Data definition language (DDL) and Data Manipulation Language (DML)

Introduction

Tables are the primary objects in a relational database because they are where the data is stored. However, there are several secondary objects which perform supporting functions. SQL provides a special set of commands for maintaining these objects known as data definition language (DDL). For each type of object there are three maintenance operations that can be performed:

- CREATE
- ALTER
- DROP

Whereas the CREATE command defines the object and allocates space for it in the database, the ALTER command changes the definition of an existing object. This could be done for example to extend the length of text string that can be stored in a particular column. The DROP command removes the object and any contents from the database.

This set of notes goes through the main types of database object and describes their purpose. Examples of the DDL statements used to maintain the objects are also provided.

Datatypes

When defining a table, the designer must first decide what kind of data will be stored in each column. The datatype of the column affects low-level implementation details such as how much storage space is allocated to each data item and the operations that can be performed on the data. The datatypes used in a relational database correspond closely to those used in other programming languages; however they are not defined by any agreed standard, and so there are slight variations between different database platforms (Oracle, SQL Server, etc.). The datatypes most commonly used in Oracle are shown in the table below.

Datatype	Used for	Example
NUMBER(n)	Integer values with up to n digits	-10, 0, 1024
NUMBER(n, m)	Floting point values of up to n digits including m decimal places	3.14159, -10.5
VARCHAR2(n)	Variable character string of maximum length n	'Edinburgh'
DATE	A combined date and time value	10-JAN-2011 12:24:00

Like other database platforms, Oracle provides a range of other datatypes that are suitable for more specific requirements, or which improve the storage efficiency. You can find details in the Oracle documentation.

DDL: Tables

In its simplest form, the definition of a table is simply a list of the columns that the table contains with their respective datatypes. The following statement could therefore be used to create the table STUDENT:

```
CREATE TABLE student (
    matric_no     VARCHAR2(8),
    first_name    VARCHAR2(20),
    last_name     VARCHAR2(20),
    date_of_birth DATE
);
```

Notice that each column definition consists of the column name and datatype and that the list is separated by commas. When writing DDL statements, it is useful to lay them out like the example so that they can be read easily. This helps to avoid simple mistakes like forgetting the comma between two columns or missing out the final bracket.

In this example, we can store first and last names of up to 20 characters each. If we discover that this is not sufficient, the table definition can be altered using a statement like the following:

```
ALTER TABLE student MODIFY last_name VARCHAR2(25);
```

The ALTER TABLE statement can be used to make other changes to the table definition besides changing column datatypes. There are certain occasions where this is useful, and some of them will be illustrated in this week's practical exercises. However, in many cases it is quicker to remove the table and create it again with an updated definition. A table can be removed using a statement such as:

```
DROP TABLE student;
```

Each of these statements can be used with further optional clauses which modify their behaviour. A full description of their syntax can be found in the Oracle SQL Language Reference.

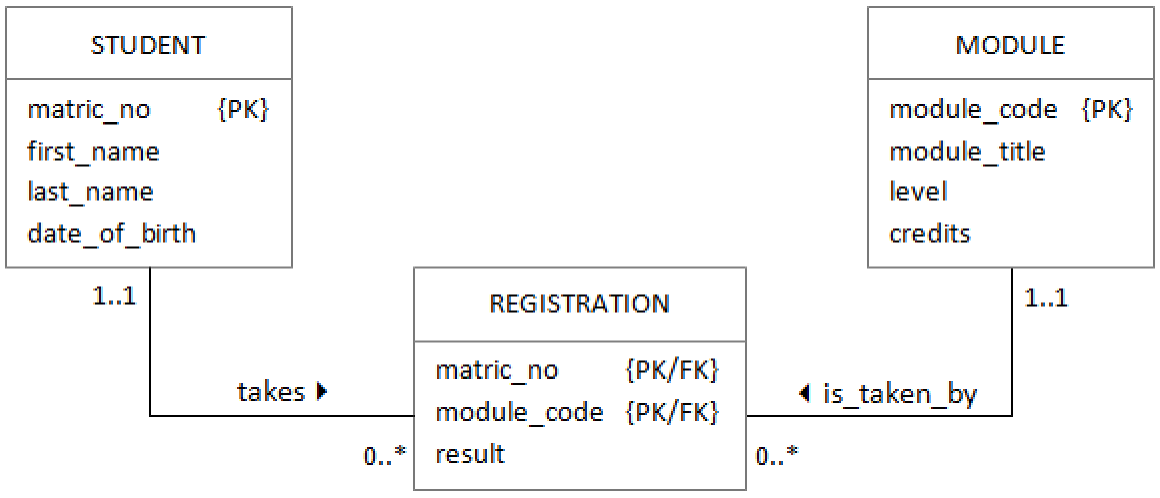
DDL: Constraints

A constraint is a restriction on the data that can be stored in a table. Like the datatype of a column, constraints form part of the definition of a table and so appear in the CREATE TABLE statement. Essentially there are two types of constraint: integrity constraints which control the structural aspects of the data, and data constraints which define the domain of a column. These correspond to the two types of business rule discussed in week 5.

