
|  22 |                   INDEX UNIQUE SCAN      | S_USER_U2        |     1 |       |     1   (0)| 00:00:01 |
|  23 |                 TABLE ACCESS BY INDEX ROWID | S_EVT_ACT      |     1 |   616 |     1   (0)| 00:00:01 |
|  24 |                  INDEX UNIQUE SCAN       | S_EVT_ACT_P1     |     1 |       |     1   (0)| 00:00:01 |
|  25 |                TABLE ACCESS BY INDEX ROWID | S_EXP_RPT      |     1 |    64 |     1   (0)| 00:00:01 |
……省略显示部分内容
|  50 |     INDEX RANGE SCAN                     | S_EVT_CAL_U1     |     1 |       |     1   (0)| 00:00:01 |
|  51 |   INDEX UNIQUE SCAN                      | S_PARTY_P1       |     1 |    12 |     1   (0)| 00:00:01 |
----------------------------------------------------------------------------------------------------------

Plan 4(PHV: 2583579266)
-----------------------
Execution Plan
----------------------------------------------------------------------------------------------------------
| Id  | Operation                                | Name             | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                         |                  |       |       |    44 (100)|          |
|   1 |  NESTED LOOPS OUTER                      |                  |    11 | 27467 |    44   (0)| 00:00:01 |
|   2 |   NESTED LOOPS OUTER                     |                  |    11 | 27324 |    43   (0)| 00:00:01 |
|   3 |    NESTED LOOPS OUTER                    |                  |    11 | 27060 |    40   (0)| 00:00:01 |
|   4 |     NESTED LOOPS OUTER                   |                  |    11 | 26939 |    39   (0)| 00:00:01 |
|   5 |      NESTED LOOPS OUTER                  |                  |    11 | 25927 |    32   (0)| 00:00:01 |
|   6 |       NESTED LOOPS OUTER                 |                  |    11 | 25124 |    26   (0)| 00:00:01 |
|   7 |        NESTED LOOPS OUTER                |                  |    11 | 24310 |    25   (0)| 00:00:01 |
|   8 |         NESTED LOOPS OUTER               |                  |    11 | 23562 |    18   (0)| 00:00:01 |
|   9 |          NESTED LOOPS OUTER              |                  |    11 | 18887 |    15   (0)| 00:00:01 |
|  10 |           NESTED LOOPS OUTER             |                  |    11 | 18183 |    14   (0)| 00:00:01 |
|  11 |            NESTED LOOPS OUTER            |                  |    11 | 13783 |    13   (0)| 00:00:01 |
|  12 |             NESTED LOOPS OUTER           |                  |    11 | 12001 |    12   (0)| 00:00:01 |
|  13 |              NESTED LOOPS OUTER          |                  |    11 | 11253 |    11   (0)| 00:00:01 |
|  14 |               NESTED LOOPS OUTER         |                  |    11 | 10549 |    10   (0)| 00:00:01 |
|  15 |                NESTED LOOPS              |                  |    11 |  7513 |     9   (0)| 00:00:01 |
|  16 |                 NESTED LOOPS OUTER       |                  |    11 |   682 |     5   (0)| 00:00:01 |
|  17 |                  NESTED LOOPS            |                  |    11 |   506 |     2   (0)| 00:00:01 |
|  18 |                   INDEX UNIQUE SCAN      | S_PARTY_P1       |     1 |    11 |     1   (0)| 00:00:01 |
|  19 |                   TABLE ACCESS BY INDEX ROWID| S_ACT_EMP     |    11 |   385 |     1   (0)| 00:00:01 |
|  20 |                    INDEX RANGE SCAN      | S_ACT_EMP_M6     |     1 |       |     1   (0)| 00:00:01 |
|  21 |                  TABLE ACCESS BY INDEX ROWID | S_USER         |     1 |    16 |     1   (0)| 00:00:01 |
|  22 |                   INDEX UNIQUE SCAN      | S_USER_U2        |     1 |       |     1   (0)| 00:00:01 |
|  23 |                 TABLE ACCESS BY INDEX ROWID | S_EVT_ACT      |     1 |   621 |     1   (0)| 00:00:01 |
|  24 |                  INDEX UNIQUE SCAN       | S_EVT_ACT_P1     |     1 |       |     1   (0)| 00:00:01 |
|  25 |                TABLE ACCESS BY INDEX ROWID | S_OPTY         |     1 |   276 |     1   (0)| 00:00:01 |
……省略显示部分内容
|  50 |   TABLE ACCESS BY INDEX ROWID            | S_CONTACT        |     1 |    13 |     1   (0)| 00:00:01 |
|  51 |    INDEX UNIQUE SCAN                     | S_CONTACT_U2     |     1 |       |     1   (0)| 00:00:01 |
----------------------------------------------------------------------------------------------------------
```

从上述显示内容中我们可以看出，其他三份执行计划在访问表 S_EVT_ACT 时均走的是对索引 S_EVT_ACT_P1 的索引唯一性扫描，并且这三份执行计划对应的总成本值分别为 41、43 和 44。

看完了上述 AWR SQL Report，意味着初始的准备工作都已经做完了，我们现在来开始分析这个目标 SQL。

注意到坐席登录非常慢时 CBO 选择的是 Plan Hash Value 为 4128147724 所对应的执行计划，在该执行计划中出现了对表 S_EVT_ACT 的全表扫描。我们来看一下表 S_EVT_ACT 的统计信息：

