Oracle 10G 新特性

作者：fuyuncat
来源：WWW.HelloDBA.COM

作者简介

黄玮，男，99 年开始从事 DBA 工作，有多年的水利、军工、电信及航运行业大型数据库 Oracle 开发、设计和维护经验。

曾供职于南方某著名电信设备制造商——H 公司。期间，作为 DB 组长，负责设计、开发和维护彩铃业务的数据库系统。目前，H 公司的彩铃系统是世界上终端用户最多的彩铃系统。最终用户数过亿。

目前供职于某世界著名物流公司，负责公司的电子物流系统的数据库开发、维护工作。

msn: fuyuncat@hotmail.com
Email：fuyuncat@gmail.com

1. 等待接口

前言

在处理被 ADDM 捕捉的突发的性能问题中，10g 等待接口提供非常有价值的数据用于诊断。作为一个 DBA，你可能遇到过很多次用户抱怨“数据库非常慢了”。那么你是如何定位这种问题的呢？你第一步要做的肯定是查看是否有会话在等待数据库内部或外部的什么资源。

Oracle 提供了一个简单但是又很有效的机制来获取这些信息：视图 V$SESSION_WAIT。这一视图提供了各种信息以帮助诊断如一个会话正在等待和已经等待的事件、及等待了多少、等待时间多长。例如，如果会话正在等待事件“db file sequential read”，字段 P1 和 P2 就标识会话正在等待的数据块的 file_id 和 block_id。

对于大多数等待事件来说，这个视图已经足够了。但由于以下两个原因，它很难成为一个强有力的优化工具：

- 这个视图只是当前状态的一个快照。当等待不存在了，会话早前关于这些等待的历史信息也没有了，使事后诊断分析变得很困难。

V$SESSION_EVENT 视图提供了累积的但不够详细的数据。

- V$SESSION_WAIT 视图仅包含了等待事件的信息。要获取其他相关信息如 USERID 和终端信息就必须和视图 V$SESSION 连接查询。

在 Oracle 10g 中，等待接口从根本上被重新设计了，提供了更多的信息，而且减少了 DBA 的干预。在这篇文章中，我们将探索这些特性，看看它们如何帮助我们来诊断性能问题。对于大多数性能问题，你能从 ADDM 中得到扩展的分析信

息，但是 ADDM 没有捕捉到哪些即时问题的信息。而等待接口提供了非常有用的数据用于诊断。

增强的会话等待

首先一个增强就是对视图 V$ESSION_WAIT 本身的。这可以从一个例子来解释。加入您的用户抱怨会话被 hang 住了。你查出这个会话的 SID，并从视图 V$SESSION_WAIT 中找出这个 SID 的记录。以下就是输出内容：

```
SID                 : 269
SEQ#                : 56
EVENT               : enq: TX - row lock contention
P1TEXT              : name|mode
P1                  : 1415053318
P1RAW               : 54580006
P2TEXT              : usn<<16 | slot
P2                  : 327681
P2RAW               : 00050001
P3TEXT              : sequence
P3                  : 43
P3RAW               : 0000002B
WAIT_CLASS_ID       : 4217450380
WAIT_CLASS#         : 1
WAIT_CLASS          : Application
WAIT_TIME           : -2
SECONDS_IN_WAIT     : 0
STATE               : WAITED UNKNOWN TIME
```

请注意哪些黑体字段。在这些字段中，WAIT_CLASS_ID、WAIT_CLASS#和 WAIT_CLASS 字段是 10g 中新增的。字段 WAIT_CLASS 表示这个等待的类型是一个有效的等待事件还是一个可以被忽略的空闲等待事件。在上面的例子中，等待类型是”Application”，表明它在等待你的干预。

这一字段让你更加明确哪些字段可以用于调优。例如，你可以用以下查询来获取等待这个事件的会话：

```
SQL> select wait_class, event, sid, state, wait_time, seconds_in_wait
2   from v$session_wait
3   order by wait_class, event, sid
4 /
WAIT_CLASS    EVENT   SID  STATE      WAIT_TIME SECONDS_IN_WAIT
----------  ---------------------- ---------- ------------------- ---------- ---------------
Application enq:  TX -       269 WAITING               0             73
        row lock contention
Idle        Queue Monitor Wait         270 WAITING     0             40
Idle        SQL*Net message from client 265 WAITING       0             73
Idle        jobq slave wait            259 WAITING       0           8485
Idle        pmon timer             280 WAITING         0             73
```

Idle rdbms ipc message 267 WAITING 0 184770
Idle wakeup time manager 268 WAITING 0 40
*Network SQL*Net message to client 272 WAITED SHORT TIME -1 0*

在这你可以看到几个事件（如 Queue Monitor Wait 和 JobQueue Slave）被明确标注为 Idle 事件。你可以把它们视为非阻塞等待忽略掉。但是，有时候这些”idle”事件预示着一个内部问题。例如，与 SQL*NET 相关的事件可能预示网络可能出现潜在问题。

另外一个需要注意的重要信息就是字段 WAIT_TIME 的值是－2。一些平台，如 Windows，不支持快速计时机制。如果在哪些平台上，初始化参数 TIMED_STATISTICS 没有设置为 on，就无法检测到精确的计时统计。在那样的情况下，在 Oracle9i 中，这个字段的值是一个非常大的值，它会让以后的问题定位产生混乱。在 10g 中，值－2 就表示是这种情况——平台不支持快速计时机制并且 TIME_STATISTICS 被设置为 off。

会话也会显示等待信息

还记得前面说需要将视图 V$SESSION_WAIT 和 V$SESSION 连接查询来获取关于会话的细节信息吗？在 10g 中，这将成为历史。10g 中的视图 V$SESSION 也会显示和 V$SESSION_WAIT 一样的等待事件信息。以下便是 V$SESSION 用于显示会话当前所等待的等待事件而新增的字段。

EVENT# NUMBER
EVENT VARCHAR2(64)
P1TEXT VARCHAR2(64)
P1 NUMBER
P1RAW RAW(4)
P2TEXT VARCHAR2(64)
P2 NUMBER
P2RAW RAW(4)
P3TEXT VARCHAR2(64)
P3 NUMBER
P3RAW RAW(4)
WAIT_CLASS_ID NUMBER
WAIT_CLASS# NUMBER
WAIT_CLASS VARCHAR2(64)
WAIT_TIME NUMBER
SECONDS_IN_WAIT NUMBER
STATE VARCHAR2(19)

这些字段和 V$SESSION_WAIT 中的是一样的，并且显示的信息也相同。这样就无需再查询 V$SESSION_WAIT 视图了，要获取到会话的等待信息就只需要查询一个视图就行了。

让我们看看前面这个例子：SID 为 269 的会话正在等待事件 enq: TX ?C row lock contention。这表示它正在等待另外一个会话持有的锁。要诊断这个问题，你就必须定位到另外一个会话。如何做呢？

在 9i 和以下版本中，你可能需要写一个复杂的查询来获取到持有锁的会话的 SID。在 10g 中，你就只需要执行以下查询就可以了：

```
SQL> select BLOCKING_SESSION_STATUS, BLOCKING_SESSION
2    from v$session
3    where sid = 269;
BLOCKING_SE BLOCKING_SESSION
----------- ----------------
VALID                  265
```

这个结果说明：SID 为 265 的会话阻塞了会话 269。

有多少等待？

有一个用户的会话一致在保持着，他的问题还没有得到满意的解决。为什么他的会话会保持这么长事件还没有完成呢？你可以通过以下查询来定位：

```
SQL> select * from v$session_wait_class where sid = 269;
SID SERIAL# WAIT_CLASS_ID WAIT_CLASS# WAIT_CLASS    TOTAL_WAITS
TIME_WAITED
---- ------- ------------- ----------- ------------- ----------- -----------
 269    1106    4217450380        1 Application         873      261537
 269    1106    3290255840        2 Configuration         4           4
 269    1106    3386400367        5 Commit                1           0
 269    1106    2723168908        6 Idle                 15      148408
 269    1106    2000153315        7 Network              15           0
 269    1106    1740759767        8 User I/O             26           1
```

请注意关于这个会话等待的诸多信息。现在你就知道了这一会话已经等待和应用相关的等待事件共 873 次、261537 厘秒，等待网络相关的事件 15 次，等等。

用同样方法，你可以通过以下查询获取到整个系统范围的等待事件分类的统计信息，时间单位还是厘秒：

```
SQL> select * from v$system_wait_class;
WAIT_CLASS_ID WAIT_CLASS# WAIT_CLASS    TOTAL_WAITS TIME_WAITED
------------- ----------- ------------- ----------- -----------
   1893977003           0 Other              2483       18108
   4217450380           1 Application        1352      386101
   3290255840           2 Configuration        82         230
   3875070507           4 Concurrency          80         395
   3386400367           5 Commit             2625        1925
   2723168908           6 Idle             645527   219397953
   2000153315           7 Network            2125           2
   1740759767           8 User I/O           5085        3006
   4108307767           9 System I/O       127979       18623
```

大多数问题的发生都不是孤立的。在它们背后还存在一些其他问题，这些问题可以通过一种样式来定位。这一样式可以通过以下查询语句来查询一个历史视图来获得：

```
SQL> select wait_class#, wait_class_id,
```

```
2    average_waiter_count "awc", dbtime_in_wait,
3    time_waited,  wait_count
4    from v$waitclassmetric
5 /
WAIT_CLASS# WAIT_CLASS_ID  AWC DBTIME_IN_WAIT TIME_WAITED
WAIT_COUNT
----------- ------------ ---- -------------- ----------
          0   1893977003    0              0          0
          1   4217450380    2             90       1499
          2   3290255840    0              0          4
          3   4166625743    0              0          0
          4   3875070507    0              0          1
          5   3386400367    0              0          0
          6   2723168908   59              0     351541
          7   2000153315    0              0          0
          8   1740759767    0              0          0
          9   4108307767    0              0          8
         10    2396326234    0              0          0
         11    3871361733    0              0          0
```

请注意字段 WAIT_CLASS_ID 和他的统计数据。对于值 4217450380，我们发现有 2 个会话在等待这一类型的事件共 5 次、1499 厘秒。但是这一类型等待是什么等待事件呢？你可以从视图 V$SYSTEM_WAIT_CLASS 中得到这个信息。如上所示，这是 Application 类型事件。

再注意 DBTIME_IN_WAIT 这一非常有用的字段。你也许还记得在 10g 的 AWR 中会在很细的粒度下记录下数据库的消耗时间。DBTIME_IN_WAIT 就表示数据库消耗的等待时间。

跟踪

当用户的问题解决后，你可能还要继续跟踪有什么其他不同的等待事件导致了他的问题。当然，你可以很轻松的通过查询 V$SESSION_WAIT 视图来获取答案。但是很不幸，这些等待事件都已经不存在了。因此也就没有它们的记录了。这时要如何做？

在 10g 中，视图 V$SESSION_WAIT_HISTORY 会自动记录下活动会话等待的最近 10 次的等待事件历史信息。要查找这些事件，只要执行以下语句：

```
SQL> select event, wait_time, wait_count
2    from v$session_wait_history
3    where sid = 265
4 /
EVENT                          WAIT_TIME WAIT_COUNT
------------------------------ ---------- ----------
log file switch completion              2          1
log file switch completion              1          1
log file switch completion              0          1
```

*SQL*Net message from client 49852 1*
*SQL*Net message to client 0 1*
enq: TX - row lock contention 28 1
*SQL*Net message from client 131 1*
*SQL*Net message to client 0 1*
log file sync 2 1
log buffer space 1 1

当会话停止或连接断开后，视图中的这些记录就没有了。然而，这些等待的历史信息会被维护在 AWR 表中以便以后的分析。查询这些会话等待的 AWR 视图是 V$ACTIVE_SESSION_HISTORY。

总结

由于 10g 的这些增强特性，分析性能问题变得十分简单。关于会话等待的历史信息帮助你诊断会话结束以后发生的问题。对等待的分类可以帮助你理解每个等待的重要性。

2.审计

10g 审计能力的提升

Oracle 10g 的审计会在非常细节的层次上捕捉用户的动作，它可以是手动审计、基于触发器审计。

假如用户 A 用以下语句更新了表里面得一条记录：

SQL> update SCOTT.EMP set salary = 12000 where empno = 123456;

你如何去跟踪这样的操作呢？在 Oracle 9i 中，审计只捕捉了谁做了操作，但不知道他做了什么？例如，它能让你知道 A 更新过用户 scott 的表 emp，但是却无法知道他更新了工号为 123456 的员工的薪水信息。它同样在改变字段 salary 之前无法显示处它的值——要捕捉这样的信息，你只能通过编写自己的触发器来在值改变前捕捉它，或者通过使用 Log Miner（日志挖掘器）从归档日志中找出来。

这两种方式都会让你付出较大代价来跟踪和记录数值的变化。使用触发器记录统计信息会造成很大的性能压力。处于性能考虑，在一些情况下（如在三层构架应用中）会禁止使用用户自定义的触发器。日志挖掘器不会产生性能问题，但它必须依赖于归档日志功能已经启动。

在 Oracle 9i 中出现的细密纹理审计（Fine-grained Auditing FGA），可以同 SCN 数重新构造老的数据来记录更低层次的变化，但是它只针对 select 查询，无法记录 update、insert 和 delete 这些 DML 语句。因此，在 oracle 10g 之前，使用触发器是唯一可以用来跟踪用户在低层次上做的数据操作的途径。

随着 10g 的带来了在审计方面的两个显著变化，这些限制将全都没有了。因为引入了两种类型的审计——标准审计（所有版本都有）和细密纹理审计（Oracle 9i 和以上版本才具备）。我们分别来了解他们，看看它们如何通过互补来提供一个单一的、强大的跟踪能力。

