第12章　大型数据库管理

“大型”数据库的概念一直在不断变化，1995年时，容量大于100GB的数据库就被认为是大型数据库。仅仅几年之后，数万亿字节的数据库已投入市场。VLDB是超大型数据库的简称。如果不用一个数字阈值来定义它，很快就会失去意义。随着技术的不断进步，更快备份、更大系统和I/O分布将不断增加最大可支持的数据库容量。

不再用某个指定的容量来定义“大型”，而是依据其恢复时间来定义：如果不能在18小时内从全联机备份中完全恢复一个数据库，这就是一个大型数据库。这个定义使得“大型”数据库的容量随操作系统及硬件性能的改进而增加。

当处理一个大型数据库时，必须从完全不同的方面来考察数据库。本章将对大型数据库的管理提出一些建议，其中包括：

- 设置环境，其中包括分区和显形图。
- 管理事务，其中包括数据装载。
- 实现备份策略。
- 协调。
- 将可迁移表空间用于数据传送。

本章的某些建议不能用于较小的系统。如果你正在管理可在18小时之内完全恢复的数据库，请在本书的其他章节中寻求帮助。

12.1　设置环境

在一个大型数据库中，数据库的大多数空间都被少量的表占用。例如，在用作决策支持目的的大型数据库中，可能有100个表，其中5个表的记录占据整个数据库记录数的90%以上，而余下的95个表只是一些代码表或专用的报表函数。为了改善对应用程序的查询性能，可以根据最大的表创建一些包含数据聚集的表，每个这样的表都比最大表小，而代码表则更小。如果应用程序和终端用户访问聚集表而不是大型事务表，调整重点就放在调整对聚集表的访问上。

在创建并管理一个大型数据库时，大量任务是管理占数据库记录行大多数的很少几个大型表。本节提供的数据库配置提示，包括了如何将大型表透明地划分为较小(更易于管理)的表及将大型表的I/O需求分布到多个设备上的方法。

12.1.1　确定大型数据库的容量

创建一个大型数据库时，可依据下列类型对所创建的每个表进行分类：

- 小型代码表　容量上很少增加的代码表。
- 大型事务表　在数据库中占据大多数记录的一种表。其容量会随时间不断变化。
- 聚集表　这种表的容量可着随时间的变化而增大，也可以保持不变；变与不变依赖于应用程序设计。理想的情况是使其容量保持不变。其数据以大型事务表中的聚集数据为基础。

• 临时工作表 在数据库装载及大批量数据处理过程中使用的临时工作表。

下面几小节描述每种表的容量设置和配置建议。

1. 设置代码表容量

代码表中包含一个代码和描述的列表，如国名缩写或国名列表。代码表在容量上始终不变。

由于代码表的数据是非常静态的数据，所以可以给它设置一个适当的尺寸，并且不必为它是否随时间的推移而变为碎片担忧。如果代码表是一些小表，在创建时可取一些适当的存储参数，以便使每一个表适合一个盘区。如果有多个具有相似容量的代码表，可以将它们存储在其缺省存储参数适合它们的一个表空间内。例如，如果所有代码表都需要 500KB到1MB的空间，就可以将它们全部存储在CODES_TABLES表空间中。

```
create tablespace CODES_TABLES
datafile '/u01/oracle/VLDB/codes_tables.dbf'
default storage
  (initial 1M next 1M pctincrease 0 pctfree 2);
```

如果在CODES_TABLES表空间中创建一个表，只要不指定表的参数值，这个表将以表空间的缺省存储参数作为自己的存储参数。在这个例子中，使用缺省存储参数值在CODES_TABLES表空间中创建的每一个表，都将有一个容量为 1BM的初始盘区，所有后续盘区容量也是1MB。使所有盘区同样大小可最大程度地重新使用撤消的盘区。如果必须使用不同大小的盘区，应调整这些盘区大小，以便最大程度地重新使用撤消的盘区。通常使用的盘区大小为1MB、2MB、4MB、8MB、16MB等。

作为CODES_TABLES表空间的缺省存储子句的一部分， pctfree设置值是非常低的。pctfree的低设置值表示，每一个数据库块中没有多余空间来容纳后续的更新内容。通常，代码表的数值很少被更新 (例如，国名是不经常被修改的)，如果一个代码表有经常被更新的值，就必须给pctfree设置一个较高的值以支持空间更新的需求。

因为代码表拥有类似的存储及使用特性，所以它们常常被存储在一起。如果有多个代码表集，可以将它们存储在多个表空间中。例如，如果有些代码表比其他代码表使用得更频繁，可以将它们移到它们自己的表空间中，这样就不会与其他代码表空间发生冲突。

另一种可能是，可以从数据库中消除代码表并将静态代码值作为数组存储在程序中。如果代码表数据是小的静态数据，将这些数值硬编码进程序对长期维护费用不会产生什么影响。例如，可能有一个跟踪测量单位的表

缩 写	测量单位
V	伏
KW	千瓦
F	华氏温度

可以按数组的形式将这个数据存储在程序中，而不是存储在一个 Oracle表中。在C和C++语言中，可以把数组数据存储在一个头文件上；在 COBOL语言中，可以把这些值存储在复制的源程序段(copybook)中。当数据作为数组存储时，就不需要数据库访问来引用测量单位。如果应用程序需要一个新的测量单位，就要修改程序的头文件并重新生成代码。为适应代码值的变化而修改程序可能不用花费大量维护费用，因为可能要编辑程序以创建新的函数、类和过程来处理新的测量单位。如果将静态代码值硬编码进程序，就将它们从数据库中完全删除。

需要根据代码表数据的易变性来权衡这种方法在现实环境中的利弊。

2. 设置事务表容量

事务表存储着数据库中的大部分数据，存储着建立聚集表的原始数据。确定事务表的大小时，首先必须了解在应用程序中如何处理历史数据。应用程序是否总是在其事务表中存储同样容量的数据？过去存储的记录是否是无限期的？

如果事务表总是存储相同容量的数据，表的容量设置就直接明了，可以估计表中的行数及每个行的大小。在大型数据库中，事务表一般通过批处理程序进行装载。因此必须明确两点：输入文件的大小及数据变化的源由。

通过分析输入文件，应能估计事务表中行的容量。数据变化的源由也同样重要。如果表被截断且由后续的数据装载重新装载，此时就不需要为表设置高的 pctfree值。如果记录将被更新，就需要设置一个足够高的 pctfree值，以便使更新的行仍能适合其原始数据块。如果一个行不能适合其原始数据块，Oracle就可能将该行迁移到一个新数据块中或者使它跨越多个数据块。

一个表的pctincrease值应支持表的增长模式。如果表以一个恒定的速率增长，则应该设pctincrease为0；如果表中数据容量呈几何级数增长，则必须为 pctincrease设置一个非零值。对于大多数表，设置pctincrease为零就可以正确支持其增长情况。

> **注意** 在大多数表中，行数是以线性速率增长的；例如，以10行、20行、30行、40行这种方式来累积记录数，每一次都是固定增长 10行。如果记录数以几何级数增长，则每次增加的记录将是10、20、40、80，从而每一次累积记录的总数将是10、30、70、150。

对于Oracle8，可以将一个大型事务表划分为多个较小的表。当将一个表划分为几个分区时，必须确定每一个分区的大小。正确确定分区尺寸需要了解表中数据值的分布。由于数据通常是分批装载，因此在装载数据之前就可以了解数据分布信息。在本章后面 12.1.4节“分区”中将对如何创建分区进行介绍。

3. 设置聚集表容量

聚集表存储事务表中的汇总数据，存储这种冗余数据一般是为了改善应用程序中频繁的屏幕显示或报表的性能。应用程序的大多数用户很少需要了解详细的事务数据。通过把事务表的聚集数据存储在聚集表中，就可以达到以下两个目的：

- 改善对应用程序的查询性能 不要使数据库必须支持那些对数据库中最大的表频繁执行聚集操作(如SUM、MIN、MAX、AVG)的用户。如果将聚集数据存储在它们自己的表集中，用户就查询小一些的聚集表而不是巨型事务数据。
- 减少访问事务表的次数 聚集表是通过批处理进程创建起来的，所以只能通过批处理程序访问事务表。减少访问事务表的次数可大大改善对事务表进行维护、装载、更改和管理的性能。例如，如果用户只直接访问聚集表，就可以截断主事务表而不会影响应用。

如果事务表随时间不断增长，聚集表也就会随时间的推移而增长。然而更普遍的是，随着时间的推移，聚集表的个数(而不是容量)在不停地增长。例如，可以创建一个容纳某一时间段内的销售数据的聚集表(如产品发出后的第一个销售期)。下个时间周期又将把销售数据存储在什么地方呢？可以扩展已存在的聚集表或创建一个容纳新数据的新表。如果数据被存储在一个新表中(通常如此)，根据对现有聚集表的存储需求就能够准确预测新表的数据量。如果新数据存储在已存在的聚集表中，就必须估计出每个周期要存储在一个聚集表中的数据量。如

