CHAPTER 8

Using and Managing Large Databases

CRITICAL SKILLS

8.1 Learn to Identify a Very Large Database
8.2 How and Why to Use Data Partitioning
8.3 Compress Your Data
8.4 Use Parallel Processing to Improve Performance
8.5 Use Materialized Views
8.6 Use SQL Aggregate and Analysis Functions
8.7 Create SQL Models
This chapter explores the topics and features available in Oracle Database 12c that you'll need to be familiar with when working with large databases. These features are among the more advanced that you will encounter, but they’re necessary because databases continue to grow larger and larger. When you start working with Oracle, you will find yourself facing the trials and tribulations associated with large databases sooner rather than later. The quicker you understand the features and know where and when to use them, the more effective you will be. Of course, these features are not just valuable for large databases; they provide value to everyone who is looking to optimize their Oracle database environment.

CRITICAL SKILL 8.1

Learn to Identify a Very Large Database

Let’s start by describing what we mean by a very large database (VLDB). Large is a relative term that changes over time. What was considered large five or ten years ago is small by today’s standards, and what is large today will be peanuts a few years from now. How many people remember buying a computer with 20MB of storage and wondering how they could ever fill it up? Today, personal computers may come with more than a terabyte of storage; with the types of information you store, you know that you will fill these storage systems to capacity sooner or later. Each release of Oracle has included new features and enhancements for addressing the need to store more and more data. For example, Oracle8i was released in 1999 and could handle databases with terabytes (1024 gigabytes) of data. In 2001, Oracle9i was released and could deal with up to 500 petabytes (1024 terabytes). Oracle Database 11g offered support for exabyte (1024 petabytes) databases. Now there are databases with exabytes, and starting to approach the zettabyte (1024 exabytes) range. Hardware is also playing a part in allowing the databases to increase to these sizes. Oracle is producing new high-performance hardware and software in the Exa-Machines—Exadata, Exalogic, and Exalytics, with more to come. The Exa-Machines will provide speed improvements and large amounts of storage as the importance of large databases continues to expand and evolve over time. You’ll see more on Exadata in the next chapter.

The most obvious examples of large database implementations are data warehouses and decision support systems. These environments usually have tables with millions or billions of rows or wide tables with large numbers of columns and many rows. There are also many OLTP systems that are very large and can benefit from the features you are about to explore. Because you have many topics to get through, let’s jump right in and start with data partitioning.
Critical Skill 8.2
How and Why to Use Data Partitioning

As user communities require more and more detailed information to remain competitive, it has fallen to database designers and administrators to help ensure that the information is managed effectively and can be retrieved for analysis efficiently. In this section, we discuss partitioning data and the reasons why it is so important when working with large databases. Afterward, you’ll follow the steps required to make it all work.

Why Use Data Partitioning

Let’s start by defining data partitioning. In its simplest form, it is a way of breaking up or subsetting data into smaller units that can be managed and accessed separately. It has been around for a long time, both as a design technique and as a technology. Let’s look at some of the issues that gave rise to the need for partitioning and the solutions to these issues.

Tables containing very large numbers of rows have always posed problems and challenges for DBAs, application developers, and end users alike. For the DBA, the problems are centered on the maintenance and manageability of the underlying data files that contain the data for these tables. For the application developers and end users, the issues are query performance and data availability.

To mitigate these issues, the standard database design technique was to create physically separate tables, identical in structure (for example, columns), but each containing a subset of the total data (this design technique will be referred to as non-partitioned here). These tables could be referred to directly or through a series of views. This technique solved some of the problems, but still meant maintenance for the DBA with regard to creating new tables and/or views as new subsets of data were acquired. In addition, if access to the entire dataset was required, a view was needed to join all subsets together.
Figure 8-1 illustrates a non-partitioned design. In this sample, separate tables with identical structures have been created to hold monthly sales information for 2005. Views have also been defined to group the monthly information into quarters using a union query. The quarterly views themselves are then grouped together into a view that represents the entire year. The same structures would be created for each year of data. In order to obtain data for a particular month or quarter, an end user would have to know which table or view to use.

Similar to the technique illustrated in Figure 8-1, the partitioning technology offered by Oracle Database 12c is a method of breaking up large amounts of data into smaller, more manageable chunks, with each of the partitions having their own unique name and their own storage definitions. But, like the non-partitioned technique, it is transparent to the end user, offering improved performance and reduced maintenance. Figure 8-2 illustrates the same SALES table, but implemented using Oracle Database 12c’s partitioning option. From the end user’s perspective, there is only one table called SALES and all that is required to access data from the correct partition is a date (or a month and year).

**FIGURE 8-1.** Partitioning using physically separate tables (non-partitioned)
Oracle partitioning was first introduced in Oracle 8, is only available with the Enterprise Edition, and is an additional option to the core database license. As previously suggested, it is one database option that is a must-have for anyone with a large volume of data that needs to be quickly retrievable or with a need for speedy data archiving. Many improvements have been made since then, and Oracle Database 12c contains all of the latest features and enhancements to maintain the objects. The remainder of this section discusses these features in more detail.

**FIGURE 8-2. Partitioning using Oracle 12c partitioning**

Oracle partitioning was first introduced in Oracle 8, is only available with the Enterprise Edition, and is an additional option to the core database license. As previously suggested, it is one database option that is a must-have for anyone with a large volume of data that needs to be quickly retrievable or with a need for speedy data archiving. Many improvements have been made since then, and Oracle Database 12c contains all of the latest features and enhancements to maintain the objects. The remainder of this section discusses these features in more detail.

**Manageability**

When administering large databases, DBAs are required to determine the most efficient and effective ways to configure the underlying data files that support the tables in the database. The decisions made at this time will affect your data accessibility and availability as well as backup and recovery.
Some of the benefits for database manageability when using partitioned tables include the following:

- Historical partitions can be made read-only and will not need to be backed up more than once. This also means faster backups. With partitions, you can move data to lower-cost storage by moving the tablespace, sending it to an archive via an export (datapump), or some other method.

- The structure of a partitioned table needs to be defined only once. As new subsets of data are acquired, they will be assigned to the correct partition, based on the partitioning method chosen. In addition, with Oracle 12c you have the ability to define intervals that allow you to define only the partitions that you need. It also allows Oracle to automatically add partitions based on data arriving in the database. This is an important feature for DBAs, who currently spend time manually adding partitions to their tables.

- Moving a partition can now be an online operation, and the global indexes are maintained and not marked unusable. ALTER TABLE…MOVE PARTITION allows DDL and DML to continue to run uninterrupted on the partition.

- Global index maintenance for the DROP and TRUNCATE PARTITION happens asynchronously so that there is no impact to the index availability.

- Individual tablespaces and/or their data files can be taken offline for maintenance or archiving without affecting access to other subsets of data. For example, assuming data for a table is partitioned by month (later in this chapter, you learn about the different types of partitioning) and only 13 months of data is to be kept online at any one time, the earliest month is archived and dropped from the table when a new month is acquired. This is accomplished using the command ALTER TABLE abc DROP PARTITION xyz and does not affect access to the remaining 12 months of data.

- Other commands that would normally apply at the table level can also be applied to a particular partition of the table. These include but are not limited to DELETE, INSERT, SELECT, TRUNCATE, and UPDATE. TRUNCATE and EXCHANGE PARTITION operations allow for cascading data maintenance for related tables. You should review the Oracle Database VLDB and Partitioning Guide for a complete list of the commands that are available with partitions and subpartitions.

**Performance**

One of the main reasons for partitioning a table is to improve I/O response time when selecting data from the table. Having a table’s data partitioned into subsets
can yield much faster query results when you are looking for data that is contained within one subset of the total. Let’s look at an illustrative example.

Assume the SALES table contains 100 million records representing daily sales revenue for the years 2008 to 2010 inclusive. You want to know what the total revenue is for February 2010. Your query might look something like this:

```sql
select sum(amount_sold)
from sales
where time_id between to_date('2008-02-01', 'YYYY-MM-DD')
and to_date('2010-02-28', 'YYYY-MM-DD');
```

With a non-partitioned table design, all 100 million rows would need to be scanned to determine if they belong to the date criteria. With a partitioned table design based on monthly partitions, with about 2.8 million rows for each month, only those rows in the February 2012 partition (and therefore only about 2.8 million rows) would be scanned. The process of eliminating data not belonging to the subset defined by the query criteria is referred to as partition pruning.

With the basic concepts of partitioning and an understanding of why you use it under your belt, you can now learn about the finer details of how to implement partitioning.

### Ask the Expert

**Q:** Can you use the analyze table command to gather statistics on partitioned tables?

**A:** No, at least not correctly. The supplied DBMS_STATS package should be used to gather statistics on partitioned tables instead. The analyze table command does not gather all required statistics for partitioned tables (in particular, global statistics). In addition, the analyze command will eventually be phased out (for all types of table and indexes), and only those statistics gathered by the DBMS_STATS package will be used by the cost-based optimizer. Oracle Database 12c provides an enhancement for using resources with DBMS_STATS. Using the parameter CONCURRENT set to TRUE, the statistics will be gathered on multiple partitions from separate statistics gathering jobs for each partition. Depending on the system resources and how many available jobs are in the queue, more jobs will be executed. Being able to run these jobs concurrently can reduce the time it takes to gather the statistics.
Implement Data Partitioning

Implementing data partitioning in Oracle Database 12c is a process that requires careful planning to ensure success. You will need to understand your database environment, hardware, structures, and data before you can make the appropriate decisions. The next few sections will outline the steps you will take when partitioning. Let’s start by looking at the characteristics of the candidate table.

Analyze the Candidate Table

The first step in the partitioning process is to analyze and understand the candidate table, its environment, and its uses. Following are some criteria to consider.

**Table Structure and Data Contents** You will need to look at the attributes that are available and the distribution of the data within each attribute. You must consider currently available data as well as projected future data. The distribution of data over each attribute is important because you want to ensure that the resulting data subsets are evenly distributed across the defined partitions.

Consider a table called PHONE_USAGE that contains detailed mobile phone call records with over 300 million records per month. It has many attributes, including the toll type (toll_type_cd) and the date of call (call_date). Table 8-1 shows a sample row count for a month by toll_type_cd. As you can see, using this attribute would probably not be an ideal choice for creating subsets because the distribution is heavily skewed toward LOCAL calls.

Table 8-2 looks at the distribution of the same data by the day of the week (for example, Sunday to Saturday based on call_date).

You can see that the day of the week provides a relatively even distribution that is more suitable for partitioning. Having a relatively equal data distribution of your data across the partitions will result in better performance during queries, as processing can be spread equally across the partitions. When a table is skewed to one partition, this can result in a very large data set within your table, which would defeat the purpose of partitioning the table in the first place.

<table>
<thead>
<tr>
<th>toll_type_cd</th>
<th>Record Count (Sample Month)</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTNL</td>
<td>27,296,802</td>
</tr>
<tr>
<td>CONTNL</td>
<td>52,227,998</td>
</tr>
<tr>
<td>LOCAL</td>
<td>189,554,584</td>
</tr>
<tr>
<td>NRTH AMRCA</td>
<td>36,367,841</td>
</tr>
</tbody>
</table>

**TABLE 8-1.  Toll Type and Record Counts**
To access the data, you will need to know what the most common data selection criteria are. This is perhaps the most important part of the analysis because, as stated earlier, query performance is the most noticeable gain of data partitioning. In order for this to be realized, your data subsets need to be defined according to the most common selection criteria so that unnecessary partitions can be pruned from the result set. The selection criteria will be determined largely by your user community and can be determined using historical query patterns (if available) or consulting business requirements.

To access the data, you will need to know what the most common data selection criteria are. This is perhaps the most important part of the analysis because, as stated earlier, query performance is the most noticeable gain of data partitioning. In order for this to be realized, your data subsets need to be defined according to the most common selection criteria so that unnecessary partitions can be pruned from the result set. The selection criteria will be determined largely by your user community and can be determined using historical query patterns (if available) or consulting business requirements.

