

LAB # 8: DDL AND DML

Objective:

In this lesson, you learn how to use the data manipulation language (DML) statements to insert rows into a table, update existing rows in a table, and delete existing rows from a table. You also learn how to control transactions with the COMMIT, SAVEPOINT, and ROLLBACK statements.

Data Manipulation Language

Data manipulation language (DML) is a core part of SQL. When you want to add, update, or delete data in the database, you execute a DML statement. A collection of DML statements that form a logical unit of work is called a *transaction*.

Consider a banking database. When a bank customer transfers money from a savings account to a checking account, the transaction might consist of three separate operations: decreasing the savings account, increasing the checking account, and recording the transaction in the transaction journal. The database server must guarantee that all the three SQL statements are performed to maintain the accounts in proper balance. When something prevents one of the statements in the transaction from executing, the other statements of the transaction must be undone.

Adding a New Row to a Table

The graphic in the following example illustrates the addition of a new department to the DEPARTMENTS table

DEPARTMENTS

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
1	10 Administration	200	1700
2	20 Marketing	201	1800
3	50 Shipping	124	1500
4	60 IT	103	1400
5	80 Sales	149	2500
6	90 Executive	100	1700
7	110 Accounting	205	1700
8	190 Contracting	(null)	1700

70 Public Relations

100

1700

New
row

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
1	10 Administration	200	1700
2	20 Marketing	201	1800
3	50 Shipping	124	1500
4	60 IT	103	1400
5	80 Sales	149	2500
6	90 Executive	100	1700
7	110 Accounting	205	1700
8	190 Contracting	(null)	1700

9 70 Public Relations

100

1700

INSERT Statement Syntax

You can add new rows to a table by issuing the INSERT statement.

Syntax:

```
INSERT INTO  table [(column [, column...])]
VALUES      (value [, value...]);
```

In the syntax:

table is the name of the table

column is the name of the column in the table to populate *value*

is the corresponding value for the column

Because you can insert a new row that contains values for each column, the column list is not required in the INSERT clause. However, if you do not use the column list, the values must be listed according to the default order of the columns in the table, and a value must be provided for each column.

Example 1:

```
INSERT INTO departments(department_id,
department_name, manager_id, location_id) VALUES
(70, 'Public Relations', 100, 1700);
```

Output:

```
1 rows inserted
```

For clarity, use the column list in the INSERT clause. Enclose character and date values within single quotation marks; however, it is not recommended that you enclose numeric values within single quotation marks.

Inserting Rows with Null Values

Be sure that you can use null values in the targeted column by verifying the Null status with the DESCRIBE command.

. Any column that is not listed explicitly obtains a null value in the new row.

Common errors that can occur during user input are checked in the following order:

- Mandatory value missing for a NOT NULL column
- Duplicate value violating any unique or primary key constraint
- Any value violating a CHECK constraint
- Referential integrity maintained for foreign key constraint
- Data type mismatches or values too wide to fit in column

Implicit method:

Omit the column from the column list.

Example:

```
INSERT INTO departments (department_id, department_name)
VALUES      (30, 'Purchasing');
```

Output:

```
1 rows inserted
```

Explicit method:

Specify the NULL keyword in the VALUES clause.

Example:

```
INSERT INTO departments VALUES
(100, 'Finance', NULL, NULL);
```

Output:

```
1 rows inserted
```

Inserting Special Values

You can use functions to enter special values in your table.

```
INSERT INTO employees (employee_id,
```

```

    first_name, last_name, email,
    phone_number, hire_date,
    job_id, salary,
    commission_pct,
    manager_id, department_id)
VALUES      (113,
    'Louis', 'Popp',
    'LPOPP', '515.124.4567',
    SYSDATE, 'AC_ACCOUNT', 6900,
    NULL, 205, 110);

```

Output:

```
1 rows inserted
```

The above example records information for employee Popp in the EMPLOYEES table. It supplies the current date and time in the HIRE_DATE column. It uses the SYSDATE function that returns the current date and time of the database server. You may also use the CURRENT_DATE function to get the current date in the session time zone. You can also use the USER function when inserting rows in a table. The USER function records the current username.

Inserting Specific Date and Time Values

The YYYY-MM-DD format is generally used to insert a date value.