果对聚集表中的数据量没有限制，这些表最终会变成大型表，并且这些表的性能和管理就可能会成为一个问题。

4. 设置临时工作表容量

临时工作表在进行批数据装载时使用。例如，可以使用一个不对输入数据进行约束条件检查的装载进程。在向事务表装载新数据之前，应先“清理”这些数据以确保其正确性和合理性。由于批装载一般是使用外部数据源，所以有一些记录在插入过程中可能因不符合系统标准而导致插入失败。可以在数据装载期间使用逻辑检查方法，或者如果使用临时工作表，就继续进行数据装载。

在数据装载后采用数据清理逻辑就可以调整数据装载以迅速处理行。然而，由于新数据在装载到产品事务表之前必须存储在一个单独的表中，这就势必增加数据库所需的空间量。临时工作表应存储在一个与产品数据表分开的表空间中，其容量大小应与容纳数据装载的记录所需的空间匹配。

除了为临时工作表分配空间外，在数据装载和验证进程中还应为在这些表上创建的索引分配空间。

5. 设置其他数据库区容量

除了产品表及其相关的索引需要的空间外，还需要为其他核心数据库对象——数据字典、回滚段和临时段——提供足够的空间。

数据字典存储在SYSTEM表空间中。包、过程、函数、触发器和方法的源代码存储在数据字典表中。如果应用程序需要大量使用这些对象，就必须增加 SYSTEM表的可用空间。使用审计(见第9章)也要增大SYSTEM表空间内潜在的表空间使用。

回滚段支持数据库中的事务，在大型数据库中，有三种截然不同的事务：

- 大型批装载　这种装载需要大型专用回滚段支持。
- 批聚集　聚集所需要的数据容量和事务大小一般比大型数据装载所需要的要小一些。需要几个专用回滚段来支持聚集，尽管各个聚集回滚段可能比用于支持初始数据装载的回滚段要小，但其总的空间分配量可能超过数据装载回滚段的空间分配量。
- 用户执行的小事务　如果用户频繁执行数据库中的小型事务，就要提供大量具有多个盘区的回滚段以支持事务处理。

有关确定回滚段数量、大小和结构的信息，请参见第 7章。

Oracle在排序操作进程中创建临时段。如果使用 SQL*Loader Direct Path选项并且装载的表已被索引，初始数据装载就可以使用临时段。通常，一个大型数据库中的大多数临时段活动都来自聚集活动和创建索引。在估计所需要的临时段的大小时，首先估计要创建的表和索引的大小。例如，假设要创建的聚集表为 50MB，再将这个容量乘以4。对于要创建的聚集表，应提供一个容量为 200MB的临时段。临时段将建在用户的临时表空间内，它使用临时表空间的缺省存储参数，因此必须确保那些存储值要大得足以支持聚集表及排序操作。

注意　如果正在使用SQL*Loader Direct Path装载选项，临时段就必须足够大，以便存储将要装载的表所使用的索引。

12.1.2 设置支持区大小

除数据库外，还有几种文件可能使用大量的磁盘空间，在实现系统之前需要对这个区域

进行规划。

首先，需要有装载原始数据文件的空间，除非直接从磁带或 CD盘上装载数据，否则可能要在系统的磁盘上存储原始数据文件。其次，需要为数据装入数据库前出现的文件处理准备空间。例如，通常在装载数据前对其进行排序，因此必须有存放原始文件和这个文件排序后的版本的空间。第三，在设置时间周期内可能需要使文件处于联机状态。确保有足够的空间来容纳多个文件集，确保当旧文件不再需要联机时将它们删除掉。

最后，根据备份策略，可能需要容纳归档重做日志文件或导出文件的磁盘空间。有关大型数据库备份策略的详细情况，请参见本章后面 12.3节“ 备份 ”。

以上三种区域的大小根据应用程序和备份策略的实现不同而不同。在最终确定新系统的空间之前，首先要把所需要的非数据库空间确定下来。

12.1.3 选择物理设计

在对一个大型数据库进行物理设计时，应有以下两个主要目标：

1) 尽可能将I/O负载分散到多个磁盘上。

2) 减少不同类型对象间的I/O冲突。

为了把I/O负载分散到多个磁盘上，许多大型数据库系统使用磁盘的 RAID阵列。一个RAID阵列将一组磁盘作为一个逻辑卷来处理；当一个文件在此逻辑卷中创建时，操作系统把该文件分散到各个磁盘上。例如，如果 RAID阵列中有4个磁盘，第一个磁盘就可以存放文件的第一个数据块，第二个磁盘可以存放文件的第二个数据块，第三个磁盘可以存放文件的第三个数据块，第四个磁盘可以存放一个奇偶校验块。如果由于介质失败而引起某个磁盘丢失，RAID部件就可以使用剩余的磁盘重新建立起丢失的数据。

RAID阵列中磁盘越多，I/O负载就分散得越广。由于 RAID阵列包含写入和维护奇偶校验信息，所以它们一般对一次写、多次读应用程序最有效。如果数据被频繁更新，则必须考虑一个可选择的存储机制 (如前面所述的镜像)。例如，在对一个数据值进行更新期间，一个RAID系统将需要读取原始数据块和原始奇偶校验块，然后更新并将这两个数据块写回磁盘上。许多大型RAID系统都使用一个磁盘缓存区来改善这种操作的性能，但是它们仍然可能遇到性能退化问题。

一个可选择的存储结构使用镜像。在一个镜像系统中，操作系统为每个磁盘保持有多个备份。例如，可能有一个磁盘 Disk1和一个Disk1的复制磁盘。在进行读和写时，操作系统可以对两个磁盘中的任何一个进行读或写。操作系统在镜像磁盘上维护文件读取的一致性。镜像文件对于读或写都有效，但使用它需要双倍的磁盘空间。

为减少不同类型对象间的 I/O冲突，可使用第 4章中描述的方式来对要存储的文件进行分类。然后根据本章前面描述的类别对应用程序使用的表进行分类 (代码表、聚集表等)。必须将临时工作表与其数据源 (外部数据文件)分开，必须将事务表与其数据源 (临时工作表或外部数据文件)分开存储，必须将聚集表与其数据源 (事务表)分开存储。所有这些又必须与回滚段、数据字典和临时段分开。

如果正在使用 RAID设备又该怎么办呢？此时，把每个 RAID磁盘集当作一个独立的磁盘来对待。应把事务表和聚集表存储在不同的 RAID集中，把表和表的索引存储在不同的 RAID集中。

注意 确定最佳RAID配置可能需要与系统管理组人员进行大量交流。可能要向系统管理员说明数据库的文件访问模式，需要弄清楚他们分配 I/O负载的方式。

12.1.4 分区

为了使大型数据库的管理更方便，可以使用分区。分区将表中的行动态地分到小一些的表中。可以使Oracle创建一个连接所有分区的视图；这样，尽管数据在物理上被分割开，但在逻辑上仍以一个整体出现。将一个大型表分成多个较小一些的表可以改进维护操作、备份、恢复、事务处理和查询的性能。

用来确定哪个行存储在哪个分区的准则作为 create table命令的一部分被指定。用这种方式将一个表的数据分到多个表中叫作划分表。被划分的表叫做分区表，其各部分叫作分区。

如本章后面所述，Oracle优化程序知道表已经被划分。也可以将在查询的 from子句中指定要使用的分区。

在Oracle8i中，Oracle支持三种划分方式：范围划分、散列划分和混合划分。在 Oracle8.0中，只支持范围划分。下节的例子集中讨论范围划分；混合划分和散列划分在本章后面“管理子分区”小节中介绍。

1. 创建分区表

创建一个分区表时，必须将用于分区的范围值作为 create table命令的一部分指定。

下面考察EMPLOYEE表：

```
create table EMPLOYEE (
EmpNo           NUMBER(10) primary key,
Name            VARCHAR2(40),
DeptNo          NUMBER(2),
Salary          NUMBER(7,2),
Birth_Date      DATE,
Soc_Sec_Num     VARCHAR2(9),
State_Code      CHAR(2),
constraint FK_DeptNO foreign key (DeptNo)
   references DEPT(DeptNo),
constraint FK_StateCode foreign key (State_Code)
   references State(State_Code)
);
```

如果要在EMPLOYEE表中存储大量记录，可能要把EMPLOYEE的行分到多个表中。若要按范围划分表的记录，可使用 create table命令的partition by range子句，这些范围确定存储在每个分区的值。

用作分区逻辑基础的列很少是表的主键，最常用的划分基础是表中一个外键，在EMPLOYEE表中，DeptNo和State_Code列是外键。如果频繁使用 DeptNo列进行查询，就可以基于这个列分开数据，然后将其用作分区键。在一个跟踪数据变化 (如工资变化和销售变化)过程的系统中，更适合基于时间列 (如销售周期或工资变动的有效日期)划分数据。

```
create table EMPLOYEE (
EmpNo           NUMBER(10) primary key,
Name            VARCHAR2(40),
DeptNo          NUMBER(2),
Salary          NUMBER(7,2),
Birth_Date      DATE,
Soc_Sec_Num     VARCHAR2(9),
```

```
 constraint FK_DeptNO foreign key (DeptNo)
   references DEPT(DeptNo)
)
partition by range (DeptNo)
 (partition PART1   values less than (11)
   tablespace PART1_TS,
  partition PART2   values less than (21)
   tablespace PART2_TS,
  partition PART3   values less than (31)
   tablespace PART3_TS,
  partition PART4   values less than (MAXVALUE)
   tablespace PART4_TS)
;
```