```
SQL> select table_name,num_rows,blocks,to_char(last_analyzed,'yyyymmdd hh24:mi:ss') from
dba_tables where table_name = 'S_EVT_ACT';

TABLE_NAME    NUM_ROWS     BLOCKS     TO_CHAR(LAST_ANALYZED,'YYYYMMD
----------  ----------  ----------  ------------------------------
S_EVT_ACT      6991640     730185     20101203 00:07:41
```

从上述查询结果中可以看到，表 S_EVT_ACT 的统计信息上一次的收集时间为 2010 年 12 月 3 日 0 点 7 分，这说明该表的统计信息并不算陈旧。上述统计信息中表 S_EVT_ACT 的记录数为 6,991,640 条（接近 700 万），其 BLOCK 数为 730,185。

上述统计信息是否准确呢？我们来验证一下。查询表 S_EVT_ACT 中的真实记录数：

```
SQL> select count(*) from SIEBEL.S_EVT_ACT;

  COUNT(*)
----------
   7349375
```

从上述查询结果中我们可以看到，表 S_EVT_ACT 的实际记录数为 7,349,375，和统计信息记录的值相差不大，这说明表 S_EVT_ACT 的统计信息是较为准确的。这同时也意味着 CBO 此时选择的执行计划需要对一个数据量为 700 多万条的大表执行全表扫描操作，那么当然其平均执行时间会达到 90 多秒，这就是造成 Call Center 系统坐席登录缓慢的原因。

注意，AWR SQL Report 中记录的目标 SQL 的另外三份执行计划在访问表 S_EVT_ACT 时均走的是对索引 S_EVT_ACT_P1 的索引唯一性扫描，而并没有走对表 S_EVT_ACT 的全表扫描。那么为什么此时 CBO 选择的执行计划并不像其他三份执行计划那样走对索引 S_EVT_ACT_P1 的索引唯一性扫描呢？是不是因为索引 S_EVT_ACT_P1 现在已经不存在了？

我们来检查一下表 S_EVT_ACT 上的索引情况：

```
SQL> select index_name,column_name,column_position from dba_ind_columns where
table_name='S_EVT_ACT' order by 1,3;

INDEX_NAME           COLUMN_NAME      COLUMN_POSITION
-------------------  ---------------  ---------------

……省略显示部分内容
S_EVT_ACT_P1         ROW_ID           1
……省略显示部分内容
```

从上述查询结果中可以看出，索引 S_EVT_ACT_P1 依然存在于表 S_EVT_ACT 中，并且它是列 ROW_ID 上的一个单键值 B 树索引。

既然索引 S_EVT_ACT_P1 依然在表 S_EVT_ACT 中，那么这里 CBO 为什么没有用该索引？通常这说明此时 CBO 认为全表扫描表 S_EVT_ACT 的成本会小于走索引 S_EVT_ACT_P1 的成本。

接下来，为了方便分析，我们暂时将其他无关紧要的表从原目标 SQL 中剔除，即形成了如下形式的简化版本：