Referring to the SALES table example, your analysis of query patterns for a three-month period (averaging 400 queries per month) yields the results shown in Table 8-3.

### Table 8-2. Counts Based on Day of the Week

<table>
<thead>
<tr>
<th>Day of the Week (Based on call_date)</th>
<th>Record Count (Sample Month)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUN</td>
<td>41,635,356</td>
</tr>
<tr>
<td>MON</td>
<td>44,235,019</td>
</tr>
<tr>
<td>TUE</td>
<td>42,875,502</td>
</tr>
<tr>
<td>WED</td>
<td>43,235,721</td>
</tr>
<tr>
<td>THU</td>
<td>43,922,997</td>
</tr>
<tr>
<td>FRI</td>
<td>45,005,293</td>
</tr>
<tr>
<td>SAT</td>
<td>44,537,337</td>
</tr>
</tbody>
</table>

### Table 8-3. Query Counts Based on Data Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Times Used in Query Selection Criteria (Average/Month)</th>
</tr>
</thead>
<tbody>
<tr>
<td>prod_id</td>
<td>33</td>
</tr>
<tr>
<td>cust_id</td>
<td>40</td>
</tr>
<tr>
<td>time_id</td>
<td>355</td>
</tr>
<tr>
<td>channel_id</td>
<td>55</td>
</tr>
<tr>
<td>promo_id</td>
<td>298</td>
</tr>
<tr>
<td>quantity_sold</td>
<td>25</td>
</tr>
<tr>
<td>amount_sold</td>
<td>20</td>
</tr>
</tbody>
</table>

Table 8-2. Counts Based on Day of the Week

Table 8-3. Query Counts Based on Data Attributes
The analysis tells you that time_id and promo_id are both frequently used as query predicates. You could use this information along with the corresponding row distribution to determine which attribute would result in the better partitioning strategy.

**Identify the Partition Key**

Once you understand the characteristics of your candidate table, the next step in the partitioning process is to select the attribute(s) of the candidate table that will define the partition subsets and how the subsets will be defined. The selected attributes will form the *partition key*. Only one set of attributes can be chosen to partition the data. This is an important decision because it will affect the manageability and usability of the table.

The results of your analysis of the candidate table should provide you with a good idea of the attributes to use. The best attributes will be those that satisfy the most criteria. Keep in mind, however, that the adage “you can satisfy some of the criteria some of the time, but you can’t satisfy all of the criteria all of the time” applies here. Despite your best efforts and planning, there will still be situations when the table will be treated as if it were non-partitioned. Take, for example, a perfectly valid query submitted by the user community that does not include the attributes of the partition key as part of the selection criteria or groups by the partition key, this would result in the query missing the core indexes and will impact performance by reading across all partitions. In this case, data from the entire table (that is, all partitions) would be scanned in order to satisfy the request.

**Select the Type of Partitioning**

After you have selected the partition key, the next step in the partitioning process is to decide which type of partitioning you want to implement on the candidate table. Partitioning options provide seven ways to partition data:

- Range partitioning
- List partitioning
- Hash partitioning
- Composite partitioning
- Reference partitioning
- Virtual column-based partitioning
- Interval partitioning

The type of partitioning you choose will depend on the results of your analysis of the candidate table. The most common type of partitioning is range partitioning so we cover it in detail. Let’s look at the characteristics of each type of partitioning.
Range partitioning has been around the longest of all partitioning types and is the type implemented most often. In most cases, the ranges are based on some date column, such as quarters, months, or, in the case of very large data volumes, days. (Assuming you have a time component, theoretically, you can go down to any level of time—hours, minutes, and so on. However, there are implications to defining this many partitions.) The ranges selected will again be based on the results of your analysis of the table, using dates, numeric values, or character values. Following is an example based on the SALES table you saw earlier in the chapter.

**NOTE**
The partitioning examples presented in this chapter do not address all of the command options available. They are meant to give you a taste of what is available.

To create your SALES table as non-partitioned, you would use the standard create table statement, as shown in this listing:

```sql
create table sales (
    prod_id number not null,
    cust_id number not null,
    time_id date not null,
    channel_id number not null,
    promo_id number not null,
    quantity_sold number (10,2) not null,
    amount_sold number (10,2) not null)
tablespace example_tblspc_1;
```

Based on the analysis of the usage patterns and row distribution, you have decided that the optimal partition strategy for this table is based on sales month. You will now redefine the SALES table using time_id as your partition key to create monthly partitions for January 2012 to December 2015, inclusive. Creation of data partitions is accomplished using extensions of the create table statement. The following listing shows the creation of the table with range partitions. Explanations of the important lines are given in Table 8-4.

```sql
1 create table sales {
2    prod_id number not null,
3    cust_id number not null,
4    time_id date not null,
5    channel_id number not null,
6    promo_id number not null,
7    quantity_sold number (10,2) not null,
8    amount_sold number (10,2) not null)
```
This defines the default table-level storage parameters that will apply to all partitions. It is possible to override these defaults at the partition level in favor of specific parameters required for a particular partition.

This defines the type of partitioning (for example, range) and the partition key (for instance, time_id).

Define each partition based on the values of time_id (repetitive lines for April 2012 to October 2014 omitted for brevity’s sake). For each partition, the upper boundary of the partition key value is specified (as defined by the function’s LESS THAN clause), as well as the name of the tablespace where the subset is to be stored. Values must be specified in ascending order and cannot overlap. It is good practice to give meaningful names to both the partitions and tablespaces.

Records that exceed the defined criteria will be placed into the final catchall partition, which uses the literal MAXVALUE. The MAXVALUE literal can be defined for the highest partition. It represents a virtual infinite value that sorts higher than any other possible value for the partitioning key, including the NULL value.

**TABLE 8-4. Explanation of Range Partitioning Syntax**
NOTE

Lines 11 to 13 define the first partition to hold data where time_id is less than February 1, 2012. The intention in this example is that this first partition will only hold data for January 2012 (our data analysis tells us that there is no data before this date). However, if there happens to be data prior to January 2012, it will also be placed in this partition and may skew the row distribution by placing many more rows than intended in this partition.

That completes the discussion on range partitioning. Let’s now have a look at list and hash partitioning.

There may be cases when, after your analysis of a candidate table, you decide that range partitioning is not the best fit for your table. Another way to subset your data is to use list partitioning, where you group a set of discrete partition key values. By using this type of partitioning, you can control the placement of the records in specified partitions, thereby allowing you to group related records together that may not otherwise have a relationship.

As an example, assume you have an INS_COVERAGE table that contains insurance coverage. Your analysis of this table and its usage leads you to decide that you should partition, based on the attribute COV_TYPE_CD, into the buckets shown in Table 8-5.

<table>
<thead>
<tr>
<th>COV_TYPE_CD</th>
<th>Grouping</th>
</tr>
</thead>
<tbody>
<tr>
<td>TERM 65</td>
<td>Life</td>
</tr>
<tr>
<td>UL</td>
<td>Life</td>
</tr>
<tr>
<td>ADB</td>
<td>Life</td>
</tr>
<tr>
<td>COLA</td>
<td>GIB</td>
</tr>
<tr>
<td>GPO</td>
<td>GIB</td>
</tr>
<tr>
<td>WP</td>
<td>Disability</td>
</tr>
<tr>
<td>DIS</td>
<td>Disability</td>
</tr>
</tbody>
</table>

Table 8-5. Insurance Coverage Groupings
Oracle Database 12c: Install, Configure & Maintain Like a Professional

The syntax of the CREATE TABLE statement is similar to that for range partitioning. Table 8-6 provides an explanation.

```
create table ins_coverage {
    plan_id number not null,
    cust_id number not null,
    time_id date not null,
    dist_channel_id number not null,
    cov_type_cd varchar2(50) not null,
    cov_amt number (10,2) not null,
    prem_amt number (10,2) not null,
    storage (initial 65536 minextents 1 maxextents 2147483645)
partition by list (cov_type_cd)
    (partition cov_life values ('TERM 65', 'UL', 'ADB')
        tablespace cov_life_ts,
    partition cov_gib values ('COLA', 'GIB')
        tablespace cov_gib_ts,
    partition cov_dis values ('WP', 'DIS')
        tablespace cov_dis_ts,
    partition cov_inv values ('MF')
        tablespace cov_inv_ts,
    partition cov_other values(default));
```

**TIP**

*If you discover missing partition key values that need to be added to existing partition definitions after the table has been created, you can issue an ALTER TABLE abc MODIFY PARTITION xyz ADD VALUES ('value1', ...).*

**TABLE 8-6. Explanation of List Partitioning Syntax**

<table>
<thead>
<tr>
<th>Lines</th>
<th>Important Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Defines the type of partitioning (for example, list) and the partition key (cov_type_cd, for instance). Note that with list partitioning, only one attribute from the table can be chosen as the partition key. In other words, multicoloumn partition keys are not permitted.</td>
</tr>
<tr>
<td>11–18</td>
<td>Defines each partition based on the groups of values of cov_type_cd.</td>
</tr>
</tbody>
</table>
If you determine from your table analysis that neither range nor list partitioning is appropriate for your table, but you still want to reap the benefits offered by partitioning, there is a third partitioning option called hash partitioning. With hash partitioning, you define up to 16 partition key attributes as well as the number of partitions you want to spread the data across. As long as each partition is on its own physical device and most of the queries use the partition key as a predicate, you should see performance gains. Hash partitioning is useful if the distribution of data is unknown or unpredictable.

The following listing is an example of hash partitioning. Table 8-7 explains the important lines:

```sql
create table sub_activations (  
sub_id number not null,  
dist_channel_id number not null,  
act_date date not null,  
deact_date date not null,  
sales_rep_id number not null)  
partition by hash (sub_id)  
partitions 4  
store in (subact_ts1, subact_ts2, subact_ts3, subact_ts4);
```

It is beyond the scope of this book to discuss the hashing algorithm used by Oracle Database. However, it is based on the number of attributes in the partition key and the number of partitions selected.

Another method of partitioning is a combination of two of the previous types. Combining two types of partitioning is called composite partitioning. There are a few basic combinations: range with hash, range with range, and range with list and others. Using composite partitioning allows you to take advantage of the features of either hash or list partitioning within the higher groupings of ranges.

### Lines Important Points

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Defines the type of partitioning (for instance, hash) and the partition key (for example, sub_id)</td>
</tr>
<tr>
<td>8</td>
<td>Specifies the number of partitions over which to spread the data</td>
</tr>
<tr>
<td>9</td>
<td>Specifies the tablespaces into which the partitions will be placed</td>
</tr>
</tbody>
</table>

**TABLE 8-7. Explanation of Hash Partitioning Syntax**
A good example of where this type of partitioning is used is the PHONE_USAGE table you saw in your candidate table analysis. In this case, you have a table that is being loaded with 300 million records per month. You could choose to implement range partitioning by month and then subdivide the monthly partitions into four hash partitions. The following listing shows the SQL syntax that accomplishes this, and Table 8-8 provides the explanation of the important lines:

```sql
create table phone_usage
    (sub_id number,
    call_date date,
    call_type_id number,
    called_location varchar2(50),
    service_carrier_id number)
    storage (initial 65536? minextents 1 maxextents 2147483645)
    partition by range (call_date)
    subpartition by hash(sub_id)
    subpartition template(
        subpartition sub1 tablespace ph_usg_ts1,
        subpartition sub2 tablespace ph_usg_ts2,
        subpartition sub3 tablespace ph_usg_ts3,
        subpartition sub4 tablespace ph_usg_ts4)
```

**Lines Important Points**

8. Defines the higher-level partitioning type (for example, range) and its partition key (for instance, call_date).

9. Specifies the secondary partitioning type (in this case, hash) and its partition key (here, sub_id).

10–14. Specifies a template that will be used to define the tablespace names for each subpartition, as well as the tablespace names. The name of each subpartition will be composed of the higher-level partition name concatenated with an underscore, then the subpartition name specified in the template. For example, data for January 2012 will be placed in tablespace ph_usg_ts1 and divided into four subpartitions called PHONEUSG_201201_SUB1, PHONEUSG_201201_SUB2, PHONEUSG_201201_SUB3, and PHONEUSG_201201_SUB4.