EMPLOYEE表将依据DeptNo列中的值进行划分;

```
partition by range (DeptNo)
```

对于任何小于 11 的DeptNo值，记录将存储在 PART1分区中。 PART1分区将存储在PART1_TS表空间中。对于11至20之间的任何DeptNO，记录都存储在PART2分区中；DeptNo值在21至30之间的记录存储在PART3分区中。所有DeptNo大于30的记录都存储在PART4分区中。这里要注意的是，在PART4分区的定义中，范围子句是：

```
partition PART4   values less than (MAXVALUE)
```

不必为最后一个分区指定一个最大值； maxvalue关键字通知Oracle使用该分区存储所有前述分区不能存储的记录。对每一个分区，只指定该范围的最大值，其最小值由 Oracle隐含确定。

对一个表进行分区时，应将这些分区存储在不同的表空间中。用表空间将它们分开存储使你能控制其物理存储位置并避免分区间的冲突。

到底应该有多少个分区呢？逻辑上需要有多少个分区来分隔数据，就应该有多少个分区。拥有许多分区所带来的额外维护工作微不足道。重点在于将数据表的行划分为逻辑组。如果分区范围值正好满足应用程序需求，那么就使用这个分区范围值对表进行划分。如果最大的表为100GB，使用100个相同容量的分区来生成 100个分区，每个分区为 1GB。尽管管理1GB的表并非那么简单，但肯定比管理 100GB的表要容易得多。所以，使用足够的分区来减少表的容量，从而使其在操作系统和硬件配置中易于管理。

注意 不能将对象表或使用LOB数据类型的表进行划分。

直接从分区中进行查询

如果知道要搜索数据的分区，就可以将分区名作为查询中 from子句的一部分指定。例如，如果想查询Departments(部门)11到20的雇员的记录，将如何处理呢？优化程序应能用分区定义来断定，只有 PART2分区含有可以解决这个查询的数据。如果愿意的话，可以在查询中指示Oracle使用PART2分区：

```
select *
  from EMPLOYEE partition (PART2)
 where DeptNo between 11 and 20;
```

这个例子明确指定 Oracle到哪个分区中去寻找符合条件的雇员记录。如果分区被修改 (例如，其范围值发生变化)，PART2就可能不再是含有所需记录的分区。因此，使用这种语法时要加倍小心。

通常不需要这种语法，因为 Oracle对每一个分区都设置有 CHECK约束条件。当查询一个分区表时，Oracle使用CHECK约束条件来确定查询涉及到哪些分区。这个过程使查询可以在少量的行上进行，从而改善了查询性能。另外，这些分区可以存储在不同表空间 (进而在不同的磁盘设备)中，从而有助于减少查询期间可能出现的 I/O冲突。

在对分区表进行插入操作时， Oracle使用分区的CHECK约束条件确定应把记录插入到哪个分区中。因此，可以把一个分区表当作一个独立的表使用，依靠 Oracle来管理数据的内部分离。

2. 分区的索引

当创建一个分区表时，应该对该表创建一个索引。表索引也随划分表的范围值进行划分。在下面的程序清单中，示出了 EMPLOYEE表的create index命令:

```
create index EMPLOYEE_DEPTNO
  on EMPLOYEE(DeptNo)
   local
   (partition PART1
     tablespace PART1_NDX_TS,
    partition PART2
     tablespace PART2_NDX_TS,
    partition PART3
     tablespace PART3_NDX_TS,
    partition PART4
     tablespace PART4_NDX_TS);
```

请注意local关键字。在这个create index命令中，没有指定范围，而是由 local关键字来指示Oracle为EMPLOYEE表的各分区创建一一对应的索引。 EMPLOYEE上创建有4个分区。这个索引将创建4个独立的索引，分别对应每一个分区。由于每个分区一个索引，所以索引是分区的“ 局部索引 ”。

局部分区的索引工作方式与 Oracle传统索引的工作方式完成类似；一个索引对应一个表。如果使用局部分区，管理索引与相应的表应该是很简单的。

也可以创建“ 全局 ”索引，全局索引可包含多个分区的值。即，该索引的值分布在多个表中。当分区中出现许多事务并且需要保证所有分区中数据值的唯一性时，通常是采用全局索引。局部索引也可以保证数据值的唯一性，但全局索引在执行唯一性检查过程中应更快一些。索引本身也可以被分区，如下例所示：

```
create index EMPLOYEE_DEPTNO
on EMPLOYEE(DeptNo)
 global partition by range (DeptNo)
 (partition PART1   values less than (11)
   tablespace PART1_NDX_TS,
  partition PART2   values less than (21)
   tablespace PART2_NDX_TS,
  partition PART3   values less than (31)
   tablespace PART3_NDX_TS,
  partition PART4   values less than (MAXVALUE)
   tablespace PART4_NDX_TS)
;
```

create index命令中的global子句允许指定索引值的范围，这个范围可以与表分区的范围不同。在这种情况下，使用相同的分区范围。即使表和全局索引使用相同的分区范围，索引分区也不直接与表分区相联系。索引分区只不过是其值跨越所有表分区的全局索引的一部分。

只有索引为局部索引时，表分区和索引分区之间才建立起直接关系。

在大多数情况下，应使用局部索引分区。如果使用索引的局部分区，就能很容易地将索引分区与表分区联系起来。局部索引比全局索引更易于管理，因为它们只是代表分区表中的部分数据。下面介绍分区管理方面的情况。

3. 分区管理

可以使用alter table命令来增加、删除、交换、移动、修改、更名、拆分和截断分区。这些命令使你可以改变已存在的分区结构，当一个分区表被使用一段时间后就可能有这种需求。例如，分区表中DeptNo值的分布可能发生改变或可能增大了最大值。

有关alter table命令的所有语法，请参见附录A。

可以使用alter table命令管理分区的存储参数。在本章前面对EMPLOYEE表进行划分时，没有为分区指定存储参数。例如，第一个分区PART1被分配到PART1_TS表空间中，没有用storage子句。

```
partition by range (DeptNo)
 (partition PART1   values less than (11)
   tablespace PART1_TS,
```

PART1分区将使用PARTl_TS表空间的缺省存储参数。如果要使用不同的存储参数，就必须在创建表时(通过storage子句)指定或在创建表后改变分区的存储参数。

例如，下面命令改变了EMPLOYEE表中PART1分区的存储参数：

```
alter table EMPLOYEE
   modify partition PART1
   storage (next 1M pctincrease 0);
```

可以截断分区，而对表的其余分区没有影响。如下所示：

```
alter table EMPLOYEE
  truncate partition PART3
  drop storage;
```

也可以使用alter table命令将分区移到一个新的表空间，将现有分区划分为多个新分区，交换分区，删除分区和增加新分区。

对于索引分区来说，选项就更受限制。与分区相关的alter index命令的语法在附录A示出。可以使用alter index命令来修改一个索引分区的存储参数，或者更名、删除、拆分或重建分区。可以使用与分区相关的对alter index命令的扩展来管理索引分区，其方法与管理常规索引相同。例如，可以通过alter index命令的rebuild partition子句来重建一个现有的索引分区。

```
alter index EMPLOYEE_DEPTNO
rebuild partition PART4
storage (initial 2M next 2M pctincrease 0);
```

将rebuild选项用于索引或索引分区时，必须有足够的空间同时存储新旧两种索引。

4. 管理子分区

在ORACL8i中，可以创建子分区——分区中的分区。可以用子分区将两种不同的分区——范围分区和散列分区——结合起来。范围分区在Oracle8.0中是唯一可用的分区。

与散列簇一样，散列分区通过对分区键值执行一个散列函数来确定数据的物理位置。在范围分区中，分区键的连续值通常存储在同一个分区中。在散列分区中，分区键的连续值不必存储在同一分区中。散列分区在比范围分区更大的分区集上分配一个记录集，从而减少了I/O冲突的可能性。

若要创建一个散列分区，可使用partition by hash子句而不是partition by range子句。例如：

```
create table EMPLOYEE (
EmpNo           NUMBER(10) primary key,
Name            VARCHAR2(40),
DeptNo          NUMBER(2),
Salary          NUMBER(7,2),
Birth_Date      DATE,
Soc_Sec_Num     VARCHAR2(9),
 constraint FK_DeptNO foreign key (DeptNo)
   references DEPT(DeptNo)
)
partition by hash (DeptNo)
partitions 10;
```

与范围分区一样，可以命名每一个分区并规定其表空间。例如：

```
create table EMPLOYEE (
EmpNo           NUMBER(10) primary key,
Name            VARCHAR2(40),
DeptNo          NUMBER(2),
Salary          NUMBER(7,2),
Birth_Date      DATE,
Soc_Sec_Num     VARCHAR2(9),
 constraint FK_DeptNO foreign key (DeptNo)
   references DEPT(DeptNo)
)
partition by hash (DeptNo)
partitions 2
store in (PART1_TS, PART2_TS);
```

在partition by hash (DeptNo)行之后，有两种格式可供选择：

- 如上面程序清单所示，可以指定使用的分区数量和表空间：

```
partitions 2
store in (PART1_TS, PART2_TS);
```