You may also supply the date value in the DD-MON-YYYY format. This is recommended because it clearly specifies the century and does not depend on the internal RR format logic of specifying the correct century.

If a date must be entered in a format other than the default format (for example, with another century or a specific time), you must use the DATE_FORMAT function.

Example:

INSERT INTO employees

```

VALUES  (114, 'Den', 'Raphealy', 'DRAPHEAL', '515.127.4561',
    DATE_FORMAT('1999-02-03', '%d %m %Y'),
    'SA_REP', 11000, 0.2, 100, 60);

```

Changing Data in a Table

You can modify the existing values in a table by using the UPDATE statement.

Syntax:

UPDATE *table*

SET *column= value*

WHERE *condition;*

In the syntax:

Table is the name of the table

Column is the name of the column in the table to populate

Value is the corresponding value or subquery for the column

Condition identifies the rows to be updated and is composed of column names, expressions, constants, subqueries, and comparison operators Confirm

the update operation by querying the table to display the updated rows.

Updating Rows in a Table

The UPDATE statement modifies the values of a specific row or rows if the WHERE clause is specified. The following example shows the transfer of employee 113 (Popp) to department 50.

Example:

UPDATE *employees*

SET *department_id = 50*

WHERE *employee_id = 113;*

Output:

```
1 rows updated
```

Values for all the rows in the table are modified if you omit the WHERE clause:

Example:

UPDATE *copy_emp*

SET *department_id = 110;*

Output:

```
22 rows updated
```

Specify SET *column_name*= NULL to update a column value to NULL.

Updating Two Columns with a Subquery

You can update multiple columns in the SET clause of an UPDATE statement by writing multiple subqueries. The syntax is as follows:

```
UPDATE table
SET   column = (SELECT column
                  FROM table
                  WHERE condition)
    [ column = (SELECT column
                  FROM table
                  WHERE condition)]
[WHERE condition ];
```

Example:

```
UPDATE employees
SET   job_id = (SELECT job_id
                FROM employees
                WHERE employee_id = 205),
      salary = (SELECT salary
                FROM employees
                WHERE employee_id = 205)
WHERE employee_id = 113;
```

Output:

```
1 rows updated
```

Updating Rows Based on Another Table

You can use the subqueries in the UPDATE statements to update values in a table.

Example:

```
UPDATE copy_emp
SET   department_id = (SELECT department_id
```

```
FROM employees  
WHERE employee_id = 100)  
WHERE job_id = (SELECT job_id  
FROM employees  
WHERE employee_id = 200);
```

Output:

```
1 rows updated
```

The example in the slide updates the COPY_EMP table based on the values from the EMPLOYEES table. It changes the department number of all employees with employee 200's job ID to employee 100's current department number.

DELETE Statement Syntax

You can remove existing rows from a table by using the DELETE statement. The syntax is as follows:

```
DELETE [FROM]   table  
[WHERE         condition];
```

In the syntax:

table is the name of the table

condition identifies the rows to be deleted, and is composed of column names, expressions, constants, subqueries, and comparison operators

Note: If no rows are deleted, the message —0 rows deleted is returned.

Deleting Rows from a Table

You can delete specific rows by specifying the WHERE clause in the DELETE statement. The following example deletes the Accounting department from the DEPARTMENTS table. You can confirm the delete operation by displaying the deleted rows using the SELECT statement.

Example:

```
DELETE FROM departments  
WHERE department_name = 'Finance';
```

Output:

```
1 rows deleted
```

All rows in the table are deleted if you omit the WHERE clause:

Example:

DELETE FROM copy_emp;

Example:

DELETE FROM departments WHERE department_id IN (30, 40);

Deleting Rows Based on Another Table

You can use the subqueries to delete rows from a table based on values from another table.

Example:

DELETE FROM employees

WHERE department_id =

(SELECT department_id

FROM departments

WHERE department_name

LIKE '%Public%');

Output:

1 rows deleted

The above example deletes all the employees in a department, where the department name contains the string Public.

The subquery searches the DEPARTMENTS table to find the department number based on the department name containing the string Public. The subquery then feeds the department number to the main query, which deletes rows of data from the EMPLOYEES table based on this department number.

TRUNCATE Statement

A more efficient method of emptying a table is by using the TRUNCATE statement.