In most cases, a constraint refers only to a single column in the table and can be included as part of the column definition in the CREATE TABLE statement. These are referred to as *column constraints*. Sometimes, however, a constraint needs to reference more than one column - an example would be a composite primary key - and then it is referred to as a *table constraint*. The definition of a table constraint appears at the end of the list of columns in the CREATE TABLE statement. The following discussion provides examples of both types of constraint.

Integrity constraints

Constraints are used to maintain entity and referential integrity. That is, they define primary and foreign keys. We will use the following schema to show how this is done:



Notice that REGISTRATION is a link table which resolves the *:* relationship between STUDENT and MODULE.

Because the primary key of STUDENT consists of a single column, it can simply be identified as such as part of the column definition:

```
CREATE TABLE student (
    matric_no     VARCHAR2(8) PRIMARY KEY,
    first_name    VARCHAR2(20),
    last_name     VARCHAR2(20),
    date_of_birth DATE
);
```

In contrast, the primary key of REGISTRATION includes two columns and must therefore be defined as a table constraint:

```
CREATE TABLE registration (
    matric_no    VARCHAR2(8),
    module_code  VARCHAR2(8),
    result       NUMBER(3),
    PRIMARY KEY (matric_no, module_code)
);
```

Notice that the table constraint definition appears as part of the column list. That is, it is separated from the last column definition with a comma and appears inside the brackets.

The columns *matric_no* and *module_code* in the REGISTRATION table are also foreign keys which correspond to primary keys in the other two tables. They also need to be identified in the CREATE TABLE statement:

```
CREATE TABLE registration (
    matric_no    VARCHAR2(8) REFERENCES student(matric_no),
    module_code  VARCHAR2(8) REFERENCES module(module_code),
    result       NUMBER(3),
    PRIMARY KEY (matric_no, module_code)
);
```

Notice that the relationship is defined by identifying the foreign key. This means that there is no mention of the relationship in the related table. After the REFERENCES keyword, the names of both the related table and the primary key column are required.

Data constraints

There are three kinds of data constraint:

- NOT NULL: the column must have a value
- UNIQUE: the column value must be unique in the table
- CHECK: the column value conforms to an arbitrary condition

NOT NULL constraints can only apply to single columns and therefore always appear as column constraints:

```
CREATE TABLE module (
    module_code     VARCHAR2(8) PRIMARY KEY,
    module_title    VARCHAR2(40) NOT NULL,
    level           NUMBER(2) NOT NULL,
    credits         NUMBER(2) NOT NULL
);
```

In the example above, a NOT NULL constraint is not required for *module_code* because it is already implied by the PRIMARY KEY constraint.

When the UNIQUE constraint is applied to a column, it essentially defines an alternative key for the table. Although looking through the University's list of modules will show that there are in fact modules in different schools with the same name, it might be more sensible to insist that module titles are also unique:

```
CREATE TABLE module (
    module_code     VARCHAR2(8) PRIMARY KEY,
    module_title    VARCHAR2(40) NOT NULL UNIQUE,
    level           NUMBER(2) NOT NULL,
    credits         NUMBER(2) NOT NULL
);
```

Notice that the *module_title* column now has two constraints which simple appear one after the other in the column definition line. The order in which they are mentioned is not important.

The CHECK constraint allows the designer to be very specific about the domain of a column by applying an arbitrary condition to the data. In Scotland, the level of a module is defined by the Scottish Credit and Qualifications Framework (SCQF). Levels 7, 8, 9 and 10 correspond to the four years of an undergraduate degree, and level 11 corresponds to Masters programmes. We could build this into the definition of the table as shown below:

```
CREATE TABLE module (
    module_code     VARCHAR2(8) PRIMARY KEY,
    module_title    VARCHAR2(40) NOT NULL UNIQUE,
    level           NUMBER(2) NOT NULL
                      CHECK (level BETWEEN 7 AND 11),
    credits         NUMBER(2) NOT NULL
);
```

Notice that the condition in the CHECK constraint has the same form as a condition in the WHERE clause of a query.

The standard size of a module is 20 credits; however, at Masters level it is possible to have half modules worth only 10 credits. This could also be made part of the table definition, but because two columns are referenced it would have to be a table constraint as shown below:

```
CREATE TABLE module (
    module_code     VARCHAR2(8) PRIMARY KEY,
    module_title    VARCHAR2(40) NOT NULL UNIQUE,
    level           NUMBER(2) NOT NULL
                      CHECK (level BETWEEN 7 AND 11),
    credits         NUMBER(2) NOT NULL,
    CHECK (credits = 20 OR (level = 11 AND credits = 10))
);
```

See the Oracle documentation for further examples.