15–27. Specifies the ranges for the higher-level partition based on the call_date partition key.

**TABLE 8-8.** Explanation of Composite Partitioning Syntax
Chapter 8: Using and Managing Large Databases

Reference partitioning is one of the latest methods that Oracle has provided you for partitioning your data. The method used by reference partitioning allows you to partition data based upon referential constraints. As previously discussed, referential integrity (RI) in a database allows data to be correct and to be consistent. The idea is that you would not have any order line items without an order. RI is implemented to ensure data is correct, complete, and consistent. This leads you to the challenge of partitioning data in a productive manner, such that you may have consistent partitioning between parent and child tables. So, if you partition ORDERS by date range, you want the ORDER_ITEMS to follow the same method. Before its introduction in Oracle 11g, you could achieve this in a manual way, which may include adding additional attributes to child records (ORDER_ITEMS) to allow you to provide the same partition keys. With reference partitioning, you can now equi-partition parent and child tables without the need to duplicate keys. In addition, partition maintenance tasks will cascade from parent to child to reduce errors or omissions in the two tables.

**Ask the Expert**

**Q:** If the partition key of record in a partitioned table is updated and the new value means that the data belongs to a different partition, does Oracle Database automatically move the record to the appropriate partition?

**A:** Yes, but the table must have the ENABLE ROW MOVEMENT option before the update is made. This option is invoked as either part of the CREATE TABLE statement or using an ALTER TABLE statement. Otherwise, the UPDATE statement will generate an Oracle error.
Let’s look at an example of how you could set up a reference partitioned table. First, you should define the parent table:

```sql
CREATE TABLE orders
  (
    order_id NUMBER(12),
    order_date DATE,
    order_meth VARCHAR2(8),
    customer_id NUMBER(6),
    order_status NUMBER(2),
    order_tot NUMBER(8,2),
    sales_rep_id NUMBER(6),
    campaign_id NUMBER(6),
    CONSTRAINT orders_pk PRIMARY KEY(order_id)
  )
PARTITION BY RANGE(order_date)
  (PARTITION Q1_2012 VALUES LESS THAN (TO_DATE('01-APR-2012','DD-MON-YYYY')),
   PARTITION Q2_2012 VALUES LESS THAN (TO_DATE('01-JUL-2012','DD-MON-YYYY')),
   PARTITION Q3_2012 VALUES LESS THAN (TO_DATE('01-OCT-2012','DD-MON-YYYY')),
   PARTITION Q4_2012 VALUES LESS THAN (TO_DATE('01-JAN-2013','DD-MON-YYYY')),
   PARTITION UNCLASSIFIED_ORDER VALUES (DEFAULT)
  )
/
```

Now that you have the parent ORDERS table defined, you should take the time to notice that you defined a primary key constraint within the table definition. This is a requirement for the next step. Oracle reference partitioning depends solely on the definition of this integrity between entities, so consider this when deciding to utilize this method of partitioning. Now you can move on to the child table definition. In this case, you will define the table and then reference back to the parent table so that the partitioning is now based upon the parent’s partitioning methods, ultimately leveraging the database’s defined referential integrity:

```sql
CREATE TABLE order_items
  (
    order_id NUMBER(12) NOT NULL,
    line_item_id NUMBER(3) NOT NULL,
    product_id NUMBER(6) NOT NULL,
    unit_price NUMBER(8,2),
    quantity NUMBER(8),
    CONSTRAINT order_items_fk
      FOREIGN KEY(order_id) REFERENCES orders(order_id)
  )
PARTITION BY REFERENCE(order_items_fk)
/
```
As can see in the previous listing, this is a powerful method for partitioning data in a manner that logically follows the way that you store and read your data in a parent-child type relationship. Although you define only one partitioning method, this will reduce your workload because it really defines the method used by two or more tables, ultimately reducing the maintenance of your database. This method should be considered in operational and more relationship-oriented data sets.

Virtual columns are columns that are defined in metadata and are provided with the ability to create in-table derivations. This is valuable to many applications. To extend the virtual column idea to partitioning is the next logical extension of this functionality. Since Oracle Database 11g, Oracle has provided virtual column-based partitioning. By implementing this method of partitioning, you can use value derivations.

Before we discuss this method, we need to discuss the concept of the INTERVAL clause. The INTERVAL clause is used by Oracle to calculate the range of partitions. In our example, we will use a one-month interval, but this range can be set to other values. It should also be noted that in version 12c that Oracle has added the combination of INTERVAL and REF partitioning to extend it even further than simple INTERVAL partitioning. Let’s now look at an example of how a table is partitioned using the basic INTERVAL method:

```
CREATE TABLE sales
  ( prod_id NUMBER(6) NOT NULL
  , cust_id NUMBER NOT NULL
  , time_id DATE NOT NULL
  , channel_id CHAR(1) NOT NULL
  , promo_id NUMBER(6) NOT NULL
  , quantity_sold NUMBER(3) NOT NULL
  , amount_sold NUMBER(10,2) NOT NULL
  , total_amount AS (quantity_sold * amount_sold)
  )
PARTITION BY RANGE (time_id) INTERVAL (NUMTOYMINTERVAL(1,'MONTH'))
SUBPARTITION BY RANGE (total_amount)
SUBPARTITION TEMPLATE
  ( SUBPARTITION p_small VALUES LESS THAN (1000)
  , SUBPARTITION p_medium VALUES LESS THAN (5000)
  , SUBPARTITION p_large VALUES LESS THAN (10000)
  , SUBPARTITION p_extreme VALUES LESS THAN (MAXVALUE)
  )
(PARTITION sales_before_2007 VALUES LESS THAN
  (TO_DATE('01-JAN-2007','dd-MON-yyyy'))
ENABLE ROW MOVEMENT
PARALLEL NOLOGGING;
```
The previous example provides you with a few partitioning features. The first is the use of the virtual column partitioning. As you can see, the virtual column total amount is the column that will be used in the subpartitions. Subpartitions are used in secondary partitioning. This is known as composite partitioning, or the combining of two types of partitioning providing two layers of partitioned data. In this example, you are combining columns using the partitioning methods of range and date or range and the virtual column to give you some guidance on how to partition data correctly. The power of virtual columns is focused on providing complete control over how and where data is stored to optimize performance.

Define the Partitioned Indexing Strategy
Okay, so now you have decided how you are going to partition your data. To really get the most out of partitioning, you will need to look at some indexing strategies. There are two types of indexes applicable to partitioned tables: local and global. Let’s take a brief look at each.

Local partitioned indexes are indexes that are partitioned in the exact same manner as the data in their associated table—that is, they have a direct one-to-one relationship with the data partitions and use the same partition key. This association is illustrated in Figure 8-3; as you can see, each partition has its own associated “local” index. This drawing is based on Figure 8-2, which you saw at the beginning of the chapter. It shows how the data and indexes for each monthly subset are related and then joins using the partitions with the local index.

Because of this relationship, the following points apply to local indexes:

- You cannot explicitly add or drop a partition to/from a local index. The index automatically adds or drops index partitions when related data partitions are added or dropped. Consider Figure 8-3: If you dropped the data partition for January 2005, the corresponding index partition would automatically be dropped as well. Likewise, if you added a new data partition for January 2012, a new index partition would automatically be created.

- One of the advantages of partitioning data is to allow access to other subsets while maintenance is being carried out on another partition. Because local index partitions are in line with the data partitions, this advantage still exists.

- Local partitioned indexes require less maintenance than global indexes, as you will see later in this chapter.
The SQL syntax for creating a local index is presented in the next listing, which refers to the SALES table you created earlier in the chapter. Using RANGE partitioning we can now add a local index. Table 8-9 contains an explanation of the syntax:

```
1 create index sales_idx_l1 on sales (time_id)
2 local
3 (partition sales_idx_200501 tablespace sales_ts_idx_200501,
4 partition sales_idx_200502 tablespace sales_ts_idx_200502,
5 partition sales_idx_200503 tablespace sales_ts_idx_200503,
...
37 partition sales_idx_200711 tablespace sales_ts_idx_200711,
38 partition sales_idx_200712 tablespace sales_ts_idx_200712,
39 partition sales_idx_max tablespace sales_ts_idx_max);
```
### Lines Important Points

2. Specifies that the index is to be local. This line alone tells you that the index is to be partitioned along the same ranges as the data.

3–39. Defines the partition names and tablespaces for each partition. These lines are optional, but without them, the partition names will be assigned by the database.

**TABLE 8-9. Syntax Highlights**

### Ask the Expert

**Q:** After a table and its local indexes have been defined using range partitioning with a default maxvalue partition, how can you add more partitions as new subsets of data are received?

**A:** Use an ALTER TABLE statement to split the default data partitions, adding your new partition ranges. For example, to add a data partition for January 2008 data in the SALES table in the previous listing, issue the following command:

```sql
alter table sales
   split partition sales_max at
   (to_date('2008-02-01','YYYY-MM-DD'))
   into (partition sales_200801 tablespace sales_ts_200801,
         partition sales_max tablespace sales_ts_max);
```

This ALTER TABLE command will split the default index partition for sales_idx_l1. However, it will use the data partition names (for example, sales_200801) and tablespaces (sales_ts_200801, for instance); remember in the local index example you explicitly specified the partition names and tablespaces for the index. Therefore, the partition names and tablespaces will need to be adjusted using ALTER INDEX commands, as follows:

```sql
alter index sales_idx_l1
    rename partition sales_200801 to sales_idx_200801;

alter index sales_idx_l1
    rebuild partition sales_idx_200801 tablespace
    sales_ts_idx_200801;

alter index sales_idx_l1
    rebuild partition sales_idx_max tablespace sales_ts_idx_max;
```
Some other points about local partitioned indexes:

- They can be unique, but only if the data partition key is part of the index key attributes.
- Bitmap indexes on partitioned tables must be local.
- Subpartitioned indexes are always local.
- They are best suited for data warehouses and decision support systems.
- Local unique indexes also work well in OLTP environments.

Global partitioned indexes are indexes that are not directly associated with the data partitions. Instead, their partitions are defined independently, with the partition key sometimes different from the data partition key. This association is illustrated in Figure 8-4. (This figure is again based on Figure 8-2, shown at the beginning of the chapter.) It shows that the data is partitioned by monthly ranges, with a global index partitioned by product.

One advantage of global indexes is that if partition pruning cannot occur for the data partitions because of the predicates of a query, index partition pruning may still be possible with the global partition index. Global partitioned indexes are available as either range-based or hash-based. When using range-based global indexes, you must specify a default partition with maxvalue. Let’s look at an example for creating a global partitioned index and then discuss how it would be used.

Referring again to the PHONE_USAGE table you created earlier in the chapter, you should now create a global index on the call_type_id. The following listing is the SQL for this, with the explanation presented in Table 8-10:

```sql
1 create index phone_usg_idx_g1 on phone_usage (call_type_id)
2 global
3 partition by range (call_type_id)
4 (partition ph_usg_idx_g1 values less than (2)
5   tablespace ph_usg_ts_idx_1,
6 partition ph_usg_idx_g2 values less than (3)
7   tablespace ph_usg_ts_idx_2,
8 partition ph_usg_idx_g3 values less than (4)
9   tablespace ph_usg_ts_idx_3,
10 partition ph_usg_idx_g4 values less than (5)
11   tablespace ph_usg_ts_idx_4,
12 partition ph_usg_idx_gmax values less than (maxvalue)
13   tablespace ph_usg_ts_idx_max);
```
Now, assume the following query is executed against the PHONE_USAGE table:

```
select count(*)
from phone_usage
where call_type_id = 3;
```

Without the global index you just defined, no partition pruning would occur because the query predicate does not refer to the data partition key call_date. But, with the global index, only the index entries from the partition ph_usg_idx_g3.
would be scanned and therefore only data records related to those entries would be used in the result set.

Some other points on global partitioned indexes:

- Before 12c, globally partitioned indexes required more maintenance than local indexes, especially when you drop data partitions, because the indexes would become invalid and required rebuilding before they are available for use. Now the indexes are maintained at the same time as the DROP or TRUNCATE PARTITION command and no longer become invalid.

