SQL Course

Introduction to Data Manipulation Language

- To describe the functions and capabilities that **DML** provides.
- To show which statements are included in **DML.**

As a relational database is a collection of tables with associated data it can be seen to be made up of two parts. The first part is the structure of the tables and their columns and the second part is the data that will be stored in these columns to form the database. SQL provides language facilities that enable the definition of the structure of the tables and their columns and language facilities for the maintenance and manipulation of the data that will be stored in the tables. In this module we are concerned with the manipulation and maintenance of the data that is stored in the database. The part of the SQL language that covers this area is know as the Data Manipulation Language or **DML** for short.
The **DML** is the subset of the SQL language which invoke actions from the DBMS to manipulate data in the database. The actions in the database are guaranteed by the DBMS to maintain the integrity of the data in the database. The application programs issue **DML** commands to change the contents of the database to reflect changes in the real world data. Using **DML** statements you can:

- Select data from a table, which can be specified precisely using clauses in the SQL statement.
- Insert data values into tables as rows.
- Update existing data values in rows.
- Delete certain rows from certain tables.

The **DML** statements all begin with one of the following verbs:

- **INSERT** - for adding rows.
- **UPDATE** - for changing existing values.
- **DELETE** - for deleting rows within the database.
- **SELECT** - for querying the database.

Note: The first three statements are concerned with populating the database, and will be covered in this section of the course. The fourth statement (Select) is concerned with manipulating the database, and will be covered in a later section of the course, **Database Retrieval**.

Because of 'Relational Closure' several of these commands can be nested together to form a single query or operation. You can modify data (**INSERT**, **UPDATE** or **DELETE**) in only one table per statement. However, in some systems, the modifications you make can be based on data in other tables, even those in other databases. You can actually pull values from one table into another, using a SQL **SELECT** statement within one of the three data modification commands given above. The data modification commands work on views as well as on tables, with some added restrictions.

SQL Course

Insert Statement

- To introduce the syntax used to add data to a table.
- To note any restrictions the SQL standard places on the use of the **INSERT** statement.
- To illustrate the use of the **INSERT** statement using examples.

- The **INSERT** statement adds new row(s) to a table.
- The **INSERT** statement can specify new values using the **VALUES** clause.
- If an **INSERT** statement only inserts some values in a row, then other values of that row are set to **NULL**.

To insert data into a table that you do not own you must use a qualified table name. The ANSI/ISO standard mandates unqualified column names in the column list, but many implementations allow qualified names.
The insert command has the form:

> **INSERT INTO** *tablename* [*column_list*] **VALUES** (*constant1, constant2, ...*) | **DEFAULT VALUES;**

If the column list is omitted then the **INSERT** statement will add a new row to the table, giving a value for every column in the row. In the **CREATE TABLE** statement the **DEFAULT** value is a value which will be automatically used by the database when a value is not specified by the **INSERT** statement for that particular column.
It is important that you type the data values in the same order as the column names in the original **CREATE TABLE** statement. Most systems require single or double quotes around character and date data types and commas to separate the values.

When you add data in some, but not all, of the columns in a row, you need to specify those columns. The order in which you list the column names does not matter as long as the order in which you list the data values matches it. The columns that were missing from the statement would have **NULL** values put into the columns in the row. If one of these columns had been assigned a **NOT NULL** status by the **CREATE TABLE** command then the **INSERT** statement would return an error.
When you omit the column list, the **NULL** keyword must be used in the values list to explicitly assign **NULL** values to columns. The **DEFAULT VALUES** is a special option that specifies a single row consisting entirely of the default values for each column.

The insert command also may be used in the form:

> **INSERT INTO** *tablename* [*column_list*] **SELECT** statement**;**

In this form, multiple rows can be added to a table. The data values for the rows are not explicitly specified within the statement text. Instead the source of the new rows are a database query, specified in the statement. This is known as a multi row **INSERT** statement.
The ANSI/ISO standard specifies several logical restrictions on the query that appears within the multi row **INSERT** statement:

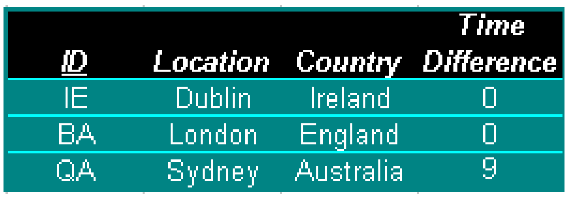
- The query cannot contain an **ORDER BY** clause. This can be seen to be implicit as any data inserted into a table is unordered anyway.

- The query results must contain the same number of columns as the column list in the **INSERT** statement (or the entire target table, if the column list is omitted), and the data types must be compatible, column by column.

Two restriction that have been relaxed since the SQL1 standard are that the query can now be the union of several **SELECT** statements and the **INSERT** statement now permits self-insertion. This is where the table itself is being referenced in the query.