Constraint independence

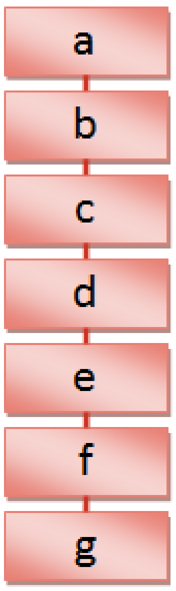
The examples above illustrate how to define constraints at the same time as the associated table. However they can be created and dropped independently by using the ALTER TABLE statement. This can be useful when creating a new schema to avoid problems when you come to insert data into the tables, or when there are parallel relationships between tables. Say for example you wanted to create the REGISTRATION table, insert some data, and only then enable the foreign key constraints, you could use the following piece of DDL:

```
ALTER TABLE registration
ADD CONSTRAINT registration_fk1
FOREIGN KEY (matric_no) REFERENCES student(matric_no);
```

Notice that in this example, we give the constraint a name which follows the CONSTRAINT keyword. This is optional, and has been omitted from the earlier examples; however, it is needed if you want to be able to drop a constraint independently of its table:

```
ALTER TABLE registration
DROP CONSTRAINT registration_fk1;
```

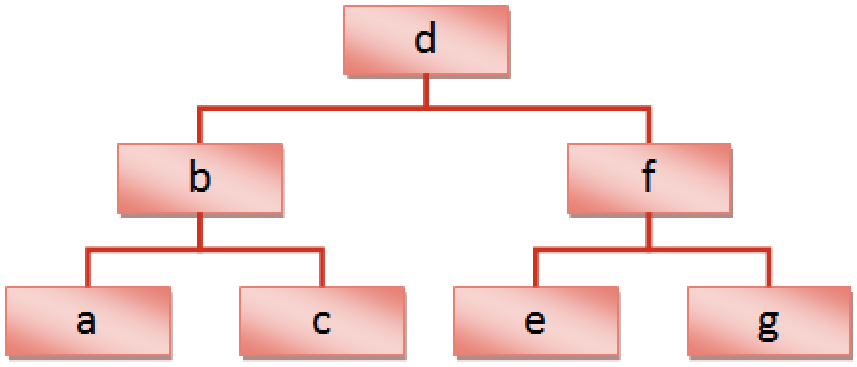
DDL: Indexes



As rows are added to a table they are stored in a relatively simple list structure in the database file in chronological order. Because there is no intrinsic ordering, retrieving a subset of rows using a query means looking at every row in the table, an operation known as a full table scan. While a table is only a few hundred rows in size, this is reasonable quick, but as more rows are added, the time required to perform a full table scan can become significant and performance can drop.

Fortunately there are much more efficient ways of storing data for quick retrieval. Although the details of their implementation are largely beyond the scope of this module, a simple illustration should be enough to convey the main principle.

The graphic on the right represents seven rows in a database table, and the labels on the boxes represent the values of the primary key. If we are trying to retrieve a row using the primary key, we know that as soon as we find the value we are looking for we can stop because the primary key is unique. If we are searching for row ***g***, we can see that starting from the top of the table we will need to perform seven comparison operations before we find it.



If we take that same data and rearrange it into a structure called a balanced binary tree as shown on the left, we see that the same operation will only take three comparisons. Just by changing the structure we have reduced the effort required by more than 50%.

The tree structure shown here could be used to improve the retrieval of data using the primary key, but often queries based on other columns are also common. When searching for modules in the module catalogue for example we could search by code, title or SCQF level. To execute these queries with optimum efficiency each one would require a slightly different tree structure. It is clearly not practical to maintain several copies of a table each of which is optimised for a particular query, but a less resource-hungry compromise is to maintain an optimised copy of *certain columns* from the table. Specifically, these would have to be columns which are known to be commonly used in queries. These parallel structures are known as indexes because they can be searched quickly and the result of the index search provides a link back to the full row in the actual table.

Most database platforms provide some indexes by default. Oracle, for example, automatically creates indexes when a PRIMARY KEY or UNIQUE constraint is defined. The database designer can define other indexes on one or more columns of a table as required. Care needs to be taken because a poorly designed index can simply take up additional space in the database without improving performance at all.

The performance of a regularly-used query can be improved by creating an index on the columns that are used to select the rows - that is to say, those used in the WHERE clause. For example, as part of a development project, we might find that student data is regularly queried using the combination of first and last names. We could create an index on the STUDENT table as shown below:

```
CREATE INDEX student_name_ind ON student(first_name,
last_name);
```

Further detail on the use of indexes is beyond the scope of this module; however, if you are interested you can visit the Oracle documentation. The options are extensive because an index is a low-level object which depends heavily on the actual database implementation. Design of effective indexes is often the responsibility of the DBA who has the best knowledge of the physical design.

Because indexes are secondary objects - their existence depends on the existence of the related tables - they are automatically dropped when the table is dropped. If necessary, though, they can be altered and dropped independently using ALTER INDEX and DROP INDEX statements.

DDL: Views

Thinking back to the three-level ANSI-SPARC architecture, we have said that the external level is a collection of user views of the same data. These views can differ in the subset of entities they contain, the set of attributes required for a particular entity and the names used for entities and attributes.