```
SELECT T18.CONFLICT_ID,
       T18.LAST_UPD,
       T18.CREATED,
```

```
      ……省略显示部分内容
      T17.ROW_ID,
      T11.EMAIL_BODY
  FROM SIEBEL.S_ACT_EMP    T1,
      ……省略显示部分内容
      SIEBEL.S_EVT_ACT    T18
 WHERE ……省略显示部分内容
   AND T1.EMP_ID = :1
   AND T18.ROW_ID = T1.ACTIVITY_ID
   ……省略显示部分内容
   AND ((T18.APPT_REPT_REPL_CD IS NULL) AND
      (T1.ACT_TEMPLATE_FLG != 'Y' AND T1.ACT_TEMPLATE_FLG != 'P' OR
      T1.ACT_TEMPLATE_FLG IS NULL))
```

从上述简化版本我们可以看到，表 S_EVT_ACT 和 S_ACT_EMP 做表连接的连接条件为“T18.ROW_ID = T1.ACTIVITY_ID”，这里完全是可以走嵌套循环连接的，并且由于在表 S_ACT_EMP 的列 EMP_ID 上存在单键值 B 树索引，在表 S_EVT_ACT 的列 ROWID 上也存在单键值 B 树索引（即索引 S_EVT_ACT_P1），所以如果此时走嵌套循环连接，则该嵌套循环连接的驱动表和被驱动表就都可以通过索引来访问数据。显然，这里嵌套循环连接的驱动表应选择表 S_ACT_EMP，被驱动表表应选择 S_EVT_ACT，这样的执行计划才是合理的。

从对 DBA_IND_COLUMNS 的查询结果我们已经知道，表 S_EVT_ACT 中和表 S_ACT_EMP 做表连接的连接列 ROW_ID 上已经有了索引 S_EVT_ACT_P1，但是 CBO 却没有用该索引，也没有选择让表 S_EVT_ACT 和 S_ACT_EMP 直接做表连接，而是选择了对表 S_EVT_ACT 的全表扫描，很明显这里是 CBO 选错了执行计划。

综上所述，CBO 本可以选择表 S_EVT_ACT 和 S_ACT_EMP 间的嵌套循环连接，并且在访问表 S_EVT_ACT 时走索引 S_EVT_ACT_P1，但是 CBO 这里却选择了对表 S_EVT_ACT 的全表扫描，又由于这张表的实际记录数达到了 730 多万条，所以就造成了上述 SQL 不佳的执行效率，进而影响到了 Call Center 系统坐席的登录速度。

分析清楚了初步的原因，现在的问题是：应该如何来解决上述问题？

我们在介绍这个案例的一开头就提到了这样一句话——“据了解，此前坐席登录慢的问题曾多次不间断出现，但未找到根本原因”，注意这句话里的关键字“多次不间断出现”，这说明 Oracle 在执行上述 SQL 时并不是什么时候都慢，而是有时快、有时慢，正是因为上述 SQL 在执行时的时快时慢才会导致坐席登录慢的问题会多次不间断出现。

我们现在已经知道，坐席登录慢是因为 CBO 在执行上述 18 个表关联的 SQL 时选择了对表 S_EVT_ACT 的全表扫描，那么当坐席登录速度正常时显然是因为 CBO 此时并没有对表 S_EVT_ACT 做全表扫描操作。再考虑到之前 AWR SQL Report 中记录的该 SQL 的另外三份执行计划在访问表 S_EVT_ACT 时均走的是对索引 S_EVT_ACT_P1 的索引唯一性扫描（并没有走对表 S_EVT_ACT 的全表扫描），那么这里上述 SQL 在执行时会时快时慢的原因就已经昭然若揭了。

很显然，坐席登录慢是因为 CBO 在执行上述 18 个表关联的 SQL 时选择了该 SQL 的四份执行计划中那个对表 S_EVT_ACT 做全表扫描所对应的执行计划，而坐席登录速度正常时 CBO 选择的则是另外三份执行计划中的任意一份（这三份执行计划在访问表 S_EVT_ACT 时均走的是对索引 S_EVT_ACT_P1 的索引唯一性

扫描）。

“坐席登录慢的问题曾多次不间断出现”意味着 CBO 有时能选对执行计划，有时又不能，这又是为什么？

我们在本章的前面介绍 CBO 时已经提到过：CBO 选择执行计划所用的判断原则为成本，即 CBO 会从目标 SQL 诸多可能的执行路径中选择一条成本值最小的执行路径来作为其执行计划，而各条执行路径的成本值则是根据目标 SQL 语句所涉及的表、索引、列等相关对象的统计信息计算出来的。

而上述 SQL 的这四份执行计划在之前的 AWR SQL Report 中所对应的总成本值分别为 42、41、43 和 44，它们非常接近！这意味着上述 SQL 中所涉及的那 18 个表以及相关对象的统计信息只要稍微发生了一点变化，就有可能使得 CBO 改变对于这四份执行计划的选择。

从 Oracle 10g 开始，Oracle 引入了自动统计信息收集作业，能够每天自动收集统计信息，所以这意味着在 Oracle 10g 及其之后的 Oracle 数据库版本中，只要不禁掉自动统计信息收集作业，则相关对象的统计信息每天都是可能发生变化的。上述 SQL 所在的 Oracle 数据库版本为 10.2.0.4，而且其自动统计信息收集作业并没有被禁掉，所以上述 SQL 中所涉及的那 18 个表以及相关对象的统计信息确实有可能每天发生变化。一旦统计信息发生了变化，CBO 就可能改变对于上述这四份执行计划的选择，因为它们的成本值实在是太接近了。

上述 SQL 的四份执行计划所对应成本值的过于接近，就是导致坐席登录慢的问题多次不间断出现的原因。

既然已经知道了上述 SQL 在执行时为什么会时快时慢，那么临时解决上述问题的方法显然就是重新对表 S_EVT_ACT 收集一下统计信息。因为一旦对表 S_EVT_ACT 重新收集了统计信息，CBO 就极有可能会改变对于执行计划的选择，而且当前这种情形下只要 CBO 改变了对于该 SQL 执行计划的选择，上述问题就会临时得到解决，因为该 SQL 的四份执行计划中除了当前 CBO 选择的这份“不好的”执行计划外，其他三份执行计划在访问表 S_EVT_ACT 时均走的是对索引 S_EVT_ACT_P1 的索引唯一性扫描。

我们使用如下命令对表 S_EVT_ACT 重新收集了统计信息：

```
SQL> exec dbms_stats.gather_table_stats(ownname => 'SIEBEL', tabname => 'S_EVT_ACT', cascade
=> true, no_invalidate => false, degree => 4);

PL/SQL procedure successfully completed
```

当对表 S_EVT_ACT 重新收集统计信息的命令执行完毕后，坐席登录速度马上就恢复了正常，恢复正常后的坐席登录时间降到了 1 s 以内，上述问题暂时得到了解决。

再次用 TOPAS 命令查看现在的系统资源状况，TOPAS 的输出结果如下所示：

