

4

Data Structures

In this chapter:

- *Datatypes*
- *Basic Data Structures*
- *Additional Data Structures*
- *Data Design*
- *Constraints*
- *Triggers*
- *Query Optimization*
- *Understanding the Execution Plan*
- *Data Dictionary Tables*

In previous chapters, we examined some distinctions between the different components that make up an Oracle database. For example, you've learned that the Oracle instance differs from the files that make up the physical storage of the data in tablespaces, that you cannot access the data in a tablespace except through an Oracle instance, and that the instance itself isn't very valuable without the data stored in those files.

In the same way, the actual tables and columns within the database are the entities stored within the database files and accessed through the database instance. The user who makes a request for data from an Oracle database probably doesn't know anything about instances and tablespaces, but does know about the structure of her data, as implemented with tables and columns. To fully leverage the power of Oracle, you must understand how the Oracle Database Server implements and uses these logical data structures.

Datatypes

The *datatype* is one of the attributes for a *column* or a variable in a stored procedure. A datatype describes and limits the type of information stored in a column, as well as some of the operations that you can perform on columns with a particular datatype.

You can divide Oracle datatype support into three basic varieties: character datatypes, numeric datatypes, and datatypes that represent other kinds of data. You use all of these datatypes when you create columns in a table, as with this SQL statement:

Datatypes

```
CREATE SAMPLE_TABLE (
    char_field CHAR(10),
    varchar_field VARCHAR2(10),
    todays_date DATE)
```

You also use these datatypes when you define variables as part of a PL/SQL procedure.

Character Datatypes

Character datatypes can store any string value, including the string representations of numeric values. Assigning a value larger than the length specified for a character datatype results in a runtime error. You can use string functions, such as UPPER, LOWER, SUBSTR, and SOUNDEx, on standard (not large) character value types.

There are several different character datatypes:

CHAR

The CHAR datatype stores character values with a fixed length. A CHAR datatype can have between 1 and 2,000 characters. If you don't explicitly specify a length for a CHAR, it assumes the default length of 1. If you assign a value that's shorter than the length specified for the CHAR datatype, Oracle will automatically pad the value with blanks. Some examples of CHAR values are:

```
CHAR(10) = "Rick      ", "Jon      ", "Stackowiak"
```

VARCHAR2

The VARCHAR2 datatype stores variable-length character strings. Although you must assign a length to a VARCHAR2 datatype, this length is the maximum length for a value rather than the required length. Values assigned to a VARCHAR2 datatype aren't padded with blanks. The VARCHAR2 datatype can have up to 4,000 characters.

A VARCHAR2 datatype can require less storage space than a CHAR datatype, because the VARCHAR2 datatype stores only the characters assigned to the column.

At this time, the VARCHAR and VARCHAR2 datatypes are synonymous in Oracle8 and later versions, but Oracle recommends the use of VARCHAR2 because future changes may cause VARCHAR and VARCHAR2 to diverge. The values shown earlier for the CHAR values, if entered as VARCHAR2 values, are:

```
VARCHAR2(10) = "Rick", "Jon", "Stackowiak"
```

NCHAR and NVARCHAR2

The NCHAR and NVARCHAR2 datatypes store fixed-length or variable-length character data using a different character set from the one used by the rest of the database. When you create a database, you specify the character set that

will be used for encoding the various characters stored in the database. You can optionally specify a secondary character set as well (this is known as the *National Language Set*, or NLS). The secondary character set will be used for NCHAR and NVARCHAR2 columns. For example, you may have a description field in which you want to store Japanese characters while the rest of the database uses English encoding. You would specify a secondary character set that supports Japanese characters when you create the database, and then use the NCHAR or NVARCHAR2 datatype for the columns in question.

With Oracle9i, you can specify that you want to indicate the length of NCHAR and NVARCHAR2 columns in terms of characters, rather than bytes. This new feature allows you to indicate, for example, that a column with one of these datatypes is 7 characters long. The Oracle9i database will automatically make the conversion to 14 bytes of storage if the character set requires double byte storage.