这种方法将用SYS_P*nnn*格式的系统生成名创建分区。 store in子句中指定的表空间数量不必与分区的数量相等。如果指定的分区比表空间多，将以循环方式把分区分配给表空间。

- 可以指定被命名的分区：

```
partition by hash (DeptNo)
(partition P1 tablespace P1_TS,
 partition P2 tablespace P2_TS);
```

在这种方法中，在使用一个附加的 lob或varray存储子句选项的情况下，给予每个分区一个名字和一个表空间。由于可以为分区指定有意义的名字，所以这种方法使你更能控制分区的位置。

注意 不能为散列分区创建全局索引。

可以把散列分区与范围分区结合起来使用，从而创建范围分区的散列分区。对于非常大的表，这种组合分区可能是把数据分成可管理和可调整部分的有效方法。

下面的样本用DeptNo列范围划分EMPLOYEE表，并用Name值散列划分DeptNo分区：

```
create table EMPLOYEE (
EmpNo           NUMBER(10) primary key,
Name            VARCHAR2(40),
DeptNo          NUMBER(2),
Salary          NUMBER(7,2),
```

```
Birth_Date      DATE,
Soc_Sec_Num     VARCHAR2(9),
 constraint FK_DeptNO foreign key (DeptNo)
   references DEPT(DeptNo)
)
partition by range (DeptNo)
subpartition by hash (Name)
subpartitions 10
 (partition PART1    values less than (11)
   tablespace PART1_TS,
  partition PART2    values less than (21)
   tablespace PART2_TS,
  partition PART3    values less than (31)
   tablespace PART3_TS,
  partition PART4    values less than (MAXVALUE)
   tablespace PART4_TS);
```

将EMPLOYEE表按范围划分为4个分区。按Name列来散列划分每个分区。

12.1.5 创建显形图

显形图的基本技术与创建和维护快照的技术类似。显形图是一个存储派生数据的表。创建它时，指定用来提供显形图的SQL。

对于大型数据库，显形图可以提供一些性能优势。根据基本 SQL的复杂性，可以提供具有增量变化的显形图(通过一个显形图日志)而不是在数据刷新期间重建它。

与快照不同，显形图可以被优化程序用来动态改变查询的执行路径。即使显形图在查询中没有被命名，这种称作查询重写的特性也可以使优化程序使用显形图来代替被显形图查询的表。例如，如果你有一个大型SALES表，就可以创建一个按区域汇总SALES数据的显形图。如果用户为获得一个区域的汇总SALES数据查询SALAS表，Oracle就可以使这个查询使用显形图来取代SALES表。这样就可以减少对一些最大表的访问量，从而改进系统性能。

若要启动查询重写的显形图，显形图的全部主表都必须处于显形图的模式，并且你必须有QUERY REWRITE系统权限。如果视图和表处于不同的模式，就必须有 GLOBAL QUERY REWRITE系统权限。通常，应按与其所基于的表相同的模式创建显形图，否则就必须管理创建和维护显形图所需要的权限与授权。

与快照一样，显形图创建一个存储数据的局部表和访问该数据的视图。根据显形图的复杂性，Oracle也可能在显形图的局部表上创建一个索引。可以搜索显形图的局部表来改进针对显形图的查询的性能。

若要创建显形图，请使用 create materialized view命令(见附录A中这个命令的全部语法)。下面是对SALES表创建一个显形图的例子：

```
create materialized view SALES_MONTH_MV
tablespace AGG_DATA
refresh complete
start with sysdate
next sysdate+1
enable query rewrite
as
select Sales_Month, SUM(Amount)
  from SALES
 group by Sales_Month;
```

如上面程序清单所示，create materialized view命令指定视图名及其刷新时间表。在这个例子中，选择对视图的完全刷新；每当刷新视图时，就完全删除并重建其数据。对于不是基于聚集的视图，可以结合使用快速刷新和显形图日志向显形图只发送增量变化。 start with和next子句指示 Oracle何时安排刷新数据。如果已启动后台作业进程 (init.ora参数通过JOB_QUEUE_PROCESS)，就可以自动刷新数据。 tablespace子句指示Oracle把显形图的局部表存储在何处。如果合适的话， enable query rewrite子句使优化程序能把对SALES的查询转变成对SALES_ MONTH_MV的查询。

显形图的快速刷新使用显形图日志。显形图日志是一个与显形图的主表存储在一起的表。当主表中的行发生变化时，此变化就被写入显形图日志中。在快速刷新期间，在主表中发生改变的行(通过显形图日志识别)被送到显形图。如果变化少于主表中行的 25%，快速刷新一般要比完全刷新快一些。有关 create materialized view log命令的全部语法情况，请参见附录 A。

12.1.6 创建完全索引表

对于只有很少几列且含有静态数据的表，应考虑完全索引。完全索引一个表就是创建一个索引集，这些索引包含表的所有列，并且每个列至少在一个索引中作为首列。例如，如果STATE表有两个列 —— State_Code和Description，那么就要为完全索引表创建两个索引：一个以State_Code为首列的两列索引和一个以 Description为首列的两列索引。下面的程序清单示出了需要的create index命令：

```
create index STATE_CODE_DESCRIPTION
on STATE(State_Code, Description);

create index STATE_DESCRIPTION_CODE
on STATE(Description, State_Code);
```

在使用where子句对STATE表进行查询，两个索引之中总有一个可以被使用。由于每个索引都包含了表中的全部有效数据，所以就不需要再进行任何表访问。索引含有用户可以从表中查询的所有数据。

完全索引表对代码表特别有用，代码表一般只有很少几个列且数据完全静态。如下节所述，在Oracle8中可以创建索引组织表。

12.1.7 创建和管理索引组织表

索引组织表使其数据根据表的主键列值进行排序。索引组织表就像整个表都存储在索引中一样存储其数据。一个普通索引只将索引列存储在索引中，而索引组织表是将表的所有列都存储在索引中。由于表的数据作为一个索引被存储，所以表的行就没有 RowID。因此，不能从一个索引组织表中选择 RowID伪列值。在Oracle8i之前，不能在表上创建附加索引；唯一有效的索引是主键索引。在 Oracle8i中，可以在一个索引组织表上创建辅助索引。

如下例所示，若要创建一个索引组织表，可使用 create table命令的organization index子句：

```
create table STATE (
State_Code      CHAR(2) primary key,
Description     VARCHAR2(25)
)
organization index;
```

如这个例子所示，若要把 STATE表创建成索引组织表，必须对它创建一个 PRIMARY KEY约束条件。当把表STATE创建成一个索引组织表时，它的数据按排序的顺序存储 (由主键值排序)。

如果总是根据State_Code列(在查询的where子句中)来访问STATE数据，索引组织表就合适。为减少管理索引的工作量，只有当表的数据非常静态时才使用索引组织表。如果表的数据频繁变化，或者需对表中的其他列进行索引，就要使用一个常规表及适当的索引。在大多数情况下，完全索引表可以提高操作性能，但其代价是占据大量空间。

相对于数据存储在常规表的情况，索引组织表所需的空间要少一些。在该索引中，没有RowID值要进行存储(因为没有表)；它所占用的空间也少于含有表中所有列的普通索引所占用的空间。

12.1.8 创建和管理位映射索引

通常，索引创建在选择性很高的那些列上，即，在这些列上的行很少有相同的值。对于一个索引来说，其值只有“ Y ”或“ N ”的列是非常糟糕的，因为该索引只含有两个值，所以通过这个列的任何访问都将返回表的一半记录。不过，如果这些索引列中的值属于一个完全静态的数值组，就应该考虑使用位映射索引。

例如，如果在非常大的表EMPLOYEE中只有很少几个不同的State_Code值，即使在where子句中经常使用这个列，通常也不在 State_Code上创建一个B*tree索引。然而， State_Code列可以利用位映射索引。

位图索引在内部将列的不同值映像到每个记录上。在这个例子中，假设在一个非常大的EMPLOYEE表中只有两个 State_Code值(NH和DE)。由于只有两个 State_Code值，所以State_Code位映射索引只有两个不同的位映射条目，如果表中前 5行的State_Code值为“ DE ”，后面5行的State_Code值为“ NH ”，那么State_Code位映射条目就如下面所示：

```
State_Code bitmaps:
   DE:  < 1 1 1 1 1 0 0 0 0 0 >
   NH:  < 0 0 0 0 0 1 1 1 1 1 >
```

在上面的程序清单中，每一个数字代表EMPLOYEE表中的一个行。由于这里只考虑10行，所以只有10个位映射值示出来。通过读取State_Code的位映射，前5个记录的值为“ DE ”(用1表示)，后5个记录的值则不为“ DE ”(用0表示)。对这个列可能有更多的值。在这种情况下，对应每一个值都将有一个独立的位映射条目。

Oracle优化程序能够在查询进程中动态地将位映射索引内容转换为 RowID。这种转换能力使优化程序可以使用那些有许多不同值的列上的索引 (通过B*tree索引)和那些有很少不同值的列上的索引(通过位映射索引)。

如下所示，通过create index命令中的bitmap子句就可以创建一个位映射索引。必须在索引名中表明它是位映射索引，以便在调整操作时可以简单地辨别出来。

```
create bitmap index EMPLOYEE$STATE_CODE$BMAP
    on EMPLOYEE(State_Code);
```

