THE EXPERT'S VOICE® IN ORACLE

Oracle Database 11g Performance Tuning Recipes

A Problem-Solution Approach

TESTED SOLUTIONS TO COMMON ORACLE DATABASE PERFORMANCE PROBLEMS

Sam R. Alapati, Darl Kuhn, and Bill Padfield

Apress®

Contents at a Glance

About the Authors xvi

About the Technical Reviewer xvii

Acknowledgments xviii

Chapter 1: Optimizing Table Performance 1

Chapter 2: Choosing and Optimizing Indexes 43

Chapter 3: Optimizing Instance Memory 83

Chapter 4: Monitoring System Performance 113

Chapter 5: Minimizing System Contention 147

Chapter 6: Analyzing Operating System Performance 185

Chapter 7: Troubleshooting the Database 209

Chapter 8: Creating Efficient SQL 253

Chapter 9: Manually Tuning SQL 299

Chapter 10: Tracing SQL Execution 327

Chapter 11: Automated SQL Tuning 367

Chapter 12: Execution Plan Optimization and Consistency 409

Chapter 13: Configuring the Optimizer 447

Chapter 14: Implementing Query Hints 491

Chapter 15: Executing SQL in Parallel 525

Index 555

CHAPTER 1

Optimizing Table Performance

This chapter details database features that impact the performance of storing and retrieving data within a table. Table performance is partially determined by database characteristics implemented prior to creating tables. For example, the physical storage features implemented when first creating a database and associated tablespaces subsequently influence the performance of tables. Similarly, performance is also impacted by your choice of initial physical features such as table types and data types. Therefore implementing practical database, tablespace, and table creation standards (with performance in mind) forms the foundation for optimizing data availability and scalability.

An *Oracle database* is comprised of the physical structures used to store, manage, secure, and retrieve data. When first building a database, there are several performance-related features that you can implement at the time of database creation. For example, the initial layout of the datafiles and the type of tablespace management are specified upon creation. Architectural decisions instantiated at this point often have long-lasting implications.

A *tablespace* is the logical structure that allows you to manage a group of datafiles. Datafiles are the physical datafiles on disk. When configuring tablespaces, there are several features to be aware of that can have far-reaching performance implications, namely locally managed tablespaces and automatic segment storage–managed tablespaces. When you reasonably implement these features, you maximize your ability to obtain acceptable future table performance.

The *table* is the object that stores data in a database. Database performance is a measure of the speed at which an application is able to insert, update, delete, and select data. Therefore it's appropriate that we begin this book with recipes that provide solutions regarding problems related to table performance.

We start by describing aspects of database and tablespace creation that impact table performance. We next move on to topics such as choosing table types and data types that meet performance-related business requirements. Later topics include managing the physical implementation of tablespace usage. We detail issues such as detecting table fragmentation, dealing with free space under the high-water mark, row chaining, and compressing data. Also described is the Oracle Segment Advisor. This handy tool helps you with automating the detection and resolution of table fragmentation and unused space.

1-1. Building a Database That Maximizes Performance

Problem

You realize when initially creating a database that some features (when enabled) have long-lasting ramifications for table performance and availability. Specifically, when creating the database, you want to do the following:

- Enforce that every tablespace ever created in the database must be locally managed. Locally managed tablespaces deliver better performance than the deprecated dictionary-managed technology.
- Ensure users are automatically assigned a default permanent tablespace. This guarantees that when users are created they are assigned a default tablespace other than `SYSTEM`. You don't want users ever creating objects in the `SYSTEM` tablespace, as this can adversely affect performance and availability.
- Ensure users are automatically assigned a default temporary tablespace. This guarantees that when users are created they are assigned a temporary tablespace other than `SYSTEM`. You don't ever want users using the `SYSTEM` tablespace for a temporary sorting space, as this can adversely affect performance and availability.

Solution

Use a script such as the following to create a database that adheres to reasonable standards that set the foundation for a well-performing database:

```
CREATE DATABASE O11R2
   MAXLOGFILES 16
   MAXLOGMEMBERS 4
   MAXDATAFILES 1024
   MAXINSTANCES 1
   MAXLOGHISTORY 680
   CHARACTER SET AL32UTF8
DATAFILE
'/ora01/dbfile/O11R2/system01.dbf'
   SIZE 500M REUSE
   EXTENT MANAGEMENT LOCAL
UNDO TABLESPACE undotbs1 DATAFILE
'/ora02/dbfile/O11R2/undotbs01.dbf'
   SIZE 800M
SYSAUX DATAFILE
'/ora03/dbfile/O11R2/sysaux01.dbf'
   SIZE 500M
DEFAULT TEMPORARY TABLESPACE TEMP TEMPFILE
'/ora02/dbfile/O11R2/temp01.dbf'
   SIZE 500M
```

```
DEFAULT TABLESPACE USERS DATAFILE
'/ora01/dbfile/O11R2/users01.dbf'
   SIZE 50M
LOGFILE GROUP 1
        ('/ora01/oraredo/O11R2/redo01a.rdo',
         '/ora02/oraredo/O11R2/redo01b.rdo') SIZE 200M,
        GROUP 2
        ('/ora01/oraredo/O11R2/redo02a.rdo',
         '/ora02/oraredo/O11R2/redo02b.rdo') SIZE 200M,
        GROUP 3
        ('/ora01/oraredo/O11R2/redo03a.rdo',
         '/ora02/oraredo/O11R2/redo03b.rdo') SIZE 200M
USER sys    IDENTIFIED BY topfoo
USER system IDENTIFIED BY topsecrectfoo;
```

The prior `CREATE DATABASE` script helps establish a good foundation for performance by enabling features such as the following:

- Defines the `SYSTEM` tablespace as locally managed via the `EXTENT MANAGEMENT LOCAL` clause; this ensures that all tablespaces ever created in database are locally managed. If you are using Oracle Database 11g R2 or higher, the `EXTENT MANAGEMENT DICTIONARY` clause has been deprecated.
- Defines a default tablespace named `USERS` for any user created without an explicitly defined default tablespace; this helps prevent users from being assigned the `SYSTEM` tablespace as the default. Users created with a default tablespace of `SYSTEM` can have an adverse impact on performance.
- Defines a default temporary tablespace named `TEMP` for all users; this helps prevent users from being assigned the `SYSTEM` tablespace as the default temporary tablespace. Users created with a default temporary tablespace of `SYSTEM` can have an adverse impact on performance, as this will cause contention for resources in the `SYSTEM` tablespace.

Solid performance starts with a correctly configured database. The prior recommendations help you create a reliable infrastructure for your table data.

How It Works

A properly configured and created database will help ensure that your database performs well. It is true that you can modify features after the database is created. However, oftentimes a poorly crafted `CREATE DATABASE` script leads to a permanent handicap on performance. In production database environments, it's sometimes difficult to get the downtime that might be required to reconfigure an improperly configured database. If possible, think about performance at every step in creating an environment, starting with how you create the database.

When creating a database, you should also consider features that affect maintainability. A sustainable database results in more uptime, which is part of the overall performance equation. The `CREATE DATABASE` statement in the "Solution" section also factors in the following sustainability features:

- Creates an automatic UNDO tablespace (automatic undo management is enabled by setting the UNDO_MANAGEMENT and UNDO_TABLESPACE initialization parameters); this allows Oracle to automatically manage the rollback segments. This relieves you of having to regularly monitor and tweak.
- Places datafiles in directories that follow standards for the environment; this helps with maintenance and manageability, which results in better long-term availability and thus better performance.
- Sets passwords to non-default values for DBA-related users; this ensures the database is more secure, which in the long run can also affect performance (for example, if a malcontent hacks into the database and deletes data, then performance will suffer).
- Establishes three groups of online redo logs, with two members each, sized appropriately for the transaction load; the size of the redo log directly affects the rate at which they switch. When redo logs switch too often, this can degrade performance.

You should take the time to ensure that each database you build adheres to commonly accepted standards that help ensure you start on a firm performance foundation.

If you've inherited a database and want to verify the default permanent tablespace setting, use a query such as this:

```
SELECT *
FROM database_properties
WHERE property_name = 'DEFAULT_PERMANENT_TABLESPACE';
```

If you need to modify the default permanent tablespace, do so as follows:

```
SQL> alter database default tablespace users;
```

To verify the setting of the default temporary tablespace, use this query:

```
SELECT *
FROM database_properties
WHERE property_name = 'DEFAULT_TEMP_TABLESPACE';
```

To change the setting of the temporary tablespace, you can do so as follows:

```
SQL> alter database default temporary tablespace temp;
```

You can verify the UNDO tablespace settings via this query:

```
select name, value
from v$parameter
where name in ('undo_management','undo_tablespace');
```

If you need to change the undo tablespace, first create a new undo tablespace and then use the ALTER SYSTEM SET UNDO_TABLESPACE statement.

1-2. Creating Tablespaces to Maximize Performance

Problem

You realize that tablespaces are the logical containers for database objects such as tables and indexes. Furthermore, you're aware that if you don't specify storage attributes when creating objects, then the tables and indexes automatically inherit the storage characteristics of the tablespaces (that the tables and indexes are created within). Therefore you want to create tablespaces in a manner that maximizes table performance and maintainability.

Solution

When you have the choice, tablespaces should always be created with the following two features enabled:

- Locally managed
- Automatic segment space management (ASSM)

Here's an example of creating a tablespace that enables the prior two features:

```
create tablespace tools
  datafile '/ora01/dbfile/INVREP/tools01.dbf'
  size 100m                                    -- Fixed datafile size
  extent management local               -- Locally managed
  uniform size 128k                          -- Uniform extent size
  segment space management auto -- ASSM
/
```

■ **Note** As of Oracle Database 11g R2, the `EXTENT MANAGEMENT DICTIONARY` clause has been deprecated.

Locally managed tablespaces are more efficient than dictionary-managed tablespaces. This feature is enabled via the `EXTENT MANAGEMENT LOCAL` clause. Furthermore, if you created your database with the `SYSTEM` tablespace as locally managed, you will not be permitted to later create a dictionary-managed tablespace. This is the desired behavior.

The ASSM feature allows for Oracle to manage many of the storage characteristics that formerly had to be manually adjusted by the DBA on a table-by-table basis. ASSM is enabled via the `SEGMENT SPACE MANAGEMENT AUTO` clause. Using ASSM relieves you of these manual tweaking activities. Furthermore, some of Oracle's space management features (such as shrinking a table and SecureFile LOBs) are allowed only when using ASSM tablespaces. If you want to take advantage of these features, then you must create your tablespaces using ASSM.

You can choose to have the extent size be consistently the same for every extent within the tablespace via the `UNIFORM SIZE` clause. Alternatively you can specify `AUTOALLOCATE`. This allows Oracle to allocate extent sizes of 64 KB, 1 MB, 8 MB, and 64 MB. You may prefer the auto-allocation behavior if the objects in the tablespace typically are of varying size.

How It Works

Prior to Oracle Database 11g R2, you had the option of creating a tablespace as dictionary-managed. This architecture uses structures in Oracle's data dictionary to manage an object's extent allocation and free space. Dictionary-managed tablespaces tend to experience poor performance as the number of extents for a table or index reaches the thousands.

You should never use dictionary-managed tablespaces; instead use locally managed tablespaces. Locally managed tablespaces use a bitmap in each datafile to manage the object extents and free space and are much more efficient than the deprecated dictionary-managed architecture.

In prior versions of Oracle, DBAs would spend endless hours monitoring and modifying the physical space management aspects of a table. The combination of locally managed and ASSM render many of these space settings obsolete. For example, the storage parameters are not valid parameters in locally managed tablespaces:

- `NEXT`
- `PCTINCREASE`
- `MINEXTENTS`
- `MAXEXTENTS`
- `DEFAULT`

The `SEGMENT SPACE MANAGEMENT AUTO` clause instructs Oracle to manage physical space within the block. When you use this clause, there is no need to specify parameters such as the following:

- `PCTUSED`
- `FREELISTS`
- `FREELIST GROUPS`

The alternative to `AUTO` space management is `MANUAL` space management. When you use `MANUAL`, you can adjust the previously mentioned parameters depending on the needs of your application. We recommend that you use `AUTO` (and do not use `MANUAL`). Using `AUTO` reduces the number of parameters you'd otherwise need to configure and manage. You can verify the use of locally managed and ASSM with the following query:

```
select
 tablespace_name
,extent_management
,segment_space_management
from dba_tablespaces;
```

Here is some sample output:

```
TABLESPACE_NAME                EXTENT_MAN SEGMENT
------------------------------ ---------- -------
SYSTEM                         LOCAL      MANUAL
SYSAUX                         LOCAL      AUTO
UNDOTBS1                       LOCAL      MANUAL
TEMP                           LOCAL      MANUAL
USERS                          LOCAL      AUTO
TOOLS                          LOCAL      AUTO
```

■ **Note** You cannot create the `SYSTEM` tablespace with automatic segment space management. Also, the ASSM feature is valid only for permanent, locally managed tablespaces.

You can also specify that a datafile automatically grow when it becomes full. This is set through the `AUTOEXTEND ON` clause. If you use this feature, we recommend that you set an overall maximum size for the datafile. This will prevent runaway or erroneous SQL from accidentally consuming all available disk space. Here's an example clause:

```
SIZE 1G AUTOEXTEND ON MAXSIZE 10G
```

When you create a tablespace, you can also specify the tablespace type to be `smallfile` or `bigfile`. Prior to Oracle Database 10g, `smallfile` was your only choice. A `smallfile` tablespace allows you to create one or more datafiles to be associated with a single tablespace. This allows you to spread out the datafiles (associated with one tablespace) across many different mount points. For many environments, you'll require this type of flexibility.

The `bigfile` tablespace can have only one datafile associated with it. The main advantage of the `bigfile` feature is that you can create very large datafiles, which in turn allows you to create very large databases. For example, with the 8 KB block size, you can create a datafile as large as 32 TB. With a 32 KB block size, you can create a datafile up to 128 TB. Also, when using `bigfile`, you will typically have fewer datafiles to manage and maintain. This behavior may be desirable in environments where you use Oracle's Automatic Storage Management (ASM) feature. In ASM environments, you typically are presented with just one logical disk location from which you allocate space.

Here's an example of creating a `bigfile` tablespace:

```
create bigfile tablespace tools_bf
  datafile '/ora01/dbfile/O11R2/tools_bf01.dbf'
  size 100m
  extent management local
  uniform size  128k
  segment space management auto
/
```

You can verify the tablespace type via this query:

```
SQL> select tablespace_name, bigfile from dba_tablespaces;
```

Unless specified, the default tablespace type is `smallfile`. You can make `bigfile` the default tablespace type for a database when you create it via the `SET DEFAULT BIGFILE TABLESPACE` clause. You can alter the default tablespace type for a database to be `bigfile` using the `ALTER DATABASE SET DEFAULT BIGFILE TABLESPACE` statement.

1-3. Matching Table Types to Business Requirements

Problem

You're new to Oracle and have read about the various table types available. For example, you can choose between heap-organized tables, index-organized tables, and so forth. You want to build a database application and need to decide which table type to use.

Solution

Oracle provides a wide variety of table types. The default table type is heap-organized. For most applications, a heap-organized table is an effective structure for storing and retrieving data. However, there are other table types that you should be aware of, and you should know the situations under which each table type should be implemented. Table 1-1 describes each table type and its appropriate use.

Table 1-1. *Oracle Table Types and Typical Uses*

Table Type/Feature	Description	Benefit/Use
Heap-organized	The default Oracle table type and the most commonly used	Table type to use unless you have a specific reason to use a different type
Temporary	Session private data, stored for the duration of a session or transaction; space is allocated in temporary segments.	Program needs a temporary table structure to store and sort data. Table isn't required after program ends.
Index-organized (IOT)	Data stored in a B-tree index structure sorted by primary key	Table is queried mainly on primary key columns; provides fast random access
Partitioned	A logical table that consists of separate physical segments	Type used with large tables with millions of rows; dramatically affects performance scalability of large tables and indexes
Materialized view (MV)	A table that stores the output of a SQL query; periodically refreshed when you want the MV table updated with a current snapshot of the SQL result set	Aggregating data for faster reporting or replicating data to offload performance to a reporting database
Clustered	A group of tables that share the same data blocks	Type used to reduce I/O for tables that are often joined on the same columns

Table Type/Feature	Description	Benefit/Use
External	Tables that use data stored in operating system files outside of the database	This type lets you efficiently access data in a file outside of the database (like a CSV or text file). External tables provide an efficient mechanism for transporting data between databases.
Nested	A table with a column with a data type that is another table	Seldom used
Object	A table with a column with a data type that is an object type	Seldom used

How It Works

In most scenarios, a heap-organized table is sufficient to meet your requirements. This Oracle table type is a proven structure used in a wide variety of database environments. If you properly design your database (normalized structure) and combine that with the appropriate indexes and constraints, the result should be a well-performing and maintainable system.

Normally most of your tables will be heap-organized. However, if you need to take advantage of a non-heap feature (and are certain of its benefits), then certainly do so. For example, Oracle partitioning is a scalable way to build very large tables and indexes. Materialized views are a solid feature for aggregating and replicating data. Index-organized tables are efficient structures when most of the columns are part of the primary key (like an intersection table in a many-to-many relationship). And so forth.

■ **Caution** You shouldn't choose a table type simply because you think it's a cool feature that you recently heard about. Sometimes folks read about a feature and decide to implement it without first knowing what the performance benefits or maintenance costs will be. You should first be able to test and prove that a feature has solid performance benefits.

1-4. Choosing Table Features for Performance

Problem

When creating tables, you want to implement the appropriate data types and constraints that maximize performance, scalability, and maintainability.

Solution

There are several performance and sustainability issues that you should consider when creating tables. Table 1-2 describes features specific to table performance.

Table 1-2. *Table Features That Impact Performance*

Recommendation	Reasoning
If a column always contains numeric data, make it a number data type.	Enforces a business rule and allows for the greatest flexibility, performance, and consistent results when using Oracle SQL math functions (which may behave differently for a "01" character vs. a 1 number); correct data types prevent unnecessary conversion of data types.
If you have a business rule that defines the length and precision of a number field, then enforce it—for example, `NUMBER(7,2)`. If you don't have a business rule, make it `NUMBER(38)`.	Enforces a business rule and keeps the data cleaner; numbers with a precision defined won't unnecessarily store digits beyond the required precision. This can affect the row length, which in turn can have an impact on I/O performance.
For character data that is of variable length, use `VARCHAR2` (and not `VARCHAR`).	Follows Oracle's recommendation of using `VARCHAR2` for character data (instead of `VARCHAR`); Oracle guarantees that the behavior of `VARCHAR2` will be consistent and not tied to an ANSI standard. The Oracle documentation states in the future `VARCHAR` will be redefined as a separate data type.
Use `DATE` and `TIMESTAMP` data types appropriately.	Enforces a business rule, ensures that the data is of the appropriate format, and allows for the greatest flexibility and performance when using SQL date functions and date arithmetic
Consider setting the physical attribute `PCTFREE` to a value higher than the default of 10% if the table initially has rows inserted with null values that are later updated with large values.	Prevents row chaining, which can impact performance if a large percent of rows in a table are chained
Most tables should be created with a primary key.	Enforces a business rule and allows you to uniquely identify each row; ensures that an index is created on primary key column(s), which allows for efficient access to primary key values
Create a numeric surrogate key to be the primary key for each table. Populate the surrogate key from a sequence.	Makes joins easier (only one column to join) and one single numeric key performs better than large concatenated columns.

Recommendation	Reasoning
Create a unique key for the logical business key—a recognizable combination of columns that makes a row unique.	Enforces a business rule and keeps the data cleaner; allows for efficient retrieval of the logical key columns that may be frequently used in `WHERE` clauses
Define foreign keys where appropriate.	Enforces a business rule and keeps the data cleaner; helps optimizer choose efficient paths to data; prevents unnecessary table-level locks in certain DML operations
Consider special features such as virtual columns, read-only, parallel, compression, no logging, and so on.	Features such as parallel DML, compression, or no logging can have a performance impact on reading and writing of data.

How It Works

The "Solution" section describes aspects of tables that relate to performance. When creating a table, you should also consider features that enhance scalability and availability. Oftentimes DBAs and developers don't think of these features as methods for improving performance. However, building a stable and supportable database goes hand in hand with good performance. Table 1-3 describes best practices features that promote ease of table management.

Table 1-3. *Table Features That Impact Scalability and Maintainability*

Recommendation	Reasoning
Use standards when naming tables, columns, constraints, triggers, indexes, and so on.	Helps document the application and simplifies maintenance
If you have a business rule that specifies the maximum length of a column, then use that length, as opposed to making all columns `VARCHAR2(4000)`.	Enforces a business rule and keeps the data cleaner
Specify a separate tablespace for the table and indexes.	Simplifies administration and maintenance
Let tables and indexes inherit storage attributes from the tablespaces.	Simplifies administration and maintenance
Create primary-key constraints out of line.	Allows you more flexibility when creating the primary key, especially if you have a situation where the primary key consists of multiple columns
Create comments for the tables and columns.	Helps document the application and eases maintenance

Continued

Recommendation	Reasoning
Avoid large object (LOB) data types if possible.	Prevents maintenance issues associated with LOB columns, like unexpected growth, performance issues when copying, and so on
If you use LOBs in Oracle Database 11g or higher, use the new SecureFiles architecture.	SecureFiles is the new LOB architecture going forward; provides new features such as compression, encryption, and deduplication
If a column should always have a value, then enforce it with a `NOT NULL` constraint.	Enforces a business rule and keeps the data cleaner
Create audit-type columns, such as `CREATE_DTT` and `UPDATE_DTT`, that are automatically populated with default values and/or triggers.	Helps with maintenance and determining when data was inserted and/or updated; other types of audit columns to consider include the users who inserted and updated the row.
Use check constraints where appropriate.	Enforces a business rule and keeps the data cleaner; use this to enforce fairly small and static lists of values.

1-5. Avoiding Extent Allocation Delays When Creating Tables

Problem

You're installing an application that has thousands of tables and indexes. Each table and index are configured to initially allocate an initial extent of 10 MB. When deploying the installation DDL to your production environment, you want install the database objects as fast as possible. You realize it will take some time to deploy the DDL if each object allocates 10 MB of disk space as it is created. You wonder if you can somehow instruct Oracle to defer the initial extent allocation for each object until data is actually inserted into a table.

Solution

The only way to defer the initial segment generation is to use Oracle Database 11g R2. With this version of the database (or higher), by default the physical allocation of the extent for a table (and associated indexes) is deferred until a record is first inserted into the table. A small example will help illustrate this concept. First a table is created:

```
SQL> create table f_regs(reg_id number, reg_name varchar2(200));
```

Now query `USER_SEGMENTS` and `USER_EXTENTS` to verify that no physical space has been allocated:

```
SQL> select count(*) from user_segments where segment_name='F_REGS';
  COUNT(*)
```

```
----------
         0

SQL> select count(*) from user_extents where segment_name='F_REGS';
  COUNT(*)
----------
         0
```

Next a record is inserted, and the prior queries are run again:

```
SQL> insert into f_regs values(1,'BRDSTN');

1 row created.

SQL>> select count(*) from user_segments where segment_name='F_REGS';
 COUNT(*)
----------
         1

SQL> select count(*) from user_extents where segment_name='F_REGS';
  COUNT(*)
----------
         1
```

The prior behavior is quite different from previous versions of Oracle. In prior versions, as soon as you create an object, the segment and associated extent are allocated.

■ **Note** Deferred segment generation also applies to partitioned tables and indexes. An extent will not be allocated until the initial record is inserted into a given extent.

How It Works

Starting with Oracle Database 11g R2, with non-partitioned heap-organized tables created in locally managed tablespaces, the initial segment creation is deferred until a record is inserted into the table. You need to be aware of Oracle's deferred segment creation feature for several reasons:

- Allows for a faster installation of applications that have a large number of tables and indexes; this improves installation speed, especially when you have thousands of objects.
- As a DBA, your space usage reports may initially confuse you when you notice that there is no space allocated for objects.
- The creation of the first row will take a slightly longer time than in previous versions (because now Oracle allocates the first extent based on the creation of the first row). For most applications, this performance degradation is not noticeable.

We realize that to take advantage of this feature the only "solution" is to upgrade to Oracle Database 11g R2, which is oftentimes not an option. However, we felt it was important to discuss this feature because you'll eventually encounter the aforementioned characteristics (when you start using the latest release of Oracle).

You can disable the deferred segment creation feature by setting the database initialization parameter `DEFERRED_SEGMENT_CREATION` to `FALSE`. The default for this parameter is `TRUE`.

You can also control the deferred segment creation behavior when you create the table. The `CREATE TABLE` statement has two new clauses: `SEGMENT CREATION IMMEDIATE` and `SEGMENT CREATION DEFERRED`—for example:

```
create table f_regs(
 reg_id number
,reg_name varchar2(2000))
segment creation immediate;
```

■ **Note** The `COMPATIBLE` initialization parameter needs to be 11.2.0.0.0 or greater before using the `SEGMENT CREATION DEFERRED` clause.

1-6. Maximizing Data Loading Speeds

Problem

You're loading a large amount of data into a table and want to insert new records as quickly as possible.

Solution

Use a combination of the following two features to maximize the speed of insert statements:

- Set the table's logging attribute to `NOLOGGING`; this minimizes the generation redo for direct path operations (this feature has no effect on regular DML operations).
- Use a direct path loading feature, such as the following:
  - `INSERT /*+ APPEND */` on queries that use a subquery for determining which records are inserted
  - `INSERT /*+ APPEND_VALUES */` on queries that use a `VALUES` clause
  - `CREATE TABLE…AS SELECT`

Here's an example to illustrate `NOLOGGING` and direct path loading. First, run the following query to verify the logging status of a table. In this example, the table name is `F_REGS`:

```
select
 table_name
,logging
from user_tables
where table_name = 'F_REGS';
```

Here is some sample output:

```
TABLE_NAME             LOG
------------------------------ ---
F_REGS                         YES
```

The prior output verifies that the table was created with `LOGGING` enabled (the default). To enable `NOLOGGING`, use the `ALTER TABLE` statement as follows:

```
SQL> alter table f_regs nologging;
```

Now that `NOLOGGING` has been enabled, there should be a minimal amount of redo generated for direct path operations. The following example uses a direct path `INSERT` statement to load data into the table:

```
insert /*+APPEND */ into f_regs
select * from reg_master;
```

The prior statement is an efficient method for loading data because direct path operations such as `INSERT /*+APPEND */` combined with `NOLOGGING` generate a minimal amount of redo.

How It Works

Direct path inserts have two performance advantages over regular insert statements:

- If `NOLOGGING` is specified, then a minimal amount of redo is generated.
- The buffer cache is bypassed and data is loaded directly into the datafiles. This can significantly improve the loading performance.

The `NOLOGGING` feature minimizes the generation of redo for direct path operations only. For direct path inserts, the `NOLOGGING` option can significantly increase the loading speed. One perception is that `NOLOGGING` eliminates redo generation for the table for all DML operations. That isn't correct. The `NOLOGGING` feature never affects redo generation for regular `INSERT`, `UPDATE`, `MERGE`, and `DELETE` statements.

One downside to reducing redo generation is that you can't recover the data created via `NOLOGGING` in the event a failure occurs after the data is loaded (and before you back up the table). If you can tolerate some risk of data loss, then use `NOLOGGING` but back up the table soon after the data is loaded. If your data is critical, then don't use `NOLOGGING`. If your data can be easily re-created, then `NOLOGGING` is desirable when you're trying to improve performance of large data loads.

What happens if you have a media failure after you've populated a table in `NOLOGGING` mode (and before you've made a backup of the table)? After a restore and recovery operation, it will appear that the table has been restored:

```
SQL> desc f_regs;

Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 REG_ID                                             NUMBER
 REG_NAME                                           VARCHAR2(2000)
```

However, when executing a query that scans every block in the table, an error is thrown.

```
SQL> select * from f_regs;
```

This indicates that there is logical corruption in the datafile:

```
ORA-01578: ORACLE data block corrupted (file # 10, block # 198)
ORA-01110: data file 10: '/ora01/dbfile/O11R2/users201.dbf'
ORA-26040: Data block was loaded using the NOLOGGING option
```

As the prior output indicates, the data in the table is unrecoverable. Use `NOLOGGING` only in situations where the data isn't critical or in scenarios where you can back up the data soon after it was created.

■ **Tip** If you're using RMAN to back up your database, you can report on unrecoverable datafiles via the `REPORT UNRECOVERABLE` command.

There are some quirks of `NOLOGGING` that need some explanation. You can specify logging characteristics at the database, tablespace, and object levels. If your database has been enabled to force logging, then this overrides any `NOLOGGING` specified for a table. If you specify a logging clause at the tablespace level, it sets the default logging for any `CREATE TABLE` statements that don't explicitly use a logging clause.

You can verify the logging mode of the database as follows:

```
SQL> select name, log_mode, force_logging from v$database;
```

The next statement verifies the logging mode of a tablespace:

```
SQL> select tablespace_name, logging from dba_tablespaces;
```

And this example verifies the logging mode of a table:

```
SQL> select owner, table_name, logging from dba_tables where logging = 'NO';
```

How do you tell whether Oracle logged redo for an operation? One way is to measure the amount of redo generated for an operation with logging enabled vs. operating in `NOLOGGING` mode. If you have a development environment for testing, you can monitor how often the redo logs switch while the transactions are taking place. Another simple test is to measure how long the operation takes with and without logging. The operation performed in `NOLOGGING` mode should occur faster because a minimal amount of redo is generated during the load.