You can use the TRUNCATE statement to quickly remove all rows from a table or cluster. Removing rows with the TRUNCATE statement is faster than removing them with the DELETE statement for the following reasons:

- The TRUNCATE statement is a data definition language (DDL) statement and generates no rollback information. Rollback information is covered later in this lesson.
- Truncating a table does not fire the delete triggers of the table.

If the table is the parent of a referential integrity constraint, you cannot truncate the table. You need to disable the constraint before issuing the TRUNCATE statement.

Syntax:


```
TRUNCATE TABLE table_name;
```

Example:

TRUNCATE TABLE copy_emp;

Database Transactions

Transactions give you more flexibility and control when changing data, and they ensure data consistency in the event of user process failure or system failure.

Transactions consist of DML statements that constitute one consistent change to the data. For example, a transfer of funds between two accounts should include the debit in one account and the credit to another account of the same amount. Both actions should either fail or succeed together; the credit should not be committed without the debit.

A transaction begins when the first DML statement is encountered and ends when one of the following occurs:

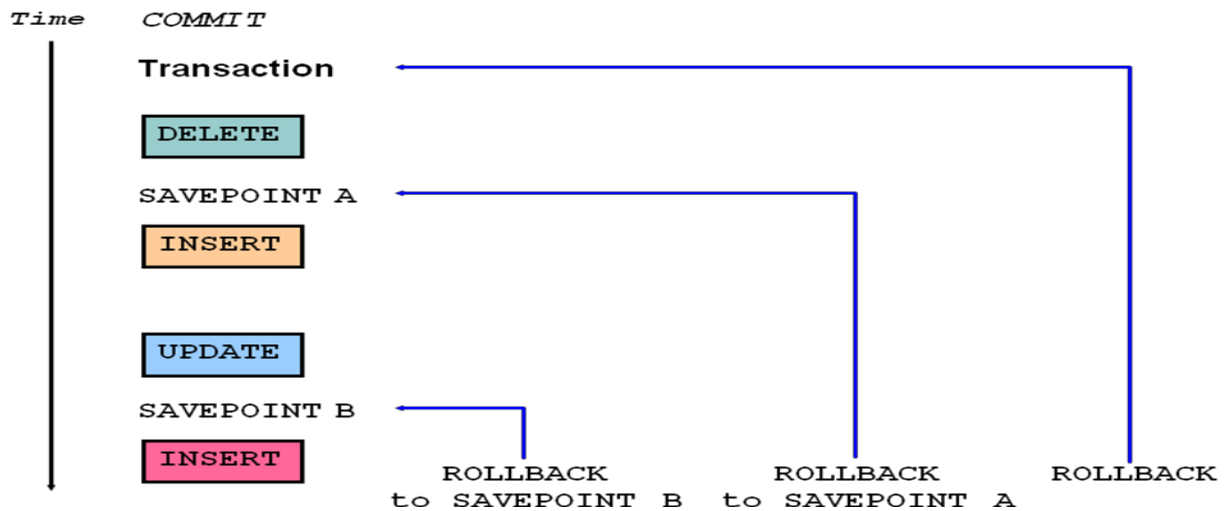
- A COMMIT or ROLLBACK statement is issued.
- A DDL statement, such as CREATE, is issued.
- A DCL statement is issued.
- The user exits workbench.
- A machine fails or the system crashes.

After one transaction ends, the next executable SQL statement automatically starts the next transaction.

A DDL statement or a DCL statement is automatically committed and therefore implicitly ends a transaction.

Explicit Transaction Control Statements

You can control the logic of transactions by using the COMMIT, SAVEPOINT, and ROLLBACK statements.



Rolling Back Changes to a Marker

You can create a marker in the current transaction by using the `SAVEPOINT` statement, which divides the transaction into smaller sections. You can then discard pending changes up to that marker by using the `ROLLBACK TO SAVEPOINT` statement.

Example:

UPDATE...

SAVEPOINT update_done;

Output:

```
SAVEPOINT update_done succeeded.
```

Example:

INSERT...

ROLLBACK TO update_done;

Output:

```
ROLLBACK TO succeeded.
```

Implicit Transaction Processing

- An automatic commit occurs in the following circumstances:
 - A DDL statement is issued

- A DCL statement is issued
- Normal exit from SQL Developer or SQL*Plus, without explicitly issuing COMMIT or ROLLBACK statements

System Failures

When a transaction is interrupted by a system failure, the entire transaction is automatically rolled back. This prevents the error from causing unwanted changes to the data and returns the tables to the state at the time of the last commit. In this way, the SQL server protects the integrity of the tables.