```
    Topas Monitor for host:    XXXXXXXX          EVENTS/QUEUES    FILE/TTY
Mon Dec  6 16:30:23 2010   Interval:  2       Cswitch    4621  Readch    95.6M
                                           Syscall   92227  Writech  205.1K
Kernel    1.1   |#                          | Reads     24668  Rawin        0
User      2.3   |#                          | Writes      301  Ttyout     531
Wait      0.1   |#                          | Forks         4  Igets        0
Idle     96.6   |############################| Execs         4  Namei      377
Physc =  0.77                    %Entc=   3.9  Runqueue    2.5  Dirblk       0
                                             Waitqueue   0.0
Network  KBPS   I-Pack  O-Pack   KB-In  KB-Out
en2     963.3   528.5   607.5   418.1   545.2  PAGING          MEMORY
```

```
en3    331.8   324.5  286.5  131.9  199.9  Faults     574  Real,MB   81920
lo0      0.3     6.0    6.0    0.2    0.2   Steals       0  % Comp    37.7
                                           PgspIn       0  % Noncomp  3.3
Disk    Busy%    KBPS     TPS KB-Read KB-Writ PgspOut      0  % Client   3.3
hdisk23   0.5   169.0    12.5   128.5   40.5  PageIn       0
hdisk12  10.5   108.2    27.0     0.0  108.2  PageOut      8  PAGING SPACE
hdisk21  11.0   108.2    27.0     0.0  108.2  Sios         8  Size,MB   98304
hdisk19   0.5    61.2     3.5     0.0   61.2                  % Used     0.0
hdisk17   0.5    54.5     6.0     0.0   54.5  NFS (calls/sec)  % Free   100.0
hdisk20   0.0    48.5     4.0     8.5   40.0  ServerV2     0
hdisk18   0.0    48.0     3.0     8.0   40.0  ClientV2     0   Press:
hdisk13   1.5    34.0     4.5     4.0   30.0  ServerV3     0   "h" for help

Name           PID  CPU%  PgSp Owner
ssmagent    2818238   1.7  23.5 root
oracle      4169760   0.6   7.0 oracle
……省略显示部分内容
oracle      4055148   0.0   6.7 oracle
oracle      2449466   0.0   7.2 oracle
Signal 2 received
```

从上述 TOPAS 的显示结果可以看出，现在系统资源的使用状况有了明显下降，磁盘的繁忙程度、en2 的网卡流量和 USER CPU 都已降到了正常水平。

至此，上述性能问题已经得到了暂时的解决，我们是不是可以就此安枕无忧了？事情远没有这么简单，我们还没有找到导致上述问题出现的根本原因。

我个人一直认为，**SQL 优化最有技术含量的部分不在于通过种种手段（比如重新收集统计信息等）调整目标 SQL 的执行计划，缩短其执行时间，解决该 SQL 的性能问题，而是在于要知道 CBO 为什么在一开始会选错执行计划，要知道 CBO 选错执行计划的根本原因。**

以上述坐席登录慢的问题为例，如果没有搞清楚 CBO 为什么会选错执行计划，那么虽然现在通过重新对表 S_EVT_ACT 收集统计信息临时解决了上述问题，但这种解决方案只是暂时性的，以后只要上述 SQL 中那 18 个表及相关对象的统计信息发生了变化，则坐席登录慢的问题还是有可能再次出现。所以，只有分析清楚 CBO 为什么会在一开始选错执行计划，才能从根本上解决坐席登录慢的问题，真正避免该问题的再次出现。

我们现在来对之前所做的分析做一下总结。

（1）造成 Call Center 系统坐席登录缓慢的原因，是 CBO 在执行上述 18 个表关联的 SQL 时错误地选择了对数据量为 730 多万的大表 S_EVT_ACT 的全表扫描（表 S_EVT_ACT 的统计信息是准确的）。

（2）上述目标 SQL 的四份执行计划所对应成本值过于接近，就是导致坐席登录慢的问题多次不间断出现的原因，坐席登录慢是因为 CBO 在执行该 SQL 时选择了那个对表 S_EVT_ACT 做全表扫描所对应的执行计划，而坐席登录速度正常时 CBO 选择的则是另外三份执行计划中的任意一份（这三份执行计划在访问表 S_EVT_ACT 时均走的是对索引 S_EVT_ACT_P1 的索引唯一性扫描）。

所以，很显然这里找出 CBO 选错执行计划的根本原因的关键，就在于为什么对数据量为 730 多万行的大表 S_EVT_ACT 做全表扫描操作所对应执行计划的成本值会与其他三份在访问表 S_EVT_ACT 时走对索引

S_EVT_ACT_P1 的索引唯一性扫描所对应执行计划的成本值如此接近。

现在再来仔细看一下那份对表 S_EVT_ACT 做全表扫描操作所对应的执行计划：