1-7. Efficiently Removing Table Data

Problem

You're experiencing performance issues when deleting data from a table. You want to remove data as efficiently as possible.

Solution

You can use either the `TRUNCATE` statement or the `DELETE` statement to remove records from a table. `TRUNCATE` is usually more efficient but has some side effects that you must be aware of. For example, `TRUNCATE` is a DDL statement. This means Oracle automatically commits the statement (and the current transaction) after it runs, so there is no way to roll back a `TRUNCATE` statement. Because a `TRUNCATE` statement is DDL, you can't truncate two separate tables as one transaction.

This example uses a `TRUNCATE` statement to remove all data from the `COMPUTER_SYSTEMS` table:

```
SQL> truncate table computer_systems;
```

When truncating a table, by default all space is de-allocated for the table except the space defined by the `MINEXTENTS` table-storage parameter. If you don't want the `TRUNCATE` statement to de-allocate the currently allocated extents, then use the `REUSE STORAGE` clause:

```
SQL> truncate table computer_systems reuse storage;
```

You can query the `DBA/ALL/USER_EXTENTS` views to verify if the extents have been de-allocated (or not)—for example:

```
select count(*)
 from user_extents where segment_name = 'COMPUTER_SYSTEMS';
```

How It Works

If you need the option of choosing to roll back (instead of committing) when removing data, then you should use the `DELETE` statement. However, the `DELETE` statement has the disadvantage that it generates a great deal of undo and redo information. Thus for large tables, a `TRUNCATE` statement is usually the most efficient way to remove data.

Another characteristic of the `TRUNCATE` statement is that it sets the high-water mark of a table back to zero. When you use a `DELETE` statement to remove data from a table, the high-water mark doesn't change. One advantage of using a `TRUNCATE` statement and resetting the high-water mark is that full table scan queries search only for rows in blocks below the high-water mark. This can have significant performance implications for queries that perform full table scans.

Another side effect of the TRUNCATE statement is that you can't truncate a parent table that has a primary key defined that is referenced by an enabled foreign-key constraint in a child table—even if the child table contains zero rows. In this scenario, Oracle will throw this error when attempting to truncate the parent table:

```
ORA-02266: unique/primary keys in table referenced by enabled foreign keys
```

Oracle prevents you from truncating the parent table because in a multiuser system, there is a possibility that another session can populate the child table with rows in between the time you truncate the child table and the time you subsequently truncate the parent table. In this situation, you must temporarily disable the child table–referenced foreign-key constraints, issue the TRUNCATE statement, and then re-enable the constraints.

Compare the TRUNCATE behavior to that of the DELETE statement. Oracle does allow you to use the DELETE statement to remove rows from a parent table while the constraints are enabled that reference a child table (assuming there are zero rows in the child table). This is because DELETE generates undo, is read-consistent, and can be rolled back. Table 1-4 summarizes the differences between DELETE and TRUNCATE.

If you need to use a DELETE statement, you must issue either a COMMIT or a ROLLBACK to complete the transaction. Committing a DELETE statement makes the data changes permanent:

```
SQL> delete from computer_systems;
SQL> commit;
```

■ **Note** Other (sometimes not so obvious) ways of committing a transaction include issuing a subsequent DDL statement (which implicitly commits an active transaction for a session) or normally exiting out of the client tool (such as SQL*Plus).

If you issue a ROLLBACK statement instead of COMMIT, the table contains data as it was before the DELETE was issued.

When working with DML statements, you can confirm the details of a transaction by querying from the V$TRANSACTION view. For example, say that you have just inserted data into a table; before you issue a COMMIT or ROLLBACK, you can view active transaction information for the currently connected session as follows:

```
SQL> insert into computer_systems(cs_id) values(1);
SQL> select xidusn, xidsqn from v$transaction;
    XIDUSN     XIDSQN
---------- ----------
         3      12878
SQL> commit;
SQL> select xidusn, xidsqn from v$transaction;
no rows selected
```

Table 1-4. *Comparison of* `DELETE` *and* `TRUNCATE`

	DELETE	TRUNCATE
Option of committing or rolling back changes	YES	NO (DDL statement is always committed after it runs.)
Generates undo	YES	NO
Resets the table high-water mark to zero	NO	YES
Affected by referenced and enabled foreign-key constraints	NO	YES
Performs well with large amounts of data	NO	YES

■ **Note** Another way to remove data from a table is to drop and re-create the table. However, this means you also have to re-create any indexes, constraints, grants, and triggers that belong to the table. Additionally, when you drop a table, it's temporarily unavailable until you re-create it and re-issue any required grants. Usually, dropping and re-creating a table is acceptable only in a development or test environment.

1-8. Displaying Automated Segment Advisor Advice

Problem

You have a poorly performing query accessing a table. Upon further investigation, you discover the table has only a few rows in it. You wonder why the query is taking so long when there are so few rows. You want to examine the output of the Segment Advisor to see if there are any space-related recommendations that might help with performance in this situation.

Solution

Use the Segment Advisor to display information regarding tables that may have space allocated to them (that was once used) but now the space is empty (due to a large number of deleted rows). Tables with large amounts of unused space can cause full table scan queries to perform poorly. This is because Oracle is scanning every block beneath the high-water mark, regardless of whether the blocks contain data.

This solution focuses on accessing the Segment Advisor's advice via the `DBMS_SPACE` PL/SQL package. This package retrieves information generated by the Segment Advisor regarding segments that may be candidates for shrinking, moving, or compressing. One simple and effective way to use the `DBMS_SPACE` package (to obtain Segment Advisor advice) is via a SQL query—for example:

```
SELECT
 'Segment Advice --------------------------'|| chr(10) ||
 'TABLESPACE_NAME  : ' || tablespace_name   || chr(10) ||
 'SEGMENT_OWNER    : ' || segment_owner     || chr(10) ||
 'SEGMENT_NAME     : ' || segment_name      || chr(10) ||
 'ALLOCATED_SPACE  : ' || allocated_space   || chr(10) ||
 'RECLAIMABLE_SPACE: ' || reclaimable_space || chr(10) ||
 'RECOMMENDATIONS  : ' || recommendations   || chr(10) ||
 'SOLUTION 1       : ' || c1                || chr(10) ||
 'SOLUTION 2       : ' || c2                || chr(10) ||
 'SOLUTION 3       : ' || c3 Advice
FROM
TABLE(dbms_space.asa_recommendations('FALSE', 'FALSE', 'FALSE'));
```

Here is some sample output:

```
Segment Advice --------------------------
TABLESPACE_NAME  : USERS
SEGMENT_OWNER    : MV_MAINT
SEGMENT_NAME     : F_REGS
ALLOCATED_SPACE  : 20971520
RECLAIMABLE_SPACE: 18209960
RECOMMENDATIONS  : Perform re-org on the object F_REGS, estimated savings is 182
09960 bytes.
SOLUTION 1       : Perform Reorg
SOLUTION 2       :
SOLUTION 3       :
```

In the prior output, the `F_REGS` table is a candidate for the shrink operation. It is consuming 20 MB, and 18 MB can be reclaimed.

How It Works

In Oracle Database 10g R2 and later, Oracle automatically schedules and runs a Segment Advisor job. This job analyzes segments in the database and stores its findings in internal tables. The output of the Segment Advisor contains findings (issues that may need to be resolved) and recommendations (actions to resolve the findings). Findings from the Segment Advisor are of the following types:

- Segments that are good candidates for shrink operations
- Segments that have significant row chaining
- Segments that might benefit from OLTP compression

When viewing the Segment Advisor's findings and recommendations, it's important to understand several aspects of this tool. First, the Segment Advisor regularly calculates advice via an automatically scheduled `DBMS_SCHEDULER` job. You can verify the last time the automatic job ran by querying the `DBA_AUTO_SEGADV_SUMMARY` view:

```
select
 segments_processed
,end_time
from dba_auto_segadv_summary
order by end_time;
```

Here is some sample output:

```
SEGMENTS_PROCESSED END_TIME
------------------ ---------------------------
                 9 30-JAN-11 02.02.46.414424 PM
                11 30-JAN-11 06.03.44.500178 PM
                17 30-JAN-11 10.04.35.688915 PM
```

You can compare the `END_TIME` date to the current date to determine if the Segment Advisor is running on a regular basis.

■ **Note** In addition to automatically generated segment advice, you have the option of manually executing the Segment Advisor to generate advice on specific tablespaces, tables, and indexes (see Recipe 1-9 for details).

When the Segment Advisor executes, it uses the Automatic Workload Repository (AWR) for the source of information for its analysis. For example, the Segment Advisor examines usage and growth statistics in the AWR to generate segment advice. When the Segment Advisor runs, it generates advice and stores the output in internal database tables. The advice and recommendations can be viewed via data dictionary views such as the following:

- `DBA_ADVISOR_EXECUTIONS`
- `DBA_ADVISOR_FINDINGS`
- `DBA_ADVISOR_OBJECTS`

There are three different tools for retrieving the Segment Advisor's output:

- Executing `DBMS_SPACE.ASA_RECOMMENDATIONS`
- Manually querying `DBA_ADVISOR_*` views
- Viewing Enterprise Manager's graphical screens

In the "Solution" section, we described how to use the `DBMS_SPACE.ASA_RECOMMENDATIONS` procedure to retrieve the Segment Advisor advice. The `ASA_RECOMMENDATIONS` output can be modified via three input parameters, which are described in Table 1-5. For example, you can instruct the procedure to show information generated when you have manually executed the Segment Advisor.

Table 1-5. *Description of* `ASA_RECOMMENDATIONS` *Input Parameters*

Parameter	Meaning
`all_runs`	`TRUE` instructs the procedure to return findings from all runs, whereas `FALSE` instructs the procedure to return only the latest run.
`show_manual`	`TRUE` instructs the procedure to return results from manual executions of the Segment Advisor. `FALSE` instructs the procedure to return results from the automatic running of the Segment Advisor.
`show_findings`	Shows only the findings and not the recommendations

You can also directly query the data dictionary views to view the advice of the Segment Advisor. Here's a query that displays Segment Advisor advice generated within the last day:

```
select
 'Task Name        : ' || f.task_name  || chr(10) ||
 'Start Run Time   : ' || TO_CHAR(execution_start, 'dd-mon-yy hh24:mi') || chr (10) ||
 'Segment Name     : ' || o.attr2    || chr(10) ||
 'Segment Type     : ' || o.type       || chr(10) ||
 'Partition Name   : ' || o.attr3      || chr(10) ||
 'Message          : ' || f.message    || chr(10) ||
 'More Info        : ' || f.more_info  || chr(10) ||
 '--------------------------------------------------' Advice
FROM dba_advisor_findings   f
    ,dba_advisor_objects    o
    ,dba_advisor_executions e
WHERE o.task_id   = f.task_id
AND   o.object_id = f.object_id
AND   f.task_id   = e.task_id
AND   e. execution_start > sysdate - 1
AND   e.advisor_name = 'Segment Advisor'
ORDER BY f.task_name;
```

Here is some sample output:

```
Task Name        : SYS_AUTO_SPCADV_53092205022011
Start Run Time   : 05-feb-11 22:09
Segment Name     : CWP_USER_PROFILE
Segment Type     : TABLE
Partition Name   :
Message          : Compress object REP_MV.CWP_USER_PROFILE, estimated savings is
 3933208576 bytes.
More Info        : Allocated Space:3934257152: Used Space:10664: Reclaimable Spa
ce :3933208576:
--------------------------------------------------
```

The prior output indicates that a table segment is a candidate for compression. The allocated, used, and reclaimable space numbers are displayed to help you determine the space savings.

You can also view Segment Advisor advice from Enterprise Manager. To view the advice, first navigate to the Advisor Central page. Next navigate to the Segment Advisor page. Then navigate to the Segment Advisor Recommendations. This page will display any recent Segment Advisor findings and recommendations.

1-9. Manually Generating Segment Advisor Advice

Problem

You have a table that experiences a large amount of updates. You have noticed that the query performance against this table has slowed down. You suspect the table may be experiencing poor performance due to row chaining. Therefore you want to manually confirm with the Segment Advisor that a table has issues with row chaining.

Solution

You can manually run the Segment Advisor and tell it to specifically analyze all segments in a tablespace or look at a specific object (such as a single table or index). You can manually generate advice for a specific segment using the `DBMS_ADVISOR` package by executing the following steps:

1. Create a task.
2. Assign an object to the task.
3. Set the task parameters.
4. Execute the task.

■ **Note** The database user executing `DBMS_ADVISOR` needs the `ADVISOR` system privilege. This privilege is administered via the `GRANT` statement.

The following example executes the `DBMS_ADVISOR` package from an anonymous block of PL/SQL. The table being examined is the `F_REGS` table.

```
DECLARE
  my_task_id   number;
  obj_id       number;
  my_task_name varchar2(100);
  my_task_desc varchar2(500);
BEGIN
  my_task_name := 'F_REGS Advice';
  my_task_desc := 'Manual Segment Advisor Run';
```

```
---------
-- Step 1
---------
  dbms_advisor.create_task (
  advisor_name => 'Segment Advisor',
  task_id      => my_task_id,
  task_name    => my_task_name,
  task_desc    => my_task_desc);
---------
-- Step 2
---------
  dbms_advisor.create_object (
  task_name   => my_task_name,
  object_type => 'TABLE',
  attr1       => 'MV_MAINT',
  attr2       => 'F_REGS',
  attr3       => NULL,
  attr4       => NULL,
  attr5       => NULL,
  object_id   => obj_id);
---------
-- Step 3
---------
  dbms_advisor.set_task_parameter(
  task_name => my_task_name,
  parameter => 'recommend_all',
  value     => 'TRUE');
---------
-- Step 4
---------
  dbms_advisor.execute_task(my_task_name);
END;
/
```

Now you can view Segment Advisor advice regarding this table by executing the `DBMS_SPACE` package and instructing it to pull information from a manual execution of the Segment Advisor (via the input parameters—see Table 1-6 for details)—for example:

```
SELECT
 'Segment Advice --------------------------'|| chr(10) ||
 'TABLESPACE_NAME  : ' || tablespace_name   || chr(10) ||
 'SEGMENT_OWNER    : ' || segment_owner     || chr(10) ||
 'SEGMENT_NAME     : ' || segment_name      || chr(10) ||
 'ALLOCATED_SPACE  : ' || allocated_space   || chr(10) ||
 'RECLAIMABLE_SPACE: ' || reclaimable_space || chr(10) ||
 'RECOMMENDATIONS  : ' || recommendations   || chr(10) ||
 'SOLUTION 1       : ' || c1                || chr(10) ||
 'SOLUTION 2       : ' || c2                || chr(10) ||
 'SOLUTION 3       : ' || c3 Advice
FROM
TABLE(dbms_space.asa_recommendations('TRUE', 'TRUE', 'FALSE'));
```

Here is some sample output:

```
Segment Advice --------------------------
TABLESPACE_NAME  : USERS
SEGMENT_OWNER    : MV_MAINT
SEGMENT_NAME     : F_REGS
ALLOCATED_SPACE  : 20971520
RECLAIMABLE_SPACE: 18209960
RECOMMENDATIONS  : Perform re-org on the object F_REGS, estimated savings is 182
09960 bytes.
SOLUTION 1       : Perform Reorg
SOLUTION 2       :
SOLUTION 3       :
```

You can also retrieve Segment Advisor advice by querying data dictionary views—for example:

```
SELECT
 'Task Name        : ' || f.task_name  || chr(10) ||
 'Segment Name     : ' || o.attr2      || chr(10) ||
 'Segment Type     : ' || o.type       || chr(10) ||
 'Partition Name   : ' || o.attr3      || chr(10) ||
 'Message          : ' || f.message    || chr(10) ||
 'More Info        : ' || f.more_info TASK_ADVICE
FROM dba_advisor_findings f
    ,dba_advisor_objects  o
WHERE o.task_id = f.task_id
AND o.object_id = f.object_id
AND f.task_name like 'F_REGS Advice'
ORDER BY f.task_name;
```

If the table has a potential issue with row chaining, then the advice output will indicate it as follows:

```
TASK_ADVICE
--------------------------------------------------------------------------------
Task Name        : F_REGS Advice
Segment Name     : F_REGS
Segment Type     : TABLE
Partition Name   :
Message          : Perform re-org on the object F_REGS, estimated savings is 182
09960 bytes.
More Info        : Allocated Space:20971520: Used Space:2761560: Reclaimable Spa
ce :18209960:
```

How It Works

The `DBMS_ADVISOR` package is used to manually instruct the Segment Advisor to generate advice for specific tables. This package contains several procedures that perform operations such as creating and executing a task. Table 1-6 lists the procedures relevant to the Segment Advisor.

Table 1-6. *DBMS_ADVISOR Procedures Applicable for the Segment Advisor*

Procedure Name	Description
CREATE_TASK	Creates the Segment Advisor task; specify "Segment Advisor" for the ADVISOR_NAME parameter of CREATE_TASK. Query DBA_ADVISOR_DEFINITIONS for a list of all valid advisors.
CREATE_OBJECT	Identifies the target object for the segment advice; Table 1-7 lists valid object types and parameters.
SET_TASK_PARAMETER	Specifies the type of advice you want to receive; Table 1-8 lists valid parameters and values.
EXECUTE_TASK	Executes the Segment Advisor task
DELETE_TASK	Deletes a task
CANCEL_TASK	Cancels a currently running task

The Segment Advisor can be invoked with various degrees of granularity. For example, you can generate advice for all objects in a tablespace or advice for a specific table, index, or partition. Table 1-7 lists the object types for which Segment Advisor advice can be obtained via the DBMS_ADVISOR.CREATE_TASK procedure.

Table 1-7. *Valid Object Types for the DBMS_ADVISOR.CREATE_TASK Procedure*

Object Type	ATTR1	ATTR2	ATTR3	ATTR4
TABLESPACE	tablespace name	NULL	NULL	NULL
TABLE	user name	table name	NULL	NULL
INDEX	user name	index name	NULL	NULL
TABLE PARTITION	user name	table name	partition name	NULL
INDEX PARTITION	user name	index name	partition name	NULL
TABLE SUBPARTITION	user name	table name	subpartition name	NULL
INDEX SUBPARTITION	user name	index name	subpartition name	NULL
LOB	user name	segment name	NULL	NULL

Object Type	ATTR1	ATTR2	ATTR3	ATTR4
LOB PARTITION	user name	segment name	partition name	NULL
LOB SUBPARTITION	user name	segment name	subpartition name	NULL

You can also specify a maximum amount of time that you want the Segment Advisor to run. This is controlled via the `SET_TASK_PARAMETER` procedure. This procedure also controls the type of advice that is generated. Table 1-8 describes valid inputs for this procedure.

Table 1-8. *Input Parameters for the* `DBMS_ADVISOR.SET_TASK_PARAMETER` *Procedure*

Parameter	Description	Valid Values
TIME_LIMIT	Limit on time (in seconds) for advisor run	N number of seconds or `UNLIMITED` (default)
RECOMMEND_ALL	Generates advice for all types of advice or just space-related advice	`TRUE` (default) for all types of advice, or `FALSE` to generate only space-related advice

1-10. Automatically E-mailing Segment Advisor Output

Problem

You realize that the Segment Advisor automatically generates advice and want to automatically e-mail yourself Segment Advisor output.

Solution

First encapsulate the SQL that displays the Segment Advisor output in a shell script—for example:

```
#!/bin/bash
if [ $# -ne 1 ]; then
  echo "Usage: $0 SID"
  exit 1
fi
# source oracle OS variables
. /var/opt/oracle/oraset $1
#
BOX=`uname -a | awk '{print$2}'`
#
sqlplus -s <<EOF
mv_maint/foo
spo $HOME/bin/log/seg.txt
```

```
set lines 80
set pages 100
SELECT
 'Segment Advice --------------------------'|| chr(10) ||
 'TABLESPACE_NAME  : ' || tablespace_name   || chr(10) ||
 'SEGMENT_OWNER    : ' || segment_owner     || chr(10) ||
 'SEGMENT_NAME     : ' || segment_name      || chr(10) ||
 'ALLOCATED_SPACE  : ' || allocated_space   || chr(10) ||
 'RECLAIMABLE_SPACE: ' || reclaimable_space || chr(10) ||
 'RECOMMENDATIONS  : ' || recommendations   || chr(10) ||
 'SOLUTION 1       : ' || c1                || chr(10) ||
 'SOLUTION 2       : ' || c2                || chr(10) ||
 'SOLUTION 3       : ' || c3 Advice
FROM
TABLE(dbms_space.asa_recommendations('FALSE', 'FALSE', 'FALSE'));
EOF
cat $HOME/bin/log/seg.txt | mailx -s "Seg. rpt. on DB: $1 $BOX" dkuhn@oracle.com
exit 0
```

The prior shell script can be regularly executed from a Linux/Unix utility such as `cron`. Here is a sample `cron` entry:

```
# Segment Advisor report
16 11 * * * /orahome/oracle/bin/seg.bsh DWREP
```

In this way, you automatically receive segment advice and proactively resolve issues before they become performance problems.

How It Works

The Segment Advisor automatically generates advice on a regular basis. Sometimes it's handy to proactively send yourself the recommendations. This allows you to periodically review the output and implement suggestions that make sense.

The shell script in the "Solution" section contains a line near the top where the OS variables are established through running an `oraset` script. This is a custom script that is the equivalent of the `oraset` script provided by Oracle. You can use a script to set the OS variables or hard-code the required lines into the script. Calling a script to set the variables is more flexible and maintainable, as it allows you to use as input any database name that appears in the `oratab` file.

1-11. Rebuilding Rows Spanning Multiple Blocks

Problem

You have a table in which individual rows are stored in more than one block. That situation leads to higher rates of I/O, and causes queries against the table to run slowly. You want to rebuild the spanned rows such that each row fits into a single block.

For example, you're running the following query, which displays Segment Advisor advice:

```
SELECT
 'Task Name        : ' || f.task_name  || chr(10) ||
 'Segment Name     : ' || o.attr2      || chr(10) ||
 'Segment Type     : ' || o.type       || chr(10) ||
 'Partition Name   : ' || o.attr3      || chr(10) ||
 'Message          : ' || f.message    || chr(10) ||
 'More Info        : ' || f.more_info TASK_ADVICE
FROM dba_advisor_findings f
    ,dba_advisor_objects  o
WHERE o.task_id   = f.task_id
AND   o.object_id = f.object_id
ORDER BY f.task_name;
```

Here is the output for this example:

```
TASK_ADVICE
--------------------------------------------------------------------------------
Task Name        : EMP Advice
Segment Name     : EMP
Segment Type     : TABLE
Partition Name   :
Message          : The object has chained rows that can be removed by re-org.
More Info        : 47 percent chained rows can be removed by re-org.
```

From the prior output, the `EMP` table has a large percentage of rows affected by row chaining and is causing performance issues when retrieving data from the table. You want to eliminate the chained rows within the table.

Solution

One method for resolving the row chaining within a table is to use the `MOVE` statement. When you move a table, Oracle requires an exclusive lock on the table; therefore you should perform this operation when there are no active transactions associated with the table being moved.

Also, as part of a `MOVE` operation, all of the rows are assigned a new `ROWID`. This will invalidate any indexes that are associated with the table. Therefore, as part of the move operation, you should rebuild all indexes associated with the table being moved. This example moves the `EMP` table:

```
SQL> alter table emp move;
```

After the move operation completes, then rebuild any indexes associated with the table being moved. You can verify the status of the indexes by interrogating the `DBA/ALL/USER_INDEXES` view:

```
select
 owner
,index_name
,status
from dba_indexes
where table_name='EMP';
```

Here is some sample output:

```
OWNER                          INDEX_NAME                     STATUS
------------------------------ ------------------------------ --------
MV_MAINT                         EMP_PK                   UNUSABLE
```

Rebuilding the index will make it usable again:

```
SQL> alter index emp_pk rebuild;
```

You can now manually generate Segment Advisor advice (see Recipe 1-9) for the segment and run the query listed in the "Problem" section of this recipe to see if the row chaining has been resolved.

How It Works

A certain amount of space is reserved in the block to accommodate growth within the row. Usually a row will increase in size due to an `UPDATE` statement that increases the size of a column value. If there isn't enough free room in the block to accommodate the increased size, then Oracle will create a pointer to a different block that does have enough space and store part of the row in this additional block. When a single row is stored in two or more blocks, this is called *row chaining*. This can cause potential performance issues because Oracle will have to retrieve data from multiple blocks (instead of one) when retrieving a chained row.

A small number of chained rows won't have much impact on performance. One rough guideline is that if more than 15% of a table's rows are chained, then you should take corrective action (such as moving the table to re-organize it).

The amount of free space reserved in a block is determined by the table's storage parameter of `PCTFREE`. The default value of `PCTFREE` is 10, meaning 10% of the block is reserved space to be used for updates that result in more space usage. If you have a table that has columns that are initially inserted as null and later updated to contain large values, then consider setting `PCTFREE` to a higher value, such as 40%. This will help prevent the row chaining.

Conversely, if you have a table that is never updated after rows are inserted, then consider setting `PCTFREE` to 0. This will result in more rows per block, which can lead to fewer disk reads (and thus better performance) when retrieving data.

You can view the setting for `PCTFREE` by querying the `DBA/ALL/USER_TABLES` view—for example:

```
select table_name, pct_free
from user_tables;
```

The move operation removes each record from the block and re-inserts the record into a new block. For chained rows, the old chained rows are deleted and rebuilt as one physical row within the block. If the table being moved has a low setting for `PCTFREE`, consider resetting this parameter to a higher value (as part of the move operation)—for example:

```
SQL> alter table emp move pctfree 40;
```

Another method for verifying row chaining (besides the Segment Advisor) is to use the `ANALYZE TABLE` statement. First you must create a table to hold output of the `ANALYZE TABLE` statement:

```
SQL> @?/rdbms/admin/utlchain.sql
```

The prior script creates a table named CHAINED_ROWS. Now you can run the ANALYZE statement to populate the CHAINED_ROWS table:

```
SQL> analyze table emp list chained rows;
```

Now query the number of rows from the CHAINED_ROWS table:

```
SQL> select count(*) from chained_rows where table_name='EMP';
```

If the issue with the chained rows has been resolved, the prior query will return zero rows. The advantage of identifying chained rows in this manner is that you can fix the rows that are chained without impacting the rest of the records in the table by doing the following:

1. Create a temporary holding table to store the chained rows.
2. Delete the chained rows from the original table.
3. Insert the rows from the temporary table into the original table.

Here's a short example to demonstrate the prior steps. First create a temporary table that contains the rows in the EMP table that have corresponding records in the CHAINED_ROWS table:

```
create table temp_emp
as select *
from emp
where rowid in
(select head_rowid from chained_rows where table_name = 'EMP');
```

Now delete the records from EMP that have corresponding records in CHAINED_ROWS:

```
delete from emp
where rowid in
(select head_rowid from chained_rows where table_name = 'EMP');
```

Now insert records in the temporary table into the EMP table:

```
insert into emp select * from temp_emp;
```

If you re-analyze the table, there should be no chained rows now. You can drop the temporary table when you're finished.

UNDERSTANDING THE ORACLE ROWID

Every row in every table has a physical address. The address of a row is determined from a combination of the following:

- Datafile number
- Block number
- Location of the row within the block
- Object number

You can display the address of a row in a table by querying the ROWID pseudo-column—for example:

```
SQL> select rowid, emp_id from emp;
```

Here's some sample output:

```
ROWID                  EMP_ID
------------------ ----------
AAAFWXAAFAAAAlWAAA          1
```

The ROWID pseudo-column value isn't physically stored in the database. Oracle calculates its value when you query it. The ROWID contents are displayed as base-64 values that can contain the characters A–Z, a–z, 0–9, +, and /. You can translate the ROWID value into meaningful information via the DBMS_ROWID package. For example, to display the file number, block number, and row number in which a row is stored, issue this statement:

```
select
 emp_id
,dbms_rowid.rowid_relative_fno(rowid) file_num
,dbms_rowid.rowid_block_number(rowid) block_num
,dbms_rowid.rowid_row_number(rowid)   row_num
from emp;
```

Here's some sample output:

```
    EMP_ID   FILE_NUM  BLOCK_NUM    ROW_NUM
---------- ---------- ---------- ----------
      2960          4        144        126
      2961          4        144        127
```

You can use the ROWID value in the SELECT and WHERE clauses of a SQL statement. In most cases, the ROWID uniquely identifies a row. However, it's possible to have rows in different tables that are stored in the same cluster and so contain rows with the same ROWID.

1-12. Freeing Unused Table Space

Problem

You've analyzed the output of the Segment Advisor and have identified a table that has a large amount of free space. You want to free up the unused space to improve the performance queries that perform full table scans of the table.

Solution

Do the following to shrink space and re-adjust the high-water mark for a table:

1. Enable row movement for the table.
2. Use the ALTER TABLE...SHRINK SPACE statement to free up unused space.

■ **Note** The shrink table feature requires that the table's tablespace use automatic space segment management. See Recipe 1-2 for details on how to create an ASSM-enabled tablespace.