Often as part of a development project, there is a requirement to provide direct end-user access to data. For example, a senior manager may want to load data into a spreadsheet on a regular basis in order to perform statistical analyses. It is important in such situations that the data structures offered by the database are easily understood by the end user. Where the internal model does not correspond exactly to the end user's requirements, an object called a *view* can be used to modify the presentation appropriately.

Unlike an index, a view does not usually take up additional space in the database. A view can be though of more as a stored query which is run whenever the view is accessed. We have already seen methods for manipulating the appearance of the results of a query. These include column aliases, calculated fields, date formats and so on, and it is these same methods that are used in the creation of a view. Once a view has been created, it can be referenced in a query in exactly the same way as a table.

As an example, assume that the university management requires data on student performance consisting of the student identifier, the SCQF level and the student's average result at that level. In addition, a target result for each level is required along with the deviation from that target. The following statement could be used to create a view called PERFORMANCE:

```
CREATE VIEW performance AS
SELECT  s.matric_no AS "Student",
        m.level AS "SCQF level",
        AVG(r.result) AS "Result",
        DECODE(m.level, 7,  56,
                        8,  60,
                        9,  65,
                        10, 65,
                        65) as "KPI",
        AVG(r.result) -
        DECODE(m.level, 7,  56,
                        8,  60,
                        9,  65,
                        10, 65,
                            65) as "Deviation"
FROM    student s
        JOIN registration r ON s.matric_no = r.matric_no
        JOIN module m ON r.module_code = m.module_code
ORDER BY s.matric_no, m.level
```

From the end user's perspective, the view will look like a table with the following structure:

PERFORMANCE
Student
SCQF level
Result
KPI
Deviation

DDL: Sequences

Synthetic keys are very common in relational databases. They are particularly useful

- when there is no combination of attributes that can guarantee a unique value
- to avoid composite primary/foreign keys
- if there is a risk that non-unique values will appear in the future
- if there is a risk that the values of a natural key might change

Most relational database platforms provide a mechanism for automatically generating new unique values for synthetic keys. In Microsoft database systems this is done simply by activating a property of a column. Oracle however provides a special database object called a sequence which must be explicitly referenced when a new value is required. This approach is slightly more complex but provides greater flexibility.

A sequence for generating new matriculation numbers can be created using a statement such as the following which illustrates the most commonly used features:

```
CREATE SEQUENCE matric_seq
       START WITH 4000000
       INCREMENT BY 1
       NOCACHE
       NOCYCLE;
```

START WITH and INCREMENT BY are fairly self-explanatory. For busy systems, Oracle allows sequence values to be cached in memory for faster access. However this can lead to missing numbers in the sequence of values actually used. The example statement specifies that no values should be cached. It also specifies that the sequence should continue indefinitely with the NOCYCLE flag. The alternative would be to generate a repeating sequence of numbers.

Each Oracle sequence provides two attributes, NEXTVAL and CURRVAL. If MATRIC_SEQ.NEXTVAL is referenced in a data insertion statement, a new value is generated and the sequence is incremented. On the other hand, MATRIC_NO.CURRVAL gives the most recent value generated, but the sequence is not incremented when this attribute is referenced. This can be very useful, for example when inserting a series of related records in a second table. CURRVAL can be used to find the appropriate value to use for the foreign key.

Sequences can be dropped using the DROP SEQUENCE statement, and they can be altered with ALTER SEQUENCE. Altering a sequence can be important for example when you first import some data into a table that has a synthetic key. It will be important to update the START WITH value so that the next row inserted into the table takes the next value in series.

The complete description of the CREATE SEQUENCE statement can be found in the Oracle documentation. The syntax to use when using sequence values in newly inserted rows of data will be described next week under the topic of data maintenance.

DDL: Triggers

SQL is a declarative language. That is to say that each statement is interpreted in isolation so that it is not possible to construct a flow of control as is possible in Java or C#. However, several database manufacturers provide proprietary extensions to standard SQL which do allow procedural control. The Oracle extension called PL/SQL allows the creation of a range of procedural code objects which can be stored in the database and accessed in a variety of ways. This can be useful, for example, to encapsulate the business logic of an application so that the same functionality can be delivered via several user interfaces without a lot of recoding.

For the most part, PL/SQL is beyond the scope of this module; however, there is one specific example that replicates the behaviour of the Microsoft synthetic key columns mentioned in the previous section. In SQL Server and MySQL, a new value is automatically generated for a column that is identified as a synthetic key. This is very convenient, and the Oracle alternative is quite cumbersome. The same behaviour can be produced, however, by creating a PL/SQL object called a *trigger*. Triggers - as the name suggests - respond to specified events such as the creation of a new row in a table. When the event occurs, the code in the trigger is executed in the context of the row involved in the event. The example below inserts the next value from the MATRIC_SEQ into the *matric_no*column of a new row in the STUDENT table.

```
CREATE TRIGGER matric_trig
BEFORE INSERT ON student
FOR EACH ROW BEGIN
    SELECT matric_seq.NEXTVAL INTO :new.matric_no FROM dual;
END;
/
```