新功能

首先，FGA 现在提供了除 select 以为的 DML 审计。这些变化记录还是存储在同一个地方：FGA_LOG$，并且可以通过视图 DBA_FGA_AUDIT_TRAIL 查询到。除了 DML 语句外，你还可以选择使用触发器跟踪所有相关的字段或者只是其中的一小部分。

通过命令 AUDIT 执行的标准审计能够很容易的跟踪某个特殊对象。例如，你需要跟踪用户 scott 的表 EMP 上的所有更新操作，可以用以下命令：

SQL> AUDIT UPDATE on scott.emp BY ACCESS;

这一命令会在任意一个用户更新 scott.emp 表时记录在审计跟踪表 AUD$中，并通过 DBA_AUDIT_TRAIL 视图可以查询。

这一功能在 10g 以前的版本中也有。但是在哪些版本中，写入的跟踪信息仅限于一小部分相关信息，如执行操作的用户，时间，终端 ID 等。而哪些修改的值这样的重要信息却没有记录。在 10g 中，除了以前版本记录的哪些信息以外，还记录

了很多这种重要信息。审计的主要表 AUD$增加了几个字段来记录这些信息，相应的的视图的 DBA_AUDIT_TRAIL 也增加了这些字段以便查询。让我们再深入了解一下。

EXTENDED_TIMESTAMP 这一字段用 TMESTAMP(6)的格式记录了审计记录的时间戳，它以格林尼治时间（UTC）记录到了秒的第 9 位小数以后的时间以及相应的时区信息，下面时一个例子：

2004-3-13 18.10.13.123456000 -5:0

这意味着时间在美国东部标准时间 2004 年 3 月 13 日，时区比 UTC 完 5 个小时。这一扩展格式的时间在一个精确得多得精度上记录了审计跟踪时间，对于哪些数据库时间存在时区差得系统来说非常有用。

GLOBAL_UID 和 **PROXY_SESSIONID** 当要鉴别一个用于鉴权的管理组件如 Oracle Internet Directory 时，用户在数据库中可能以不同的形式可见。例如，他们可能作为一个企业用户鉴权进入数据库。审计这些用户时将无法在视图 DBA_AUDIT_TRAIL 的字段 USERNAME 中记录他们的企业 userid，这样记录下来的信息是无用的。在 Oracle 10g 中，如果没有进一步的操作或设置，全局（或企业）用户会被唯一的记录在字段 GLOBAL_UID 中。这一字段可以被用于查询目录服务器以找出这个企业用户的完整信息。

有时候企业用户需要通过一个代理用户连接数据库，特别是在多点应用时。一个用户可以通过以下方式授予代理权限：

SQL> alter user scott grant connect to appuser;

这一命令将使用户 scott 可以最为 appuser 连接数据库，成为一个代理用户。在这一例子中，字段 COMMENT_TEXT 将通过存入值 PROXY 来记录下 scott 作为代理用户的情况。但在 9i 中，代理用户的会话 ID 不会记录下来。在 10g 中，字段 PROXY_SESSIONID 记录下来代理用户的 session id。

INSTANCE_NUMBER 在 Oracle RAC 环境中，能够知道用户从哪个实例连上数据库会比较有用。在 10G 中，这个字段就记录了实例初始化参数设置的唯一的实例号。

OS_PROCESS 在 9i 及以下版本中，审计跟踪记录中只会记录 SID，而不会记录操作系统的进程 ID。但是操作系统进程 ID 对于以后对照 trace 文件是非常必要的。在 10g 中，这个字段就记录了操作系统进程 id。

TRANSACTIONID 这是一个极其重要的信息。加入用户执行了以下语句：

```
SQL> update CLASS set size = 10 where class_id = 123;
SQL> commit;
```

这一操作产生了一个事务，并记录一条审计记录。但你如何知道记录中实际记录的是什么呢？如果这条记录是一个事务，这个字段就记录了事务的 ID。你可以用它遇视图 FLASHBACK_TRANSACTION_QUERY 连接查询。下面就是一个例子：

```
SQL> select start_scn,  start_timestamp,
2       commit_scn, commit_timestamp, undo_change#, row_id, undo_sql
3    from flashback_transaction_query
4    where xid = '<the transaction id>';
```

除了哪些关于这个事务的统计信息，如 undo change#、rowid 等等，10g 中还记录了回归事务变化的 SQL 语句在字段 UNDO_SQL 中，记录了回归影响的行的 rowid 在字段 ROW_ID 中。

System Change Number (SCN) 最后，来看下记录的变化前的数据。如何得到它呢。在 9i 的 FGA 中，这个数据可以通过闪回查询得到。但是你必须首先从审计记录的 SCN 字段中中得到 System Change Number (SCN)。执行以下命令：

```
SQL> select size from class as of SCN 123456
2    where where class_id = 123;
```

这样可以得到变化前的数据了。

扩展 DB 审计

我们最想得到的是：捕捉在标准审计中没有记录的用户所执行的 SQL 语句和绑定的变量。通过设置一个参数可以进入 10g 的扩展审计，在参数文件中写入以下内容：

```
audit_trail = db_extended
```

这一设置激活了在相应字段中记录 SQL 语句和绑定的变量的值。在早期版本中是没有这一参数的。

什么时候需要触发器

避免错误 审计数据都是通过一个自治事务从源事务中从产生的。因此，及时源事务
已经回滚，它们还是会被提交。这有一个简单的例子来说明这一点。加入我们已经设定在表 CLASS 上审计 UPDATE 操作。一个用户执行一个 SQL 将 SIZE 数据从 20 更新到 10，然后回滚事务，如下所示：

```
SQL> update class set size = 10 where class_id = 123;
SQL> rollback
```

现在字段 SIZE 的数据是 20 而不是 10。尽管回归了，审计跟踪还是会捕捉到这些变化。在某些情况下，我们并不希望它这样做，特别是用户回滚了很多事务。在这种情况下，你就需要在 CLASS 表上建一个触发器捕捉仅事务提交时的变化。如果在表 CLASS 上有一个插入数据到用户定义的审计跟踪中的触发器，当回归时审计跟踪也会回滚。

捕捉变化前的数据 Oracle 提供的审计跟踪没有记录变化前后的数据。例如。上面的变化会产生一条审计记录，记录下 SQL 语句和变化时的 SCN 数，但是没有变化前的数据（20）。这个数据可以通过使用 SCN 数从闪回查询中获取到，但是这依赖于必须能从回滚段中获取到相应信息。这些信息能否获取到就看参数 undo_retention 设置的大小了，如果在它设置的时间范围内，之前的数据就可以获取到。使用触发器捕捉这个值就不需要依赖遇这个参数的设定了，可以随时获取到变化前的值。

统一审计跟踪

由于 FGA 和标准审计捕捉的时类似类型的信息。他们可以共同提供很多十分有用的信息。Oracle 10g 通过视图 DBA_COMMON_AUDIT_TRAIL 将这些跟踪信息统一成一个统一的跟踪信息。它是通过 UNION ALL 将视图 DBA_AUDIT_TRAIL 和 DBA_FGA_AUDIT_TRAIL 联合起来。但是这两个统计跟踪数据还是有很大的不同的。

3. 工作量自动收集

学习使用 10G 新特性来收集数据库性能统计和度量数据来分析和调优。显示数据的确切执行时间，以及保存的会话信息。

10G 的性能分析新特性

当数据库发生了性能问题时，如何去定位？比较常用的方法是采用一个既定的模式：解决诸如“是不是同一问题的再现？”、“是否在某一特殊时间段发生？”、“两个问题之间是否存在联系？”等问题，这样通常能得到一个比较好的诊断结果。 作为一个 DBA，你可能使用一个第三方或者自己开发的工具来收集数据库运行期间的精细统计数据，并从中得到性能度量数据。你需要将这些发生问题时的度量数据与当前数据进行比较。重现以前的时间能使现在的问题变得明朗。因此，持续的收集相关统计数据对于性能分析来说十分重要。在某些情况下，在解决收集统计数据这方面的问题上有自己内置的工具——statspack。尽管在某些情况下的作用非常大，但它缺乏解决性能问题所必须的健壮性。 提供了一个标志性的改进特性：自动工作量存储（Automatic Workload Repository AWR）。AWR 是随着数据库一起被安装的，它不仅能收集统计数据，还能从统计数据中分析出度量数据。

通过运行$ORACLE_HOME/rdbms/admin 目录下的 awrrpt.sql 脚本可以生产 AWR 从统计和度量数据中分析报告。这个分析报告最能体现出 AWR 的性能分析能力。这个脚本看起来很像 statspack，它会列出所有可用的 AWR 快照 并要求输入两个特定的快照编号作为一个间隔段。它能产生两种类型的输出：文本格式（除了 AWR 统计信息外和 statspack 报告基本类似）和默认的格式（通过超连接等方式来提供一个友好的界面）。下面来运行以下这个脚本， 对它产生的分析报告及 AWR 的性能分析能力做一个认识。

AWR 的使用

首先来了解一下 AWR 是如何设计的，并了解一下它的构造。基本上来说，AWR 应该是一个 Oracle 用来收集性能相关统计数据并从中得出性能度量数据来追踪潜在问题的内置工具。和 statspack 不一样，AWR 的快照信息是由一个新的后台进程 MMON 及其线程来每隔一个小时自动收集的。为了节省空间，这些收集的数据会在 7 天后自动清除。快照收集的频率和保留时间都是可以被用户修改的。可以通过以下脚本查看当前的设置：

```
SQL> select snap_interval, retention from dba_hist_wr_control;
SNAP_INTERVAL       RETENTION
------------------- -------------------
+00000 01:00:00.0   +00007 00:00:00.0
```

这个结果表明当前的快照是每隔一个小时收集一次，并且会被保留 7 天。要改变这个设置，比如需要设置成每隔半小时收集一次，并且只保留 3 天，可以使用以下语句（参数的单位都是分）：

```
SQL> begin
  2   dbms_workload_repository.modify_snapshot_settings (
  3     interval => 30,
  4     retention => 3*24*60
  5   );
  6 end;
SQL> select snap_interval, retention from dba_hist_wr_control;
SNAP_INTERVAL       RETENTION
------------------- -------------------
+00000 00:30:00.0   +00003 00:00:00.0
```

AWR 使用多个表来存储收集的统计数据，它们都被放置在 SYS 这个方案中，存储在一个新的特殊表空间 SYSAUX 上，并且以 WRM$_*和 WRH$_*的格式存储。前面一种格式的表存储诸如数据库检查和采集的快照等元数据信息，而后面一种格式的表存储了实际收集的统计数据（M 表示“Metadata 元数据”，H 表示“Historical 历史”）。另外还有多个以“DBA_HIST_”为前缀，由这些表构造出的视图。你可以利用这些视图写出自己的性能分析工具。这些视图的命名与它相关的表直接相关。比如，视图 DBA_HIST_SYSMETRIC_SUMMAY 是以表 WRH$_SYSMETRIC_SMMURY 为基础构造的。

AWR 的历史信息表捕捉了比 statspack 多得多的信息，包括表空间的使用情况、文件系统的使用情况、甚至操作系统的统计信息。可以用以下语句从数据字典中得到这些表的完整列表：

```
select view_name from user_views where view_name like 'DBA\_HIST\_%' escape '\';
```

视图 DBA_HIST_METRIC_NAME 定义了 AWR 收集的重要度量数据、它们所属哪个组以及他们实在哪个单元被收集的。下面是一条记录例子：

```
DBID              : 4133493568
GROUP_ID          : 2
GROUP_NAME        : System Metrics Long Duration
METRIC_ID         : 2075
METRIC_NAME       : CPU Usage Per Sec
METRIC_UNIT       : CentiSeconds Per Second
```

这个数据显示了一个隶属于 “系统长期度量数据”的度量数据组中的“每秒的厘秒数”单元的“CPU 每秒使用情况”的度量项。这条记录可以和其他表如“DBA_HIST_SYSMETRIC_SUMMARY”联合查询来起作用：

```
SQL> select begin_time, intsize, num_interval, minval, maxval, average,
standard_deviation sd
from dba_hist_sysmetric_summary where metric_id = 2075;

BEGIN  INTSIZE NUM_INTERVAL  MINVAL  MAXVAL  AVERAGE  SD
----- --------- ------------  ------- ------- --------  ----------
11:39    179916          30       0      33        3  9.81553548
11:09    180023          30      21      35       28  5.91543912
```

… …

这显示的是以厘秒为单位的 CPU 消耗时间。在分析数据中的标准偏差（Standard Deviation SD）可以帮助我们来确定平均数据图是否反映实际工组负荷。在第一条记录中，平均值是每秒钟 CPU 消耗 3 厘秒，而标准偏差是 9.81，这说明平均为 3 并没有反映出实际工作负荷。在第二条记录中，平均值是 28，而标准偏差是 5.9，所以这条记录更具代表性。这种类型的信息有助于帮助理解在性能度量中的几个环境参数的作用。

使用这些统计数据

上面我们了解了 AWR 是如何收集统计数据的，下面就来了解以下如何利用这些数据。大多数性能问题并非孤立的，但是也不要相信哪些可以找出问题的最终根源的流言。让我们来做一个典型的调优练习：你发现系统变得很慢，决定检查一下等待事件。通过检查发现，“缓存忙等待（buffer busy wait）”非常高。如何解决这个问题呢？存在几种可能：可能是索引的单一增长引起的；某张表饱和了，需要立即转载一个数据块到内存中；以及其他的原因引起的。在任何情况下，你都需要先定位出发生问题的段。如果它是一个索引段，你可以决定是否重建索引、把索引改为相反键索引、或者将索引转换为在 Oracle 10G 特有的哈希分区索引。如果是一张表，可以考虑修改表的存储参数使它密度降低，或者将它转移到一个自动段空间管理的表空间上去。你的这些计划、措施一般都是系统的，并且是基于你对各种事件的了解和你在处理这些问题所积累的经验。想向一下，如果这些事情是由机器驱动来完成——这个驱动能捕捉度量数据并且在基于预先定义的逻辑能演绎出可能的计划、措施，那么你的工作不是能变得很轻松吗？