When you shrink a table, this requires that rows (if any) be moved. This means you must enable row movement. This example enables row movement for the `INV` table:

```
SQL> alter table inv enable row movement;
```

Next the table shrink operation is executed via an `ALTER TABLE` statement:

```
SQL> alter table inv shrink space;
```

You can also shrink the space associated with any index segments via the `CASCADE` clause:

```
SQL> alter table inv shrink space cascade;
```

How It Works

When you shrink a table, Oracle re-organizes the blocks in a manner that consumes the least amount of space. Oracle also re-adjusts the table's high-water mark. This has performance implications for queries that result in full table scans. In these scenarios, Oracle will inspect every block below the high-water mark. If you notice that it takes a long time for a query to return results when there aren't many rows in the table, this may be an indication that there are many unused blocks (because data was deleted) below the high-water mark.

You can instruct Oracle to *not* re-adjust the high-water mark when shrinking a table. This is done via the `COMPACT` clause—for example:

```
SQL> alter table inv shrink space compact;
```

When you use `COMPACT`, Oracle defragments the table but doesn't alter the high-water mark. You will need to use the `ALTER TABLE…SHRINK SPACE` statement to reset the high-water mark. You might want to do this because you're concerned about the time it takes to defragment and adjust the high-water mark. This allows you to shrink a table in two shorter steps instead of one longer operation.

1-13. Compressing Data for Direct Path Loading

Problem

You're working with a decision support system (DSS)-type database and you want to improve the performance of an associated reporting application. This environment contains large tables that are loaded once and then frequently subjected to full table scans. You want to compress data as it is loaded because this will compact the data into fewer database blocks and thus will require less I/O for subsequent reads from the table. Because fewer blocks need to be read for compressed data, this will improve data retrieval performance.

Solution

Use Oracle's basic compression feature to compress direct path–loaded data into a heap-organized table. Basic compression is enabled as follows:

1. Use the `COMPRESS` clause to enable compression either when creating, altering, or moving an existing table.
2. Load data via a direct path mechanism such as `CREATE TABLE…AS SELECT` or `INSERT /*+ APPEND */`.

■ **Note** Prior to Oracle Database 11g R2, basic compression was referred to as DSS compression and enabled via the `COMPRESS FOR DIRECT_LOAD OPERATION` clause. This syntax is deprecated in Oracle Database 11g R2 and higher.

Here's an example that uses the `CREATE TABLE…AS SELECT` statement to create a basic compression-enabled table and direct path–load the data:

```
create table regs_dss
compress
as select reg_id, reg_name
from regs;
```

The prior statement creates a table with compressed data in it. Any subsequent direct path–load operations will also load the data in a compressed format.

■ **Tip** You can use either the `COMPRESS` clause or the `COMPRESS BASIC` clause to enable the basic table compression feature. The `COMPRESS` clause and `COMPRESS BASIC` clause are synonymous.

You can verify that compression has been enabled for a table by querying the appropriate `DBA/ALL/USER_TABLES` view. This example assumes that you're connected to the database as the owner of the table:

```
select table_name, compression, compress_for
from user_tables
where table_name='REGS_DSS';
```

Here is some sample output:

```
TABLE_NAME                     COMPRESS COMPRESS_FOR
------------------------------ -------- ------------
REGS_DSS                       ENABLED  BASIC
```

The prior output shows that compression has been enabled in the basic mode for this table. If you're working with a table has that already been created, then you can alter its basic compression characteristics with the `ALTER TABLE` statement—for example:

```
SQL> alter table regs_dss compress;
```

When you alter a table to enable basic compression, this does not affect any data currently existing in the table; rather it only compresses subsequent direct path data load operations.

If you want to enable basic compression for data in an existing table, use the `MOVE COMPRESS` clause:

```
SQL> alter table regs_dss move compress;
```

Keep in mind that when you move a table, all of the associated indexes are invalidated. You'll have to rebuild any indexes associated with the moved table.

If you have enabled basic compression for a table, you can disable it via the `NOCOMPRESS` clause—for example:

```
SQL> alter table regs_dss nocompress;
```

When you alter a table to disable basic compression, this does not uncompress existing data within the table. Rather this operation instructs Oracle to not compress data for subsequent direct path operations. If you need to uncompress existing compressed data, then use the `MOVE NOCOMPRESS` clause:

```
SQL> alter table regs_dss move nocompress;
```

How It Works

The basic compression feature is available at no extra cost with the Oracle Enterprise Edition. Any heap-organized table that has been created or altered to use basic compression will be a candidate for data loaded in a compressed format for subsequent direct path–load operations. There is some additional CPU overhead associated with compressing the data, but you may find in many circumstances that this overhead is offset by performance gains due to less I/O.

From a performance perspective, the main advantage to using basic compression is that once the data is loaded as compressed, any subsequent I/O operations will use fewer resources because there are fewer blocks required to read and write data. You will need to test the performance benefits for your environment. In general, tables that hold large amounts of character data are candidates for basic compression—especially in scenarios where data is direct path–loaded once, and thereafter selected from many times.

Keep in mind that Oracle's basic compression feature has no effect on regular DML statements such as `INSERT`, `UPDATE`, `MERGE`, and `DELETE`. If you require compression to occur on all DML statements, then consider using OLTP compression (see Recipe 1-14 for details).

You can also specify basic compression at the partition and tablespace level. Any table created within a tablespace created with the `COMPRESS` clause will have basic compression enabled by default. Here's an example of creating a tablespace with the `COMPRESS` clause:

```
CREATE TABLESPACE comp_data
  DATAFILE '/ora01/dbfile/O11R2/comp_data01.dbf'
  SIZE 500M
  EXTENT MANAGEMENT LOCAL
  UNIFORM SIZE 512K
  SEGMENT SPACE MANAGEMENT AUTO
  DEFAULT COMPRESS;
```

You can also alter an existing tablespace to set the default degree of compression:

```
SQL> alter tablespace comp_data default compress;
```

Run this query to verify that basic compression for a tablespace is enabled:

```
select tablespace_name, def_tab_compression, compress_for
from dba_tablespaces
where tablespace_name = 'COMP_DATA';
```

Here is some sample output:

```
TABLESPACE_NAME                DEF_TAB_ COMPRESS_FOR
------------------------------ -------- ------------
COMP_DATA                      ENABLED  BASIC
```

■ **Tip** You cannot drop a column from a table created with basic compression enabled. However, you can mark a column as unused.

1-14. Compressing Data for All DML

Problem

You're in an OLTP environment and have noticed that there is a great deal of disk I/O occurring when reading data from a table. You wonder if you can increase I/O performance by compressing the data within the table. The idea is that compressed table data will consume less physical storage and thus require less I/O to read from disk.

Solution

Use the `COMPRESS FOR OLTP` clause when creating a table to enable data compression when using regular DML statements to manipulate data. This example creates an OLTP compression–enabled table:

```
create table regs
(reg_id   number
,reg_name varchar2(2000)
) compress for oltp;
```

■ **Note** Prior to Oracle Database 11g R2, OLTP table compression was enabled using the `COMPRESS FOR ALL OPERATIONS` clause. This syntax is deprecated in Oracle Database 11g R2 and higher.

You can verify that compression has been enabled for a table by querying the appropriate `DBA/ALL/USER_TABLES` view. This example assumes that you're connected to the database as the owner of the table:

```
select table_name, compression, compress_for
from user_tables
where table_name='REGS';
```

Here is some sample output:

```
TABLE_NAME                     COMPRESS COMPRESS_FOR
------------------------------ -------- ------------
REGS                           ENABLED  OLTP
```

If you've already created the table, you can use the `ALTER TABLE` statement to enable compression on an existing table—for example:

```
SQL> alter table regs compress for oltp;
```

When you alter a table's compression mode, it doesn't impact any of the data currently within the table. Subsequent DML statements will result in data stored in a compressed fashion.

If you want to enable OLTP compression for data in an existing table, use the `MOVE COMPRESS FOR OLTP` clause:

```
SQL> alter table regs move compress for oltp;
```

Keep in mind that when you move a table, all of the associated indexes are invalidated. You'll have to rebuild any indexes associated with the moved table.

If you have enabled OLTP compression for a table, you can disable it via the `NOCOMPRESS` clause—for example:

```
SQL> alter table regs nocompress;
```

When you alter a table to disable OLTP compression, this does not uncompress existing data within the table. Rather this operation instructs Oracle to not compress data for subsequent DML operations.

How It Works

OLTP compression requires the Oracle Enterprise Edition and the Advanced Compression Option (extra cost license). The `COMPRESS FOR OLTP` clause enables compression for all DML operations. The OLTP compression doesn't immediately compress data as it is inserted or updated in a table. Rather the compression occurs in a batch mode when the degree of change within the block reaches a certain threshold. When the threshold is reached, all of the uncompressed rows are compressed at the same time. The threshold at which compression occurs is determined by an internal algorithm (over which you have no control).

You can also specify OLTP compression at the tablespace level. Any table created in an OLTP compression–enabled tablespace will by default inherit this compression setting. Here's an example of tablespace creation script specifying OLTP compression:

```
CREATE TABLESPACE comp_data
  DATAFILE '/ora01/dbfile/O11R2/comp_data01.dbf'
  SIZE 10M
  EXTENT MANAGEMENT LOCAL
  UNIFORM SIZE 1M
  SEGMENT SPACE MANAGEMENT AUTO
  DEFAULT COMPRESS FOR OLTP;
```

You can also alter an existing tablespace to set the default degree of compression:

```
SQL> alter tablespace comp_data default compress for oltp;
```

You can verify the default compression characteristics with this query:

```
select tablespace_name, def_tab_compression, compress_for
from dba_tablespaces
where tablespace_name = 'COMP_DATA';
```

Here is some sample output:

```
TABLESPACE_NAME                DEF_TAB_ COMPRESS_FOR
------------------------------ -------- ------------
COMP_DATA                      ENABLED  OLTP
```

1-15. Compressing Data at the Column Level

Problem

You're using the Oracle Exadata product and you want to efficiently compress data. You have determined that compressed data will result in much more efficient I/O operations, especially when reading data from disk. The idea is that compressed data will result in much fewer blocks read for `SELECT` statements.

Solution

To enable hybrid columnar compression, when creating a table, use either the `COMPRESS FOR QUERY` or the `COMPRESS FOR ARCHIVE` clause—for example:

```
create table f_regs(
 reg_id number
,reg_desc varchar2(4000))
compress for query;
```

You can also specify a degree of compression of either `LOW` or `HIGH`:

```
create table f_regs(
 reg_id number
,reg_desc varchar2(4000))
compress for query high;
```

The default level of compression for `QUERY` is `HIGH`, and the default level of compression for `ARCHIVE` is `LOW`. You can validate the level of compression enabled via this query:

```
select table_name, compression, compress_for
from user_tables
where table_name='F_REGS';
```

Here is some sample output:

```
TABLE_NAME                     COMPRESS COMPRESS_FOR
------------------------------ -------- ------------
F_REGS                         ENABLED  QUERY HIGH
```

If you attempt to use hybrid columnar compression in an environment other than Exadata, you'll receive the following error:

```
ERROR at line 1:
ORA-64307: hybrid columnar compression is only supported in tablespaces
residing on Exadata storage
```

How It Works

Exadata is Oracle's high-performance database machine. It is designed to deliver high performance for both data warehouse and OLTP databases. Exadata storage supports hybrid columnar compression and is available starting with Oracle Database 11g R2.

Hybrid columnar compression compresses the data on a column-by-column basis. Column-level compression results in higher compression ratios than Oracle basic compression (see Recipe 1-13) or OLTP compression (see Recipe 1-14). There are four levels of hybrid columnar compression. These levels are listed here from the lowest level of compression to the highest level:

- `COMPRESS FOR QUERY LOW`
- `COMPRESS FOR QUERY HIGH`
- `COMPRESS FOR ARCHIVE LOW`
- `COMPRESS FOR ARCHIVE HIGH`

`COMPRESS FOR QUERY` is appropriate for bulk load operations on heap-organized tables that are infrequently updated. This type of compression is optimized for query performance and is therefore more appropriate for DSS and data warehouse databases, whereas `COMPRESS FOR ARCHIVE` maximizes the degree of compression and is more appropriate for data that is stored for long periods of time and will not be updated.

■ **Note** Refer to the Oracle Exadata Storage Server Software documentation for more information on hybrid columnar compression.

1-16. Monitoring Table Usage

Problem

You've recently inherited a database that contains hundreds of tables. The application is experiencing performance issues. As part of your overall tuning strategy, you want to obtain a better understanding of the application by determining which tables are being used by what types of SQL statements. Tables that aren't being used can be renamed and later dropped. By removing unused tables, you can free up space, reduce the clutter, and focus your performance analysis on actively used tables.

Solution

Use Oracle's standard auditing feature to determine which tables are being used. Auditing is enabled as follows:

1. Set the `AUDIT_TRAIL` parameter.
2. Stop and start your database to enable the setting of `AUDIT_TRAIL`.
3. Use the `AUDIT` statement to enable auditing of specific database operations.

Oracle's standard auditing feature is enabled through setting the `AUDIT_TRAIL` initialization parameter. When you set the `AUDIT_TRAIL` parameter to `DB`, this specifies that Oracle will write audit records to an internal database table named `AUD$`. For example, when using an `spfile`, here's how to set the `AUDIT_TRAIL` parameter:

```
SQL> alter system set audit_trail=db scope=spfile;
```

If you are using an `init.ora` file, open it with a text editor and set the `AUDIT_TRAIL` value to `DB`. After you've set the `AUDIT_TRAIL` parameter, you'll need to stop and restart your database for it to take effect.

■ **Tip** When first setting up a database, we recommend that you set the `AUDIT_TRAIL` parameter to `DB`. This way, when you want to enable auditing for a specific action, you can do so without having to stop and restart (bounce) the database.

Now you can enable auditing for a specific database operation. For example, the following statement enables auditing on all DML statements on the `EMP` table owned by `INV_MGMT`:

```
SQL> audit select, insert, update, delete on inv_mgmt.emp;
```

From this point on, any DML access to the EMP table will be recorded in the SYS.AUD$ table. Oracle provides several auditing views based on the AUD$ table, such as DBA_AUDIT_TRAIL or DBA_AUDIT_OBJECT. You can query these views to report on auditing actions—for example:

```
select
  username
 ,obj_name
 ,to_char(timestamp,'dd-mon-yy hh24:mi') event_time
 ,substr(ses_actions,4,1)  del
 ,substr(ses_actions,7,1)  ins
 ,substr(ses_actions,10,1) sel
 ,substr(ses_actions,11,1) upd
from dba_audit_object;
```

Here is some sample output:

```
USERNAME                       OBJ_NAME   EVENT_TIME            DEL INS SEL UPD
------------------------------ ---------- -------------------- --- --- --- ---
INV_MGMT                       EMP        05-feb-11 15:08       -   S   -   S
INV_MGMT                       EMP        05-feb-11 15:10       -   -   S   -
INV_MGMT                       EMP        05-feb-11 15:10       S   -   -   -
```

In the prior SQL statement, notice the use of the SUBSTR function to reference the SES_ACTIONS column of the DBA_AUDIT_OBJECT view. That column contains a 16-character string in which each character means that a certain operation has occurred. The 16 characters represent the following operations in this order: ALTER, AUDIT, COMMENT, DELETE, GRANT, INDEX, INSERT, LOCK, RENAME, SELECT, UPDATE, REFERENCES, and EXECUTE. Positions 14, 15, and 16 are reserved by Oracle for future use. The character of S represents success, F represents failure, and B represents both success and failure.

To turn off auditing on an object, use the NOAUDIT statement:

```
SQL> noaudit select, insert, update, delete on inv_mgmt.emp;
```

■ **Tip** If you need to view the SQL_TEXT or SQL_BIND columns of the AUD$ table, then set the AUDIT_TRAIL initialization parameter to DB_EXTENDED.

How It Works

Sometimes it's handy when troubleshooting disk space or performance issues to know which tables in the database are actually being used by the application. If you've inherited a database that contains a large number of tables, it may not be obvious which objects are being accessed. Enabling auditing allows you to identify which types of SQL statements are accessing a table of interest.

Once you have identified tables that are not being used, you can simply rename the tables and see if this breaks the application or if any users complain. If there are no complaints, then after some time you can consider dropping the tables. Make sure you take a good backup of your database with both RMAN and Data Pump before you drop any tables you might have to later recover.

If you simply need to know whether a table is being inserted, updated, or deleted from, you can use the `DBA/ALL/USER_TAB_MODIFICATIONS` view to report on that type of activity. This view has columns, such as `INSERTS`, `UPDATES`, `DELETES`, and `TRUNCATED`, that will provide information as to how data in the table is being modified—for example:

```
select table_name, inserts, updates, deletes, truncated
from user_tab_modifications;
```

In normal conditions, this view is not instantly updated by Oracle. If you need to immediately view table modifications, then use the `DBMS_STATS.FLUSH_DATABASE_MONITORING_INFO` procedure to update the view:

```
SQL> exec DBMS_STATS.FLUSH_DATABASE_MONITORING_INFO();
```

CHAPTER 2

Choosing and Optimizing Indexes

An index is a database object used primarily to improve the performance of SQL queries. The function of a database index is similar to an index in the back of a book. A book index associates a topic with a page number. When you're locating information in a book, it's usually much faster to inspect the index first, find the topic of interest, and identify associated page numbers. With this information, you can navigate directly to specific page numbers in the book. In this situation, the number of pages you need to inspect is minimal.

If there were no index, you would have to inspect every page of the book to find information. This results in a great deal of page turning, especially with large books. This is similar to an Oracle query that does not use an index and therefore has to scan every used block within a table. For large tables, this results in a great deal of I/O.

The book index's usefulness is directly correlated with the uniqueness of a topic within the book. For example, take this book; it would do no good to create an index on the topic of "performance" because every page in this book deals with performance. However, creating an index on the topic of "bitmap indexes" would be effective because there are only a few pages within the book that are applicable to this feature.

Keep in mind that the index isn't free. It consumes space in the back of the book, and if the material in the book is ever updated (like a second edition), every modification (insert, update, delete) potentially requires a corresponding change to the index. It's important to keep in mind that indexes consume space and require resources when updates occur.

Also, the person who creates the index for the book must consider which topics will be frequently looked up. Topics that are selective and frequently accessed should be included in the book index. If an index in the back of the book is never looked up by a reader, then it unnecessarily wastes space.

Much like the process of creating an index in the back of the book, there are many factors that must be considered when creating an Oracle index. Oracle provides a wide assortment of indexing features and options. These objects are manually created by the DBA or a developer. Therefore, you need to be aware of the various features and how to utilize them. If you choose the wrong type of index or use a feature incorrectly, there may be detrimental performance implications. Listed next are aspects to consider before you create an index:

- Type of index
- Table column(s) to include
- Whether to use a single column or a combination of columns
- Special features such as parallelism, turning off logging, compression, invisible indexes, and so on

- Uniqueness
- Naming conventions
- Tablespace placement
- Initial sizing requirements and growth
- Impact on performance of `SELECT` statements (improvement)
- Impact on performance of `INSERT`, `UPDATE`, and `DELETE` statements
- Global or local index, if the underlying table is partitioned

When you create an index, you should give some thought to every aspect mentioned in the previous list. One of the first decisions you need to make is the type of index and the columns to include. Oracle provides a robust variety of index types. For most scenarios, you can use the default B-tree (balanced tree) index. Other commonly used types are concatenated, bitmap, and function-based indexes. Table 2-1 describes the types of indexes available with Oracle.

Table 2-1. *Oracle Index Type Descriptions*

Index Type	Usage
B-tree	Default, balanced tree index, good for high-cardinality (high degree of distinct values) columns
B-tree cluster	Used with clustered tables
Hash cluster	Used with hash clusters
Function-based	Good for columns that have SQL functions applied to them
Indexed virtual column	Good for columns that have SQL functions applied to them; viable alternative to using a function-based index
Reverse-key	Useful to balance I/O in an index that has many sequential inserts
Key-compressed	Useful for concatenated indexes where the leading column is often repeated; compresses leaf block entries
Bitmap	Useful in data warehouse environments with low-cardinality columns; these indexes aren't appropriate for online transaction processing (OLTP) databases where rows are heavily updated.
Bitmap join	Useful in data warehouse environments for queries that join fact and dimension tables

Index Type	Usage
Global partitioned	Global index across all partitions in a partitioned table
Local partitioned	Local index based on individual partitions in a partitioned table
Domain	Specific for an application or cartridge

This chapter focuses on the most commonly used indexes and features. Hash cluster indexes, partitioned indexes, and domain indexes are not covered in this book. If you need more information regarding index types or features not covered in this chapter or book, see Oracle's SQL Reference Guide at `http://otn.oracle.com`.

The first recipe in this chapter deals with the mechanics of B-tree indexes. It's critical that you understand how this database object works. Even if you've been around Oracle for a while, we feel it's useful to work through the various scenarios outlined in this first recipe to ensure that you know how the optimizer uses this type of index. This will lay the foundation for solving many different types of performance problems (especially SQL tuning).

2-1. Understanding B-tree Indexes

Problem

You want to create an index. You understand that the default type of index in Oracle is the B-tree, but you don't quite understand how an index is physically implemented. You want to fully comprehend the B-tree index internals so as to make intelligent performance decisions when building database applications.

Solution

An example with a good diagram will help illustrate the mechanics of a B-tree index. Even if you've been working with B-tree indexes for quite some time, a good example may illuminate technical aspects of using an index. To get started, suppose you have a table created as follows:

```
create table cust(
 cust_id number
,last_name varchar2(30)
,first_name varchar2(30));
```

You determine that several SQL queries will frequently use `LAST_NAME` in the `WHERE` clause. This prompts you to create an index:

```
SQL> create index cust_idx1 on cust(last_name);
```

Several hundred rows are now inserted into the table (not all of the rows are shown here):

```
insert into cust values(7, 'ACER','SCOTT');
insert into cust values(5, 'STARK','JIM');
insert into cust values(3, 'GREY','BOB');
```

```
insert into cust values(11,'KHAN','BRAD');
.....
insert into cust values(274, 'ACER','SID');
```

After the rows are inserted, we ensure that the table statistics are up to date so as to provide the query optimizer sufficient information to make good choices on how to retrieve the data:

```
SQL> exec dbms_stats.gather_table_stats(ownname=>'MV_MAINT', -
          tabname=>'CUST',cascade=>true);
```

As rows are inserted into the table, Oracle will allocate extents that consist of physical database blocks. Oracle will also allocate blocks for the index. For each record inserted into the table, Oracle will also create an entry in the index that consists of the `ROWID` and column value (the value in `LAST_NAME` in this example). The `ROWID` for each index entry points to the datafile and block that the table column value is stored in. Figure 2-1 shows a graphical representation of how data is stored in the table and the corresponding B-tree index. For this example, datafiles 10 and 15 contain table data stored in associated blocks and datafile 22 stores the index blocks.

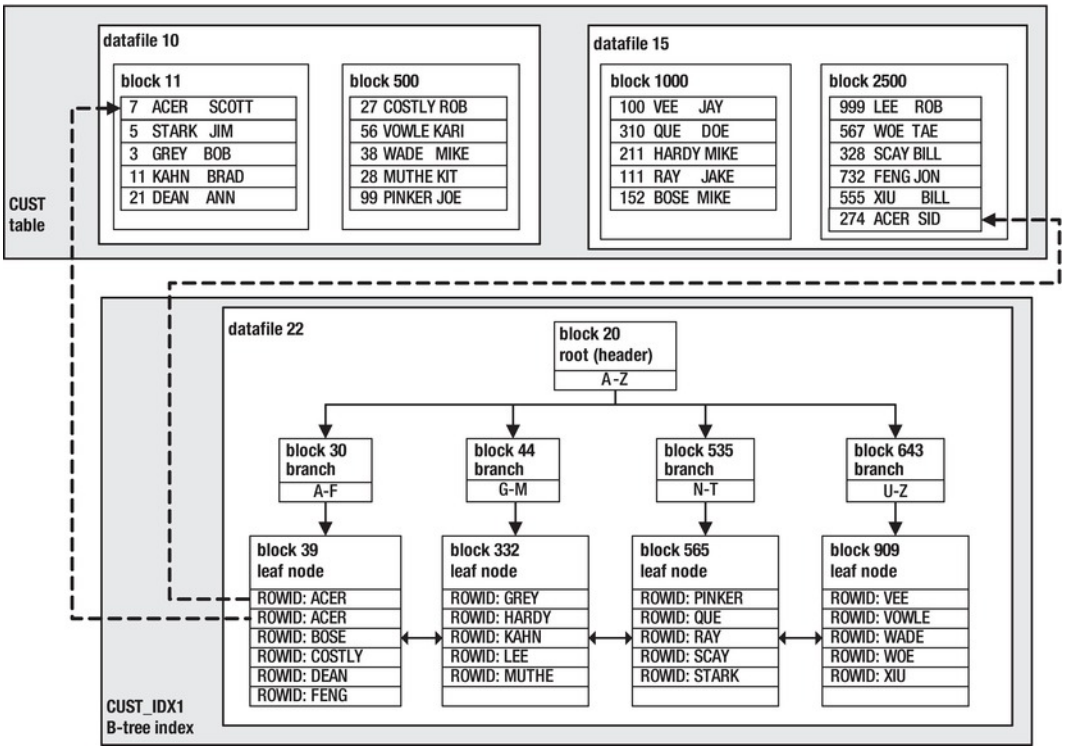

Figure 2-1. *Physical layout of a table and B-tree index*

There are two dotted lines in Figure 2-1. These lines depict how the ROWID (in the index structure) points to the physical location in the table for the column values of ACER. These particular values will be used in the scenarios in this solution.

When selecting data from a table and its corresponding index, there are three basic scenarios:

- All table data required by the SQL query is contained in the index structure. Therefore only the index blocks need to be accessed. The blocks from the table are never read.
- All of the information required by the query is not contained in the index blocks. Therefore the query optimizer chooses to access both the index blocks and the table blocks to retrieve the data needed to satisfy the results of the query.
- The query optimizer chooses not to access the index. Therefore only the table blocks are accessed.

The prior situations are covered in the next three subsections.

Scenario 1: All Data Lies in the Index Blocks

There are two scenarios that will be shown in this section:

- *Index range scan*: This occurs when the optimizer determines it is efficient to use the index structure to retrieve multiple rows required by the query. Index range scans are used extensively in a wide variety of situations.
- *Index fast full scan*: This occurs when the optimizer determines that most of the rows in the table will need to be retrieved. However, all of the information required is stored in the index. Since the index structure is usually smaller than the table structure, the optimizer determines that a full scan of the index is more efficient. This scenario is common for queries that count values.

First the index range scan is demonstrated. For this example, suppose this query is issued that selects from the table:

```
SQL> select last_name from cust where last_name='ACER';
```

Before reading on, look at Figure 2-1 and try to answer this question: "What are the minimal number of blocks Oracle will need to read to return the data for this query?" In other words, what is the most efficient way to access the physical blocks in order to satisfy the results of this query? The optimizer could choose to read through every block in the table structure. However, that would result in a great deal of I/O, and thus it is not the most optimal way to retrieve the data.

For this example, the most efficient way to retrieve the data is to use the index structure. To return the rows that contain the value of ACER in the LAST_NAME column, Oracle will need to read three blocks: block 20, block 30, and block 39. We can verify that this is occurring by using Oracle's Autotrace utility:

```
SQL> set autotrace on;
SQL> select last_name from cust where last_name='ACER';
```

Here is a partial snippet of the output:

```
-------------------------------------------------------------------------------
| Id  | Operation        | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT |           |   101 |   808 |     1   (0)| 00:00:01 |
|*  1 |  INDEX RANGE SCAN| CUST_IDX1 |   101 |   808 |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------
```

The prior output shows that Oracle needed to use only the `CUST_IDX1` index to retrieve the data to satisfy the result set of the query. *The table data blocks were not accessed*; only the index blocks were required. This is a particularly efficient indexing strategy for the given query. Listed next are the statistics displayed by Autotrace for this example:

```
Statistics
----------------------------------------------------------
          1  recursive calls
          0  db block gets
          3  consistent gets
          0  physical reads
```

The `consistent gets` value indicates that three blocks were read from memory (`db block gets` plus `consistent gets` equals the total blocks read from memory). Since the index blocks were already in memory, no physical reads were required to return the result set of this query.

Next an example that results in an index fast full scan is demonstrated. Consider this query:

```
SQL> select count(last_name) from cust;
```

Using `SET AUTOTRACE ON`, an execution plan is generated. Here is the corresponding output:

```
-------------------------------------------------------------------------------------
| Id  | Operation             | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |           |     1 |     8 |     3   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE       |           |     1 |     8 |            |          |
|   2 |   INDEX FAST FULL SCAN| CUST_IDX1 |  1509 | 12072 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------------
```

The prior output shows that only the index structure was used to determine the count within the table. In this situation, the optimizer determined that a full scan of the index was more efficient than a full scan of the table.

Scenario 2: All Information Is Not Contained in the Index

Now consider this situation: suppose we need additional information from the `CUST` table. This query additionally selects the `FIRST_NAME` column:

```
SQL> select last_name, first_name from cust where last_name = 'ACER';
```

Using SET AUTOTRACE ON and executing the prior query results in the following execution plan:

```
-----------------------------------------------------------------------------------------
| Id  | Operation                    | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |           |   101 |  1414 |     3   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| CUST      |   101 |  1414 |     3   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN           | CUST_IDX1 |   101 |       |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------
```

The prior output indicates that the CUST_IDX1 index was accessed via an INDEX RANGE SCAN. The INDEX RANGE SCAN identifies the index blocks required to satisfy the results of this query. Additionally the table is read by TABLE ACCESS BY INDEX ROWID. The access to the table by the index's ROWID means that Oracle uses the ROWID (stored in the index) to locate the data contained within the table blocks. In Figure 2-1, this is indicated by the dotted lines that map the ROWID to the appropriate table blocks that contain the value of ACER in the LAST_NAME column.