如果选择使用位映射索引，就要对比查询期间的性能获益和数据处理期间所付出的性能代价。表上的位映射索引越多，每个事务处理所付出的代价就越大。对于一个频繁增加新值的列不应该使用位映射索引。因为每当给 State_Code列增加一个新值，就要为新的 State_Code

值创建一个新位映射。

在创建位映射索引时，Oracle将存储的位映射进行压缩。其结果是位映射索引需要的空间只是正常索引所占空间的5%～10%。因此，对于频繁出现在where子句中的任何非选择性列(假定该列的不同值数量有限制)，应考虑使用位映射索引。如果不断地向列的值清单中添加新值，就必须经常调整位图。

在一个大型数据库中，位映射索引使用在事务表及聚集表的列上时将产生最大的效果。代码表采用完全索引是较好的选择；较大的表能够从位映射索引中受益。对于同一个表，可以既创建位映射索引又创建常规(B*tree)索引。Oracle在查询进程中将对所需的索引进行动态转换。

位映射索引一般不适合于涉及许多插入、更新和删除操作的表。对于具有很少值的一个列，一个位映射索引条目将含有一些对若干RowID的引用，因此在更新位映射的事务处理期间要锁定一些行。锁定位映射索引更新的一些行可能大大降低DML操作的性能。当位映射索引作为频繁查询决策支持系统的一部分使用时，不会受到同样的性能损失。

12.2 事务管理

在一个大型数据库中，批量数据装载通常会引起大型的事务处理。原因很简单：用计算机收集和插入数据肯定比人工要快一些。假定有一个处理数据效率非常高的操作员，他每分钟可以向数据库中输入两个采购单记录。按这种速度，每小时可向系统插入120个记录，每天可插入960个记录。

现在来与批量装载进程所做的工作进行一个对比，使用SQL*Loader Direct Path进行装载，不到1秒钟的时间就可以插入1000个记录。因此，一个非常熟练的高效率操作员一天的工作在不到一秒的时间内就可以由机器来完成。从某种意义上讲，操作员最好是不进行数据输入而将精力转到分析数据趋势一类的工作上。所有数据的收集和装载过程不需要任何人输入任何命令就能够完成，其速度比任何人工操作系统要快得多。

大型数据库中可能有一些小型事务处理，但其中的大多数都是由批操作产生。因此，当管理大型数据库中的事务时，应特别注意包含装载和聚集数据的批事务。

批事务的时间安排极为关键。如前面第7章所述，在处理小型联机事务的同时执行大型批量操作是引起数据库中回滚段问题的常见原因。另外，批处理(例如SQL*Loader Direct Path装载或Oracle8i OCI直接装载)使用直接装载选项将使索引处于装载状态。当索引处于装载状态时，将无法对表进行联机更新。

理想情况是，在没有联机处理出现时执行批量装载。批量装载事务的独立程度越高，其成功的可能性就越大。

12.2.1 配置批事务环境

在下面几小节中，将提供向数据库的大表中装载数据或从中删除数据的建议。在执行大型批量事务之前，应首先创建一个能够支持该事务的环境。

1. 创建专用回滚段

与那些由联机用户输入的事务所使用的回滚段相比，批事务使用的回滚段有不同的特征。批量系统拥有少量的较大事务而不是大量的小型事务。因此，支持批事务的回滚段数量上一

般比支持联机用户事务的回滚段要少，但在盘区容量上就要大一些。例如，可能需要总共 10个有20个盘区的回滚段来支持联机用户，但是只需要一个有 10个盘区的大型回滚段来支持批事务。

若要强制一个事务使用一个特殊的回滚段，可以使用 SQL*Plus中的下列命令：

```
set transaction use rollback segment SEGMENT_NAME;
```

即使在会话中还没有输入任何事务，该命令也应该紧跟在一个 commit命令后面。还需要给用于事务处理的回滚段取一个名字。如果正在处理多个数据库，还应给批事务回滚段起一个标准化的名字以简化装载进程。例如，可以将每一个数据库的批数据装载回滚段取名为ROLL_BATCH。如果使用一个一致的命名标准，在把数据装载程序从一个数据库转移到另一个数据库时，就不需要对该程序进行改变。

2. 禁止重做日志归档

对联机重做日志文件的内容进行归档，就可以在介质失败时进行恢复。然而，要考虑在批装载期间写入联机重做日志文件的事务：它们是数据装载所引起的 insert语句。如果可以通过执行数据装载来完全重建这些 insert语句，就不需要事务的归档处理。

例如，如果已经在 ARCHIVELOG(归档)模式下运行，在数据装载前 (称为时间T1)必须先关闭数据库；以NOARCHIVELOG(非归档)模式打开数据库(称为时间T2)；执行数据装载直到完成(称为时间T3)为止。然后，关闭数据库并以 ARCHVELOG模式重新打开它。如果在时间T1前出现介质失败，可以使用归档重做日志文件恢复数据。如果失败出现在时间 T2和T3之间，可以进行如下恢复操作：恢复到时间 T1并且重新执行数据装载。在时间 T3出现失败时，可执行数据库的一个新备份。

如果在数据装载时关闭重做日志文件的归档功能，必须确保此时没有其他事务出现。如果同时出现其他事务，那么在恢复时这些事务的数据将要丢失。如下节所述，对于 Oracle8，可以使用nologging参数来避免对特定表或表的一部分进行事务记录。

3. 禁止大型数据表事务记录

在Oracle7.2版本中，一个新关键字 unrecoverable(不可恢复)被加入到create table as select和create index命令中。关键字 unrecoverable禁止在命令执行时向联机重做日志文件写入。在Oracle8中，unrecoverable被应用更广泛的关键字 nologging(非记录)所代替。

使用nologging关键字创建表时，最初提供给表的事务都不写入联机重做日志文件中。另外，后续的SQL*Loader Direct Path装载和使用 APPEND提示的任何 insert命令都不把重做日志条目写入联机重做日志文件中。因此，可以指定一些特定表 (例如大型事务表或聚集表)为不记录。有了 nologging关键字，就可以不把这些事务写入联机重做日志文件，从而使整个数据库都运行在 ARCHIVELOG模式下，但是最大的表还是运行在 NOARCHIVELOG模式下。也可以规定nologging用于表中的LOB部分，这部分采用 BLOB或CLOB数据类型。

4. 数据装载后重建索引

高效率索引是在大型数据库中快速进行数据访问的关键。 Oracle不能完善地管理索引中的数据，所以必须定期重建索引。每次大量装载数据都必须重建索引。可以使用 alter index命令的rebuild子句创建一个以旧索引作为其数据源的新索引。

即使使用 SQL*Loader Direct Path选项(在数据装载期间使索引开着)也必须周期性地重建索引。如果不使用这个装载选项，就要在数据装载前先撤消索引，待数据全部装载完毕后再

重建索引。

在进行大量删除或更新后也应重建索引。索引组织得越好，与之相关的数据访问就越快。

12.2.2 装载数据

在把数据从文件装载到事务表时，应设法消除那些降低插入速度的因素。应禁止表上的约束条件并禁止表上的任何触发器(尽管批装载的表一般不应有触发器)；应在数据装载前删除索引。如果不删除索引且数据在装载前已被排序，就可以使用 SQL*Loader Direct Path选项。除了管理索引外，SQL*Loader Direct Path选项允许一次插入整个数据块而不是插入一行。如果通过Direct Path装载而被装入的分区或表被标记为 nologging，装载的数据块就不写入重做日志文件中。

在正常插入一个行时，Oracle检查表中自由块的列表——这些新块中拥有比pctfree空间更大的自由空间。Oracle找到第一个可以容纳记录的块并在此插入一个记录。对于下一个记录，Oracle再次查找空余的空间。每一个记录都要重复这种查找过程。通过一次插入一个整块，SQL*Loader Direct Path就可以避免这些查找开销。

为确定到底在哪里装载数据，SQL*Loader首先确定表的高水位标志。高水位标志是在表中曾容纳数据的最大编号的数据块。例如，如果把 1000块的行装载到表中，随后删除这些行，高水位标志将指向块号 1000。在SQL*Loader Direct Path插入期间，Oracle不搜索当前使用块中的打开空间。因此，它将高水位标志后的第一个数据块作为装载数据的开始块。即使高水位标志前有空余空间，SQL*Loader Direct Path也不会使用它们。

为一个表重新设置高水位标志的方法有两种：删除并重建表或者截断表。因此需要了解从一个表中删除记录的方法。如果在一个表中装载 1000块的行并且随后对其进行删除，此时高水位标志就不发生变化。后面的 SQL*Loader Direct Path再装载同样多的数据时要从第 1001块开始；现在表将使用 2000块，而不是使用1000块。

除了使你能使用效率极高的 Direct Path选项外，SQL*Loader Direct Path还有并行操作和禁止记录 (unlogging)操作的选项。本章后面 12.4节“调整”将详述并行操作的功能。有关nologging参数的信息，请参见本章前面“禁止大型数据表事务记录”小节。

12.2.3 插入数据

如果要插入的数据来自另一个表 (如用于提供聚集表的 insert as select)，就可以利用Oracle8中提供的一个新提示。这个新提示叫 APPEND(追加)，它使用高水位标志作为插入数据块的基础，SQL*Loader Direct Path也采用同样的方法。