现在 Oracle 10G 就提供了这样的驱动，它就是自动数据库诊断监视器（Automatic Database Diagnostic Monitor ADDM）。ADDM 使用 AWR 收集的数据来达到那样的效果。在上述的例子中，ADDM 能够发现发生了 buffer busy waits，找出适当的数据来检查在哪个段上面发生的，计算出它的本来数据和混合数据，并最终为 DBA 提供解决办法。每当 AWR 收集了一个快照数据，ADDM 就会检查这些度量数据，并产生出相应建议。因此，你就拥有了一个全天候工作的机器人 DBA 来为你分析数据、提前为你给出建议，让你由更多的时间来关注战略问题。通过使用新的 10G 企业管理器平台——DB Home——可以查看 ADDM 的建议和 AWR 存储的数据。你可以从管理界面的导航器上查看 AWR 的报告、负荷数据以及快照信息。将来可以安装更多组件来在更多细节上来检查 ADDM。

你还可以指定在特定条件下产生告警信息。这些告警，如服务器产生的告警（Server Generated Alerts），会被推入一个高级队列。在任意一个客户端上可以监听这个队列。一种客户端就是 10G 企业管理器，在上面可以显著的显示出告警信息。

时间模型

当你遇到一个性能问题时，首先想起降低哪个响应时间呢？你当然希望能消除或降低引起问题的最终因素的时间。但是你怎么才能知道时间被消耗在哪呢——不是等待，而是实际的工作时间？

Oracle 10G 介绍了使用时间模型来通过不同途径定位时间消耗。整个系统的时间消耗被记录在视图 V$SYS_TIME_MODEL 中。下面是一个对这个视图查询的结果：

```
STAT_NAME                                    VALUE
---------------------------------------- ----------
DB time                                    58211645
DB CPU                                     54500000
background cpu time                       254490000
sequence load elapsed time                       0
parse time elapsed                         1867816
hard parse elapsed time                    1758922
sql execute elapsed time                  57632352
connection management call elapsed time     288819
failed parse elapsed time                    50794
hard parse (sharing criteria) elapsed time  220345
hard parse (bind mismatch) elapsed time       5040
PL/SQL execution elapsed time               197792
inbound PL/SQL rpc elapsed time                  0
PL/SQL compilation elapsed time             593992
Java execution elapsed time                      0
bind/define call elapsed time                    0
```

注意 DB Time 这个统计项，它表明了自从实例启动后数据库消耗的时间。重新运行查询这个视图的语句，数据库消耗时间的数据将和之前不同。通过一轮调优，再作同样的分析，可以看出调优后的 DB Time 的改变，通过和第一的数据比较发生的变化，可以检查调优对于数据库时间产生的影响。除了数据库时间，视图 V$SYS_TIME_MODEL 还能显示其他很多统计数据，如消耗在不同类型的语句分析（Parsing）上的时间，甚至 PL/SQL 的编译时间。

这个视图显示的整个系统的时间模型。但是你可能对更细粒度上的视图感兴趣：会话级的时间模型。视图 V$SESS_TIME_MODEL 能显示在会话级捕捉到的时间统计数据。这些数据统计的是当前的会话，包括所有的活动的和不活动的都可见。多出的字段 SID 表明了所统计的会话的 SID。

在 Oracle 的以前版本，这些数据根本无法获取到，需要用户从各种其他数据中猜测出来。在 Oracle 10g 中，获取这些数据简直是小菜一碟。

活动会话历史

视图 V$SESSION 在 oracle 10G 中被增强了，其中最有价值的一项增强就是包括了等待时间和他们持续的时间，而不需要通过 V$SESSION_WAIT 来查看了。然而，因为这个视图很少反映出真实的时间值，所以一些重要信息就丢失了。例如，如果你从这个视图中查询看是否存在某个会话在等待某个非空闲的事件，而如果这个事件的数据存在问题，你将无法找到你想要的东西，因为等待事件必须依赖于你所查询到的时间数据。Oracle 10G 的另一个特性活动会话历史（Active Session History ASH）和 AWR 类似，将会话的性能统计数据存储在一个缓存中以便于将来的分析。但是，和 AWR 不一样的是，这些数据的存储并非永久的存储在表当中，而是存在内存当中，可以通过视图 V$ACTIVE_SESSION_HISTORY 来查到。这些数据每秒中被收集一次，并且只有哪些活动的会话才会被收集。随着时间的推进，老的数据被移出、新的数据被收集到内存，如此循环。为了找到有多少会话在等待某些事件，可以使用以下脚本：

```
select session_id||','||session_serial# SID, n.name, wait_time,
time_waited
from v$active_session_history a, v$event_name n
where n.event# = a.event#
```

这条语句的执行结果可以显示出事件的名称和花费了多少事件等待。增加 ASH 的字段可以帮你对某个特定的等待事件进行深入定位。例如，如果这些会话等待的事件中有一个是 buffer busy wait，那就必须再做适当的诊断来定位是在哪个段上发生了等待事件。你可以通过将 ASH 视图中的 CURRENT_OBJ#字段与 DBA_OBJECTS 连接查询来查到发生问题的段。

ASH 还记录了并行查询服务的会话信息，这些信息对于诊断并行查询的等待事件很有用。如果记录的是并行查询的从进程的信息，协同服务会话的 SID 会被表示在 QC_SESSION_ID 字段中。字段 SQL_ID 记录了产生等待事件的 SQL 语句的 ID，通过将它与视图 V$SQL 联合查询，可以找出这个令人讨厌的 SQL 语句。CLIENT_ID 可以使在如 web 应用这样的共享用户环境中更容易定位是哪个客户端，这个字段可以通过存储过程 DBMS_SESSION.SET_IDENTIFIER 来设置。

既然 ASH 的信息如此有价值，那么不是把它的数据像 AWR 一样永久的存储起来不是更好吗？MMON 进程会将这些信息存储到磁盘以服务于 AWR 表，并且可以通过视图 DBA_HIST_ACTIVE_SESS_HISTORY 来查询。

手工收集

在默认情况下，这些快照信息是自动收集的。但是你也可以随时手工收集。所有的 AWR 的功能都可以通过包 DBMS_WORKLOAD_REPOSITORY 来实现。要产生一个快照，只要执行：exec dbms_workload_repository.create_snapshot 就可以了。

这样会立即产生一个快照记录在表 WRM$_SNAPSHOT 中，并且在典型（TYPICAL）级别上收集度量数据。如果需要收集更细的统计数据，可以在上述存储过程执行时设定参数 FLUSH_LEVEL 为 ALL。统计数据的删除也是自动的，但也可以通过执行存储过程 drop_snapshot_range()来手工删除。

基准线

一个典型的性能调优过程时从收集一个度量数据集的基准线开始，然后调整，再收集另一个基准线集。通过比较这两个基准数据集来观察调整产生的效果。在 AWR 中，可以通过已经存在的多个快照做处理来进行类似的推理。假设一个名叫 apply_interest 的显著产生性能压力的进程在下午 1：00 到 3：00 之间运行，相应的快照 ID 是从 56 到 59，我们可以用以下存储过程为这些快照定一个名叫 apply_interest_1 的基准线：

```
exec dbms_workload_repository.create_baseline(56,59,'apply_interest_1');
```

这一操作标识了从 56 到 59 的快照作为名为“apply_interest_1”的基准线的一部分。

用以下语句检查已经存在的基准线：

```
SQL> select * from dba_hist_baseline;
DBID BASELINE_ID BASELINE_NAME  START_SNAP_ID END_SNAP_ID
---------- ----------- -------------------- ------------ -----------
4133493568           1 apply_interest_1               56          59
```

经过一些调优步骤，我们再创建另外一个基准线，名叫“apply_interest_2”，并比较与仅与两个基准线相关的快照的度量数据。像这样将一些快照独立成这样一些基准线能帮助了解性能调优产生的效果。分析完毕后，可以通过调用存储过程 drop_baselin()来删除这些基准线，而快照还是会被保留。同样的，当清除规则需要清除掉旧的快照信息时，与这些旧快照信息相关的基准线不会被清除，以便于以后的进一步深入分析。

4. SQL*PLUS 的改进

在 Oracle 10G 中，SQL Plus 这一小小而又强大的 DBA 工具已经得到了显著的改进，包括十分有用的提示和高级文件操作。

SQLPlus 的改进

哪个工具是 DBA 每天最常用的？除了很多 DBA 使用图形界面的工具外，使用最多的就是 SQL Plus 这个命令行方式的工具了。

尽管在 oracle 10G 中由于企业管理器的大大增强可能使 SQL Plus 的地位有所改变，但这个普遍使用的小工具仍然被保留下来了——它对于初学者和有经验的 DBA 都适用。我们下面将会探讨一下 SQL*Plus 10.1.0.2 中一些有用的新特性。但是你必须要有随 Oracle 10G 软件安装的 SQL Plus，而不能使用 9i 客户端连接 10G 服务器的 9i 的 SQL Plus。

提示

我当前是以哪个用户登录的？当前身份是什么？在长时间使用 9i 的 sqlplus 后，很多 DBA 会犯这种“迷糊”。你需要通过查询语句来解决这些迷惑。在 10g 的 sqlplus 中，你可以通过设置在 SQL>提示标识中增加这些信息了：

SQL> set sqlprompt "_user _privilege> "

通过以上设置，你的 sqlplus 提示标识就成了以下格式了：

SYS AS SYSDBA>

这就表明当前的登录帐号是 SYS，身份是 SYSDBA。请注意在上面语句中使用了两个特殊的预定义道德变量：_user 和_privilege，分别定义了当前用户和这个用户的登录
身份（权限）。现在我们再增加一些其他内容。比如我除了想知道以上信息外，还想知道当前的登录时间：

SQL> set sqlprompt "_user _privilege on _date>"

SYS AS SYSDBA on 31-8 月 -05>

再增加一个连接信息看，这样就可以指出你登录的是哪个服务器的：

SQL>set sqlprompt "_user on _date at _connect_identifier>"

SYS on 31-8 月 -05 at teng>

但是前面的当前时间信息只有日期，如果需要精确到当前的确切时间呢：

SYS on 31-8 月 -05 at teng>alter session set nls_date_format='yyyy-mm-dd hh24:mi:ss';

会话已更改。

SYS on 2005-08-31 16:20:15 at teng>

你只要将上面的内容存在$ORACLE_HOME/sqlplus/glogin.sql 中，那每次登录的提示信息都是你所需要的了。

不再需要双引号

自从 internal 的登录方式在 9i 当中不再被支持后，很多 DBA 就抱怨：如何可以不输入 sys 用户的密码来维护系统的安全？答案就是在操作系统提示符下用双引号的方法进入：

$sqlplus "/as sysdba"

需要多输一对双引号也引起了不少抱怨。而在 10g 当中你就无需再输入双引号了：

$sqlplus / as sysdba

增强的文件管理性

在 9i 中，当你在 sqlplus 进行操作时，你会希望将一些有用的语句存下来以便以后使用，这时你可以使用 save 命令。Save 命令会将从最后一次执行 save 开始以后的所有脚本存下来，这样就会将你执行过的脚本存成分散的文件。如：

SQL> select 1 ...
SQL> save 1
Created file 1.sql
SQL> select 2 ...
SQL> save 2
Created file 2.sql
SQL> select 3 ...
SQL> save 3
Created file 3.sql

这样存下来 1.sql、2.sql、3.slq 三个脚本，三个脚本的内容分别为 select 1 …、select 2 …、select 3 …。这样的话你就需要手工将他们再合成一个脚本以便以后使用。

在 sqlplus 10.1.0.2 中，就不需要这么费劲了，你可以用添加的方式存储：

SQL> select 1 ...
SQL> save myscript
Created file myscript.sql
SQL> select 2 ...
SQL> save myscript append
Append file myscript sql
SQL> select 3 ...
SQL> save myscript append
Append file myscript.sql

这样，所有的脚本都被存储在 myscript.sql 这个脚本中了。

这一特性对于 spool 同样适用。在 9i 中，输入 spool res 后，如果当前目录下不存在 res.lst 这个文件，就会创建它，如果已经存在，就会直接覆盖（不会给任何提示），然后将以后直到 spool off 的所有 sqlplus 上的信息存储在 res.lst 中。这样的话，如果原先有一个很重要的 res.lst 文件可能就无法恢复了。

在 10g 中，spool 命令可以向已经存在的文件中添加内容：

SQL> spool res append

如果忽略掉 append 子句就和 9i 中一样，会将已有文件覆盖，或者将 append 换为 replace 也会覆盖。而如果使用 create 子句就会先检查文件是否存在，如果存在，就会报错：

SYS on 2005-08-31 17:25:46 at teng>spool abc create
SP2-0771: 文件 "abc.LST" 已存在。
请使用其它名称或 "SPOOL filename[.ext] REPLACE"

Login.sql

每次登录 sqlplus 时，会先执行$ORACLE_HOME/sqlplus/glogin.sql 或者当前目录的 login.sql 脚本。但是，会存在各种各样的限制。在 9i 或以下版本里，假如你的脚本里有如下内容：

set sqlprompt "_connect_identifier >"

当第一次启动并连接到数据库 DB1 时，提示信息为：

DB1>

然后再连接到另外一个数据库上：

DB1> connect scott/tiger@db2
connected
DB1>

尽管连到了 DB2，但提示信息还是 DB1。说明这个提示有问题。其实原因很简单，login.sql 只是在第一次启动 sqlplus 时执行，而在每次重新连接时不会执行。所以提示信息没有变。