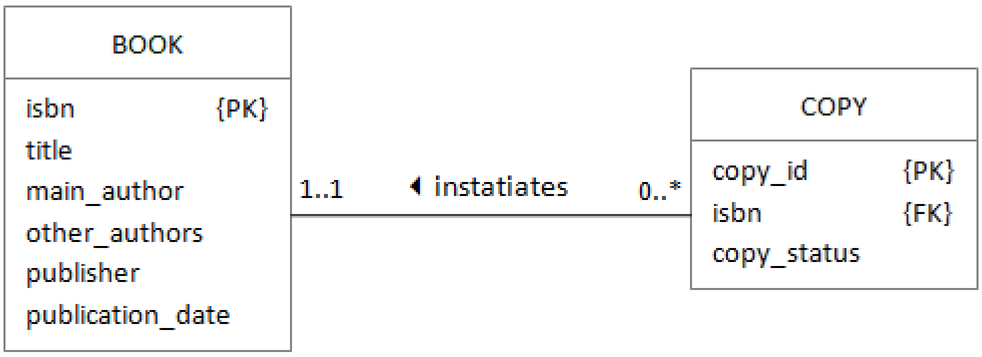
Some points of interest about the CREATE TRIGGER statement are:

- ***:new*** is a special variable that refers to the row currently being created
- ***dual*** is a dummy table provided in every Oracle database. It is used when the data being SELECTed does not actually come from a table. It could be used for example when displaying the current system date
- The final forward slash is responsible for running the statement. The semicolon just marks the end of the PL/SQL code block

The syntax of PL/SQL is very different from standard SQL. If you are interested in knowing more, please consult the Oracle documentation. For the purposes of this module you will not need to know anything beyond the simple example above.

DML: Insert

Inserting a new row into a table is done using the SQL INSERT statement. There are two forms that you need to know about, and they are both covered below. For the purposes of the examples, we will be using the following pair of tables which might be found in a larger library system:



Inserting values directly

Often, you will already have a set of values that you want to use to create a new row in a table. In this case, you simply need to tell the database which value goes in which column. This is done by listing the columns first, and then by providing the values in the same order as shown in the statement below:

```
INSERT INTO book (isbn,
                  title,
                  main_author,
                  other_authors,
                  publisher,
                  publication_date)
          VALUES ('978-0321210258',
                  'Database Systems: A Practical Approach to
Design,
                   Implementation and Management',
                  'Connolly, T.',
                  'Begg, C.',
                  'Addison Wesley',
                  '24-FEB-2009');
```

This book happens to have two authors, but that is not always the case. For a book with only one author, the *other_authors* column would be NULL, and there are two ways to do this. You can either list all the columns as before, and use the keyword NULL in place of the *other_authors*value, or you can just omit *other_authors* from the column list. The following statement is therefore also valid:

```
INSERT INTO book (isbn,
                  title,
                  main_author,
                  publisher,
                  publication_date)
          VALUES ('978-1111969592',
                  'Database Systems',
                  'Coronel, C.',
                  'South-Western College Publishing',
                  '31-JAN-2012');
```

Inserting rows using SELECT

Sometimes the values that you need for a new row are already stored in the database somewhere. For example, if you wanted to add a copy of Connolly and Begg to the library, you could find the ISBN number which is used as the primary key like this:

```
INSERT INTO copy (copy_id,
                  isbn,
                  copy_status)
           SELECT 1200,
                  isbn,
                  'On shelf'
           FROM   book
           WHERE  title = 'Database Systems: A Practical
Approach to
                           Design, Implementation and
Management';
```

Notice that in the SELECT clause there is only one column value that comes from the BOOK table. The other two items - 1200 and 'On shelf' - are literals. This means that they are ordinary values that do not depend on a variable or column in a table.

Referencing sequences in INSERT statements

The example above uses a literal value for the *copy_id*, but of course in a library with thousands of books, it is unlikely you would happen to know what value to use. In practice you would want to use the next available value from a sequence. To do this explicitly, you would need to reference the

NEXTVAL property of the sequence in the INSERT statement. The effect of this is to generate a new value ant to increment the sequence, and can be done as shown below assuming that there is a sequence called COPY_SEQ:

```
INSERT INTO copy (copy_id,
                  isbn,
                  copy_status)
           SELECT copy_seq.nextval,
                  isbn,
                  'On shelf'
           FROM   book
           WHERE  title = 'Database Systems: A Practical
Approach to
                           Design, Implementation and
Management';
```

Occasionally, it can also be useful to find out what value was last generated by a sequence without incrementing it. This can be useful for example if you have just generated a new primary key in one table, and you then have to insert a set of records in a related table. For the related records, you would need to use the recently-generated value as a foreign key. To do this, you simply reference the CURRVAL property rather than NEXTVAL.