State of the Data Before COMMIT or ROLLBACK

Every data change made during the transaction is temporary until the transaction is committed.

The state of the data before COMMIT or ROLLBACK statements are issued can be described as follows:

- Data manipulation operations primarily affect the database buffer; therefore, the previous state of the data can be recovered.
- The current user can review the results of the data manipulation operations by querying the tables.
- Other users cannot view the results of the data manipulation operations made by the current user. The SQL server institutes read consistency to ensure that each user sees data as it existed at the last commit.
- The affected rows are locked; other users cannot change the data in the affected rows.

State of the Data After COMMIT

Make all pending changes permanent by using the COMMIT statement. Here is what happens after a COMMIT statement:

- Data changes are written to the database.
- The previous state of the data is no longer available with normal SQL queries.
- All users can view the results of the transaction.
- The locks on the affected rows are released; the rows are now available for other users to perform new data changes.
- All savepoints are erased.

Committing Data

Example:

DELETE FROM employees

WHERE employee_id = 99999;

Output:

```
1 rows deleted
```

Example:

INSERT INTO departments

VALUES (290, 'Corporate Tax', NULL, 1700);

Output:

```
1 rows inserted
```

COMMIT ;

Output:

```
COMMIT succeeded.
```

State of the Data After ROLLBACK

Discard all pending changes by using the ROLLBACK statement, which results in the following:

- Data changes are undone.
- The previous state of the data is restored.
- Locks on the affected rows are released.

Exempl:

DELETE FROM copy_emp;

ROLLBACK ;

Database Objects

The SQL database can contain multiple data structures. Each structure should be outlined in the database design so that it can be created during the build stage of database development.

- **Table:** Stores data
- **View:** Subset of data from one or more tables
- **Sequence:** Generates numeric values
- **Index:** Improves the performance of some queries
- **Synonym:** Gives alternative name to an object

Naming Rules

You name database tables and columns according to the standard rules for naming any SQL database object:

- Table names and column names must begin with a letter and be 1–30 characters long.
- Names must contain only the characters A–Z, a–z, 0–9, _ (underscore), \$, and # (legal characters, but their use is discouraged).
- Names must not be reserved word.
 - You may also use quoted identifiers to represent the name of an object. A quoted identifier begins and ends with double quotation marks (—). If you name a schema object using a quoted identifier, then you must use the double quotation marks whenever you refer to that object. Quoted identifiers can be reserved words, although this is not recommended.

Naming Guidelines

Use descriptive names for tables and other database objects.

Note: Names are not case-sensitive. For example, EMPLOYEES is treated to be the same name as eMPloyees or eMpLOYEES. However, quoted identifiers are case-sensitive.

CREATE TABLE Statement

You create tables to store data by executing the SQL CREATE TABLE statement. This statement is one of the DDL statements that are a subset of the SQL statements used to create, modify, or remove SQL database structures. These statements have an immediate effect on the database and they also record information in the data dictionary.

To create a table, a user must have the CREATE TABLE privilege and a storage area in which to create objects. The database administrator (DBA) uses data control language (DCL) statements to grant privileges to users.

Syntax:

```
CREATE TABLE [schema.]table
      (column datatype [DEFAULT expr] [, ...]);
```

In the syntax:

schema Is the same as the owner's name

table Is the name of the table

DEFAULT *expr* Specifies a default value if a value is omitted in the
INSERT statement

column Is the name of the column

datatype Is the column's data type and length

Referencing Another User's Tables

A schema is a collection of logical structures of data or *schema objects*. A schema is owned by a database user and has the same name as that user. Each user owns a single schema.

Schema objects can be created and manipulated with SQL and include tables, views, synonyms, sequences, stored procedures, indexes, clusters, and database links.