在 10g 中，这种限制没有了。脚本不仅在启动 sqlplus 时执行，还会在每次连接数据库时也会执行：

DB1> connect scott/tiger@db2
connected
DB2>

这样信息就是正确了。

使用原先版本的 sqlplus

如果你出于某些原因不想使用这种增强过的 sqlplus 该怎么办呢？很简单，只要在运行 sqlplus 时加上-c 的选项就可以了：

sqlplus –c 9.2

这样 sqlplus 的环境就和 9.2 版本的是一样的了。

轻松使用 dual

有多少人经常使用例如以下的语句：

select USER into <some variable> from DUAL

也许有很多。每次调用 DUAL 都会产生逻辑 IO，而这些逻辑 IO 都可以被数据库避免的。在一些情况下，调用 DUAL 是为了能达到像“<somevariable> := USER”这样的效果。因为 Oracle 代码将 DUAL 看作一张特殊的表，因此一些用于优化普通表的方法可能对它无效。

在 10g 中，这些担心都不存在了。因为 DUAL 是一张特殊的表，通过设定 10046 跟踪事件，”consistent gets”在相当程度上被降低了，查询计划也不一样了。

在 9i 中：

```
Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: CHOOSE
      1   TABLE ACCESS (FULL) OF 'DUAL'
```

在 10G 中：

```
Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   MODE: ALL_ROWS
      0   FAST DUAL
```

请注意 10g 中这个新的查询计划：“FAST DUAL”，与 9i 中的“TABLE ACCESS (FULL) OF 'DUAL'”完全相反。这一改进显著的降低了“consistent reads”，使频繁用到了 DUAL 表的应用效率更高。

5. RMAN

RMAN 增量备份方案、增量备份的离线恢复、恢复预览、从 resetlogs 中恢复、文件压缩等被重新设计后变得更加强大了。

大多数人都赞同 RMAN 就是 Oracle 事实上的数据库备份工具。尽管早期版本的 RMAN 已经很强大，但是人们对它的期待还是有很多。很多 DBA 对于一些很希望有但实际上没有的特性很烦恼。很幸运，在 10g 中解决了很多问题并且增加了很多受期待的特性，下面就一起看一下。

增量备份

RMAN 有一项增量备份的功能。但实际上你是否经常用它呢？或许偶尔，或许从来没有。

这项功能使 RMAN 备份上一次同级别或者更低级别的增量备份以后发生变化的数据块。例如，在第一天执行了一次全备份（level_0），在第二、三天执行了两次增量备份（level_1）。后面两次备份仅仅备份在第一天和第二天之间变化的数据块、第二天和第三天之间变化的数据块，而不是备份整个数据。这种策略降低了备份数据大小，只需要较少的空间，并且使备份窗口变得更小，降低了网络传输数量。使用增量备份的最重要的因素为了和数据仓库环境相关联。因为在数据仓库中，很多操作都是在 NOLOGGING 模式下进行的，并且数据的变化并没有记录在归档日志文件中，因此，没有可用来恢复数据的媒质了。由于如今数据仓库非常盘大，所以根本不会考虑使用全备份，同时也不可行。因而采用增量备份是一个可选的方法。

但为什么那么多 DBA 很少采用增量备份呢？一个原因就是在 Oracle 9i 和更低版本中，RMAN 会扫描所有数据块以定位哪些块需要被备份。这一操作给系统造成了很大的压力，因此增量备份不具备操作性。

Oracle 10G 的 RMAN 对增量备份的方式进行了改进。它利用一个和文件系统中日志文件类似的文件，来跟踪从上次备份以来发生变化的数据块。RMAN 需要读这个文件决定哪些块需要备份。

你可以通过执行以下命令来激活这种跟踪机制：

SQL> alter database enable block change tracking using file '/rman_bkups/change.log';

可以通过以下查询语句确定当前跟踪机制是否被激活：

SQL> select filename, status from v$block_change_tracking;

闪动恢复区域

在 9i 中的闪回功能依赖于回归表空间闪回到一个早期状态，这样就限制它闪回到很早的的状态。通过创建闪回日志，闪动恢复提供了一个新的解决方法。闪回日志和重做日志类似，使数据库恢复到一个早期状态。总之，你可以通过以下 SQL

语句为数据库创建一个闪动恢复区域，指定它的大小，并将数据库设置为闪动恢复模式：

SQL> alter system set db_recovery_file_dest = '/ora_flash_area';
SQL> alter system set db_recovery_file_dest_size = 2g;
SQL> alter system set db_flashback_retention_target = 1440;
SQL> alter database flashback on;

为了使闪回功能激活，数据库必须在归档日志模式。上述操作会在目录/ora_flash_area 下创建 oracle 管理文件（Oracle Managed Files OMF），总的大小使 2GB。数据库的变化都会记录在这些文件中，可以使数据库迅速恢复到以前的某一点。

默认情况下，RMAN 也会使用/ora_flash_area 目录来存储备份文件。因此，RMAN 的备份全市存储在磁盘上，而不是磁带上。这样的话，你就可以设定备份数据保留多少天，时间到了后，如果需要更多空间时这些文件会被自动删除。

然而，闪动恢复区域可以不需要一个文件系统或目录，它可以是一个自动存储管理（Automatic Storage Management ASM）磁盘组。在这种情况下，闪动恢复区域可以用以下语句指定：

SQL> alter system set db_recovery_file_dest = '+dskgrp1';

通过 ASM 和 RMAN 的结合使用，你可以通过使用哪些如 Serial ATA 和 SCSI 盘等廉价的磁盘来构建可扩展的、容错性强的存储系统。这种方式不能是备份过程更快，而可以使用比磁带方式更便宜的磁盘来完成同样的事情。

另外一个好处就是避免了用户错误。永伟 ASM 文件不是实际的文件系统，他们被 DBA 和系统管理员损坏的几率更小。

增量合并

假如你有以下的备份计划：星期天做 level 0 的完全备份，标识为 level_0；星期一做 level 1 的增量备份，标识为 level_1_mon；星期四做 level 1 的增量备份，标识为 level_1_tue。如果数据库在星期六被损坏了，在 10G 之前你不得不恢复 level_0 然后再将所有 6 个增量备份实施上去，这样会消耗很长一段时间。这也是很多 dba 避免使用增量备份的原因之一。

Oracle 10g 的 RMAN 从根本上改变了这种方式，现在的增量备份命令如以下这个样子：

RMAN> backup incremental level_1 for recover of copy with tag level_0 database;

这样 RMAN 再做增量备份 level_1 备份时会和标识为 level_0 的完全备份合并。经过这样的备份，level_0 变成了那天的完全备份了。

因此，在周四，标识为 level_0 的备份实际与 level_1 的增量备份合并，成了在周四做的完全备份。如果在周六数据库损坏了，你只需要将 level_0 的备份加上一些归档日志共同恢复就可以了。而不需要将增量备份也恢复。这种方式大大减少了恢复时间，使备份加速，并且避免了重新做一个增量备份。

压缩文件

在基于磁盘备份的闪动恢复区域功能中，你还有一个很大的限制：磁盘容量。特别使当通过网络实现时——实际也经常是这么用的——强烈建议创建一个尽可能小的备份。在 10G 的 RMAN 中，你可以在备份命令中插入压缩文件的命令：

RMAN> backup as compressed backupset incremental level 1 database;

请注意这使用了 COMPRESS 子句。它压缩的备份文件有一个很重要的特点：当恢复时，RMAN 可以无需解压文件直接读取它。为了确认是否压缩，可以在输出信息中检测是否有以下内容：

channel ORA_DISK_1: starting compressed incremental level 1 datafile backupset

你还可以通过在 RMAN 中 list output 确认备份是否被压缩：

RMAN> list output;

BS Key Type LV Size Device Type Elapsed Time Completion Time
------- ---- -- ---------- ----------- ------------ ---------------
3 Incr 1 2M DISK 00:00:00 26-FEB-04
BP Key: 3 Status: AVAILABLE Compressed: YES Tag: TAG20040226T100154
Piece Name: /ora_flash_area/SMILEY10/backupset/2004_02_26/o1_mf_ncsn1_TAG20040226T100154_03w2m3lr_.bkp
Controlfile Included: Ckp SCN: 318556 Ckp time: 26-FEB-04
SPFILE Included: Modification time: 26-FEB-04

就如所有的压缩动作一样，这一方法会增大 CPU 的压力。但这也使你可以保留更多的备份在磁盘上以备恢复。另外，你还可以用 RMAN 来备份物理备份数据库以用于恢复主数据库。这一方法可以将备份资源从其他主机上卸载下来。

恢复预览

通过提供了能预览恢复操作功能，Oracle 10g 变得很先进了：

RMAN> restore database preview;
… ...

你还可以预览特定的恢复操作，如：

RMAN>restore tablespace users preview;
… ...

预览功能使你能通过定期的检查来确认恢复时要做什么样的准备。

Resetlogs 和恢复

假如你丢失了当前的在线重做日志文件又不得不做一次不完全的数据库恢复。最大的问题时 resetlogs。当不完全恢复后，你必须使用 resetlogs 子句来打开数据，

它会设置日志线程的序列号为 1，删除 RMAN 中早期的备份，使恢复操作更容易。在 Oracle 9i 和更低版本中，如果你需要将数据库从 resetlogs 中恢复到一个早期状态，你不得不把它恢复成一个不同的样子。在 Oracle 10G 中，你就不需要这样做了。由于控制文件增加了一些结构，RMAN 可以在一次 resetlogs 操作之前或之后随时利用所有的备份来恢复数据库。做备份使没有必要关闭数据库了。这一新功能意味着在一次 resetlogs 操作以后数据库可以迅速的被用户打开。

6. 增强的 CONNECT BY 子句

为了更好的查询一个树状结构的表，在 Oracle 的 PL/SQL 中提供乐一个诱人的特性——CONNECT BY 子句。它大大的方便了我们查找树状表：遍历一棵树、寻找某个分支……，但还是存在一些不足。在 Oracle 10G，就对这个特性做了增强。下面就举例说明一下：

CONNECT_BY_ROOT

一张表，有多颗子树（根节点为 0），现在我想知道每个节点属于哪个子树。举例：铃音目录结构下有多个大分类：中外名曲、流行经典、浪漫舞曲……，每个大类下面又有多个子类，子类下面还可以细分。那现在想要知道每个子类分属哪个大类，或者要统计每个大类下面有多少个子类。

看下面的例子，DIRINDEX 分别为 1、2、3 的就是大分类，其他编号的都是子类或孙子类：

```
select dirindex, fatherindex, RPAD(' ', 2*(LEVEL-1)) || dirname from
t_tonedirlib
start with fatherindex = 0
connect by  fatherindex =  prior dirindex
```

DIRINDEX	FATHERINDEX	DIRNAME
1	0	中文经典
52	1	kkkkkkk
70	52	222
58	52	sixx
59	52	seven
69	52	uiouoooo
55	52	four
7	1	流行风云
8	1	影视金曲
1111	8	aaa
1112	8	bbb
1113	8	ccc
9	1	古典音乐
81	1	小熊之家
104	81	龙珠
105	81	snoppy
101	81	叮当 1
102	81	龙猫
103	81	叮当 2
2	0	热门流行
31	2	有奖活动
32	2	相约香格里拉
50	2	新浪彩铃
3	0	老歌回放
333	3	老电影
335	3	怀旧金曲

26 rows selected

如何统计 1、2、3 三个大类下有哪些子类，有多少个子类？在 9i 及以前要做这样的统计十分麻烦。现在 10G 提供了一个新特性：CONNECT_BY_ROOT， 他的作用就是使结果不是当前的节

点 ID，而满足查询条件下的根节点的 ID。以上面为例，我们需要得到以上结果只需要执行以下语句就可以搞定了：

```
select CONNECT_BY_ROOT dirindex, fatherindex, RPAD(' ', 2*(LEVEL-1)) ||
dirname from t_tonedirlib
start with fatherindex = 0
connect by  fatherindex =  prior dirindex
CONNECT_BY_ROOTDIRINDEX   FATHERINDEX RPAD('',2*(LEVEL-1))||DIRNAME
----------------------- ------------- ----------------------------
                      1                     0 中文经典
                      1                     1   kkkkkkk
                      1                    52     222
                      1                    52     sixx
                      1                    52     seven
                      1                    52     uiouoooo
                      1                    52     four
                      1                     1   流行风云
                      1                     1   影视金曲
                      1                     8     aaa
                      1                     8     bbb
                      1                     8     ccc
                      1                     1   古典音乐
                      1                     1   小熊之家
                      1                    81     龙珠
                      1                    81     snoppy
                      1                    81     叮当 1
                      1                    81     龙猫
                      1                    81     叮当 2
                      2                     0 热门流行
                      2                     2   有奖活动
                      2                     2   相约香格里拉
                      2                     2   新浪彩铃
                      3                     0 老歌回放
                      3                     3   老电影
                      3                     3   怀旧金曲
26 rows selected
```

查出来的结果中，CONNECT_BY_ROOTDIRINDEX 就是各个子类（孙子类）所属的大类编号，如果需要统计，就只要执行以下语句马上可以统计出来了：

```
select rootindex, count('X') from
      (select CONNECT_BY_ROOT dirindex as rootindex
      from t_tonedirlib
      start with fatherindex = 0
      connect by  fatherindex =  prior dirindex) a
group by a.rootindex
ROOTINDEX COUNT('X')
--------- ----------
        1         19
        2          4
        3          3
3 rows selected
```

CONNECT_BY_ISLEAF

经常有 DBA 因为要查找树状表中的叶子节点而苦恼。大部分 DBA 为了解决这个问题就给表增加了一个字段来描述这 个节点是否为叶子节点。但这样做有很大的弊端：需要通代码逻辑来保证这个字段的正确性。