If you have created a trigger that automatically inserts a new value when a new record is created, you should leave the associated column out of the INSERT statement as if you were leaving it as NULL. The trigger will insert the key value before the operation completes. In this case, your INSERT statement would look like this:

```
INSERT INTO copy (isbn,
                  copy_status)
           SELECT isbn,
                  'On shelf'
           FROM   book
           WHERE  title = 'Database Systems: A Practical
Approach to
                           Design, Implementation and
Management';
```

Finally, if you have specified a default value for a column in your table definition, you can also omit this column from the INSERT statement and the specified value will automatically be used. This could be done for example for the *status* column in the COPY table. By default, copies are available on the shelf when first recorded, and so the statement would become:

```
INSERT INTO copy (isbn)
           SELECT isbn
           FROM   book
           WHERE  title = 'Database Systems: A Practical
Approach to
                           Design, Implementation and
Management';
```

DML: Update

There are any number of reasons why you might need to update the values stored in a record in your database. Whether you are actually changing a value or filling in a column that was previously NULL, the format of the UPDATE statement is always the same. It works in a very similar way to an ordinary query in that you need to locate the row or rows you want to change using a WHERE clause. So for example, if we want to show that someone has borrowed the new copy of Connolly and Begg from our library we could use the following statement:

```
UPDATE copy
SET    status = 'On loan'
WHERE  copy_id = 1200;
```

Because you need to be certain that your update statement is correct before actually changing any data, it is often a good idea to write a query first which uses the same WHERE clause. When your results show only the records that you want to update, exchange the SELECT clause for the UPDATE and SET clauses.

As you might expect, there are some variations on this basic form of the UPDATE statement. If, for example, you want to update more than one column in the target records, you can do this in a single statement by specifying all of the value changes in the SET clause. Using this method, we could change the authors' names on a book to include the full first name rather than just the initial:

```
UPDATE book
SET    main_author = 'Connolly, Thomas',
       other_authors = 'Begg, Caroline'
WHERE  isbn = '978-0321210258';
```

Another variation is to use a subquery to select the new value from the database. Say for example that a librarian had mistakenly entered the isbn for "Database Systems" by Coronel instead of that for Connolly and Begg when creating a new COPY record. This could be corrected using the following statement:

```
UPDATE copy
SET    isbn = (SELECT isbn
               FROM   book
               WHERE  title LIKE 'Database Systems%'
               AND    main_author LIKE 'CONNOLLY%')
WHERE  copy_id = 1200;
```

DML: Delete

Deleting a row from a table is similar to updating a row in that the require record is also identified using a WHERE clause. The same warnings therefore apply - make sure that you are deleting the correct row(s) before you actually make the change. If a copy of a book is lost, for example, we might want to delete it from the database. This could be done using the following statement:

```
DELETE FROM copy
WHERE  copy_id = 1200;
```

The FROM keyword is actually optional, but it makes the DELETE statement a little more natural.

Date and time values

The system time

While simple datatypes like integers and text strings behave as you would expect them to, dates are particularly difficult to manipulate. All database platforms provide their own set of functions for handling dates and times, but they are not particularly intuitive.

Date and time values are stored in an internal format which means nothing to the ordinary database user. Oracle, for example, stores dates as the time elapsed since 1 January, 4712BC. This is known as a Julian date format, and thus 1 February 4712 would be stored as 31 (equal to the number of days in January). Dates are counted from midnight, and periods of time less than a day are calculated as a fraction of a day. Thus 1200 on 1 February 4712 would be 31.5.

The Julian date format makes the comparison of dates and time very simple for the DBMS to calculate, but it is no good for representing dates for the user.

Another complication is that there is a huge range of formats for dates and times that are appropriate under different circumstances. For example, British dates are typically written in the form *DD/MM/YYYY* where *DD* represents the number of the day in the month,*MM* represents the number of the month in the year, and *YYYY* is the four-digit representation of the year. American dates on the other hand are typically written in the format *MM/DD/YYYY*. To interpret the date 1/2/2012 therefore, we need to know whether it is in British or American format since it could represent 1 February, or 2 January.

Proper handling of dates and times usually means converting from the internal date format to a text representation. Likewise, storing a date value usually means converting from a text representation to the internal date format.

This set of notes contains some practical examples that you may want to try for yourself. If so, you will need to create a temporary table:

```
CREATE TABLE date_tab (date_col DATE);
```

Check that the table has been created correctly using the command

```
desc date_tab
```

All computer systems maintain an internal clock that can be referenced in the programming language you are using. Oracle provides a *pseudocolumn* called SYSDATE that can be used in SQL statements. In SQL*Plus, type the following:

```
SELECT SYSDATE
FROM   dual;
```

There are two things to note:

- The table *dual* is a dummy table created automatically in Oracle databases. It gives you a table to use in your FROM clause in cases like this one where the data you want is not stored in a table.
- The displayed value contains day, month and year elements as expected.

What is not obvious from this last result is that the system date actually contains a time component as well, but it is not displayed by default. To see how this can lead to errors, we will store a record in our date_tab table, and compare it to the system date. Use the statement below:

```
INSERT INTO date_tab VALUES (SYSDATE);
```

Check that the record has been created properly using the query:

```
SELECT * from date_tab;
```

The displayed result should be self-explanatory. Now what will happen if we include a WHERE clause that restricts the results to those where the stored date is equal to today? Let’s try:

```
SELECT * from date_tab
WHERE date_code = SYSDATE;
```

We stored today's date; it's still the same day – why are there no results? The reason is that the small number of seconds that have elapsed since you created the record means that the stored value is different from the current SYSDATE because of the time component.