If a table does not belong to the user, the owner's name must be prefixed to the table. For example, if there are schemas named USERA and USERB, and both have an EMPLOYEES table, then if USERA wants to access the EMPLOYEES table that belongs to USERB, USERA must prefix the table name with the schema name:

SELECT *

FROM userb.employees;

If USERB wants to access the EMPLOYEES table that is owned by USERA, USERB must prefix the table name with the schema name:

SELECT *

FROM usera.employees;

DEFAULT Option

When you define a table, you can specify that a column should be given a default value by using the DEFAULT option. This option prevents null values from entering the columns when a row is inserted without a value for the column. The default value can be a literal, an expression, or a SQL function (such as SYSDATE or USER), but the value cannot be the name of another column or a pseudocolumn (such as NEXTVAL or CURRVAL). The default expression must match the data type of the column.

Example:

```
CREATE TABLE hire_dates (id int,
                        hire_date DATE DEFAULT SYSDATE);
```

Consider the following examples:

INSERT INTO hire_dates values(45, NULL);

The above statement will insert the null value rather than the default value.

INSERT INTO hire_dates(id) values(35);

The above statement will insert SYSDATE for the HIRE_DATE column.

Creating Tables

Example:

***CREATE TABLE dept
(deptno int, dname
VARCHAR2(14), loc
VARCHAR2(13), create_date DATE
DEFAULT SYSDATE);*** The above
example creates the DEPT table with
four columns: DEPTNO, DNAME,
LOC, and CREATE_DATE. The
CREATE_DATE column has a
default value. If a value is not
provided for an INSERT statement,
the system date is automatically
inserted.

To confirm that the table was created, run the DESCRIBE command.

EXEC sp_help 'dept'

DESCRIBE dept		
Name	Null	Type

DEPTNO		NUMBER(2)
DNAME		VARCHAR2(14)
LOC		VARCHAR2(13)
CREATE_DATE		DATE

Because creating a table is a DDL statement, an automatic commit takes place when this statement is executed.

Data Types

When you identify a column for a table, you need to provide a data type for the column.
There are several data types available:

Data type	Description	Max size	Storage
char(n)	Fixed width character string	8,000 characters	Defined width
varchar(n)	Variable width character string	8,000 characters	2 bytes + number of chars
varchar(max)	Variable width character string	1,073,741,824 characters	2 bytes + number of chars
text	Variable width character string	2GB of text data	4 bytes + number of chars
nchar	Fixed width Unicode string	4,000 characters	Defined width x 2
nvarchar	Variable width Unicode string	4,000 characters	
nvarchar(max)	Variable width Unicode string	536,870,912 characters	
ntext	Variable width Unicode string	2GB of text data	
binary(n)	Fixed width binary string	8,000 bytes	
varbinary	Variable width binary string	8,000 bytes	
varbinary(max)	Variable width binary string	2GB	
image	Variable width binary string	2GB	

Data type	Description	Storage
bit	Integer that can be 0, 1, or NULL	
tinyint	Allows whole numbers from 0 to 255	1 byte
smallint	Allows whole numbers between -32,768 and 32,767	2 bytes
int	Allows whole numbers between -2,147,483,648 and 2,147,483,647	4 bytes
bigint	Allows whole numbers between -9,223,372,036,854,775,808 and 9,223,372,036,854,775,807	8 bytes
decimal(p,s)	Fixed precision and scale numbers. Allows numbers from $-10^{38} + 1$ to $10^{38} - 1$. The p parameter indicates the maximum total number of digits that can be stored (both to the left and to the right of the decimal point). p must be a value from 1 to 38. Default is 18. The s parameter indicates the maximum number of digits stored to the right of the decimal point. s must be a value from 0 to p. Default value is 0	5-17 bytes
numeric(p,s)	Fixed precision and scale numbers. Allows numbers from $-10^{38} + 1$ to $10^{38} - 1$. The p parameter indicates the maximum total number of digits that can be stored (both to the left and to the right of the decimal point). p must be a value from 1 to 38. Default is 18. The s parameter indicates the maximum number of digits stored to the right of the decimal point. s must be a value from 0 to p. Default value is 0	5-17 bytes

Data type	Description	Storage
datetime	From January 1, 1753 to December 31, 9999 with an accuracy of 3.33 milliseconds	8 bytes
datetime2	From January 1, 0001 to December 31, 9999 with an accuracy of 100 nanoseconds	6-8 bytes
smalldatetime	From January 1, 1900 to June 6, 2079 with an accuracy of 1 minute	4 bytes
date	Store a date only. From January 1, 0001 to December 31, 9999	3 bytes
time	Store a time only to an accuracy of 100 nanoseconds	3-5 bytes
datetimeoffset	The same as datetime2 with the addition of a time zone offset	8-10 bytes
timestamp	Stores a unique number that gets updated every time a row gets created or modified. The timestamp value is based upon an internal clock and does not correspond to real time. Each table may have only one timestamp variable	