APPEND提示指示Oracle去查找表中曾插入的最后一块数据，新记录将在高水位标志后面的一块开始插入。

例如，如果一个表已在数据库中使用了 20块，那么使用APPEND提示的insert命令将其数据写入第21块。由于数据被写入表的新块中，因此在 insert期间没有什么数据库的空间管理工作。所以，使用APPEND提示时，insert操作可能完成得较快一些。由于高水位标志下的空间未被使用，所以表的空间需求可能会增加。

可以在insert命令中指定APPEND提示，提示好像是一个注释，它以“/*”开始并以“*/”结束。语法中的唯一不同之处是，开始的字符集在提示名字前要加上一个“+”。下面示出的

insert命令将其数据添加到表上。

```
insert /*+ APPEND */ into SALES_PERIOD_CUST_AGG
select Period_ID, Customer_ID, SUM(Sales)
  from SALES
 group by Period_ID, Customer_ID;
```

SALES事务表的记录将插入到SALES_PERIOD_CUST_AGG聚集表中。不是试图重新使用SALES_PERIOD_CUST_AGG表中以前使用过的空间，而是将新记录放在表的物理存储空间的最后。

由于新记录不打算重新使用表以前使用过的空间，所以SALES_PERIOD_CUST_AGG表将需要更多的空间。通常，只有在向没有可重用空间的表中插入大容量数据时才使用APPEND提示。

由于聚集表的数据源是一个存储在数据库中其他地方的表，所以APPEND提示是创建聚集表的理想方式。因为聚集表存储冗余数据，所以它还适于应用本章前面讨论的nologging参数。如果装载的表具有nologging参数，通过APPEND提示插入的行并不生成重做日志内容。

注意 有些买来的应用程序被设计成与数据库无关，不能将SQL*Loader用于数据装载。如果应用程序提供执行装载的一个可执行文件，请与销售商或应用程序支援小组人员一起检查，以确保在插入期间执行数组处理。避免逐行插入，甚至是在批方式中。

12.2.4 删除数据

当管理大量数据时，应该设法使创建的表能够使用truncate命令。如果数据全部存储在一个独立的大表中，由于可以通过本章前面介绍的alter table命令截断分区，所以应考虑使用分区。如果需要通过delete命令同时删除大量数据，就需要或是设置一个支持大事务的环境，或是使用一种过程方法将事务分成更小的事务。

1. 设置环境

大量删除操作所需的环境与大量插入操作一样；需要创建并维持一个足以支持事务的回滚段。为迫使这个回滚段由这个事务使用，可在一个commit命令之后立即使用set transaction use rollback segment命令。应该将大量的删除操作安排在数据库中很少有其他事务出现时进行，以避免可能出现的并发读取问题。

2. 使用过程方法

可以使用PL/SQL将一个删除操作分为多个事务进行。可以创建一个PC/SQL块，当输入一个delete命令及每批提交的记录数后，这个PL/SQL块就开始执行。例如，如果有1000000个记录要被删除，又不能使用truncate命令，就可以在每删除1000个记录后进行一次提交。为此，需要使用动态PL/SQL和循环。在下面的PL/SQL过程(由Oracle技术支持开发并分发)中，两个输入参数分别是SQL语句和每批提交的记录个数。例如，假若delete命令是：

```
delete from SALES where Customer_ID=12;
```

并且希望在每删除1000个记录后进行一次提交，将用两个参数调用DELETE_COMMIT过程，如下面所示：

```
execute DELETE_COMMIT('delete from SALES where Customer_ID=12',1000);
```

如果where子句中的值是字符串，需要用双引号将其括起来：

```
execute DELETE_COMMIT('delete from SALES where State_Code = ''NH''',500)
```

DELETE_COMMIT过程的代码如下：

```
create or replace procedure DELETE_COMMIT
( p_statement in varchar2,
  p_commit_batch_size   in number default 10000)
is
        cid                             integer;
        changed_statement               varchar2(2000);
        finished                        boolean;
        nofrows                         integer;
        lrowid                          rowid;
        rowcnt                          integer;
        errpsn                          integer;
         sqlfcd                          integer;
         errc                            integer;
         errm                            varchar2(2000);
begin
         /* If the actual statement contains a WHERE clause, then
           append a rownum < n clause after that using AND, else
           use WHERE rownum < n clause */
         if ( upper(p_statement) like '% WHERE %') then
                 changed_statement := p_statement||' AND rownum < '
                ||to_char(p_commit_batch_size + 1);
         else
changed_statement := p_statement||' WHERE rownum < '
||to_char(p_commit_batch_size + 1);
         end if;
         begin
  cid := dbms_sql.open_cursor; -- Open a cursor for the task
                 dbms_sql.parse(cid,changed_statement, dbms_sql.native);
                        -- parse the cursor.
  rowcnt := dbms_sql.last_row_count;
                       -- store for some future reporting
         exception
            when others then
                       errpsn := dbms_sql.last_error_position;
                          -- gives the error position in the changed sql
                          -- delete statement if anything happens
     sqlfcd := dbms_sql.last_sql_function_code;
                          -- function code can be found in the OCI manual
                       lrowid := dbms_sql.last_row_id;
                          -- store all these values for error reporting.
                          -- However all these are really useful in a
                          -- stand-alone proc execution for dbms_output
                          -- to be successful, not possible when called
                          -- from a form or front-end tool.
                       errc := SQLCODE;
                       errm := SQLERRM;
                       dbms_output.put_line('Error '||to_char(errc)||
                             ' Posn '||to_char(errpsn)||
               ' SQL fCode '||to_char(sqlfcd)||
         ' rowid '||rowidtochar(lrowid));
                     raise_application_error(-20000,errm);
                       -- this will ensure the display of at least
                       -- the error message if something happens,
                       -- even in a front-end tool.
         end;
         finished := FALSE;
         while not (finished)
         loop -- keep on executing the cursor till there is no more
              -- to process.
```

```
                begin
nofrows := dbms_sql.execute(cid);
                        rowcnt := dbms_sql.last_row_count;
                exception
                        when others then
                                errpsn := dbms_sql.last_error_position;
                                sqlfcd := dbms_sql.last_sql_function_code;
                                lrowid := dbms_sql.last_row_id;
                                  errc := SQLCODE;
                                  errm := SQLERRM;
                             dbms_output.put_line('Error '||to_char(errc)||
                                    ' Posn '||to_char(errpsn)||
                         ' SQL fCode '||to_char(sqlfcd)||
          ' rowid '||rowidtochar(lrowid));
                                raise_application_error(-20000,errm);
                end;
                if nofrows = 0 then
                        finished := TRUE;
                else
                 finished := FALSE;
                end if;
                commit;
        end loop;
        begin
                dbms_sql.close_cursor(cid);
                        -- close the cursor for a clean finish
        exception
                when others then
                        errpsn := dbms_sql.last_error_position;
                        sqlfcd := dbms_sql.last_sql_function_code;
                        lrowid := dbms_sql.last_row_id;
     errc := SQLCODE;
                        errm := SQLERRM;
       dbms_output.put_line('Error '||to_char(errc)||
      ' Posn '||to_char(errpsn)||
                                ' SQL fCode '||to_char(sqlfcd)||
                                ' rowid '||rowidtochar(lrowid));
                        raise_application_error(-20000,errm);
        end;
end;
/
```

DELETE_COMMIT过程中的大部分代码用来处理语句执行过程中出现的意外情况。理论上讲，可执行命令应按如下逻辑进行：如果 delete命令已含有一个 where子句，就给该语句添加一个and子句，否则就追加一个 where子句，这些子句用于限制一次删除的记录个数；处理并为指定数目的记录提交删除，再执行一次 delete操作，第二个记录集将被删除，重复执行delete操作直到不再有与where子句匹配的记录为止。

可能出现的意外情况记录在过程中，一般很少出现。最好的情况是，确保 delete命令的where子句可以使用索引。在执行delete命令前，可使用explain plan命令确定delete命令是否使用索引。有关explain plan命令的信息，请参见第8章。

12.3 备份

是什么困扰着大型数据库的备份呢？

对于数据库管理人员来说，这个问题似乎问得有点古怪。但它却是个切合实际的问题。

根据定义，不能在18小时内恢复一个大型数据库。重新装载数据库和重建聚集表所用的时间比使用Oracle备份和恢复工具来恢复系统数据所用的时间要少一些。如果重新装载数据比恢复数据要快，那么还需要备份数据吗？答案取决于数据装载进程出现的方式。

12.3.1 备份需求及策略评估

如本章前面所述，大多数大型数据库有四种类型的表：

- 大型事务表，含有数据库中大部分原始数据。
- 聚集表，存储从事务表中聚集来的数据。
- 代码表。
- 临时工作表。

在估计数据库的备份需求时，应估计每种类型的表的备份需求。

事务表的备份需求由所使用的数据装载处理方法确定。如果每个数据装载都能完全替换事务数据，就可以使用数据装载进程来恢复数据，而不必依赖于 Oracle的工具(例如Export)。如果事务数据不能完全由每个数据装载来代替，则可以使用一套备份方法来恢复数据。