Again, looking at Figure 2-1, how many table and index blocks need to be read in this scenario? The index requires that blocks 20, 30, and 39 must be read. Since FIRST_NAME is not included in the index, Oracle must read the table blocks to retrieve these values. Oracle knows the ROWID of the table blocks and directly reads blocks 11 and 2500 to retrieve that data. That makes a total of five blocks. Here is a partial snippet of the statistics generated by Autotrace that confirms the number of blocks read is five:

```
Statistics
----------------------------------------------------------
          1  recursive calls
          0  db block gets
          5  consistent gets
          0  physical reads
```

Scenario 3: Only the Table Blocks Are Accessed

In some situations, even if there is an index, Oracle will determine that it's more efficient to use only the table blocks. When Oracle inspects every row within a table, this is known a *full table scan*. For example, take this query:

```
SQL> select * from cust;
```

Here are the corresponding execution plan and statistics:

```
-----------------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |  1509 | 24144 |    12   (0)| 00:00:01 |
|   1 |  TABLE ACCESS FULL| CUST |  1509 | 24144 |    12   (0)| 00:00:01 |
-----------------------------------------------------------------------------
Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
        119  consistent gets
          0  physical reads
```

The prior output shows that a total of 119 blocks were inspected. Oracle searched every row in the table to bring back the results required to satisfy the query. In this situation, all blocks of the table must be read, and there is no way for Oracle to use the index to speed up the retrieval of the data.

■ **Note** For the examples in this recipe, your results may vary slightly, depending on how many rows you initially insert into the table. We used approximately 1,500 rows for this example.

How It Works

The B-tree index is the default index type in Oracle. For most OLTP-type applications, this index type is sufficient. This index type is known as B-tree because the `ROWID` and associated column values are stored within blocks in a *balanced* tree-like structure (see Figure 2-1). The B stands for balanced.

B-tree indexes are efficient because, when properly used, they result in a query retrieving data far faster than it would without the index. If the index structure itself contains the required column values to satisfy the result of the query, then the table data blocks need not be accessed. Understanding these mechanics will guide your indexing decision-making process. For example, this will help you decide which columns to index and whether a concatenated index might be more efficient for certain queries and less optimal for others. These topics are covered in detail in subsequent recipes in this chapter.

ESTIMATING THE SPACE AN INDEX WILL REQUIRE

Before you create an index, you can estimate how much space it will take via the `DBMS_SPACE.CREATE_INDEX_COST` procedure—for example:

```
SQL> set serverout on
SQL> exec dbms_stats.gather_table_stats(user,'CUST');
SQL> variable used_bytes number
SQL> variable alloc_bytes number
SQL> exec dbms_space.create_index_cost( 'create index cust_idx2 on cust(first_name)', -
               :used_bytes, :alloc_bytes );
SQL> print :used_bytes
```

Here is some sample output for this example:

```
USED_BYTES
----------
    363690

SQL> print :alloc_bytes
```

Here is some sample output for this example:

```
ALLOC_BYTES
-----------
    2097152
```

The `used_bytes` variable gives you an estimate of how much room is required for the index data. The `alloc_bytes` variable provides an estimate of how much space will be allocated within the tablespace.

2-2. Deciding Which Columns to Index

Problem

A database you manage contains hundreds of tables. Each table typically contains a dozen or more columns. You wonder which columns should be indexed.

Solution

Listed next are general guidelines for deciding which columns to index.

- Define a primary key constraint for each table that results in an index automatically being created on the columns specified in the primary key (see Recipe 2-3).
- Create unique key constraints on non-null column values that are required to be unique (different from the primary key columns). This results in an index automatically being created on the columns specified in unique key constraints (see Recipe 2-4).
- Explicitly create indexes on foreign key columns (see Recipe 2-5).
- Create indexes on columns used often as predicates in the `WHERE` clause of frequently executed SQL queries.

After you have decided to create indexes, we recommend that you adhere to index creation standards that facilitate the ease of maintenance. Specifically, follow these guidelines when creating an index:

- Use the default B-tree index unless you have a solid reason to use a different index type.
- Create a separate tablespace for the indexes. This allows you to more easily manage indexes separately from tables for tasks such as backup and recovery.
- Let the index inherit its storage properties from the tablespace. This allows you to specify the storage properties when you create the tablespace and not have to manage storage properties for individual indexes.
- If you have a variety of storage requirements for indexes, then consider creating separate tablespaces for each type of index—for example, `INDEX_LARGE`, `INDEX_MEDIUM`, and `INDEX_SMALL` tablespaces, each defined with storage characteristics appropriate for the size of the index.

Listed next is a sample script that encapsulates the foregoing recommendations from the prior two bulleted lists:

```
CREATE TABLE cust(
 cust_id    NUMBER
,last_name  VARCHAR2(30)
,first_name VARCHAR2(30));
--
ALTER TABLE cust ADD CONSTRAINT cust_pk PRIMARY KEY (cust_id)
USING INDEX TABLESPACE reporting_index;
--
ALTER TABLE cust ADD CONSTRAINT cust_uk1 UNIQUE (last_name, first_name)
USING INDEX TABLESPACE reporting_index;
--
CREATE TABLE address(
 address_id NUMBER,
 cust_id    NUMBER
,street     VARCHAR2(30)
,city       VARCHAR2(30)
,state      VARCHAR2(30))
TABLESPACE reporting_data;
--
ALTER TABLE address ADD CONSTRAINT addr_fk1
FOREIGN KEY (cust_id) REFERENCES cust(cust_id);
--
CREATE INDEX addr_fk1 ON address(cust_id)
TABLESPACE reporting_index;
```

In the prior script, two tables are created. The parent table is `CUST` and its primary key is `CUST_ID`. The child table is `ADDRESS` and its primary key is `ADDRESS_ID`. The `CUST_ID` column exists in `ADDRESS` as a foreign key mapping back to the `CUST_ID` column in the `CUST` table.

How It Works

You should add an index only when you're certain it will improve performance. Misusing indexes can have serious negative performance effects. Indexes created of the wrong type or on the wrong columns do nothing but consume space and processing resources. As a DBA, you must have a strategy to ensure that indexes enhance performance and don't negatively impact applications.

Table 2-2 encapsulates many of the index management concepts covered in this chapter. These recommendations aren't written in stone: adapt and modify them as needed for your environment.

Table 2-2. *Index Creation and Maintenance Guidelines*

Guideline	Reasoning
Add indexes judiciously. Test first to determine quantifiable performance gains.	Indexes consume disk space and processing resources. Don't add indexes unnecessarily.
Use the correct type of index.	Correct index usage maximizes performance. See Table 2-1 for more details.
Use consistent naming standards.	This makes maintenance and troubleshooting easier.
Monitor your indexes, and drop indexes that aren't used. See Recipe 2-15 for details on monitoring indexes.	Doing this frees up physical space and improves the performance of Data Manipulation Language (DML) statements.
Don't rebuild indexes unless you have a solid reason to do so. See Recipe 2-17 for details on rebuilding an index.	Rebuilding an index is generally unnecessary unless the index is corrupt or you want to change a physical characteristic (such as the tablespace) without dropping the index.
Before dropping an index, consider marking it as unusable or invisible.	This allows you to better determine if there are any performance issues before you drop the index. These options let you rebuild or re-enable the index without requiring the Data Definition Language (DDL) index creation statement.
Consider creating concatenated indexes that result in only the index structure being required to return the result set.	Avoids having to scan any table blocks; when queries are able to use the index only, this results in very efficient execution plans.
Consider creating indexes on columns used in the `ORDER BY`, `GROUP BY`, `UNION`, or `DISTINCT` clauses.	This may result in more efficient queries that frequently use these SQL constructs.

Refer to these guidelines as you create and manage indexes in your databases. These recommendations are intended to help you correctly use index technology.

INDEXES WITH NO SEGMENTS

You can instruct Oracle to create an index that will never be used and won't have any extents allocated to it via the `NOSEGMENT` clause:

```
SQL> create index cust_idx1 on cust(first_name) nosegment;
```

Even though this index will never be used, you can instruct Oracle to determine if the index might be used by the optimizer via the `_USE_NOSEGMENT_INDEXES` initialization parameter—for example:

```
SQL> alter session set "_use_nosegment_indexes"=true;
SQL> set autotrace trace explain;
SQL> select first_name from cust where first_name = 'JIM';
```

Here's a sample execution plan showing the optimizer would use the index (assuming that you dropped and re-created it normally without the NOSEGMENT clause):

```
-------------------------------------------------------------------------------
 Id  | Operation        | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
   0 | SELECT STATEMENT |           |     1 |    17 |     1   (0)| 00:00:01 |
*  1 |  INDEX RANGE SCAN| CUST_IDX1 |     1 |    17 |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------
```

That begs the question, why would you ever create an index with NOSEGMENT? If you have a very large index that you want to create without allocating space, to determine if the index would be used by the optimizer, creating an index with NOSEGMENT allows you to test that scenario. If you determine that the index would be useful, you can drop the index and re-create it without the NOSEGMENT clause.

2-3. Creating a Primary Key Index

Problem

You want to enforce that the primary key columns are unique within a table. Furthermore many of the columns in the primary key are frequently used within the WHERE clause of several queries. You want to ensure that indexes are created on primary key columns.

Solution

When you define a primary key constraint for a table, Oracle will automatically create an associated index for you. There are several methods available for creating a primary key constraint. Our preferred approach is to use the ALTER TABLE...ADD CONSTRAINT statement. This will create the index and the constraint at the same time. This example creates a primary key constraint named CUST_PK and also instructs Oracle to create the corresponding index (also named CUST_PK) in the USERS tablespace:

```
alter table cust add constraint cust_pk primary key (cust_id)
using index tablespace users;
```

The following queries and output provide details about the constraint and index that Oracle created. The first query displays the constraint information:

```
select
 constraint_name
,constraint_type
from user_constraints
where table_name = 'CUST';
```

```
CONSTRAINT_NAME               C
----------------------------- -
CUST_PK                       P
```

This query displays the index information:

```
select
 index_name
,tablespace_name
,index_type
,uniqueness
from user_indexes
where table_name = 'CUST';

INDEX_NAME      TABLESPACE_NAME INDEX_TYPE      UNIQUENESS
--------------- --------------- --------------- ---------------
CUST_PK         USERS           NORMAL          UNIQUE
```

How It Works

The solution for this recipe shows the method that we prefer to create primary key constraints and the corresponding index. In most situations, this approach is acceptable. However, you should be aware that there are several other methods for creating the primary key constraint and index. These methods are listed here:

- Create an index first, and then use `ALTER TABLE...ADD CONSTRAINT`.
- Specify the constraint inline (with the column) in the `CREATE TABLE` statement.
- Specify the constraint out of line (from the column) within the `CREATE TABLE` statement.

These techniques are described in the next several subsections.

Create Index and Constraint Separately

You have the option of first creating an index and then altering the table to apply the primary key constraint. Here's an example:

```
SQL> create index cust_pk on cust(cust_id);
SQL> alter table cust add constraint cust_pk primary key(cust_id);
```

The advantage to this approach is that you can drop or disable the primary key constraint independently of the index. If you work with large data volumes, you may require this sort of flexibility. This approach allows you to disable/re-enable a constraint without having to later rebuild the index.

Create Constraint Inline

You can directly create an index inline (with the column) in the `CREATE TABLE` statement. This approach is simple but doesn't allow for multiple column primary keys and doesn't name the constraint:

```
SQL> create table cust(cust_id number primary key);
```

If you don't explicitly name the constraint (as in the prior statement), Oracle automatically generates a name like SYS_C123456. If you want to explicitly provide a name, you can do so as follows:

```
create table cust(cust_id number constraint cust_pk primary key
using index tablespace users);
```

The advantage of this approach is that it's very simple. If you're experimenting in a development or test environment, this approach is quick and effective.

Create Constraint Out of Line

You can also define the primary key constraint out of line (from the column) within the CREATE TABLE statement:

```
create table cust(cust_id number
,constraint cust_pk primary key (cust_id)
using index tablespace users);
```

The out-of-line approach has one advantage over the inline approach in that you can specify multiple columns for the primary key.

All of the prior techniques for creating a primary key constraint and corresponding index are valid. It's often a matter of DBA or developer preference as to which technique is used.

2-4. Creating a Unique Index

Problem

You have a column (or combination of columns) that contains values that should always be unique. You want to create an index on this column (or combination of columns) that enforces the uniqueness and also provides efficient access to the table when using the unique column in the WHERE clause of a query.

■ **Note** If you want to create a unique constraint on the primary key column(s), then you should explicitly create a primary key constraint (see Recipe 2-3 for details). One difference between a primary key and a unique key is that you can have only one primary key definition per table, whereas you can have multiple unique keys. Also, unique key constraints allow for null values, whereas primary key constraints do not.

Solution

This solution focuses on using the ALTER TABLE...ADD CONSTRAINT statement. When you create a unique key constraint, Oracle will automatically create an index for you. This is our recommended approach for creating unique key constraints and indexes. This example creates a unique constraint named CUST_UX1 on the combination of the LAST_NAME and FIRST_NAME columns of the CUST table:

```
alter table cust add constraint cust_ux1 unique (last_name, first_name)
```

```
using index tablespace users;
```

The prior statement creates the unique constraint, and additionally Oracle automatically creates an associated index. The following query displays the constraint that was created successfully:

```
select
 constraint_name
,constraint_type
from user_constraints
where table_name = 'CUST';
```

Here is a snippet of the output:

```
CONSTRAINT_NAME                C
------------------------------ -
CUST_UX1                       U
```

This query shows the index that was automatically created along with the constraint:

```
select
 index_name
,tablespace_name
,index_type
,uniqueness
from user_indexes
where table_name = 'CUST';
```

Here is some sample output:

```
INDEX_NAME           TABLESPACE INDEX_TYPE UNIQUENESS
-------------------- ---------- ---------- ---------
CUST_UX1             USERS      NORMAL     UNIQUE
```

How It Works

Defining a unique constraint ensures that when you insert or update column values, then any combination of non-null values are unique. Besides the approach we displayed in the "Solution" section, there are several additional techniques for creating unique constraints:

- Use the `CREATE TABLE` statement.
- Create a regular index, and then use `ALTER TABLE` to add a constraint.
- Create a unique index and don't add the constraint.

These techniques are described in the next few subsections.

Use CREATE TABLE

Listed next is an example of using the `CREATE TABLE` statement to include a unique constraint.

```
create table cust(
 cust_id number
,last_name varchar2(30)
,first_name varchar2(30)
,constraint cust_ux1 unique(last_name, first_name)
 using index tablespace users);
```

The advantage of this approach is that it's simple and encapsulates the constraint and index creation within one statement.

Create Index First, Then Add Constraint

You have the option of first creating an index and then adding the constraint as a separate statement—for example:

```
SQL> create unique index cust_uidx1 on cust(last_name, first_name) tablespace users;
SQL> alter table cust add constraint cust_uidx1 unique (last_name, first_name);
```

The advantage of creating the index separate from the constraint is that you can drop or disable the constraint without dropping the underlying index. When working with large indexes, you may want to consider this approach. If you need to disable the constraint for any reason and then re-enable it later, you can do so without dropping the index (which may take a long time for large indexes).

Creating Only a Unique Index

You can also create just a unique index without adding the unique constraint—for example:

```
SQL> create unique index cust_uidx1 on cust(last_name, first_name) tablespace users;
```

When you create only a unique index explicitly (as in the prior statement), Oracle creates a unique index but doesn't add an entry for a constraint in `DBA/ALL/USER_CONSTRAINTS`. Why does this matter? Consider this scenario:

```
SQL> insert into cust values (1, 'STARK', 'JIM');
SQL> insert into cust values (1, 'STARK', 'JIM');
```

Here's the corresponding error message that is thrown:

```
ERROR at line 1:
ORA-00001: unique constraint (MV_MAINT.CUST_UIDX1) violated
```

If you're asked to troubleshoot this issue, the first place you look is in `DBA_CONSTRAINTS` for a constraint named `CUST_UIDX1`. However, there is no information:

```
select
  constraint_name
from dba_constraints
where constraint_name='CUST_UIDX1';
no rows selected
```

The "no rows selected" message can be confusing: the error message thrown when you insert into the table indicates that a unique constraint has been violated, yet there is no information in the constraint-related data-dictionary views. In this situation, you have to look at `DBA_INDEXES` to view the details of the unique index that has been created—for example:

```
select index_name, uniqueness
from dba_indexes where index_name='CUST_UIDX1';

INDEX_NAME                     UNIQUENESS
------------------------------ ----------
CUST_UIDX1                     UNIQUE
```

2-5. Indexing Foreign Key Columns

Problem

A large number of the queries in your application use foreign key columns as predicates in the `WHERE` clause. Therefore, for performance reasons, you want to ensure that you have all foreign key columns indexed.

Solution

Unlike primary key constraints, Oracle does not automatically create indexes on foreign key columns. For example, say you have a requirement that every record in the `ADDRESS` table be assigned a corresponding `CUST_ID` column that exists in the `CUST` table. To enforce this relationship, you create a foreign key constraint on the `ADDRESS` table as follows:

```
alter table address add constraint addr_fk1
foreign key (cust_id) references cust(cust_id);
```

■ **Note** A foreign key column must reference a column in the parent table that has a primary key or unique key constraint defined on it. Otherwise you'll receive the error "ORA-02270: no matching unique or primary key for this column-list."

You realize the foreign key column is used extensively when joining the `CUST` and `ADDRESS` tables and that an index on the foreign key column will dramatically increase performance. You have to manually create an index in this situation. For example, a regular B-tree index is created on the foreign key column of `CUST_ID` in the `ADDRESS` table:

```
SQL> create index addr_fk1 on address(cust_id);
```

You don't have to name the index the same as the foreign key name (as we did in the prior lines of code). It's a personal preference as to whether you do that. We feel it's easier to maintain environments when the constraint and corresponding index have the same name.

How It Works

Foreign keys exist to ensure that when inserting into a child table, a corresponding parent table record exists. This is the mechanism to guarantee that data conforms to parent/child business relationship rules. From a performance perspective, it's usually a good idea to create an index on foreign key columns. This is because parent/child tables are frequently joined on the foreign key column(s) in the child table to the primary key column(s) in the parent table—for example:

```
select
 a.last_name, a.first_name, b.state
from cust a
    ,address b
where a.cust_id = b.cust_id;
```

In most scenarios, the Oracle query optimizer will choose to use the index on the foreign key column to identify the child records that are required to satisfy the results of the query. If no index exists, Oracle has to perform a full table scan on the child table.

If you've inherited a database, then it's prudent to check if columns with foreign key constraints defined on them have a corresponding index. The following query displays indexes associated with foreign key constraints:

```
select
  a.constraint_name cons_name
 ,a.table_name  tab_name
 ,b.column_name cons_column
 ,nvl(c.column_name,'***No Index***') ind_column
from user_constraints  a
     join
     user_cons_columns b on a.constraint_name = b.constraint_name
     left outer join
     user_ind_columns  c on b.column_name = c.column_name
                        and b.table_name  = c.table_name
where constraint_type = 'R'
order by 2,1;
```

If there is no index on the foreign key column, the ***No Index*** message is displayed. For example, suppose the index in the "Solution" section was accidentally dropped and then the prior query was run. Here is some sample output:

```
CONS_NAME          TAB_NAME                CONS_COLUMN              IND_COLUMN
------------------ ----------------------- ------------------------ --------------------
ADDR_FK1                 ADDRESS                   CUST_ID                   ***No Index***
```

2-6. Deciding When to Use a Concatenated Index

Problem

You have a combination of columns (from the same table) that are often used in the `WHERE` clause of several SQL queries. For example, you use `LAST_NAME` in combination with `FIRST_NAME` to identify a customer:

```
select last_name, first_name
from cust
where last_name = 'SMITH'
and first_name = 'STEVE';
```

You wonder if it would be more efficient to create a single concatenated index on the combination of LAST_NAME and FIRST_NAME columns or if performance would be better if two indexes were created separately on LAST_NAME and FIRST_NAME.

Solution

When frequently accessing two or more columns in conjunction in the WHERE clause, a concatenated index is often more selective than two single indexes. For this example, here's the table creation script:

```
create table cust(
 cust_id number primary key
,last_name varchar2(30)
,first_name varchar2(30));
```

Here's an example of a concatenated index created on LAST_NAME and FIRST_NAME:

```
SQL> create index cust_idx1 on cust(last_name, first_name);
```

To determine whether the concatenated index is used, several rows are inserted (only a subset of the rows is shown here):

```
SQL> insert into cust values(1,'SMITH','JOHN');
SQL> insert into cust values(2,'JONES','DAVE');
..........
SQL> insert into cust values(3,'FORD','SUE');
```

Next, statistics are generated for the table and index:

```
SQL> exec dbms_stats.gather_table_stats(ownname=>'MV_MAINT',-
           tabname=>'CUST',cascade=>true);
```

Now Autotrace is turned on so that the execution plan is displayed when a query is run:

```
SQL> set autotrace on;
```

Here's the query to execute:

```
select last_name, first_name
from cust
where last_name = 'SMITH'
and first_name = 'JOHN';
```

Listed next is an explain plan that shows the optimizer is using the index:

```
-------------------------------------------------------------------------------
| Id  | Operation        | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT |           |    13 |   143 |     1   (0)| 00:00:01 |
|*  1 |  INDEX RANGE SCAN| CUST_IDX1 |    13 |   143 |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------
```

The prior output indicates that an INDEX RANGE SCAN was used to access the CUST_IDX1 index. Notice that all of the information required to satisfy the results of this query was contained within the index. The table data was not required. Oracle accessed only the index.

One other item to consider: suppose you have this query that additionally selects the CUST_ID column:

```
select cust_id, last_name, first_name
from cust
where last_name = 'SMITH'
and first_name = 'JOHN';
```

If you frequently access CUST_ID in combination with LAST_NAME and FIRST_NAME, consider adding CUST_ID to the concatenated index. This will provide all of the information that the query needs in the index. Oracle will be able to retrieve the required data from the index blocks and thus not have to access the table blocks.

How It Works

Oracle allows you to create an index that contains more than one column. Multicolumn indexes are known as *concatenated indexes*. These indexes are especially effective when you often use multiple columns in the WHERE clause when accessing a table. Here are some factors to consider when using concatenated indexes:

- If columns are often used together in the WHERE clause, consider creating a concatenated index.
- If a column is also used (in other queries) by itself in the WHERE clause, place that column at the leading edge of the index (first column defined).
- Keep in mind that Oracle can still use a lagging edge index (not the first column defined) if the lagging column appears by itself in the WHERE clause (see the next few paragraphs here for details).

In older versions of Oracle (circa v8), the optimizer would use a concatenated index only if the leading edge column(s) appeared in the WHERE clause. In modern versions, the optimizer uses a concatenated index even if the leading edge column(s) aren't present in the WHERE clause. This ability to use an index without reference to leading edge columns is known as the *skip-scan* feature. For example, say you have this query that uses the FIRST_NAME column (which is a lagging column in the concatenated index created in the "Solution" section of this recipe):

```
SQL> select last_name from cust where first_name='DAVE';
```

Here is the corresponding explain plan showing that the skip-scan feature is in play:

```
-------------------------------------------------------------------------------
| Id  | Operation        | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT |           |    38 |   418 |     1   (0)| 00:00:01 |
|*  1 |  INDEX SKIP SCAN | CUST_IDX1 |    38 |   418 |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------
```

A concatenated index that is used for skip-scanning is more efficient than a full table scan. However, if you're consistently using only a lagging edge column of a concatenated index, then consider creating a single-column index on the lagging column.

2-7. Reducing Index Size Through Compression

Problem

You want to create an index that efficiently handles cases in which many rows have the same values in one or more indexed columns. For example, suppose you have a table defined as follows:

```
create table cust(
 cust_id number
,last_name varchar2(30)
,first_name varchar2(30)
,middle_name varchar2(30));
```

Furthermore, you inspect the data inserted into the prior table with this query:

```
SQL> select last_name, first_name, middle_name from cust;
```

You notice that there is a great deal of duplication in the LAST_NAME and FIRST_NAME columns:

```
LEE                  JOHN                 Q
LEE                  JOHN                 B
LEE                  JOHN                 A
LEE                  JOE                  D
SMITH                BOB                  A
SMITH                BOB                  C
SMITH                BOB                  D
SMITH                JOHN                 J
SMITH                JOHN                 A
SMITH                MIKE                 K
SMITH                MIKE                 R
SMITH                MIKE                 S
```

You want to create an index that compresses the values so as to compact entries into the blocks. When the index is accessed, the compression will result in fewer block reads and thus improve performance. Specifically you want to create a key-compressed index on the `LAST_NAME` and `FIRST_NAME` columns of this table.

Solution

Use the `COMPRESS N` clause to create a compressed index:

```
SQL> create index cust_cidx1 on cust(last_name, first_name) compress 2;
```

The prior line of code instructs Oracle to create a compressed index on two columns (`LAST_NAME` and `FIRST_NAME`). For this example, if we determined that there was a high degree of duplication only in the first column, we could instruct the `COMPRESS N` clause to compress only the first column (`LAST_NAME`) by specifying an integer of 1:

```
SQL> create index cust_cidx1 on cust(last_name, first_name) compress 1;
```

How It Works

Index compression is useful for indexes that contain multiple columns where the leading index column value is often repeated. Compressed indexes have the following advantages:

- Reduced storage
- More rows stored in leaf blocks, which can result in less I/O when accessing a compressed index

The degree of compression will vary by the amount of duplication in the index columns specified for compression. You can verify the degree of compression and the number of leaf blocks used by running the following two queries before and after creating an index with compression enabled:

```
SQL> select sum(bytes) from user_extents where segment_name='&&ind_name';
SQL> select index_name, leaf_blocks from user_indexes where index_name='&&ind_name';
```

You can verify the index compression is in use and the corresponding prefix length as follows:

```
select  index_name, compression, prefix_length
from user_indexes
where index_name = 'CUST_CIDX1';
```

Here's some sample output indicating that compression is enabled for the index with a prefix length of 2:

```
INDEX_NAME                     COMPRESS PREFIX_LENGTH
------------------------------ -------- -------------
CUST_CIDX1                     ENABLED              2
```

You can modify the prefix length by rebuilding the index. The following code changes the prefix length to 1:

```
SQL> alter index cust_cidx1 rebuild compress 1;
```

You can enable or disable compression for an existing index by rebuilding it. This example rebuilds the index with no compression:

```
SQL> alter index cust_cidx1 rebuild nocompress;
```

■ **Note** You cannot create a key-compressed index on a bitmap index.

2-8. Implementing a Function-Based Index

Problem

A query is running slow. You examine the `WHERE` clause and notice that a SQL `UPPER` function has been applied to a column. The `UPPER` function blocks the use of the existing index on that column. You want to create a function-based index to support the query. Here's an example of such a query:

```
SELECT first_name
FROM cust
WHERE UPPER(first_name) = 'DAVE';
```

You inspect USER_INDEXES and discover that an index exists on the FIRST_NAME column:

```
select  index_name, column_name
from user_ind_columns
where table_name = 'CUST';

INDEX_NAME           COLUMN_NAME
-------------------- --------------------
CUST_IDX1            FIRST_NAME
```

You generate an explain plan via SET AUTOTRACE TRACE EXPLAIN and notice that with the UPPER function applied to the column, the index is not used:

```
-------------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |     1 |    17 |     2   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| CUST |     1 |    17 |     2   (0)| 00:00:01 |
-------------------------------------------------------------------------
```

You need to create an index that Oracle will use in this situation.

Solution

There are two ways to resolve this issue:

- Create a function-based index.
- If using Oracle Database 11g or higher, create an indexed virtual column (see Recipe 2-9 for details).

This solution focuses on using a function-based index. You create a function-based index by referencing the SQL function and column in the index creation statement. For this example, a function-based index is created on UPPER(name):

```
SQL> create index cust_fidx1 on cust(UPPER(first_name));
```

To verify if the index is used, the Autotrace facility is turned on:

```
SQL> set autotrace trace explain;
```

Now the query is executed:

```
SELECT first_name
FROM cust
WHERE UPPER(first_name) = 'DAVE';
```

Here is the resulting execution plan showing that the function-based index is used:

```
-----------------------------------------------------------------------------------------
| Id  | Operation                    | Name       | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |            |     1 |    34 |     1   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| CUST       |     1 |    34 |     1   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN           | CUST_FIDX1 |     1 |       |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------
```

■ **Note** You can't modify a column that has a function-based index applied to it. You'll have to drop the index, modify the column, and then re-create the index.

How It Works

Function-based indexes are created with functions or expressions in their definitions. Function-based indexes allow index lookups on columns referenced by SQL functions in the `WHERE` clause of a query. The index can be as simple as the example in the "Solution" section of this recipe, or it can be based on complex logic stored in a PL/SQL function.

■ **Note** Any user-created SQL functions must be declared deterministic before they can be used in a function-based index. *Deterministic* means that for a given set of inputs, the function always returns the same results. You must use the keyword `DETERMINISTIC` when creating a user-defined function that you want to use in a function-based index.