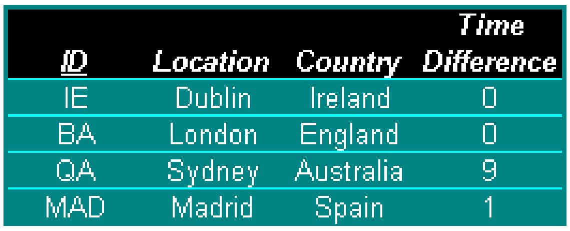
The following example enters one new record into the *Airport* table.

INSERT INTO *airport* **VALUES** (*'MAD', 'Madrid', 'Spain', 1*)**;**



ID	Location	Country	Time Difference
IE	Dublin	Ireland	0
BA	London	England	0
QA	Sydney	Australia	9

Table after insert statement



ID	Location	Country	Time Difference
IE	Dublin	Ireland	0
BA	London	England	0
QA	Sydney	Australia	9
MAD	Madrid	Spain	1

The following example enters one new record into the *Aircraft* table. Two of the columns are not given values, so after the insertion they will contain their default values or be NULL.

INSERT INTO *aircraft (call_sign, model, no_club_seats)* **VALUES**(*'C171', 'Cesena', 10*);

SQL Course

Update Statement

- To introduce the syntax used to update data which is already in the database.
- To note any restrictions the SQL standard places on the use of the **UPDATE** statement.
- To illustrate the use of the **UPDATE** statement using examples.

The **UPDATE** statement alters values within existing rows of a table. The smallest unit of data that can be modified in a database is a single column of a single row.

- The **UPDATE** statement can update a single row or multiple rows.
- We may restrict the update statement to a particular row or value by using the **WHERE** clause.

This command is used to update rows from a table where the rows to be updated are identified by a search condition.
The Update statement also allows a **SELECT** statement to be used to retrieve rows from one or more tables to update another table. The rows to be updated are identified by the search condition. If no **WHERE** clause is specified then all the rows of the table will be updated.
The update command has the form:

> **UPDATE** *tablename* **SET** *columnname* = {<*value expression*>|**NULL**|**DEFAULT**} [*columnname* = {<*value expression*>|**NULL**|**DEFAULT**}, ...] **WHERE** <*search condition*>]**;**

- To update multiple rows from a table you could use a **SELECT** statement in the **WHERE** clause.
- **DEFAULT** implies the substitution of the current default value defined for the target column.
- The **SET** clause may also specify **NULL** as the key word if the column is to be updated to the null value. The target table to be updated is named in the statement, and you must have the required permission to update that table as well as the individual columns within the table. The **SET** clause specifies which columns are to be updated and calculates the new values for them.
- The **WHERE** clause selects the rows of the table to be modified. If the **WHERE** clause is omitted then all rows of the target table are updated.

Conceptually, SQL processes the **UPDATE** statement by going through the target table row by row, updating those rows for which the search condition yields a **FALSE** or **NULL** result. Because it searches the table, this form of the update statement is sometimes referred to as a searched **UPDATE**. This term distinguishes it from a different form of the **UPDATE** statement called a positioned **UPDATE**, which always updates a single row.

The expression in each assignment must be computable based on the values of the row currently being updated in the target table.

- It may not include any column functions or sub-queries.
- If an expression in the assignment list references one of the columns of the target table,

the value used to calculate the expression is the value of that column in the current row before any updates are applied.

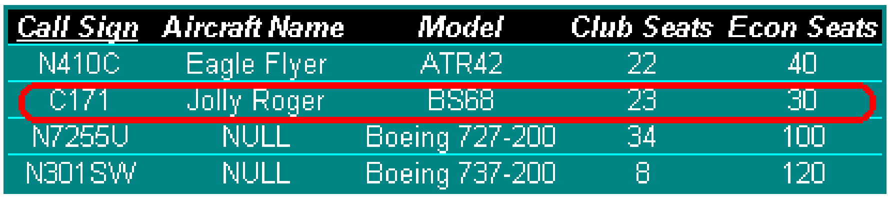
ISO Standards
The ANSI/ISO standard mandates unqualified names for the target columns, but many implementations allow qualified column names. An update may be performed on a table or view with certain restrictions on updating views.
SQL1 applies the following restriction when using the **UPDATE** statement:

- The target table cannot appear in the **FROM** clause of any sub-query at any level of nesting. This prevents the sub-queries from referencing the target table.

SQL2 removes this restriction, and specifies that a reference to the target table in a sub-query is evaluated as if none of the target table has been updated.

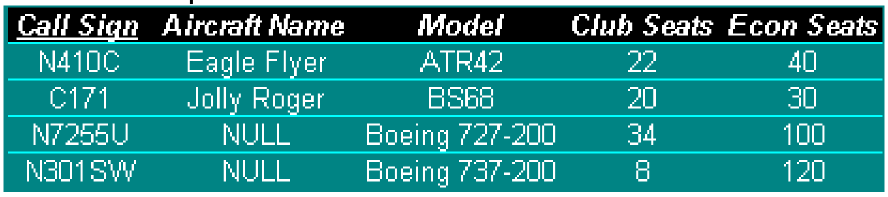
The following example modifies table *Aircraft* by changing the value in the column *no_club_seats* to 20 for all records where the value in column *call_sign* is equal to 'C171'.