- They can be unique.

- They cannot be bitmap indexes.

- They are best suited for OLTP systems for direct access to specific records.

- Partial indexes are now available in 12c. These are indexes on a subset of the partitions of a table, which provides more flexibility for the globally partitioned indexes.

In your travels through the world of partitioning, you will hear the terms prefixed and nonprefixed partition indexes. These terms apply to both local and global indexes. An index is prefixed when the leftmost column of the index key is the same as the leftmost column of the index partition key. If the columns are not the same, the index is non-prefixed. That’s all well and good, but what effect does it have?

It is a matter of performance: Non-prefixed indexes cost more, from a query perspective, than prefixed indexes. When a query is submitted against a partitioned table and the predicate(s) of the query include(s) the index keys of a prefixed index, pruning of the index partition can occur. If the same index is non-prefixed instead, all index partitions may need to be scanned. (Scanning of all index partitions will depend on the predicate in the query and the type of index: global or local. If the data partition key is included as a predicate and the index is local, then the index partitions to be scanned will be based on pruned data partitions.)
Project 8-1 Create a Range-Partitioned Table and a Local-Partitioned Index

Data and index partitioning are an important part in maintaining large databases. We have discussed the reasons for partitioning and shown the steps to implement it. In this project, you will create a range-partitioned table and a related local-partitioned index.

Step by Step

1. Create two tablespaces, called inv_ts_2007q1 and inv_2007q2, using the following SQL statements. These will be used to store data partitions:

   ```sql
   create tablespace inv_ts_2007q1
       datafile 'inv_ts_2007q1_1.dat' size 10m;
   create tablespace inv_ts_2007q2
       datafile 'inv_ts_2007q2_1.dat' size 10m;
   ```

2. Create two tablespaces, called inv_idx_ts_2007q1 and inv_idx_2007q2, using the following SQL statements. These will be used to store index partitions:

   ```sql
   create tablespace inv_idx_ts_2007q1
       datafile 'inv_idx_ts_2007q1_f1.dat' size 10m;
   create tablespace inv_idx_ts_2007q2
       datafile 'inv_idx_ts_2007q2_f1.dat' size 10m;
   ```

3. Create a partitioned table called INVOICE with column names and types based on the following information:

   a. Define the table with the columns identified in Table 8-11.

   b. Use order_date as the partition key and then subset the data into the first and second calendar quarters 2007.

   ```sql
   Column Name       Data Type
   invoice_id        NUMBER
   customer_id       NUMBER
   order_date        DATE
   ship_date         DATE
   ```

   TABLE 8-11. INVOICE Table Columns
c. Define the table with the data partitions and tablespaces identified in Table 8-12.

d. Use the enable row movement option:

```sql
create table invoice (
    invoice_id number,
    customer_id number,
    order_date date,
    ship_date date
) partition by range (order_date)
(partition INV_2007Q1 values less than
(to_date(2007-04-01,'YYYY-MM-DD'))
    tablespace inv_ts_2007Q1,
    partition INV_2007Q2 values less than
(to_date('2007-07-01','YYYY-MM-DD'))
    tablespace inv_ts_2007Q2,
    partition inv_max values less than (maxvalue)
    tablespace inv_ts_max)
enable row movement;
```

4. Create a local partitioned index called inv_order_dt_idx on order_date using the following listing as well as the index partitions and tablespaces identified in Table 8-12.

```sql
create index inv_order_dt_idx on invoice(order_date)
local
(partition inv_idx_2007q1 tablespace inv_idx_ts_2007q1,
    partition inv_idx_2007q2 tablespace inv_idx_ts_2007q2,
    partition inv_idx_max tablespace inv_idx_ts_max);
```

---

<table>
<thead>
<tr>
<th>Partition Name</th>
<th>Tablespace Name</th>
<th>Upper Range Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Partitions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INV_2007Q1</td>
<td>INV_TS_2007Q1</td>
<td>Apr 1, 2007</td>
</tr>
<tr>
<td>INV_2007Q2</td>
<td>INV_TS_2007Q2</td>
<td>July 1, 2007</td>
</tr>
<tr>
<td>INV_MAX</td>
<td>INV_TS_MAX</td>
<td>MAXVALUE</td>
</tr>
<tr>
<td>Index Partitions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INV_IDX_2007Q1</td>
<td>INV_IDX_TS_2007Q1</td>
<td>Apr 1, 2007</td>
</tr>
<tr>
<td>INV_IDX_2007Q2</td>
<td>INV_IDX_TS_2007Q2</td>
<td>July 1, 2007</td>
</tr>
<tr>
<td>INV_IDX_MAX</td>
<td>INV_IDX_TS_MAX</td>
<td>MAXVALUE</td>
</tr>
</tbody>
</table>

**TABLE 8-12.** *INVOICE Table Data and Index Partitions*
Project Summary

The steps in this project reinforce some of the more common scenarios you will encounter: range-based partitioning and prefixed local partitioned indexes. Separate tablespaces were used for data and indexes, quarterly partitions were defined, a local index was defined, and the enable row movement was used to allow the database to automatically redistribute rows to their related partitions in the event of an update to the partition key.

Well, you have certainly covered a lot in this section. Having the background information on these topics will serve you well when maintaining and tuning large databases. Before you move on to the next section, let’s take a quick progress check to make sure it all sank in.

Progress Check

1. List at least three DML commands that can be applied to partitions as well as tables.
2. What does partition pruning mean?
3. How many table attributes can be used to define the partition key in list partitioning?
4. Which type of partitioning is most commonly used with a date-based partition key?
5. Which partitioning types cannot be combined together for composite partitioning?
6. How many partition keys can be defined for a partitioned table?
7. Which type of partitioned index has a one-to-one relationship between the data and index partitions?
8. What is meant by a prefixed partitioned index?

Progress Check Answers

1. The following DML commands can be applied to partitions as well as tables: delete, insert, select, truncate, and update.
2. Partition pruning is the process of eliminating data not belonging to the subset defined by the criteria of a query.
3. Only one table attribute can be used to define the partition key in list partitioning.
4. Range partitioning is most commonly used with a date-based partition key.
5. List and hash partitioning cannot be combined for composite partitioning.
6. Only one partition key may be defined.
CRITICAL SKILL 8.3

Compress Your Data

As you load more and more data into your database, performance and storage maintenance can quickly become concerns. Usually at the start of an implementation of a database, data volumes are estimated and projected a year or two ahead. However, oftentimes these estimates turn out to be on the low side and you find yourself scrambling for more space in order to load new data. In addition to the partitioning abilities discussed in the previous section, there is the ability to compress your data and indexes to further address the concerns of performance and maintenance.

Compression can be performed at the data or index levels. The tablespace can also be set up for compression, which means the objects in the tablespace will be compressed depending on the method. In this section, you learn about the options available and the impact these options have on your database.

Data Compression

With data compression, duplicate values in a database block are removed, leaving only a reference to the removed value, which is placed at the beginning of the block. All of the information required to rebuild the data in a block is contained within the block.

By compressing data, the physical disk space required is reduced; disk I/O and memory usage are also reduced, thereby improving performance. However, there are some cases when data compression is not appropriate. The following should be considered when looking at whether or not to compress data:

- Does the table exist in an OLTP or data warehousing environment? Data compression is best suited for data that is updated infrequently or, better yet, is read-only. Because most data in a data warehouse is considered read-only, data compression is more compatible with this type of environment.
- Does the table have many foreign keys? Foreign keys result in a lot of duplicate values in data. Tables with these structures are ideal candidates for data compression.
- How will data be loaded into the table? The loading method will determine the compression method. With basic compression, data is only compressed during bulk loading (for example, SQL*Loader), even when compression is enabled. To compress data when using a standard INSERT INTO statement, OLTP compression should be used.

Compression can be specified for various data-related objects using the CREATE or ALTER object commands. Table 8-13 identifies these objects and their first-level
Object Type | Compression Property Inheritance | Parent
---|---|---
Table | Tablespace | Table
Materialized View | Tablespace | Table
Partition | | 

**TABLE 8-13. Compression Property Inheritance**

parent object, from which default compression properties are inherited if not specified for the base object. For example, if no compression property is specified for a table, it will inherit the property from its tablespace. The same applies to a data partition—if not specified at the partition level, the default property from the table will be used.

The following listing demonstrates the creation of a table with compression enabled. Line 7 contains the keyword COMPRESS to tell Oracle that data compression is to be enabled:

```sql
1 create table commission (2 sales_rep_id number,
3 prod_id number,
4 comm_date date,
5 comm_amt number(10,2))
6 tablespace comm_ts
7 compress;
```

**Ask the Expert**

**Q:** Can existing data in a table be compressed and uncompressed?

**A:** Yes. There are two methods. The first is to use an alter table statement such as

```sql
alter table sales move compress;
```

To uncompress:

```sql
alter table sales move nocompress;
```

The second method is to use the utilities contained in the dbms_redefinition package.
COMPRESS in the previous code can be exchanged for different compression methods. COMPRESS FOR OLTP, COMPRESS FOR QUERY, and COMPRESS FOR ARCHIVE will compress data for both direct-path and DML, such as inserts and updates. The difference is in the level of compression, and with a higher level of compression, the CPU consumption will also be higher.

Because compression can be enabled or disabled at different points in an object’s lifetime (say, by using an alter command), and because the compression action occurs only on new data being loaded, it is possible for an object to contain both compressed and uncompressed data at the same time.

**Index Key Compression**

Index key compression works in a similar manner to data compression in that duplicated values are removed from the index entries. It’s a little more complicated, however, because it has more restrictions and considerations than data compression, partly due to the way indexes are structured. Because the details of these structures are beyond the scope of this book, you will focus on the benefits of, and the mechanisms for, defining index compression.

Compressing indexes offers the same benefits as data compression—that is, reduced storage and improved (usually) performance. However, performance may suffer during index scans as the burden on the CPU is increased in order to rebuild the key values. One restriction we should mention is that index compression cannot be used on a unique index that has only one attribute.

Enabling index compression is done using the create index statement. If you need to compress or uncompress an existing index, you must drop the index first and then re-create it with or without the compression option enabled. The following listing illustrates the syntax for creating a compressed index. Table 8-14 provides an explanation of the syntax.

```sql
1 create index comm_sr_prod_idx
2 on commission (sales_rep_id, prod_id)
3 compress 1;
```

Using data and index compression can provide substantial benefits in the areas of storage and performance.

**Automatic Data Optimization**

Compression, partitioning, and archiving are all part of managing the data and Information Lifecycle Management (ILM). With VLDB, it is important to manage the underlying data such that it allows what is most important to be accessed most efficiently first while still having older and not as frequently accessed data available. Different levels of compression are good for this to set older data at a high level,
where frequently accessed data might not be compressed or at a lower level. Managing this level of ILM might be a difficult process if there wasn’t some way to automate it.

In Oracle 12c, the Automatic Data Optimization (ADO) manages ILM by providing a way to specify policies at the table and row level. These policies can be set up to compress data at different levels of compression and to move data to different tiers of storage within the database.