If you want to see the definition of a function-based index, select from the `DBA/ALL/USER_IND_EXPRESSIONS` view to display the SQL associated with the index. If you're using SQL*Plus, be sure to issue a `SET LONG` command first—for example:

```
SQL> SET LONG 500
SQL> select index_name, column_expression from user_ind_expressions;
```

The `SET LONG` command in this example tells SQL*Plus to display up to 500 characters from the `COLUMN_EXPRESSION` column, which is of type `LONG`.

2-9. Indexing a Virtual Column

Problem

You're currently using a function-based index but need better performance. You want to replace the function-based index with a virtual column and place an index on the virtual column.

■ **Note** The virtual column feature requires Oracle Database 11g or higher.

Solution

Using a virtual column in combination with an index provides you with an alternative method for achieving performance gains when using SQL functions on columns in the `WHERE` clause. For example, suppose you have this query:

```
SELECT first_name
FROM cust
WHERE UPPER(first_name) = 'DAVE';
```

Normally, the optimizer will ignore any indexes on the column `FIRST_NAME` because of the SQL function applied to the column. There are two ways to improve performance in this situation:

- Create a function-based index (see Recipe 2-8 for details).
- Use a virtual column in combination with an index.

This solution focuses on the latter bullet. First a virtual column is added to the table that encapsulates the SQL function:

```
SQL> alter table cust add(up_name generated always as (UPPER(first_name)) virtual);
```

Next an index is created on the virtual column:

```
SQL> create index cust_vidx1 on cust(up_name);
```

This creates a very efficient mechanism to retrieve data when referencing a column with a SQL function.

How It Works

You might be asking this question: "Which performs better, a function-based index or an indexed virtual column?" In our testing, we were able to create several scenarios where the virtual column performed better than the function-based index. Results may vary depending on your data.

The purpose of this recipe is not to convince you to immediately start replacing all function-based indexes in your system with virtual columns; rather we want you to be aware of an alternative method for solving a common performance issue.

A virtual column is not free. If you have an existing table, you have to create and maintain the DDL required to create the virtual column, whereas a function-based index can be added, modified, and dropped independently from the table.

Several caveats are associated with virtual columns:

- You can define a virtual column only on a regular heap-organized table. You can't define a virtual column on an index-organized table, an external table, a temporary table, object tables, or cluster tables.
- Virtual columns can't reference other virtual columns.
- Virtual columns can reference columns only from the table in which the virtual column is defined.
- The output of a virtual column must be a scalar value (a single value, not a set of values).

To view the definition of a virtual column, use the `DBMS_METADATA` package to view the DDL associated with the table. If you're selecting from SQL*Plus, you need to set the `LONG` variable to a value large enough to show all data returned:

```
SQL> set long 10000;
SQL> select dbms_metadata.get_ddl('TABLE','CUST') from dual;
```

Here's a partial snippet of the output showing the virtual column details:

```
"UP_NAME" VARCHAR2(30) GENERATED ALWAYS AS (UPPER("FIRST_NAME"))
VIRTUAL VISIBLE) SEGMENT CREATION IMMEDIATE
```

You can also view the definition of the virtual column by querying the `DBA/ALL/USER_IND_EXPRESSIONS` view. If you're using SQL*Plus, be sure to issue a `SET LONG` command first—for example:

```
SQL> SET LONG 500
SQL> select index_name, column_expression from user_ind_expressions;
```

The `SET LONG` command in this example tells SQL*Plus to display up to 500 characters from the `COLUMN_EXPRESSION` column, which is of type `LONG`.

2-10. Avoiding Concentrated I/O for Index

Problem

You use a sequence to populate the primary key of a table and realize that this can cause contention on the leading edge of the index because the index values are nearly similar. This leads to multiple inserts into the same block, which causes contention. You want to spread out the inserts into the index so that the inserts more evenly distribute values across the index structure. You want to use a reverse-key index to accomplish this.

Solution

Use the REVERSE clause to create a reverse-key index:

```
SQL> create index inv_idx1 on inv(inv_id) reverse;
```

You can verify that an index is reverse-key by running the following query:

```
SQL> select index_name, index_type from user_indexes;
```

Here's some sample output showing that the INV_IDX1 index is reverse-key:

```
INDEX_NAME                     INDEX_TYPE
------------------------------ --------------------------
INV_IDX1                       NORMAL/REV
USERS_IDX1                     NORMAL
```

■ **Note** You can't specify REVERSE for a bitmap index or an index-organized table.

How It Works

Reverse-key indexes are similar to B-tree indexes except that the bytes of the index key are reversed when an index entry is created. For example, if the index values are 100, 101, and 102, the reverse-key index values are 001, 101, and 201:

```
Index value             Reverse key value
------------            --------------------
100                     001
101                     101
102                     201
```

Reverse-key indexes can perform better in scenarios where you need a way to evenly distribute index data that would otherwise have similar values clustered together. Thus, when using a reverse-key index, you avoid having I/O concentrated in one physical disk location within the index during large inserts of sequential values. The downside to this type of index is that it can't be used for index range scans, which therefore limits its usefulness.

You can rebuild an existing index to be reverse-key by using the REBUILD REVERSE clause—for example:

```
SQL> alter index f_regs_idx1 rebuild reverse;
```

Similarly, if you want to make an index that is reverse-key into a normally ordered index, then use the REBUILD NOREVERSE clause:

```
SQL> alter index f_regs_idx1 rebuild noreverse;
```

2-11. Adding an Index Without Impacting Existing Applications

Problem

You know from experience that sometimes when an index is added to a third-party application, this can cause performance issues and also can be a violation of the support agreement with the vendor. You want to implement an index in such a way that the application won't ever use the index.

Solution

Often, third-party vendors don't support customers adding their own indexes to an application. However, there may be a scenario in which you're certain you can increase a query's performance without impacting other queries in the application. You can create the index as invisible and then explicitly instruct a query to use the index via a hint—for example:

```
SQL> create index inv_idx1 on inv(inv_id) invisible;
```

Next, ensure that the `OPTIMIZER_USE_INVISIBLE_INDEXES` initialization parameter is set to `TRUE` (the default is `FALSE`). This instructs the optimizer to consider invisible indexes:

```
SQL> alter system set optimizer_use_invisible_indexes=true;
```

Now, use a hint to tell the optimizer that the index exists:

```
SQL> select /*+ index (inv INV_IDX1) */ inv_id from inv where inv_id=1;
```

You can verify that the index is being used by setting `AUTOTRACE TRACE EXPLAIN` and running the `SELECT` statement:

```
SQL> set autotrace trace explain;
SQL> select /*+ index (inv INV_IDX1) */ inv_id from inv where inv_id=1;
```

Here's some sample output indicating that the optimizer chose to use the invisible index:

```
------------------------------------------------------------------------------
| Id  | Operation        | Name     | Rows  | Bytes | Cost (%CPU)| Time     |
------------------------------------------------------------------------------
|   0 | SELECT STATEMENT |          |     1 |    13 |     1   (0)| 00:00:01 |
|*  1 |  INDEX RANGE SCAN| INV_IDX1 |     1 |    13 |     1   (0)| 00:00:01 |
------------------------------------------------------------------------------
```

Keep in mind that an invisible index means only that the optimizer can't see the index. Just like any other index, an invisible index consumes space and resources when executing DML statements.

How It Works

In Oracle Database 11g and higher, you have the option of making an index invisible to the optimizer. Oracle still maintains invisible indexes but doesn't make them available for use by the optimizer. If you

want the optimizer to use an invisible index, you can do so with a SQL hint. Invisible indexes have a couple of interesting uses:

- You can add an invisible index to a third-party application without affecting existing code or support agreements.
- Altering an index to invisible before dropping it allows you to quickly recover if you later determine that the index is required.

The first bulleted item was discussed in the "Solution" section of this recipe. The second scenario is discussed in this section. For example, suppose you've identified an index that isn't being used and are considering dropping it. In earlier releases of Oracle, you could mark the index as `UNUSABLE` and then later drop indexes that you were certain weren't being used. If you later determined that you needed an unusable index, the only way to re-enable the index was to rebuild it. For large indexes, this could take a long time and consume considerable database resources.

Making an index invisible has the advantage that it tells the optimizer only to not use the index. The invisible index is still maintained as the underlying table has records inserted, updated, or deleted. If you decide that you later need the index, there is no need to rebuild it. You simply have to mark it as visible again—for example:

```
SQL> alter index inv_idx1 visible;
```

You can verify the visibility of an index via this query:

```
SQL> select index_name, status, visibility from user_indexes;
```

Here's some sample output:

```
INDEX_NAME                     STATUS   VISIBILITY
------------------------------ -------- ----------
INV_IDX1                       VALID    VISIBLE
```

OLD SCHOOL: INSTRUCTING THE OPTIMIZER NOT TO USE AN INDEX

In the olden days, sometimes the rule-based optimizer (deprecated) would choose to use an index that would significantly decrease performance. In these situations, DBAs and developers would manually instruct the optimizer not to use an index on a numeric-based column as follows:

```
SQL> select cust_id from cust where cust_id+0 = 12345;
```

In the prior statement, the +0 adds nothing to the logic of the SQL statement (and therefore has no impact on the result set). In this scenario, the optimizer will automatically not use an index on a numeric column that has been modified with an arithmetic expression.

Similarly with character-based columns, indexes will be ignored for columns that have characters concatenated to them—for example:

```
SQL> select last_name from cust where last_name || '' = 'SMITH';
```

In the prior statement, the ||'' adds nothing to the logic of the SQL, but results in the optimizer not using an index on the `LAST_NAME` column (if one exists).

2-12. Creating a Bitmap Index in Support of a Star Schema

Problem

You have a data warehouse that contains a star schema. The star schema consists of a large fact table and several dimension (lookup) tables. The primary key columns of the dimension tables map to foreign key columns in the fact table. You would like to create bitmap indexes on all of the foreign key columns in the fact table.

Solution

You use the `BITMAP` keyword to create a bitmap index. The next line of code creates a bitmap index on the `CUST_ID` column of the `F_SALES` table:

```
SQL> create bitmap index f_sales_cust_fk1 on f_sales(cust_id);
```

The type of index is verified with the following query:

```
SQL> select index_name, index_type from user_indexes where index_name='F_SALES_CUST_FK1';

INDEX_NAME                     INDEX_TYPE
------------------------------ ---------------------------
F_SALES_CUST_FK1               BITMAP
```

How It Works

A bitmap index stores the `ROWID` of a row and a corresponding bitmap. You can think of the bitmap as a combination of ones and zeros. A one indicates the presence of a value, and a zero indicates that the value doesn't exist. Bitmap indexes are ideal for low-cardinality columns (few distinct values) and where the application is not frequently updating the table. Bitmap indexes are commonly used in data warehouse environments where you have star schema design.

A typical star schema structure consists of a large fact table and many small dimension (lookup) tables. In these scenarios, it's common to create bitmap indexes on fact table–foreign key columns. The fact tables are typically loaded on a daily basis and (usually) aren't subsequently updated or deleted.

You shouldn't use bitmap indexes on OLTP databases with high `INSERT/UPDATE/DELETE` activities, due to locking issues. Locking issues arise because the structure of the bitmap index results in potentially many rows being locked during DML operations, which results in locking problems for high-transaction OLTP systems.

■ **Note** Bitmap indexes and bitmap join indexes are available only with the Oracle Enterprise Edition of the database.

2-13. Creating a Bitmap Join Index

Problem

You're working in a data warehouse environment. You have a fairly large dimension table that is often joined to an extremely large fact table. You wonder if there's a way to create a bitmap index in such a way that it can eliminate the need for the optimizer to access the dimension table blocks to satisfy the results of a query.

Solution

Here's the basic syntax for creating a bitmap join index:

```
create bitmap index <index_name>
on <fact_table> (<dimension_table.dimension_column>)
from <fact_table>, <dimension_table>
where <fact_table>.<foreign_key_column> = <dimension_table>.<primary_key_column>;
```

Bitmap join indexes are appropriate in situations where you're joining two tables using the foreign key column(s) in one table that relate to primary key column(s) in another table. For example, suppose you typically retrieve the `CUST_NAME` from the `D_CUSTOMERS` table while joining to a large `F_SHIPMENTS` fact table. This example creates a bitmap join index between the `F_SHIPMENTS` and `D_CUSTOMERS` tables:

```
create bitmap index f_shipments_bm_idx1
on f_shipments(d_customers.cust_name)
from f_shipments, d_customers
where f_shipments.d_cust_id = d_customers.d_cust_id;
```

Now, consider a query such as this:

```
select
  d.cust_name
from f_shipments f, d_customers d
where f.d_cust_id = d.d_cust_id
and d.cust_name = 'Sun';
```

The optimizer can choose to use the bitmap join index and thus avoid the expense of having to join the tables.

How It Works

Bitmap join indexes store the results of a join between two tables in an index. Bitmap indexes are beneficial because they avoid joining tables to retrieve results. The syntax for a bitmap join index differs from a regular bitmap index in that it contains `FROM` and `WHERE` clauses.

Bitmap join indexes are usually suitable only for data warehouse environments where you have tables that get loaded and then are not updated. When updating tables that have bitmap join indexes declared, this potentially results in several rows being locked. Therefore this type of an index is not suitable for an OLTP database.

2-14. Creating an Index-Organized Table

Problem

You want to create a table that is the intersection of a many-to-many relationship between two tables. The intersection table will consist of two columns. Each column is a foreign key that maps back to a corresponding primary key in a parent table.

Solution

Index-organized tables (IOTs) are efficient objects when the table data is typically accessed through querying on the primary key. Use the `ORGANIZATION INDEX` clause to create an IOT:

```
create table cust_assoc
(cust_id number
,user_group_id number
,create_dtt timestamp(5)
,update_dtt timestamp(5)
,constraint cust_assoc_pk primary key(cust_id, user_group_id)
)
organization index
including create_dtt
pctthreshold 30
tablespace nsestar_index
overflow
tablespace dim_index;
```

Notice that `DBA/ALL/USER_TABLES` includes an entry for the table name used when creating an IOT. The following two queries show how Oracle records the information regarding the IOT in the data dictionary:

```
select table_name, iot_name
from user_tables
where iot_name = 'CUST_ASSOC';
```

Here is some sample output:

```
TABLE_NAME                     IOT_NAME
------------------------------ ------------------------------
SYS_IOT_OVER_184185            CUST_ASSOC
```

Listed next is another slightly different query with its output:

```
select table_name, iot_name
from user_tables
where table_name = 'CUST_ASSOC';
```

Here is some sample output:

```
TABLE_NAME                     IOT_NAME
------------------------------ ------------------------------
CUST_ASSOC
```

Additionally, DBA/ALL/USER_INDEXES contains a record with the name of the primary key constraint specified. The INDEX_TYPE column contains a value of IOT - TOP for IOTs:

```
select index_name, index_type
from user_indexes
where table_name = 'CUST_ASSOC';
```

Here is some sample output:

```
INDEX_NAME                     INDEX_TYPE
------------------------------ ---------------------------
CUST_ASSOC_PK                  IOT - TOP
```

How It Works

An IOT stores the entire contents of the table's row in a B-tree index structure. IOTs provide fast access for queries that have exact matches and/or range searches on the primary key.

All columns specified up to and including the column specified in the INCLUDING clause are stored in the same block as the CUST_ASSOC_PK primary key column. In other words, the INCLUDING clause specifies the last column to keep in the table segment. Columns listed after the column specified in the INCLUDING clause are stored in the overflow data segment. In the previous example, the UPDATE_DTT column is stored in the overflow segment.

PCTTHRESHOLD specifies the percentage of space reserved in the index block for the IOT row. This value can be from 1 to 50, and defaults to 50 if no value is specified. There must be enough space in the index block to store the primary key. The OVERFLOW clause details which tablespace should be used to store overflow data segments.

2-15. Monitoring Index Usage

Problem

You maintain a large database that contains thousands of indexes. As part of your proactive maintenance, you want to determine if any indexes are not being used. You realize that unused indexes have a detrimental impact on performance, because every time a row is inserted, updated, and deleted, the corresponding index has to be maintained. This consumes CPU resources and disk space. If an index isn't being used, it should be dropped.

Solution

Use the ALTER INDEX...MONITORING USAGE statement to enable basic index monitoring. The following example enables index monitoring on an index named F_REGS_IDX1:

```
SQL> alter index f_regs_idx1 monitoring usage;
```

The first time the index is accessed, Oracle records this; you can view whether an index has been accessed via the V$OBJECT_USAGE view. To report which indexes are being monitored and have ever been used, run this query:

```
SQL> select index_name, table_name, monitoring, used from v$object_usage;
```

If the index has ever been used in a SELECT statement, then the USED column will contain the YES value. Here is some sample output from the prior query:

```
INDEX_NAME                     TABLE_NAME                     MON USED
------------------------------ ------------------------------ --- ----
F_REGS_IDX1                    F_REGS                         YES YES
```

Most likely, you won't monitor only one index. Rather, you'll want to monitor all indexes for a user. In this situation, use SQL to generate SQL to create a script you can run to turn on monitoring for all indexes. Here's such a script:

```
set pagesize 0 head off linesize 132
spool enable_mon.sql
select
  'alter index ' || index_name || ' monitoring usage;'
from user_indexes;
spool off;
```

To disable monitoring on an index, use the NOMONITORING USAGE clause—for example:

```
SQL> alter index f_regs_idx1 nomonitoring usage;
```

How It Works

The main advantage to monitoring index usage is to identify indexes not being used. This allows you to identify indexes that can be dropped. This will free up disk space and improve the performance of DML statements.

The V$OBJECT_USAGE view shows information only for the currently connected user. You can verify this behavior by inspecting the TEXT column of DBA_VIEWS for the V$OBJECT_USAGE definition:

```
SQL> select text from dba_views where view_name = 'V$OBJECT_USAGE';
```

Notice the following line in the output:

```
where io.owner# = userenv('SCHEMAID')
```

That line instructs the view to display information only for the currently connected user. If you're logged in as a DBA privileged user and want to view the status of all indexes that have monitoring enabled (regardless of the user), execute this query:

```
select io.name, t.name,
       decode(bitand(i.flags, 65536), 0, 'NO', 'YES'),
       decode(bitand(ou.flags, 1), 0, 'NO', 'YES'),
       ou.start_monitoring,
       ou.end_monitoring
from sys.obj$ io
     ,sys.obj$ t
     ,sys.ind$ i
     ,sys.object_usage ou
where i.obj# = ou.obj#
and io.obj# = ou.obj#
and t.obj# = i.bo#;
```

The prior query removes the line from the query that restricts output to display information only for the currently logged-in user. This provides you with a convenient way to view all monitored indexes.

2-16. Maximizing Index Creation Speed

Problem

You're creating an index based on a table that contains millions of rows. You want to create the index as fast as possible.

Solution

This solution describes two techniques for increasing the speed of index creation:

- Turning off redo generation
- Increasing the degree of parallelism

You can use the prior two features independently of each other, or they can be used in conjunction.

Turning Off Redo Generation

You can optionally create an index with the `NOLOGGING` clause. Doing so has these implications:

- The redo isn't generated that would be required to recover the index in the event of a media failure.
- Subsequent direct-path operations also won't generate the redo required to recover the index information in the event of a media failure.

Here's an example of creating an index with the `NOLOGGING` clause:

```
create index inv_idx1 on inv(inv_id, inv_id2)
nologging
tablespace inv_mgmt_index;
```

You can run this query to determine whether an index has been created with `NOLOGGING`:

```
SQL> select index_name, logging from user_indexes;
```

Increasing the Degree of Parallelism

In large database environments where you're attempting to create an index on a table that is populated with many rows, you may be able to reduce the time it takes to create the index by using the `PARALLEL` clause. For example, this sets the degree of parallelism to 2 when creating the index:

```
create index inv_idx1 on inv(inv_id)
parallel 2
tablespace inv_mgmt_data;
```

You can verify the degree of parallelism on an index via this query:

```
SQL> select index_name, degreel from user_indexes;
```

■ **Note** If you don't specify a degree of parallelism, Oracle selects a degree based on the number of CPUs on the box times the value of `PARALLEL_THREADS_PER_CPU`.

How It Works

The main advantage of `NOLOGGING` is that when you create the index, a minimal amount of redo information is generated, which can have significant performance implications when creating a large index. The disadvantage is that if you experience a media failure soon after the index is created (or have records inserted via a direct-path operation), and subsequently have a failure that causes you to restore from a backup (taken prior to the index creation), then you may see this error when the index is accessed:

```
ORA-01578: ORACLE data block corrupted (file # 4, block # 11407)
ORA-01110: data file 4: '/ora01/dbfile/O11R2/inv_mgmt_index01.dbf'
ORA-26040: Data block was loaded using the NOLOGGING option
```

This error indicates that the index is logically corrupt. In this scenario, you must re-create or rebuild the index before it's usable. In most scenarios, it's acceptable to use the `NOLOGGING` clause when creating an index, because the index can be re-created or rebuilt without affecting the table on which the index is based.

In addition to `NOLOGGING`, you can use the `PARALLEL` clause to increase the speed of an index creation. For large indexes, this can significantly decrease the time required to create an index.

Keep in mind that you can use `NOLOGGING` in combination with `PARALLEL`. This next example rebuilds an index in parallel while generating a minimal amount of redo:

```
SQL> alter index inv_idx1 rebuild parallel nologging;
```

2-17. Reclaiming Unused Index Space

Problem

You have an index consuming space in a segment, but without actually using that space. For example, you're running the following query to display the Segment Advisor's advice:

```
SELECT
 'Task Name        : ' || f.task_name  || CHR(10) ||
 'Start Run Time   : ' || TO_CHAR(execution_start, 'dd-mon-yy hh24:mi') || chr (10) ||
 'Segment Name     : ' || o.attr2      || CHR(10) ||
 'Segment Type     : ' || o.type       || CHR(10) ||
 'Partition Name   : ' || o.attr3      || CHR(10) ||
 'Message          : ' || f.message    || CHR(10) ||
 'More Info        : ' || f.more_info  || CHR(10) ||
```

```
 '----------------------------------------------------' Advice
FROM dba_advisor_findings   f
    ,dba_advisor_objects    o
    ,dba_advisor_executions e
WHERE o.task_id   = f.task_id
AND   o.object_id = f.object_id
AND   f.task_id   = e.task_id
AND   e. execution_start > sysdate - 1
AND   e.advisor_name = 'Segment Advisor'
ORDER BY f.task_name;
```

The following output is displayed:

```
ADVICE
--------------------------------------------------------------------------------
Task Name        : F_REGS Advice
Start Run Time   : 19-feb-11 09:32
Segment Name     : F_REGS_IDX1
Segment Type     : INDEX
Partition Name   :
Message          : Perform shrink, estimated savings is 84392870 bytes.
More Info        : Allocated Space:166723584: Used Space:82330714: Reclaimable S
pace :84392870:
--------------------------------------------------------
```

You want to shrink the index to free up the unused space.

Solution

There are a couple of effective methods for freeing up unused space associated with an index:

- Rebuilding the index
- Shrinking the index

Before you perform either of these operations, first check `USER_SEGMENTS` to verify that the amount of space used corresponds with the Segment Advisor's advice. In this example, the segment name is `F_REGS_IDX1`:

```
SQL> select bytes from user_segments where segment_name = 'F_REGS_IDX1';

BYTES
----------
 166723584
```

This example uses the `ALTER INDEX...REBUILD` statement to re-organize and compact the space used by an index:

```
SQL> alter index f_regs_idx1 rebuild;
```

Alternatively, use the `ALTER INDEX...SHRINK SPACE` statement to free up unused space in an index—for example:

```
SQL> alter index f_regs_idx1 shrink space;

Index altered.
```

Now query `USER_SEGMENTS` again to verify that the space has been de-allocated. Here is the output for this example:

```
  BYTES
----------
    524288
```

The space consumed by the index has considerably decreased.

How It Works

Usually rebuilding an index is the fastest and most effective way to reclaim unused space consumed by an index. Therefore this is the approach we recommend for reclaiming unused index space. Freeing up space is desirable because it ensures that you use only the amount of space required by an object. It also has the performance benefit that Oracle has fewer blocks to manage and sort through when performing read operations.

Besides freeing up space, you may want to consider rebuilding an index for these additional reasons:

- The index has become corrupt.
- You want to modify storage characteristics (such as changing the tablespace).
- An index that was previously marked as unusable now needs to be rebuilt to make it usable again.

Keep in mind that Oracle attempts to acquire a lock on the table and rebuild the index online. If there are any active transactions that haven't committed, then Oracle won't be able to obtain a lock, and the following error will be thrown:

```
ORA-00054: resource busy and acquire with NOWAIT specified or timeout expired
```

In this scenario, you can either wait until the there is little activity in the database or try setting the `DDL_LOCK_TIMEOUT` parameter:

```
SQL> alter session set ddl_lock_timeout=15;
```

The `DDL_LOCK_TIMEOUT` initialization parameter is available in Oracle Database 11g or higher. It instructs Oracle to repeatedly attempt to obtain a lock for the specified amount of time.

If no tablespace is specified, Oracle rebuilds the index in the tablespace in which the index currently exists. Specify a tablespace if you want the index rebuilt in a different tablespace:

```
SQL> alter index inv_idx1 rebuild tablespace inv_index;
```

■ **Tip** If you're working with a large index, you may want to consider using features such as `NOLOGGING` and/or `PARALLEL` (see Recipe 2-16 for details).

If you use the `ALTER INDEX...SHRINK SPACE` operation to free up unused index space, keep in mind that this feature requires that the target object must be created within a tablespace with automatic segment space management enabled. If you attempt to shrink a table or index that has been created in a tablespace using manual segment space management, you'll receive this error:

```
ORA-10635: Invalid segment or tablespace type
```

As we've noted elsewhere in this chapter, we recommend that you use the ASSM feature whenever possible. This allows you to take advantage of all the Oracle segment management features.

The `ALTER INDEX...SHRINK SPACE` statement has a few nuances to be aware of. For example, you can instruct Oracle to attempt only to merge the contents of index blocks (and not free up space) via the `COMPACT` clause:

```
SQL> alter index f_regs_idx1 shrink space compact;
```

The prior operation is equivalent to the `ALTER INDEX...COALESCE` statement. Here's an example of using `COALESCE`:

```
SQL> alter index f_regs_idx1 coalesce;
```

If you want to maximize the space compacted, either rebuild the index or use the `SHRINK SPACE` clause as shown in the "Solution" section of this recipe. It's somewhat counterintuitive that the `COMPACT` space doesn't actually initiate a greater degree of realized free space. The `COMPACT` clause instructs Oracle to only merge index blocks where possible and not to maximize the amount of space being freed up.

CHAPTER 3

Optimizing Instance Memory

Optimizing the memory you allocate to an Oracle database is one of the most critical tasks you need to perform as a DBA. Over the years, Oracle DBAs were used to spending vast amounts of their time analyzing memory usage by the database and trying to come up with the best possible allocation of memory. In Oracle Database 11g, the burden of allocating Oracle's memory is shifted almost completely to the database itself. This chapter shows you how to take advantage of Oracle's automatic memory management feature, so you can leave the database to optimize memory usage, while you focus on more important matters.

This chapter starts off by explaining how to set up automatic memory management for a database. The chapter also shows you how to set minimum values for certain components of memory even under automatic memory management. It also includes recipes that explain how to create multiple buffer pools, how to monitor Oracle's usage of memory, and how to use the Oracle Enterprise Manager's Database Control (or Grid Control) tool to get advice from Oracle regarding the optimal sizing of memory allocation. You'll also learn how to optimize the use of the Program Global Area (PGA), a key Oracle memory component, especially in data warehouse environments.

In Oracle Database 11g, Oracle has introduced an exciting new result caching feature. Oracle can now cache the results of both SQL queries and PL/SQL functions in the shared pool component of Oracle's memory. We discuss that server result cache in this chapter. In addition, you'll also find a recipe that explains how to take advantage of Oracle's client-side result caching feature. Finally, we show how to use the exciting new Oracle feature called the Oracle Database Smart Flash Cache.

3-1. Automating Memory Management

Problem

You want to automate memory management in your Oracle database. You have both OLTP and batch jobs running in this database. You want to take advantage of the automatic memory management feature built into Oracle Database 11g.

Solution

Here are the steps to implement automatic memory management in your database, if you've already set either the `SGA_TARGET` or the `PGA_AGGREGATE_TARGET` parameters (or both). We assume that we are going to allocate 2,000 MB to the `MEMORY_MAX_TARGET` parameter and 1,000 MB to the `MEMORY_TARGET` parameter.

1. Connect to the database with the SYSDBA privilege.
2. Assuming you're using the SPFILE, first set a value for the MEMORY_MAX_TARGET parameter:

```
SQL> alter system set memory_max_target=2G scope=spfile;
System altered.
```

You must specify the SCOPE parameter in the alter system command, because MEMORY_MAX_TARGET isn't a dynamic parameter, which means you can't change it on the fly while the instance is running.

3. Note that if you've started the instance with a traditional init.ora parameter file instead of the SPFILE, you must add the following to your init.ora file:

```
memory_max_target = 2000M
memory_target = 1000M
```

4. Bounce the database.
5. Turn off the SGA_TARGET and the PGA_AGGREGATE_TARGET parameters by issuing the following ALTER SYSTEM commands:

```
SQL> alter system set sga_target = 0;
SQL> alter system set pga_aggregate_target = 0;
```

6. Turn on automatic memory management by setting the MEMORY_TARGET parameter:

```
SQL> alter system set memory_target = 1000M;
```

From this point on, the database runs under the automatic memory management mode, with it shrinking and growing the individual allocations to the various components of Oracle memory according to the requirements of the ongoing workload. You can change the value of the MEMORY_TARGET parameter dynamically anytime, as long as you don't exceed the value you set for the MEMORY_MAX_TARGET parameter.