Constraints

The server uses constraints to prevent invalid data entry into tables. You can use constraints to do the following:

- Enforce rules on the data in a table whenever a row is inserted, updated, or deleted from that table. The constraint must be satisfied for the operation to succeed.
- Prevent the deletion of a table if there are dependencies from other tables.
- Provide rules for SQL tools, such as SQL Developer.

Data Integrity Constraints

Constraint	Description
NOT NULL	Specifies that the column cannot contain a null value
UNIQUE	Specifies a column or combination of columns whose values must be unique for all rows in the table
PRIMARY KEY	Uniquely identifies each row of the table
FOREIGN KEY	Establishes and enforces a referential integrity between the column and a column of the referenced table such that values in one table match values in another table.
CHECK	Specifies a condition that must be true

Constraint Guidelines

All constraints are stored in the data dictionary. Constraints are easy to reference if you give them a meaningful name. Constraint names must follow the standard object-naming rules, except that the name cannot be the same as another object owned by the same user. If you do not name your constraint, the SQL server generates a name with the format SYS_Cn, where *n* is an integer so that the constraint name is unique.

Constraints can be defined at the time of table creation or after the creation of the table. You can define a constraint at the column or table level. Functionally, a table-level constraint is the same as a column-level constraint.

Defining Constraints

You can create constraints at either the column level or table level. Constraints defined at the column level are included when the column is defined. Table-level constraints are defined at the end of the table definition and must refer to the column or columns on which the constraint pertains in a set of parentheses. It is mainly the syntax that differentiates the two; otherwise, functionally, a column-level constraint is the same as a table-level constraint.

NOT NULL constraints must be defined at the column level.

Constraints that apply to more than one column must be defined at the table level.

Syntax:

```
CREATE TABLE [schema.] table
  (column datatype [DEFAULT expr]
  [column_constraint],
  ...
  [table_constraint][,...]) ;
```

In the syntax: schema Is the same as the
owner's name table Is the name of
the table

DEFAULT expr Specifies a default value to be used if a value is omitted in the INSERT
statement

column Is the name of the column datatype

Is the column's data type and length

column_constraint Is an integrity constraint as part of the column definition

table_constraint Is an integrity constraint as part of the table definition

Example of a column-level constraint:

```
CREATE TABLE employees(
  employee_id int
  first_name VARCHAR (20),
  primary key(employee_id)
  ...);
```

The above example uses the column-level syntax to define the constraint.

Example of a table-level constraint:

```
CREATE TABLE employees(
  employee_id int,
  first_name VARCHAR(20),
  ...
  job_id VARCHAR(10) NOT NULL,
```

PRIMARY KEY (EMPLOYEE_ID);

This example uses the table-level syntax to define the constraint.

NOT NULL Constraint

The NOT NULL constraint ensures that the column contains no null values. Columns without the NOT NULL constraint can contain null values by default. NOT NULL constraints must be defined at the column level.

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	HIRE_DATE	JOB_ID	COMMISSION_PCT
100	Steven	King	SKING	17-JUN-87	AD_PRES	(null)
101	Neena	Kochhar	NKOCHHAR	21-SEP-89	AD_VP	(null)
102	Lex	De Haan	LDEHAAN	13-JAN-93	AD_VP	(null)
103	Alex	De Haan	LDEHAAN	13-JAN-93	AD_VP	(null)