LONG

The LONG datatype can hold up to 2 GB of character data. It is regarded as a legacy datatype from earlier versions of Oracle. If you want to store large amounts of character data, Oracle now recommends that you use the CLOB and NCLOB datatypes. There are many restrictions on the use of LONG datatypes in a table and within SQL statements, such as the fact that you cannot use LONGs in WHERE, GROUP BY, ORDER BY, or CONNECT BY clauses or in SQL statements with the DISTINCT qualifier. You also cannot create an index on a LONG column.

CLOB and NCLOB

The CLOB and NCLOB datatypes can store up to 4 GB of character data. The NCLOB datatype stores the NLS data. For more information on CLOBs and NCLOBs, please refer to the discussion about large objects (LOBs) in “Other Datatypes,” later in this chapter.

Numeric Datatype

Oracle uses a standard, variable-length internal format for storing numbers. This internal format can maintain a precision of up to 38 digits.

The numeric datatype for Oracle is NUMBER. Declaring a column or variable as NUMBER will automatically provide a precision of 38 digits. The NUMBER datatype can also accept two qualifiers, as in:

```
column NUMBER( precision, scale )
```

The *precision* of the datatype is the total number of significant digits in the number. You can designate a precision for a number as any number of digits up

to 38. If no value is declared for *precision*, Oracle will use a precision of 38. The *scale* represents the number of digits to the right of the decimal point. If no scale is specified, Oracle will use a scale of 0.

If you assign a negative number to the *scale*, Oracle will round the number up to the designated place to the *left* of the decimal point. For example, the following code snippet:

```
column_round NUMBER(10,-2)
column_round = 1,234,567
```

will give `column_round` a value of 1,234,600.

The NUMBER datatype is the only datatype that stores numeric values in Oracle. The ANSI datatypes of DECIMAL, NUMBER, INTEGER, INT, SMALLINT, FLOAT, DOUBLE PRECISION, and REAL are all stored in the NUMBER datatype. The language or product you're using to access Oracle data may support these datatypes, but they're all stored in a NUMBER datatype column.

Date Datatype

As with the NUMERIC datatype, Oracle stores all dates and times in a standard internal format. The standard Oracle date format for input takes the form of DD-MON-YY HH:MI:SS, where DD represents up to two digits for the day of the month, MON is a three-character abbreviation for the month, YY is a two-digit representation of the year, and HH, MI, and SS are two-digit representations of hours, minutes, and seconds, respectively. If you don't specify any time values, their default values are all zeros in the internal storage.

You can change the format you use for inserting dates for an instance by changing the NLS_DATE_FORMAT parameter for the instance. You can do this for a session by using the ALTER SESSION SQL command or for a specific value by using parameters with the TO_DATE expression in your SQL statement.

If you use the default date format, Oracle will assume that the dates you enter are in the 1900s and store them as such. To prevent Year 2000 (Y2K) problems, you should make sure that you use one of the format options mentioned in the sidebar "Y2K Issues and Oracle" to change the default date format.

Oracle SQL supports date arithmetic in which integers represent days and fractions represent the fractional component represented by hours, minutes, and seconds. For example, adding .5 to a date value results in a date and time combination 12 hours later than the initial value. Some examples of date arithmetic are:

```
12-DEC-99 + 10 = 22-DEC-99
31-DEC-1999:23:59:59 + .25 = 1-JAN-2000:5:59:59
```

Y2K Issues and Oracle

At the time of this writing, the problems anticipated in dealing with date values as the century turned are over a year behind us. However, because you may still be working with databases that contain data entered in the last century, we are continuing to include this sidebar in the second edition of this book.

Y2K isn't a problem for the Oracle database itself because Oracle's internal date format stores all dates in a format that can accept dates from 4712 BC to 4712 AD.

However, Oracle uses a date format to specify how dates are input to and output from this internal date format. You set the format with the `NLS_DATE_FORMAT` parameter in the `INIT.ORA` or `SPFILE` file for an instance or with the `ALTER SESSION` command for an individual session.

If the date format isn't set to recognize centuries, the Oracle database might not properly store the dates entered. For instance, the default date format for Oracle is `DD-MM-YY`. Using this default date format, the value "01-01-00" would be recognized as January 1, 1900 and would be stored accordingly.