SYSDATE is often used as a default value when storing time-dependent entries in a table. It is important to remember therefore that the stored values include the time of storage as well as the date, or to explicitly truncate the value so that the time element is removed.

Date formatting

When a date value such as SYSDATE is displayed, Oracle implicitly converts the internal Julian format to a character string. The default display format as you have seen is DD-MON-YY. As noted earlier though, different date and time formats are required to suit different situations, so all database platforms provide a way to control the format in which the date/time value is presented.

In Oracle, the display of a date/time value makes explicit reference to the fact that there is a datatype conversion being performed. The general form of the formatting function is

```
TO_CHAR(date_value, required_format)
```

where *date_value* is either a stored value, or a system value like SYSDATE, and *required_format* is a pattern made up format codes. The pattern in *required_format* takes the form of a character string including single quotes. We can explicitly format SYSDATE and the value that we stored in the test table to reveal the time components that were previously hidden:

```
SELECT TO_CHAR(date_col,'DD-MON-YYYY HH:MI:SS'),
       TO_CHAR(SYSDATE, 'DD-MON-YYYY HH:MI:SS')
FROM date_tab;
```

If you run this query several times, you will notice how the SYSDATE changes while the stored value remains the same.

Format	Description	Example
DD	Day of the month	1 January = 1
FMDDTH	Ordinal day of the month	1 January = 1st
DAY	Name of the day	Monday
DY	Abbreviated day name	Mon
MM	Month number	January = 1
MONTH	Month name	January
MON	Abbreviated month name	Jan
YYYY	Four-digit year	2012
YY	Two-digit year	2012 = 12
HH	Hour (1-12)	1pm = 1
PM	Meridian indicator	1pm = PM
HH24	Hour (0-23)	1pm = 13
MI	Minutes past the hour	
SS	Seconds past the minute	

A comprehensive list of formatting elements can be found in the Oracle SQL Reference. A brief look at this list will tell you that there is a format for just about any date or time format you could need. The table below summarises some of the codes that you may find yourself using on a regular basis.

Comparing dates

Because datatype conversion and formatting is complex, working with dates can be a frustrating business. The usual rules apply though, and if you break the task down into smaller steps, you should find that you can deal with each step on its own quite easily. For example, imagine that you want to find the employees taken on before 1990. Using the Oracle HR schema, you would need to compare the value in the *hire_date* column of the employees table with a literal date value. You would therefore need to go through the following steps:

1. Decide on the exact literal date for comparison
2. Represent that date as a character string
3. Convert that string into an internal date value
4. Use the date value in the WHERE clause of a query

Going through these steps:

1. 'Before 1990' actually means 'before 1 Jan 1990'
2. A valid representation of that date is '01-JAN-1990'
3. To convert the string into a date, use the TO_DATE function with the appropriate format:

```
TO_DATE('01-JAN-1990', 'DD-MON-YYYY')
```

4. Construct a query:

```
SELECT employee_id
FROM   employees
WHERE  hire_date < TO_DATE('01-JAN-1990', 'DD-MON-
YYYY');
```

Try this query in SQL*Plus.

Oracle does provide one shortcut: because DD-MON-YYYY is the default date format, the format string can be left out in this case:

```
SELECT employee_id
FROM   employees
WHERE  hire_date < TO_DATE('01-JAN-1990');
```

Note the use of the standard arithmetic *less than* operator. Comparisons of this kind work intuitively enough. Another intuitive use of arithmetic operators is to add or subtract days from a given date value. The example below might be used to add 14 days to an invoice date to calculate the date that payment is due:

```
SELECT SYSDATE + 14 AS "Due date"
FROM   dual;
```

A date value can also be subtracted from a later date value to give the number of days between them; however, you must remember that the time component of the values will also be included. You will need to handle this explicitly in order to get an integer result.

Oracle provides many useful functions for manipulating date/time values, and going through each one explicitly would take a very long time. This is one of the most obvious examples of a situation where it is much more important to know where to find information about the topic when you need it rather than trying to memorise all of the details.

Referring back to the problem highlighted in the system time section, you can use the TRUNC function to remove the time element from SYSDATE:

```
SELECT *
FROM   date_tab
WHERE TRUNC(date_col, 'DAY') = TRUNC(SYSDATE, 'DAY');
```

To finish this topic, read through the information about the following functions in the Oracle SQL Reference:

- ADD_MONTHS
- EXTRACT
- LAST_DAY
- ROUND

The data dictionary

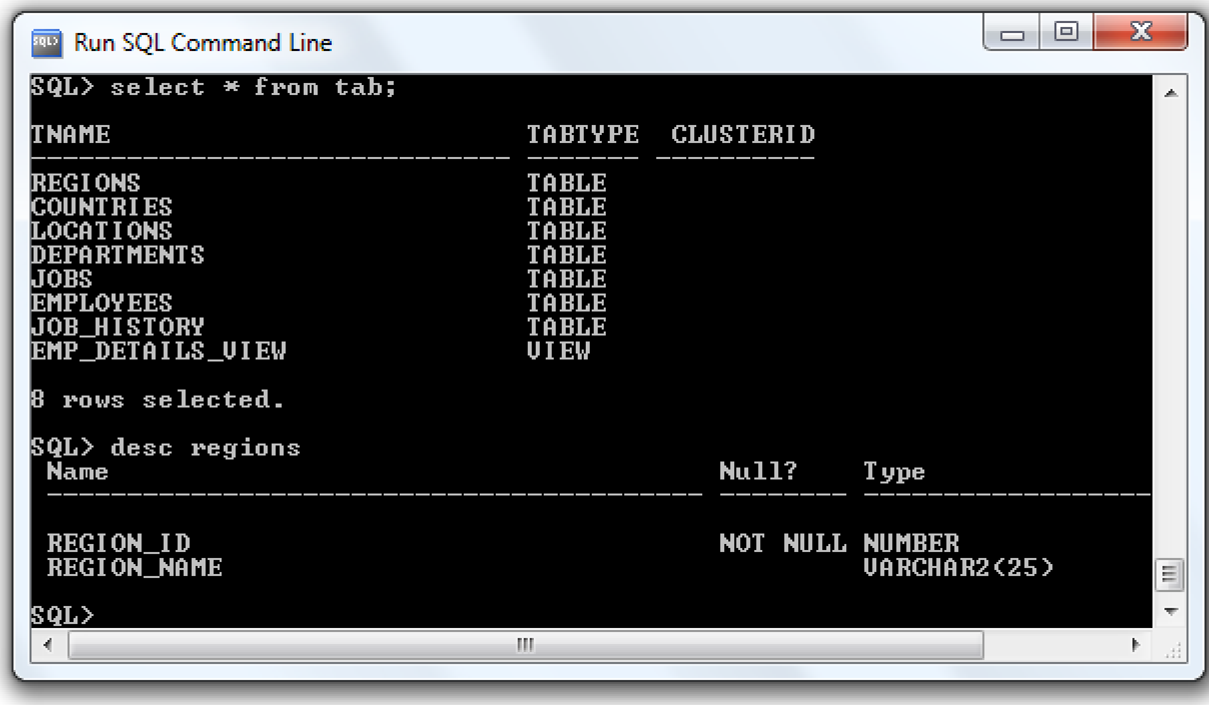
A database is designed to store information in a simple, readable format that is easy to query and maintain. It is only sensible, therefore, that the information about the tables and other objects in the database is stored in the same way. Because the information about schema objects constitutes a definitive reference for other database operations, it is known as the *data dictionary*. You have already seen this term used in a different sense in the context of the database design process, and the two uses of the term have no particular relationship with each other. You could say though, that the design data dictionary and the set of tables that store the schema details perform the same function at different stages in the database lifecycle: they both define the structure of the data model. You may also see the term *system catalog* used instead of *data dictionary* to refer to the schema tables.

The data dictionary tables are only updated by the DBMS where a DDL statement is processed, and they are not directly accessible to ordinary users. However, Oracle and other database platforms provide a series of views which allow users access to certain parts of the data dictionary. Although there are many such views, and a lot of the information they provide is about low-level storage and other administrative issues, there are some particularly useful ones that you should know about.

Oracle maintains a set of data dictionary views that start with the prefix USER_ to provide information about a user's own objects. The full list can be found in the Oracle documentation along with two other sets of views. Those starting with ALL_ provide information on all objects that the user has access to, while those starting DBA_ list all objects for all users. The DBA_ views may not be visible to ordinary users. Some particularly useful data dictionary objects are:

- USER_TABLES: Provides details about a user's own tables
- USER_CONSTRAINTS: Provides details about all constraints currently defined on the user's objects. This can be useful when checking that all constraints have been correctly created, or when you need to find the name of a constraint which needs to be dropped or altered
- USER_TRIGGERS: Provides details about triggers, including the trigger body
- ALL_USERS: Lists the user accounts in the database

As well as the views mentioned above that conform to the standard naming conventions, Oracle also provides another called TAB for ease of use. TAB lists the tables and views owner by the current user, and so provides a quick reminder of their names. This can be very useful when working with the command line, and can be used in conjunction with the DESCRIBE command which shows the columns in a particular table. The graphic below show the use of these two statements using the HR schema:



```
SQL> select * from tab;

TNAME                          TABTYPE  CLUSTERID
------------------------------ ------- ----------
REGIONS                        TABLE
COUNTRIES                      TABLE
LOCATIONS                      TABLE
DEPARTMENTS                    TABLE
JOBS                           TABLE
EMPLOYEES                      TABLE
JOB_HISTORY                    TABLE
EMP_DETAILS_VIEW               VIEW

8 rows selected.

SQL> desc regions
 Name                                      Null?    Type
 ----------------------------------------- -------- ------------------

 REGION_ID                                 NOT NULL NUMBER
 REGION_NAME                                        VARCHAR2(25)

SQL>
```