```
Plan 1(PHV: 4128147724)
-----------------------
Execution Plan
----------------------------------------------------------------------------------------------------
| Id  | Operation                                | Name              | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                         |                   |       |       |    42 (100)|          |
|   1 |  NESTED LOOPS OUTER                      |                   |    10 | 25020 |    42   (0)| 00:00:01 |
|   2 |   NESTED LOOPS OUTER                     |                   |    10 | 24880 |    41   (0)| 00:00:01 |
|   3 |    NESTED LOOPS OUTER                    |                   |    10 | 24620 |    38   (0)| 00:00:01 |
|   4 |     NESTED LOOPS OUTER                   |                   |    10 | 24500 |    37   (0)| 00:00:01 |
|   5 |      NESTED LOOPS OUTER                  |                   |    10 | 23770 |    31   (0)| 00:00:01 |
|   6 |       NESTED LOOPS OUTER                 |                   |    10 | 22830 |    25   (0)| 00:00:01 |
|   7 |        NESTED LOOPS OUTER                |                   |    10 | 22080 |    24   (0)| 00:00:01 |
|   8 |         NESTED LOOPS OUTER               |                   |    10 | 21390 |    18   (0)| 00:00:01 |
|   9 |          NESTED LOOPS                    |                   |    10 | 21230 |    15   (0)| 00:00:01 |
|  10 |           NESTED LOOPS OUTER             |                   |    10 | 20880 |    12   (0)| 00:00:01 |
|  11 |            NESTED LOOPS OUTER            |                   |    10 | 16630 |     9   (0)| 00:00:01 |
|  12 |             NESTED LOOPS OUTER           |                   |    10 | 15990 |     8   (0)| 00:00:01 |
|  13 |              NESTED LOOPS OUTER          |                   |    10 | 11990 |     7   (0)| 00:00:01 |
|  14 |               NESTED LOOPS OUTER         |                   |    10 | 10370 |     6   (0)| 00:00:01 |
|  15 |                NESTED LOOPS OUTER        |                   |    10 |  9690 |     5   (0)| 00:00:01 |
|  16 |                 NESTED LOOPS OUTER       |                   |    10 |  9050 |     4   (0)| 00:00:01 |
|  17 |                  NESTED LOOPS            |                   |    10 |  6290 |     3   (0)| 00:00:01 |
|  18 |                   INDEX UNIQUE SCAN      | S_PARTY_P1        |     1 |    12 |     1   (0)| 00:00:01 |
|  19 |                   TABLE ACCESS FULL      | S_EVT_ACT         |    10 |  6170 |     2   (0)| 00:00:01 |
|  20 |                  TABLE ACCESS BY INDEX ROWID| S_OPTY          |     1 |   276 |     1   (0)| 00:00:01 |
|  21 |                   INDEX UNIQUE SCAN      | S_OPTY_P1         |     1 |       |     1   (0)| 00:00:01 |
|  22 |                 TABLE ACCESS BY INDEX ROWID | S_TMSHT_LINE     |     1 |    64 |     1   (0)| 00:00:01 |
|  23 |                  INDEX UNIQUE SCAN       | S_TMSHT_LINE_P1   |     1 |       |     1   (0)| 00:00:01 |
|  24 |                TABLE ACCESS BY INDEX ROWID | S_PROJITEM       |     1 |    68 |     1   (0)| 00:00:01 |
|  25 |                 INDEX UNIQUE SCAN        | S_PROJITEM_P1     |     1 |       |     1   (0)| 00:00:01 |
|  26 |               TABLE ACCESS BY INDEX ROWID | S_PROD_DEFECT     |     1 |   162 |     1   (0)| 00:00:01 |
|  27 |                INDEX UNIQUE SCAN         | S_PROD_DEFECT_P1  |     1 |       |     1   (0)| 00:00:01 |
|  28 |              TABLE ACCESS BY INDEX ROWID | S_PROJ            |     1 |   400 |     1   (0)| 00:00:01 |