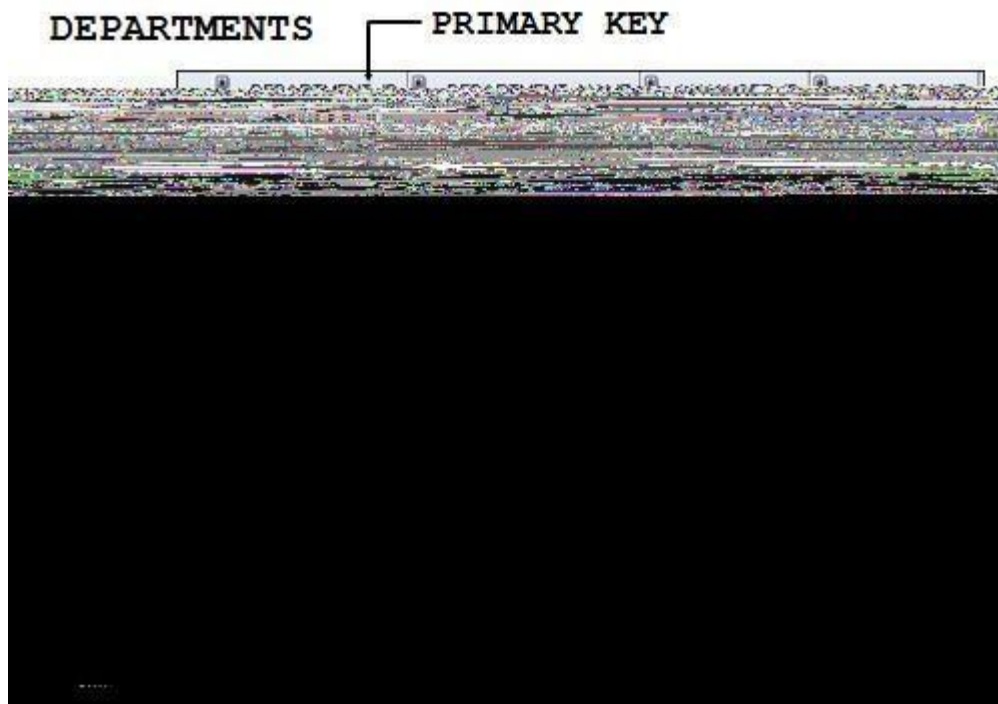
In the EMPLOYEES table, the EMPLOYEE_ID column inherits a NOT NULL constraint as it is defined as a primary key. Otherwise, the LAST_NAME, EMAIL, HIRE_DATE, and JOB_ID columns have the NOT NULL constraint enforced on them.

UNIQUE Constraint

A UNIQUE key integrity constraint requires that every value in a column or a set of columns (key) be unique—that is, no two rows of a table can have duplicate values in a specified column or a set of columns. The column (or set of columns) included in the definition of the UNIQUE key constraint is called the *unique key*. If the UNIQUE constraint comprises more than one column, that group of columns is called a *composite unique key*.

UNIQUE constraints enable the input of nulls unless you also define NOT NULL constraints for the same columns. In fact, any number of rows can include nulls for columns without the NOT NULL constraints because nulls are not considered equal to anything. A null in a **PRIMARY KEY Constraint**

A **PRIMARY KEY** constraint creates a primary key for the table. Only one primary key can be created for each table. The **PRIMARY KEY** constraint is a column or a set of columns that uniquely identifies each row in a table. This constraint enforces the uniqueness of the column or column combination and ensures that no column that is part of the primary key can contain a null value.



FOREIGN KEY Constraint

The **FOREIGN KEY** (or referential integrity) constraint designates a column or a combination of columns as a foreign key and establishes a relationship with a primary key or a unique key in the same table or a different table.

In the example in the slide, `DEPARTMENT_ID` has been defined as the foreign key in the `EMPLOYEES` table (dependent or child table); it references the `DEPARTMENT_ID` column of the `DEPARTMENTS` table (the referenced or parent table).

Guidelines

- A foreign key value must match an existing value in the parent table or be `NULL`.
- Foreign keys are based on data values and are purely logical, rather than physical, pointers.

DEPARTMENTS				
PRIMARY KEY	1	DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID
	2			LOCATION_ID

1	10	Administration	200	1700
2	20	Marketing	201	1800

Example:

CREATE TABLE employees(

employee_id int,

last_name VARCHAR(25) NOT

NULL, email VARCHAR(25), salary

int,

commission_pct int,

hire_date DATE NOT

NULL,

...

department_id NUMBER(4),

CONSTRAINT

emp_dept_fk

FOREIGN KEY

(department_id)

REFERENCES

departments(dep CONSTRAINT emp_email_uk UNIQUE(email)); artment_id),

FOREIGN KEY constraints can be defined at the column or table constraint level. A composite foreign key must be created by using the table-level definition.