The PL/SQL procedure, EXECUTE_ILM, provides a way to enforce the ADO policies immediately and outside of any scheduled optimization policies. Along with the procedures, there are tables to verify the policies and results of the ILM executes. The following example provides a high-level look at this:

```sql
-- Enable tracking for the activity statistics
> alter table emp_ilm2 ilm enable activity tracking segment access;
Table altered.
-- Add policy for ILM compression
>alter table emp_ilm2 ilm add emp_ilm2_com_pol compress for OLTP segment after 15 days of no modification;
Table altered.
--View the policy added in the DBA_ILMDATAMOVEMENTPOLICIES
select policy_name, action_type, compression_level, condition_type, condition_days
from dba_ilmdatamovementpolicies
order by policy_name;

POLICY_NAME ACTION_TYPE COMPRESSION_LEVEL CONDITION_TYPE CONDITION_DAYS
EMP_ILM2_POL COMPRESSION OLTP LAST MODIFICATION TIME 15
--Execute immediately a policy using DBMS_ILM.EXECUTE_ILM
declare
v_executionid number;
>Begin
```

### TABLE 8-14. Explanation of Index Compression Syntax

- **Lines 1–2**: This specifies that the index is to be created on columns sales_rep_id and prod_id.
- **Line 3**: This specifies that the index is to be compressed, with the number of prefixing (leading) columns to compress. In this case, you used a value of 1 to indicate that duplicate values of the first column, sales_rep_id, are to be removed.
dbms_ilm.execute_ILM (ilm_scope => dbms_ilm.scope_database,
execution_mode => dbms_ilm.ilm_execution_offline,
execution_id => v_executionid);
end;

With this very brief example, we hope you realize that these policies can help automate the management of the data and information. Oracle Database 12c features for ILM to manage compression and archiving are extremely helpful in VLDB environments. And this will help with I/O and accessing the data efficiencies. In the next section, you see how to improve query performance using parallel processing options.

**CRITICAL SKILL 8.4**

**Use Parallel Processing to Improve Performance**

Improving performance (and by this we usually mean query performance) is always a hot item with database administrators and users. One of the best and easiest ways to boost performance is to take advantage of the parallel processing option offered by Oracle Database 12c (Enterprise Edition only).

Using normal (that is, serial) processing, the data involved in a single request (for example, user query) is handled by one database process. Using parallel processing, the request is broken down into multiple units to be worked on by multiple database processes. Each process looks at only a portion of the total data for the request. Serial and parallel processing are illustrated in Figures 8-5 and 8-6, respectively.

Parallel processing can help improve performance in situations where large amounts of data need to be examined or processed, such as scanning large tables, joining large tables, creating large indexes, and scanning partitioned indexes. In order to realize the benefits of parallel processing, your database environment should not already be running at, or near, capacity. Parallel processing requires

---

**FIGURE 8-5.** Oracle serial processing
more processing, memory, and I/O resources than serial processing. Before implementing parallel processing, you may need to add hardware resources. Let’s forge ahead by looking at the components involved in parallel processing.

**Parallel Processing Database Components**

The parallel processing components are the *parallel execution coordinator* and the *parallel execution servers*. The parallel execution coordinator is responsible for breaking down the request into as many processes as specified by the request. Each process is passed to a parallel execution server for execution during which only a portion of the total data is worked on. The coordinator then assembles the results from each server and presents the complete results to the requester.

**Parallel Processing Configuration**

Generally, not much configuration is required for the database to perform parallel processing. There are, however, a number of configuration options that are required and will affect the effectiveness of parallelism.

---

**FIGURE 8-6. Oracle parallel processing**
To begin with, parallel processing is enabled by default for DDL (for example, create and alter) and query (for example, select) commands, but disabled for DML (say, insert, update, delete and merge) commands. If you wish to execute a DML command in parallel mode, you must first issue the following command for the session in which the command is to be executed:

```
alter session enable parallel dml;
```

Parallel processing can also be disabled for queries, DML, and DDL:

```
alter session disable parallel ddl;
```

It can also be forced with a degree of parallelism:

```
alter session force parallel ddl parallel 5;
```

Several database initialization parameters affect parallel processing. When an Oracle instance starts, the parameters in the initialization file are used to define or specify the settings for the instance. Table 8-15 identifies the initialization parameters that affect parallel processing. In many cases, the default values will provide adequate results for your large database. Specifics of your own environment will influence your decisions on the best values to use.

As you can see from Table 8-15, there are dependencies between parameters. Modifying one may necessitate modifying others. If you modify any of the parallel processing parameters, you may also have to modify the following database/instance parameters:

- INSTANCE GROUPS
- PROCESSES
- SESSIONS
- TRANSACTIONS

### Invoke Parallel Execution

Parallel execution can be applied to tables, views, and materialized views. If all necessary configurations have been made, there are several ways to invoke parallel execution. The first way is during table creation (including materialized views), using the parallel clause. If the table is being created using the results of a subquery, the loading of the table will be parallelized. In addition, by default, all queries that
are executed against the table will be parallelized to the same extent. The next listing shows an example of specifying the parallel option for a table creation:

```
1 create table commission (  
2   sales_rep_id   number,  
3   prod_id   number,  
4   comm_date   date,  
5   comm_amt   number(10,2)  
6   tablespace comm_ts  
7   parallel;
```

The important line here is Line 7, specifying the parallel clause. This line could also have included an integer to specify the degree of parallelism—that is, the number of processes that are to be used to execute the parallel process. As the degree of
parallelism is omitted in this example, the number of processes used will be calculated as the number of CPUs × the value of the PARALLEL_THREADS_PER_CPU initialization parameter. The degree of parallelism for a table or materialized view can be changed using an alter statement.

Parallel processing can also be invoked when the parallel hint is used in a select statement. This hint will override any default parallel processing options specified during table creation. The following listing illustrates the use of the parallel hint.

Line 1 contains the parallel hint, specifying the table to be parallelized (commission) and the degree of parallelism (4):

```sql
1 select /*+ parallel (commission, 4) */
2 prod_id, sum(comm_amt), count(*)
3 from commission
4 group by prod_id;
```

In some cases, Oracle Database 12c will alter how, or if, parallel processing is executed. Examples of these include the following:

- Parallel processing will be disabled for DML commands (for example, insert, update, delete, and merge) on tables with triggers or referential integrity constraints.

- If a table has a bitmap index, DML commands are always executed using serial processing if the table is non-partitioned. If the table is partitioned, parallel processing will occur, but Oracle will limit the degree of parallelism to the number of partitions affected by the command.

Parallel processing can have a significant positive impact on performance. Impacts on performance are even greater when you combine range- or hash-based partitioning with parallel processing. With this configuration, each parallel process can act on a particular partition. For example, if you had a table partitioned by month, the parallel execution coordinator could divide the work up according to those partitions. This way, partitioning and parallelism work together to provide results even faster.

**CRITICAL SKILL 8.5**

**Use Materialized Views**

So far, you have reviewed several features and techniques that you can use to improve performance in large databases. In this section, we discuss another feature of the Oracle Database that you can include in your arsenal: materialized views.
Originally called snapshots, materialized views were introduced in Oracle8i and are only available in the Enterprise Edition. Like a regular view, the data in a materialized view results from a query. However, the results of a regular view are transitory—they are lost once the query is complete and, if needed again, the query must be re-executed. In contrast, the results from a materialized view are kept and physically stored in a database object that resembles a table. This feature means that the underlying query only needs to be executed once and then the results are available to all who need them.

Oracle Database 12c allows for synchronous refreshes of the materialized views when configured to use a refresh method besides manual or on-demand. It utilizes partitioning and dependencies between the objects to minimize the time it takes to refresh and maintain the data as close to the underlying tables as possible.

From a database perspective, materialized views are treated like tables:

- You can perform most DML and query commands such as insert, delete, update, and select.
- They can be partitioned.
- They can be compressed.
- They can be parallelized.
- You can create indexes on them.

Materialized views are different in other ways and have some interesting features associated with them. Before we talk about those, let’s look at some ways to use materialized views.

Uses for Materialized Views

Materialized views are used as a performance-enhancing technique. In this section, you learn about the following uses of these views, as they are applicable to the topic of large databases.

- Performing data summarization (for example, sums and averages)
- Prejoining tables
- Performing CPU-intensive calculations
- Replicating and distributing data

In large databases, particularly data warehousing environments, there is always a need to summarize, join, perform calculations, or do all three operations at once.
on large numbers of records for the purposes of reporting and analysis. To improve performance in the past, a combination of views and physical tables were usually implemented that contained the results of these operations. The summary tables would require some type of extraction, transformation, and load (ETL) process to populate and refresh them. In addition to the base tables containing the detailed data, the users would need to know which combinations of the views and/or summary tables to use. These structures are illustrated in Figure 8-7.

Using materialized views has several advantages over more traditional methods. These include the following:

■ Materialized views have a built-in data refresh process, which can provide an automatic update or repopulation of a materialized view without any programming on the part of the DBA.

■ As mentioned earlier, the data in materialized views can be partitioned, using the same techniques that apply to tables.

■ Materialized views are transparent to the users. This is probably the most attractive feature of using materialized views. We expand more on this in the next section when we discuss automatic query rewriting.

Figure 8-8 illustrates summarization using materialized views.

**FIGURE 8-7.** Summarization using views and summary tables
Query Rewrite

Earlier, you learned that one of the benefits of using materialized views was that they are transparent to the users. But what exactly does that mean and how can they be used if the users can’t see them? In fact, because materialized views are so much like tables, you can give the users access to materialized views, although generally this is not done.

Instead, as indicated in Figure 8-8, the users always query the tables with the detail data—they don’t usually query the materialized views directly because the query optimizer in Oracle Database 12c knows about the materialized views and their relationships to the detail tables and can rewrite the query on-the-fly to access the materialized views instead. This results in huge performance gains without the user having to do anything special—just query the detail data. There is a maintenance benefit of this feature for the user as well: The queries do not have to change to point to different summary tables, as is the case with the more traditional summarization approach.

In order for the query to be rewritten, the structure of the materialized view must satisfy the criteria of the query. The following two listings demonstrate the query
rewrite process. Let’s assume you need to summarize the COMMISSION table you saw in the data compression section using the following query:

```sql
select prod_id, to_char(comm_date, 'YYYY-MM'), count(*), sum(comm_amt)
from commission
group by prod_id, to_char(comm_date, 'YYYY-MM');
```

Assume further that a materialized view (called comm_prod_mv) exists that contains summarized commission data by sales_rep_id, prod_id and comm_date (full date). In this case, the query would be automatically rewritten as follows:

```sql
select prod_id, to_char(comm_date, 'YYYY-MM'), count(*), sum(comm_amt)
from comm_prod_mv
group by prod_id, to_char(comm_date, 'YYYY-MM');
```

By rewriting the query to use the materialized view instead, a large amount of data-crunching has been saved and the results will return much more quickly. Now turn your attention to determining what materialized views should be created.

**When to Create Materialized Views**

At this point, you may be asking yourself: “How do I determine what materialized views to create and at what level of summarization?” Oracle Database 12c has some utilities to help. These utilities are collectively called the SQL Tuning Advisor and will recommend materialized views based on historical queries, or based on theoretical scenarios. They can be run from the Oracle Enterprise Manager (OEM) Grid Control, or by calling the dbms_advisor package.

**Create Materialized Views**

Materialized views are created using a create materialized view statement, which is similar to a create table statement. This can be performed using SQL Developer, SQL*Plus, or OEM. The following listing shows a simple example of how to create the comm_prod_mv materialized view mentioned earlier, and Table 8-16 provides an explanation of the syntax:

```sql
create materialized view comm_prod_mv
  tablespace comm_prod_mv_ts
  refresh complete next sysdate + 7
  enable query rewrite
  as select sales_rep_id, prod_id, comm_date, count(*), sum(comm_amt)
  from commission
  group by sales_rep_id, prod_id, comm_date;
```
CRITICAL SKILL 8.6

Use SQL Aggregate and Analysis Functions

Once your database has been loaded with data, your users or applications will, of course, want to use that data to run queries, perform analysis, produce reports, extract data, and so forth. Having large sets of data is only valuable to the business if it can be aggregated and reported on in an efficient manner. Oracle Database 12c provides many sophisticated aggregation and analysis functions that can help ease the pain sometimes associated with analyzing data in large databases.