UPDATE *aircraft* **SET** *no_club_seats* = 20 **WHERE** *call_sign* = 'C171'**;**



Call Sign	Aircraft Name	Model	Club Seats	Econ Seats
N410C	Eagle Flyer	ATR42	22	40
C171	Jolly Roger	BS68	23	30
N7255U	NULL	Boeing 727-200	34	100
N301SW	NULL	Boeing 737-200	8	120

Table after update statement



Call Sign	Aircraft Name	Model	Club Seats	Econ Seats
N410C	Eagle Flyer	ATR42	22	40
C171	Jolly Roger	BS68	20	30
N7255U	NULL	Boeing 727-200	34	100
N301SW	NULL	Boeing 737-200	8	120

The following example modifies table *Aircraft* by changing the value in the column *no_club_seats* to 50 for all records.

UPDATE *aircraft* **SET** *club_seats* = 50**;**

SQL Course

Delete Statement

- To introduce the syntax used to delete data from the database.
- To note any restrictions the SQL standard places on the use of the **DELETE** statement.
- To illustrate the use of the **DELETE** statement using examples.

To delete a particular value then the **UPDATE** command must be used by setting the intended value to **NULL** , as in the following example:

UPDATE *aircraft* **SET** *no_club_seats* = **NULL;**

The delete command has the form:

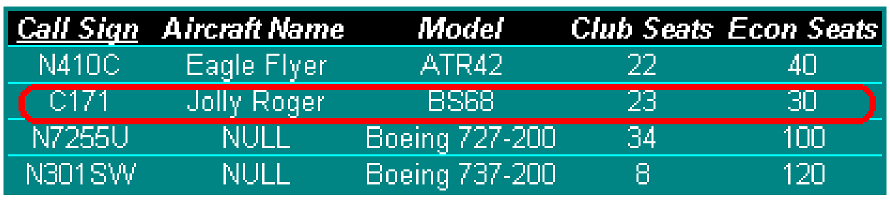
DELETE FROM *tablename* [**WHERE** <*search condition*>]**;**

- The table identified by the table name must be update-able.
- Which rows are to be deleted is determined by the **WHERE** clause.
- The number of rows identified can be any number between zero and the total number of rows in the table.
- If the **WHERE** clause is omitted then all the rows in the table will be deleted.
- The **WHERE** clause can contain **SELECT** queries. The result returned by the query will determine which rows are to be deleted from the table.
- If the table identified is a view, then the rows deleted are the rows in the base table from which the identified rows in the view are defined.

It is important to note that even when the table is empty i.e. contains no rows, it still exists. If you wish to get rid of a table totally you have to execute a **DROP TABLE** statement.

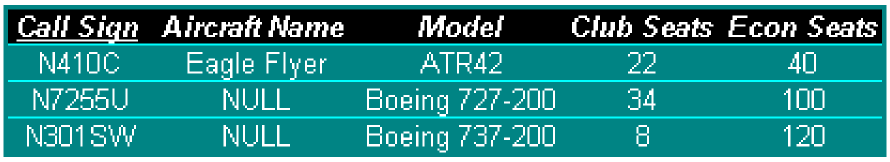
Since the **FROM** clause in the **DELETE** identifies a single table, there is no way to use a join to identify the rows to be deleted. Sub-queries can be used instead. Sub-queries can also be used to delete rows from one table based on the values found in another table.
Conceptually, SQL applies the **WHERE** clause to each row of the target table, deleting those where the search condition yields a **TRUE** result and retaining those where the search condition yields a **NULL** or **FALSE** result. Because this type of **DELETE** statement searches through a table for the rows to be deleted, it is sometimes called a searched **DELETE** statement. There is also a positional **DELETE** which always deletes a single row.
The following example modifies table *Aircraft* by deleting all the records where the value in column *call_sign* is equal to 'C171'.

DELETE FROM *aircraft* **WHERE** *call_sign* = 'C171';



Call Sign	Aircraft Name	Model	Club Seats	Econ Seats
N410C	Eagle Flyer	ATR42	22	40
C171	Jolly Roger	BS68	23	30
N7255U	NULL	Boeing 727-200	34	100
N301SW	NULL	Boeing 737-200	8	120

Table after the delete statement



Call Sign	Aircraft Name	Model	Club Seats	Econ Seats
N410C	Eagle Flyer	ATR42	22	40
N7255U	NULL	Boeing 727-200	34	100
N301SW	NULL	Boeing 737-200	8	120

The following example modifies table *Aircraft* by deleting all records in the table.

```
DELETE FROM aircraft;
```