The above example defines a FOREIGN KEY constraint on the DEPARTMENT_ID column of the EMPLOYEES table, using table-level syntax. The name of the constraint is EMP_DEPT_FK.

The foreign key can also be defined at the column level, provided that the constraint is based on a single column. The syntax differs in that the keywords FOREIGN KEY do not appear.

For example:

CREATE TABLE employees

(...

department_id NUMBER(4) *foreign*

key(department_id) REFERENCES

departments(department_id),

...

)

FOREIGN KEY Constraint: Keywords

The foreign key is defined in the child table and the table containing the referenced column is the parent table. The foreign key is defined using a combination of the following keywords:

- FOREIGN KEY is used to define the column in the child table at the table constraint level.
- REFERENCES identifies the table and the column in the parent table.
- ON DELETE CASCADE indicates that when a row in the parent table is deleted, the dependent rows in the child table are also deleted.
- ON DELETE SET NULL indicates that when a row in the parent table is deleted, the foreign key values are set to null.

The default behavior is called the *restrict rule*, which disallows the update or deletion of referenced data.

Without the ON DELETE CASCADE or the ON DELETE SET NULL options, the row in the parent table cannot be deleted if it is referenced in the child table.

Creating a Table Using a Subquery

A second method for creating a table is to apply the AS *subquery* clause, which both creates the table and inserts rows returned from the subquery.

Syntax:

```
CREATE TABLE table
      [(column, column...)]
AS subquery;
```

In the syntax:

table is the name of the table

column is the name of the column, default value, and integrity constraint *subquery* is the SELECT statement that defines the set of rows to be inserted into the new table

Guidelines

- The table is created with the specified column names, and the rows retrieved by the SELECT statement are inserted into the table.
- The column definition can contain only the column name and default value.
- If column specifications are given, the number of columns must equal the number of columns in the subquery SELECT list.
- If no column specifications are given, the column names of the table are the same as the column names in the subquery.
- The column data type definitions and the NOT NULL constraint are passed to the new table. Note that only the explicit NOT NULL constraint will be inherited. The PRIMARY KEY column will not pass the NOT NULL feature to the new column. Any other constraint rules are not passed to the new table. However, you can add constraints in the column definition.

Example:

```
CREATE TABLE dept80
AS
  SELECT employee_id, last_name,
         salary*12 ANNSAL,
         hire_date
  FROM employees
 WHERE department_id = 80;
```

```
DESCRIBE dept80
```

Output:

Name	Null	Type
-----	-----	-----
EMPLOYEE_ID		NUMBER(6)
LAST_NAME	NOT NULL	VARCHAR2(25)
ANNSAL		NUMBER
HIRE_DATE	NOT NULL	DATE

ALTER

TABLE Statement

After you create a table, you may need to change the table structure for any of the following reasons:

- You omitted a column.
- Your column definition or its name needs to be changed.
- You need to remove columns.
- You want to put the table into the read-only mode

You can do this by using the ALTER TABLE statement.

Read-Only Tables

When the table is in the READ-ONLY mode, you cannot issue any DML statements that affect the table or any SELECT ... FOR UPDATE statements. You can issue DDL statements as long as they do not modify any data in the table. Operations on indexes associated with the table are allowed when the table is in the READ ONLY mode.

Specify READ/WRITE to return a read-only table to the read/write mode.

```
ALTER TABLE employees READ ONLY;

-- perform table maintenance and then
-- return table back to read/write mode

ALTER TABLE employees READ WRITE;
```

Dropping a Table

The DROP TABLE statement moves a table to the recycle bin or removes the table and all its data from the database entirely. Unless you specify the PURGE clause, the DROP TABLE statement does not result in space being released back to the tablespace for use by other objects, and the space continues to count towards the user's space quota.

Dropping a table invalidates the dependent objects and removes object privileges on the table.

When you drop a table, the database loses all the data in the table and all the indexes associated with it.

Syntax

DROP TABLE *table* [PURGE]

In the syntax, *table* is the name of the table.

Guidelines

- All the data is deleted from the table.
- Any views and synonyms remain, but are invalid.

- Any pending transactions are committed.
- Only the creator of the table or a user with the **DROP ANY TABLE** privilege can remove a table

Output:

```
DROP TABLE dept80 succeeded.
```

```
DROP TABLE dept80;
```