Aggregation Functions

Oracle Database provides extensions to the standard SQL group by clause of the select statement that generate other totals as part of the result set that previously required multiple queries, nested subqueries, or importing into spreadsheet type applications. These extensions are rollup and cube.

rollup

The rollup extension generates subtotals for attributes specified in the group by clause, plus another row representing the grand total. The following example of the rollup extension uses the SALES table:

```sql
SELECT c.cust_gender gender,
       b.channel_class channel_class,
       TO_CHAR(a.time_id, 'yyyy-mm') month,
       COUNT(*) unit_count,
       SUM(a.amount_sold) amount_sold
FROM sales a,
     channel b,
     customer c
GROUP BY c.cust_gender,
         b.channel_class,
         TO_CHAR(a.time_id, 'yyyy-mm')
HAVING COUNT(*) > 0
ORDER BY c.cust_gender,
         b.channel_class,
         TO_CHAR(a.time_id, 'yyyy-mm');
```
```sql
from sales a, channels b, customers c
where a.channel_id = b.channel_id
and a.cust_id = c.cust_id
and to_char(a.time_id, 'yyyy-mm') between '2001-01' and '2001-02'
group by rollup(c.cust_gender, b.channel_class, 
        to_char(a.time_id, 'yyyy-mm'));
```

<table>
<thead>
<tr>
<th>GENDER</th>
<th>CHANNEL_CLASS</th>
<th>MONTH</th>
<th>UNIT_COUNT</th>
<th>AMOUNT_SOLD</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>Direct</td>
<td>2001-01</td>
<td>4001</td>
<td>387000.9</td>
</tr>
<tr>
<td>F</td>
<td>Direct</td>
<td>2001-02</td>
<td>3208</td>
<td>365860.13</td>
</tr>
<tr>
<td>F</td>
<td>Direct</td>
<td>7209</td>
<td>752861.03</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>Others</td>
<td>2001-01</td>
<td>2486</td>
<td>242615.9</td>
</tr>
<tr>
<td>F</td>
<td>Others</td>
<td>2001-02</td>
<td>2056</td>
<td>229633.52</td>
</tr>
<tr>
<td>F</td>
<td>Others</td>
<td>4542</td>
<td>472249.42</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>Indirect</td>
<td>2001-01</td>
<td>1053</td>
<td>138395.21</td>
</tr>
<tr>
<td>F</td>
<td>Indirect</td>
<td>2001-02</td>
<td>1470</td>
<td>189425.88</td>
</tr>
<tr>
<td>F</td>
<td>Indirect</td>
<td>2523</td>
<td>327821.09</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>Indirect</td>
<td>14274</td>
<td>1552931.54</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>Direct</td>
<td>2001-01</td>
<td>7018</td>
<td>719146.28</td>
</tr>
<tr>
<td>M</td>
<td>Direct</td>
<td>2001-02</td>
<td>6180</td>
<td>641192.61</td>
</tr>
<tr>
<td>M</td>
<td>Direct</td>
<td>13218</td>
<td>1360338.89</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>Others</td>
<td>2001-01</td>
<td>4310</td>
<td>414603.03</td>
</tr>
<tr>
<td>M</td>
<td>Others</td>
<td>2001-02</td>
<td>3751</td>
<td>391792.61</td>
</tr>
<tr>
<td>M</td>
<td>Others</td>
<td>8061</td>
<td>806395.64</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>Indirect</td>
<td>2001-01</td>
<td>1851</td>
<td>211947.81</td>
</tr>
<tr>
<td>M</td>
<td>Indirect</td>
<td>2001-02</td>
<td>2520</td>
<td>285219.79</td>
</tr>
<tr>
<td>M</td>
<td>Indirect</td>
<td>4371</td>
<td>497167.6</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>Indirect</td>
<td>25650</td>
<td>2663902.13</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td></td>
<td>39924</td>
<td>4216833.67</td>
<td></td>
</tr>
</tbody>
</table>

In the results, you can see that counts and sums of amount_sold are returned at the following levels:

- By GENDER, CHANNEL_CLASS, and MONTH
- Subtotals by CHANNEL_CLASS within GENDER
- Subtotals by GENDER
- Grand total
362 Oracle Database 12c: Install, Configure & Maintain Like a Professional

cube

The cube extension takes rollup a step further by generating subtotals for each combination of the group by attributes, totals by attribute, and the grand total. The following example of the cube extension uses the same query you used for rollup:

```sql
select c.cust_gender gender,
       b.channel_class channel_class,
       to_char(a.time_id, 'yyyy-mm') month,
       count(*) unit_count,
       sum(a.amount_sold) amount_sold
from sales a, channels b, customers c
where a.channel_id = b.channel_id
  and a.cust_id = c.cust_id
  and to_char(a.time_id, 'yyyy-mm') between '2001-01' and '2001-02'
group by cube(c.cust_gender,
               b.channel_class,
               to_char(a.time_id, 'yyyy-mm'));
```

<table>
<thead>
<tr>
<th>GENDER</th>
<th>CHANNEL_CLASS</th>
<th>MONTH</th>
<th>UNIT_COUNT</th>
<th>AMOUNT_SOLD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct</td>
<td></td>
<td>2001-01</td>
<td>20739</td>
<td>2113709.13</td>
</tr>
<tr>
<td>Direct</td>
<td></td>
<td>2001-02</td>
<td>19185</td>
<td>2103124.54</td>
</tr>
<tr>
<td>Direct</td>
<td></td>
<td>2001-01</td>
<td>11039</td>
<td>1106147.18</td>
</tr>
<tr>
<td>Direct</td>
<td></td>
<td>2001-02</td>
<td>9388</td>
<td>1007052.74</td>
</tr>
<tr>
<td>Others</td>
<td></td>
<td>2001-01</td>
<td>6796</td>
<td>657218.93</td>
</tr>
<tr>
<td>Others</td>
<td></td>
<td>2001-02</td>
<td>5807</td>
<td>621426.13</td>
</tr>
<tr>
<td>Indirect</td>
<td></td>
<td>2001-01</td>
<td>2904</td>
<td>350343.02</td>
</tr>
<tr>
<td>Indirect</td>
<td></td>
<td>2001-02</td>
<td>3990</td>
<td>474645.67</td>
</tr>
<tr>
<td>F</td>
<td></td>
<td>2001-01</td>
<td>7540</td>
<td>768012.01</td>
</tr>
<tr>
<td>F</td>
<td></td>
<td>2001-02</td>
<td>6734</td>
<td>784919.53</td>
</tr>
<tr>
<td>F</td>
<td>Direct</td>
<td>2001-01</td>
<td>4001</td>
<td>387000.9</td>
</tr>
<tr>
<td>F</td>
<td>Direct</td>
<td>2001-02</td>
<td>3208</td>
<td>365860.13</td>
</tr>
<tr>
<td>F</td>
<td>Others</td>
<td>2001-01</td>
<td>2486</td>
<td>242615.9</td>
</tr>
<tr>
<td>F</td>
<td>Others</td>
<td>2001-02</td>
<td>2056</td>
<td>229633.52</td>
</tr>
<tr>
<td>F</td>
<td>Indirect</td>
<td>2001-01</td>
<td>1053</td>
<td>138395.21</td>
</tr>
<tr>
<td>F</td>
<td>Indirect</td>
<td>2001-02</td>
<td>1470</td>
<td>189425.88</td>
</tr>
<tr>
<td>M</td>
<td></td>
<td>2001-01</td>
<td>25650</td>
<td>2663902.13</td>
</tr>
<tr>
<td>M</td>
<td></td>
<td>2001-02</td>
<td>13199</td>
<td>1345697.12</td>
</tr>
</tbody>
</table>
Chapter 8:  Using and Managing Large Databases  363

In the results, you can see that counts and sums of amount_sold are returned at the following levels:

- By GENDER, CHANNEL_CLASS, and MONTH
- Subtotals by MONTH within CHANNEL_CLASS
- Subtotals by MONTH within GENDER
- Subtotals by CHANNEL_CLASS within GENDER
- Subtotals by MONTH
- Subtotals by CHANNEL_CLASS
- Subtotals by GENDER
- Grand total

Analytic Functions

Oracle Database 11c provides a number of ranking and statistical functions that previously would have required some pretty heavy SQL to perform or an extract to a third-party application. In this section, you look at the available analysis functions and examine examples of their use where appropriate.

NOTE

Some of the functions in this section are based on complex statistical calculations. Don’t be worried if you are unfamiliar with these concepts. It is more important for you to know that these functions exist than it is to understand the theory behind them.
Ranking Functions
Ranking functions provide the capability to rank a row of a query result relative to the other rows in the result set. Common examples of uses for these functions include identifying the top ten selling products for a period, or classifying or grouping a salesperson’s commissions into one of four buckets. The ranking functions included in Oracle Database are

- RANK
- DENSE_RANK
- CUME_DIST
- PERCENT_RANK
- NTILE
- ROW_NUMBER

The simplest ranking functions are RANK and DENSE_RANK. These functions are very similar and determine the ordinal position of each row within the query result set. The difference between these two functions is that rank will leave a gap in the sequence when there is a tie for position, whereas dense_rank does not leave a gap. The results of the following listing illustrate the difference between the two:

```sql
select prod_id,
       sum(quantity_sold),
       rank() over (order by sum(quantity_sold) desc) as rank,
       dense_rank() over (order by sum(quantity_sold) desc) as dense_rank
from sales
where to_char(time_id, 'yyyy-mm') = '2001-06'
group by prod_id;
```

<table>
<thead>
<tr>
<th>PROD_ID</th>
<th>SUM(QUANTITY_SOLD)</th>
<th>RANK</th>
<th>DENSE_RANK</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>762</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>30</td>
<td>627</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>147</td>
<td>578</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>33</td>
<td>552</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>40</td>
<td>550</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>133</td>
<td>550</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>48</td>
<td>541</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>120</td>
<td>538</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>23</td>
<td>535</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>119</td>
<td>512</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>124</td>
<td>503</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>140</td>
<td>484</td>
<td>12</td>
<td>11</td>
</tr>
</tbody>
</table>
As you can see, the ordinal position 6 does not exist as a value for RANK, but it does for DENSE_RANK. If, from this result set, you wanted to see the top ten listings for prod_id, you would use the original query as a subquery, as in the following listing:

```sql
select * from
(select prod_id,
  sum(quantity_sold),
  rank () over (order by sum(quantity_sold) desc) as rank,
  dense_rank () over (order by sum(quantity_sold) desc) as dense_rank
from sales
where to_char(time_id, 'yyyy-mm') = '2001-06'
group by prod_id)
where rank < 11;
```

To see the bottom ten prod_ids, use the same query, but change the order by option from descending (DESC) to ascending (ASC). The next two functions, cume_dist and percent_rank, are statistical in nature but still part of the family of ranking functions, so the important feature is that these functions properly handle ties in the rankings.