|  29 |               INDEX UNIQUE SCAN          | S_PROJ_P1         |     1 |       |     1   (0)| 00:00:01 |
|  30 |             TABLE ACCESS BY INDEX ROWID  | S_EXP_RPT         |     1 |    64 |     1   (0)| 00:00:01 |
|  31 |              INDEX UNIQUE SCAN           | S_EXP_RPT_P1      |     1 |       |     1   (0)| 00:00:01 |
|  32 |            TABLE ACCESS BY INDEX ROWID   | S_EVT_ACT_SS      |     1 |   425 |     1   (0)| 00:00:01 |
|  33 |             INDEX RANGE SCAN             | S_EVT_ACT_SS_U1   |     1 |       |     1   (0)| 00:00:01 |
|  34 |           TABLE ACCESS BY INDEX ROWID    | S_ACT_EMP         |     1 |    35 |     1   (0)| 00:00:01 |
|  35 |            INDEX RANGE SCAN              | S_ACT_EMP_F1      |     1 |       |     1   (0)| 00:00:01 |
|  36 |          TABLE ACCESS BY INDEX ROWID     | S_USER            |     1 |    16 |     1   (0)| 00:00:01 |
|  37 |           INDEX UNIQUE SCAN              | S_USER_U2         |     1 |       |     1   (0)| 00:00:01 |
|  38 |         TABLE ACCESS BY INDEX ROWID      | S_EVT_MAIL        |     1 |    69 |     1   (0)| 00:00:01 |
|  39 |          INDEX RANGE SCAN                | S_EVT_MAIL_U1     |     1 |       |     1   (0)| 00:00:01 |
|  40 |        TABLE ACCESS BY INDEX ROWID       | S_ORG_EXT         |     1 |    75 |     1   (0)| 00:00:01 |
|  41 |         INDEX UNIQUE SCAN                | S_ORG_EXT_U3      |     1 |       |     1   (0)| 00:00:01 |
|  42 |       TABLE ACCESS BY INDEX ROWID        | S_EVT_MKTG        |     1 |    94 |     1   (0)| 00:00:01 |
|  43 |        INDEX RANGE SCAN                  | S_EVT_MKTG_U1     |     1 |       |     1   (0)| 00:00:01 |
|  44 |      TABLE ACCESS BY INDEX ROWID         | S_EVT_CAL         |     1 |    73 |     1   (0)| 00:00:01 |
|  45 |       INDEX RANGE SCAN                   | S_EVT_CAL_U1      |     1 |       |     1   (0)| 00:00:01 |
|  46 |     INDEX UNIQUE SCAN                    | S_PARTY_P1        |     1 |    12 |     1   (0)| 00:00:01 |
|  47 |    TABLE ACCESS BY INDEX ROWID           | S_CONTACT         |     1 |    26 |     1   (0)| 00:00:01 |
|  48 |     INDEX UNIQUE SCAN                    | S_CONTACT_U2      |     1 |       |     1   (0)| 00:00:01 |
|  49 |   TABLE ACCESS BY INDEX ROWID            | S_CONTACT         |     1 |    14 |     1   (0)| 00:00:01 |
|  50 |    INDEX UNIQUE SCAN                     | S_CONTACT_U2      |     1 |       |     1   (0)| 00:00:01 |
----------------------------------------------------------------------------------------------------
```

上述执行计划中 Id = 19 的执行步骤所对应的列 Rows 的值为 10，列 Cost 的值为 2，这说明 CBO 评估出对一个实际数据量为 730 多万且统计信息准确的大表 S_EVT_ACT 执行全表扫描操作后返回结果集的 Cardinality 的值为 10，成本值为 2。这里 CBO 估算的成本值仅为 2，这是一个明显异常的数字，在默认情况下，对一个实际数据量为 730 多万且统计信息准确的表执行全表扫描操作，CBO 估算出来的成本值肯定过万了（关于 Oracle 数据库中全表扫描成本值的计算方法，会在第 5 章的“5.9 系统统计信息”中详细描述，这里不再赘述）。如果 CBO 这里对全表扫描表 S_EVT_ACT 的成本值估算正常的话，那么上述执行计划总的成本值也会过万，而不会是现在的 42，这样就会远大于其他三份执行计划的成本值 41 、43 和 44；如果是这样的话，CBO 就不可能会选择那份对表 S_EVT_ACT 做全表扫描操作所对应的执行计划了。

所以，这里 CBO 评估出对大表 S_EVT_ACT 执行全表扫描操作后的成本值仅为 2 就是导致 CBO 此时选错执行计划的根本原因，这同时也是导致坐席登录慢的问题多次不间断出现的根本原因。

在“1.2.1 优化器的模式”中曾经提到过：当 OPTIMIZER_MODE 的值为 FIRST_ROWS_*n*（*n* = 1, 10, 100, 1000）时，Oracle 会把那些能够以最快的响应速度返回头 *n*（*n* = 1, 10, 100, 1000）条记录所对应的执行步骤的成本值修改成一个很小的值（远远小于默认情况下 CBO 对同样执行步骤所计算出的成本值）。

基于上述知识点，我们当时的第一感觉是是不是参数 OPTIMIZER_MODE 的值被修改成 FIRST_ROWS_*n*（*n* = 1, 10, 100, 1000）了？

但从如下查询结果来看，OPTIMIZER_MODE 的值在系统级别并没有被修改，其值还是默认的 ALL_ROWS：