Oracle has a date format that's ideal for handling Y2K issues: the `RR` year format. The `RR` format automatically assigns two-digit years from 50 to 99 to the 1900s, and two-digit years from 00 to 49 to the 2000s. The `RR` format gives users the option of entering dates with only two digits, while still properly assigning those digits at least until the year 2050 (when it definitely won't be our problem any more!).

Some application developers store date values in Oracle databases as character strings. These developers will have to correct their own work to prevent any potential Y2K problems, because there is no way for Oracle to know that a character string is *supposed* to be a date.

If you're reading this page in the year 2002 or beyond, you won't have to worry about this issue for at least 40 years or so—but it helps to be prepared anyway.

Other Datatypes

Aside from the basic character, number, and date datatypes, Oracle supports a number of specialized datatypes:

RAW and LONG RAW

Normally, your Oracle database not only stores data but also interprets it. When data is requested or exported from the database, the Oracle database sometimes massages the requested data. For instance, when you dump the values from a `NUMBER` column, the values written to the dump file are the representations of the numbers, not the internally stored numbers.

The RAW and LONG RAW datatypes circumvent any interpretation on the part of the Oracle database. When you specify one of these datatypes, Oracle will store the data as the exact series of bits presented to it. The RAW datatypes typically store objects with their own internal format, such as bitmaps. A RAW datatype can hold 2 KB, while a LONG RAW datatype can hold 2 GB.

ROWID

The ROWID is a special type of column known as a *pseudocolumn*. The ROWID pseudocolumn can be accessed just like a column in a SQL SELECT statement. There is a ROWID pseudocolumn for every row in an Oracle database. The ROWID represents the specific address of a particular row. The ROWID pseudocolumn is defined with a ROWID datatype.

The ROWID relates to a specific location on a disk drive. Because of this, the ROWID is the fastest way to retrieve an individual row. However, the ROWID for a row can change as the result of dumping and reloading the database. For this reason, we don't recommend using the value for the ROWID pseudocolumn across transaction lines. For example, there is no reason to store a reference to the ROWID of a row once you've finished using the row in your current application.

You cannot set the value of the standard ROWID pseudocolumn with any SQL statement.

The format of the ROWID pseudocolumn changed with Oracle8. Beginning with Oracle8, the ROWID includes an identifier that points to the database object number in addition to the identifiers that point to the datafile, block, and row. You can parse the value returned from the ROWID pseudocolumn to understand the physical storage of rows in your Oracle database.

You can define a column or variable with a ROWID datatype, but Oracle doesn't guarantee that any value placed in this column or variable is a valid ROWID.

LOB

A LOB, or large object datatype, can store up to 4 GB of information. LOBs come in three varieties:

- CLOB, which can only store character data
- NCLOB, which stores National Language character set data
- BLOB, which stores data as binary information

You can designate that a LOB should store its data within the Oracle database or that it should point to an external file that contains the data.

LOBs can participate in transactions. Selecting a LOB datatype from Oracle will return a pointer to the LOB. You must use either the DBMS_LOB PL/SQL built-in package or the OCI interface to actually manipulate the data in a LOB.

To facilitate the conversion of LONG datatypes to LOBs, Oracle9i includes support for LOBs in most functions that support LONGs, as well as a new option to the ALTER TABLE statement that allows the automatic migration of LONG datatypes to LOBs.

BFILE

The BFILE datatype acts as a pointer to a file stored outside of the Oracle database. Because of this fact, columns or variables with BFILE datatypes don't participate in transactions, and the data stored in these columns is available only for reading. The file size limitations of the underlying operating system limit the amount of data in a BFILE.

XMLType

As part of its support for XML, Oracle9i includes a new datatype called XMLType. A column defined as this type of data will store an XML document in a character LOB column. There are built-in functions that allow you to extract individual nodes from the document, and you can also build indexes on any particular node in the XMLType document.