Oracle10G 中提供了一个新特性——CONNECT_BY_ISLEAF——来解决这个问题了。简单点说，这个属性结果表明当前节点在满足条件的查询结果中是否为叶子节点，0 不是，1 是：

```
select CONNECT_BY_ISLEAF, dirindex, fatherindex, RPAD(' ', 2*(LEVEL-1))
|| dirname
from t_tonedirlib
start with fatherindex = 0
connect by  fatherindex =  prior dirindex
 CONNECT_BY_ISLEAF DIRINDEX FATHERINDEX RPAD(' ',2*(LEVEL-1))||dirname
------------------ ---------- ----------- --------------------------------
                 0          1           0 中文经典
                 0         52           1   kkkkkkk
                 1         70          52     222
                 1         58          52     sixx
                 1         59          52     seven
                 1         69          52
uiouoooo
                 1         55          52     four
                 1          7           1   流行风云
                 0          8           1   影视金曲
                 1       1111           8     aaa
                 1       1112           8     bbb
                 1       1113           8     ccc
                 1          9           1   古典音乐
                 0         81           1   小熊之家
                 1        104          81     龙珠
                 1        105          81
snoppy
                 1        101          81     叮当 1
                 1        102          81     龙猫
                 1        103          81     叮当 2
                 0          2           0 热门流行
                 1         31           2   有奖活动
                 1         32           2   相约香格
里拉
                 1         50           2   新浪彩铃
                 0          3           0 老歌回放
                 1        333           3   老电影
                 1        335           3   怀旧金曲
26 rows selected
```

一看结果，清晰明了！

CONNECT_BY_ISCYCLE

我们的树状属性一般都是在一条记录中记录一个当前节点的 ID 和这个节点的父 ID 来实现。但是，一旦数据中出现了循环记录，如两个节点互为对方父节点，系统就会报 ORA-01436 错误：

```
insert into t_tonedirlib(dirindex, fatherindex, dirname, status) values
(666, 667, '123', 5);
1 row inserted
insert into t_tonedirlib(dirindex, fatherindex, dirname, status) values
(667, 666, '456', 5);
 1 row inserted

select dirindex, fatherindex, RPAD(' ', 2*(LEVEL-1)) || dirname from
t_tonedirlib
start with fatherindex = 666
connect by  fatherindex =  prior dirindex
 ORA-01436: 用户数据中的 CONNECT BY 循环
```

10G 中，可以通过加上 NOCYCLE 关键字避免报错。并且通过 CONNECT_BY_ISCYCLE 属性就知道哪些节点产生了循环：

```
select CONNECT_BY_ISCYCLE, dirindex, fatherindex, RPAD(' ', 2*(LEVEL-1))
|| dirname
from t_tonedirlib
start with fatherindex = 666
connect by NOCYCLE fatherindex =  prior dirindex
CONNECT_BY_ISCYCLE DIRINDEX FATHERINDEX RPAD(' ',2*(LEVEL-1))||dirname
------------------ ---------- ---------- --------------------------------
                 0        667        666 456
                 1        666        667   123
2 rows selected
```

以上就是在 10G 中增强的 CONNECT BY 了。当然对于这些增强特性的作用肯定不止如上介绍的，还需要更多高人去挖掘了。

7. 闪回表

删除表的恢复

如果某个用户不小心删除了一个十分重要的表，后果将非常严重。在 9i 中提供的闪回特性只能恢复 DML 语句造成的影响，而无法恢复 DDL 语句的影响。DBA 只能通过重建一张表，然后从备份数据中导入。

利用 Oracle 10G 中的闪回表的特性，DBA 可以轻松完成这项工作，并将影响降到最小。下面就举一个例子说明：

- 创建表：

```
SQL> create table abc (f number(9));
表已创建。
SQL> create index idx_test on abc(f);
索引已创建。
SQL> insert into abc values(1);
已创建 1 行。
SQL> insert into abc values(2);
已创建 1 行。
SQL> insert into abc values(3);
已创建 1 行。
SQL> select * from tab;
TNAME                          TABTYPE        CLUSTERID
------------------             -------------- ----------
ABC                            TABLE
SQL> select index_name, index_type, table_name from ind;
INDEX_NAME                     INDEX_TYPE   TABLE_NAME
-------------------            -------------------  ----------------------
IDX_TEST                       NORMAL       ABC
```

- 删除表：

```
SQL> drop table abc;
表已删除。
SQL> select * from tab;
TNAME                          TABTYPE        CLUSTERID
------------------             -------------- ----------
BIN$XXUGsbYvSqa8Mrd6GstP+g==$0      TABLE
```

请注意，在原表 abc 被删除后，abc 没有了，却出现了一张新表 BIN$XXUGsbYvSqa8Mrd6GstP+g==$0。这就是 Oracle 10G 中对删除表的处理，原表实际上并没有完全删除掉，而是被系统重新命名成了一个系统定的新表。它还存在于原先的表空间，并且保持了原有的结构。

依赖于原表的存储过程都失效了。而建在表上的索引和触发器也会被重新命名。

```
SQL> select index_name, index_type, table_name from ind;
```

INDEX_NAME INDEX_TYPE TABLE_NAME
---------- ---------- ----------
BIN$1++ilvsQQ7mfPh2pvont5A==$0
NORMAL BIN$XXUGsbYvSqa8Mrd6GstP+g==$0

被删除的表及其相关对象都会被放置在一个称为 recyclebin 的逻辑容器中：

SQL> show recyclebin
ORIGINAL NAME RECYCLEBIN NAME OBJECT TYPE DROP TIME
---------- ---------- ---------- ----------
ABC BIN$XXUGsbYvSqa8Mrd6GstP+g==$0 TABLE 2005-08-29:18:03:10

显示了被删除对象的原有名字，删除后的名字，对象类型以及删除时间。

通过使用 flashback table 语句就可以恢复表！

SQL> flashback table abc to before drop;
闪回完成。
*SQL> select * from tab;*
TNAME TABTYPE CLUSTERID
---------- ---------- ----------
ABC TABLE

这样就轻松的将对象恢复了！

注意，对象被恢复后，它在 recyclebin 中占用的空间并不会被释放。必须使用 PURGERECYCLEBIN 来清空占用的空间。

SQL> PURGE RECYCLEBIN;
回收站已清空。

当然，如果想要彻底删除一个对象，让它不占用回收站的空间，可以用以下语句实现：

SQL> DROP TABLE ABC PURGE;

管理回收站

一旦哪些没有被真正删除的对象占满了表空间将会怎样呢？其实答案很简单：如果表空间被回收站中的数据占满了，并且数据文件也无法再扩展了（即产生了表空间压力）。那么回收站中的对象将会以“先入先出”（FIFO）的原则被自动清除掉。并且，依赖表的对象（如索引）将会比表对象先清除。

同样的，当存在用户配额时也会发生表空间压力的情况。当一个用户的配额空间被占满了，尽管此时表空间还可能有足够的空间，系统也会以 FIFO 的原子释放回收站中属于这个用户的对象。

另外，还有多种途径来手工控制回收站。比如可以用对象的原有名字从回收站中清除指定对象：

PURGE TABLE ABC

或者用对象被删除后系统自动重命名的名字来指定清除它：

PURGE TABLE “BIN$XXUGsbYvSqa8Mrd6GstP+g==$0”

清除表时，同时也会清除依赖这张表的约束，如索引。可以指定只清除表相应的约束，如：

PURGE INDEX IDX_TEST

此外，还可以将整个表空间的回收站内容全部清除：

PURGE TABLESPACE RING

也可以清除某个表空间上的回收站中某个用户的对象：

PURGE TABLESPACE RING USER TEST

当用某个普通用户登录时，只会清除它自己的对象。

PURGE RECYCLEBIN

当以 DBA 身份登录时，可以清除所有表空间上回收站

PURGE DBA_RECYCLEBIN

表的版本和闪回

一旦一张表被多次删除又重建，该如何恢复呢？

```
SQL> CREATE TABLE TEST (COL1 NUMBER);
表已创建。
SQL> INSERT INTO TEST VALUES (1);
已创建 1 行。
SQL> COMMIT;
提交完成。
SQL> DROP TABLE TEST;
表已删除。
SQL> CREATE TABLE TEST (COL1 NUMBER);
表已创建。
SQL> INSERT INTO TEST VALUES (2);
已创建 1 行。
SQL> COMMIT;
提交完成。
SQL> DROP TABLE TEST;
表已删除。
SQL> CREATE TABLE TEST (COL1 NUMBER);
表已创建。
SQL> INSERT INTO TEST VALUES (3);
已创建 1 行。
SQL> COMMIT;
提交完成。
SQL> DROP TABLE TEST;
表已删除。
```

这时，系统在每一次删除时都会在回收站中为这张表重命名一张表：

```
SQL> select * from tab;
TNAME                          TABTYPE        CLUSTERID
------------------             -------------- ----------
BIN$IE1x0mwfSe6p6yhLn8/mBw==$0       TABLE
BIN$SUj0n3ghTaSQu0AFGheUYA==$0     TABLE
BIN$khjCqP4fRqeOrE/Eg/XUWQ==$0       TABLE
```

```
SQL> show recyclebin
ORIGINAL NAME    RECYCLEBIN NAME  OBJECT TYPE  DROP TIME
---------------  ---------------------------  ------------  ---------------
TEST     BIN$IE1x0mwfSe6p6yhLn8/mBw==$0 TABLE       2005-08-29:20:44:47
TEST     BIN$khjCqP4fRqeOrE/Eg/XUWQ==$0 TABLE       2005-08-29:20:44:47
TEST     BIN$SUj0n3ghTaSQu0AFGheUYA==$0 TABLE       2005-08-29:20:44:46
```

这时，如果使用 flashback table test to before drop 就只会最后一次删除的表的状态，即字段 col1 的内容为 3。可以使用以下方式将表闪回并重命名：

```
SQL> flashback table test to before drop rename to test2;
闪回完成。
SQL> flashback table test to before drop rename to test3;
闪回完成。
SQL> select * from tab;
TNAME                          TABTYPE        CLUSTERID
------------------             -------------- ----------
TEST                           TABLE
TEST2                          TABLE
TEST3                          TABLE
```

这时的闪回是以删除的相反顺序闪回的。

所以可以直接指定某一次被删除的表：

```
SQL> flashback table “BIN$khjCqP4fRqeOrE/Eg/XUWQ==$0” to before drop
rename to test2;
闪回完成。
SQL> flashback table “BIN$SUj0n3ghTaSQu0AFGheUYA==$0” to before drop
rename to test3;
闪回完成。
```

需要注意的

表上的对象如索引、触发器在表被闪回后是不会被同时闪回的，而是保持了在回收站中名字。一些依赖这张表的代码对象如视图、存储过程在表被删除后会失效，在表被闪回后不会被自动重新编译，而要手工重新编译他们。

相关的信息被保存在视图 USER_RECYCLE 中。可以用以下语句来获得这些索引、触发器对象的原有名称：

```
SQL> SELECT OBJECT_NAME, ORIGINAL_NAME, TYPE
  2  FROM USER_RECYCLEBIN
  3  WHERE BASE_OBJECT = (SELECT BASE_OBJECT FROM
USER_RECYCLEBIN
  4  WHERE ORIGINAL_NAME = 'ABC')
  5  AND ORIGINAL_NAME != 'ABC';
OBJECT_NAME                    ORIGINAL_N TYPE
------------------------------ ---------- --------
BIN$1++ilvsQQ7mfPh2pvont5A==$0  IDX_TEST   INDEX
```

可以用以下方式来恢复索引：

SQL> ALTER INDEX " BIN$1++ilvsQQ7mfPh2pvont5A==$0" RENAME TO IDX_TEST;

一个例外就是位图索引被删除后是不会被保存在回收站中的，也无法从上述视图中查到，需要用其他方式来恢复。

闪回表的其他用途

闪回表功能并不仅仅用于恢复被删除的表。像闪回语句那样，闪回表可以闪回表在某一时间点的状态来完全取代现在的这张表。如下面语句将表闪回到 SCN 为 2202666520 的状态下：

SQL> FLASHBACK TABLE RECYCLETEST TO SCN 2202666520;

8. 表空间管理

设置默认表空间

DBA 们经常会遇到一个这样令人头疼的问题：不知道谁在 Oracle 上创建了一个用户，创建时，没有给这个用户指定默认表空间，所以这个用户就会采用默认的表空间——system 表空间。导致系统表空间迅速被用户数据占满，直至宕机。

在 10G 中，DBA 有办法避免这种问题了——在线指定系统默认表空间：

ALTER DATABASE DEFAULT TABLESPACE <tsname>;

通过执行以上命令，可以设定系统的默认表空间。这样的话，在创建新用户时，如果不指定他的默认表空间，就会采用上面指定的系统默认表空间作为这个用的默认表空间。

```
SQL>conn /as sysdba
SQL> create user test1 identified by test1 default tablespace ringidx;
用户已创建。
SQL> alter database default tablespace ring;
数据库已更改。
SQL> create user test identified by test;
用户已创建。
SQL> select username, default_tablespace defspace from dba_users where
username='TEST';
USERNAME                       DEFSPACE
------------------------------ ------------------------------
TEST                           RING
```

但是要注意的是，一旦将系统默认表空间修改了以后，原有的所有普通用户的默认表空间都会被指定为这个表空间，如上例中 test1，创建时指定了他的默认表空间为 ringidx，执行了’alter database default tablespace ring’后,他的默认表空间也被改为了 ring。

```
SQL> select username, default_tablespace defspace from dba_users where
username='TEST1';
USERNAME                       DEFSPACE
------------------------------ ------------------------------
TEST1                          RING
```