例如，如果每个数据装载只包含 SALES表中一个时期的数据，就必须备份前面各个阶段的数据及当前时期的数据。有两种方法实现这个目的。

1) 导出旧数据并使用数据装载进程来重建当前时期的数据。

2) 保存旧数据文件并且在恢复期间分别执行每一个时期的数据装载。

根据数据装载步骤，第二种选项使恢复操作完成得更快一些。可以一直使用 Export工具作为数据恢复的备份方法。

如果事务表中的数据在装载之后被修改，就必须能够重建这些事务。既可以在每一个事务后执行Export操作，也可以在ARCHIVELOG方式下运行数据库。如果依赖于归档重做日志文件重建事务数据，就不能将事务表设置为 nologging方式。在系统备份期间还可以使用RMAN(见第10章)将数据备份的数量降至最低限度。

理想的方案是，事务表在每次数据装载过程中被完全重新装载，并且随后没有任何更新。若需要恢复事务表，只需简单地执行装载过程并将该表保持在 nologging方式中。作为一种附加的备份方法，可以在每一次数据装载后执行导出操作。如果在数据装载后能对数据进行修改，或者是每次装载不能构成表中的全部数据，就必须备份用户事务或历史数据。

聚集表存储冗余数据。聚集表中的所有数据都能够通过重新运行创建聚集表的命令来生成。如果丢失了一个聚集表(例如，被偶然删除)，可以通过执行create table as select命令来重建这个表。聚集表创建后就不允许对它进行修改。如果需要修改数据，应在聚集表的数据源——事务表——中进行修改，随后应重建聚集表。

根据聚集表的这些特点，对聚集表就不需要使用 ARCHIVELOG模式进行备份。可以在表创建后导出数据，但是通过 SQL重建表比通过导入重建表的速度一般要快一些。如果使用本章前面所讲的insert命令的APPEND提示，性能差异最大。

代码表存储静态数据，在代码表中应该很少出现事务。因此导出对代码表来说是非常有效的备份方式。由于代码表中很少出现事务，所以就不必使用归档重做日志文件来恢复它们。如果正确安排导出时间，在数据恢复时使用导出操作所备份的文件就不会丢失数据。

临时工作表在数据装载过程中使用并且一般不是作为产品应用程序的一部分被访问。因

此，临时工作表的恢复需求只是数据装载进程的一部分。例如，可以在数据装载的不同点备份临时工作表，以减少数据装载失败时所需的恢复行动。由于临时工作表中没有联机事务发生，所以通常就采用导出操作来备份这些表。

12.3.2 备份方案

假如有一个大型数据库，它的事务表在每次装载后就完全被重新装载 (没有后续事务)并且代码表保持不变。合适的备份方案是什么呢？对于这样的系统，根本不需要使用ARCHIVELOG模式，实际上可以使许多表都处于nologging状态。备份策略可能是：

1) 数据装载后导出事务表；恢复时主要依赖于数据装载。

2) 聚集表创建后导出聚集表，恢复时主要依赖于表重建。

3) 在代码表较大变动后将其导出。如果代码表非常小，也可以通过 create table as select命令对其进行复制，恢复时主要依赖于导出操作。

4) 在数据装载进程的每一步之后都要导出临时工作表。恢复主要依赖于数据装载进程。

尽管每种表类型都依赖于导出操作，但在数据处理的不同时间进行导出操作。只有一种表类型(代码表)将导出操作作为其主要恢复方法。

这种备份策略很可能与其他联机事务系统上使用的备份策略不一致。但是，无论使用什么备份方法都有一些主要相似之处。

1) 每个表都有一个主要备份方法和一个辅助备份方法。例如，如果使用 create table as select命令无法恢复聚集表时，可以使用这个表的导出操作作为辅助恢复方法。不能只依赖一种备份方法。

2) 每一种恢复类型都应彻底进行测试，如果没有对一个恢复操作进行测试，就不能轻易相信其功能。

如果遵循这些准则，即使只依靠导出操作作为代码表的备份，也可以恢复大型数据库。调整好数据装载进程，就可以减少数据库恢复所需的时间，从而在管理上会有更大的灵活性。

12.4 调整

调整一个大型数据库有两个过程：首先是调整环境，然后是调整给数据库施加巨大负载的查询和事务。数据库环境的调整在第 8章中已进行了讨论，本章前面也提供了事务调整的提示。例如，数据装载进程应使用 SQL*Loader Direct Path中的块插入方法和 insert命令中的APPEND提示。

对于一个大型数据库， System Global Area(SGA)的数据块缓冲区部分大约为数据库总容量的2%，对于一个100GB的数据库来说，其SGA为2GB。一个足够大的SGA意指数据库运行在能够支持大型内存区域管理的主机和操作系统上。

创建数据库时，应将数据块大小设置为操作系统上 Oracle所支持的最大值。数据块容量越大，数据的存储效率就越高。当管理一个小型数据库时，使用较大的数据块容量所带来的存储效率及数据访问性能的改进是极为明显的。管理大型数据库时，存储效率及数据访问性能的改善是实质性的。对数据块的容量加倍后，大多数批量操作都可以提高 40%的效率。

注意 数据库创建后就不能再改变数据库的块容量。数据库数据块的容量由创建数据库时所使用的init.ora文件中的DB_BLOCK_SIZE值决定。

如本章12.1.4节“分区”所述，对大型数据库的主表应采用分区的方法。通过对表进行分区，就可以改善管理表的能力。例如，可以根据主表的值，使用分区来跨多个较小表分布表访问所产生的I/O负载。可以截断或修改表的一个分区而不影响表的其他部分。在管理索引分区时，使用局部索引可得到最大的灵活性。也可以使用显形图自动进行聚集处理并激活优化程序中的查询重写能力。

在数据库外部，可以利用设备及磁盘存储结构来恰当地分布系统 I/O需求。有关磁盘选项和结构(例如RAID设备和磁盘镜像)的信息，请参见本章前面的12.1.3节“选择物理设计”。

如果主机有多个可以使用的CPU，就可以充分利用Oracle的Parallel Query Option(PQO)。当使用PQO时，可创建多个进程来完成一个任务(例如查询或索引创建)。由于涉及到大型事务处理及排序，所以大型数据库可以从使用PQO中获益。有关PQO的使用情况，请参见第8章。

调整大型表查询

除了创建完全索引表、创建位映射索引、分区表和索引表、使用 PQO等方法对大型表进行查询外，还可以进一步调整大型表查询来减少对数据库其他部分的影响。尽管多个用户都可以通过共享SGA中小表的数据获益，但是当访问超大型表时，这种受益就消失殆尽。对于超大型表，索引访问可能会对数据库的其他部分产生负作用。

当表和表的索引比较小时，在SGA中的数据共享程度就比较高。多个用户对表进行读或索引范围搜索时可以重复使用相同数据块。重复使用 SGA数据块的结果是命中率(SGA中数据块的重复使用度量)增加。

当一个表容量增加时，表索引也随之增大。如果表及其索引大到 SGA的有效空间容纳不下时，基本上不可能在SGA中查找到范围搜索所需的下一行。 SGA数据块缓冲存储区中数据的重用能力将减弱，数据库的命中率也随之降低。最终，每一个逻辑读都需要一个单独的物理读。

SGA用来最大化从数据文件中读取的数据块的重用能力(在多个用户之间)。为达到这一目的，SGA保持一个已经被读过的数据块列表；如果数据块由 RowID通过索引访问或通过表访问来读取，它们在SGA中的保存时间就最长。如果数据块通过全表搜索读入 SGA中，当数据块缓冲区需要更多空间时，首先将这些数据块从SGA中移走。

对于具有小表的应用程序， SGA中的数据块缓冲区管理使数据块的重用性最大化并增加命中率。然而，如果对一个大型表进行索引范围搜索的话，会发生什么情况呢？即使可能没有其他用户能够使用索引块中的值，索引块还是要在 SGA中保存较长时间。由于索引很大，所以索引中的许多数据块都被读取，从而占据了SGA数据块缓冲区中相当大一部分可用空间，更大的空间将被RowID访问的表块占据，而这些块将来可能很少再次使用。令人啼笑皆非的是，命中率开始下降，其原因是正在执行索引搜索。因此，对于超大型表，其调整方法要集中在特殊的索引技术和对索引的可选择性上。

1. 管理临近数据

在访问超大型表时，如果要连续使用索引，必须考虑数据的临近性 —— 逻辑相关记录之间的物理关系。若要最大化数据临近性，按表中范围搜索常用的列进行排序，按顺序将记录插入表中。例如，大型SALES表的主键是由Period_ID、Customer_ID和Sale_No组成的一个集合。使用 Period_ID作为限制条件的访问将能使用主键上的唯一索引，但是如果通常对

SALES.Customer_ID列进行范围搜索，就应按Customer_ID顺序存储。

如果数据以排序格式存储，那么在

```
where Customer_ID between 123 and 241
```

之类的范围查找中，最有可能重复使用读入SGA的表和索引块，这是因为所有值为123的Customer_ID值将存储在一起。更少的索引和表块读入到SGA数据块缓冲区中，从而大大降低了索引范围搜索对SGA的影响。按顺序存储数据有助于范围搜索(无论表的容量)，但是对于对大型范围搜索具有负面影响的大型表来说，这是非常需要的。