```
SQL> select name,value from v$parameter where name='optimizer_mode';

NAME                      VALUE
------------------    ---------------
optimizer_mode          ALL_ROWS
```

OPTIMIZER_MODE 的值在系统级别没有被修改并不意味着它在 Session 级就还是默认值，所以现在还有一种可能，就是 OPTIMIZER_MODE 的值在 Session 级被修改成了 FIRST_ROWS_*n*（*n* = 1, 10, 100, 1000）。

怎么来看 OPTIMIZER_MODE 的值是不是在 Session 级被修改成了 FIRST_ROWS_*n*（*n* = 1, 10, 100, 1000）呢？可以用如下方法随便从 Call Center 系统的后台数据库中挑几个 Session 并对其做 Process Dump：

```
SQL> conn / as sysdba;
SQL> oradebug setospid <process ID>
SQL> oradebug unlimit
SQL> oradebug dump processstate 10
SQL> oradebug tracefile_name
```

从产生的 TRACE 文件中均能看到如下内容：

```
Optimizer environment:
  ……省略显示部分内容
  optimizer_mode                 = first_rows_10
  ……省略显示部分内容

Cursor frame dump
----------------------------------------
 sqltxt(700000308f29da0)=
ALTER SESSION SET OPTIMIZER_MODE = FIRST_ROWS_10
  hash=12d4e0328ec07bc2dff05c8b9aacc525
  parent=7000002d0f7e0a0 maxchild=00 plk=7000002b6a25c30 ppn=n
cursor instantiation=11043a968 used=1291603055
 child#0(0) pcs=0
  clk=0 ci=0 pn=0 ctx=0
 kgsccflg=1 llk[11045bd08,11043ad38] idx=26
 xscflg=100008 fl2=0 fl3=20000 fl4=40
 Frames pfr 0 siz=0 efr 0 siz=0
```

注意，上述 TRACE 文件的内容中有“optimizer_mode = first_rows_10”和“sqltxt(700000308f29da0) = ALTER

SESSION SET OPTIMIZER_MODE = FIRST_ROWS_10”，这说明 OPTIMIZER_MODE 的值确实是在 Session 级被修改成了 FIRST_ROWS_10。

我们现在来验证一下当 OPTIMIZER_MODE 的值被修改成 FIRST_ROWS_10 后对 CBO 估算全表扫描表 S_EVT_ACT 的成本值的影响。

上述 SQL 中对表 S_EVT_ACT 的直接谓词条件就只有“T18.APPT_REPT_REPL_CD IS NULL”，这个谓词条件在全表扫描表 S_EVT_ACT 时会被当作过滤查询条件，并且它会影响 CBO 对于全表扫描表 S_EVT_ACT 的结果集的 Cardinality 和成本值的估算，所以我们在验证时应该把这个谓词条件加上。

我们只是想验证当 OPTIMIZER_MODE 的值被修改成 FIRST_ROWS_10 后对 CBO 估算全表扫描表 S_EVT_ACT 的成本值的影响，所以这里并不需要完整地执行上述 SQL，而是只需执行如下 SQL 即可：

```
SELECT * FROM
SIEBEL.S_EVT_ACT T18
 WHERE T18.APPT_REPT_REPL_CD IS NULL;
```

在 Oracle 10g 及其以上的 Oracle 数据库版本中，OPTIMIZER_MODE 的默认值为 ALL_ROWS，我们现在先在默认情况下对上述 SQL 执行 EXPLAIN PLAN 命令：

```
SQL> explain plan for
  2  SELECT * FROM
  3   SIEBEL.S_EVT_ACT T18
  4  WHERE T18.APPT_REPT_REPL_CD IS NULL;

Explained.
```

上述 SQL 的执行计划如下所示：

```
SQL> select * from table(dbms_xplan.display);
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------
Plan hash value: 4199372896
------------------------------------------------------------------------------------
| Id  | Operation          | Name       | Rows  | Bytes | Cost (%CPU)| Time     |
------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |            |  7349K|  4990M|   150K  (3)| 00:30:03 |
|*  1 |  TABLE ACCESS FULL | S_EVT_ACT  |  7349K|  4990M|   150K  (3)| 00:30:03 |
------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("T18"."APPT_REPT_REPL_CD" IS NULL)

13 rows selected
```

从显示内容可以看出，在 OPTIMIZER_MODE 的值为 ALL_ROWS 的情况下，CBO 评估出来全表扫描表 S_EVT_ACT 的结果集的 Cardinality 的值为 7349K，成本值为 150K。这说明如果 OPTIMIZER_MODE 的值是默认值 ALL_ROWS，则 CBO 评估出来的上述 18 个表关联 SQL 的总成本值就会大于 150K，如果是这样，那么 CBO 就不可能会选错执行计划了。

我们现在在当前 Session 中把 OPTIMIZER_MODE 的值改成 FIRST_ROWS_10：

```
SQL> alter session set optimizer_mode = first_rows_10;

Session altered.
```

然后再次对上述 SQL 执行 EXPLAIN PLAN 命令：

```
SQL> explain plan for
  2  SELECT * FROM
  3   SIEBEL.S_EVT_ACT T18
  4  WHERE T18.APPT_REPT_REPL_CD IS NULL;

Explained.
```

现在的执行计划如下所示：

```
SQL> select * from table(dbms_xplan.display);
PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
Plan hash value: 4199372896
-------------------------------------------------------------------------------
| Id  | Operation          | Name       | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |            |    10 |  7120 |     2   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL | S_EVT_ACT  |    10 |  7120 |     2   (0)| 00:00:01 |
-------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("T18"."APPT_REPT_REPL_CD" IS NULL)