为非核心的系统用户指定一个特殊的默认表空间

在创建 Oracle 实例时，除了创建了如 sys、system 等系统核心的用户外，还会创建一些诸如 dbsnmp、odm、perfstat 等非核心的用户。这些用户在 9i 中都是以

system 作为他们的默认表空间。这些用户一旦被使用，也会产生较大的数据量占用 system 表空间。

在 Oracle10G 当中，使用了一个新的表空间 SYSAUX 作为这些用户的默认表空间。这个表空间在实例创建是就创建了，除了他的数据文件名可以被修改外，其他都不允许被修改。

Oracle 的这一改变可以使当 system 表空间损坏时对数据库做全库恢复。在 sysaux 中的对象可以恢复成普通对象，而数据库能保持正常运行。

如果 DBA 想要将 sysaux 表空间中的用户转移到其他表空间去该如何做呢。在 10G 中，专门为此提供一个视图 V$SYSAUX_OCCUPANTS 来描述如何转移这些用户的表空间。

```
select * from V$SYSAUX_OCCUPANTS where OCCUPANT_NAME = 'ODM'
OCCUPANT_NAME OCCUPANT_DESC SCHEMA_NAME MOVE_PROCEDURE MOVE_PROCEDURE_DESC  SPACE_USAGE_KBYTES
------------- ------------- ----------- -------------- -------------------  ------------------
ODM   Oracle Data Mining  DMSYS     MOVE_ODM    Move Procedure for Oracle Data Mining       5568

1 rows selected
```

如上，如果要将改变 ODM 的表空间，可以使用存储过程 MOVE_ODM，当前它占用了 5568kb 的表空间。

为表空间改名

这 Oracle10G 表空间增强中一个令人心动的改变。

这项功能允许改变数据库中除 system 和 sysaux 外任意一个表空间的名字。

```
ALTER TABLESPACE <oldname> RENAME TO <newname>;
```

有了这项功能，将会让很多事情变得非常简单。

有 DBA 可能会担心，一旦一个表空间的名字改变了，而且它已经被使用了很长时间了，会不会引起系统的混乱？这个不用担心，执行了上面的语句后，Oracle 会将系统中所有相关的数据字典的内容全部更新：

```
SQL> alter tablespace ring rename to ring1;
表空间已更改。
SQL> select username, default_tablespace defspace from dba_users where username='TEST';
USERNAME                       DEFSPACE
--------------                 ----------------
TEST                           RING1
```

9. 物化视图

Advisor

在 10g 将查询重写并且引进了新的强大的调优建议者使管理物化视图变得容易多了。

物化视图(Materialized Views MVs)，也被称为快照，现在已经被广泛应用了。MV 将一个查询的结果存储在一个段中，并且当用户提交查询时返回查询结果，而不需要重新执行查询——如果查询会被执行多次（经常出现在数据仓库环境中），这就会非常有效。MV 可以从基础表中完全刷新或通过使用快速刷新机制增量刷新。

假如你有如下定义的 MV：

```
create materialized view mv_hotel_resv
refresh fast
enable query rewrite
as
select distinct city, resv_id, cust_name
from hotels h, reservations r
where r.hotel_id = h.hotel_id';
```

你如何知道使这个 MV 正常工作的所有必须对象都已经被创建呢？在 10g 之前，这一检测是通过包 DBMS_MVIEW 的存储过程 EXPLAIN_MVIEW 和 EXPLAIN_REWRITE 实现的。这些存储过程在 10g 还存在，它们的功能很简单——检测 MV 是否具备快速刷新能力和查询重新能力，但它们并不提供如何使这些能力有效的建议。相反，要求对于每个 MV 的结构都做检查是不切实际的。

在 10g 中，有一个新的包 DBMS_ADVISOR，它有一个存储过程 TUNE_MVIEW 使这项工作变得非常容易：你可以在调用这个包时输入一个输入参数，参数内容为创建 MV 的整个脚本。这个存储过程创建了一个建议者任务(Advisor Task)，它的名字会通常存储过程唯一的输出参数返回给用户。

这有一个例子。由于第一个参数是一个输出参数，所以你必须定义一个变量：

```
SQL> -- first define a variable to hold the OUT parameter
SQL> var adv_name varchar2(20)
SQL>  begin
 2  dbms_advisor.tune_mview
 3    (
 4      :adv_name,
 5      'create materialized view mv_hotel_resv refresh fast enable query rewrite as
       select distinct city, resv_id, cust_name from hotels h,
         reservations r where r.hotel_id = h.hotel_id');
 6* end;
Now you can find out the name of the Advisor from the variable.
SQL> print adv_name
ADV_NAME
----------------------
```

```
TASK_117
```

接下来，可以通过一个新视图 DBA_TUNE_MVIEW 从 Advisor 那获取到所提供的建议。在执行查询前记得先执行设置 SET LONG 999999，因为这个视图中的这个字段是一个 CLOB 类型，而默认知会显示 80 个字符。

```
SQL> select script_type, statement
2   from   dba_tune_mview
3   where  task_name = 'TASK_117'
4   order  by script_type, action_id;
SCRIPT_TYPE    STATEMENT
-------------- ------------------------------------------------------------
IMPLEMENTATION CREATE MATERIALIZED VIEW LOG ON
"ARUP"."HOTELS" WITH ROWID,
               SEQUENCE ("HOTEL_ID","CITY")  INCLUDING NEW VALUES

IMPLEMENTATION ALTER MATERIALIZED VIEW LOG FORCE ON
"ARUP"."HOTELS" ADD
               ROWID, SEQUENCE ("HOTEL_ID","CITY")  INCLUDING NEW
VALUES

IMPLEMENTATION CREATE MATERIALIZED VIEW LOG ON
"ARUP"."RESERVATIONS" WITH
               ROWID, SEQUENCE ("RESV_ID","HOTEL_ID","CUST_NAME")
               INCLUDING NEW VALUES

IMPLEMENTATION ALTER MATERIALIZED VIEW LOG FORCE ON
"ARUP"."RESERVATIONS"
               ADD ROWID, SEQUENCE ("RESV_ID","HOTEL_ID","CUST_NAME")
               INCLUDING NEW VALUES

IMPLEMENTATION CREATE MATERIALIZED VIEW
ARUP.MV_HOTEL_RESV   REFRESH FAST
               WITH ROWID ENABLE QUERY REWRITE AS SELECT
              ARUP.RESERVATIONS.CUST_NAME C1,
ARUP.RESERVATIONS.RESV_ID
             C2, ARUP.HOTELS.CITY C3, COUNT(*) M1 FROM
ARUP.RESERVATIONS,
               ARUP.HOTELS WHERE ARUP.HOTELS.HOTEL_ID =
               ARUP.RESERVATIONS.HOTEL_ID GROUP BY
               ARUP.RESERVATIONS.CUST_NAME, ARUP.RESERVATIONS.RESV_ID,
               ARUP.HOTELS.CITY
UNDO           DROP MATERIALIZED VIEW ARUP.MV_HOTEL_RESV
```

字段 SCRIPT_TYPE 的内容就是建议。大多数行都是要被实施的，因此被命名成 IMPLEMENTATION。如果接受了这些建议，需要从字段 ACTION_ID 中得到一个特殊的序列号。

如果重新仔细检查一下这些自动产生的建议，你会发现它们和你自己分析得出需要做的操作很相似。这些建议是逻辑上的；如果存在快速刷新，那就需要通过包括这些新值的子句在基础表上建立物化视图日志(MATERIALIZED VIEW LOG)。STATEMENT 字段甚至提供了一个实施这些建议的准确的 SQL 语句。

在实施的最后步骤，Advisor 建议对 MV 的创建方式做一些修改。注意我们例子中的一个不同点：在 MV 上加了一个 count(*)。由于我们定义这个 MV 是快速刷新，而 count(*)又是必须的，所以 Advisor 修正了这一冗余。

存储过程 TUNE_MVIEW 与 EXPLAIN_MVIEW 和 EXPLAIN_REWRITE 的不同之处不仅仅在于建议，它还能更容易鉴别出并提供一个效率更好的方式建立相同的 MV。有时候 Advisor 能建议比使用一个单一的 MV 效率更高的的查询。

你可能会问，如果一个经验丰富的 DBA 能找出 MV 创建脚本中的却些并且能自己调整它，那这些有什么用？当然，Advisor 就是一个经验丰富、精力充沛、机器人似的的 DBA，它能给出和人差不多的建议。但是和人有一个很大的不同：它可以随时工作而不需要假期和涨薪。这一好处可以使有经验的 DBA 从日常任务中解放出来，把这些工作留给普通的 DBA 去做。而把它们自己的经验发挥到更具战略意义的任务中。

你也可以在嗲用 TUNE_MVIEW 时传入 Advisor 的名字，这样就不会使用系统自己生产的名字了。

更容易实施

既然你知道了这些建议，你当然希望去实施它们了。一个方法就是将字段 STATEMENT 中内容取出，存到一个脚本文件中，并执行它。另外一个方法就是执行一个包里面的存储过程：

```
begin
  dbms_advisor.create_file (
    dbms_advisor.get_task_script ('TASK_117'),
  'MVTUNE_OUTDIR',
  'mvtune_script.sql'
);
end;
/
```

这一存储过程是假定你已经定义了一个目录对象的情况下调用的，如：

```
create directory mvtune_outdir as '/home/oracle/mvtune_outdir';
```

调用包 dbms_advisor 的这个存储过程会在目录/home/oracle/mvtune_outdir 中生成一个名叫 mvtune_script.sql 的脚本文件。如果查看文件，它有如下内容：

```
Rem  SQL Access Advisor: Version 10.1.0.1 - Production
Rem
Rem  Username:        ARUP
Rem  Task:            TASK_117
Rem  Execution date:
Rem
```

```
set feedback 1
set linesize 80
set trimspool on
set tab off
set pagesize 60

whenever sqlerror CONTINUE

CREATE MATERIALIZED VIEW LOG ON
   "ARUP"."HOTELS"
   WITH ROWID, SEQUENCE("HOTEL_ID","CITY")
   INCLUDING NEW VALUES;

ALTER MATERIALIZED VIEW LOG FORCE ON
   "ARUP"."HOTELS"
   ADD ROWID, SEQUENCE("HOTEL_ID","CITY")
   INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW LOG ON
   "ARUP"."RESERVATIONS"
   WITH ROWID, SEQUENCE("RESV_ID","HOTEL_ID","CUST_NAME")
   INCLUDING NEW VALUES;

ALTER MATERIALIZED VIEW LOG FORCE ON
   "ARUP"."RESERVATIONS"
   ADD ROWID, SEQUENCE("RESV_ID","HOTEL_ID","CUST_NAME")
   INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW ARUP.MV_HOTEL_RESV
   REFRESH FAST WITH ROWID
   ENABLE QUERY REWRITE
   AS SELECT ARUP.RESERVATIONS.CUST_NAME C1,
RUP.RESERVATIONS.RESV_ID C2, ARUP.HOTELS.CITY
      C3, COUNT(*) M1 FROM ARUP.RESERVATIONS, ARUP.HOTELS
WHERE
RUP.HOTELS.HOTEL_ID
      = ARUP.RESERVATIONS.HOTEL_ID GROUP BY
RUP.RESERVATIONS.CUST_NAME, ARUP.RESERVATIONS.RESV_ID,
      ARUP.HOTELS.CITY;

whenever sqlerror EXIT SQL.SQLCODE

begin
  dbms_advisor.mark_recommendation('TASK_117',1,'IMPLEMENTED');
end;
```

/

这一文件包含了所有你需要实施的建议的内容，而不需要你手工去创建一个脚本。机器 DBA 又一次替你做了你需要做的工作。

重写还是报错

现在你可能已经认识到了查询重写特性是多么有用和重要。它能大大降低 I/O 和处理过程、返回结果更快。

还是假定以上的例子，用户执行一个下面的查询：

```
SQL> Select city, sum(actual_rate)
2    from hotels h, reservations r, trans t
3    where t.resv_id = r.resv_id
4    and h.hotel_id = r.hotel_id
5    group by city;

0   recursive calls
0   db block gets
6   consistent gets
0   physical reads
0   redo size
478 bytes sent via SQL*Net to client
496 bytes received via SQL*Net from client
2   SQL*Net roundtrips to/from client
1   sorts (memory)
0   sorts (disk)
```

请注意 consistent gets 的值是 6——一个非常低的值。这一结果是基于这个查询已经基于从 3 张表创建的 2 个视图的查询重写。不是从表查询，而是从 MV 查询，一次消耗了更少的如磁盘 IO 和 CPU 的资源。

但是如果查询重写失败了会怎么样呢？可能会以为几个原因失败：如果初始化参数 query_rewrite_integrity 被设置为 TRUSTED 并且 MV 的状态为 STALE，查询就不会被重写。你可以通过设置会话的参数来模拟这一过程。

```
SQL> alter session set query_rewrite_enabled = false;
```

执行这一命令后，查询计划显示是从 3 张表查询数据，而不是从 MV：

```
0   recursive calls
0   db block gets
16  consistent gets
0   physical reads
0   redo size
478 bytes sent via SQL*Net to client
496 bytes received via SQL*Net from client
2   SQL*Net roundtrips to/from client
```

```
2   sorts (memory)
0   sorts (disk)
```

请注意 consistent gets 的值：从 6 上升到了 16。在真实环境中，这一结果恐怕是无法接受的。因为多出来的资源请求可能无法获得，而你就必须自己重写这一查询了。在那样的情况下，你就必须确保查询一定能被重写。