2. 避免无益的索引搜索

如果准备使用一个索引对一个大型表进行搜索，就不能假定索引搜索比全表搜索执行得更快。不紧随表访问的索引唯一性搜索或范围搜索可以进行得很好；但是紧随表访问(通过RowID)的索引范围搜索却进行得非常糟糕。当一个表增大到明显大于数据块缓冲区时，索引搜索与全表搜索之间的等效点就开始下降，如果要从一个有10000000个记录的表中读取多于1%的记录，最好还是采用全表搜索而不是采用索引范围搜索与RowID表访问的组合。

由于Oracle管理SGA的数据块缓冲存储区，所以全表搜索可以进行得较好。如果在一个大型表中执行一个大型索引搜索，就有许多索引块存储在SGA中，并且将在SGA中保存尽可能长的时间。比较而言，由全表搜索读取的块将尽快从SGA中清除。因此，如果SGA中还有其他对象的数据块，全表搜索将不会影响它们，而基于索引的访问则要把它们从缓存区中清除。

全表搜索只有少量的数据块保存在SGA中。这些块是全表搜索最后读取的数据块，这些数据块的数量通过设置init.ora参数DB_FILE_MULTIBLOCK_READ_COUNT决定。全表搜索数据块使用的缓存区大小由DB_FILE_MULTIBLOCK_READ_COUNT参数和DB_BLOCK_SIZE参数共同决定。

在Oracle8i中，也可以指定多个缓冲池并把对象赋予这些缓冲池。例如，可以创建一个RECYCLE池，将最大的表赋予这个池。使用RECYCLE池将把搜索大型表对SGA中内存管理造成的影响减至最小。

全表搜索不是一种典型的调整方法。然而，如果一个表大到了基于索引的访问将覆盖SGA时，应考虑在多用户环境中使用全表搜索来查询超大型表。为改善全表搜索的性能，可考虑并行操作。

12.5 使用可迁移表空间

Oracle采用了大量改进数据移动操作性能的特征。如本章前面几节所述，这些数据移动特征包括：

- 供批量删除用的truncate命令。
- 供插入用的APPEND提示和SQL*Loader Direct Path装载。
- 用于将表和索引分成更小数据集的分区。
- nologging操作。

在Oracle8i中，可以使用另一个新特征——可迁移表空间——来改善数据移动操作的性能。可迁移表空间应改善数据库之间大量移动数据的操作性能。可以移动数据和索引，但不能移动位映射索引或含有收集程序的表(嵌套表和可变数组)。

若要移动表空间，必须生成一个表空间集，将这个表空间集移动到新数据库，并把它插

入新数据库。下节，将介绍要执行的步骤及实现提示。数据库应在同一个操作系统上，具有相同版本的Oracle、数据库块大小和字符集。

12.5.1 生成可迁移的表空间集

除了为表空间导出元数据外，可迁移表空间集还含有被移动的表空间的全部数据文件。被移动的表空间应是独立的表空间，不应含有任何依赖这个表空间外部对象的对象。例如，如果要移动一个表，还必须移动含有该表索引的的表空间。表空间中的对象组织和分布得越好(见第3章)，要移动的独立表空间集就越容易生成。

可以选择是否包括引用完整性约束条件作为可迁移表空间集的一部分。如果选择使用引用完整性约束条件，可迁移表空间集将增加，以便包括保持关键字关系所需要的一些表。引用完整性是可选项，因为在多个数据库中可以有相同的代码表。例如，你可能正计划把一个表空间从测试数据库移动到产品数据库。如果在测试数据库中有一个 COUNTRY表，就可能已经有一个恒等的 COUNTRY表在产品数据库中。由于代码表在两个数据库中恒等，所以不必移动这部分引用完整性约束条件。可以移动该表空间，然后在目标数据库中重新启用引用完整性；从而简化可迁移表空间集的创建。

若要确定表空间集是否独立，可执行 DBMS_TTS软件包中的TRANSPORT_SET_CHECK过程。如果要考虑引用完整性约束条件，这个过程采用两个输入参数：表空间集和设置为TRUE的Boolean标记。在下面的例子中，没有为 AGG_DATA和AGG_INDEXES表空间的移动考虑引用完整性约束条件：

```
execute DBMS_TTS.TRANSPORT_SET_CHECK('AGG_DATA,AGG_INDEXES','FALSE');
```

如果在规定的集中有任何独立违规现象， Oracle将提供TRANSPORT_SET_VIOLATIONS数据字典视图。如果没有违规现象，该视图将是空白。

一旦选择了一个独立的表空间集，就要使这些表空间为只读方式。例如：

```
alter tablespace AGG_DATA read only;
alter tablespace AGG_INDEXES read only;
```

接着，使用TRANSPORT_TABLESPACE和TABLESPACE参数导出表空间的元数据：

```
exp TRANSPORT_TABLESPACE=Y TABLESPACES=(AGG_DATA,AGG_INDEXES) CONSTRAINTS=N  GRANTS=Y TRIGGERS=N
```

如例子中所示，可以指定是否一起导出触发器、约束条件和授权与表空间元数据。还应注意在可迁移表空间集中拥有对象的帐户名称。现在可以把表空间的数据文件复制到一个独立的区域。如果需要的话，可以把其当前数据库中的这些表空间设为读 /写方式。在生成可迁移表空间集后，就能将其文件(其中包括导出)移到目标数据库可以访问的一个区域上。

12.5.2 插入可迁移的表空间集

一旦可迁移的表空间集移到了对目标数据库可访问的一个区域，就能把该集插入目标数据库中。首先，使用Import来导入被导出的元数据：

```
imp TRANSPORT_TABLESPACE=Y DATAFILES=(agg_data.dbf, agg_indexes.dbf)
```

导入时，指定属于可迁移表空间集部分的数据文件。可以有选择地指定表空间 (通过TABLESPACES参数)和对象拥有者(通过OWNERS参数)。

导入操作完成后，可迁移表空间集中的所有表空间都设为只读方式中。可以在目标数据库中发出alter tablespace read write命令来把新表空间设为读/写方式中。

```
alter tablespace AGG_DATA read write;
alter tablespace AGG_INDEXES read write;
```

这里要注意的是，不能改变被移动对象的拥有关系。

可迁移表空间支持非常快速地移动大型数据集。在一个数据仓库中，可以使用可迁移表空间把聚集表从全局数据仓库发到数据市场，或者从数据市场发到全局数据仓库。任何只读数据都能被迅速地分配到多个数据库，可以发送数据文件和导出的元数据而不是发送 SQL脚本文件。这种经过修改的数据传送处理，可以大大简化管理远程数据库、远程数据市场和大型数据传送操作的过程。

12.6 局部管理的表空间

在Oracle8i中，可以用两种不同的方法跟踪一个表空间中的自由空间和已用空间。第一种方法是通过数据字典处理盘区管理(字典管理的表空间)，这种方法在表空间出现后就可采用而且是缺省行为。在字典管理的表空间中，每当为表空间的重用而分配或释放一个盘区时，数据字典表中就更新一个合适的条目。每次更新字典表时还要存储回滚信息。字典表和回滚段是数据库的一部分，应用于其他数据操作的规则也应用于它们。因此，获得并释放盘区时，就会出现递归空间管理操作。

第二种方法用于Oracle8i，用于在表空间内部管理盘区(局部管理的表空间)。在局部管理的表空间中，表空间通过维护每个数据文件中的自由块和已用块或者数据文件中的块集合的位映射，来管理其自己的空间。每当为重新使用而分配或释放一个盘区时， Oracle就更新位映射以显示新的状态。

使用局部管理的表空间时，不更新字典，不产成回滚活动。局部管理的表空间自动跟踪相邻的自由空间，因此不需要合并盘区。在一个局部管理的表空间中，全部盘区都可以有相同的尺寸，或者系统能自动决定盘区的大小。

若要使用这种功能部件，必须为 create tablespace命令中的extent management子句指定local选项。这里示出了声明局部管理的表空间的一个 create tablespace命令例子：

```
create tablespace CODES_TABLES
datafile '/u01/oracle/VLDB/codes_tables.dbf'
size 10M
extent_management local uniform size 256K;
```

假定创建这个表空间的数据库块尺寸为 4KB，在这个例子中，表空间用表示局部管理的local关键字和256KB的统一尺寸创建。位映射中的每一位都描述 64块(256/4)。如果uniform size子句被省略，缺省是autoallocate。uniform的缺省尺寸为1MB。

注意 如果指定create tablespace命令中的local，就不能指定 default storage子句、minextents或temporary。如果使用create temporary tablespace命令创建表空间，就可以指定extent_management local。

在SYSTEM表空间的情况下，可以用 create database语句中的local表示局部管理。如果创建局部管理的 SYSTEM表空间，所有回滚段就必须在局部管理的表空间中创建。不过，数据库中的任何其他表空间都可以有字典管理的盘区。使用局部盘区管理并指定 local size，就可以确保有效地重用表空间中的空间。

随着盘区中数据库对象增加，局部管理的表空间就显得更加重要。如第 4章所述，由于涉及到数据字典管理问题，在具有数千个盘区的表上执行 DDL空间管理操作就不太合适。如果使用局部管理的表空间，就会大大减少性能损失。