■ **Tip** The term "target" in parameters such as memory_target and pga_memory_target means just that—Oracle will try to stay under the target level, but there's no guarantee that it'll never go beyond that. It may exceed the target allocation on occasion, if necessary.

You don't have to set the SGA_TARGET and PGA_AGGREGATE_TARGET parameters to 0 in order to use automatic memory management. In Recipe 3-3, we show how to set minimum values for these parameters even when you choose to implement automatic memory management. That recipe assumes you're implementing automatic memory management, but that for some reason, you need to specify your own minimum values for components such as the SGA and the PGA.

How It Works

In earlier releases of the Oracle database, DBAs used to set values for the various SGA components, or would specify values for the SGA and the PGA. Starting with the Oracle Database 11g release, Oracle enables you to completely automate the entire instance memory allocation, by just setting a single initialization parameter, `MEMORY_TARGET`, under what's known as automatic memory management. In this Recipe, we show you how to set up the automatic memory management feature in your database.

If you're creating a new Oracle database with the help of the Database Configuration Assistant (DBCA), you're given a choice among automatic memory management, shared memory management, and manual memory management. Select the automatic memory management option, and specify the values for two automatic memory-related parameters: `MEMORY_TARGET` and `MEMORY_MAX_TARGET`. The first parameter sets the current value of the memory allocation to the database, and the second parameter sets the limit to which you can raise the first parameter if necessary.

Oracle's memory structures consist of two distinct memory areas. The system global area (SGA) contains the data and control information and is shared by all server and background processes. The SGA holds the data blocks retrieved from disk by Oracle. The program global area (PGA) contains data and control information for a server process. Each server process is allocated its own chunk of the PGA. Managing Oracle's memory allocation involves careful calibration of the needs of the database. Some database instances need more memory for certain components of the memory. For example, a data warehouse will need more PGA memory in order to perform huge sorts that are common in such an environment. Also, during the course of a day, the memory needs of the instance might vary; during business hours, for example, the instance might be processing more online transaction processing (OLTP) work, whereas after business hours, it might be running huge batch jobs that involve data warehouse processing, jobs that typically need higher PGA allocations per each process.

In prior versions of the Oracle database, DBAs had to carefully decide the optimal allocation of memory to the individual components of the memory one allocated to the database. Technically, you can still manually set the values of the individual components of the SGA as well as set a value for the PGA, or partially automate the process by setting parameters such as `SGA_TARGET` and `PGA_AGGREGATE_TARGET`. Although Oracle still allows you to manually configure the various components of memory, automatic memory management is the recommended approach to managing Oracle's memory allocation. Once you specify a certain amount of memory by setting the `MEMORY_TARGET` and `MEMORY_MAX_TARGET` parameters, Oracle automatically tunes the actual memory allocation, by redistributing memory between the SGA and the PGA.

■ **Tip** When you create a database with the Database Configuration Assistant (DBCA), automatic memory management is the default.

Oracle Database 11g lets you automate all the memory allocations for an instance, including shared memory and the PGA memory, if you choose to implement automatic memory management by setting the `MEMORY_TARGET` and `MEMORY_MAX_TARGET` parameters. Under an automatic memory management regime, Oracle automatically tunes the total SGA size, the SGA component sizes, the instance PGA size, and the individual PGA size. This dynamic memory tuning by the Oracle instance optimizes database performance, as memory allocations are changed automatically by Oracle to match changing database workloads. Automatic memory management means that once you set the `MEMORY_TARGET` parameter, you can simply ignore the following parameters by not setting them at all:

- SGA_TARGET
- PGA_AGGREGATE_TARGET
- DB_CACHE_SIZE
- SHARED_POOL_SIZE
- LARGE_POOL_SIZE
- JAVA_POOL_SIZE

If you're moving from a system where you were using the SGA_TARGET and PGA_AGGREGATE_TARGET parameters, you can follow the procedures shown in the "Solution" section of this recipe to move to the newer automatic memory management mode of managing Oracle's memory allocation. Note that while setting the MEMORY_TARGET parameter is mandatory for implementing automatic memory management, the MEMORY_MAX_TARGET parameter isn't—if you don't set this parameter, Oracle sets its value internally to that of the MEMORY_TARGET parameter. Also, the MEMORY_MAX_TARGET parameter acts as the upper bound for the MEMORY_TARGET parameter. Oracle has different minimum permissible settings for the MEMORY_TARGET parameter, depending on the operating system. If you try to set this parameter below its minimum allowable value, the database will issue an error. Some of the memory components can't shrink quickly and some components must have a minimum size for the database to function properly. Therefore, Oracle won't let you set too low a value for the MEMORY_TARGET parameter. The following example shows this:

```
SQL> alter system set memory_target=360m scope=both;
alter system set memory_target=360m scope=both
*
ERROR at line 1:
ORA-02097: parameter cannot be modified because specified value is invalid
ORA-00838: Specified value of MEMORY_TARGET is too small, needs to be at least
544M

SQL> alter system set memory_target=544m scope=both;

alter system set memory_target=544m scope=both
*
ERROR at line 1:
ORA-02097: parameter cannot be modified because specified value is invalid
ORA-00838: Specified value of MEMORY_TARGET is too small, needs to be at least
624M

SQL>  alter system set memory_target=624m scope=both;

System altered.

SQL>
```

You'll notice that Oracle issued an error when we tried to set a very low value for the MEMORY_TARGET parameter. Note that Oracle took iterations to decide to let you know the minimum allowable level for the MEMORY_TARGET parameter.

How does one go about setting the value of the MEMORY_MAX_TARGET parameter? It's simple—you just pick a value that's high enough to accommodate not only the current workloads, but also the future needs of the database. Since the MEMORY_TARGET parameter is dynamic, you can alter it on the fly and if necessary, re-allocate memory among multiple instances running on a server. Just be sure that you set the value of the MEMORY_MAX_TARGET parameter to a size that's at least equal to the combined value of the present settings of the SGA_TARGET and the PGA_AGGREGATE_TARGET parameters (if you're migrating from the 10g release). Always make sure to check with your system administrator, so you don't allocate too high an amount of memory for your Oracle instance, which could result in problems such as paging and swapping at the operating system level, which will affect not only your Oracle database, but also everything else that's running on that server.

3-2. Managing Multiple Buffer Pools

Problem

You're using automatic memory management, but have decided to allocate a minimum value for the buffer pool component. You'd like to configure the buffer pool so it retains frequently accessed segments, which may run the risk of being aged out of the buffer pool.

Solution

You can use multiple buffer pools instead of Oracle's single default buffer pool, to ensure that frequently used segments stay cached in the buffer pool without being recycled out of the buffer pool. In order to implement multiple buffer pools in your database, you need to do two things: create two separate buffer pools—the KEEP buffer pool and the RECYCLE buffer pool. Once you do this, you must specify the BUFFER_POOL keyword in the STORAGE clause when you create a segment. For example, if you want a segment to be cached (or pinned) in the buffer pool, you must specify the KEEP buffer pool.

■ **Note** Neither the KEEP nor the RECYCLE pool is part of the default BUFFER CACHE. Both of these pools are outside the default buffer cache.

Here's how you create the two types of buffer pools.

In the SPFILE or the init.ora file, specify the two parameters and the sizes you want to assign to each of the pools:

```
db_keep_cache_size=1000m
db_recycle_cache_size=100m
```

Here's how you specify the default buffer pool for a segment:

```
SQL> alter table employees
     storage (buffer_pool=keep);

Table altered.
SQL>
```

How It Works

Configuring a KEEP buffer pool is an ideal solution in situations where your database contains tables that are referenced numerous times. You can store such frequently accessed segments in the KEEP buffer cache. By doing this, you not only isolate those segments to a specific part of the buffer pool, but also ensure that your physical I/O is minimized as well. How large the KEEP buffer pool ought to be depends on the total size of the objects you want to assign to the pool. You can get a rough idea by summing up the size of all candidate objects for this pool, or you can check the value of the DBA_TABLES view (BLOCKS column) to figure this out.

While we're on this topic, we'd like to point out the counterpart to the KEEP buffer pool—the RECYCLE buffer pool. Normally, the Oracle database uses a least recently used algorithm to decide which objects it should jettison from the buffer cache, when it needs more free space. If your database accesses very large objects once or so every day, you can keep these objects from occupying a big chunk of your buffer cache, and instead make those objects age right out of the buffer cache after they've been accessed. You can configure such behavior by allowing candidate objects to use the RECYCLED buffer pool either when you create those objects, or even later on, by setting the appropriate storage parameters, as shown in the following examples (note that you must first set the DB_RECYCLE_CACHE_SIZE initialization parameter, as shown in the "Solution" section of this recipe.

You can execute the following query to figure out how many blocks for each segment are currently in the buffer cache:

```
SQL> select o.object_name, count(*) number_of_blocks
     from dba_objects o, v$bh v
     where o.data_object_id = v.objd
     and o.owner !='SYS'
     group by o.object_name
     order by count(*);
```

When your database accesses large segments and retrieves data to process a query, it may sometimes age out other segments from the buffer pool prematurely. If you need these prematurely aged-out segments later on, it requires additional I/O. What exactly constitutes a large table or index segment is subject to your interpretation. It's probably safe to think of the size of the object you're concerned with by considering its size relative to the total memory you have allocated to the buffer cache. Oracle figures that if a single segment takes up more than 10% of (non-sequential) physical reads, it's a large segment, for the purpose of deciding if it needs to use the KEEP or RECYCLE buffer pools. So, a handful of such large segments can occupy a significant proportion of the buffer cache and hurt the performance of the database.

If you have other segments that the database accesses, let's say, every other second, they won't age out of the buffer pool since they are constantly in use. However, there may be other segments that will be adversely affected by the few large segments the database has read into its buffer cache. It's in such situations that your database can benefit most by devoting the RECYCLE pool for the large segments. Of course, if you want to absolutely, positively ensure that key segments never age out at all, then you can create the KEEP buffer cache and assign these objects to this pool.

3-3. Setting Minimum Values for Memory

Problem

You're using automatic memory management, but you think that the database sometimes doesn't allocate enough memory for the PGA_AGGREGATE_TARGET component.

Solution

Although automatic memory management is supposed to do what it says—automate memory allocation—there are times when you realize that Oracle isn't allocating certain memory components optimally. You can set a minimum value for any of the main Oracle memory components—buffer cache, shared pool, large pool, Java pool, and the PGA memory. For example, even after specifying automatic memory management, you can specify a target for the instance PGA with the following command, without having to restart the database:

```
SQL> alter system set pga_aggregate_target=1000m;
```

Oracle will, from this point forward, never decrease the PGA memory allocation to less than the value you've set—this value implicitly sets a minimum value for the memory parameter. The database will continue to automatically allocate memory to the various components of the SGA, but first it subtracts the memory you've allocated explicitly to the PGA—in this case, 1,000 MB, from the MEMORY_TARGET parameter's value. What remains is what the database will allocate to the instance's SGA.

How It Works

Ever since Oracle introduced the SGA_TARGET (to automate shared memory management) in Oracle Database 10g and the MEMORY_TARGET parameter (to automate shared memory and PGA memory management) in Oracle Database 11g, some DBAs have complained that these parameters sometimes weren't appropriately sizing some of the components of Oracle memory, such as the buffer cache.

There's some evidence that under automatic memory management, the database could lag behind an event that requires a sudden increase in the allocation to either one of the individual components of the SGA or to the PGA. For example, you may have a spurt of activity in the database that requires a quick adjustment to the shared pool component of memory—the database may get to the optimal shared pool size allocation level only after it notices the events that require the higher memory. As a result, the database may undergo a temporary performance hit. Several DBAs have, as a result, found that automatic memory management will work fine, as long as you set a minimum value for, say, the buffer cache or the PGA or both, by specifying explicit values for the SGA_TARGET and the PGA_AGGREGATE_TARGET initialization parameters, instead of leaving them at their default value of zero. The database will still use automatic memory management, but will now use the specific values you set for any of the memory components as minimum values. Having said this, in our experience, automatic memory management works as advertised most of the time; however, your mileage may vary, depending on any special time-based workload changes in a specific database. At times like this, it's perfectly all right to set minimum values that represent your own understanding of your processing requirements, instead of blindly depending on Oracle's automatic memory algorithms.

3-4. Monitoring Memory Resizing Operations

Problem

You've implemented automatic memory management in your database and would like to monitor how the database is currently allocating the various dynamically tuned memory components.

Solution

Under an automatic memory management mode, you can view the current allocations of memory in any instance by querying the V$MEMORY_DYNAMIC_COMPONENTS view. Querying this view provides vital information to help you tune the MEMORY_TARGET parameter. Here's how you execute a query against this view:

```
SQL> select * from v$memory_target_advice order by memory_size;

MEMORY_SIZE MEMORY_SIZE_FACTOR ESTD_DB_TIME ESTD_DB_TIME_FACTOR    VERSION
----------- ------------------ ------------ -------------------- ----------
        468                .75        43598               1.0061          0
        624                  1        43334                    1          0
        780               1.25        43334                    1          0
        936                1.5        43330                .9999          0
       1092               1.75        43330                .9999          0
       1248                  2        43330                .9999          0

6 rows selected.

SQL>
```

Your current memory allocation is shown by the row with the MEMORY_SIZE_FACTOR value of 1 (624 MB in our case). The MEMORY_SIZE_FACTOR column shows alternate sizes of the MEMORY_TARGET parameter as a multiple of the current MEMORY_TARGET parameter value. The ESTD_DB_TIME column shows the time Oracle estimates it will need to complete the current workload with a specific MEMORY_TARGET value. Thus, the query results show you how much faster the database can process its work by varying the value of the MEMORY_TARGET parameter.

How It Works

Use the V$MEMORY_TARGET_ADVICE view to get a quick idea about how optimal your MEMORY_TARGET allocation is. You need to run a query based on this view after a representative workload has been processed by the database, to get useful results. If the view reports that there are no gains to be had by increasing the MEMORY_TARGET setting, you don't have to throw away precious system memory by allocating more memory to the database instance. Oftentimes, the query may report that potential performance, as indicated by the ESTD_DB_TIME column of the V$MEMORY_TARGET_ADVICE view, doesn't decrease at a MEMORY_SIZE_FACTOR value that's less than 1. You can safely reduce the setting of the MEMORY_TARGET parameter in such cases.

You can also use the V$MEMORY_RESIZE_OPS view to view how the instance resized various memory components over a past interval of 800 completed memory resizing operations. You'll see that the database automatically increases or shrinks the values of the SGA_TARGET and PGA_AGGREGATE_TARGET parameters based on the workload it encounters. The following query shows how to use the V$MEMORY_RESIZE_OPS view to understand Oracle's dynamic allocation of instance memory:

```
SQL> select component,oper_type,oper_mode,parameter, final_size,target_size
     from v$memory_resize_ops

COMPONENT               OPER_TYPE   OPER_MODE      PARAMETER      FINAL_SIZE   TARGET_SIZE
---------------------  -----------  ---------   ---------------  ------------  ------------

DEFAULT buffer cache    GROW        DEFERRED   db_cache_size        180355072     180355072
shared pool             GROW        DEFERRED   shared_pool_size     264241152     264241152
…
20 rows selected.
SQL>
```

The OPER_TYPE column can take two values - GROW or SHRINK, depending on whether the database grows or shrinks individual memory components as the database workload fluctuates over time. It's this ability to respond to these changes by automatically provisioning the necessary memory to the various memory components that makes this "automatic" memory management. The DBA will do well by monitoring this view over time, to ensure that automatic memory management works well for his or her databases.

3-5. Optimizing Memory Usage

Problem

You've set up automatic memory management in your databases and would like to optimize memory usage with the help of Oracle's memory advisors.

Solution

Regardless of whether you set up automatic memory management (AMM) or automatic shared memory management (ASMM), or even a manual memory management scheme, you can use Oracle's Memory Advisors to guide your memory tuning efforts. In this example, we show how to use Oracle Enterprise Manager Database Control to easily tune memory usage. Here are the steps:

1. Go to the Database Home page in Database Control. Click Advisor Central at the bottom of the page.
2. In the Memory Advisors page that appears, click Advice next to the Total Memory Size box under Automatic Memory Management.

The Memory Size Advice graph appears in a separate window, as shown in Figure 3-1. In this graph, the improvement in DB time is plotted against the total memory that you've currently set for the MEMORY_TARGET parameter. The higher the value of improvement in DB time, the better off will be the performance. The graph shows how performance (improvement in DB time) will vary as you change the

MEMORY_TARGET parameter value. The total memory you allocate can't be more than the maximum allowed memory for this instance, which is indicated by the dotted straight line in the graph.

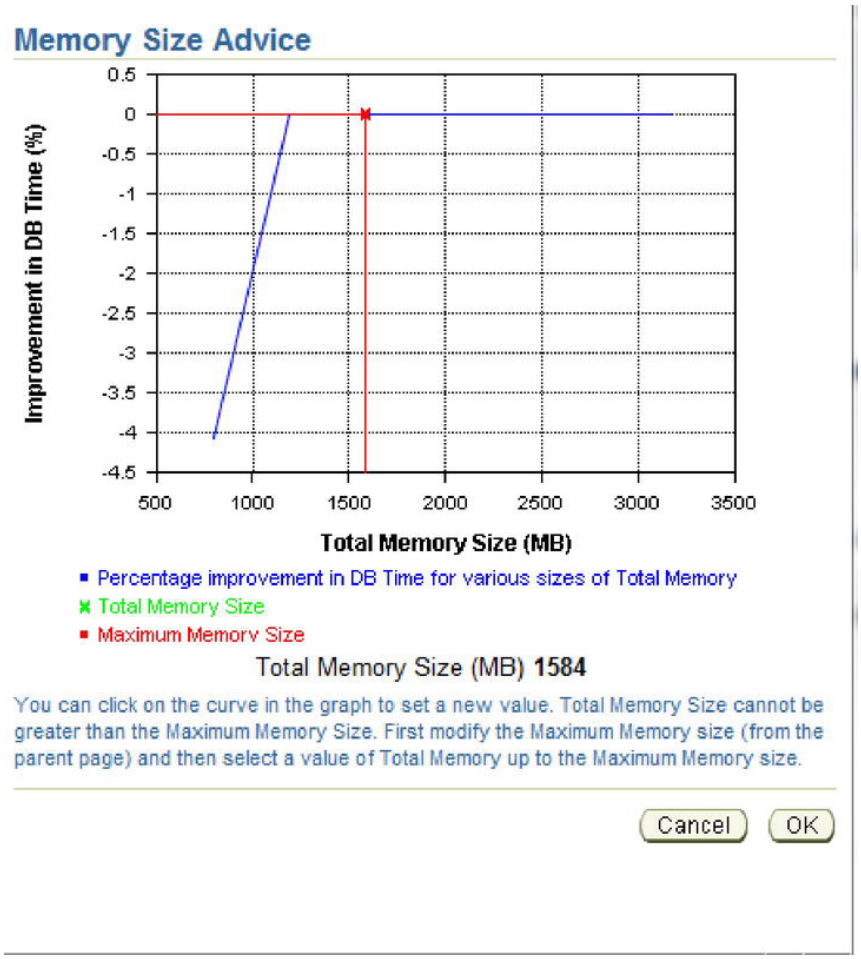

Figure 3-1. *The Memory Size Advice graph in Database Control*

How It Works

When you implement automatic memory management, Oracle automatically adjusts memory between the various components of total memory—the SGA and the PGA—during the course of the instance, depending on the workload characteristics of the instance. Instead of running queries using various views to figure out if your current memory allocation is optimal, you can follow a couple of easy steps to figure things out quickly. You can review the Automatic Database Diagnostic Monitor(ADDM) reports to see if they contain any comments or recommendations about inadequate memory. ADDM recommends

that you add more memory to the MEMORY_TARGET parameter, if it considers that the current memory allocation is insufficient for optimal performance.

If the ADDM reports recommend that you increase the size of the MEMORY_TARGET parameter, the next question is by how much you should increase the memory allocation. Oracle's built-in Memory Advisors come in handy for just this purpose. Even in the absence of a recommendation by the ADDM, you can play with the Memory Advisors to get an idea of how an increase or decrease in the MEMORY_TARGET parameter will impact performance.

You can also choose to optimize your PGA memory allocation from the same Memory Advisors page by clicking PGA at the top of the page. In the case of the PGA, in the PGA Target Advice page, the graphs plots the PGA cache hit percentage against the PGA target size. Ideally, you'd want the PGA cache ratio somewhere upward of around 70%. The PGA Target Advice page will help you determine approximately what value you should assign to the PGA_AGGREGATE_TARGET parameter to achieve your performance goals.

3-6. Tuning PGA Memory Allocation

Problem

You've decided to set a specific minimum memory size for the PGA_AGGREGATE_TARGET parameter, although you're using Oracle's automatic memory management to allocate memory. You'd like to know the best way to figure out the optimal PGA memory allocation.

Solution

There are no hard and fast rules for allocating the size of the PGA_AGGREGATE_TARGET parameter. Having said that, if you're operating a data warehouse, you're more likely to need much larger amounts of memory set apart for the PGA component of Oracle's memory allocation. You can follow these basic steps to allocate PGA memory levels:

1. Use a starting allocation more or less by guessing how much memory you might need for the PGA.
2. Let the database run for an entire cycle or two of representative workload. You can then access various Oracle views to figure out if your first stab at the PGA estimation was on target.

How It Works

Although automatic memory management is designed to optimally allocate memory between the two major components of Oracle memory—the SGA and the PGA—it's certainly not uncommon for many DBAs to decide to set their own values for both the SGA and the PGA, either as part of the alternative mode of memory management, automatic shared memory management, wherein you set the SGA_TARGET and the PGA_AGGREGATE_TARGET parameters explicitly to manage memory, or even under the newer automatic memory management system. Unlike the SGA_TARGET parameter, where cache hit ratios could mislead you as to the efficacy of the instance, you'll find that an analysis of the hit ratios for the PGA_AGGREGATE_TARGET parameter are not only valid, but also highly useful in configuring the appropriate sizes for this parameter.

The Oracle database uses PGA memory to perform operations such as sorting and hashing. The memory you allocate to the PGA component is used by various SQL work areas running in the database (along with other users of the PGA memory such as PL/SQL and Java programs). Ideally, you'd want all work areas to have an optimal PGA memory allocation. When memory allocation is ideal, a work area performs the entire operation in memory. For example, if a SQL operation involves sorting operations, under optimal PGA allocation, all of the sorting is done within the PGA memory allocated to that process. If the PGA memory allocation isn't optimal, the work areas make one or more passes over the data—this means they have to perform the operations on disk, involving time consuming I/O. The more passes the database is forced to make, the more I/O and the longer it takes to process the work.

Oracle computes the PGA cache hit percentage with the following formula:

```
Bytes Processed * 100 /(Bytes processed + Extra Bytes Processed)
```

Bytes Processed is the total number of bytes processed by all the PGA memory using SQL operations since the instance started. You should seek to get this ratio as close to 100 as possible—if your PGA cache hit percentage is something like 33.37%, it's definitely time to increase PGA memory allocation by raising the value you've set for the PGA_AGGREGATE_TARGET parameter. Fortunately, the PGA_AGGREGATE_TARGET parameter is dynamic, so you can adjust this on the fly without a database restart, to address a sudden slowdown in database performance due to heavy sorting and hashing activity.

You can issue the following simple query to find out the PGA cache hit percentage as well as a number of PGA performance-related values.

```
SQL>select  * from v$pgastat;
NAME                                     VALUE            UNIT
-------------------------------       -------------     ----------
aggregate PGA target parameter             570425344     bytes
aggregate PGA auto target                  481397760     bytes
total PGA inuse                             35661824     bytes
total PGA allocated                         70365184     bytes
maximum PGA allocated                      195681280     bytes
over allocation count                              0     bytes processed
extra bytes read/written                           0     bytes
cache hit percentage                             100     percent
SQL>
```

Since we're using our test database here, the cache hit percentage is a full 100%, but don't expect that in a real-life database, especially if it is processing a lot of data warehouse–type operations!

You can also use the V$SQL_WORKAREA_HISTOGRAM view to find out how much of its work the database is performing in an optimal fashion. If a work area performs its work optimally, that is, entirely within PGA memory, it's counted as part of the OPTIMAL_COUNT column. If it makes one or more passes, it will go under the ONEPASS_COUNT or the MULTIPASS_COUNT columns. Here's a query that shows how to do this:

```
SQL> select optimal_count, round(optimal_count*100/total, 2) optimal_perc,
  2  onepass_count, round(onepass_count*100/total, 2) onepass_perc,
  3  multipass_count, round(multipass_count*100/total, 2) multipass_perc
  4  from
  5   (select decode(sum(total_executions), 0, 1, sum(total_executions)) total,
  6   sum(OPTIMAL_EXECUTIONS) optimal_count,
  7   sum(ONEPASS_EXECUTIONS) onepass_count,
  8   sum(MULTIPASSES_EXECUTIONS) multipass_count
  9   from  v$sql_workarea_histogram
 10* where low_optimal_size > (64*1024))
SQL> /
```

```
OPTI_COUNT     OPTI_PERC    ONEPASS_CT     ONEPASS_PERC   MULTIPASS_CT     MULTIPASS_PERC
-----------   -----------  -----------    -------------  -------------   ---------------
8069              100             0             0              0                0
```

One pass is slower than none at all, but a multi-pass operation is a sure sign of trouble in your database, especially if it involves large work areas. You'll most likely find that your database has slowed to a crawl and is unable to scale efficiently when the database is forced to make even a moderate amount of multi-pass executions that involve large work areas, such as those that are sized 256 MB to 2 GB. To make sure that you don't have any huge work areas running in the multi-pass mode, issue the following query:

```
SQl> select low_optimal_size/1024 low,
     (high_optimal_size+1)/1024 high,
     optimal_executions, onepass_executions, multipasses_executions
     from v$sql_workarea_histogram
     where total_executions !=0;
```

You can also execute simple queries involving views such as `V$SYSSTAT` and `V$SESSTAT` as shown here, to find out exactly how many work areas the database has executed with an optimal memory size (in the PGA), one-pass memory size, and multi-pass memory sizes.

```
SQL>select name profile, cnt, decode(total, 0, 0, round(cnt*100/total)) percentage
    from (SELECT name, value cnt, (sum(value) over ()) total
    from V$SYSSTAT
    where name like 'workarea exec%');
```

Remember that this query shows the total number of work areas executed under each of the three different execution modes (optimal, one-pass, and multi-pass), since the database was started. To get the same information for a specific period of time, you can use queries involving Automatic Session History (ASH).

You can also view the contents of the Automatic Workload Repository (AWR) for information regarding how the database used its PGA memory for any interval you choose. If you regularly create these reports and save them, you can have a historical record of how well the PGA allocation has been over a period of time. You can also view the ADDM report for a specific time period to evaluate what proportion of work the database is executing in each of the three modes of execution we discussed earlier. In a data warehouse environment, where the database processes huge sorts and hashes, the optimal allocation of the PGA memory is one of the most critical tasks a DBA can perform.

3-7. Configuring the Server Query Cache

Problem

You'd like to set up the server query cache that's part of Oracle's memory allocation.

Solution

You can control the behavior of the server query cache by setting three initialization parameters: `RESULT_CACHE_MAX_SIZE`, `RESULT_CACHE_MAX_RESULT`, and `RESULT_CACHE_REMOTE_EXPIRATION`. For example, you can use the following set of values for the three server result cache-related initialization parameters:

```
RESULT_CACHE_MAX_SIZE=500M      /* Megabytes
RESULT_CACHE_MAX_RESULT=20     /* Percentage
RESULT_CACHE_REMOTE_EXPIRATION=3600       /* Minutes
```

You can disable the server result cache by setting the `RESULT_CACHE_MAX_SIZE` parameter to 0 (any non-zero value for this parameter enables the cache).

If you set the `RESULT_CACHE_MODE` initialization parameter to `FORCE`, the database caches all query results unless you specify the `/*+ NO_RESULT_CACHE */` hint to exclude a query's results from the cache. The default (and the recommended) value of this parameter is `MANUAL`, meaning that the database caches query results only if you use the appropriate query hint or table annotation (explained later).You can set this parameter at the system level or at the session level, as shown here:

```
SQL> alter session set result_cache_mode=force;
```

You can remove cached results from the server result cache by using the `FLUSH` procedure from the `DBMS_RESULT_CACHE` package, as shown here:

```
SQL> execute dbms_result_cache.flush
```

How It Works

The server result cache offers a great way to store results of frequently executed SQL queries and PL/SQL functions. This feature is easy to configure with the help of the three initialization parameters we described in the "Solution" section. However, remember that Oracle doesn't guarantee that a specific query or PL/SQL function result will be cached no matter what.