The CUME_DIST function calculates the cumulative distribution of a value in a group of values. Because this is not a statistics beginner’s guide, we will not attempt to provide the theoretical background on cumulative distribution. However, we can offer these points:

- The range of values returned by cume_dist is always between 0 and 1.
- The value returned by cume_dist is always the same in the case of tie values in the group.
- The formula for cumulative distribution is
  
  $\frac{\text{# of rows with values } \leq \text{ value of row being evaluated}}{\text{# of rows being evaluated}}$


Consider the query we used when discussing rank; the following listing calculates the cumulative distribution for quantity_sold. The results immediately follow the listing.

```sql
select prod_id, 
       sum(quantity_sold), 
       cume_dist() over (order by sum(quantity_sold) asc) as cume_dist 
from sales 
where to_char(time_id, 'yyyy-mm') = '2001-06' 
group by prod_id 
order by sum(quantity_sold) desc;
```

<table>
<thead>
<tr>
<th>PROD_ID</th>
<th>SUM(QUANTITY_SOLD)</th>
<th>CUME_DIST</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>762</td>
<td>1</td>
</tr>
<tr>
<td>30</td>
<td>627 .985915493</td>
<td></td>
</tr>
<tr>
<td>147</td>
<td>578 .971830986</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>552 .957746479</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>550 .943661972</td>
<td></td>
</tr>
<tr>
<td>133</td>
<td>550 .943661972</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>541 .915492958</td>
<td></td>
</tr>
<tr>
<td>120</td>
<td>538 .901408451</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>535 .887323944</td>
<td></td>
</tr>
<tr>
<td>119</td>
<td>512 .873239437</td>
<td></td>
</tr>
<tr>
<td>124</td>
<td>503 .85915493</td>
<td></td>
</tr>
<tr>
<td>140</td>
<td>484 .845070423</td>
<td></td>
</tr>
<tr>
<td>148</td>
<td>472 .830985915</td>
<td></td>
</tr>
<tr>
<td>139</td>
<td>464 .816901408</td>
<td></td>
</tr>
<tr>
<td>123</td>
<td>459 .802816901</td>
<td></td>
</tr>
<tr>
<td>131</td>
<td>447 .788732394</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>420 .774647887</td>
<td></td>
</tr>
<tr>
<td>135</td>
<td>415 .76056338</td>
<td></td>
</tr>
<tr>
<td>137</td>
<td>407 .746478873</td>
<td></td>
</tr>
<tr>
<td>146</td>
<td>401 .732394366</td>
<td></td>
</tr>
</tbody>
</table>

The percent_rank function is similar to the cume_dist function, but calculates a percentage ranking of a value relative to its group. Again, without getting into the theory, we can make some points about percent_rank:

- The range of values returned by the function is always between 0 and 1.
- The row with a rank of 1 will have a percent rank of 0.
- The formula for calculating percent rank is
  \[
  \text{Percent Rank} = \frac{\text{Rank of row within its group} - 1}{\text{# of rows in the group} - 1}
  \]
The next listing and its results use the base query you have been using in this section to demonstrate the PERCENT_RANK function:

```sql
select prod_id,
       sum(quantity_sold),
       rank() over (order by sum(quantity_sold) desc) as rank,
       percent_rank() over (order by sum(quantity_sold) asc) as percent_rank
from sales
where to_char(time_id, 'yyyy-mm') = '2001-06'
group by prod_id
order by sum(quantity_sold) desc;
```

<table>
<thead>
<tr>
<th>PROD_ID</th>
<th>SUM(QUANTITY_SOLD)</th>
<th>RANK</th>
<th>PERCENT_RANK</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>762</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>30</td>
<td>627</td>
<td>2</td>
<td>.985714286</td>
</tr>
<tr>
<td>147</td>
<td>578</td>
<td>3</td>
<td>.971428571</td>
</tr>
<tr>
<td>33</td>
<td>552</td>
<td>4</td>
<td>.957142857</td>
</tr>
<tr>
<td>40</td>
<td>550</td>
<td>5</td>
<td>.928571429</td>
</tr>
<tr>
<td>133</td>
<td>550</td>
<td>5</td>
<td>.928571429</td>
</tr>
<tr>
<td>48</td>
<td>541</td>
<td>7</td>
<td>.914285714</td>
</tr>
<tr>
<td>120</td>
<td>538</td>
<td>8</td>
<td>.9</td>
</tr>
<tr>
<td>23</td>
<td>535</td>
<td>9</td>
<td>.885714286</td>
</tr>
<tr>
<td>119</td>
<td>512</td>
<td>10</td>
<td>.871428571</td>
</tr>
<tr>
<td>124</td>
<td>503</td>
<td>11</td>
<td>.857142857</td>
</tr>
<tr>
<td>140</td>
<td>484</td>
<td>12</td>
<td>.842857143</td>
</tr>
<tr>
<td>148</td>
<td>472</td>
<td>13</td>
<td>.828571429</td>
</tr>
<tr>
<td>139</td>
<td>464</td>
<td>14</td>
<td>.814285714</td>
</tr>
<tr>
<td>123</td>
<td>459</td>
<td>15</td>
<td>.8</td>
</tr>
<tr>
<td>131</td>
<td>447</td>
<td>16</td>
<td>.785714286</td>
</tr>
<tr>
<td>25</td>
<td>420</td>
<td>17</td>
<td>.771428571</td>
</tr>
<tr>
<td>135</td>
<td>415</td>
<td>18</td>
<td>.757142857</td>
</tr>
<tr>
<td>137</td>
<td>407</td>
<td>19</td>
<td>.742857143</td>
</tr>
<tr>
<td>146</td>
<td>401</td>
<td>20</td>
<td>.728571429</td>
</tr>
</tbody>
</table>

The ntile function divides a result set into a number of buckets specified at query time by the user, and then assigns each row in the result set a bucket number. The most common numbers of buckets used are 3 (tertiles), 4 (quartiles), and 10 (deciles). Each bucket will have the same number of rows, except in the case when the number of rows does not divide evenly by the number of buckets. In this case, each of the leftover rows will be assigned to buckets with the lowest bucket numbers until all leftover rows are assigned. For example, if four buckets were specified and the number of rows in the result set was 98, buckets 1 and 2 would have 25 rows each, and buckets 3 and 4 would have 24 rows each.
Let's look at an example. Using your base query of `amount_sold` in the SALES table, you want to look at `amount_sold` by product subcategory and rank the amounts into four buckets. Here's the SQL:

```sql
select b.prod_subcategory,
       sum(a.quantity_sold),
       ntile(4) over (ORDER BY SUM(a.quantity_sold) desc) as quartile
from sales a, products b
where a.prod_id = b.prod_id
and to_char(a.time_id, 'yyyy-mm') = '2001-06'
group by b.prod_subcategory;
```

As you can see in the following results, the number of product subcategories was not evenly divisible by the number of buckets specified (in this case, 4). Therefore, six subcategories were assigned to the first quartile (bucket 1) and five subcategories were assigned to the second, third, and fourth quartiles.

<table>
<thead>
<tr>
<th>PROD_SUBCATEGORY</th>
<th>COUNT(*)</th>
<th>SUM(A.QUANTITY_SOLD)</th>
<th>QUARTILE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accessories</td>
<td>3230</td>
<td>3230</td>
<td>1</td>
</tr>
<tr>
<td>Y Box Games</td>
<td>2572</td>
<td>2572</td>
<td>1</td>
</tr>
<tr>
<td>Recordable CDs</td>
<td>2278</td>
<td>2278</td>
<td>1</td>
</tr>
<tr>
<td>Camera Batteries</td>
<td>2192</td>
<td>2192</td>
<td>1</td>
</tr>
<tr>
<td>Recordable DVD Discs</td>
<td>2115</td>
<td>2115</td>
<td>1</td>
</tr>
<tr>
<td>Documentation</td>
<td>1931</td>
<td>1931</td>
<td>1</td>
</tr>
<tr>
<td>Modems/Fax</td>
<td>1314</td>
<td>1314</td>
<td>2</td>
</tr>
<tr>
<td>CD-ROM</td>
<td>1076</td>
<td>1076</td>
<td>2</td>
</tr>
<tr>
<td>Y Box Accessories</td>
<td>1050</td>
<td>1050</td>
<td>2</td>
</tr>
<tr>
<td>Printer Supplies</td>
<td>956</td>
<td>956</td>
<td>2</td>
</tr>
<tr>
<td>Memory</td>
<td>748</td>
<td>748</td>
<td>2</td>
</tr>
<tr>
<td>Camera Media</td>
<td>664</td>
<td>664</td>
<td>3</td>
</tr>
<tr>
<td>Home Audio</td>
<td>370</td>
<td>370</td>
<td>3</td>
</tr>
<tr>
<td>Game Consoles</td>
<td>352</td>
<td>352</td>
<td>3</td>
</tr>
<tr>
<td>Operating Systems</td>
<td>343</td>
<td>343</td>
<td>3</td>
</tr>
<tr>
<td>Bulk Pack Diskettes</td>
<td>270</td>
<td>270</td>
<td>3</td>
</tr>
<tr>
<td>Portable PCs</td>
<td>215</td>
<td>215</td>
<td>4</td>
</tr>
<tr>
<td>Desktop PCs</td>
<td>214</td>
<td>214</td>
<td>4</td>
</tr>
<tr>
<td>Camcorders</td>
<td>196</td>
<td>196</td>
<td>4</td>
</tr>
<tr>
<td>Monitors</td>
<td>178</td>
<td>178</td>
<td>4</td>
</tr>
<tr>
<td>Cameras</td>
<td>173</td>
<td>173</td>
<td>4</td>
</tr>
</tbody>
</table>

The `ROW_NUMBER` function is a simple one that assigns a unique number to each row in a result set. The numbers are sequential, starting at 1, and are based on the order by clause of the query.
You will again use your SALES table query as an example. Using the query from your ntile example and adding a new column using row_count, you get the following:

```sql
select b.prod_subcategory,
       sum(a.quantity_sold),
       ntile(4) over (ORDER BY SUM(a.quantity_sold) desc) as quartile,
       row_number () over (order by sum(quantity_sold) desc) as rownumber
from sales a, products b
where a.prod_id = b.prod_id
and to_char(a.time_id, 'yyyy-mm') = '2001-06'
group by b.prod_subcategory;
```

In the following results, each row is assigned a number, depending on its position defined by the order by clause. As you can see, the ROWNUMBER is simply the row position in the list without any intelligence, whereas quartile is a calculated field and could have repeating values:

<table>
<thead>
<tr>
<th>PROD_SUBCATEGORY</th>
<th>SUM(A.QUANTITY_SOLD)</th>
<th>QUARTILE</th>
<th>ROWNUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accessories</td>
<td>3230</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Y Box Games</td>
<td>2572</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Recordable CDs</td>
<td>2278</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Camera Batteries</td>
<td>2192</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Recordable DVD Discs</td>
<td>2115</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Documentation</td>
<td>1931</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>Modems/Fax</td>
<td>1314</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>CD-ROM</td>
<td>1076</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>Y Box Accessories</td>
<td>1050</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>Printer Supplies</td>
<td>956</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>Memory</td>
<td>748</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>Camera Media</td>
<td>664</td>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>Home Audio</td>
<td>370</td>
<td>3</td>
<td>13</td>
</tr>
<tr>
<td>Game Consoles</td>
<td>352</td>
<td>3</td>
<td>14</td>
</tr>
<tr>
<td>Operating Systems</td>
<td>343</td>
<td>3</td>
<td>15</td>
</tr>
<tr>
<td>Bulk Pack Diskettes</td>
<td>270</td>
<td>3</td>
<td>16</td>
</tr>
<tr>
<td>Portable PCs</td>
<td>215</td>
<td>4</td>
<td>17</td>
</tr>
<tr>
<td>Desktop PCs</td>
<td>214</td>
<td>4</td>
<td>18</td>
</tr>
<tr>
<td>Camcorders</td>
<td>196</td>
<td>4</td>
<td>19</td>
</tr>
<tr>
<td>Monitors</td>
<td>178</td>
<td>4</td>
<td>20</td>
</tr>
<tr>
<td>Cameras</td>
<td>173</td>
<td>4</td>
<td>21</td>
</tr>
</tbody>
</table>
Windowing Functions

Before you get into the details, you need to learn a couple of terms: analytic partitioning and analytic window:

- **Analytic partitioning** is the division of the results of an analytic function into groups within which the analytic function operates. This is accomplished using the partition by clause of the analytic function. Do not confuse this partitioning with data partitioning discussed earlier in this chapter. Analytic partitioning can be used with any of the analytic functions we have discussed so far.

- **An analytic window** is a subset of an analytic partition in which the values of each row depend on the values of other rows in the window. There are two types of windows: physical and logical. A physical window is defined by a specified number of rows. A logical window is defined by the order by values.

Windowing functions can only be used in the select and order by clauses. They can be used to calculate the following:

- Moving sum
- Moving average
- Moving min/max
- Cumulative sum
- Statistical functions

Now look at an example of a moving sum function. The following shows the listing and results for calculating the moving sum from the SALES table by product category for a six-month period:

```sql
select b.prod_category,
       to_char(a.time_id, 'yyyy-mm'),
       sum(a.quantity_sold),
       sum(sum(a.quantity_sold)) over (partition by b.prod_category
       order by to_char(a.time_id, 'yyyy-mm')
       rows unbounded preceding) as cume_sum
from sales a, products b
where a.prod_id = b.prod_id
  and b.prod_category_id between 202 and 204
  and to_char(a.time_id, 'yyyy-mm') between '2001-01' and '2001-06'
```
As you can see in the results, the moving sum is contained within each product category and resets when a new product category starts. In the past, windowing analysis used to require third-party products such as spreadsheet applications. Having the capabilities to perform these functions right in the database can streamline analysis and report generation efforts.