13 rows selected
```

从上述显示内容中我们可以看出，在 OPTIMIZER_MODE 的值为 FIRST_ROWS_10 的情况下，CBO 评估出来全表扫描表 S_EVT_ACT 的结果集的 Cardinality 的值从之前的 7349K 降到了现在的 10，成本值也从之前的 150K 降到了现在的 2，这里的“10”和“2”与 18 个表关联 SQL 的执行计划中对表 S_EVT_ACT 执行全表扫描操作后所对应的 Cardinality 和成本值完全吻合！

问题至此真相大白！

我们来总结一下：**导致 CBO 评估出对一个实际数据量为 730 多万且统计信息准确的大表 S_EVT_ACT 执行全表扫描操作后的成本值仅为 2 的原因，是参数 OPTIMIZER_MODE 的值在 Session 级别被修改成了 FIRST_ROWS_10，这同时也是导致坐席登录慢的问题多次不间断出现的根本原因。**

分析清楚了根本原因，从根本上解决上述问题就非常容易了，我们至少会有如下这两种解决方法。

（1）修改各个 Session 中对于参数 OPTIMIZER_MODE 的设置，将其值修改为默认值 ALL_ROWS，SIEBEL 的系统提供了界面控制 Session 级优化器参数设置的功能，所以在 Session 级修改参数 OPTIMIZER_MODE 的值应该是非常容易的。

（2）如果不能在 Session 级修改参数 OPTIMIZER_MODE 的值，我们还可以使用 SQL Profile。在上述 18 个表关联 SQL 中加入 Hint（即/*+ index(T18 S_EVT_ACT_P1) */），并用加入 Hint 后改写 SQL 的执行计划替换原 SQL 的执行计划，这样就可以实现在不改变原 SQL 的 SQL 文本的情况下更改其执行计划的目的。注意，虽然上述 SQL 使用了绑定变量（AND T1.EMP_ID = :1），但 SQL Profile 同样适用于包含绑定变量的 SQL，关于 SQL Profile 的用法，我们会在第 2 章的“2.6.1 使用 SQL Profile 来稳定执行计划”中详细说明，这里不再赘述。

1.4 总结

本章详细介绍了 Oracle 数据库中与优化器相关的各个方面的内容，包括优化器的模式、结果集（Row Source）、集的势（Cardinality）、可选择率（Selectivity）、可传递性（Transitivity）、各种数据访问的方法，以及与表连接相关的内容，最后还介绍了一个优化器模式对 CBO 计算成本带来巨大影响的实例。

以下是对本章主要内容的回顾。

- 在使用 RBO 的情况下，我们可以通过调整相关对象在数据字典缓存中的缓存顺序，改变目标 SQL 中所涉及的各个对象在该 SQL 文本中出现的先后顺序，或者等价改写该 SQL 来调整其执行计划。
- 成本是指 Oracle 根据相关对象的统计信息计算出来的一个值，它实际上代表了 Oracle 根据相关统计信息估算出来的目标 SQL 的对应执行路径的 I/O、CPU 和网络资源的消耗量。
- Cardinality 和 Selectivity 的值会直接影响 CBO 对于相关执行步骤成本值的估算，进而影响 CBO 对于目标 SQL 执行计划的选择。
- 可传递性的意义在于提供了更多的执行路径（Access Path）给 CBO 做选择，增加了走出更高效执行计划的可能性。
- 优化器的模式对 CBO 计算成本（进而对 CBO 选择执行计划）有着决定性的影响。
- 不是说全表扫描不好，事实上 Oracle 在做全表扫描操作时会使用多块读，这在目标表的数据量不大时执行效率是非常高的，但全表扫描最大的问题就在于走全表扫描的目标 SQL 的执行时间会不稳定、不可控，这个执行时间一定会随着目标表数据量的递增而递增。
- 通过 B 树索引访问表里行记录的效率并不会随着相关表的数据量的递增而显著降低，即通过走索引访问数据的时间是可控的、基本稳定的，这也是走索引和全表扫描的最大区别。
- Oracle 中索引全扫描的执行结果是有序的，并且是按照该索引的索引键值列来排序的，这意味着走索引全扫描能够既达到排序的效果，同时又能避免对该索引的索引键值列的真正排序操作。另外，Oracle 中能做索引全扫描的前提条件是目标索引至少有一个索引键值列的属性是 NOT NULL。
- 索引快速全扫描是可以并行执行的，它的执行结果不一定是有序的。
- Oracle 中的索引跳跃式扫描仅仅适用于那些目标索引的前导列的 distinct 值数量较少、后续非前导列的可选择性又非常好的情形，因为索引跳跃式扫描的执行效率一定会随着目标索引前导列的 distinct 值数量的递增而递减。
- 关键字“(+)”出现在哪个表的连接列后面，就表明哪个表会以 NULL 值来填充那些不满足连接条件并位于该表中的查询列，此时应该以关键字“(+)”对面的表来作为外连接的驱动表，这里的关键是决定哪个表是驱动表。
- 通常情况下，排序合并连接的执行效率会远不如哈希连接的执行效率高，但排序合并连接的使用范围更广，因为哈希连接只能用于等值连接条件，而排序合并连接还能用于其他连接条件（例如<、<=、>、>=）。
- 通常情况下，排序合并连接并不适合 OLTP 类型的系统，其本质原因是对于 OLTP 类型的系统而言，排序是非常昂贵的操作，当然，如果能避免排序操作，那么即使是 OLTP 类型的系统，还是可以使用排序合并连接的。
- 如果驱动表所对应的驱动结果集的记录数较少，同时在被驱动表的连接列上又存在唯一性索引（或者在被驱动表的连接列上存在选择性很好的非唯一性索引），那么此时使用嵌套循环连接的执行效率就会

非常高；但如果驱动表所对应的驱动结果集的记录数很大，即便在被驱动表的连接列上存在索引，此时使用嵌套循环连接的执行效率也不会高。

- 大表也可以作为嵌套循环连接的驱动表，关键看目标 SQL 中指定的谓词条件（如果有的话）能否将驱动结果集的数据量降下来。
- 哈希连接只适用于 CBO，它也只能用于等值连接条件（即使是哈希反连接，Oracle 实际上也是将其转换成了等价的等值连接）。
- 哈希连接很适合于小表和大表之间做表连接且连接结果集的记录数较大的情形，特别是在小表的连接列的可选择性非常好的情况下，这时候哈希连接的执行时间就可以近似看作是和全表扫描那个大表所耗费的时间相当。
- 半连接和普通的内连接不同，半连接实际上会去重。

总之，对优化器的认识是在 Oracle 数据库中做 SQL 优化基础中的基础，只有全面、深入地了解与优化器相关的基础知识，才能在 Oracle 数据库中做好 SQL 优化。