在 Oracle 9i 和以下版本中，可能只有一个方法实现：使查询重写失效而不是使基础表的访问失败。在 10g 中通过一个特殊的提示可以提供这样的机制：REWRITE_OR_ERROR。上面这个查询就可以这样写了：

```
SQL> select /*+ REWRITE_OR_ERROR */ city, sum(actual_rate)
2   from hotels h, reservations r, trans t
3   where t.resv_id = r.resv_id
4   and h.hotel_id = r.hotel_id
5   group by city;

from hotels h, reservations r, trans t
     *
ERROR at line 2:
ORA-30393: a query block in the statement did not rewrite
```

这样就会产生一个 ora-30393 的错误信息。这个信息表示查询不能通过使用 MV 来重写，因此语句失败。这一错误保护可以防止查询长期运行后系统发生资源缺乏问题。但是还要注意一个潜在问题：如果一个查询成功了，而不是所有都成功了，这些 MV 就能被用于查询的重写。因此，如果 MV_ACTUAL_SALES 而不是 MV_HOTL_RESV 能被使用，查询将会重写，错误也不会产生。这种情况下，查询计划就如以下：

```
Execution Plan
----------------------------------------------------------
0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=11 Card=6 Bytes=156)
1    0   SORT (GROUP BY) (Cost=11 Card=6 Bytes=156)
2    1     HASH JOIN (Cost=10 Card=80 Bytes=2080)
3    2       MERGE JOIN (Cost=6 Card=80 Bytes=1520)
4    3         TABLE ACCESS (BY INDEX ROWID) OF 'HOTELS' (TABLE) (Cost=2
Card=8 Bytes=104)
5    4           INDEX (FULL SCAN) OF 'PK_HOTELS' (INDEX (UNIQUE)) (Cost=1
Card=8)
6    3         SORT (JOIN) (Cost=4 Card=80 Bytes=480)
7    6           TABLE ACCESS (FULL) OF 'RESERVATIONS' (TABLE) (Cost=3
Card=80 Bytes=480)
8    2       MAT_VIEW REWRITE ACCESS (FULL) OF 'MV_ACTUAL_SALES'
(MAT_VIEW REWRITE) (Cost=3 Card=80 Bytes=560)
```

这一查询使用了 MV_ACTUAL_SALES 而不是 MV_HOTEL_RESV，这样，表 HOTELS 和 RESERVATIONS 就能够访问。这种情况下，特别使对后面两种表会做全表扫描的情况下，将会消耗更多的资源——在你创建 MV 和设计查询语句时要特别注意。

尽管你已经通过资源管理器（Resource Manager）控制资源使用，使用这一提示能防止在资源管理器被调用前查询被执行。资源管理器基于优化器的统计数据来降低资源的消耗，因此统计数据的有无将会影响这一过程。而“重写还是报错”这一特性将会不管有误统计数据都会阻值表的访问。

更佳的查询计划

在前面的例子中，请注意在查询计划中有这样一行：

MAT_VIEW REWRITE ACCESS (FULL) OF 'MV_ACTUAL_SALES' (MAT_VIEW REWRITE)

这个访问 MAT_VIEW REWRITE 的方法是新出现的。它表示已经访问了 MV，而不是表或者段。这就可以让你在从名字上无法区分时知道是在使用表还是 MV。

总结

在 10g 中，由于增加了新的 Tuning Advisor，它能向你提供很多关于 MV 设计方面的建议而无需通过猜测方式进行。这使得管理 MV 容易多了。我特别细化能将建议生成一个完整的脚本使实施更加迅速，能节省很多时间这一特性。强制重写或取消查询这一特性在决策支持系统中非常有用。因为这样的系统不允许一个没有被重写的查询在数据库内疯狂执行。

10. ADDM 和查询优化建议器

利用 ADDM 解决性能问题

从最终权威那获得 SQL 调优的帮助：Oracle 数据库本身！通过使用 SQL profiles 来确定查询行为、学习如何使用 ADDM 快速和轻松地解决普通的性能问题。

作为一个高级 DBA，你当然不愿总是被调优某条 SQL 语句这种杂事缠身。

在 10g 中，你有了自动数据库诊断监视器（Automatic Database Diagnostic Monitor ADDM），他是一个不知疲倦的收集数据库性能统计信息来定位性能瓶颈、分析 SQL 语句和不停的提供各种类型的建议以提高性能的机器 DBA，它一般和其他“建议器”如 SQL Tuning Advisor 一起工作。在本文中，你将了解到它是如何工作的。

自动数据库诊断监视器 ADDM

你已经了解了 AWR，它是从数据库的定期采集数据（即快照）中收集与性能相关的细节度量数据的工具。获取到一个快照后，ADDM 就会彻底的分析从不同快照中比较得出的度量数据，并给出必要动作的建议。找到一个问题后，ADDM 可能接着就调用其他 Advisor（如 SQL Tuning Advisor）以提供提高性能的建议。

不用文字来解释这个特性了，下面来展示一下它是如何工作的。假如你需要定位一个无法解释的性能问题。在这一例子中，你已经知道了哪个 SQL 语句需要调优，或者你知道哪个语句有问题。而实际情况下，你可能根本没有什么有用信息。

在 10g 中要做一次诊断，你就必须在相关的快照中选择几个快照以做深入分析。在 10g 企业管理器中，选择“Advisor Central”页，然后点击“ADDM”链接，打开页面如下：

图一 创建一个 ADDM 任务

在这个页面中，你能创建被 ADDM 分析的任务。你知道性能问题出现在下午 11 点，因此选择这一范围（输入一个起始时间、一个结束时间）内的快照。也可以通过点击哪个照相机图标来选择起始、结束范围。选择好时间范围后，点击 OK 按钮，打开页面如下：

图二 ADDM 查找结果

在这，ADDM 发现在这个时间范围内有两个与性能相关的问题：一些 SQL 语句消耗了大量的 CPU 时间，导致数据库显著变慢。基于以上查询结果，ADDM 建议优化这些语句，并将它们高亮显示在图中。

如果你点击每个查询结果，ADDM 会显示更多细节。例如，点击点击以上一个问题，显示结果如下：

图三 ADDM 查询结果的细节

这你可以看到导致这一问题的 SQL 语句。ADDM 在操作部分里建议你将这条语句提交给 SQL Tuning Advisor 来处理。你可以通过点击它后面的按钮立即运行这一任务，那会启动 SQL Tuning Advisor。

在图 2 中，你可能已经注意到了一个名叫“视图报告”的按钮。除了在每个 web 页面上提供建议外，ADDM 还可以为一个更快的一次分析提供文本报告。列表 1 显示了在我们的文本报告例子中复杂的建议。请注意报告中提供的相关的细节，如问题 SQL 语句、它的 hash 值等等。SQL ID 是用于通过命令行在企业管理器的 SQL Tuning Advisor 页面中独立分析的。

表 1：

```
        DETAILED ADDM REPORT FOR TASK 'TASK_2024' WITH ID 2024
        ------------------------------------------------------
           Analysis Period: 16-MAY-2004 from 22:00:31 to 23:01:54
         Database ID/Instance: 3607854283/1
       Database/Instance Names: STARZ10/starz10
                 Host Name: starz
            Database Version: 10.1.0.2.0
             Snapshot Range: from 1863 to 1865
              Database Time: 1123 seconds
        Average Database Load: .3 active sessions
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
FINDING 1: 93% impact (1041 seconds)
-----------------------------------
SQL statements consuming significant database time were found.
   RECOMMENDATION 1: SQL Tuning, 93% benefit (1041 seconds)
     ACTION: Run SQL Tuning Advisor on the SQL statement with SQL_ID
       "8np5s8nvpv7v3".
       RELEVANT OBJECT: SQL statement with SQL_ID 8np5s8nvpv7v3 and
       PLAN_HASH 101258408
       select cust_name from bookings o
       where status not in
       (select status from bookings_hist
       where folio_id = o.folio_id
       and last_mod_time = o.last_mod_time)
 FINDING 2: 89% impact (1000 seconds)
-----------------------------------
Time spent on the CPU by the instance was responsible for a substantial part
```

```
of database time.
   RECOMMENDATION 1: SQL Tuning, 89% benefit (1000 seconds)
     ACTION: Run SQL Tuning Advisor on the SQL statement with SQL_ID
       "8np5s8nvpv7v3".
       RELEVANT OBJECT: SQL statement with SQL_ID 8np5s8nvpv7v3 and
       PLAN_HASH 101258408
       select cust_name from bookings o
       where status not in
       (select status from bookings_hist
       where folio_id = o.folio_id
       and last_mod_time = o.last_mod_time)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~

          ADDITIONAL INFORMATION
          ----------------------
Wait class "Administrative" was not consuming significant database time.
Wait class "Application" was not consuming significant database time.
Wait class "Cluster" was not consuming significant database time.
Wait class "Commit" was not consuming significant database time.
Wait class "Concurrency" was not consuming significant database time.
Wait class "Configuration" was not consuming significant database time.
Wait class "Network" was not consuming significant database time.
Wait class "Scheduler" was not consuming significant database time.
Wait class "Other" was not consuming significant database time.
Wait class "User I/O" was not consuming significant database time.

The analysis of I/O performance is based on the default assumption that the
average read time for one database block is 10000 micro-seconds.

An explanation of the terminology used in this report is available when you
run the report with the 'ALL' level of detail.
```

收集了每个 AWR 快照后再使用 ADDM，因此建议都是基于相邻的两个快照信息。所以，如果分析范围仅仅是两个相邻的快照，你就没有必要像上面一样创建一个 ADDM 任务。如果你想分析两个不相邻的快照，那就需要创建 ADDM 任务。记住，这并不是 ADDM 的全部功能。它还有很多对于内存管理、段管理、redo/undo 等等其他方面的分析和建议。我们无法将 ADDM 的的所有功能在这一篇文章中描述清除，所以这儿我们只关注 SQL Tuning Advisor。下面看看它是如何工作的：

使用 SQL Tuning Advisor 进行分析

在典型的运行的优化器运作过程中，优化器会在所有基于对象统计值的优化路径中选择一个代价最低的。但是在那时，它就没有时间定位哪个语句可以调优、分析数据是否陈旧、是否可以创建索引等等。而 SQL Tuning Advisor 就能像“专家系

统”那样思考。实际上，优化器回答了这些问题：“基于哪些可获得的数据？获得结果的最佳方式是哪个？”；而 SQL Tuning Advisor 则回答了：“基于哪些用户所期望的数据，还可以如何进一步优化？”

这样的“思考”是会消耗如 CPU 那样的资源的。因此 SQL Tuning Advisor 是基于 SQL 在一个调优模式阶段工作的，这个模式能够运行在一个非高峰时期。通过设置创建调优任务函数的参数 SCOPE 和 TIME 可以设置这个模式。最好是在一个数据库的非繁忙时期运行调优模式，这样就不会影响日常用户的使用，把分析留到后面再做。

这一概念可以通过一个例子来解释。例如以下语句：

```
select account_no from accounts where old_account_no = 11
```

这个语句不是很难调优。有两个方法启动建议器：使用企业管理器或使用命令行。

首先看下如何使用命令行来启动。可以通过调用包 dbms_sqltune 来启动。

```
declare

  l_task_id     varchar2(20);
  l_sql         varchar2(2000);
begin
  l_sql := 'select account_no from accounts where old_account_no = 11';
  dbms_sqltune.drop_tuning_task ('FOLIO_COUNT');
  l_task_id := dbms_sqltune.create_tuning_task (
    sql_text  => l_sql,
    user_name  => 'ARUP',
    scope      => 'COMPREHENSIVE',
    time_limit => 120,
    task_name  => 'FOLIO_COUNT'
  );
  dbms_sqltune.execute_tuning_task ('FOLIO_COUNT');
end;
/
```

以上创建和执行了一个名叫 FOLIO_COUNT 的调优任务。接下来就可以见到任务执行的结果了：

```
set serveroutput on size 999999
set long 999999
select dbms_sqltune.report_tuning_task ('FOLIO_COUNT') from dual;
```

结果输出再表 2 中：

表 2：

```
DBMS_SQLTUNE.REPORT_TUNING_TASK('FOLIO_COUNT')
--------------------------------------------------------------------------------
GENERAL INFORMATION SECTION
--------------------------------------------------------------------------------
Tuning Task Name   : FOLIO_COUNT
Scope              : COMPREHENSIVE
Time Limit(seconds): 120
```

```
Completion Status  : COMPLETED
Started at         : 04/06/2004 01:01:31
Completed at       : 04/06/2004 01:01:31
-------------------------------------------------------------------------------
SQL ID  : 1mzhrcv0bg0pw
SQL Text: select account_no from accounts where old_account_no = 11
-------------------------------------------------------------------------------
FINDINGS SECTION (1 finding)
-------------------------------------------------------------------------------
1- Index Finding (see explain plans section below)
--------------------------------------------------
  The execution plan of this statement can be improved by creating one or more
  indices.

  Recommendation (estimated benefit: 94.26%)
  ------------------------------------------
    Consider running the Access Advisor to improve the physical schema design
    or creating the recommended index.
    create index ARUP.IDX$_00001 on ARUP.ACCOUNTS("OLD_ACCOUNT_NO")

  Rationale
  ---------
    Creating the recommended indices significantly improves the execution plan
    of this statement. However, it might be preferable to run "Access Advisor"
    using a representative SQL workload as opposed to a single statement. This
    will allow to get comprehensive index recommendations which takes into
    account index maintenance overhead and additional space consumption.

-------------------------------------------------------------------------------
EXPLAIN PLANS SECTION
-------------------------------------------------------------------------------

1- Original
-----------

DBMS_SQLTUNE.REPORT_TUNING_TASK('FOLIO_COUNT')
-------------------------------------------------------------------------------
Plan hash value: 290945073

-------------------------------------------------------------------------------
| Id  | Operation         | Name     | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |          |     1 |    10 |    34   (0)| 00:00:01 |
|   1 |  TABLE ACCESS FULL| ACCOUNTS |     1 |    10 |    34   (0)| 00:00:01 |
-------------------------------------------------------------------------------
```

```
2- Using New Indices
-------------------
Plan hash value: 633506680

-----------------------------------------------------------------------------------------
| Id  | Operation            | Name       | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |            |     1 |    10 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| ACCOUNTS   |     1 |    10 |     2   (0)|
0:00:01 |
|   2 |   INDEX RANGE SCAN | IDX$_00001 |     1 |       |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------
```