User-defined data

Oracle8 and later versions allow users to define their own complex datatypes, which are created as combinations of the basic Oracle datatypes previously discussed. These versions of Oracle also allow users to create objects composed of both basic datatypes and user-defined datatypes. For more information about objects within Oracle, see Chapter 13.

AnyType, AnyData, AnyDataSet

Oracle9i includes three new datatypes that can be used to explicitly define data structures that exist outside the realm of existing datatypes. Each of these datatypes must be defined with program units that let Oracle9i know how to process any specific implementation of these types.

Type Conversion

Oracle automatically converts some datatypes to other datatypes, depending on the SQL syntax in which the value occurs.

When you assign a character value to a numeric datatype, Oracle performs an implicit conversion of the ASCII value represented by the character string into a number. For instance, assigning a character value such as 10 to a NUMBER column results in an automatic data conversion.

If you attempt to assign an alphabetic value to a numeric datatype, you will end up with an unexpected (and invalid) numeric value, so you should make sure that you're assigning values appropriately.

You can also perform explicit conversions on data, using a variety of conversion functions available with Oracle. Explicit data conversions are better to use if a conversion is called for, since they document the conversion and avoid the possibility of going unnoticed, as implicit conversions sometimes do.

Concatenation and Comparisons

The concatenation operator for Oracle SQL on most platforms is two vertical lines (`||`). Concatenation is performed with two character values. Oracle's automatic type conversion allows you to seemingly concatenate two numeric values. If NUM1 is a numeric column with a value of 1, NUM2 is a numeric column with a value of 2, and NUM3 is a numeric column with a value of 3, the following expressions are TRUE:

```
NUM1 || NUM2 || NUM3 = "123"  
NUM1 || NUM2 + NUM3 = "15" (12 + 3)  
NUM1 + NUM2 || NUM3 = "33" (1+ 2 || 3)
```

The result for each of these expressions is a character string, but that character string can be automatically converted back to a numeric column for further calculations.

Comparisons between values of the same datatype work as you would expect. For example, a date that occurs later in time is larger than an earlier date, and 0 or any positive number is larger than any negative number. You can use relational operators to compare numeric values or date values. For character values, comparisons of single characters are based on the underlying code pages for the characters. For multicharacter strings, comparisons are made until the first character that differs between the two strings appears. The result of the comparison between these two characters is the result of the overall comparison.

If two character strings of different lengths are compared, Oracle uses two different types of comparison semantics: *blank-padded comparisons* and *nonpadded comparisons*. For a blank-padded comparison, the shorter string is padded with blanks and the comparison operates as previously described. For nonpadded comparisons, if both strings are identical for the length of the shorter string, the shorter string is identified as smaller. For example, in a blank-padded comparison the string "A " (a capital A followed by a blank) and the string "A" (a capital A by itself) would be seen as equal, because the second value would be padded with a blank. In a nonpadded comparison, the second string would be identified as smaller because it is shorter than the first string. Nonpadded comparisons are used for comparisons in which one or both of the values are VARCHAR2 or NVARCHAR2 datatypes, while blank-padded comparisons are used when neither of the values is one of these datatypes.

NULLs

The NULL value is one of the key features of the relational database. The NULL, in fact, doesn't represent any value at all—it represents the lack of a value. When you create a column for a table that must have a value, you specify it as NOT NULL, meaning that it cannot contain a NULL value. If you try to write a row to a database table that doesn't assign a value to a NOT NULL column, Oracle will return an error.

You can assign NULL as a value for any datatype. The NULL value introduces what is called *three-state logic* to your SQL operators. A normal comparison has only two states: TRUE or FALSE. If you're making a comparison that involves a NULL value, there are three logical states: TRUE, FALSE, and none of the above.

None of the following conditions are true for Column A if the column contains a NULL value:

A > 0
A < 0
A = 0
A != 0

The existence of three-state logic can be confusing for end users, but your data may frequently require you to allow for NULL values for columns or variables.

You have to test for the presence of a NULL value with the relational operator IS NULL, since a NULL value is not equal to 0 or any other value. Even the expression:

NULL = NULL

will always evaluate to FALSE, since a NULL value doesn't equal any other value.