In some ways, you can compare the Oracle result cache feature to other Oracle result storing mechanisms such as a shared PL/SQL collection, as well as a materialized view. Note, however, that whereas Oracle stores a PL/SQL collection in private PGA areas, it stores the result cache in the shared pool as one of the shared pool components. As you know, the shared pool is part of the SGA. Materialized views are stored on disk, whereas a result cache is in the much faster random access memory. Thus, you can expect far superior performance when you utilize the result cache for storing result sets, as opposed to storing pre-computed results in a materialized view. Best of all, the result cache offers the Oracle DBA a completely hands-off mode of storing frequently accessed result sets—you don't need to create any objects, as in the case of materialized views, index them, or refresh them. Oracle takes care of everything for you.

The server query cache is part of the shared pool component of the SGA. You can use this cache to store both SQL query results as well as PL/SQL function results. Oracle can cache SQL results in the SQL result cache and PL/SQL function results in the PL/SQL function result cache. You usually use the server query cache to make the database cache queries that are frequently executed but need to access a large number of rows per execution. You configure the server query cache by setting the following initialization parameters in your database.

- `RESULT_CACHE_MAX_SIZE`: This sets the memory allocated to the server result cache.
- `RESULT_CACHE_MAX_RESULT`: This is the maximum amount of memory a single result in the cache can use, in percentage terms. The default is 5% of the server result cache.

- RESULT_CACHE_REMOTE_EXPIRATION: By default, any result that involves remote objects is not cached. Thus, the default setting of the RESULT_CACHE_REMOTE_EXPIRATION parameter is 0. You can, however, enable the caching of results involving remote objects by setting an explicit value for the RESULT_CACHE_REMOTE_EXPIRATION parameter.

Setting the three initialization parameters for the server result cache merely enables the cache. To actually use the cache for your SQL query results, or for PL/SQL function results, you have to either enable the cache database-wide, or for specific queries, as the following recipes explain.

Once the database stores a result in the server result cache, it retains it there until you either remove it manually with the DBMS_RESULT_CACHE.FLUSH procedure, or until the cache reaches its maximum size set by the RESULT_CACHE_MAX_SIZE parameter. The database will remove the oldest results from the cache when it needs to make room for newer results when it exhausts the capacity of the server result cache.

3-8. Managing the Server Result Cache

Problem

You've enabled the server result cache, but you aren't sure if queries are taking advantage of it. You also would like to find out how well the server result cache is functioning.

Solution

You can check the status of the server result cache by using the DBMS_RESULT_CACHE package. For example, use the following query to check whether the cache is enabled:

```
SQL> select dbms_result_cache.status() from dual;

DBMS_RESULT_CACHE.STATUS()
--------------------------------------------------
ENABLED
SQL>
```

You can view a query's explain plan to check whether a query is indeed using the SQL query cache, after you enable that query for caching, as shown in the following example. The explain plan for the query shows that the query is indeed making use of the SQL query cache component of the result cache.

```
SQL> select  /*+ RESULT_CACHE */ department_id, AVG(salary)
     from hr.employees
     group by department_id;
.
.
.
```

```
-------------------------------------------------------------
| Id | Operation          | Name                        |Rows
-------------------------------------------------------------
| 0 | SELECT STATEMENT    |                             | 11
| 1 |  RESULT CACHE       | 8fpza04gtwsfr6n595au15yj4y |
| 2 |   HASH GROUP BY     |                             | 11
| 3 |    TABLE ACCESS FULL| EMPLOYEES                   | 107
-------------------------------------------------------------
```

You can use the MEMORY_REPORT procedure of the DBMS_RESULT_CACHE package to view how Oracle is allocating memory to the result cache, as shown here:

```
SQL> SET SERVEROUTPUT ON
SQL> execute dbms_result_cache.memory_report

R e s u l t  C a c h e  M e m o r y  R e p o r t
[Parameters]
Block Size = 1024 bytes
Maximum Cache Size = 950272 bytes (928 blocks)
Maximum Result Size = 47104 bytes (46 blocks)
[Memory]
Total Memory = 46340 bytes [0.048% of the Shared Pool]
... Fixed Memory = 10696 bytes [0.011% of the Shared Pool]
... State Object Pool = 2852 bytes [0.003% of the Shared Pool]
... Cache Memory = 32792 bytes (32 blocks) [0.034% of the Shared Pool]
....... Unused Memory = 30 blocks
....... Used Memory = 2 blocks
........... Dependencies = 1 blocks
........... Results = 1 blocks
............... SQL = 1 blocks

PL/SQL procedure successfully completed.
SQL>
```

You can monitor the server result cache statistics by executing the following query:

```
SQL> select name, value from V$RESULT_CACHE_STATISTICS;

NAME                          VALUE
--------------------          ----------
Block Size (Bytes)            1024
Block Count Maximum           3136
Block Count Current             32
Result Size Maximum (Blocks)   156
Create Count Success             2
Create Count Failure             0
Find Count                       0
Invalidation Count               0
Delete Count Invalid             0
Delete Count Valid               0
SQL>
```

The `Create Count Success` column shows the number of queries that the database has cached in the server result cache, and the `Invalidation Count` column shows the number of times the database has invalidated a cached result.

How It Works

You can monitor and manage the server result cache with the `DBMS_RESULT_CACHE` package provided by Oracle. This package lets you manage both components of the server result cache, the SQL result cache and the PL/SQL function result cache. You can use the `DBMS_RESULT_CACHE` package to manage the server result cache memory allocation, as well as to bypass and re-enable the cache (when recompiling PL/SQL packages, for example), flush the cache, and to view statistics relating to the server query cache memory usage.

The server result cache is part of Oracle's shared pool component of the SGA. Depending on the memory management system in use, Oracle allocates a certain proportion of memory to the server result cache upon starting the database. Here are the rules that Oracle uses for deciding what percentage of the shared pool it allocates to the server result cache:

- If you're using automatic memory management by setting the `MEMORY_TARGET` parameter, Oracle allocates 0.25% of the `MEMORY_TARGET` parameter's value to the server result cache.
- If you're using automatic shared memory management with the `SGA_TARGET` parameter, the allocation is 0.5% of the `SGA_TARGET` parameter.
- If you're using manual memory management by setting the `SHARED_POOL_SIZE` parameter, the allocation is 1% of the `SHARED_POOL_SIZE` parameter.

In an Oracle RAC environment, you can size the server cache differently on each instance, just as you do with the `MEMORY_TARGET` and other instance-related parameters. Similarly, when you disable the server result cache by setting the `RESULT_CACHE_MAX_SIZE` to 0, you must do so on all the instances of the cluster.

The server result cache can potentially reduce your CPU overhead by avoiding recomputation of results, where data may have to be fetched repeatedly from the buffer cache, which results in a higher number of logical I/Os. When you opt to cache the results in the cache instead of pre-computing them and storing them in materialized views, you can also potentially reduce the database disk I/O as well. Just remember that the primary purpose of the result cache isn't to store just any results in memory—it's mainly designed to help with the performance of queries that involve static or mostly static data. Thus, a data warehouse or decision support system is likely to derive the greatest benefit from this new performance feature.

3-9. Caching SQL Query Results

Problem

You've configured the server result cache in your database. You would now like to configure a set of queries whose result you would like to be cached in the server result cache.

Solution

Set the RESULT_CACHE_MODE initialization parameter to the appropriate value for making queries eligible for caching in the server result cache. You can set the RESULT_CACHE_MODE to the value FORCE, to force all SQL results to be cached by the database. Oracle recommends that you set the RESULT_CACHE_MODE parameter value to MANUAL, which happens to be the default setting. The RESULT_CACHE_MODE parameter is dynamic, so you can set this parameter with the ALTER SYSTEM (or the ALTER SESSION) command.

When you set the RESULT_CACHE_MODE parameter to the value MANUAL, the database caches the results of only specific queries—queries that you enable for caching by using either a query hint or a table annotation. The following example shows how to use the query hint method to specify a query's results to be cached in the server result cache.

```
SQL> select /*+ RESULT_CACHE */ prod_id, sum(amount_sold)
     from  sales
     group by prod_id
     order by prod_id;
```

The query hint /*+RESULT_CACHE */ tells the database to cache the results of the previous query. You can turn off the result caching for this query by using the /+ NO_RESULT_CACHE */ hint, as shown in the following example.

```
SQL> select /*+ NO_RESULT_CACHE */ prod_id, SUM(amount_sold)
     from  sales
     group by prod_id
     order by prod_id;
```

When you run this query, the server won't cache the results of this query any longer, because you've specified the MANUAL setting for the RESULT_CACHE_MODE initialization parameter.

The alternative way to specify the caching of a query's results is to use the table annotation method. Under this method, you specify the RESULT_CACHE attribute when you create a table or alter it. You can annotate a CREATE TABLE or ALTER TABLE statement with the RESULT_CACHE attribute in two different modes: DEFAULT or FORCE, as shown in the following examples:

```
SQL> create table stores (...) RESULT_CACHE (MODE DEFAULT);
SQL> alter table stores RESULT_CACHE (MODE FORCE);
```

We explain the implications of setting the RESULT_CACHE_MODE initialization parameter to FORCE in the "How it Works" section.

How It Works

If you set the value of the RESULT_CACHE_MODE parameter to FORCE, Oracle executes all subsequent queries only once. Upon subsequent executions of those queries, the database retrieves the results from the cache. Obviously, you don't want to store the results of each and every SQL statement, because of the performance implications, as well as the fact that the server result cache may run out of room. Thus Oracle recommends that you specify the MANUAL setting for the RESULT_CACHE_MODE parameter.

If you can set the result caching behavior with the use of SQL hints, why use table annotations? Table annotations are an easy way to specify caching without having to modify the application queries directly by adding the SQL hints. It's easier to simply issue an ALTER TABLE statement for a set of tables, to enable the caching of several queries that use that set of tables. Note that when you annotate a table, those annotations apply to the entire query, but not for fragments of that query.

If you annotate a table *and* specify a SQL hint for query caching, which method will Oracle choose to determine whether to cache a query's results? Query hints are given precedence by the database over table annotations. However, the relationship between SQL hints and table annotations is complex, and whether the database caches a query's results also depends on the specific value of the table annotation, as summarized in the following discussion.

Table Annotations and Query Hints

As mentioned earlier, you can use both SQL hints and table annotations to specify which query results you want the result cache to store, with hints overriding annotations. Use either the `ALTER TABLE` or the `CREATE TABLE` statements to annotate tables with the result cache mode. Here's the syntax for a `CREATE` or `ALTER TABLE` statement when annotating a table or a set of tables:

```
CREATE|ALTER TABLE [<schema>.]<table> ... [RESULT_CACHE (MODE {FORCE|DEFAULT})]
```

Note the following important points about table annotations:

- The mode value `DEFAULT` is, of course, the default value, and this merely removes any table annotations you may have set and doesn't permit caching of results that involve this table.
- If you set at least one table to the `DEFAULT` mode, any query involving that table won't be allowed to store its results in the cache.
- If you set all the tables in a query to the `FORCE` mode, Oracle will always consider that query for caching—unless you turn off the caching with the `NO_RESULT_CACHE` hint within the query.
- If you set at least one table in a query to `DEFAULT` by annotating a `CREATE TABLE` statement, as shown here, Oracle caches results of this query only if you've either set the `RESULT_CACHE_MODE` parameter to `FORCE` or specified the `RESULT_CACHE` hint.

```
SQL> CREATE TABLE sales (id number) RESULT_CACHE (MODE DEFAULT);
```

Note that the previous statement is equivalent to the following statement, because the default value of the attribute `RESULT_CACHE` is `DEFAULT`.

```
SQL> CREATE TABLE sales (id number);
```

You can check that the database created the table `SALES` with the `RESULT_CACHE` attribute set to the value `DEFAULT`:

```
SQL> select table_name, result_cache from user_tables where table_name ='SALES';

TABLE_NAME                     RESULT_
------------------------------ -------
SALES                          DEFAULT

SQL>
```

If you specify the table creation statement with the `RESULT_CACHE(MODE FORCE)` option, this will prevail over the `MANUAL` setting of the `RESULT_CACHE_MODE` initialization parameter that you've set at the session level. The following example illustrates how this works.

1. First alter the table STORES to specify the RESULT_CACHE attribute with the MODE FORCE option:

```
SQL> alter table stores result_cache (mode force);
```

2. Then ensure that you've set the RESULT_CACHE_MODE initialization parameter to the value MANUAL.

3. Then execute the following query:

```
SQL> select prod_id, sum(amount_sold)
     from stores
     group by prod_id
     having prod_id=999;
```

On subsequent executions, the database will retrieve the results for the preceding query from the server result cache. The reason this is so is that when you specify the RESULT_CACHE (MODE FORCE) annotation, it overrides the MANUAL setting for the RESULT_CACHE_MODE parameter. Remember that when you set this parameter to the MANUAL mode, Oracle will cache query results only if you specify a query hint or annotation. The query shown here doesn't involve the use of a hint, but its results are cached because the RESULT_CACHE (MODE FORCE) annotation makes the database behave the same way as it does when the RESULT_CACHE_MODE parameter is set to FORCE—it caches the query results of all eligible queries.

Query hints, however, ultimately trump the RESULT_CACHE (MODE FORCE) annotation, as shown in the following example.

First alter the table STORES to specify the RESULT_CACHE attribute with the MODE FORCE option:

```
SQL> alter table stores result_cache (mode force);
```

Ensure that you've set the RESULT_CACHE_MODE initialization parameter to the value MANUAL.

Execute the following query:

```
SQL>select /*+ no_result_cache */ *
     from stores
     order by time_id desc;
```

In this example, even though you've annotated the STORES table to allow caching with the MODE FORCE option, the /*+ no_result_cache */ hint overrides the annotation and prevents the caching of the results of any query that involves the STORES table.

Requirements for Using the Result Cache

There are a few read consistency requirements that a query must satisfy, in order for the database to use the result cache:

- In cases involving a snapshot, if a read-consistent snapshot builds a result, it must retrieve the latest committed state of the data, or the query must use a flashback query to point to a specific point in time.
- Whenever a session transaction is actively referencing the tables or views in a query, the database won't cache the results from this query for read consistency purposes.

In addition to the read consistency requirements for result caching, the following objects or functions, when present in a query, make that query ineligible for caching:

- CURRVAL and NEXTVAL pseudo columns
- The CURRENT_DATE, CURRENT_TIMESTAMP, USERENV_CONTEXT, SYS_CONTEXT (with non-constant variables), SYSDATE, and SYS_TIMESTAMP
- Temporary tables
- Tables owned by SYS and SYSTEM

3-10. Caching Client Result Sets

Problem

You use a lot of OCI applications that involve repetitive queries. You would like to explore how you can cache the result sets on the client.

Solution

You can enable client-side query caching of SQL query results by enabling the client result cache. The client result cache works similarly to the server result cache in many ways, but is separate from the server cache. You set the client result cache by setting the following initialization parameters:

- CLIENT_RESULT_CACHE_SIZE: To enable the client result cache, set this parameter to 32 KB or higher, up to a maximum of 2 GB. By default, this parameter is set to zero, meaning the client query cache is disabled. Unlike in the case of the server result cache, the CLIENT_RESULT_CACHE_SIZE parameter value sets the maximum size of result set cache per process, not for the entire instance. Since this parameter isn't a dynamic one, a reset requires that you bounce the instance. You have to determine the size of this parameter based on the potential number of results that'll be cached, as well as the average size of the result set, which depends both on the size of the rows and the number of rows in the result set.

■ **Tip** Oracle cautions you not to set the CLIENT_RESULT_CACHE_SIZE during database creation, due to potential errors.

- CLIENT_RESULT_CACHE_LAG: This parameter lets you specify the maximum amount of time the client result cache can fall behind a change that affects the result set values. By default, the value of this parameter is set to 3,000 milliseconds, so you can omit this parameter if this time interval is adequate for you. Changes in this parameter also need a restart of the database, because it's a static parameter.

■ **Note** You can use the client query cache with all OCI applications and drivers built using OCI.

In addition, you must specify the value of the initialization parameter `COMPATIBLE` at 11.0.0 or higher to enable the client result cache. If you want to cache views on the client side, the value of the `COMPATIBLE` parameter must be 11.2.0.0 or higher.

In addition to the initialization parameters you must set on the database server, Oracle lets you also include an optional client-side configuration file to specify values that override the values of the client query cache–related parameters in the initialization file. If you specify any of these parameters, the value of that parameter will override the value of the corresponding parameter in the server initialization file. You can specify one or more of the following parameters in the optional client configuration file, which you can include in the `sqlnet.ora` file on the client:

`OCI_RESULT_CACHE_MAX_SIZE`: Maximum size (in bytes) for the query cache for each individual process

`OCI_RESULT_CACHE_MAX_RSET_SIZE`: Maximum size of a result set in bytes in any process

`OCI_RESULT_CACHE_MAX_RSET_ROWS`: Maximum size of the result set in terms of rows, in any process

You can't set any query cache lag–related parameters in the client-side file. Once you set the appropriate initialization parameters and the optional client-side configuration file, you must enable and disable queries for caching with either the `/*+ result_cache +/` (and the `/*+ no_result_cache +/`) hint, or table annotations. Once you do this, the database will attempt to cache all eligible queries in the client query cache.

How It Works

You can deploy client-side query result caching to speed up the responses of queries that your database frequently executes in an OCI application. The database keeps the result set data consistent with any database changes, including session changes. You can potentially see a huge performance improvement for repetitive queries because the database retrieves the results from the local OCI client process rather than having to re-access the server via the network and re-execute the same query there and fetch those results. When an OCI application issues an `OCIStmtExecute()` or `OCIStmtFetch()` call, Oracle processes those calls locally on the client, if the query results are already cached in the client query cache.

The big advantage of using a client-side query cache is that it conserves your server memory usage and helps you scale up your applications to serve more processes. The client query cache is organized on a per-process basis rather than a per-session basis. Multiple client sessions can share the same cached result sets, all of which can concurrently access the same result sets through multiple threads and multiple statements. The cache automatically invalidates the cached result sets if an OCI process finds significant database changes on the database server. Once a result set is invalidated, the query will be executed again and a fresh result set is stored in the cache.

■ **Tip** Oracle recommends that you use client-side caching only for read-only or mostly read-only queries.

You can optionally set the RESULT_CACHE_MODE parameter (see Recipe 3-7) to control caching behavior, but by default, this parameter is set to the value MANUAL, so you can leave it alone. You really don't want to set this parameter to its alternative value FORCE, which compels the database to cache the results of every eligible SQL query—obviously, the cache will run out of room before too long! You can then specify either the appropriate query hint at the SQL level or table annotations at the table level to control the client-side result caching. What if you've already set up server-side result caching through the server query cache? No matter. You can still enable client result caching. Just remember that by default, client-side caching is disabled and server-side result caching is enabled.

When implementing client query caching, it's important that an OCIStmtExecute() call is made so a statement handle can match a cached result. The very first OCIStmtExecute() call for an OCI statement handle goes to the server regardless of the existence of a cached result set. Subsequent OCIStmtExecute() calls will use the cached results if there's a match. Similarly, only the first OCIStmtFetch() call fetches rows until it gets the "Data Not Found" error—subsequent fetch calls don't need to fetch the data until they get this error, if the call matches the cached result set. Oracle recommends that your OCI applications either cache OCI statements or use statement caching for OCI statement handles, so they can return OCI statements that have already been executed. The cached set allows multiple accesses from OCI statement handles from single or multiple sessions.

As with the server-side cache, you can set the RESULT_CACHE_MODE parameter to FORCE to specify query caching for all queries. Oracle recommends you set this parameter to the alternative value of MANUAL and use SQL hints (/*+ result_cache */) in the SQL code the application passes to the OCIStmtPrepare(), and OCIStmtPrepare2() calls. You can also use table annotations, as explained in Recipe 3-7, to specify caching when you create or alter a table. All queries that include that table will follow the caching specifications subsequently.

You can query the V$CLIENT_RESULT_CACHE_STATS view for details such as the number of cached result sets, number of cached result sets invalidated, and the number of cache misses. The statistic Create Count Success, for example, shows the number of cached result sets the database didn't validate before caching all rows of the result set. The statistic Create Count Failure shows the number of the cached result sets that didn't fetch all rows in the result set.

3-11. Caching PL/SQL Function Results

Problem

You've set up a server query cache in your database and would like to implement the caching of certain PL/SQL function results.

Solution

Oracle's server query cache (Recipe 3-7) helps you cache both normal SQL query results as well as PL/SQL function results. By using the server result cache, you can instruct the database to cache the results of PL/SQL functions in the SGA. Other sessions can use these cached results, just as they can use cached query results with the query result cache. Once you've configured the server query cache by setting the appropriate initialization parameters (please see Recipe 3-7), you are ready to make use of this feature.

You must specify the RESULT_CACHE clause inside a function to make the database cache the function's results in the PL/SQL function result cache. When a session invokes a function after you enable caching, it first checks to see if the cache holds results for the function with identical parameter

values. If so, it fetches the cached results and doesn't have to execute the function body. Note that if you declare a function first, you must also specify the RESULT_CACHE clause in the function declaration, in addition to specifying the clause within the function itself.

Listing 3-1 shows how to cache a PL/SQL function's results.

Listing 3-1. *Creating a PL/SQL Function with the /*+result_cache*/ Hint*

```
SQL> create or replace package store_pkg is
      type store_sales_record is record (
      store_name stores.store_name%TYPE,
     mgr_name   employees.last_name%type,
     store_size  PLS_INTEGER
     );
     function get_store_info (store_id PLS_INTEGER)
      RETURN store_info_record
      RESULT_CACHE;
     END store_pkg;
    /
Create or replace package body store_pkg is
    FUNCTION get_store_sales (store_id PLS_INTEGER)
    RETURN store_sales_record
    RESULT_CACHE RELIES_ON (stores, employees)
  IS
    rec  store_sales_record;
  BEGIN
    SELECT store_name INTO rec.store_name
    FROM stores
    WHERE store_id = store_id;
    SELECT e.last_name INTO rec.mgr_name
    FROM stores d, employees e
    WHERE d.store_id = store_id
    AND d.manager_id = e.employee_id;
     SELECT COUNT(*) INTO rec.store_size
    FROM EMPLOYEES
    WHERE store_id = store_id;
     RETURN rec;
  END get_store_sales;
END store_pkg;
/
```

Let's say you invoke the function with the following values:

```
SQL> execute store_pkg.get_store_sales(999)
```

The first execution will cache the PL/SQL function's results in the server result cache. Any future executions of this function with the same parameters (999) won't require the database to re-execute this function—it merely fetches the results from the server result cache.

Note that in addition to specifying the RESULT_CACHE clause in the function declaration, you can optionally specify the RESULT_CACHE RELIES ON clause in the function body, as we did in this example. In this case, specifying the RESULT_CACHE_RELIES_ON clause means that the result cache relies on the tables STORES and EMPLOYEES. What this means is that whenever these tables change, the database invalidates all the cached results for the get_store_info function.

How It Works

The PL/SQL function result cache uses the same server-side result cache as the query result cache, and you set the size of the cache using the `RESULT_CACHE_MAX_SIZE` and `RESULT_CACHE_MAX_RESULT` initialization parameters, with the first parameter fixing the maximum SGA memory that the cache can use, and the latter fixing the maximum percentage of the cache a single result can use. Unlike the query result cache, the PL/SQL function cache may quickly gather numerous results for caching, because the cache will store multiple values for the same function, based on the parameter values. If there's space pressure within the cache, older cached function results are removed to make room for new results.

Oracle recommends that you employ the PL/SQL function result cache to cache the results of functions that execute frequently but rely on static or mostly static data. The reason for specifying the static data requirement is simple: Oracle automatically invalidates cache results of any function whose underlying views or tables undergo committed changes. When this happens, the invocation of the function will result in a fresh execution.

Whenever you introduce a modified version of a package on which a result cache function depends (such as in Listing 3-1), the database is supposed to automatically flush that function's cached results from the PL/SQL function cache. In our example, when you hot-patch (recompile) the package `store_pkg`, Oracle technically must flush the cached results associated with the `get_store_info` function. However, sometimes the database may fail to automatically flush these results. In order to ensure that the cached results of the function are removed, follow these steps whenever you recompile a PL/SQL unit such as a package that includes a cache-enabled function.

1. Place the result cache in the bypass mode.

   ```
   SQL> execute DBMS_RESULT_CACHE.bypass(true);
   ```

2. Clear the cache with the `flush` procedure.

   ```
   SQL> execute DBMS_RESULT_CACHE.flush;
   ```

3. Recompile the package that contains the cache-enabled function.

   ```
   SQL> alter package store_pkg compile;
   ```

4. Re-enable the result cache with the `bypass` procedure.

   ```
   SQL> execute DBMS_RESULT_CACHE.bypass(false);
   ```

■ **Tip** If you're using both the SQL query cache and PL/SQL function result cache simultaneously, remember that both caches are actually part of the same server query cache. In cases such as this, ensure that you've sized the `RESULT_CACHE_SIZE` parameter high enough to hold cached results from both SQL queries and PL/SQL functions.

If you're operating in an Oracle RAC environment, you must run the cache enabling and disabling steps on each RAC instance.

Of course, when you bypass the cache temporarily in this manner, during that time the cache is bypassed, the database will execute the function, instead of seeking to retrieve its results from the cache. The database will also bypass the result cache on its own for a function if a session is in the process of performing a DML statement on a table or view that the function depends on. This automatic bypassing

of the result cache by the database ensures that users won't see uncommitted changes of another session in their own session, thus ensuring read consistency.

You can use the V$RESULT_CACHE_STATISTICS, V$RESULT_CACHE_MEMORY, V$RESULT_CACHE_OBJECTS, and V$RESULT_CACHE_DEPENDENCY views to monitor the usage of the server result cache, which includes both the SQL result cache as well as the PL/SQL function result cache.

Important Considerations

While a PL/SQL function cache gets you results much faster than repetitive execution of a function, PL/SQL collections (arrays)–based processing could be even faster because the PL/SQL runtime engine gets the results from the collection cache based in the PGA rather than the SGA. However, since this requires additional PGA memory for each session, you'll have problems with the collections approach as the number of sessions grows large. The PL/SQL function is easily shareable by all concurrent sessions, whereas you can set up collections for sharing only through additional coding.

You must be alert to the possibility that if your database undergoes frequent DML changes, the PL/SQL function cache may not be ideal for you—it's mostly meant for data that never changes, or does so only infrequently. Even if you set the RESULT_CACHE_REMOTE_EXPIRATION parameter to a high value, any DML changes will force the database to invalidate the cached PL/SQL function cache result sets.

Oracle will invalidate result cache output when it becomes out of date, so when a DML statement modifies the rows of a table that is part of a PL/SQL function that you've enabled for the function cache, the database invalidates the cached results of that function. This could happen if the specific rows that were modified aren't part of the PL/SQL function result set. Again, remember that this limitation could be "bypassed" by using the PL/SQL function cache in databases that are predominantly read-only.

Restrictions on the PL/SQL Function Cache

In order for its results to be cached, a fuction must satisfy the following requirements:

- The function doesn't use an OUT or an IN OUT parameter.
- An IN parameter of a function has one of these types: BLOB, CLOB, NCLOB, REF CURSOR, Collection Object, and Record.
- The function isn't a pipelined table function.
- The function isn't part of an anonymous block.
- The function isn't part of any module that uses invoker's rights as opposed to definer's rights.
- The function's return type doesn't include the following types: BLOB, CLOB, NCLOB, REF CURSOR, Object, Record, or a PL/SQL collection that contains one of the preceding unsupported return types.

3-12. Configuring the Oracle Database Smart Flash Cache

Problem

Your AWR (Automatic Workload Repository) report indicates that you need a much larger buffer cache. You also notice that the shared pool is sized correctly, and so you can't set a higher minimum level for the buffer cache by reducing the shared pool memory allocation. In addition, you're limited in the amount of additional memory you can allocate to Oracle.

Solution

Depending on your operating system, you can use the new Oracle Database Smart Flash Cache feature, in cases where the database indicates that it needs a much larger amount of memory for the buffer cache. Right now, the Flash Cache feature is limited to Solaris and Oracle Linux operating systems.

Set the following parameters to turn the Flash Cache feature on:

- `DB_FLASH_CACHE_FILE`: This parameter sets the pathname and the file name for the flash cache. The file name you specify will hold the flash cache. You must use a flash device for the flash cache file, and it could be located in the operating system file system, a raw disk, or an Oracle ASM disk group—for example:

```
DB_FLASH_CACHE_FILE= "/dev/sdc"
DB_FLASH_CACHE_FILE = "/export/home/oracle/file_raw"         /* raw file
DB_FLASH_CACHE_FILE = "+dg1/file_asm"                        /* using ASM storage
```

- DB_FLASH_CACHE_SIZE: This parameter sets the size of the flash cache storage. Here's an example:

```
DB_FLASH_CACHE_SIZE = 8GB
```

You can toggle between a system with a flash cache and one without, by using the `alter system` command as shown here:

```
SQL> alter system set db_flash_cache_size = 0;       /* disables the flash cache
SQL> alter system set db_flash_cache_size = 8G;      /* reenables the flash cache
```

Note that although you can successfully enable and disable the flash cache dynamically as shown here, Oracle doesn't support this method.

■ **Note** If you're using Oracle RAC, in order to utilize the Flash Cache feature, you must enable it on all the nodes of the cluster.

How It Works

Oracle Database Smart Flash Cache, a feature of the Oracle Database 11.2 release, is included as part of the enterprise edition of the database server. Flash Cache takes advantage of the I/O speed of flash-

based devices, which perform much better than disk-based storage. For example, small disk-based reads offer a 4-millisecond response, whereas a flash-based device takes only 0.4 milliseconds to perform the same read.