请仔细看这些建议。建议器说需要在字段 OLD_ACCOUNT_NO 上创建一个索引。如果索引建立后，建议器会计算成本，使之潜在的节省更多成本。

当然，考虑到这个例子很简单，你也许可以通过手工检测就能得出同样的结论。但是建议器对于哪些非常复杂的语句就是非常有用的。

中级调优：查询重组

假如查询是以下这个更复杂一些的例子：

```
select account_no from accounts a
where account_name = 'HARRY'
and sub_account_name not in
  ( select account_name from accounts
where account_no = a.old_account_no and status is not null);
```

建议器的建议如下：

```
1- Restructure SQL finding (see plan 1 in explain plans section)
------------------------------------------------------------
  The optimizer could not unnest the subquery at line ID 1 of the execution   plan.
  Recommendation
  -------------
    Consider replacing "NOT IN" with "NOT EXISTS" or ensure that columns used
    on both sides of the "NOT IN" operator are declared "NOT NULL" by adding
    either "NOT NULL" constraints or "IS NOT NULL" predicates.

  Rationale
  ---------
    A "FILTER" operation can be very expensive because it evaluates the
    subquery for each row in the parent query. The subquery, when unnested can
    drastically improve the execution time because the "FILTER" operation is
    converted into a join. Be aware that "NOT IN" and "NOT EXISTS" might
    produce different results for "NULL" values.
```

这次建议器不再对结构改变如索引方面提出建议了，而是更加灵活的提出正确的方法应该是使用 NOT EXISTS 来代替 NOT IN。因为这两个结果比较类似，建议

器给出了它们的关系，让 DBA 或者应用开发人员决定在这个环境下是否要采用这个建议。

高级调优：SQL Profiles

众所周知，优化器通过检查基于与查询相关的对象的统计数据的最低代价来决定一个查询的查询计划。典型情况下，查询与多个表相关，优化器计算通过检查这些表的统计数据计算最低代价，但是它却不知道它们之间的关系。

例如，一个状态为 DELINQUENT 的帐号拥有少于$1000 的余额。如果谓词中仅有关于 DELINQUENT 的过滤的子句，一个连接表 ACCOUNTS 和 BALANCES 的查询将返回较少的记录。优化器不止这一复杂关系；但是建议器知道，它通过数据并存储在一个 SQL Profile 中“猜测”到这个关系。通过读取 SQL Profile，优化器不仅知道表中数据的分布，还可以知道数据之间的关系。这一额外的信息让优化器产生一个更佳的查询计划，因此会有一个调优的更好的查询。

SQL Profile 不需要手工在查询代码中增加提示来调优 SQL 语句。因而，SQL Tuning Advisor 可以不需要改变代码来调优应用了。

这最主要的一点就是，和对象的统计数据不同，SQL Profiles 和一个查询映射，而不是和对象映射。另外一个关于同样两张表——ACCOUNTS 和 BALANCES——的查询可以有一个完全不同的 profile。通过使用这些关于查询的元数据，Oracle 能提高优化性能。

如果能够创建一个 profile（它是在 SQL Tuning Advisor 的会话期间创建的，在这期间，Advisor 产生 Profile 并建议你接受它。）。除非 profile 被接受，否则它不会绑定到语句。你可以通过以下语句随时接受 profile：

```
begin
  dbms_sqltune.accept_sql_profile (
    task_name   => 'FOLIO_COUNT',
    name        => 'FOLIO_COUNT_PROFILE'
    description => 'Folio Count Profile',
    category    => 'FOLIO_COUNT');
end;
```

这一命令将一个早先由 Advisor 产生的名为 FOLIO_COUNT_PROFILE 的 profile 绑定到与前面例子中所描述名叫 FOLIO_COUNT 的调优任务相关的语句上。（请注意，尽管不是 DBA，而是 Advisor 能创建一个 SQL Profile，但只有你能决定是否使用它）。

可以通过视图 DBA_SQL_PROFILES 查看已经产生的 SQL Profiles。字段 SQL_TEXT 的内容是与 profile 相关的 SQL 语句；字段 STATUS 表明 profile 是否激活可用。（尽管一个 profile 已经绑定到一条语句，但它还是需要通过激活来影响查询计划。）

使用 ADDM 和 SQL Tuning Advisor

除了上面描述的三种情况，SQL Tuning Advisor 还会鉴别出与一个查询相关的对象中哪些没有统计数据。所以，Advisor 执行了四种不同类型的任务：

- 检查对象是否由合法、可用的统计数据用于相应的优化。
- 试图重写语句以获得更好的性能，并提供重写建议
- 检查访问路径，看是否可以通过增加额外结构（如索引、物化视图）来提高性能
- 创建 SQL Profiles 并将它们绑定到查询语句

基于这些功能，我认为 ADDM 和 SQL Tuning Advisor 最少在三种情况下是十分强大的工具：

- **反应调优：**你的应用一下子变得性能很差。通过使用 ADDM，你可以将问题定位到一条或一组 SQL 语句，它们会被显示在 ADDM 页面中。随着 ADDM 的建议，你可以调用 SQL Tuning Advisor 和修正问题。
- **预先调优：**应用运行良好。然而你希望确认所有必要的维护任务都在执行，并且想要知道查询能否调优得更好。你可以在 standalone 模式下启动 SQL Tuning Advisor 来确定调优得可能性。
- **开发调优：**当在开发的代码测试阶段时，有很多机会来调优查询，而不要等到 QA 或者生产阶段。你可以在代码最终开发出来前使用命令行方式来调优单个的 SQL 语句。

使用企业管理器

前面的例子时特地介绍如何在命令行方式下使用 SQL Tuning Advisor，这对于创建预先任务脚本比较有用。然而，在大多数情况下。你需要针对用户报告的问题实施调优。10g 企业管理器对这种情况就非常方便。

如何使用企业管理器来诊断和调优 SQL 语句：在数据库主页，点击屏幕底部的“Advisor Central”链接，这就会启动包括所有 advisor 的页面。下一步，点击图 4 所示屏幕上方的“SQL Tuning Advisor”

图 4：企业管理器的 Advisor Central

启动了 SQL Tuning Advisor 后，如图 5 所示，选择“Top SQL”：

图 5：SQL Tuning Advisors

这样就启动了一个如图 6 所示的页面，里面有一张图，包括了在一个时间维度里跟踪的各种等待分类。

图 6：Top SQL Chooser

上图中用红色圈标明了一块灰色矩形区域。通过鼠标拖动矩形区域到 CPU 等待很高的地方。页面的下面部分将显示那一时间段的相关的 SQL 语句：

图 7：基于活动信息选择 SQL 语句

如你所见，显示在最上面的 SQL 语句消耗了最多的 CPU。点击语句 ID 查看关于它的细节，会打开以下图：

图 8：SQL 细节

在这个图中，你可以见到那段时间导致 CPU 消耗这么搞的那条 SQL 语句。你可以点击按钮“Run SQL Tuning Advisor”（用红圈标出）来运行 Advisor。如下图所示：

图 9：SQL Tuning Advisor 计划

在 advisor 计划中，你可以决定任务类型和执行多少分析。例如，在上图中，选择“comprehensive”分析，Advisor 会立即运行。Advisor 运行完成后，你可以见到它的建议，如图 10 所示：

图 9：Advisor 建议

上面藐视的过程和你在命令行方式见到的很类似。然而，这个流程对于你在实际环境中遇到的问题更有用：发现导致问题的原因、接受如何修正它的建议。

总结

ADDM 是一个强大的工具，它拥有能分析性能度量数据的“大脑”，并能基于经验丰富的 Oracle 专家的总结出的最佳经验和方法学给出优化建议，并且全是自动的。这些功能不仅能告诉 DBA 发生了什么问题及为什么会产生这样的问题，最终要的是，它能告诉下一步如何去做。

11. 选择性编译

Oracle10g 中提供了一个十分方便开发人员的新特性——选择性编译。即可以通过条件，只编译 PL/SQL 中的部分代码。如果你对 C++很熟悉，那你会对这个特性感觉非常亲切，因为它和 C++中的条件宏十分相似。在 9i 的时候，开发人员将自己的调试信息加入到程序中，往往 release 之后都没有删掉，这些信息轻则影响可读性，严重的话会影响系统性能。有了这个特性后，这个问题就可以解决了。下面举个简单的例子解释一下：

创建一个有条件宏的函数：

```
SQL> CREATE OR REPLACE FUNCTION F_TESTDEBUG
  2  RETURN NUMBER IS
  3    v_count number;
  4  BEGIN
  5    select count(*) into v_count from user_tables;
  6
  7    $IF $$my_debug $THEN
  8       DBMS_OUTPUT.PUT_LINE('Tables number is: ' ||
v_count);
  9    $END
 10
 11    return v_count;
 12  END;
 13  /
```

Function created

激活 debug 信息（通过变量$$my_debug 控制）：

```
SQL> ALTER FUNCTION F_TESTDEBUG COMPILE PLSQL_CCFLAGS =
'my_debug:TRUE' REUSE SETTINGS;

Function altered
```

运行函数：

```
SQL> set serveroutput on
SQL> declare
  2  v_res number;
  3  begin
  4  v_res := F_TESTDEBUG;
  5  end;
  6  /

Tables number is: 22

PL/SQL procedure successfully completed
```

这时候，可以看到 debug 信息被打印出来了。

我们再去掉调试信息：
SQL> ALTER FUNCTION F_TESTDEBUG COMPILE PLSQL_CCFLAGS =
'my_debug:FALSE' REUSE SETTINGS;

Function altered

执行函数：
```
SQL> declare
  2  v_res number;
  3  begin
  4  v_res := F_TESTDEBUG;
  5  end;
  6  /
```

PL/SQL procedure successfully completed

我们可以看到，调试信息没有了。

当然，我们在实际 release 的时候就不需要用 alter 了（除非现场调试），可以在 release 脚本的头部定义好 my_debug 变量就 OK 了。

12. 代码混淆

Oracle 9i 之前，提供了一个命令'warp'来加密代码。但它是命令行方式的，是针对脚本的。10g 中，Oracle 包 DBMS_DDL 提供了一个新的函数 CREATE_WRAPPED 来实现对代码的混淆。举例：

```
SQL> declare
  2  V_PROTEX varchar2(2000):='
  3  CREATE OR REPLACE PROCEDURE P_WRAPTEST
  4  IS
  5    v_count number;
  6  BEGIN
  7    select count(*) into v_count from user_tables;
  8    DBMS_OUTPUT.PUT_LINE("Tables number is: " || v_count);
  9  END;
 10  ';
 11  begin
 12  SYS.DBMS_DDL.CREATE_WRAPPED(V_PROTEX);
 13  end;
 14  /

PL/SQL procedure successfully completed
```

再看混淆后的代码：

```
SQL> SELECT text FROM USER_SOURCE WHERE name = 'P_WRAPTEST';

TEXT
------------------------------------------------------------------------------
PROCEDURE P_WRAPTEST wrapped
a000000
367
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
7
a0 db
p9j84ore7Zt5Hy6AyFLId/b/8yMwgwBf2ssVfC+iJhAYzfFpuKZxlTM5k5uuM9GJb/cJyvtE
80zL0AI2lSFOiy6XzIzuX8j8bND0S8hL6aZuCkp25+ESApOeAkn9ow9QaxvOFT8kqbKOtYY+
gChxfOqrjWpmE3CJliplbOiEDDLfcJqzMryW1Xqa0EJCr3kA5lsFM9qrGt1wGrQQpg5d1Y8=
```

13. 正则表达式

在进行查询时，有时候需要按照一定的特殊规则来查找某个字符串，比如，你可能需要查询第三位为 5-8，最后四位为'8888'的所有电话。在 9i 之前，你可能需要写一个很复杂的条件：

```
Select username from t_userinfo
where (phonenumber like '135%8888'
or phonenumber like '136%8888'
or phonenumber like '137%8888'
or phonenumber like '138%8888')
and length(phonenumber) = 13;
```

那时就会很羡慕 java 程序员可以使用一个正则表达式轻松搞定。10g 中，再也不需要这么复杂了， oracle 也提供了几个正则表达式函数，大大方便了开发人员：REGEXP_LIKE、REGEXP_REPLACE、REGEXP_INSTR、REGEXP_SUBSTR，分别用于模糊匹配、代替、插入、截取字符串。关于正则表达式的规则这就不详细描述了， 可以查相关资料得到。简单举例。以上面例子为例，我们的查询语句可以写成：

```
SQL> create table t_userinfo (username varchar2(10), phonenumber varchar2(13));

Table created

SQL> insert into t_userinfo values ('zhansan', '13012323434');

1 row inserted

SQL> insert into t_userinfo values ('lisi', '13512348888');

1 row inserted

SQL> insert into t_userinfo values ('wangwu', '13912328888');

1 row inserted

SQL> insert into t_userinfo values ('zhaoliu', '13743218888');

1 row inserted

SQL> insert into t_userinfo values ('sunqi', '1361234888');

1 row inserted

SQL> commit;

Commit complete

SQL> Select username, phonenumber from t_userinfo
  2  where REGEXP_LIKE(phonenumber, '13[5-8][0-9][0-9][0-9][0-9]8{4}');

USERNAME   PHONENUMBER
---------- -------------
```

lisi 13512348888
zhaoliu 13743218888