Basic Data Structures

This section describes the three basic Oracle data structures: tables, views, and indexes.

Tables

The *table* is the basic data structure used in a relational database. A table is a collection of rows. Each *row* in a table contains one or more *columns*. If you're unfamiliar with relational databases, you can map a table to the concept of a file or database in a nonrelational database, just as you can map a row to the concept of a record in a nonrelational database.

Should You Use NULLs?

The idea of three-state logic may seem somewhat confusing, especially when you imagine your poor end users executing ad hoc queries and trying to account for a value that's neither TRUE nor FALSE. This prospect may concern you, so you may decide not to use NULL values at all.

We feel that NULLs have an appropriate use. The NULL value covers a very specific situation: a time when a column has not had a value assigned. The alternative to using a NULL is using a value with another meaning—such as 0 for numbers—and then trying to somehow determine whether that value has actually been assigned or simply exists as a replacement for NULL.

If you choose not to use NULL values, you're forcing a value to be assigned to a column for every row. You are, in effect, eliminating the possibility of having a column that doesn't require a value, as well as potentially assigning misleading values for certain columns. This situation can be misleading for end users and can lead to inaccurate results for summary actions such as AVG (average).

Avoiding NULL values simply replaces one problem—educating users or providing them with an interface that implicitly understands NULL values—with another set of problems, which can lead to a loss of data integrity.

With the Enterprise Editions of Oracle8 and beyond, you can purchase an option called Partitioning that, as the name implies, allows you to partition tables and indexes, which are described later in this chapter. Partitioning a data structure means that you can divide the information in the structure between multiple physical storage areas. A partitioned data structure is divided based on column values in the table. You can partition tables based on the range of column values in the table, the result of a hash function (which returns a value based on a calculation performed on the values in one or more columns), or a combination of the two. With Oracle9i you can also use a list of values to define a partition, which can be particularly useful in a data warehouse environment.

Oracle is smart enough to take advantage of partitions to improve performance in two ways:

- Oracle won't bother to access partitions that won't contain any data to satisfy the query.
- If all the data in a partition satisfies a part of the WHERE clause for the query, Oracle simply selects all the rows for the partition without bothering to evaluate the clause for each row.

Partitioning tables can also be useful in a data warehouse, in which data can be partitioned based on the time period it spans.

You can perform all maintenance operations, such as backup, recovery, and loading, on a single partition. This flexibility makes it possible to handle extremely large data structures while still performing those maintenance operations in a reasonable amount of time. In addition, if you must recover one partition in a table for some reason, the other partitions in the table can remain online during the recovery operation.

If you've been working with other databases that don't offer the same type of partitioning, you may have tried to implement a similar functionality by dividing a table into several separate tables and then using a UNION SQL command to view the data in several tables at once. Partitioned tables give you all the advantages of having several identical tables joined by a UNION command without the complexity that implementation requires.

To maximize the benefits of partitioning, it sometimes makes sense to partition a table and an index identically so that both the table partition and the index partition map to the same set of rows. You can automatically implement this type of partitioning, which is called *equipartitioning*, by specifying an index for a partitioned table as a LOCAL index.

For more details about the structure and limitations associated with partitioned tables, refer to your Oracle documentation.

One more enhancement in Oracle9i is the ability to define external tables. As its name implies, the data for an external table is stored outside the database, typically in a flat file. The external table is read-only; you cannot update the data it contains. The external table is good for loading and unloading data to files from a database, among other purposes.

Views

A *view* is an Oracle data structure constructed with a SQL statement. The SQL statement is stored in the database. When you use a view in a query, the stored query is executed and the base table data is returned to the user. Views do not contain data, but represent ways to look at the base table data in the way the query specifies.

You can use a view for several purposes:

- To simplify access to data stored in multiple tables
- To implement specific security for the data in a table (e.g., by creating a view that includes a WHERE clause that limits the data you can access through the view)
- To isolate an application from the specific structure of the underlying tables
-

以上内容仅为本文档的试下载部分，为可阅读页数的一半内容。如要下载或阅读全文，请访问：<https://d.book118.com/295300043220011100>