**Other Functions**

Many other functions are included with Oracle Database 12c that can be used to analyze data in large databases. While we will not be going into any detail for these, they are listed here for completeness:

- Statistical functions, including the following:
  - Linear regression functions
  - Descriptive statistics functions
  - Hypothetical testing and crosstab statistics functions (containing a new PL/SQL package called dbms_statistics)
- first/last functions
- lag/lead functions
As stated earlier, database administrators do not have to know the theory behind the functions provided by Oracle Database 12c or even how to use the results. However, you should be able to let your users know what capabilities are available. Knowing this, your users will be able to take advantage of these functions and construct efficient queries. In the next section, you learn about a new feature in Oracle Database 12c—SQL models.

**CRITICAL SKILL 8.7**

**Create SQL Models**

One of the more powerful data analysis features introduced in Oracle Database 12c is *SQL models*. SQL models allow a user to create multidimensional arrays from query results. Formulas, both simple and complex, can then be applied to the arrays to generate results in which the user is interested. SQL models allow inter-row calculations to be applied without doing expensive self-joins.

SQL models are similar to other multidimensional structures used in business intelligence applications. However, because they are part of the database, they can take advantage of Oracle Database’s built-in features of scalability, manageability, security, and so on. In addition, when using SQL models, there is no need to transfer large amounts of data to external business intelligence applications.

A SQL model is defined by the model extension of the select statement. Columns of a query result are classified into one of three groups:

- **Partitioning** This is the same as the analytic partitioning defined in the “Windowing Functions” section.
- **Dimensions** These are the attributes used to describe or fully qualify a measure within a partition. Examples could include product, sales rep ID, and phone call type.
- **Measures** These are the numeric (usually) values to which calculations are applied. Examples could include quantity sold, commission amount, and call duration.
One of the main applications of SQL models is projecting or forecasting measures based on existing measures. Let’s look at an example of the model clause to illustrate. The listing and its results show an aggregate query using the SALES table:

```sql
select c.channel_desc, p.prod_category, t.calendar_year year, 
    sum(s.quantity_sold) quantity_sold 
from sales s, products p, channels c, times t 
where s.prod_id = p.prod_id 
and s.channel_id = c.channel_id 
and s.time_id = t.time_id 
and c.channel_desc = 'Direct Sales' 
group by c.channel_desc, p.prod_category, t.calendar_year 
order by c.channel_desc, p.prod_category, t.calendar_year;
```

<table>
<thead>
<tr>
<th>CHANNEL_DESC</th>
<th>PROD_CATEGORY</th>
<th>YEAR</th>
<th>QUANTITY_SOLD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Sales</td>
<td>Electronics</td>
<td>1998</td>
<td>7758</td>
</tr>
<tr>
<td>Direct Sales</td>
<td>Electronics</td>
<td>1999</td>
<td>15007</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Direct Sales</td>
<td>Hardware</td>
<td>2000</td>
<td>1970</td>
</tr>
<tr>
<td>Direct Sales</td>
<td>Hardware</td>
<td>2001</td>
<td>2399</td>
</tr>
<tr>
<td>Direct Sales</td>
<td>Peripherals and Accessories</td>
<td>1998</td>
<td>44258</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Direct Sales</td>
<td>Software/Other</td>
<td>2000</td>
<td>64483</td>
</tr>
<tr>
<td>Direct Sales</td>
<td>Software/Other</td>
<td>2001</td>
<td>49146</td>
</tr>
</tbody>
</table>

In the results, you can see the historical aggregate quantity_sold for each year by product category for the Direct Sales channel. You can use the model clause to project the quantity_sold. In the following listing, you’ll project values for 2002 for the product category Hardware in the channel. The quantity_sold will be based on the previous year’s value (2001), plus 10 percent. Table 8-17 explains the syntax of the listing.

```sql
1 select channel_desc, prod_category, year, quantity_sold 
2 from 
3 (select c.channel_desc, p.prod_category, t.calendar_year year, 
4    sum(s.quantity_sold) quantity_sold 
5 from sales s, products p, channels c, times t 
6 where s.prod_id = p.prod_id 
7 and s.channel_id = c.channel_id 
8 and s.time_id = t.time_id 
9 group by c.channel_desc, p.prod_category, t.calendar_year) sales 
10 where channel_desc = 'Direct Sales' 
11 model 
12 partition by (channel_desc) 
13 dimension by (prod_category, year) 
14 measures (quantity_sold)
```
Oracle Database 12c: Install, Configure & Maintain Like a Professional

15 rules (quantity_sold['Hardware', 2002]
16 = quantity_sold['Hardware', 2001] * 1.10)
17 order by channel_desc, prod_category, year;

Following are the results of the previous query. Notice that a new row has been added for Hardware in 2002. Its quantity_sold is 2638.9, which is the previous year’s value (2399) plus 10 percent.

<table>
<thead>
<tr>
<th>CHANNEL_DESC</th>
<th>PROD_CATEGORY</th>
<th>YEAR</th>
<th>QUANTITY_SOLD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Sales</td>
<td>Electronics</td>
<td>1998</td>
<td>7758</td>
</tr>
<tr>
<td>Direct Sales</td>
<td>Electronics</td>
<td>1999</td>
<td>15007</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Direct Sales</td>
<td>Hardware</td>
<td>2000</td>
<td>1970</td>
</tr>
<tr>
<td>Direct Sales</td>
<td>Hardware</td>
<td>2001</td>
<td>2399</td>
</tr>
<tr>
<td>Direct Sales</td>
<td>Hardware</td>
<td>2002</td>
<td>2638.9</td>
</tr>
<tr>
<td>Direct Sales</td>
<td>Peripherals and Accessories</td>
<td>1998</td>
<td>44258</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Direct Sales</td>
<td>Software/Other</td>
<td>2000</td>
<td>64483</td>
</tr>
<tr>
<td>Direct Sales</td>
<td>Software/Other</td>
<td>2001</td>
<td>49146</td>
</tr>
</tbody>
</table>

**Lines Important Points**

3–9 Defines an in-line select that will be the source for the query. It is basically the same query you started with before the model clause.

10 Specifies that you are only going to look at Direct Sales channels.

11 Specifies the model clause.

12 Specifies the partition by clause (in this case, channel_desc).

13 Specifies the dimension by clause (here, prod_category and year). These elements will fully qualify the measure within the channel_desc partition.

14 Specifies the measures clause (quantity_sold).

15–16 Specifies the rules clause—that is, calculations you want to perform on the measure. In this example, we are referring to a specific cell of quantity_sold, described by the dimensions prod_category (Hardware) and year (2002).

**TABLE 8-17.** Explanation of Model Clause Syntax
The model clause has many variations and allows for very powerful calculations. We want to point out some of the characteristics and/or features you should be aware of. Supported functionalities include the following:

- Looping (for example, FOR loops)
- Recursive calculations
- Regression calculations
- Nested cell references
- Dimension wildcards and ranges
- The model clause does not update any base table, although in theory, you could create a table or materialized view from the results of the query using the model clause.

Restrictions include the following:

- The rules clause cannot include any analytic SQL or windowing functions.
- A maximum of 20,000 rules may be specified. This may seem like plenty, but a FOR loop is expanded into many single-cell rules at execution time.

Project 8-2 Use Analytic SQL Functions and Models

Once all the database structures have been put in place and data has been loaded, the users will want to analyze this data which has been placed in the database. Knowing what functions are available is important, and so is their use as well, at least to some extent. So, this project walks you through a more complex analytical example that includes using the lag function and creating a SQL model.

Step by Step

1. Create a view of the SALES table using the following listing. (The SALES table should have been created during your Oracle installation process.) This view will calculate the percentage change (called percent_chng) of quantity_sold from one year to the next using the lag function, summarized by prod_category, channel_desc, and calendar_year.

   ```sql
   create or replace view sales_trends
   as
   select p.prod_category, c.channel_desc, t.calendar_year year,
   sum(s.quantity_sold) quantity_sold,
   round((sum(s.quantity_sold) - lag(sum(s.quantity_sold)) over (partition by p.prod_category, c.channel_desc, calendar_year)) / lag(sum(s.quantity_sold) over (partition by p.prod_category, c.channel_desc, calendar_year)), 2) percent_chng
   from sales s, prod p, channel c, time t
   group by p.prod_category, c.channel_desc, calendar_year
   ```

   (continued)
lag(sum(s.quantity_sold),1) over (partition by p.prod_category, c.channel_desc order by t.calendar_year)) / lag(sum(s.quantity_sold),1) over (partition by p.prod_category, c.channel_desc order by t.calendar_year) * 100 ,2) as percent_chng from sales s, products p, channels c, times t where s.prod_id = p.prod_id and s.channel_id = c.channel_id and s.time_id = t.time_id group by p.prod_category, c.channel_desc, t.calendar_year;

2. Select from the sales_trends view using the following listing. Notice that quantity_sold and percent_chng reset after each channel_desc. This is a result of the lag function’s partition by clauses in the view definition.

```sql
select prod_category, channel_desc, year, quantity_sold, percent_chng from sales_trends where prod_category = 'Electronics' order by prod_category, channel_desc, year;
```

3. Select from the sales_trends view using the following listing, which contains a model clause. In this query, you are projecting quantity_sold and percent_chng according to the following rules:

a. Filter the prod_category to select only Electronics.

b. Project for years 2002 to 2006 inclusive.

c. The projected quantity_sold is calculated as the previous year’s value plus the average percent_chng over the previous three years.

d. The projected percent_chng is the average percent_chng over the previous three years:

```sql
select prod_category, channel_desc, year, quantity_sold, percent_chng from sales_trends where prod_category = 'Electronics' model partition by (prod_category, channel_desc) dimension by (year) measures (quantity_sold, percent_chng)
```
rules {
    percent_chng[for year from 2002 to 2006 increment 1] =
        round(avg(percent_chng)[year between currentv() - 3 and
            currentv() - 1], 2),
    quantity_sold[for year from 2002 to 2006 increment 1] =
        round(quantity_sold[currentv() - 1] *
            (1 + (round(avg(percent_chng)[year between
            currentv() - 3 and
            currentv() - 1], 2) / 100))),
    order by prod_category, channel_desc, year;

4. Notice the projected values for 2002 to 2006 for each channel_desc.

Project Summary

The steps in this project build on the discussions you’ve had on Oracle Database’s analytic capabilities. You used the lag function to calculate percentage change and used the model clause of the select statement to project sales five years into the future based on past trends. By going to the next level of examples, you can start to appreciate the significance of these functions and how they can be used.

Oracle Database 12c’s analytic functions provide powerful and efficient analysis capabilities that would otherwise require complex SQL and/or third-party tools. All of these functions are part of the core database—ready and waiting to be exploited by your users.

So, now you have come to the end of your exploration of large database features. We’ve presented a great deal of material in this chapter, and you have really only seen the tip of the iceberg! However, you can feel confident that with this background information, you are primed to tackle almost any large database environment out there.

☑ Chapter 8 Mastery Check

1. What data population methods can be used on a compressed table that results in the data being compressed?

2. What are the three basic types of partitioning?

3. ___________ partitioned indexes are defined independently of the data partitions, and ___________ partitioned indexes have a one-to-one relationship with the data partitions.

4. Explain the functions of the parallel execution coordinator in parallel processing.
5. For which of the following SQL commands is parallel processing not enabled by default?
   A. SELECT
   B. INSERT
   C. CREATE
   D. ALTER

6. What is meant by “degree of parallelism”?

7. What analytic feature can be used to forecast values based on existing measures?

8. What two methods can be used to run the SQL Tuning Advisor utilities for materialized views?

9. __________ partitioning allows you to control the placement of records in specified partitions based on sets of partition key values.

10. What analytic function would you use to assign each row from a query to one of five buckets?

11. What are the two types of windows that can be used in analytic functions?

12. The ___________ automatically collects workload and performance statistics used for database self-management activities.

13. When creating partitioned tables, what option should you use to ensure that rows are redistributed to their correct partition if their partition key is updated?

14. List the ways in which parallel processing can be invoked.