Note that Flash Cache is really a read-only cache—when clean (unmodified) data blocks are evicted from the buffer cache due to space pressure, those blocks are then moved to the flash cache. If they're required later on, the database will move transferred data blocks back to the SGA from the flash cache. It's not always realistic to assume that you and the Oracle database will have access to unlimited memory. What if you can allocate only a maximum of 12 GB for your Oracle SGA, but it turns out that if you have 50 GB of memory, the database will run a whole lot faster? Oracle Database Smart Flash Cache is designed for those types of situations.

Oracle recommends that you size the flash cache to a value that's a multiple of your buffer cache size. There's no hard and fast rule here: use a trial and error method by just setting it to anywhere between one and ten times the size of the buffer cache size and calibrate the results. Oracle also suggests that if you encounter the db file sequential read wait event as a top wait event and if you have sufficient CPU capacity, you should consider using the flash cache.

Once you enable the flash cache, Oracle moves data blocks from the buffer cache to the flash cache (the file you've created) and saves metadata about the blocks in the database buffer cache. Depending on the number of blocks moved into the flash cache, you may want to bump up the size of the `MEMORY_TARGET` parameter so the accumulated metadata doesn't impact the amount of memory left for the other components.

Oracle offers two devices for flash cache storage—Sun Storage F5100 Flash Array and the Sun Flash Accelerator F20 PCIe Card. Since you can specify only a single flash device, you will need a volume manager. It turns out that Oracle ASM is the best volume manager for these devices, based on Oracle's tests.

If you're using the flash cache in an Oracle RAC environment, you must create a separate flash cache file path for each of the instances, and you will also need to create a separate ASM disk group for each instance's flash cache.

Oracle testing of the Database Smart Flash Cache feature shows that it's ideally suited for workloads that are I/O bound. If you have a very heavy amount of concurrent read-only transactions, the disk system could be saturated after some point. Oracle Database Smart Flash Cache increases such a system's throughput by processing more IOPS (I/O per second). Oracle's testing results of this feature also show that response times increased by five times when Smart Flash Cache was introduced to deal with workloads facing significant performance deterioration due to maxing out of their disk I/O throughput. As of the writing of this book, Oracle makes these claims only for workloads that are exclusively or mostly read-only operations. While Oracle is still in the process of testing the flash cache for write-intensive workloads, note that even for read-only operations, the reduced load on your disk system due to using the flash cache will mean that you'll have more I/O bandwidth to handle your writes.

3-13. Tuning the Redo Log Buffer

Problem

You'd like to know how to tune the redo log buffer, as you've reviewed several AWR reports that pointed out that the redo log buffer setting for your production database is too small.

Solution

You configure the size of the redo log buffer by setting the value of the initialization parameter LOG_BUFFER. This parameter is static, so any changes to it require a restart of the database. You set the parameter in your init.ora file as follows:

```
log_buffer=4096000
```

You can also change the size of the log buffer, with the following ALTER SYSTEM statement:

```
SQL> alter system set log_buffer=4096000 scope=spfile;

System altered
SQL>
```

The default value of the LOG_BUFFER parameter is way too small for many databases. Oracle states that you don't normally need to set a value larger than 1 MB for this parameter. However, you shouldn't hesitate to raise it to a much larger amount, if it's warranted by circumstances.

How It Works

When the Oracle server processes change data blocks in the buffer cache, those changes are written to the redo logs in the form of redo log entries, before they are written to disk. The redo log entries enable the database to redo or reconstruct the database changes by using various operations such as INSERT, UPDATE, and DELETE, as well as DDL operations. The Oracle redo logs are thus critical for any database recovery, because it's these redo log entries that enable the database to apply all the changes made to the database from a point in time in the past. The changed data doesn't directly go to the redo logs, however; Oracle first writes the changes to a memory area called the redo log buffer. It's the value of this memory buffer that you can configure with the LOG_BUFFER parameter. The Oracle log writer (LGWR) process writes the redo log buffer entries to the active redo log file (or group of files). LGWR flushes the contents of the buffer to disk whenever the buffer is one-third full, or if the database writer requests the LGWR to write to the redo log file. Also, upon each COMMIT or ROLLBACK by a server process, the LGWR process writes the contents of the buffer to the redo log file on disk.

The redo log buffer is a re-usable cache, so as entries are written out to the redo log file, user processes copy new entries into the redo log buffer. While the LGWR usually works fast enough so there's space in the buffer, a larger buffer will always have more room for new entries. Since there's no cost whatsoever to increasing the LOG_BUFFER size, feel free to set it to higher than the suggested maximum of 1 MB for this parameter.

If your database is processing large updates, the LGWR has to frequently flush the redo log buffer to the redo log files even in the absence of a COMMIT statement, so as to keep the buffer no more than a third full. Raising the size of the redo log buffer is an acceptable solution in this situation, and allows the LGWR to catch up with the heavy amount of entries into the redo log buffer. This also offsets a slow I/O system in some ways, if you think the performance of the LGWR process is not fast enough. There are a couple of ways in which you keep the pressure on the redo log buffer down: you can batch COMMIT operations for all batch jobs and also specify the NOLOGGING option where possible, say during regular data loads. When you specify the NOLOGGING option during a data load, Oracle doesn't need to use the redo log files, and hence it also bypasses the redo log buffer as well.

It's fairly easy to tune the size of the LOG_BUFFER parameter. Just execute the following statement to get the current "redo log space request ratio":

```
SQL> select round(t.value/s.value,5) "Redo Log Space Request Ratio"
     from v$sysstat s, v$sysstat t
     where s.name = 'redo log space requests'
     and t.name = 'redo entries'
```

The redo log space request ratio is simply the ratio of total redo log space requests to redo entries. You can also query the V$SYSSTAT view to find the value of the statistic `redo buffer allocation retries`. This statistic shows the number of times processes waited for space in the redo log buffer:

```
SQL> select name,value from V$SYSSTAT
     where name= 'redo buffer allocation retries';
```

Execute this SQL query multiple times over a period of time. If the value of the "redo buffer allocation retries" statistic is rising steadily over this period, it indicates that the redo log buffer is under space pressure and as a result, processes are waiting to write their redo log entries to the redo log buffer. You must increase the size of the redo log buffer if you continue to see this.

CHAPTER 4

Monitoring System Performance

Monitoring system and database performance is a complex task, and there can be many aspects to managing performance, including memory, disk, CPU, database objects, and database user sessions—just for starters. This chapter zeroes in on using Oracle's Automatic Workload Repository (AWR) to gather data about the activities occurring within your database, and help convert that raw data into useful information to help gauge performance within your database for a specific period of time. Usually, when there are performance issues occurring within a database, it's easy to know when the performance problems are occurring because database activity is "slow" during that given time frame. Knowing this time frame is the starting point to perform the analysis using the AWR information.

The AWR is created automatically when you create your Oracle database, and automatically gathers statistics on the activities occurring within your database. Some of this data is real-time or very near real-time, and some of the data represents historical statistics on these activities. The most current data on active sessions is stored in the Active Session History (ASH) component of the performance statistics repository. The more historical snapshots of data are generally known as the AWR snapshots.

The AWR process captures hourly snapshots by default from Oracle's dynamic performance views, and stores them within the AWR. This gives the DBA the ability to view database activity over any period of time, whether it is a single-hour time frame, up to all activity that is stored within the AWR. For instance, if you have a period of time where your database is performing poorly, you can generate an AWR report that will give statistics on the database activity for only that specific period of time.

The ASH component of the AWR is really meant to give the DBA a more real-time look at session information that is not captured within the AWR snapshots. The session information stored within the ASH repository is data that is sampled every second from the dynamic performance views.

Oracle has had similar information within the database for many years with its predecessors UTLBSTAT/UTLESTAT and Statspack, but the report data hasn't been generated or saved automatically until AWR came along with Oracle 10g. This information can now help monitor your database performance much more easily, whether it be analyzing real-time data on activities currently going on within your database, or historical information that could be days, weeks, or even months old, depending on the configuration of the AWR within your database.

4-1. Implementing Automatic Workload Repository (AWR)

Problem

You want to store historical database performance statistics on your database for tuning purposes.

Solution

By implementing and using the Automatic Workload Repository (AWR) within your database, Oracle will store interval-based historical statistics in your database for future reference. This information can be used to see what was going on within your database within a given period of time. By default, the AWR should be enabled within your database. The key initialization parameter to validate is the STATISTICS_LEVEL parameter:

```
SQL> show parameter statistics_level

NAME                                 TYPE        VALUE
------------------------------------ ----------- ------------------------------
statistics_level                     string      TYPICAL
```

This parameter can be set to BASIC, TYPICAL (which is the default), and ALL. As long as the parameter is set to TYPICAL or ALL, statistics will be gathered for the AWR. If the parameter is set to BASIC, you simply need to modify the parameter in order to start gathering AWR statistics for your database:

```
alter system set statistics_level=TYPICAL scope=both;
```

How It Works

The predecessor of AWR, which is Statspack, requires manual setup and configuration to enable the statistics gathering. As stated, there generally is no setup required, unless the STATISTICS_LEVEL parameter has been changed to the BASIC setting. By default, an AWR snapshot is taken every hour on your database, and is stored, by default, for eight days. These are configurable settings that can be modified, if desired. See Recipe 4-2 for information on modifying the default settings of the AWR snapshots.

In addition to simply seeing the value of the STATISTICS_LEVEL parameter, you can also view the V$STATISTICS_LEVEL view to see this information, which has information on the STATISTICS_LEVEL setting, as well as all other relevant statistical components within your database:

```
SELECT statistics_name, activation_level, system_status
FROM v$statistics_level;

STATISTICS_NAME                          ACTIVAT SYSTEM_S
---------------------------------------- ------- --------
Buffer Cache Advice                      TYPICAL ENABLED
MTTR Advice                              TYPICAL ENABLED
Timed Statistics                         TYPICAL ENABLED
Timed OS Statistics                      ALL     DISABLED
Segment Level Statistics                 TYPICAL ENABLED
PGA Advice                               TYPICAL ENABLED
Plan Execution Statistics                ALL     DISABLED
Shared Pool Advice                       TYPICAL ENABLED
Modification Monitoring                  TYPICAL ENABLED
Longops Statistics                       TYPICAL ENABLED
Bind Data Capture                        TYPICAL ENABLED
Ultrafast Latch Statistics               TYPICAL ENABLED
Threshold-based Alerts                   TYPICAL ENABLED
Global Cache Statistics                  TYPICAL ENABLED
Active Session History                   TYPICAL ENABLED
```

```
Undo Advisor, Alerts and Fast Ramp up    TYPICAL ENABLED
Streams Pool Advice                      TYPICAL ENABLED
Time Model Events                        TYPICAL ENABLED
Plan Execution Sampling                  TYPICAL ENABLED
Automated Maintenance Tasks              TYPICAL ENABLED
SQL Monitoring                           TYPICAL ENABLED
Adaptive Thresholds Enabled              TYPICAL ENABLED
V$IOSTAT_* statistics                    TYPICAL ENABLED
Session Wait Stack                       TYPICAL ENABLED

24 rows selected.
```

The type of information that is stored in the AWR includes the following:

- Statistics regarding object access and usage
- Time model statistics
- System statistics
- Session statistics
- SQL statements

The information gathered is then grouped and formatted by category. Some of the categories found on the report include the following:

- Instance efficiency
- Top 5 timed events
- Memory and CPU statistics
- Wait information
- SQL statement information
- Miscellaneous operating system and database statistics
- Database file and tablespace usage information

■ **Note** To use AWR functionality, the following must apply. First, you must be licensed for the Oracle Diagnostics Pack, otherwise you need to use Statspack. Second, the `CONTROL_MANAGEMENT_PACK_ACCESS` parameter must be set to `DIAGNOSTIC+TUNING` or `DIAGNOSTIC`.

4-2. Modifying the Statistics Interval and Retention Periods

Problem

You need to set an interval or retention period for your AWR snapshots to values other than the default.

Solution

By using the DBMS_WORKLOAD_REPOSITORY PL/SQL package, you can modify the default snapshot settings for your database. In order to first validate your current retention and interval settings for your AWR snapshots, run the following query:

```
SQL> column awr_snapshot_retention_period format a40
SQL> SELECT EXTRACT(day from retention) || ':' ||
       EXTRACT(hour from retention) || ':' ||
       EXTRACT (minute from retention)  awr_snapshot_retention_period,
       EXTRACT (day from snap_interval) *24*60+
       EXTRACT (hour from snap_interval) *60+
       EXTRACT (minute from snap_interval) awr_snapshot_interval
FROM dba_hist_wr_control;

AWR_SNAPSHOT_RETENTION_PERIOD  AWR_SNAPSHOT_INTERVAL
------------------------------ --------------------
8:13:45                                          60
```

The retention period output just shown is in day:hour:minute format. So, our current retention period is 8 days, 13 hours, and 45 minutes. The interval, or how often the AWR snapshots will be gathered, is 60 minutes in the foregoing example. To then modify the retention period and interval settings, you can use the MODIFY_SNAPSHOT_SETTINGS procedure of the DBMS_WORKLOAD_REPOSITORY package. To change these settings for your database, issue a command such as the following example, which modifies the retention period to 30 days (specified by number of minutes), and the snapshot interval at which snapshots are taken to 30 minutes. Of course, you can choose to simply set one parameter or the other, and do not have to change both settings. The following example shows both parameters simply for demonstration purposes:

```
SQL> exec DBMS_WORKLOAD_REPOSITORY.MODIFY_SNAPSHOT_SETTINGS(retention=>43200, interval=>30);

PL/SQL procedure successfully completed.
```

You can then simply rerun the query from the DBA_HIST_WR_CONTROL data dictionary view in order to validate that your change is now in effect:

```
SQL> /

AWR_SNAPSHOT_RETENTION_PERIOD            AWR_SNAPSHOT_INTERVAL
---------------------------------------- --------------------
30:0:0                                                     30
```

How It Works

It is generally a good idea to modify the default settings for your database, as eight days of retention is often not enough when diagnosing possible database issues or performing database tuning activities on your database. If you have been notified of a problem for a monthly process, for example, the last time frame that denoted an ordinary and successful execution of the process would no longer be available, unless snapshots were stored for the given interval. Because of this, it is a good idea to store a minimum of 45 days of snapshots, if at all possible, or even longer if storage is not an issue on your database. If you want your snapshots to be stored for an unlimited amount of time, you can specify a zero value, which tells Oracle to keep the snapshot information indefinitely (actually, for 40,150 days, or 110 years). See the following example:

```
SQL> exec DBMS_WORKLOAD_REPOSITORY.MODIFY_SNAPSHOT_SETTINGS(retention=>0);

PL/SQL procedure successfully completed.

SQL> /

AWR_SNAPSHOT_RETENTION_PERIOD            AWR_SNAPSHOT_INTERVAL
---------------------------------------- ---------------------
40150:0:0                                                   30
```

The default snapshot interval of one hour is usually granular enough for most databases, as when there are more frequent or closer to real-time needs, you can use the Active Session History (ASH) information. By increasing the default snapshot interval to greater than one hour, it can actually make it more difficult to diagnose performance issues, as statistics for the increased window may make it harder to distinguish and identify performance issues for a given time period.

4-3. Generating an AWR Report Manually

Problem

You want to generate an AWR report, and know the time frame on which to gather information.

Solution

In order to generate an AWR report, run the `awrrpt.sql` script found under the `$ORACLE_HOME/rdbms/admin` directory. In this example, we needed to enter information for the following:

- Report type (text or html)
- Number of days you want displayed from which to choose snapshots
- The starting snapshot number for the period on which you want to generate a report

- The ending snapshot number for the period on which you want to generate a report
- The name of the report (enter a name if you do not want the default report name and location)

The lines in bold type here denote points where user input is required:

```
$ sqlplus / as sysdba @awrrpt

Current Instance
~~~~~~~~~~~~~~~~
   DB Id    DB Name      Inst Num Instance
----------- ------------ -------- ------------
 2334201269 ORCL                1 ORCL

Specify the Report Type
~~~~~~~~~~~~~~~~~~~~~~~
Would you like an HTML report, or a plain text report?
Enter 'html' for an HTML report, or 'text' for plain text
Defaults to 'html'
Enter value for report_type: text

Type Specified:  text

Instances in this Workload Repository schema
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

   DB Id     Inst Num DB Name      Instance     Host
------------ -------- ------------ ------------ ------------
* 2334201269        1 ORCL         ORCL         ora

Using 2334201269 for database Id
Using          1 for instance number

Specify the number of days of snapshots to choose from
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Enter value for num_days: 7
Listing the last 7 days of Completed Snapshots

                                                        Snap
Instance     DB Name        Snap Id    Snap Started    Level
------------ ------------ --------- ------------------ -----
ORCL         ORCL               257 28 May 2011 00:00      2

                                258 28 May 2011 13:39      2
                                259 28 May 2011 15:00      2
                                260 28 May 2011 16:00      2
                                261 28 May 2011 17:00      2
                                262 28 May 2011 18:00      2
                                263 28 May 2011 19:00      2
                                264 28 May 2011 20:00      2
```

```
                                  265 28 May 2011 21:00      2
                                  266 28 May 2011 22:00      2
                                  267 28 May 2011 23:00      2
                                  268 29 May 2011 00:00      2

                                  269 29 May 2011 11:52      2
                                  270 29 May 2011 13:00      2
                                  271 29 May 2011 14:00      2
                                  272 29 May 2011 15:00      2
                                  273 29 May 2011 16:00      2
                                  274 29 May 2011 17:00      2

                                  275 30 May 2011 17:00      2
                                  276 30 May 2011 18:00      2
                                  277 30 May 2011 19:00      2
                                  278 30 May 2011 20:00      2

Specify the Begin and End Snapshot Ids
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Enter value for begin_snap: 258
Begin Snapshot Id specified: 258

Enter value for end_snap: 268
End   Snapshot Id specified: 268

Specify the Report Name
~~~~~~~~~~~~~~~~~~~~~~~

The default report file name is awrrpt_1_258_268.txt.  To use this name,
press <return> to continue, otherwise enter an alternative.

Enter value for report_name: /tmp/awrrpt_1_258_268.txt

Using the report name /tmp/awrrpt_1_258_268.txt

< Output of report is shown across the screen >

End of Report
Report written to /tmp/awrrpt_1_258_268.txt
```

How It Works

In the foregoing example, note that between some of the snapshots listed there is a blank line. Since we are getting information based on the dynamic performance views of the data dictionary, you cannot specify a snapshot period that spans bounces of the database instance, as all statistics in the dynamic performance views are lost when a database instance is shut down. Therefore, choose a snapshot period only for the life of an instance; otherwise you can experience the following error:

```
Enter value for begin_snap: 274
Begin Snapshot Id specified: 274

Enter value for end_snap: 275
End   Snapshot Id specified: 275

declare
*
ERROR at line 1:
ORA-20200: The instance was shutdown between snapshots 274 and 275
ORA-06512: at line 42
```

Although it is recommended to use the `awrrpt.sql` script to generate the desired AWR report, you can manually use the `AWR_REPORT_TEXT` or `AWR_REPORT_HTML` functions within the `DBMS_WORKLOAD_REPOSITORY` package to generate an AWR report, if needed. You need to also have your database's DBID and instance number handy as well when running either of these functions. See the following for an example:

```
SELECT  dbms_workload_repository.awr_report_text
        (l_dbid=>2334201269,l_inst_num=>1,l_bid=>258,l_eid=>268)
FROM dual;
```

4-4. Generating an AWR Report via Enterprise Manager

Problem

You want to generate an AWR report from within Enterprise Manager.

Solution

Within Enterprise Manager, depending on your version, the manner in which to generate an AWR report may differ. There is also generally more than one way to generate an AWR report. In Figure 4-1, this particular screen shows that you enter the beginning and ending snapshot ranges, and after you click the Generate Report button, an AWR HTML report will immediately be displayed within your browser window. A sample screen of the resulting AWR report is shown in Figure 4-2.

Figure 4-1. *Generating an AWR report within Enterprise Manager*

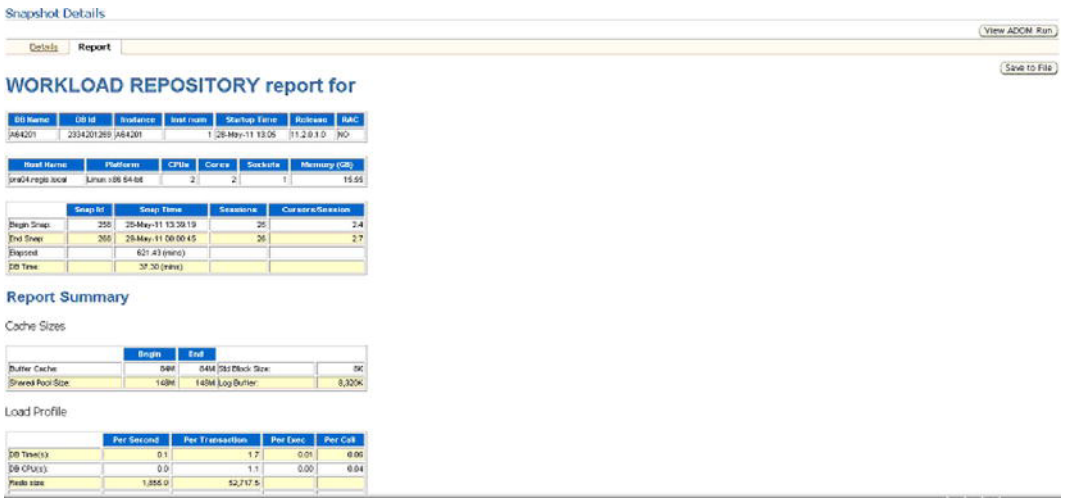

Figure 4-2. *HTML AWR report*

How It Works

The AWR report via Enterprise Manager can be generated if you have Database Control configured, or if you are using Grid Control. You need to be licensed for the Oracle Diagnostics Pack in order to be able to use this feature.

4-5. Generating an AWR Report for a Single SQL Statement

Problem

You want to see statistics for a single SQL statement, and do not want all other associated statistics generated from an AWR report.

Solution

You can run the `awrsqrpt.sql` script, which is very similar to `awrrpt.sql`. You will be prompted for all of the same information, except you will have an additional prompt for a specific SQL ID value—for example:

```
Specify the SQL Id
~~~~~~~~~~~~~~~~~~

Enter value for sql_id: 5z1b3z8rhutn6
SQL ID specified:  5z1b3z8rhutn6
```

The resulting report zeroes in on information specifically for your SQL statement, including CPU Time, Disk Reads, and Buffer Gets. It also gives a detailed execution plan for review. See the following snippet from the report:

```
Stat Name                                Statement   Per Execution % Snap
---------------------------------------- ---------- -------------- -------
Elapsed Time (ms)                           210,421      105,210.3     9.4
CPU Time (ms)                                22,285       11,142.3     1.6
Executions                                        2            N/A     N/A
Buffer Gets                               1,942,525      971,262.5    12.5
Disk Reads                                1,940,578      970,289.0    14.0
Parse Calls                                       9            4.5     0.0
Rows                                              0            0.0     N/A
User I/O Wait Time (ms)                     195,394            N/A     N/A
Cluster Wait Time (ms)                            0            N/A     N/A
Application Wait Time (ms)                        0            N/A     N/A
Concurrency Wait Time (ms)                        0            N/A     N/A
Invalidations                                     0            N/A     N/A
Version Count                                     2            N/A     N/A
Sharable Mem(KB)                                 22            N/A     N/A

Execution Plan
----------------------------------------------------------------------------------------------
| Id  | Operation            | Name     | Rows  | Bytes | Cost (%CPU)| Time     |PQ Dis
----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |          |       |       | 73425 (100)|          |
|   1 |  PX COORDINATOR      |          |       |       |            |          |
|   2 |   PX SEND QC (RANDOM)| :TQ10000 |     1 |    39 | 73425   (1)| 00:14:42 | P->S
|   3 |    PX BLOCK ITERATOR |          |     1 |    39 | 73425   (1)| 00:14:42 | PCWC
|   4 |     TABLE ACCESS FULL| EMPPART  |     1 |    39 | 73425   (1)| 00:14:42 | PCWP
----------------------------------------------------------------------------------------------

Full SQL Text

SQL ID       SQL Text
------------ ------------------------------------------------------------------
5z1b3z8rhutn /* SQL Analyze(98, 0) */ select * from emppart where empno > 12345
```

How It Works

Utilizing this feature is a handy way to get historical statistics for a given SQL statement. For current statements, you can continue to use other mechanisms such as `AUTOTRACE`, but after a SQL statement has been run, using the `awrsqrpt.sql` script provides an easy mechanism to help analyze past run statements and help retroactively tune poorly performing SQL statements.

4-6. Creating a Statistical Baseline for Your Database

Problem

You want to establish baseline statistics that represent a normal view of database operations.

Solution

You can create AWR baselines in order to establish a saved workload view for your database, which can be used later for comparison to other AWR snapshots. The purpose of a baseline is to establish a normal workload view of your database for a predefined time period. Performance statistics for an AWR baseline are saved in your database, and are not purged automatically. There are two types of baselines—fixed and moving.

Fixed Baselines

The most common type of baseline is called a fixed baseline. This is a single, static view that is meant to represent a normal system workload. To manually create an AWR baseline, you can use the `CREATE_BASELINE` procedure of the `DBMS_WORKLOAD_REPOSITORY` PL/SQL package. The following example illustrates how to create a baseline based on a known begin and end date and time for which the baseline will be created:

```
SQL> exec dbms_workload_repository.create_baseline -
    (to_date('2011-06-01:00:00:00','yyyy-mm-dd:hh24:mi:ss'), -
    to_date('2011-06-01:06:00:00','yyyy-mm-dd:hh24:mi:ss'),'Batch Baseline #1');

PL/SQL procedure successfully completed.
```

For the foregoing baseline, we want to establish a normal workload for a data warehouse batch window, which is between midnight and 6 a.m. This baseline will be held indefinitely unless explicitly dropped (see Recipe 4-7 for managing AWR baselines). Any fixed baseline you create stays in effect until a new baseline is created. If you want to have a set expiration for a baseline, you can simply specify the retention period for a baseline when creating it by using the `EXPIRATION` parameter, which is specified in days:

```
exec dbms_workload_repository.create_baseline( -
start_time=>to_date('2011-06-01:00:00:00','yyyy-mm-dd:hh24:mi:ss'), -
end_time=>to_date('2011-06-01:06:00:00','yyyy-mm-dd:hh24:mi:ss'), -
baseline_name=>'Batch Baseline #1', -
expiration=>30);
```

You can also create a baseline based on already created AWR snapshot IDs. In order to do this, you could run the `CREATE_BASELINE` procedure as follows:

```
exec dbms_workload_repository.create_baseline( -
start_snap_id=>258,end_snap_id=>268,baseline_name=>'Batch Baseline #1', -
expiration=>30);
```

Moving Baselines

Like the fixed baseline, the moving baseline is used to capture metrics over a period of time. The big difference is the metrics for moving baselines are captured based on the entire AWR retention period. For instance, the default AWR retention is eight days (see Recipe 4-2 on changing the AWR retention period). These metrics, also called adaptive thresholds, are captured based on the entire eight-day window. Furthermore, the baseline changes with each passing day, as the AWR window for a given database moves day by day. Because of this, the metrics over a given period of time can change as a database evolves and performance loads change over time. A default moving baseline is automatically created—the `SYSTEM_MOVING_BASELINE`. It is recommended to increase the default AWR retention period, as this may give a more complete set of metrics on which to accurately analyze performance. The maximum size of the moving window is the AWR retention period. To modify the moving window baseline, use the `MODIFY_BASELINE_WINDOW_SIZE` procedure of the `DBMS_WORKLOAD_REPOSITORY` package, as in the following example:

```
SQL>  exec dbms_workload_repository.modify_baseline_window_size(30);

PL/SQL procedure successfully completed.
```

How It Works

Setting the AWR retention period is probably the most important thing to configure when utilizing the moving baseline, as all adaptive threshold metrics are based on information from the entire retention period. When setting the retention period for the moving baseline, remember again that it cannot exceed the AWR retention period, else you may get the following error:

```
SQL> exec dbms_workload_repository.modify_baseline_window_size(45);
BEGIN dbms_workload_repository.modify_baseline_window_size(45); END;
*
ERROR at line 1:
ORA-13541: system moving window baseline size (3888000) greater than retention
(2592000)
ORA-06512: at "SYS.DBMS_WORKLOAD_REPOSITORY", line 686
ORA-06512: at line 1
```

If you set your AWR retention to an unlimited value, there still is an upper bound to the moving baseline retention period, and you could receive the following error if you set your moving baseline retention period too high, and your AWR retention period is set to unlimited:

```
exec dbms_workload_repository.modify_baseline_window_size(92);
BEGIN dbms_workload_repository.modify_baseline_window_size(92); END;
*
ERROR at line 1:
ORA-13539: invalid input for modify baseline window size (window_size, 92)
ORA-06512: at "SYS.DBMS_WORKLOAD_REPOSITORY", line 686
ORA-06512: at line 1
```

For fixed baselines, the AWR retention isn't a factor, and is a consideration only based on how far back in time you want to compare a snapshot to your baseline. After you have set up any baselines, you can get information on baselines from the data dictionary. To get information on the baselines in your database, you can use a query such as the following one, which would show you any fixed baselines you have configured, as well as the automatically configured moving baseline: