Various Syntax in SQL

All the examples given in this tutorial have been tested with a MySQL server.

SQL SELECT Statement

SELECT column1, column2....columnN

FROM table\_name;

SQL DISTINCT Clause

SELECT DISTINCT column1, column2....columnN

FROM table\_name;

SQL WHERE Clause

SELECT column1, column2....columnN

FROM table\_name

WHERE CONDITION;

SQL AND/OR Clause

SELECT column1, column2....columnN

FROM table\_name

WHERE CONDITION-1 {AND|OR} CONDITION-2;

SQL IN Clause

SELECT column1, column2....columnN

FROM table\_name

WHERE column\_name IN (val-1, val-2,...val-N);

SQL BETWEEN Clause

SELECT column1, column2....columnN

FROM table\_name

WHERE column\_name BETWEEN val-1 AND val-2;

SQL LIKE Clause

SELECT column1, column2....columnN

FROM table\_name

WHERE column\_name LIKE { PATTERN };

SQL ORDER BY Clause

SELECT column1, column2....columnN

FROM table\_name

WHERE CONDITION

ORDER BY column\_name {ASC|DESC};

SQL GROUP BY Clause

SELECT SUM(column\_name)

FROM table\_name

WHERE CONDITION

GROUP BY column\_name;

SQL COUNT Clause

SELECT COUNT(column\_name)

FROM table\_name

WHERE CONDITION;

SQL HAVING Clause

SELECT SUM(column\_name)

FROM table\_name

WHERE CONDITION

GROUP BY column\_name

HAVING (arithematic function condition);

SQL CREATE TABLE Statement

CREATE TABLE table\_name(

column1 datatype,

column2 datatype,

column3 datatype,

.....

columnN datatype,

PRIMARY KEY( one or more columns )

);

SQL DROP TABLE Statement

DROP TABLE table\_name;

SQL CREATE INDEX Statement

CREATE UNIQUE INDEX index\_name

ON table\_name ( column1, column2,...columnN);

SQL DROP INDEX Statement

ALTER TABLE table\_name

DROP INDEX index\_name;

SQL DESC Statement

DESC table\_name;

SQL TRUNCATE TABLE Statement

TRUNCATE TABLE table\_name;

SQL ALTER TABLE Statement

ALTER TABLE table\_name {ADD|DROP|MODIFY} column\_name {data\_ype};

SQL ALTER TABLE Statement (Rename)

ALTER TABLE table\_name RENAME TO new\_table\_name;

SQL INSERT INTO Statement

INSERT INTO table\_name( column1, column2....columnN)

VALUES ( value1, value2....valueN);

SQL UPDATE Statement

UPDATE table\_name

SET column1 = value1, column2 = value2....columnN=valueN

[ WHERE CONDITION ];

SQL DELETE Statement

DELETE FROM table\_name

WHERE {CONDITION};

SQL CREATE DATABASE Statement

CREATE DATABASE database\_name;

SQL DROP DATABASE Statement

DROP DATABASE database\_name;

SQL USE Statement

USE database\_name;

SQL COMMIT Statement

COMMIT;

SQL ROLLBACK Statement

ROLLBACK;

|  |  |
| --- | --- |
| **SQL Statement** | **Syntax** |
| AND / OR | SELECT column\_name(s) FROM table\_name WHERE condition AND|OR condition |
| ALTER TABLE | ALTER TABLE table\_name  ADD column\_name datatype  or  ALTER TABLE table\_name  DROP COLUMN column\_name |
| AS (alias) | SELECT column\_name AS column\_alias FROM table\_name  or  SELECT column\_name FROM table\_name  AS table\_alias |
| BETWEEN | SELECT column\_name(s) FROM table\_name WHERE column\_name BETWEEN value1 AND value2 |
| CREATE DATABASE | CREATE DATABASE database\_name |
| CREATE TABLE | CREATE TABLE table\_name ( column\_name1 data\_type, column\_name2 data\_type, column\_name3 data\_type, ... ) |
| CREATE INDEX | CREATE INDEX index\_name ON table\_name (column\_name)  or  CREATE UNIQUE INDEX index\_name ON table\_name (column\_name) |
| CREATE VIEW | CREATE VIEW view\_name AS SELECT column\_name(s) FROM table\_name WHERE condition |
| DELETE | DELETE FROM table\_name WHERE some\_column=some\_value  or  DELETE FROM table\_name  (**Note:**Deletes the entire table!!)  DELETE \* FROM table\_name  (**Note:**Deletes the entire table!!) |
| DROP DATABASE | DROP DATABASE database\_name |
| DROP INDEX | DROP INDEX table\_name.index\_name (SQL Server) DROP INDEX index\_name ON table\_name (MS Access) DROP INDEX index\_name (DB2/Oracle) ALTER TABLE table\_name DROP INDEX index\_name (MySQL) |
| DROP TABLE | DROP TABLE table\_name |
| EXISTS | IF EXISTS (SELECT \* FROM table\_name WHERE id = ?) BEGIN --do what needs to be done if exists END ELSE BEGIN --do what needs to be done if not END |
| GROUP BY | SELECT column\_name, aggregate\_function(column\_name) FROM table\_name WHERE column\_name operator value GROUP BY column\_name |
| HAVING | SELECT column\_name, aggregate\_function(column\_name) FROM table\_name WHERE column\_name operator value GROUP BY column\_name HAVING aggregate\_function(column\_name) operator value |
| IN | SELECT column\_name(s) FROM table\_name WHERE column\_name IN (value1,value2,..) |
| INSERT INTO | INSERT INTO table\_name VALUES (value1, value2, value3,....)  *or*  INSERT INTO table\_name (column1, column2, column3,...) VALUES (value1, value2, value3,....) |
| INNER JOIN | SELECT column\_name(s) FROM table\_name1 INNER JOIN table\_name2  ON table\_name1.column\_name=table\_name2.column\_name |
| LEFT JOIN | SELECT column\_name(s) FROM table\_name1 LEFT JOIN table\_name2  ON table\_name1.column\_name=table\_name2.column\_name |
| RIGHT JOIN | SELECT column\_name(s) FROM table\_name1 RIGHT JOIN table\_name2  ON table\_name1.column\_name=table\_name2.column\_name |
| FULL JOIN | SELECT column\_name(s) FROM table\_name1 FULL JOIN table\_name2  ON table\_name1.column\_name=table\_name2.column\_name |
| LIKE | SELECT column\_name(s) FROM table\_name WHERE column\_name LIKE pattern |
| ORDER BY | SELECT column\_name(s) FROM table\_name ORDER BY column\_name [ASC|DESC] |
| SELECT | SELECT column\_name(s) FROM table\_name |
| SELECT \* | SELECT \* FROM table\_name |
| SELECT DISTINCT | SELECT DISTINCT column\_name(s) FROM table\_name |
| SELECT INTO | SELECT \* INTO new\_table\_name [IN externaldatabase] FROM old\_table\_name  *or*  SELECT column\_name(s) INTO new\_table\_name [IN externaldatabase] FROM old\_table\_name |
| SELECT TOP | SELECT TOP number|percent column\_name(s) FROM table\_name |
| TRUNCATE TABLE | TRUNCATE TABLE table\_name |
| UNION | SELECT column\_name(s) FROM table\_name1 UNION SELECT column\_name(s) FROM table\_name2 |
| UNION ALL | SELECT column\_name(s) FROM table\_name1 UNION ALL SELECT column\_name(s) FROM table\_name2 |
| UPDATE | UPDATE table\_name SET column1=value, column2=value,... WHERE some\_column=some\_value |
| WHERE | SELECT column\_name(s) FROM table\_name WHERE column\_name operator value |

In the era where data is being generated in humongous amounts, there is a constant need to handle data in databases. Relational databases are one of the most popular databases, and SQL is the basis of relational databases. Therefore [**SQL skills**](https://www.edureka.co/mysql-dba) are indispensable in most of the job roles. In this article on SQL Commands, I will discuss the top commands and statements that you need to understand in SQL.

The topics covered in this blog are mainly divided into 4 categories:

* [**Data Definition Language(DDL)**](https://www.edureka.co/blog/sql-commands#DDL) – Consists of commands which are used to define the database.
* [**Data Manipulation Language(DML)**](https://www.edureka.co/blog/sql-commands#DML) –  Consists of commands which are used to manipulate the data present in the database.
* [**Data Control Language(DCL)**](https://www.edureka.co/blog/sql-commands#DCL) – Consists of commands which deal with the user permissions and controls of the database system.
* [**Transaction Control Language(TCL)**](https://www.edureka.co/blog/sql-commands#TCL) – Consist of commands which deal with the transaction of the database.

Apart from the above commands, the following topics will also be covered in this article:

* [Comments in SQL](https://www.edureka.co/blog/sql-commands#Comments)
* [Different Types Of Keys In Database](https://www.edureka.co/blog/sql-commands#Keys%20In%20Database)
* [Constraints Used In Database](https://www.edureka.co/blog/sql-commands#Constraints)
* [Nested Queries](https://www.edureka.co/blog/sql-commands#Nested%20Queries)
* [Joins](https://www.edureka.co/blog/sql-commands#Joins)
* [Set Operations](https://www.edureka.co/blog/sql-commands#Set%20Operations)
* [Dates & Auto Increment](https://www.edureka.co/blog/sql-commands#Dates%20and%20Auto%20Increment)
* [Views](https://www.edureka.co/blog/sql-commands#Views)
* [Stored Procedures](https://www.edureka.co/blog/sql-commands#Stored%20Procedures)
* [Triggers](https://www.edureka.co/blog/sql-commands#Triggers)

In this article on SQL Commands, I am going to consider the below database as an example, to show you how to write commands.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **EmployeeID** | **EmployeeName** | **Emergency ContactName** | **PhoneNumber** | **Address** | **City** | **Country** |
| 01 | Shanaya | Abhinay | 9898765612 | Oberoi Street 23 | Mumbai | India |
| 02 | Anay | Soumya | 9432156783 | Marathalli House No 23 | Delhi | India |
| 03 | Preeti | Rohan | 9764234519 | Queens Road 45 | Bangalore | India |
| 04 | Vihaan | Akriti | 9966442211 | Brigade Road Block 4 | Hyderabad | India |
| 05 | Manasa | Shourya | 9543176246 | Mayo Road 23 | Kolkata | India |

So, let’s get started now!

## ****Comments in SQL****

There are two ways in which you can comment in SQL, i.e. either the [Single-Line Comments](https://www.edureka.co/blog/sql-commands#Single-Line%20Comments) or the [Multi-Line Comments](https://www.edureka.co/blog/sql-commands#Multi-Line%20Comments).

### ****Single-Line Comments****

The single line comment starts with two hyphens (–). So, any text mentioned after (–), till the end of a single line will be ignored by the compiler.

#### **Example:**

|  |  |
| --- | --- |
| 1  2 | --Select all:  SELECT \* FROM Employee\_Info; |

### ****Multi-Line Comments****

The Multi-line comments start with **/\*** and end with **\*/**. So, any text mentioned between /\* and \*/ will be ignored by the compiler.

#### **Example:**

|  |  |
| --- | --- |
| 1  2  3  4 | /\*Select all the columns  of all the records  from the Employee\_Info table:\*/  SELECT \* FROM Students; |

## ****SQL Commands: Data Definition Language Commands (DDL)****

This section of the article will give you an insight into the commands through which you can define your database. The commands are as follows:

* + [CREATE](https://www.edureka.co/blog/sql-commands#CREATE)
  + [DROP](https://www.edureka.co/blog/sql-commands#DROP)
  + [TRUNCATE](https://www.edureka.co/blog/sql-commands#TRUNCATE)
  + [ALTER](https://www.edureka.co/blog/sql-commands#ALTER)
  + [BACKUP DATABASE](https://www.edureka.co/blog/sql-commands#BACKUP%20DATABASE)

### ****CREATE****

This statement is used to create a table or a database.

#### **The ‘CREATE DATABASE’ Statement**

As the name suggests, this statement is used to create a database.

##### **Syntax**

CREATE DATABASE DatabaseName;

##### **Example**

|  |  |
| --- | --- |
| 1 | CREATE DATABASE Employee; |

#### **The ‘CREATE TABLE’ Statement**

This statement is used to create a table.

##### **Syntax**

CREATE TABLE TableName (

Column1 datatype,

Column2 datatype,

Column3 datatype,

....

ColumnN datatype

);

##### **Example**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | CREATE TABLE Employee\_Info  (  EmployeeID int,  EmployeeName varchar(255),  Emergency ContactName varchar(255),  PhoneNumber int,  Address varchar(255),  City varchar(255),  Country varchar(255)  ); |

You can also create a table using another table. Refer the below sytax and example:

#### **The ‘CREATE TABLE AS’ Statement**

##### **Syntax**

CREATE TABLE NewTableName AS

SELECT Column1, column2,..., ColumnN

FROM ExistingTableName

WHERE ....;

##### **Example**

|  |  |
| --- | --- |
| 1  2  3 | CREATE TABLE ExampleTable AS  SELECT EmployeeName, PhoneNumber  FROM Employee\_Info; |

### ****DROP****

This statement is used to drop an existing table or a database.

#### **The ‘DROP DATABASE’ Statement**

This statement is used to drop an existing database. When you use this statement, complete information present in the database will be lost.

##### **Syntax**

DROP DATABASE DatabaseName;

##### **Example**

|  |  |
| --- | --- |
| 1 | DROP DATABASE Employee; |

#### **The ‘DROP TABLE’ Statement**

This statement is used to drop an existing table. When you use this statement, complete information present in the table will be lost.

##### **Syntax**

DROP TABLE TableName;

##### **Example**

|  |  |
| --- | --- |
| 1 | DROP Table Employee\_Info; |

### ****TRUNCATE****

This command is used to delete the information present in the table but does not delete the table. So, once you use this command, your information will be lost, but not the table.

##### **Syntax**

TRUNCATE TABLE TableName;

##### **Example**

|  |  |
| --- | --- |
| 1 | TRUNCATE Table Employee\_Info; |

### ****ALTER****

This command is used to delete, modify or add constraints or columns in an existing table.

#### **The ‘ALTER TABLE’ Statement**

This statement is used to add, delete, modify columns in an existing table.

#### **The ‘ALTER TABLE’ Statement with ADD/DROP COLUMN**

You can use the ALTER TABLE statement with ADD/DROP Column command according to your need. If you wish to add a column, then you will use the ADD command, and if you wish to delete a column, then you will use the DROP COLUMN command.

##### **Syntax**

ALTER TABLE TableName

ADD ColumnName Datatype;

ALTER TABLE TableName

DROP COLUMN ColumnName;

##### **Example**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | --ADD Column BloodGroup:    ALTER TABLE Employee\_Info  ADD BloodGroup varchar(255);    --DROP Column BloodGroup:    ALTER TABLE Employee\_Info  DROP COLUMN BloodGroup ; |

#### **The ‘ALTER TABLE’ Statement with ALTER/MODIFY COLUMN**

This statement is used to change the datatype of an existing column in a table.

##### **Syntax**

ALTER TABLE TableName

ALTER COLUMN ColumnName Datatype;

##### **Example**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | --Add a column DOB and change the data type to Date.    ALTER TABLE Employee\_Info  ADD DOB year;    ALTER TABLE Employee\_Info  ALTER DOB date; |

### ****BACKUP DATABASE****

This statement is used to create a full backup of an existing database.

##### **Syntax**

BACKUP DATABASE DatabaseName

TO DISK = 'filepath';

##### **Example**

|  |  |
| --- | --- |
| 1  2 | BACKUP DATABASE Employee  TO DISK = 'C:\Users\Sahiti\Desktop'; |

You can also use a ***differential back up.*** This type of back up only backs up the parts of the database, which have changed since the last complete backup of the database.

##### **Syntax**

BACKUP DATABASE DatabaseName

TO DISK = 'filepath'

WITH DIFFERENTIAL;

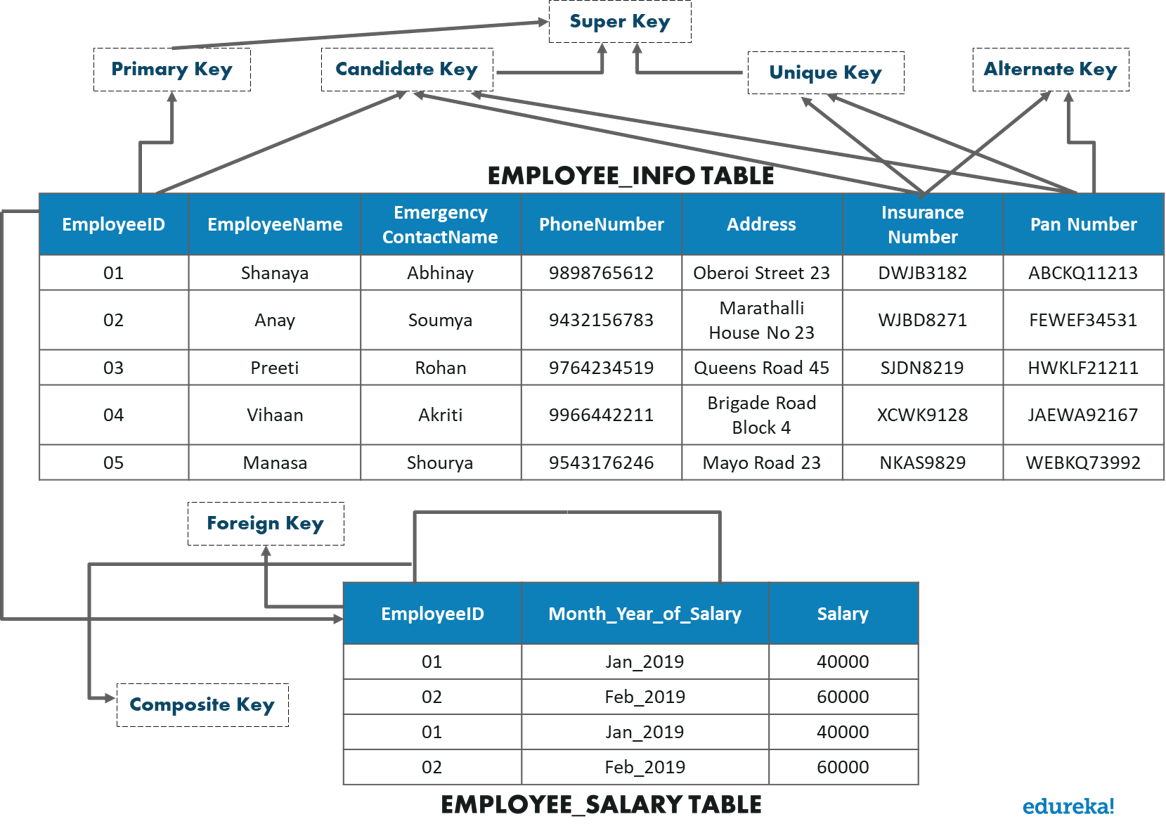
##### **Example**

|  |  |
| --- | --- |
| 1  2  3 | BACKUP DATABASE Employee  TO DISK = 'C:\Users\Sahiti\Desktop'  WITH DIFFERENTIAL; |

Now that you know the data definition commands, let me take you through the various types of Keys and Constraints that you need to understand before learning how to manipulate the databases.

## ****SQL Commands: Different Types Of Keys In Database****

There are mainly 7 types of Keys, that can be considered in a database. I am going to consider the below tables to explain to you the various keys.



* **Candidate Key –**A set of attributes which can uniquely identify a table can be termed as a Candidate Key. A table can have more than one candidate key, and out of the chosen candidate keys, one key can be chosen as a Primary Key. In the above example, since EmployeeID, InsuranceNumber and PanNumber can uniquely identify every tuple, they would be considered as a Candidate Key.
* **Super Key –**The set of attributes which can uniquely identify a tuple is known as Super Key. So, a candidate key, primary key, and a unique key is a superkey, but vice-versa isn’t true.
* **Primary Key –** A set of attributes which are used to uniquely identify every tuple is also a primary key. In the above example, since EmployeeID, InsuranceNumber and PanNumber are candidate keys, any one of them can be chosen as a Primary Key. Here EmployeeID is chosen as the primary key.
* **Alternate Key –**Alternate Keys are the candidate keys, which are not chosen as a Primary key. From the above example, the alternate keys are PanNumber and Insurance Number.
* **Unique Key –** The unique key is similar to the primary key, but allows one NULL value in the column. Here the Insurance Number and the Pan Number can be considered as unique keys.
* **Foreign Key –**An attribute that can only take the values present as the values of some other attribute, is the foreign key to the attribute to which it refers. in the above example, the Employee\_ID from the Employee\_Information Table is referred to the Employee\_ID from the Employee\_Salary Table.
* **Composite Key –**A composite key is a combination of two or more columns that identify each tuple uniquely. Here, the Employee\_ID and Month-Year\_Of\_Salary can be grouped together to uniquely identify every tuple in the table.

## ****SQL Commands: Constraints Used In Database****

Constraints are used in a database to specify the rules for data in a table. The following are the different types of constraints:

* [NOT NULL](https://www.edureka.co/blog/sql-commands#NOT%20NULL)
* [UNIQUE](https://www.edureka.co/blog/sql-commands#UNIQUE)
* [CHECK](https://www.edureka.co/blog/sql-commands#CHECK)
* [DEFAULT](https://www.edureka.co/blog/sql-commands#DEFAULT)
* [INDEX](https://www.edureka.co/blog/sql-commands#INDEX)

### ****NOT NULL****

This constraint ensures that a column cannot have a NULL value.

#### **Example**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | --NOT NULL on Create Table    CREATE TABLE Employee\_Info  (  EmployeeID int NOT NULL,  EmployeeName varchar(255) NOT NULL,  Emergency ContactName varchar(255),  PhoneNumber int NOT NULL,  Address varchar(255),  City varchar(255),  Country varchar(255)  );    --NOT NULL on ALTER TABLE    ALTER TABLE Employee\_Info  MODIFY PhoneNumber int NOT NULL; |

### ****UNIQUE****

This constraint ensures that all the values in a column are unique.

#### **Example**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | --UNIQUE on Create Table    CREATE TABLE Employee\_Info  (  EmployeeID int NOT NULL UNIQUE,  EmployeeName varchar(255) NOT NULL,  Emergency ContactName varchar(255),  PhoneNumber int NOT NULL,  Address varchar(255),  City varchar(255),  Country varchar(255)  );    --UNIQUE on Multiple Columns    CREATE TABLE Employee\_Info  (  EmployeeID int NOT NULL,  EmployeeName varchar(255) NOT NULL,  Emergency ContactName varchar(255),  PhoneNumber int NOT NULL,  Address varchar(255),  City varchar(255),  Country varchar(255),  CONSTRAINT UC\_Employee\_Info UNIQUE(Employee\_ID, PhoneNumber)  );    --UNIQUE on ALTER TABLE    ALTER TABLE Employee\_Info  ADD UNIQUE (Employee\_ID);    --To drop a UNIQUE constraint    ALTER TABLE  Employee\_Info  DROP CONSTRAINT UC\_Employee\_Info; |

### ****CHECK****

This constraint ensures that all the values in a column satisfy a specific condition.

#### **Example**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40 | --CHECK Constraint on CREATE TABLE    CREATE TABLE Employee\_Info  (  EmployeeID int NOT NULL,  EmployeeName varchar(255),  Emergency ContactName varchar(255),  PhoneNumber int,  Address varchar(255),  City varchar(255),  Country varchar(255) CHECK (Country=='India')  );    --CHECK Constraint on multiple columns    CREATE TABLE Employee\_Info  (  EmployeeID int NOT NULL,  EmployeeName varchar(255),  Emergency ContactName varchar(255),  PhoneNumber int,  Address varchar(255),  City varchar(255),  Country varchar(255) CHECK (Country = 'India' AND Cite = 'Hyderabad')  );    --CHECK Constraint on ALTER TABLE    ALTER TABLE Employee\_Info  ADD CHECK (Country=='India');    --To give a name to the CHECK Constraint    ALTER TABLE Employee\_Info  ADD CONSTRAINT CheckConstraintName CHECK (Country=='India');    --To drop a CHECK Constraint    ALTER TABLE Employee\_Info  DROP CONSTRAINT CheckConstraintName; |

### ****DEFAULT****

This constraint consists of a set of default values for a column when no value is specified.

#### **Example**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | --DEFAULT Constraint on CREATE TABLE    CREATE TABLE Employee\_Info  (  EmployeeID int NOT NULL,  EmployeeName varchar(255),  Emergency ContactName varchar(255),  PhoneNumber int,  Address varchar(255),  City varchar(255),  Country varchar(255) DEFAULT 'India'  );    --DEFAULT Constraint on ALTER TABLE    ALTER TABLE Employee\_Info  ADD CONSTRAINT defau\_Country  DEFAULT 'India' FOR Country;    --To drop the Default Constraint    ALTER TABLE Employee\_Info  ALTER COLUMN Country DROP DEFAULT; |

### ****INDEX****

This constraint is used to create indexes in the table, through which you can create and retrieve data from the database very quickly.

#### **Syntax**

--Create an Index where duplicate values are allowed

CREATE INDEX IndexName

ON TableName (Column1, Column2, ...ColumnN);

--Create an Index where duplicate values are not allowed

CREATE UNIQUE INDEX IndexName

ON TableName (Column1, Column2, ...ColumnN);

#### **Example**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | CREATE INDEX idex\_EmployeeName  ON Persons (EmployeeName);    --To delete an index in a table    DROP INDEX Employee\_Info.idex\_EmployeeName; |

Now, let us look into the next part of this article i.e. DML Commands.

## ****SQL Commands: Data Manipulation Language Commands (DML)****

This section of the article will give you an insight into the commands through which you can manipulate the database. The commands are as follows:

[[](https://www.edureka.co/mysql-dba)](https://www.edureka.co/mysql-dba" \t "_blank)

### [MySQL DBA Certification Training](https://www.edureka.co/mysql-dba" \t "_blank)

* *[Instructor-led Sessions](https://www.edureka.co/mysql-dba" \t "_blank)*
* *[Real-life Case Studies](https://www.edureka.co/mysql-dba" \t "_blank)*
* *[Assignments](https://www.edureka.co/mysql-dba" \t "_blank)*
* *[Lifetime Access](https://www.edureka.co/mysql-dba" \t "_blank)*

[Explore Curriculum](https://www.edureka.co/mysql-dba" \t "_blank)

* + [USE](https://www.edureka.co/blog/sql-commands#USE)
  + [INSERT INTO](https://www.edureka.co/blog/sql-commands#INSERT%20INTO)
  + [UPDATE](https://www.edureka.co/blog/sql-commands#UPDATE)
  + [DELETE](https://www.edureka.co/blog/sql-commands#DELETE)
  + [SELECT](https://www.edureka.co/blog/sql-commands#SELECT)

Apart from these commands, there are also other manipulative operators/functions such as:

* + [Operators](https://www.edureka.co/blog/sql-commands#Operators)
  + [Aggregate Functions](https://www.edureka.co/blog/sql-commands#Aggregate%20Functions)
  + [NULL Functions](https://www.edureka.co/blog/sql-commands#NULL%20Functions)
  + [Aliases & Case Statement](https://www.edureka.co/blog/sql-commands#Aliases%20and%20Case%20Statement)

### ****USE****

The USE statement is used to select the database on which you want to perform operations.

#### **Syntax**

USE DatabaseName;

#### **Example**

|  |  |
| --- | --- |
| 1 | USE Employee; |

### ****INSERT INTO****

This statement is used to insert new records into the table.

#### **Syntax**

INSERT INTO TableName (Column1, Column2, Column3, ...,ColumnN)

VALUES (value1, value2, value3, ...);

--If you don't want to mention the column names then use the below syntax

INSERT INTO TableName

VALUES (Value1, Value2, Value3, ...);

**Example**

|  |  |
| --- | --- |
| 1  2  3  4  5 | INSERT INTO Employee\_Info(EmployeeID, EmployeeName, Emergency ContactName, PhoneNumber, Address, City, Country)  VALUES ('06', 'Sanjana','Jagannath', '9921321141', 'Camel Street House No 12', 'Chennai', 'India');    INSERT INTO Employee\_Info  VALUES ('07', 'Sayantini','Praveen', '9934567654', 'Nice Road 21', 'Pune', 'India'); |

### ****UPDATE****

This statement is used to modify the records already present in the table.

#### **Syntax**

UPDATE TableName

SET Column1 = Value1, Column2 = Value2, ...

WHERE Condition;

#### **Example**

|  |  |
| --- | --- |
| 1  2  3 | UPDATE Employee\_Info  SET EmployeeName = 'Aahana', City= 'Ahmedabad'  WHERE EmployeeID = 1; |

### ****DELETE****

This statement is used to delete the existing records in a table.

#### **Syntax**

DELETE FROM TableName WHERE Condition;

#### **Example**

|  |  |
| --- | --- |
| 1  2 | DELETE FROM Employee\_Info  WHERE EmployeeName='Preeti'; |

### ****SELECT****

This statement is used to select data from a database and the data returned is stored in a result table, called the **result-set**.

#### **Syntax**

SELECT Column1, Column2, ...ColumN

FROM TableName;

--(\*) is used to select all from the table

SELECT \* FROM table\_name;

-- To select the number of records to return use:

SELECT TOP 3 \* FROM TableName;

#### **Example**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | SELECT EmployeeID, EmployeeName  FROM Employee\_Info;    --(\*) is used to select all from the table  SELECT \* FROM Employee\_Info;    -- To select the number of records to return use:  SELECT TOP 3 \* FROM Employee\_Info; |

Apart from just using the SELECT keyword individually, you can use the following keywords with the SELECT statement:

* + [DISTINCT](https://www.edureka.co/blog/sql-commands#DISTINCT)
  + [ORDER BY](https://www.edureka.co/blog/sql-commands#ORDER%20BY)
  + [GROUP BY](https://www.edureka.co/blog/sql-commands#GROUP%20BY)
  + [HAVING Clause](https://www.edureka.co/blog/sql-commands#HAVING%20Clause)
  + [INTO](https://www.edureka.co/blog/sql-commands#INTO)

#### **The ‘SELECT DISTINCT’ Statement**

This statement is used to return only different values.

##### **Syntax**

SELECT DISTINCT Column1, Column2, ...ColumnN

FROM TableName;

##### **Example**

|  |  |
| --- | --- |
| 1 | SELECT DISTINCT PhoneNumber FROM Employee\_Info; |

#### **The ‘ORDER BY’ Statement**

The ‘ORDER BY’ statement is used to sort the required results in ascending or descending order. The results are sorted in ascending order by default. Yet, if you wish to get the required results in descending order, you have to use the **DESC** keyword.

##### **Syntax**

SELECT Column1, Column2, ...ColumnN

FROM TableName

ORDER BY Column1, Column2, ... ASC|DESC;

##### **Example**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | -- Select all employees from the 'Employee\_Info' table sorted by EmergencyContactName:  SELECT \* FROM Employee\_Info  ORDER BY EmergencyContactName;    -- Select all employees from the 'Employee\_Info' table sorted by EmergencyContactName in Descending order:  SELECT \* FROM Employee\_Info  ORDER BY EmergencyContactName DESC;    -- Select all employees from the 'Employee\_Info' table sorted by EmergencyContactName and EmployeeName:  SELECT \* FROM Employee\_Info  ORDER BY EmergencyContactName, EmployeeName;    /\* Select all employees from the 'Employee\_Info' table sorted by EmergencyContactName in Descending order and EmployeeName in Ascending order: \*/  SELECT \* FROM Employee\_Info  ORDER BY EmergencyContactName ASC, EmployeeName DESC; |

#### **The ‘GROUP BY’ Statement**

This ‘GROUP BY’ statement is used with the aggregate functions to group the result-set by one or more columns.

##### **Syntax**

SELECT Column1, Column2,..., ColumnN

FROM TableName

WHERE Condition

GROUP BY ColumnName(s)

ORDER BY ColumnName(s);

##### **Example**

|  |  |
| --- | --- |
| 1  2  3  4  5 | -- To list the number of employees from each city.    SELECT COUNT(EmployeeID), City  FROM Employee\_Info  GROUP BY City; |

#### **The ‘HAVING’ Clause**

The ‘HAVING’ clause is used in SQL because the **WHERE keyword** cannot be used everywhere.

##### **Syntax**

SELECT ColumnName(s)

FROM TableName

WHERE Condition

GROUP BY ColumnName(s)

HAVING Condition

ORDER BY ColumnName(s);

##### **Example**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | /\*  To list the number of employees in each city. The employees should be sorted high to low and only those cities must be included who have more than 5 employees:\*/    SELECT COUNT(EmployeeID), City  FROM Employee\_Info  GROUP BY City  HAVING COUNT(EmployeeID) > 2  ORDER BY COUNT(EmployeeID) DESC; |

#### **The ‘SELECT INTO’ Statement**

The ‘SELECT INTO’ statement is used to copy data from one table to another.

##### **Syntax**

SELECT \*

INTO NewTable [IN ExternalDB]

FROM OldTable

WHERE Condition;

##### **Example**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | -- To create a backup of database 'Employee'  SELECT \* INTO EmployeeBackup  FROM Employee;    --To select only few columns from Employee  SELECT EmployeeName, PhoneNumber INTO EmployeeContactDetails  FROM Employee;    SELECT \* INTO BlrEmployee  FROM Employee  WHERE City = 'Bangalore'; |

Now, as I mentioned before, let us move onto our next section in this article on SQL Commands, i.e. the Operators.

### ****Operators in SQL****

The different set of operators available in SQL are as follows:



Let us look into each one of them, one by one.

#### **Arithmetic Operators**

|  |  |
| --- | --- |
| **Operator** | **Description** |
| % | Modulous [A % B] |
| / | Division [A / B] |
| \* | Multiplication [A \* B] |
| – | Subtraction  [A – B] |
| + | Addition [A + B] |

#### **Bitwise Operators**

|  |  |
| --- | --- |
| **Operator** | **Description** |
| ^ | Bitwise Exclusive OR (XOR) [A ^ B] |
| | | Bitwise OR [A | B] |
| & | Bitwise AND [A & B] |

#### **Comparison Operators**

|  |  |
| --- | --- |
| **Operator** | **Description** |
| < > | Not Equal to [A < > B] |
| <= | Less than or equal to [A <= B] |
| >= | Greater than or equal to [A >= B] |
| < | Less than [A < B] |
| > | Greater than [A > B] |
| = | Equal to [A = B] |

#### **Compound Operators**

|  |  |
| --- | --- |
| **Operator** | **Description** |
| |\*= | Bitwise OR equals [A |\*= B] |
| ^-= | Bitwise Exclusive equals [A ^-= B] |
| &= | Bitwise AND equals [A &= B] |
| %= | Modulo equals [A %= B] |
| /= | Divide equals [A /= B] |
| \*= | Multiply equals [A\*= B] |
| -= | Subtract equals [A-= B] |
| += | Add equals [A+= B] |

#### **Logical Operators**

The Logical operators present in SQL are as follows:

* + [AND](https://www.edureka.co/blog/sql-commands#AND)
  + [OR](https://www.edureka.co/blog/sql-commands#OR)
  + [NOT](https://www.edureka.co/blog/sql-commands#NOT)
  + [BETWEEN](https://www.edureka.co/blog/sql-commands#BETWEEN)
  + [LIKE](https://www.edureka.co/blog/sql-commands#LIKE)
  + [IN](https://www.edureka.co/blog/sql-commands#IN)
  + [EXISTS](https://www.edureka.co/blog/sql-commands#EXISTS)
  + [ALL](https://www.edureka.co/blog/sql-commands#ALL)
  + [ANY](https://www.edureka.co/blog/sql-commands#ANY)

##### **AND Operator**

This operator is used to filter records that rely on more than one condition. This operator displays the records, which satisfy all the conditions separated by AND, and give the output TRUE.

###### ****Syntax****

SELECT Column1, Column2, ..., ColumnN

FROM TableName

WHERE Condition1 AND Condition2 AND Condition3 ...;

###### ****Example****

|  |  |
| --- | --- |
| 1  2 | SELECT \* FROM Employee\_Info  WHERE City='Mumbai' AND City='Hyderabad';</pre> |

##### **OR Operator**

This operator displays all those records which satisfy any of the conditions separated by OR and give the output TRUE.

###### ****Syntax****

SELECT Column1, Column2, ..., ColumnN

FROM TableName

WHERE Condition1 OR Condition2 OR Condition3 ...;

###### ****Example****

|  |  |
| --- | --- |
| 1  2 | SELECT \* FROM Employee\_Info  WHERE City='Mumbai' OR City='Hyderabad'; |

##### **NOT Operator**

The NOT operator is used, when you want to display the records which do not satisfy a condition.

###### ****Syntax****

SELECT Column1, Column2, ..., ColumnN

FROM TableName

WHERE NOT Condition;

###### ****Example****

|  |  |
| --- | --- |
| 1  2 | SELECT \* FROM Employee\_Info  WHERE NOT City='Mumbai'; |

**NOTE:** You can also combine the above three operators and write a query as follows:

|  |  |
| --- | --- |
| 1  2 | SELECT \* FROM Employee\_Info  WHERE NOT Country='India' AND (City='Bangalore' OR City='Hyderabad'); |

**NOTE:** You can also combine the above three operators and write a query as follows:

|  |  |
| --- | --- |
| 1  2 | SELECT \* FROM Employee\_Info  WHERE NOT Country='India' AND (City='Bangalore' OR City='Hyderabad'); |

##### **BETWEEN Operator**

The BETWEEN operator is used, when you want to select values within a given range. Since this is an inclusive operator, both the starting and ending values are considered.

###### ****Syntax****

SELECT ColumnName(s)

FROM TableName

WHERE ColumnName BETWEEN Value1 AND Value2;

###### ****Example****

|  |  |
| --- | --- |
| 1  2 | SELECT \* FROM Employee\_Salary  WHERE Salary BETWEEN 40000 AND 50000; |

##### **LIKE Operator**

The LIKE operator is used in a WHERE clause to search for a specified pattern in a column of a table. There are mainly two wildcards that are used in conjunction with the LIKE operator:

* **%** – It matches 0 or more character.
* **\_** – It matches exactly one character.

###### ****Syntax****

SELECT ColumnName(s)

FROM TableName

WHERE ColumnName LIKE pattern;

Refer to the following table for the various patterns that you can mention with the LIKE operator.

|  |  |
| --- | --- |
| **Like Operator Condition** | **Description** |
| WHERE CustomerName LIKE ‘v% | Finds any values that start with “v” |
| WHERE CustomerName LIKE ‘%v’ | Finds any values that end with “v” |
| WHERE CustomerName LIKE ‘%and%’ | Finds any values that have “and” in any position |
| WHERE CustomerName LIKE ‘\_q%’ | Finds any values that have “q” in the second position. |
| WHERE CustomerName LIKE ‘u\_%\_%’ | Finds any values that start with “u” and are at least 3 characters in length |
| WHERE ContactName LIKE ‘m%a’ | Finds any values that start with “m” and end with “a” |

###### ****Example****

|  |  |
| --- | --- |
| 1  2 | SELECT \* FROM Employee\_Info  WHERE EmployeeName LIKE 'S%'; |

##### **IN Operator**

This operator is used for multiple OR conditions. This allows you to specify multiple values in a WHERE clause.

###### ****Syntax****

SELECT ColumnName(s)

FROM TableName

WHERE ColumnName IN (Value1,Value2...);

###### ****Example****

|  |  |
| --- | --- |
| 1  2 | SELECT \* FROM Employee\_Info  WHERE City IN ('Mumbai', 'Bangalore', 'Hyderabad'); |

**NOTE:** You can also use IN while writing [Nested Queries](https://www.edureka.co/blog/sql-commands#Nested%20Queries).

##### **EXISTS Operator**

The EXISTS operator is used to test if a record exists or not.

###### ****Syntax****

SELECT ColumnName(s)

FROM TableName

WHERE EXISTS

(SELECT ColumnName FROM TableName WHERE condition);

###### ****Example****

|  |  |
| --- | --- |
| 1  2  3 | SELECT EmergencyContactName  FROM Employee\_Info  WHERE EXISTS (SELECT EmergencyContactName FROM Employee\_Info WHERE EmployeeId = 05 AND City = 'Kolkata'); |

##### **ALL Operator**

The ALL operator is used with a WHERE or [HAVING clause](https://www.edureka.co/blog/sql-commands#HAVING%20Clause) and returns TRUE if all of the subquery values meet the condition.

###### ****Syntax****

SELECT ColumnName(s)

FROM TableName

WHERE ColumnName operator ALL

(SELECT ColumnName FROM TableName WHERE condition);

###### ****Example****

|  |  |
| --- | --- |
| 1  2  3 | SELECT EmployeeName  FROM Employee\_Info  WHERE EmployeeID = ALL (SELECT EmployeeID FROM Employee\_Info WHERE City = 'Hyderabad'); |

##### **ANY Operator**

Similar to the ALL operator, the ANY operator is also used with a WHERE or [HAVING clause](https://www.edureka.co/blog/sql-commands#HAVING%20Clause) and returns true if any of the subquery values meet the condition.

###### ****Syntax****

SELECT ColumnName(s)

FROM TableName

WHERE ColumnName operator ANY

(SELECT ColumnName FROM TableName WHERE condition);

###### ****Example****

|  |  |
| --- | --- |
| 1  2  3 | SELECT EmployeeName  FROM Employee\_Info  WHERE EmployeeID = ANY (SELECT EmployeeID FROM Employee\_Info WHERE City = 'Hyderabad' OR City = 'Kolkata'); |

Next, in this article on SQL Commands, let us look into the various Aggregate Functions provided in SQL.

### ****Aggregate Functions****

This section of the article will include the following functions:

* + [MIN()](https://www.edureka.co/blog/sql-commands#MIN())
  + [MAX()](https://www.edureka.co/blog/sql-commands#MAX())
  + [COUNT()](https://www.edureka.co/blog/sql-commands#COUNT())
  + [SUM()](https://www.edureka.co/blog/sql-commands#SUM())
  + [AVG()](https://www.edureka.co/blog/sql-commands#AVG())

#### **MIN() Function**

The MIN function returns the smallest value of the selected column in a table.

##### **Syntax**

SELECT MIN(ColumnName)

FROM TableName

WHERE Condition;

##### **Example**

|  |  |
| --- | --- |
| 1  2 | SELECT MIN(EmployeeID) AS SmallestID  FROM Employee\_Info; |

#### **MAX() Function**

The MAX function returns the largest value of the selected column in a table.

##### **Syntax**

SELECT MAX(ColumnName)

FROM TableName

WHERE Condition;

##### **Example**

|  |  |
| --- | --- |
| 1  2 | SELECT MAX(Salary) AS LargestFees  FROM Employee\_Salary; |

#### **COUNT() Function**

The COUNT function returns the number of rows which match the specified criteria.

[[](https://www.edureka.co/mysql-dba)](https://www.edureka.co/mysql-dba" \t "_blank)

### [MySQL DBA Certification Training](https://www.edureka.co/mysql-dba" \t "_blank)

[Watch The Course Preview](https://www.edureka.co/mysql-dba" \t "_blank)

##### **Syntax**

SELECT COUNT(ColumnName)

FROM TableName

WHERE Condition;

##### **Example**

|  |  |
| --- | --- |
| 1  2 | SELECT COUNT(EmployeeID)  FROM Employee\_Info; |

#### **SUM() Function**

The SUM function returns the total sum of a numeric column that you choose.

##### **Syntax**

SELECT SUM(ColumnName)

FROM TableName

WHERE Condition;

##### **Example**

|  |  |
| --- | --- |
| 1  2 | SELECT SUM(Salary)  FROM Employee\_Salary; |

#### **AVG() Function**

The AVG function returns the average value of a numeric column that you choose.

##### **Syntax**

SELECT AVG(ColumnName)

FROM TableName

WHERE Condition;

##### **Example**

|  |  |
| --- | --- |
| 1  2 | SELECT AVG(Salary)  FROM Employee\_Salary; |

### ****NULL Functions****

The NULL functions are those functions which let you return an alternative value if an expression is NULL. In the SQL Server, the function is **ISNULL()**.

#### **Example**

|  |  |
| --- | --- |
| 1  2 | SELECT EmployeeID \* (Month\_Year\_of\_Salary + ISNULL(Salary, 0))  FROM Employee\_Salary; |

### ****Aliases & Case Statement****

In this section of this article on SQL Commands, you will go through the [Aliases](https://www.edureka.co/blog/sql-commands#Aliases) and [Case statement](https://www.edureka.co/blog/sql-commands#Case%20statement)one after the other.

#### **Aliases**

Aliases are used to give a column/table a temporary name and only exists for a duration of the query.

##### **Syntax**

--Alias Column Syntax

SELECT ColumnName AS AliasName

FROM TableName;

--Alias Table Syntax

SELECT ColumnName(s)

FROM TableName AS AliasName;

##### **Example**

|  |  |
| --- | --- |
| 1  2  3  4  5 | SELECT EmployeeID AS ID, EmployeeName AS EmpName  FROM Employee\_Info;    SELECT EmployeeName AS EmpName, EmergencyContactName AS [Contact Name]  FROM Employee\_Info; |

#### **Case Statement**

This statement goes through all the conditions and returns a value when the first condition is met. So, if no conditions are TRUE, it returns the value in the ELSE clause. Also, if no conditions are true and there is no ELSE part, then it returns NULL.

##### **Syntax**

CASE

WHEN Condition1 THEN Result1

WHEN Condition2 THEN Result2

WHEN ConditionN THEN ResultN

ELSE Result

END;

##### **Example**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | SELECT EmployeeName, City  FROM Employee\_Info  ORDER BY  (CASE      WHEN City IS NULL THEN 'Country is India by default'      ELSE City  END); |

Now, that I have told you a lot about DML commands in this article on SQL Commands, let me just tell you in short about [Nested Queries,](https://www.edureka.co/blog/sql-commands#Nested%20Queries) [Joins](https://www.edureka.co/blog/sql-commands#Joins), [Set Operations](https://www.edureka.co/blog/sql-commands#Set%20Operations), and [Dates & Auto Increment](https://www.edureka.co/blog/sql-commands#Dates%20and%20Auto%20Increment).

## ****SQL Commands: Nested Queries****

**Nested queries**are those queries which have an outer query and inner subquery. So, basically, the subquery is a query which is nested within another query such as [SELECT](https://www.edureka.co/blog/sql-commands#SELECT), [INSERT](https://www.edureka.co/blog/sql-commands#INSERT%20INTO), [UPDATE](https://www.edureka.co/blog/sql-commands#UPDATE) or [DELETE](https://www.edureka.co/blog/sql-commands#DELETE). Refer to the image below:

## Nested Queries - SQL Commands - Edureka****SQL Commands: Joins****

JOINS are used to combine rows from two or more tables, based on a related column between those tables. The following are the types of joins:

* [**INNER JOIN:**](https://www.edureka.co/blog/sql-commands#INNER%20JOIN:) This join returns those records which have matching values in both the tables.
* [**FULL JOIN:**](https://www.edureka.co/blog/sql-commands#FULL%20JOIN:) This join returns all those records which either have a match in the left or the right table.
* [**LEFT JOIN:**](https://www.edureka.co/blog/sql-commands#LEFT%20JOIN:) This join returns records from the left table, and also those records which satisfy the condition from the right table.
* [**RIGHT JOIN:**](https://www.edureka.co/blog/sql-commands#RIGHT%20JOIN:) This join returns records from the right table, and also those records which satisfy the condition from the left table.

Refer to the image below.

## Joins in SQL - SQL Commands - Edureka

Let’s consider the below table apart from the Employee\_Info table, to understand the syntax of joins.

|  |  |  |  |
| --- | --- | --- | --- |
| **TechID** | **EmpID** | **TechName** | **ProjectStartDate** |
| 1 | 10 | DevOps | 04-01-2019 |
| 2 | 11 | Blockchain | 06-07-2019 |
| 3 | 12 | Python | 01-03-2019 |

**INNER JOIN**

#### **Syntax**

SELECT ColumnName(s)

FROM Table1

INNER JOIN Table2 ON Table1.ColumnName = Table2.ColumnName;

#### **Example**

|  |  |
| --- | --- |
| 1  2  3 | SELECT Technologies.TechID, Employee\_Info.EmployeeName  FROM Technologies  INNER JOIN Employee\_Info ON Technologies.EmpID = Employee\_Info.EmpID; |

### ****FULL JOIN****

#### **Syntax**

SELECT ColumnName(s)

FROM Table1

FULL OUTER JOIN Table2 ON Table1.ColumnName = Table2.ColumnName;

#### **Example**

|  |  |
| --- | --- |
| 1  2  3  4 | SELECT Employee\_Info.EmployeeName, Technologies.TechID  FROM Employee\_Info  FULL OUTER JOIN Orders ON Employee\_Info.EmpID=Employee\_Salary.EmpID  ORDER BY Employee\_Info.EmployeeName; |

### ****LEFT JOIN****

#### **Syntax**

SELECT ColumnName(s)

FROM Table1

LEFT JOIN Table2 ON Table1.ColumnName = Table2.ColumnName;

#### **Example**

|  |  |
| --- | --- |
| 1  2  3  4 | SELECT Employee\_Info.EmployeeName, Technologies.TechID  FROM Employee\_Info  LEFT JOIN Technologies ON Employee\_Info.EmployeeID = Technologies.EmpIDID  ORDER BY Employee\_Info.EmployeeName; |

### ****RIGHT JOIN****

#### **Syntax**

SELECT ColumnName(s)

FROM Table1

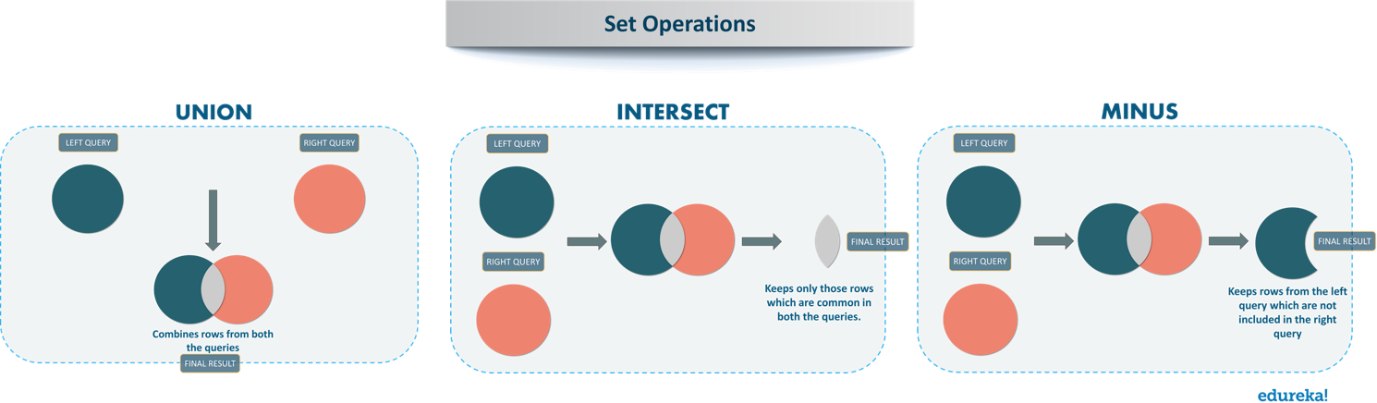
RIGHT JOIN Table2 ON Table1.ColumnName = Table2.ColumnName;

#### **Example**

|  |  |
| --- | --- |
| 1  2  3  4 | SELECT Technologies.TechID  FROM Technologies  RIGHT JOIN Employee\_Info ON Technologies.EmpID = Employee\_Info.EmployeeID  ORDER BY Technologies.TechID; |

## ****SQL Commands: Set Operations****

There are mainly three set operations:[UNION](https://www.edureka.co/blog/sql-commands#UNION), [INTERSECT](https://www.edureka.co/blog/sql-commands#INTERSECT), [EXCEPT](https://www.edureka.co/blog/sql-commands#EXCEPT). You can refer to the image below to understand the set operations in SQL.



### ****UNION****

This operator is used to combine the result-set of two or more [SELECT](https://www.edureka.co/blog/sql-commands#SELECT) statements.

#### **Syntax**

SELECT ColumnName(s) FROM Table1

UNION

SELECT ColumnName(s) FROM Table2;

### ****INTERSECT****

This clause used to combine two [SELECT](https://www.edureka.co/blog/sql-commands#SELECT) statements and return the intersection of the data-sets of both the SELECT statements.

#### **Syntax**

SELECT Column1 , Column2 ....

FROM TableName

WHERE Condition

INTERSECT

SELECT Column1 , Column2 ....

FROM TableName

WHERE Condition

### ****EXCEPT****

This operator returns those tuples that are returned by the first [SELECT](https://www.edureka.co/blog/sql-commands#SELECT) operation, and are not returned by the second SELECT operation.

#### **Syntax**

SELECT ColumnName

FROM TableName

EXCEPT

SELECT ColumnName

FROM TableName;

Next, in this article, let us look into the date functions and auto-increment fields.

## ****SQL Commands: Dates & Auto Increment****

In this section of this article, I will explain to you how to use the [Date functions](https://www.edureka.co/blog/sql-commands#Dates) and also the [Auto-Increment](https://www.edureka.co/blog/sql-commands#Auto-Increment) fields.

### ****Dates****

The following data types are present in a SQL Server to store a date or a date/time value in a database.

|  |  |
| --- | --- |
| **Data Type** | **Format** |
| DATE | YYYY-MM-DD |
| DATETIME | YYYY-MM-DD HH:MI:SS |
| SMALLDATETIME | YYYY-MM-DD HH:MI:SS |
| TIMESTAMP | A Unique Number |

#### **Example**

|  |  |
| --- | --- |
| 1 | SELECT \* FROM Technologies WHERE ProjectStartDate='2019-04-01' |

### ****Auto Increment****

This field generates a unique number automatically when a new record is inserted into a table. The MS SQL Server uses the **IDENTITY** keyword for this feature.

#### **Example**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <span>/\* To define the "EmployeeID" column to be an auto-increment primary key field in the "Employee\_Info" table \*/</span>    <span>CREATE TABLE Employee\_Info (</span>  <span>EmployeeID INT IDENTITY(1,1) PRIMARY KEY,</span>  <span>EmployeeName VARCHAR(255) NOT NULL</span>  <span>EmergencyContactName VARCHAR(255) NOT NULL,</span>  <span>);</span> |

Now, that you guys know the DML commands, let’s move onto our next section in this article on SQL Commands i.e. the DCL commands.

## ****SQL Commands: Data Control Language Commands (DCL)****

This section of the article will give you an insight into the commands which are used to enforce database security in multiple user database environments. The commands are as follows:

* + [GRANT](https://www.edureka.co/blog/sql-commands#GRANT)
  + [REVOKE](https://www.edureka.co/blog/sql-commands#REVOKE)

### ****GRANT****

This command is used to provide access or privileges on the database and its objects to the users.

#### **Syntax**

GRANT PrivilegeName

ON ObjectName

TO {UserName |PUBLIC |RoleName}

[WITH GRANT OPTION];

where,

* **PrivilegeName** – Is the privilege/right/access granted to the user.
* **ObjectName** – Name of a database object like TABLE/VIEW/STORED PROC.
* **UserName** – Name of the user who is given the access/rights/privileges.
* **PUBLIC** – To grant access rights to all users.
* **RoleName** – The name of a set of privileges grouped together.
* **WITH GRANT OPTION** – To give the user access to grant other users with rights.

#### **Example**

|  |  |
| --- | --- |
| 1  2 | -- To grant SELECT permission to Employee\_Info table to user1  GRANT SELECT ON Employee\_Info TO user1; |

### ****REVOKE****

This command is used to withdraw the user’s access privileges given by using the [GRANT](https://www.edureka.co/blog/sql-commands#GRANT) command.

#### **Syntax**

REVOKE PrivilegeName

ON ObjectName

FROM {UserName |PUBLIC |RoleName}

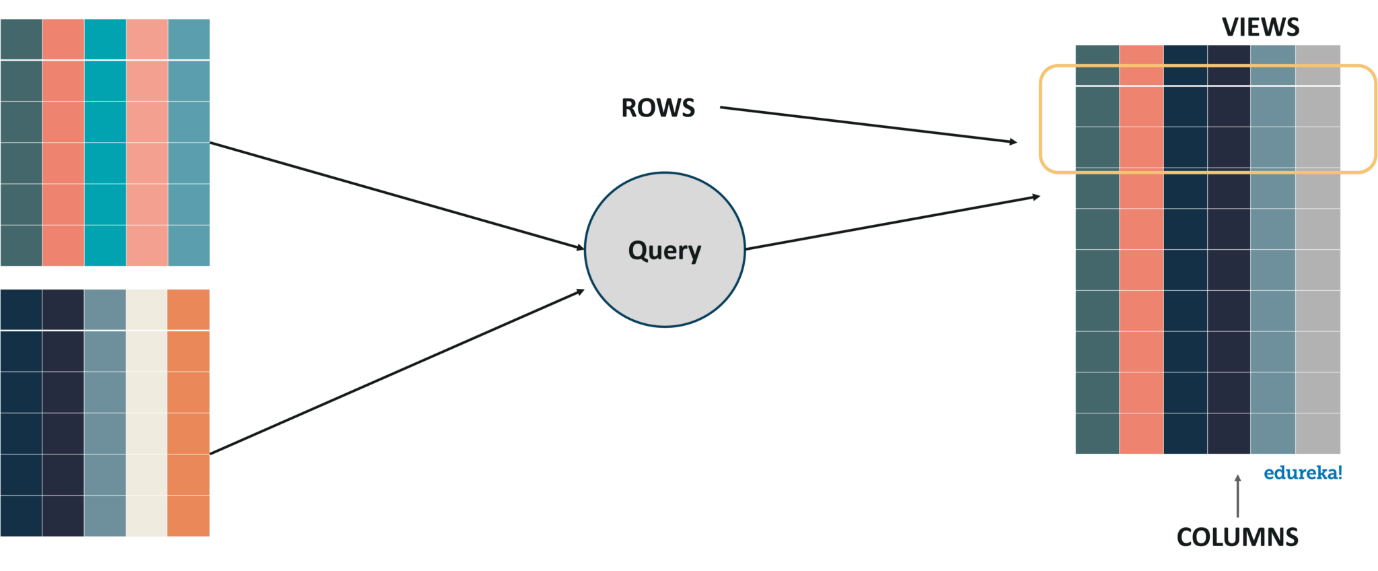
#### **Example**

|  |  |
| --- | --- |
| 1  2 | -- To revoke the granted permission from user1  REVOKE SELECT ON Employee\_Info TO user1; |

Now, next in this article on SQL Commands, I will discuss [Views](https://www.edureka.co/blog/sql-commands#Views), [Stored Procedures](https://www.edureka.co/blog/sql-commands#Stored%20Procedures), and [Triggers](https://www.edureka.co/blog/sql-commands#Triggers).

## ****SQL Commands: Views****

A view in SQL is a single table, which is derived from other tables. So, a view contains rows and columns similar to a real table and has fields from one or more table.



### ****The ‘CREATE VIEW’ statement****

This statement is used to create a view, from a table.

#### **Syntax**

CREATE VIEW ViewName AS

SELECT Column1, Column2, ..., ColumnN

FROM TableName

WHERE Condition;

#### **Example**

|  |  |
| --- | --- |
| 1  2  3  4 | CREATE VIEW [Kolkata Employees] AS  SELECT EmployeeName, PhoneNumber  FROM Employee\_Info  WHERE City = "Kolkata"; |

### ****The ‘CREATE OR REPLACE VIEW’ statement****

This statement is used to update a view.

#### **Syntax**

CREATE VIEW OR REPLACE ViewName AS

SELECT Column1, Column2, ..., ColumnN

FROM TableName

WHERE Condition;

#### **Example**

|  |  |
| --- | --- |
| 1  2  3  4 | CREATE VIEW OR REPLACE [Kolkata Employees] AS  SELECT EmployeeName, PhoneNumber  FROM Employee\_Info  WHERE City = "Kolkata"; |

### ****The ‘DROP VIEW’ statement****

This statement is used to delete a view.

#### **Syntax**

DROP VIEW ViewName;

#### **Example**

|  |  |
| --- | --- |
| 1 | DROP VIEW [Kolkata Employees]; |

## ****SQL Commands: Stored Procedures****

A code which you can save and reuse it again is known as StoredProcedures.

#### **Syntax**

CREATE PROCEDURE ProcedureName

AS

SQLStatement

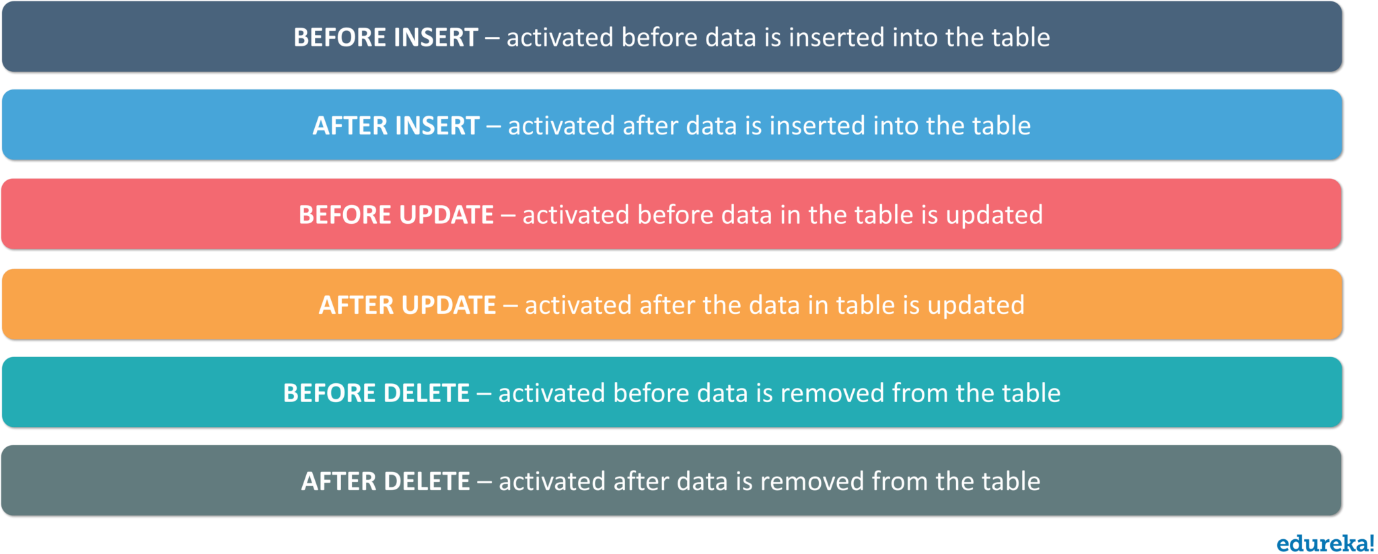
GO;

#### **Example**

|  |  |
| --- | --- |
| 1 | EXEC ProcedureName; |

## ****SQL Commands: Triggers****

Triggers are a set of  SQL statements which are stored in the database catalog. These statements are executed whenever an event associated with a table occurs. So, a **trigger** can be invoked either **BEFORE** or **AFTER** the data is changed by**INSERT**, **UPDATE** or **DELETE** statement. Refer to the image below.



### ****Syntax****

CREATE TRIGGER [TriggerName]

[BEFORE | AFTER]

{INSERT | UPDATE | DELETE}

on [TableName]

[FOR EACH ROW]

[TriggerBody]

Now, let’s move on to the last section of this article on SQL Commands i.e. the Transaction Control Language Commands.

[[](https://www.edureka.co/mysql-dba)](https://www.edureka.co/mysql-dba" \t "_blank)

### [MySQL DBA Certification Training](https://www.edureka.co/mysql-dba" \t "_blank)

[Weekday / Weekend BatchesSee Batch Details](https://www.edureka.co/mysql-dba" \t "_blank)

## ****SQL Commands: Transaction Control Language Commands (TCL)****

This section of the article will give you an insight into the commands which are used to manage transactions in the database. The commands are as follows:

* + [COMMIT](https://www.edureka.co/blog/sql-commands#COMMIT)
  + [ROLLBACK](https://www.edureka.co/blog/sql-commands#ROLLBACK)
  + [SAVEPOINT](https://www.edureka.co/blog/sql-commands#SAVEPOINT)

### ****COMMIT****

This command is used to save the transaction into the database.

#### **Syntax**

COMMIT;

### ****ROLLBACK****

This command is used to restore the database to the last committed state.

#### **Syntax**

ROLLBACK;

**NOTE:** When you use ROLLBACK with SAVEPOINT, then you can directly jump to a savepoint in an ongoing transaction.

***Syntax:***ROLLBACK TO SavepointName;

### ****SAVEPOINT****

This command is used to temporarily save a transaction.  So if you wish to rollback to any point, then you can save that point as a ‘SAVEPOINT’.

#### **Syntax**

SAVEPOINT SAVEPOINTNAME;

Consider the below example to understand the working of transactions in the database.

|  |  |
| --- | --- |
| **EmployeeID** | **EmployeeName** |
| 01 | Ruhaan |
| 02 | Suhana |
| 03 | Aayush |
| 04 | Rashi |

Now, use the below SQL queries to understand the transactions in the database.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | INSERT INTO Employee\_Table VALUES(05, 'Avinash');  COMMIT;  UPDATE Employee\_Table SET name = 'Akash' WHERE id = '05';  SAVEPOINT S1;  INSERT INTO Employee\_Table VALUES(06, 'Sanjana');  SAVEPOINT S2;  INSERT INTO Employee\_Table VALUES(07, 'Sanjay');  SAVEPOINT S3;  INSERT INTO Employee\_Table VALUES(08, 'Veena');  SAVEPOINT S4;  SELECT \* FROM Employee\_Table; |

***The output to the above set of queries would be as follows:***

|  |  |
| --- | --- |
| **EmployeeID** | **EmployeeName** |
| 01 | Ruhaan |
| 02 | Suhana |
| 03 | Aayush |
| 04 | Rashi |
| 05 | Akash |
| 06 | Sanjana |
| 07 | Sanjay |
| 08 | Veena |

***Now, if you rollback to S2 using the below queries, the output is mentioned in the below table.***

|  |  |  |
| --- | --- | --- |
| 1  2 | ROLLBACK TO S2;  SELECT \* FROM Employee\_Table; | |
| **EmployeeID** | **EmployeeName** |
| 01 | Ruhaan |
| 02 | Suhana |
| 03 | Aayush |
| 04 | Rashi |
| 05 | Akash |
| 06 | Sanjana |

By this, I come to the end of this article on SQL Commands. I hope you enjoyed reading this article on SQL Commands. We have seen the different commands that will help you write queries and play around with your databases. If you wish to learn more about [*MySQL*](https://www.edureka.co/blog/what-is-mysql/) and get to know this open source relational database, then check out our[***MySQL DBA Certification Training***](https://www.edureka.co/mysql-dba) which comes with instructor-led live training and real-life project experience. This training will help you understand MySQL in depth and help you achieve mastery over the subject.

# Performing SQL Operations from PL/SQL

Knowledge is of two kinds. We know a subject ourselves, or we know where we can find information upon it. —Samuel Johnson

This chapter shows how PL/SQL supports the SQL commands, functions, and operators that let you manipulate Oracle data.

This chapter contains these topics:

* [Overview of SQL Support in PL/SQL](https://docs.oracle.com/cd/B12037_01/appdev.101/b10807/06_ora.htm#i7112)
* [Performing DML Operations from PL/SQL (INSERT, UPDATE, and DELETE)](https://docs.oracle.com/cd/B12037_01/appdev.101/b10807/06_ora.htm#i45288)
* [Issuing Queries from PL/SQL](https://docs.oracle.com/cd/B12037_01/appdev.101/b10807/06_ora.htm#i45320)
* [Querying Data with PL/SQL](https://docs.oracle.com/cd/B12037_01/appdev.101/b10807/06_ora.htm#i36655)
* [Querying Data with PL/SQL: Explicit Cursor FOR Loops](https://docs.oracle.com/cd/B12037_01/appdev.101/b10807/06_ora.htm#i45507)
* [Using Cursor Variables (REF CURSORs)](https://docs.oracle.com/cd/B12037_01/appdev.101/b10807/06_ora.htm#i7106)
* [Using Cursor Expressions](https://docs.oracle.com/cd/B12037_01/appdev.101/b10807/06_ora.htm#i44913)
* [Overview of Transaction Processing in PL/SQL](https://docs.oracle.com/cd/B12037_01/appdev.101/b10807/06_ora.htm#i7105)
* [Doing Independent Units of Work with Autonomous Transactions](https://docs.oracle.com/cd/B12037_01/appdev.101/b10807/06_ora.htm#i36056)

## Overview of SQL Support in PL/SQL

By extending SQL, PL/SQL offers a unique combination of power and ease of use. You can manipulate Oracle data flexibly and safely because PL/SQL fully supports all SQL data manipulation statements (except EXPLAIN PLAN), transaction control statements, functions, pseudocolumns, and operators. PL/SQL also supports dynamic SQL, which enables you to execute SQL data definition, data control, and session control statements dynamically. In addition, PL/SQL conforms to the current ANSI/ISO SQL standard.

### Data Manipulation

To manipulate Oracle data, you use the INSERT, UPDATE, DELETE, SELECT, and LOCK TABLE commands. INSERT adds new rows of data to database tables; UPDATE modifies rows; DELETE removes unwanted rows; SELECT retrieves rows that meet your search criteria; and LOCK TABLE temporarily limits access to a table.

### Transaction Control

Oracle is transaction oriented; that is, Oracle uses transactions to ensure data integrity. A transaction is a series of SQL data manipulation statements that does a logical unit of work. For example, two UPDATEstatements might credit one bank account and debit another. It is important not to allow one operation to succeed while the other fails.

At the end of a transaction that makes database changes, Oracle makes all the changes permanent or undoes them all. If your program fails in the middle of a transaction, Oracle detects the error and rolls back the transaction, restoring the database to its former state.

You use the COMMIT, ROLLBACK, SAVEPOINT, and SET TRANSACTION commands to control transactions. COMMIT makes permanent any database changes made during the current transaction. ROLLBACK ends the current transaction and undoes any changes made since the transaction began. SAVEPOINT marks the current point in the processing of a transaction. Used with ROLLBACK, SAVEPOINT undoes part of a transaction. SETTRANSACTION sets transaction properties such as read-write access and isolation level.

### SQL Functions

For example, the following example shows some queries that call SQL functions:

DECLARE

job\_count NUMBER;

emp\_count NUMBER;

BEGIN

SELECT COUNT(DISTINCT job\_id) INTO job\_count FROM employees;

SELECT COUNT(\*) INTO emp\_count FROM employees;

END;

/

### SQL Pseudocolumns

PL/SQL recognizes the SQL pseudocolumns: CURRVAL, LEVEL, NEXTVAL, ROWID, and ROWNUM. In PL/SQL, pseudocolumns are only allowed in SQL queries, not in INSERT / UPDATE / DELETE statements, or in other PL/SQL statements such as assignments or conditional tests.

**CURRVAL and NEXTVAL**

A **sequence** is a schema object that generates sequential numbers. When you create a sequence, you can specify its initial value and an increment. CURRVAL returns the current value in a specified sequence.

Before you can reference CURRVAL in a session, you must use NEXTVAL to generate a number. A reference to NEXTVAL stores the current sequence number in CURRVAL. NEXTVAL increments the sequence and returns the next value. To get the current or next value in a sequence, use dot notation:

sequence\_name.CURRVAL

sequence\_name.NEXTVAL

After creating a sequence, you can use it to generate unique sequence numbers for transaction processing. You can use CURRVAL and NEXTVAL only in a SELECT list, the VALUES clause, and the SET clause. The following example shows how to generate a new sequence number and refer to that same number in more than one statement:

CREATE TABLE employees\_temp AS SELECT employee\_id, first\_name FROM employees;

CREATE TABLE employees\_temp2 AS SELECT employee\_id, first\_name FROM employees;

DECLARE

next\_value NUMBER;

BEGIN

-- The NEXTVAL value is the same no matter what table you select from.

SELECT employees\_seq.NEXTVAL INTO next\_value FROM dual;

-- You usually use NEXTVAL to create unique numbers when inserting data.

INSERT INTO employees\_temp VALUES (employees\_seq.NEXTVAL, 'value 1');

-- If you need to store the same value somewhere else, you use CURRVAL.

INSERT INTO employees\_temp2 VALUES (employees\_seq.CURRVAL, 'value 1');

-- Because NEXTVAL values might be referenced by different users and

-- applications, and some NEXTVAL values might not be stored in the

-- database, there might be gaps in the sequence.

END;

/

DROP TABLE employees\_temp;

DROP TABLE employees\_temp2;

Each time you reference the NEXTVAL value of a sequence, the sequence is incremented immediately and permanently, whether you commit or roll back the transaction.

**LEVEL**

You use LEVEL with the SELECT CONNECT BY statement to organize rows from a database table into a tree structure. You might use sequence numbers to give each row a unique identifier, and refer to those identifiers from other rows to set up parent-child relationships.

LEVEL returns the level number of a node in a tree structure. The root is level 1, children of the root are level 2, grandchildren are level 3, and so on.

In the START WITH clause, you specify a condition that identifies the root of the tree. You specify the direction in which the query traverses the tree (down from the root or up from the branches) with the PRIORoperator.

**ROWID**

ROWID returns the rowid (binary address) of a row in a database table. You can use variables of type UROWID to store rowids in a readable format.

When you select or fetch a physical rowid into a UROWID variable, you can use the function ROWIDTOCHAR, which converts the binary value to a character string. You can compare the UROWID variable to the ROWIDpseudocolumn in the WHERE clause of an UPDATE or DELETE statement to identify the latest row fetched from a cursor. For an example, see ["Fetching Across Commits"](https://docs.oracle.com/cd/B12037_01/appdev.101/b10807/06_ora.htm#i3160).

**ROWNUM**

ROWNUM returns a number indicating the order in which a row was selected from a table. The first row selected has a ROWNUM of 1, the second row has a ROWNUM of 2, and so on. If a SELECT statement includes an ORDER BY clause, ROWNUMs are assigned to the retrieved rows before the sort is done; use a subselect (shown in the following example) to get the first n sorted rows.

You can use ROWNUM in an UPDATE statement to assign unique values to each row in a table, or in the WHERE clause of a SELECT statement to limit the number of rows retrieved:

CREATE TABLE employees\_temp AS SELECT \* FROM employees;

DECLARE

CURSOR c1 IS SELECT employee\_id, salary FROM employees\_temp

WHERE salary > 2000 AND ROWNUM <= 10; -- 10 arbitrary rows

CURSOR c2 IS SELECT \* FROM

(SELECT employee\_id, salary FROM employees\_temp

WHERE salary > 2000 ORDER BY salary DESC)

WHERE ROWNUM < 5; -- first 5 rows, in sorted order

BEGIN

-- Each row gets assigned a different number

UPDATE employees\_temp SET employee\_id = ROWNUM;

END;

/

DROP TABLE employees\_temp;

The value of ROWNUM increases only when a row is retrieved, so the only meaningful uses of ROWNUM in a WHERE clause are

... WHERE ROWNUM < constant;

... WHERE ROWNUM <= constant;

### SQL Operators

PL/SQL lets you use all the SQL comparison, set, and row operators in SQL statements. This section briefly describes some of these operators. For more information, see [Oracle Database SQL Reference](https://docs.oracle.com/cd/B12037_01/server.101/b10759/toc.htm).

**Comparison Operators**

Typically, you use comparison operators in the WHERE clause of a data manipulation statement to form **predicates**, which compare one expression to another and yield TRUE, FALSE, or NULL. You can use the comparison operators listed below to form predicates. You can combine predicates using the logical operators AND, OR, and NOT.

| **Operator** | **Description** |
| --- | --- |
| ALL | Compares a value to each value in a list or returned by a subquery and yields TRUE if all of the individual comparisons yield TRUE. |
| ANY, SOME | Compares a value to each value in a list or returned by a subquery and yields TRUE if any of the individual comparisons yields TRUE. |
| BETWEEN | Tests whether a value lies in a specified range. |
| EXISTS | Returns TRUE if a subquery returns at least one row. |
| IN | Tests for set membership. |
| IS NULL | Tests for nulls. |
| LIKE | Tests whether a character string matches a specified pattern, which can include wildcards. |

**Set Operators**

Set operators combine the results of two queries into one result. INTERSECT returns all distinct rows selected by both queries. MINUS returns all distinct rows selected by the first query but not by the second. UNION returns all distinct rows selected by either query. UNION ALL returns all rows selected by either query, including all duplicates.

**Row Operators**

Row operators return or reference particular rows. ALL retains duplicate rows in the result of a query or in an aggregate expression. DISTINCT eliminates duplicate rows from the result of a query or from an aggregate expression. PRIOR refers to the parent row of the current row returned by a tree-structured query.

## Performing DML Operations from PL/SQL (INSERT, UPDATE, and DELETE)

You can write INSERT, UPDATE, and DELETE statements directly in PL/SQL programs, without any special notation:

CREATE table1 AS SELECT object\_name, object\_type FROM user\_objects;

BEGIN

INSERT INTO table1(col1, col2) VALUES('value1','value2');

UPDATE table1 SET col1 = 'another value' WHERE col2 IS NULL;

DELETE FROM table1 WHERE col1 = col2;

COMMIT;

END;

/

DROP table1;

To find out how many rows are affected by these statements, you can check the value of SQL%ROWCOUNT:

SET SERVEROUTPUT ON;

BEGIN

UPDATE employees SET salary = salary \* 1.05 WHERE ...;

dbms\_output.put\_line('Updated ' || SQL%ROWCOUNT || ' salaries.');

END;

/

Wherever you would use literal values, or bind variables in some other programming language, you can directly substitute PL/SQL variables:

CREATE table1 AS SELECT object\_name, object\_type FROM user\_objects;

DECLARE

x VARCHAR2(128) := 'value1';

y NUMBER := 10;

BEGIN

INSERT INTO table1(col1, col2) VALUES(x, x);

UPDATE table1 SET col1 = x WHERE col3 < y;

DELETE FROM table1 WHERE col1 = x;

COMMIT;

END;

/

DROP table1;

With this notation, you can use variables in place of values in the WHERE clause. To use variables in place of table names, column names, and so on, requires the EXECUTE IMMEDIATE statement that is explained in ...

### Overview of Implicit Cursor Attributes

Implicit cursor attributes return information about the execution of an INSERT, UPDATE, DELETE, or SELECT INTO statement. The values of the cursor attributes always refer to the most recently executed SQL statement. Before Oracle opens the SQL cursor, the implicit cursor attributes yield NULL.

**Note:** The SQL cursor has another attribute, %BULK\_ROWCOUNT, designed for use with the FORALL statement. For more information, see ["Counting Rows Affected by FORALL with the %BULK\_ROWCOUNT Attribute"](https://docs.oracle.com/cd/B12037_01/appdev.101/b10807/12_tune.htm#i49056).

#### %FOUND Attribute: Has a DML Statement Changed Rows?

Until a SQL data manipulation statement is executed, %FOUND yields NULL. Thereafter, %FOUND yields TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows, or a SELECT INTO statement returned one or more rows. Otherwise, %FOUND yields FALSE. In the following example, you use %FOUND to insert a row if a delete succeeds:

DELETE FROM emp WHERE empno = my\_empno;

IF SQL%FOUND THEN -- delete succeeded

INSERT INTO new\_emp VALUES (my\_empno, my\_ename, ...);

#### %ISOPEN Attribute: Always FALSE for Implicit Cursors

Oracle closes the SQL cursor automatically after executing its associated SQL statement. As a result, %ISOPEN always yields FALSE.

#### %NOTFOUND Attribute: Has a DML Statement Failed to Change Rows?

%NOTFOUND is the logical opposite of %FOUND. %NOTFOUND yields TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, %NOTFOUND yields FALSE.

#### %ROWCOUNT Attribute: How Many Rows Affected So Far?

%ROWCOUNT yields the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement. %ROWCOUNT yields 0 if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. In the following example, you use %ROWCOUNT to take action if more than ten rows have been deleted:

DELETE FROM emp WHERE ...

IF SQL%ROWCOUNT > 10 THEN -- more than 10 rows were deleted

...

END IF;

If a SELECT INTO statement returns more than one row, PL/SQL raises the predefined exception TOO\_MANY\_ROWS and %ROWCOUNT yields 1, not the actual number of rows that satisfy the query.

#### Guidelines for Using Implicit Cursor Attributes

The values of the cursor attributes always refer to the most recently executed SQL statement, wherever that statement is. It might be in a different scope (for example, in a sub-block). To save an attribute value for later use, assign it to a Boolean variable immediately. Doing other operations, such as procedure calls, might change the value of %NOTFOUND before you can test it.

The %NOTFOUND attribute is not useful in combination with the SELECT INTO statement:

If a SELECT INTO statement fails to return a row, PL/SQL raises the predefined exception NO\_DATA\_FOUND immediately, interrupting the flow of control before you can check %NOTFOUND.

A SELECT INTO statement that calls a SQL aggregate function always returns a value or a null. After such a statement, the %NOTFOUND attribute is always FALSE, so checking it is unnecessary.

## Using PL/SQL Records in SQL INSERT and UPDATE Statements

Instead of listing each field of a PL/SQL record in INSERT and UPDATE statements, you can use PL/SQL records directly. The most convenient technique is to declare the record using a %ROWTYPE attribute, so that it has exactly the same fields as the SQL table:

DECLARE

emp\_rec emp%ROWTYPE;

BEGIN

emp\_rec.eno := 1500;

emp\_rec.ename := 'Steven Hill';

emp\_rec.sal := '40000';

-- A %ROWTYPE value can fill in all the row fields.

INSERT INTO emp VALUES emp\_rec;

-- The fields of a %ROWTYPE can completely replace the table columns.

UPDATE emp SET ROW = emp\_rec WHERE eno = 100;

END;

/

Although this technique integrates PL/SQL variables and types with SQL DML statements, you cannot use PL/SQL records as bind variables in dynamic SQL statements.

|  |
| --- |
| **See Also:**  ["What Is a PL/SQL Record?"](https://docs.oracle.com/cd/B12037_01/appdev.101/b10807/05_colls.htm#i20716) for more information about PL/SQL records. |

## Issuing Queries from PL/SQL

PL/SQL lets you perform queries (SELECT statements in SQL) and access individual fields or entire rows from the result set. Depending on the complexity of the processing that you want to do on the query results, you can use various notations.

### Selecting At Most One Row: SELECT INTO Statement

If you expect a query to only return one row, you can write a regular SQL SELECT statement with an additional INTO clause specifying the PL/SQL variable to hold the result:

If the query might return more than one row, but you do not care about values after the first, you can restrict any result set to a single row by comparing the ROWNUM value:

If the query might return no rows at all, use an exception handler to specify any actions to take when no data is found:

If you just want to check whether a condition exists in your data, you might be able to code the query with the COUNT(\*) operator, which always returns a number and never raises the NO\_DATA\_FOUND exception:

### Selecting Multiple Rows: BULK COLLECT Clause

If you need to bring a large quantity of data into local PL/SQL variables, rather than looping through a result set one row at a time, you can use the BULK COLLECT clause. When you query only certain columns, you can store all the results for each column in a separate collection variable:

SELECT employee\_id, last\_name, salary FROM employees

BULK COLLECT INTO all\_employee\_ids, all\_last\_names, all\_salaries;

When you query all the columns of a table, you can store the entire result set in a collection of records, which makes it convenient to loop through the results and refer to different columns:

SELECT \* FROM employees BULK COLLECT INTO all\_employees;

FOR i IN all\_employees.FIRST .. all\_employees.LAST

LOOP

...

END LOOP;

This technique can be very fast, but also very memory-intensive. If you use it often, you might be able to improve your code by doing more of the work in SQL:

* If you only need to loop once through the result set, use a FOR loop as described in the following sections. This technique avoids the memory overhead of storing a copy of the result set.
* If you are looping through the result set to scan for certain values or filter the results into a smaller set, do this scanning or filtering in the original query instead. You can add more WHERE clauses in simple cases, or use set operators such as INTERSECT and MINUS if you are comparing two or more sets of results.
* If you are looping through the result set and running another query or a DML statement for each result row, you can probably find a more efficient technique. For queries, look at including subqueries or EXISTS or NOT EXISTS clauses in the original query. For DML statements, look at the FORALL statement, which is much faster than coding these statements inside a regular loop.

### Looping Through Multiple Rows: Cursor FOR Loop

Perhaps the most common case of a query is one where you issue the SELECT statement, then immediately loop once through the rows of the result set. PL/SQL lets you use a simple FOR loop for this kind of query:

The iterator variable for the FOR loop does not need to be declared in advance. It is a %ROWTYPE record whose field names match the column names from the query, and that exists only during the loop. When you use expressions rather than explicit column names, use column aliases so that you can refer to the corresponding values inside the loop:

### Performing Complicated Query Processing: Explicit Cursors

For full control over query processing, you can use explicit cursors in combination with the OPEN, FETCH, and CLOSE statements.

You might want to specify a query in one place but retrieve the rows somewhere else, even in another subprogram. Or you might want to choose very different query parameters, such as ORDER BY or GROUP BYclauses, depending on the situation. Or you might want to process some rows differently than others, and so need more than a simple loop.

Because explicit cursors are so flexible, you can choose from different notations depending on your needs. The following sections describe all the query-processing features that explicit cursors provide.

## Querying Data with PL/SQL

In traditional database programming, you process query results using an internal data structure called a **cursor**. In most situations, PL/SQL can manage the cursor for you, so that code to process query results is straightforward and compact. This section discusses how to process both simple queries where PL/SQL manages everything, and complex queries where you interact with the cursor.

### Querying Data with PL/SQL: Implicit Cursor FOR Loop

With PL/SQL, it is very simple to issue a query, retrieve each row of the result into a %ROWTYPE record, and process each row in a loop:

* You include the text of the query directly in the FOR loop.
* PL/SQL creates a record variable with fields corresponding to the columns of the result set.
* You refer to the fields of this record variable inside the loop. You can perform tests and calculations, display output, or store the results somewhere else.

Here is an example that you can run in SQL\*Plus. It does a query to get the name and status of every index that you can access.

BEGIN

FOR item IN

(

SELECT object\_name, status FROM user\_objects WHERE object\_type = 'INDEX'

AND object\_name NOT LIKE '%$%'

)

LOOP

dbms\_output.put\_line('Index = ' || item.object\_name ||

', Status = ' || item.status);

END LOOP;

END;

/

Before each iteration of the FOR loop, PL/SQL fetches into the implicitly declared record.

The sequence of statements inside the loop is executed once for each row that satisfies the query. When you leave the loop, the cursor is closed automatically. The cursor is closed even if you use an EXIT or GOTO statement to leave the loop before all rows are fetched, or an exception is raised inside the loop.

See also: [LOOP Statements](https://docs.oracle.com/cd/B12037_01/appdev.101/b10807/13_elems028.htm#i34785)

### Querying Data with PL/SQL: Explicit Cursor FOR Loops

IIf you need to reference the same query from different parts of the same procedure, you can declare a cursor that specifies the query, and process the results using a FOR loop.

The following PL/SQ block runs two variations of the same query, first finding all the tables you can access, then all the indexes you can access:

DECLARE

CURSOR c1 IS

SELECT object\_name, status FROM user\_objects WHERE object\_type = 'TABLE'

AND object\_name NOT LIKE '%$%';

BEGIN

FOR item IN c1 LOOP

dbms\_output.put\_line('Table = ' || item.object\_name ||

', Status = ' || item.status);

END LOOP;

END;

/

See also: [LOOP Statements](https://docs.oracle.com/cd/B12037_01/appdev.101/b10807/13_elems028.htm#i34785)

### Defining Aliases for Expression Values in a Cursor FOR Loop

In a cursor FOR loop, PL/SQL creates a %ROWTYPE record with fields corresponding to columns in the result set. The fields have the same names as corresponding columns in the SELECT list.

The select list might contain an expression, such as a column plus a constant, or two columns concatenated together. If so, use a column alias to give unique names to the appropriate columns.

In the following example, full\_name and dream\_salary are aliases for expressions in the query:

SET SERVEROUTPUT ON;

BEGIN

FOR item IN

(

SELECT

first\_name || ' ' || last\_name AS full\_name,

salary \* 10 AS dream\_salary

FROM employees

WHERE ROWNUM <= 5

)

LOOP

dbms\_output.put\_line(item.full\_name || ' dreams of making ' ||

item.dream\_salary);

END LOOP;

END;

/

### Overview of Explicit Cursors

When you need precise control over query processing, you can explicitly declare a cursor in the declarative part of any PL/SQL block, subprogram, or package.

You use three commands to control a cursor: OPEN, FETCH, and CLOSE. First, you initialize the cursor with the OPEN statement, which identifies the result set. Then, you can execute FETCH repeatedly until all rows have been retrieved, or you can use the BULK COLLECT clause to fetch all rows at once. When the last row has been processed, you release the cursor with the CLOSE statement.

This technique requires more code than other techniques such as the implicit cursor FOR loop. Its advantage is flexibility. You can:

* Process several queries in parallel by declaring and opening multiple cursors.
* Process multiple rows in a single loop iteration, skip rows, or split the processing into more than one loop.

#### Declaring a Cursor

You must declare a cursor before referencing it in other statements. You give the cursor a name and associate it with a specific query. You can optionally declare a return type for the cursor (such as *table\_name*%ROWTYPE). You can optionally specify parameters that you use in the WHERE clause instead of referring to local variables. These parameters can have default values.

For example, you might declare cursors like these:

DECLARE

CURSOR c1 IS SELECT empno, ename, job, sal FROM emp

WHERE sal > 2000;

CURSOR c2 RETURN dept%ROWTYPE IS

SELECT \* FROM dept WHERE deptno = 10;

The cursor is not a PL/SQL variable: you cannot assign values to a cursor or use it in an expression. Cursors and variables follow the same scoping rules. Naming cursors after database tables is possible but not recommended.

A cursor can take parameters, which can appear in the associated query wherever constants can appear. The formal parameters of a cursor must be IN parameters; they supply values in the query, but do not return any values from the query. You cannot impose the constraint NOT NULL on a cursor parameter.

As the example below shows, you can initialize cursor parameters to default values. You can pass different numbers of actual parameters to a cursor, accepting or overriding the default values as you please. Also, you can add new formal parameters without having to change existing references to the cursor.

DECLARE

CURSOR c1 (low INTEGER DEFAULT 0,

high INTEGER DEFAULT 99) IS SELECT ...

Cursor parameters can be referenced only within the query specified in the cursor declaration. The parameter values are used by the associated query when the cursor is opened.

#### Opening a Cursor

Opening the cursor executes the query and identifies the result set, which consists of all rows that meet the query search criteria. For cursors declared using the FOR UPDATE clause, the OPEN statement also locks those rows. An example of the OPEN statement follows:

DECLARE

CURSOR c1 IS SELECT ename, job FROM emp WHERE sal < 3000;

...

BEGIN

OPEN c1;

...

END;

Rows in the result set are retrieved by the FETCH statement, not when the OPEN statement is executed.

#### Fetching with a Cursor

Unless you use the BULK COLLECT clause (discussed in the next section), the FETCH statement retrieves the rows in the result set one at a time. Each fetch retrieves the current row and advances the cursor to the next row in the result set.

You can store each column in a separate variable, or store the entire row in a record that has the appropriate fields (usually declared using %ROWTYPE):

-- This cursor queries 3 columns.

-- Each column is fetched into a separate variable.

FETCH c1 INTO my\_empno, my\_ename, my\_deptno;

-- This cursor was declared as SELECT \* FROM employees.

-- An entire row is fetched into the my\_employees record, which

-- is declared with the type employees%ROWTYPE.

FETCH c2 INTO my\_employees;

For each column value returned by the query associated with the cursor, there must be a corresponding, type-compatible variable in the INTO list. Typically, you use the FETCH statement in the following way:

LOOP

FETCH c1 INTO my\_record;

EXIT WHEN c1%NOTFOUND;

-- process data record

END LOOP;

The query can reference PL/SQL variables within its scope. Any variables in the query are evaluated only when the cursor is opened. In the following example, each retrieved salary is multiplied by 2, even though factor is incremented after every fetch:

DECLARE

my\_sal employees.salary%TYPE;

my\_job employees.job\_id%TYPE;

factor INTEGER := 2;

CURSOR c1 IS

SELECT factor\*salary FROM employees WHERE job\_id = my\_job;

BEGIN

OPEN c1; -- here factor equals 2

LOOP

FETCH c1 INTO my\_sal;

EXIT WHEN c1%NOTFOUND;

factor := factor + 1; -- does not affect FETCH

END LOOP;

END;

/

To change the result set or the values of variables in the query, you must close and reopen the cursor with the input variables set to their new values.

However, you can use a different INTO list on separate fetches with the same cursor. Each fetch retrieves another row and assigns values to the target variables, as the following example shows:

DECLARE

CURSOR c1 IS SELECT last\_name FROM employees ORDER BY last\_name;

name1 employees.last\_name%TYPE;

name2 employees.last\_name%TYPE;

name3 employees.last\_name%TYPE;

BEGIN

OPEN c1;

FETCH c1 INTO name1; -- this fetches first row

FETCH c1 INTO name2; -- this fetches second row

FETCH c1 INTO name3; -- this fetches third row

CLOSE c1;

END;

/

If you fetch past the last row in the result set, the values of the target variables are undefined.

**Note:** Eventually, the FETCH statement fails to return a row. When that happens, no exception is raised. To detect the failure, use the cursor attribute %FOUND or %NOTFOUND. For more information, see ["Using Cursor Expressions"](https://docs.oracle.com/cd/B12037_01/appdev.101/b10807/06_ora.htm#i44913).

#### Fetching Bulk Data with a Cursor

The BULK COLLECT clause lets you fetch all rows from the result set at once (see ["Retrieving Query Results into Collections with the BULK COLLECT Clause"](https://docs.oracle.com/cd/B12037_01/appdev.101/b10807/12_tune.htm#i49139)). In the following example, you bulk-fetch from a cursor into two collections:

DECLARE

TYPE NumTab IS TABLE OF employees.employee\_id%TYPE;

TYPE NameTab IS TABLE OF employees.last\_name%TYPE;

nums NumTab;

names NameTab;

CURSOR c1 IS

SELECT employee\_id, last\_name

FROM employees

WHERE job\_id = 'ST\_CLERK';

BEGIN

OPEN c1;

FETCH c1 BULK COLLECT INTO nums, names;

-- Here is where you iterate through the elements in the NUMS and

-- NAMES collections.

NULL;

CLOSE c1;

END;

/

#### Closing a Cursor

The CLOSE statement disables the cursor, and the result set becomes undefined. Once a cursor is closed, you can reopen it, which runs the query again with the latest values of any cursor parameters and variables referenced in the WHERE clause. Any other operation on a closed cursor raises the predefined exception INVALID\_CURSOR.

## Using Subqueries

A **subquery** is a query (usually enclosed by parentheses) that appears within another SQL data manipulation statement. The statement acts upon the single value or set of values returned by the subquery. For example:

* You can use a subquery to find the MAX(), MIN(), or AVG() value for a column, and use that single value in a comparison in a WHERE clause.
* You can use a subquery to find a set of values, and use this values in an IN or NOT IN comparison in a WHERE clause. This technique can avoid joins.
* You can filter a set of values with a subquery, and apply other operations like ORDER BY and GROUP BY in the outer query.
* You can use a subquery in place of a table name, in the FROM clause of a query. This technique lets you join a table with a small set of rows from another table, instead of joining the entire tables.
* You can create a table or insert into a table, using a set of rows defined by a subquery.

DECLARE

CURSOR c1 IS

-- The main query returns only rows where the salary is greater than the average salary.

SELECT employee\_id, last\_name FROM employees WHERE salary > (SELECT AVG(salary) FROM employees);

CURSOR c2 IS

-- The subquery returns all the rows in descending order of salary.

-- The main query returns just the top 10 highest-paid employees.

SELECT \* FROM

(SELECT last\_name, salary FROM employees ORDER BY salary DESC, last\_name)

WHERE ROWNUM < 11;

BEGIN

FOR person IN c1

LOOP

dbms\_output.put\_line('Above-average salary: ' || person.last\_name);

END LOOP;

FOR person IN c2

LOOP

dbms\_output.put\_line('Highest paid: ' || person.last\_name || ' $' || person.salary);

END LOOP;

-- The subquery identifies a set of rows to use with CREATE TABLE or INSERT.

EXECUTE IMMEDIATE

'CREATE TABLE temp AS (SELECT \* FROM employees WHERE salary > 5000)';

EXECUTE IMMEDIATE 'DROP TABLE temp';

END;

/

Using a subquery in the FROM clause, the following query returns the number and name of each department with five or more employees:

DECLARE

CURSOR c1 IS

SELECT t1.department\_id, department\_name, staff

FROM departments t1,

(

SELECT department\_id, COUNT(\*) as staff

FROM employees

GROUP BY department\_id

) t2

WHERE

t1.department\_id = t2.department\_id

AND staff >= 5;

BEGIN

FOR dept IN c1

LOOP

dbms\_output.put\_line('Department = ' || dept.department\_name ||

', staff = ' || dept.staff);

END LOOP;

END;

/

## Using Correlated Subqueries

While a subquery is evaluated only once for each table, a **correlated subquery** is evaluated once for each row. The following example returns the name and salary of each employee whose salary exceeds the departmental average. For each row in the table, the correlated subquery computes the average salary for the corresponding epartment.

DECLARE

-- For each department, we find the average salary.

-- Then we find all the employees in that department making

-- more than that average salary.

CURSOR c1 IS

SELECT department\_id, last\_name, salary

FROM employees t

WHERE

salary >

(

SELECT AVG(salary)

FROM employees

WHERE

t.department\_id = department\_id

)

ORDER BY department\_id;

BEGIN

FOR person IN c1

LOOP

dbms\_output.put\_line('Making above-average salary = ' ||

person.last\_name);

END LOOP;

END;

/

### Writing Maintainable PL/SQL Queries

Instead of referring to local variables, you can declare a cursor that accepts parameters, and pass values for those parameters when you open the cursor. If the query is usually issued with certain values, you can make those values the defaults. You can use either positional notation or named notation to pass the parameter values.

**Example 6-1 Passing Parameters to a Cursor FOR Loop**

The following example computes the total wages paid to employees in a specified department.

DECLARE

CURSOR c1 (name VARCHAR2, max\_wage NUMBER) IS

SELECT \* FROM employees WHERE last\_name = name and salary < max\_wage;

BEGIN

FOR person IN c1('Austin', 30000)

LOOP

-- process data record

dbms\_output.put\_line('Name = ' || person.last\_name ||

', salary = ' || person.salary);

END LOOP;

END;

/

**Example 6-2 Passing Parameters to Explicit Cursors**

For example, here are several ways to open a cursor:

DECLARE

emp\_name employees.last\_name%TYPE := 'Austin';

emp\_salary employees.salary%TYPE := 30000;

my\_record employees%ROWTYPE;

CURSOR c1 (name VARCHAR2, max\_wage NUMBER) IS

SELECT \* FROM employees WHERE last\_name = name and salary < max\_wage;

BEGIN

-- Any of the following statements opens the cursor:

-- OPEN c1('Austin', 3000);

-- OPEN c1('Austin', emp\_salary);

-- OPEN c1(emp\_name, 3000);

-- OPEN c1(emp\_name, emp\_salary);

OPEN c1(emp\_name, emp\_salary);

LOOP

FETCH c1 INTO my\_record;

EXIT WHEN c1%NOTFOUND;

-- process data record

dbms\_output.put\_line('Name = ' || my\_record.last\_name ||

', salary = ' || my\_record.salary);

END LOOP;

END;

/

To avoid confusion, use different names for cursor parameters and the PL/SQL variables that you pass into those parameters.

Formal parameters declared with a default value do not need a corresponding actual parameter. If you omit them, they assume their default values when the OPEN statement is executed.

## Using Cursor Attributes

Every explicit cursor and cursor variable has four attributes: %FOUND, %ISOPEN %NOTFOUND, and %ROWCOUNT. When appended to the cursor or cursor variable, these attributes return useful information about the execution of a data manipulation statement. You can use cursor attributes in procedural statements but not in SQL statements.

### Overview of Explicit Cursor Attributes

Explicit cursor attributes return information about the execution of a multi-row query. When an explicit cursor or a cursor variable is opened, the rows that satisfy the associated query are identified and form the result set. Rows are fetched from the result set.

#### %FOUND Attribute: Has a Row Been Fetched?

After a cursor or cursor variable is opened but before the first fetch, %FOUND returns NULL. After any fetches, it returns TRUE if the last fetch returned a row, or FALSE if the last fetch did not return a row. The following example uses %FOUND to select an action:

DECLARE

CURSOR c1 IS SELECT last\_name, salary FROM employees WHERE ROWNUM < 11;

my\_ename employees.last\_name%TYPE;

my\_salary employees.salary%TYPE;

BEGIN

OPEN c1;

LOOP

FETCH c1 INTO my\_ename, my\_salary;

IF c1%FOUND THEN -- fetch succeeded

dbms\_output.put\_line('Name = ' || my\_ename || ', salary = ' ||

my\_salary);

ELSE -- fetch failed, so exit loop

EXIT;

END IF;

END LOOP;

END;

/

If a cursor or cursor variable is not open, referencing it with %FOUND raises the predefined exception INVALID\_CURSOR.

#### %ISOPEN Attribute: Is the Cursor Open?

%ISOPEN returns TRUE if its cursor or cursor variable is open; otherwise, %ISOPEN returns FALSE. The following example uses %ISOPEN to select an action:

DECLARE

CURSOR c1 IS SELECT last\_name, salary FROM employees WHERE ROWNUM < 11;

the\_name employees.last\_name%TYPE;

the\_salary employees.salary%TYPE;

BEGIN

IF c1%ISOPEN = FALSE THEN -- cursor was not already open

OPEN c1;

END IF;

FETCH c1 INTO the\_name, the\_salary;

CLOSE c1;

END;

/

#### %NOTFOUND Attribute: Has a Fetch Failed?

%NOTFOUND is the logical opposite of %FOUND. %NOTFOUND yields FALSE if the last fetch returned a row, or TRUE if the last fetch failed to return a row. In the following example, you use %NOTFOUND to exit a loop when FETCH fails to return a row:

DECLARE

CURSOR c1 IS SELECT last\_name, salary FROM employees WHERE ROWNUM < 11;

my\_ename employees.last\_name%TYPE;

my\_salary employees.salary%TYPE;

BEGIN

OPEN c1;

LOOP

FETCH c1 INTO my\_ename, my\_salary;

IF c1%NOTFOUND THEN -- fetch failed, so exit loop

-- A shorter form of this test is "EXIT WHEN c1%NOTFOUND;"

EXIT;

ELSE -- fetch succeeded

dbms\_output.put\_line('Name = ' || my\_ename || ', salary = ' ||

my\_salary);

END IF;

END LOOP;

END;

/

Before the first fetch, %NOTFOUND returns NULL. If FETCH never executes successfully, the loop is never exited, because the EXIT WHEN statement executes only if its WHEN condition is true. To be safe, you might want to use the following EXIT statement instead:

EXIT WHEN c1%NOTFOUND OR c1%NOTFOUND IS NULL;

If a cursor or cursor variable is not open, referencing it with %NOTFOUND raises an INVALID\_CURSOR exception.

#### %ROWCOUNT Attribute: How Many Rows Fetched So Far?

When its cursor or cursor variable is opened, %ROWCOUNT is zeroed. Before the first fetch, %ROWCOUNT yields 0. Thereafter, it yields the number of rows fetched so far. The number is incremented if the last fetch returned a row. The following example uses %ROWCOUNT to test if more than ten rows have been fetched:

DECLARE

CURSOR c1 IS SELECT last\_name FROM employees WHERE ROWNUM < 11;

name employees.last\_name%TYPE;

BEGIN

OPEN c1;

LOOP

FETCH c1 INTO name;

EXIT WHEN c1%NOTFOUND;

dbms\_output.put\_line(c1%ROWCOUNT || '. ' || name);

IF c1%ROWCOUNT = 5 THEN

dbms\_output.put\_line('--- Fetched 5th record ---');

END IF;

END LOOP;

CLOSE c1;

END;

/

If a cursor or cursor variable is not open, referencing it with %ROWCOUNT raises INVALID\_CURSOR.

[Table 6-1](https://docs.oracle.com/cd/B12037_01/appdev.101/b10807/06_ora.htm" \l "i46335) shows what each cursor attribute returns before and after you execute an OPEN, FETCH, or CLOSE statement.

**Table 6-1****Cursor Attribute Values**

|  |  | **%FOUND** | **%ISOPEN** | **%NOTFOUND** | **%ROWCOUNT** |
| --- | --- | --- | --- | --- | --- |
| OPEN | before | exception | FALSE | exception | exception |
|  | after | NULL | TRUE | NULL | 0 |
| First FETCH | before | NULL | TRUE | NULL | 0 |
|  | after | TRUE | TRUE | FALSE | 1 |
| Next FETCH(es) | before | TRUE | TRUE | FALSE | 1 |
|  | after | TRUE | TRUE | FALSE | data dependent |
| Last FETCH | before | TRUE | TRUE | FALSE | data dependent |
|  | after | FALSE | TRUE | TRUE | data dependent |
| CLOSE | before | FALSE | TRUE | TRUE | data dependent |
|  | after | exception | FALSE | exception | exception |
| Notes:   1. Referencing %FOUND, %NOTFOUND, or %ROWCOUNT before a cursor is opened or after it is closed raises INVALID\_CURSOR. 2. After the first FETCH, if the result set was empty, %FOUND yields FALSE, %NOTFOUND yields TRUE, and %ROWCOUNT yields 0. | | | | | |

## Using Cursor Variables (REF CURSORs)

Like a cursor, a cursor variable points to the current row in the result set of a multi-row query. A cursor variable is more flexible because it is not tied to a specific query. You can open a cursor variable for any query that returns the right set of columns.

You pass a cursor variable as a parameter to local and stored subprograms. Opening the cursor variable in one subprogram, and processing it in a different subprogram, helps to centralize data retrieval. This technique is also useful for multi-language applications, where a PL/SQL subprogram might return a result set to a subprogram written in a different language.

Cursor variables are available to every PL/SQL client. For example, you can declare a cursor variable in a PL/SQL host environment such as an OCI or Pro\*C program, then pass it as an input host variable (bind variable) to PL/SQL. Application development tools such as Oracle Forms and Oracle Reports, which have a PL/SQL engine, can use cursor variables entirely on the client side. Or, you can pass cursor variables back and forth between a client and the database server through remote procedure calls.

### What Are Cursor Variables (REF CURSORs)?

Cursor variables are like pointers to result sets. You use them when you want to perform a query in one subprogram, and process the results in a different subprogram (possibly one written in a different language). A cursor variable has datatype REF CURSOR, and you might see them referred to informally as REF CURSORs.

Unlike an explicit cursor, which always refers to the same query work area, a cursor variable can refer to different work areas. You cannot use a cursor variable where a cursor is expected, or vice versa.

### Why Use Cursor Variables?

You use cursor variables to pass query result sets between PL/SQL stored subprograms and various clients. PL/SQL and its clients share a pointer to the query work area in which the result set is stored. For example, an OCI client, Oracle Forms application, and Oracle database server can all refer to the same work area.

A query work area remains accessible as long as any cursor variable points to it, as you pass the value of a cursor variable from one scope to another. For example, if you pass a host cursor variable to a PL/SQL block embedded in a Pro\*C program, the work area to which the cursor variable points remains accessible after the block completes.

If you have a PL/SQL engine on the client side, calls from client to server impose no restrictions. For example, you can declare a cursor variable on the client side, open and fetch from it on the server side, then continue to fetch from it back on the client side. You can also reduce network traffic by having a PL/SQL block open or close several host cursor variables in a single round trip.

### Declaring REF CURSOR Types and Cursor Variables

To create cursor variables, you define a REF CURSOR type, then declare cursor variables of that type. You can define REF CURSOR types in any PL/SQL block, subprogram, or package. In the following example, you declare a REF CURSOR type that represents a result set from the DEPARTMENTS table:

DECLARE

TYPE DeptCurTyp IS REF CURSOR RETURN departments%ROWTYPE;

REF CURSOR types can be strong (with a return type) or weak (with no return type).

Strong REF CURSOR types are less error prone because the PL/SQL compiler lets you associate a strongly typed cursor variable only with queries that return the right set of columns. Weak REF CURSOR types are more flexible because the compiler lets you associate a weakly typed cursor variable with any query.

Because there is no type checking with a weak REF CURSOR, all such types are interchangeable. Instead of creating a new type, you can use the predefined type SYS\_REFCURSOR.

Once you define a REF CURSOR type, you can declare cursor variables of that type in any PL/SQL block or subprogram.

DECLARE

TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE; -- strong

TYPE GenericCurTyp IS REF CURSOR; -- weak

cursor1 EmpCurTyp;

cursor2 GenericCurTyp;

my\_cursor SYS\_REFCURSOR; -- didn't need to declare a new type above

The following example declares the cursor variable dept\_cv:

DECLARE

TYPE DeptCurTyp IS REF CURSOR RETURN dept%ROWTYPE;

dept\_cv DeptCurTyp; -- declare cursor variable

To avoid declaring the same REF CURSOR type in each subprogram that uses it, you can put the REF CURSOR declaration in a package spec. You can declare cursor variables of that type in the corresponding package body, or within your own procedure or function.

**Example 6-3 Cursor Variable Returning %ROWTYPE**

In the RETURN clause of a REF CURSOR type definition, you can use %ROWTYPE to refer to a strongly typed cursor variable:

DECLARE

TYPE TmpCurTyp IS REF CURSOR RETURN employees%ROWTYPE;

tmp\_cv TmpCurTyp; -- declare cursor variable

TYPE EmpCurTyp IS REF CURSOR RETURN tmp\_cv%ROWTYPE;

emp\_cv EmpCurTyp; -- declare cursor variable

BEGIN

NULL;

END;

/

**Example 6-4 Cursor Variable Returning %TYPE**

You can also use %TYPE to provide the datatype of a record variable:

DECLARE

dept\_rec departments%ROWTYPE; -- declare record variable

TYPE DeptCurTyp IS REF CURSOR RETURN dept\_rec%TYPE;

dept\_cv DeptCurTyp; -- declare cursor variable

BEGIN

NULL;

END;

/

**Example 6-5 Cursor Variable Returning Record Type**

This example specifies a user-defined RECORD type in the RETURN clause:

DECLARE

TYPE EmpRecTyp IS RECORD (

employee\_id NUMBER,

last\_name VARCHAR2(30),

salary NUMBER(7,2));

TYPE EmpCurTyp IS REF CURSOR RETURN EmpRecTyp;

emp\_cv EmpCurTyp; -- declare cursor variable

BEGIN

NULL;

END;

/

#### Passing Cursor Variables As Parameters

You can declare cursor variables as the formal parameters of functions and procedures. The following example defines a REF CURSOR type, then declares a cursor variable of that type as a formal parameter:

DECLARE

TYPE EmpCurTyp IS REF CURSOR RETURN employees%ROWTYPE;

emp EmpCurTyp;

-- Once we have a result set, we can process all the rows

-- inside a single procedure rather than calling a procedure

-- for each row.

PROCEDURE process\_emp\_cv (emp\_cv IN EmpCurTyp) IS

person employees%ROWTYPE;

BEGIN

dbms\_output.put\_line('-----');

dbms\_output.put\_line('Here are the names from the result set:');

LOOP

FETCH emp\_cv INTO person;

EXIT WHEN emp\_cv%NOTFOUND;

dbms\_output.put\_line('Name = ' || person.first\_name ||

' ' || person.last\_name);

END LOOP;

END;

BEGIN

-- First find 10 arbitrary employees.

OPEN emp FOR SELECT \* FROM employees WHERE ROWNUM < 11;

process\_emp\_cv(emp);

CLOSE emp;

-- Then find employees matching a condition.

OPEN emp FOR SELECT \* FROM employees WHERE last\_name LIKE 'R%';

process\_emp\_cv(emp);

CLOSE emp;

END;

/

**Note:** Like all pointers, cursor variables increase the possibility of parameter aliasing. See ["Overloading Subprogram Names"](https://docs.oracle.com/cd/B12037_01/appdev.101/b10807/08_subs.htm#i12352).

### Controlling Cursor Variables: OPEN-FOR, FETCH, and CLOSE

You use three statements to control a cursor variable: OPEN-FOR, FETCH, and CLOSE. First, you OPEN a cursor variable FOR a multi-row query. Then, you FETCH rows from the result set. When all the rows are processed, you CLOSE the cursor variable.

#### Opening a Cursor Variable

The OPEN-FOR statement associates a cursor variable with a multi-row query, executes the query, and identifies the result set.

OPEN {cursor\_variable | :host\_cursor\_variable} FOR

{ select\_statement

| dynamic\_string [USING bind\_argument[, bind\_argument]...] };

The cursor variable can be declared directly in PL/SQL, or in a PL/SQL host environment such as an OCI program.

The SELECT statement for the query can be coded directly in the statement, or can be a string variable or string literal. When you use a string as the query, it can include placeholders for bind variables, and you specify the corresponding values with a USING clause.

**Note:** This section discusses the static SQL case, in which select\_statement is used. For the dynamic SQL case, in which dynamic\_string is used, see ["OPEN-FOR-USING Statement"](https://docs.oracle.com/cd/B12037_01/appdev.101/b10807/13_elems034.htm#i35315).

Unlike cursors, cursor variables take no parameters. Instead, you can pass whole queries (not just parameters) to a cursor variable. The query can reference host variables and PL/SQL variables, parameters, and functions.

The example below opens a cursor variable. Notice that you can apply cursor attributes (%FOUND, %NOTFOUND, %ISOPEN, and %ROWCOUNT) to a cursor variable.

DECLARE

TYPE EmpCurTyp IS REF CURSOR RETURN employees%ROWTYPE;

emp\_cv EmpCurTyp;

BEGIN

IF NOT emp\_cv%ISOPEN THEN

/\* Open cursor variable. \*/

OPEN emp\_cv FOR SELECT \* FROM employees;

END IF;

CLOSE emp\_cv;

END;

/

Other OPEN-FOR statements can open the same cursor variable for different queries. You need not close a cursor variable before reopening it. (Recall that consecutive OPENs of a static cursor raise the predefined exception CURSOR\_ALREADY\_OPEN.) When you reopen a cursor variable for a different query, the previous query is lost.

**Example 6-6 Stored Procedure to Open a Ref Cursor**

Typically, you open a cursor variable by passing it to a stored procedure that declares an IN OUT parameter that is a cursor variable. For example, the following procedure opens a cursor variable:

CREATE PACKAGE emp\_data AS

TYPE EmpCurTyp IS REF CURSOR RETURN employees%ROWTYPE;

PROCEDURE open\_emp\_cv (emp\_cv IN OUT EmpCurTyp);

END emp\_data;

/

CREATE PACKAGE BODY emp\_data AS

PROCEDURE open\_emp\_cv (emp\_cv IN OUT EmpCurTyp) IS

BEGIN

OPEN emp\_cv FOR SELECT \* FROM employees;

END open\_emp\_cv;

END emp\_data;

/

DROP PACKAGE emp\_data;

You can also use a standalone stored procedure to open the cursor variable. Define the REF CURSOR type in a package, then reference that type in the parameter declaration for the stored procedure.

**Example 6-7 Stored Procedure to Open Ref Cursors with Different Queries**

To centralize data retrieval, you can group type-compatible queries in a stored procedure. In the example below, the packaged procedure declares a selector as one of its formal parameters. When called, the procedure opens the cursor variable emp\_cv for the chosen query.

CREATE PACKAGE emp\_data AS

TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;

PROCEDURE open\_emp\_cv (emp\_cv IN OUT EmpCurTyp, choice INT);

END emp\_data;

CREATE PACKAGE BODY emp\_data AS

PROCEDURE open\_emp\_cv (emp\_cv IN OUT EmpCurTyp, choice INT) IS

BEGIN

IF choice = 1 THEN

OPEN emp\_cv FOR SELECT \* FROM emp WHERE comm IS NOT NULL;

ELSIF choice = 2 THEN

OPEN emp\_cv FOR SELECT \* FROM emp WHERE sal > 2500;

ELSIF choice = 3 THEN

OPEN emp\_cv FOR SELECT \* FROM emp WHERE deptno = 20;

END IF;

END;

END emp\_data;

**Example 6-8 Cursor Variable with Different Return Types**

For more flexibility, a stored procedure can execute queries with different return types:

CREATE PACKAGE admin\_data AS

TYPE GenCurTyp IS REF CURSOR;

PROCEDURE open\_cv (generic\_cv IN OUT GenCurTyp, choice INT);

END admin\_data;

CREATE PACKAGE BODY admin\_data AS

PROCEDURE open\_cv (generic\_cv IN OUT GenCurTyp, choice INT) IS

BEGIN

IF choice = 1 THEN

OPEN generic\_cv FOR SELECT \* FROM emp;

ELSIF choice = 2 THEN

OPEN generic\_cv FOR SELECT \* FROM dept;

ELSIF choice = 3 THEN

OPEN generic\_cv FOR SELECT \* FROM salgrade;

END IF;

END;

END admin\_data;

#### **Using a Cursor Variable as a Host Variable**

You can declare a cursor variable in a PL/SQL host environment such as an OCI or Pro\*C program. To use the cursor variable, you must pass it as a host variable to PL/SQL. In the following Pro\*C example, you pass a host cursor variable and selector to a PL/SQL block, which opens the cursor variable for the chosen query:

EXEC SQL BEGIN DECLARE SECTION;

...

/\* Declare host cursor variable. \*/

SQL\_CURSOR generic\_cv;

int choice;

EXEC SQL END DECLARE SECTION;

...

/\* Initialize host cursor variable. \*/

EXEC SQL ALLOCATE :generic\_cv;

...

/\* Pass host cursor variable and selector to PL/SQL block. \*/

EXEC SQL EXECUTE

BEGIN

IF :choice = 1 THEN

OPEN :generic\_cv FOR SELECT \* FROM emp;

ELSIF :choice = 2 THEN

OPEN :generic\_cv FOR SELECT \* FROM dept;

ELSIF :choice = 3 THEN

OPEN :generic\_cv FOR SELECT \* FROM salgrade;

END IF;

END;

END-EXEC;

Host cursor variables are compatible with any query return type. They behave just like weakly typed PL/SQL cursor variables.

#### Fetching from a Cursor Variable

The FETCH statement retrieves rows from the result set of a multi-row query. It works the same with cursor variables as with explicit cursors.

**Example 6-9 Fetching from a Cursor Variable into a Record**

The following example fetches rows one at a time from a cursor variable into a record:

DECLARE

TYPE EmpCurTyp IS REF CURSOR RETURN employees%ROWTYPE;

emp\_cv EmpCurTyp;

emp\_rec employees%ROWTYPE;

BEGIN

OPEN emp\_cv FOR SELECT \* FROM employees WHERE salary < 3000;

LOOP

/\* Fetch from cursor variable. \*/

FETCH emp\_cv INTO emp\_rec;

EXIT WHEN emp\_cv%NOTFOUND; -- exit when last row is fetched

-- process data record

dbms\_output.put\_line('Name = ' || emp\_rec.first\_name || ' ' ||

emp\_rec.last\_name);

END LOOP;

CLOSE emp\_cv;

END;

/

**Example 6-10 Fetching from a Cursor Variable into Collections**

Using the BULK COLLECT clause, you can bulk fetch rows from a cursor variable into one or more collections:

DECLARE

TYPE EmpCurTyp IS REF CURSOR;

TYPE NameList IS TABLE OF employees.last\_name%TYPE;

TYPE SalList IS TABLE OF employees.salary%TYPE;

emp\_cv EmpCurTyp;

names NameList;

sals SalList;

BEGIN

OPEN emp\_cv FOR SELECT last\_name, salary FROM employees WHERE salary < 3000;

FETCH emp\_cv BULK COLLECT INTO names, sals;

CLOSE emp\_cv;

-- Now loop through the NAMES and SALS collections.

FOR i IN names.FIRST .. names.LAST

LOOP

dbms\_output.put\_line('Name = ' || names(i) || ', salary = ' ||

sals(i));

END LOOP;

END;

/

Any variables in the associated query are evaluated only when the cursor variable is opened. To change the result set or the values of variables in the query, reopen the cursor variable with the variables set to new values. You can use a different INTO clause on separate fetches with the same cursor variable. Each fetch retrieves another row from the same result set.

PL/SQL makes sure the return type of the cursor variable is compatible with the INTO clause of the FETCH statement. If there is a mismatch, an error occurs at compile time if the cursor variable is strongly typed, or at run time if it is weakly typed. At run time, PL/SQL raises the predefined exception ROWTYPE\_MISMATCH before the first fetch. If you trap the error and execute the FETCH statement using a different (compatible) INTO clause, no rows are lost.

When you declare a cursor variable as the formal parameter of a subprogram that fetches from the cursor variable, you must specify the IN or IN OUT mode. If the subprogram also opens the cursor variable, you must specify the IN OUT mode.

If you try to fetch from a closed or never-opened cursor variable, PL/SQL raises the predefined exception INVALID\_CURSOR.

#### Closing a Cursor Variable

The CLOSE statement disables a cursor variable and makes the associated result set undefined. Close the cursor variable after the last row is processed.

When declaring a cursor variable as the formal parameter of a subprogram that closes the cursor variable, you must specify the IN or IN OUT mode.

If you try to close an already-closed or never-opened cursor variable, PL/SQL raises the predefined exception INVALID\_CURSOR.

### Reducing Network Traffic When Passing Host Cursor Variables to PL/SQL

When passing host cursor variables to PL/SQL, you can reduce network traffic by grouping OPEN-FOR statements. For example, the following PL/SQL block opens multiple cursor variables in a single round trip:

/\* anonymous PL/SQL block in host environment \*/

BEGIN

OPEN :emp\_cv FOR SELECT \* FROM employees;

OPEN :dept\_cv FOR SELECT \* FROM departments;

OPEN :loc\_cv FOR SELECT \* FROM locations;

END;

This technique might be useful in Oracle Forms, for instance, when you want to populate a multi-block form.

When you pass host cursor variables to a PL/SQL block for opening, the query work areas to which they point remain accessible after the block completes, so your OCI or Pro\*C program can use these work areas for ordinary cursor operations. In the following example, you open several such work areas in a single round trip:

BEGIN

OPEN :c1 FOR SELECT 1 FROM dual;

OPEN :c2 FOR SELECT 1 FROM dual;

OPEN :c3 FOR SELECT 1 FROM dual;

END;

The cursors assigned to c1, c2, and c3 behave normally, and you can use them for any purpose. When finished, release the cursors as follows:

BEGIN

CLOSE :c1;

CLOSE :c2;

CLOSE :c3;

END;

### Avoiding Errors with Cursor Variables

If both cursor variables involved in an assignment are strongly typed, they must have exactly the same datatype (not just the same return type). If one or both cursor variables are weakly typed, they can have different datatypes.

If you try to fetch from, close, or refer to cursor attributes of a cursor variable that does not point to a query work area, PL/SQL raises the INVALID\_CURSOR exception. You can make a cursor variable (or parameter) point to a query work area in two ways:

* OPEN the cursor variable FOR the query.
* Assign to the cursor variable the value of an already OPENed host cursor variable or PL/SQL cursor variable.

If you assign an unopened cursor variable to another cursor variable, the second one remains invalid even after you open the first one.

Be careful when passing cursor variables as parameters. At run time, PL/SQL raises ROWTYPE\_MISMATCH if the return types of the actual and formal parameters are incompatible.

### Restrictions on Cursor Variables

Currently, cursor variables are subject to the following restrictions:

* You cannot declare cursor variables in a package spec. For example, the following declaration is not allowed:
* CREATE PACKAGE emp\_stuff AS
* TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
* emp\_cv EmpCurTyp; -- not allowed
* END emp\_stuff;
* You cannot pass cursor variables to a procedure that is called through a database link.
* If you pass a host cursor variable to PL/SQL, you cannot fetch from it on the server side unless you also open it there on the same server call.
* You cannot use comparison operators to test cursor variables for equality, inequality, or nullity.
* You cannot assign nulls to a cursor variable.
* Database columns cannot store the values of cursor variables. There is no equivalent type to use in a CREATE TABLE statement.
* You cannot store cursor variables in an associative array, nested table, or varray.
* Cursors and cursor variables are not interoperable; that is, you cannot use one where the other is expected. For example, you cannot reference a cursor variable in a cursor FOR loop.

## Using Cursor Expressions

A cursor expression returns a nested cursor. Each row in the result set can contain values as usual, plus cursors produced by subqueries involving the other values in the row. A single query can return a large set of related values retrieved from multiple tables. You can process the result set with nested loops that fetch first from the rows of the result set, then from any nested cursors within those rows.

PL/SQL supports queries with cursor expressions as part of cursor declarations, REF CURSOR declarations and ref cursor variables. You can also use cursor expressions in dynamic SQL queries. Here is the syntax:

CURSOR(subquery)

A nested cursor is implicitly opened when the containing row is fetched from the parent cursor. The nested cursor is closed only when:

* The nested cursor is explicitly closed by the user
* The parent cursor is reexecuted
* The parent cursor is closed
* The parent cursor is canceled
* An error arises during a fetch on one of its parent cursors. The nested cursor is closed as part of the clean-up.

### Restrictions on Cursor Expressions

* You cannot use a cursor expression with an implicit cursor.
* Cursor expressions can appear only:
  + In a SELECT statement that is not nested in any other query expression, except when it is a subquery of the cursor expression itself.
  + As arguments to table functions, in the FROM clause of a SELECT statement.
* Cursor expressions can appear only in the outermost SELECT list of the query specification.
* Cursor expressions cannot appear in view declarations.
* You cannot perform BIND and EXECUTE operations on cursor expressions.

### Example of Cursor Expressions

In this example, we find a specified location ID, and a cursor from which we can fetch all the departments in that location. As we fetch each department's name, we also get another cursor that lets us fetch their associated employee details from another table.

DECLARE

TYPE emp\_cur\_typ IS REF CURSOR;

emp\_cur emp\_cur\_typ;

dept\_name departments.department\_name%TYPE;

emp\_name employees.last\_name%TYPE;

CURSOR c1 IS SELECT

department\_name,

-- The 2nd item in the result set is another result set,

-- which is represented as a ref cursor and labelled "employees".

CURSOR

(

SELECT e.last\_name FROM employees e

WHERE e.department\_id = d.department\_id

) employees

FROM departments d

WHERE department\_name like 'A%';

BEGIN

OPEN c1;

LOOP

FETCH c1 INTO dept\_name, emp\_cur;

EXIT WHEN c1%NOTFOUND;

dbms\_output.put\_line('Department: ' || dept\_name);

-- For each row in the result set, we can process the result

-- set from a subquery. We could pass the ref cursor to a procedure

-- instead of processing it here in the loop.

LOOP

FETCH emp\_cur INTO emp\_name;

EXIT WHEN emp\_cur%NOTFOUND;

dbms\_output.put\_line(' Employee: ' || emp\_name);

END LOOP;

END LOOP;

CLOSE c1;

END;

/

### Constructing REF CURSORs with Cursor Subqueries

You can use cursor subqueries, also know as cursor expressions, to pass sets of rows as parameters to functions. For example, this statement passes a parameter to the StockPivot function consisting of a REF CURSOR that represents the rows returned by the cursor subquery:

SELECT \* FROM TABLE(StockPivot(CURSOR(SELECT \* FROM StockTable)));

Cursor subqueries are often used with table functions, which are explained in ["Setting Up Transformation Pipelines with Table Functions "](https://docs.oracle.com/cd/B12037_01/appdev.101/b10807/12_tune.htm#i52932).

## Overview of Transaction Processing in PL/SQL

This section explains how to do transaction processing with PL/SQL.

You should already be familiar with the idea of [transactions](https://docs.oracle.com/cd/B12037_01/server.101/b10743/transact.htm#CNCPT117), and how to ensure the consistency of a database, such as the [COMMIT](https://docs.oracle.com/cd/B12037_01/server.101/b10759/statements_4010.htm#SQLRF01110), [SAVEPOINT](https://docs.oracle.com/cd/B12037_01/server.101/b10759/statements_10001.htm#SQLRF01701), and [ROLLBACK](https://docs.oracle.com/cd/B12037_01/server.101/b10759/statements_9010.htm#SQLRF01601) statements. These are Oracle features, available through all programming languages, that let multiple users work on the database concurrently, and ensure that each user sees a consistent version of data and that all changes are applied in the right order.

You usually do not need to write extra code to prevent problems with multiple users accessing data concurrently. Oracle uses **locks** to control concurrent access to data, and locks only the minimum amount of data necessary, for as little time as possible. You can request locks on tables or rows if you really do need this level of control. You can choose from several **modes** of locking such as **row share** and **exclusive**.

### Using COMMIT, SAVEPOINT, and ROLLBACK in PL/SQL

You can include COMMIT, SAVEPOINT, and ROLLBACK statements directly in your PL/SQL programs.

The COMMIT statement ends the current transaction, making any changes made during that transaction permanent, and visible to other users.

The ROLLBACK statement ends the current transaction and undoes any changes made during that transaction. If you make a mistake, such as deleting the wrong row from a table, a rollback restores the original data. If you cannot finish a transaction because an exception is raised or a SQL statement fails, a rollback lets you take corrective action and perhaps start over.

SAVEPOINT names and marks the current point in the processing of a transaction. Savepoints let you roll back part of a transaction instead of the whole transaction.

Consider a transaction that transfers money from one bank account to another. It is important that the money come out of one account, and into the other, at exactly the same moment. Otherwise, a problem partway through might make the money be lost from both accounts or be duplicated in both accounts.

BEGIN

UPDATE accts SET bal = my\_bal - debit

WHERE acctno = 7715;

UPDATE accts SET bal = my\_bal + credit

WHERE acctno = 7720;

COMMIT WORK;

END;

Transactions are not tied to PL/SQL BEGIN-END blocks. A block can contain multiple transactions, and a transaction can span multiple blocks.

The optional COMMENT clause lets you specify a comment to be associated with a distributed transaction. If a network or machine fails during the commit, the state of the distributed transaction might be unknown or in doubt. In that case, Oracle stores the text specified by COMMENT in the data dictionary along with the transaction ID. The text must be a quoted literal up to 50 characters long:

COMMIT COMMENT 'In-doubt order transaction; notify Order Entry';

PL/SQL does not support the FORCE clause of SQL, which manually commits an in-doubt distributed transaction.

The following example inserts information about an employee into three different database tables. If an INSERT statement tries to store a duplicate employee number, the predefined exception DUP\_VAL\_ON\_INDEX is raised. To make sure that changes to all three tables are undone, the exception handler executes a ROLLBACK.

DECLARE

emp\_id INTEGER;

BEGIN

SELECT empno, ... INTO emp\_id, ... FROM new\_emp WHERE ...

INSERT INTO emp VALUES (emp\_id, ...);

INSERT INTO tax VALUES (emp\_id, ...);

INSERT INTO pay VALUES (emp\_id, ...);

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

ROLLBACK;

END;

#### Statement-Level Rollbacks

Before executing a SQL statement, Oracle marks an implicit savepoint. Then, if the statement fails, Oracle rolls it back automatically. For example, if an INSERT statement raises an exception by trying to insert a duplicate value in a unique index, the statement is rolled back. Only work started by the failed SQL statement is lost. Work done before that statement in the current transaction is kept.

Oracle can also roll back single SQL statements to break deadlocks. Oracle signals an error to one of the participating transactions and rolls back the current statement in that transaction.

Before executing a SQL statement, Oracle must parse it, that is, examine it to make sure it follows syntax rules and refers to valid schema objects. Errors detected while executing a SQL statement cause a rollback, but errors detected while parsing the statement do not.

The following example marks a savepoint before doing an insert. If the INSERT statement tries to store a duplicate value in the empno column, the predefined exception DUP\_VAL\_ON\_INDEX is raised. In that case, you roll back to the savepoint, undoing just the insert.

DECLARE

emp\_id emp.empno%TYPE;

BEGIN

UPDATE emp SET ... WHERE empno = emp\_id;

DELETE FROM emp WHERE ...

SAVEPOINT do\_insert;

INSERT INTO emp VALUES (emp\_id, ...);

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

ROLLBACK TO do\_insert;

END;

When you roll back to a savepoint, any savepoints marked after that savepoint are erased. The savepoint to which you roll back is not erased. A simple rollback or commit erases all savepoints.

If you mark a savepoint within a recursive subprogram, new instances of the SAVEPOINT statement are executed at each level in the recursive descent, but you can only roll back to the most recently marked savepoint.

Savepoint names are undeclared identifiers. Reusing a savepoint name within a transaction moves the savepoint from its old position to the current point in the transaction. Thus, a rollback to the savepoint affects only the current part of your transaction:

BEGIN

SAVEPOINT my\_point;

UPDATE emp SET ... WHERE empno = emp\_id;

SAVEPOINT my\_point; -- move my\_point to current point

INSERT INTO emp VALUES (emp\_id, ...);

EXCEPTION

WHEN OTHERS THEN

ROLLBACK TO my\_point;

END;

The number of active savepoints for each session is unlimited.

### How Oracle Does Implicit Rollbacks

Before executing an INSERT, UPDATE, or DELETE statement, Oracle marks an implicit savepoint (unavailable to you). If the statement fails, Oracle rolls back to the savepoint. Normally, just the failed SQL statement is rolled back, not the whole transaction. If the statement raises an unhandled exception, the host environment determines what is rolled back.

If you exit a stored subprogram with an unhandled exception, PL/SQL does not assign values to OUT parameters, and does not do any rollback.

### Ending Transactions

You should explicitly commit or roll back every transaction. Whether you issue the commit or rollback in your PL/SQL program or from a client program depends on the application logic. If you do not commit or roll back a transaction explicitly, the client environment determines its final state.

For example, in the SQL\*Plus environment, if your PL/SQL block does not include a COMMIT or ROLLBACK statement, the final state of your transaction depends on what you do after running the block. If you execute a data definition, data control, or COMMIT statement or if you issue the EXIT, DISCONNECT, or QUIT command, Oracle commits the transaction. If you execute a ROLLBACK statement or abort the SQL\*Plus session, Oracle rolls back the transaction.

Oracle precompiler programs roll back the transaction unless the program explicitly commits or rolls back work, and disconnects using the RELEASE parameter:

EXEC SQL COMMIT WORK RELEASE;

### Setting Transaction Properties with SET TRANSACTION

You use the SET TRANSACTION statement to begin a read-only or read-write transaction, establish an isolation level, or assign your current transaction to a specified rollback segment. Read-only transactions are useful for running multiple queries while other users update the same tables.

During a read-only transaction, all queries refer to the same snapshot of the database, providing a multi-table, multi-query, read-consistent view. Other users can continue to query or update data as usual. A commit or rollback ends the transaction. In the example below a store manager uses a read-only transaction to gather sales figures for the day, the past week, and the past month. The figures are unaffected by other users updating the database during the transaction.

DECLARE

daily\_sales REAL;

weekly\_sales REAL;

monthly\_sales REAL;

BEGIN

COMMIT; -- ends previous transaction

SET TRANSACTION READ ONLY NAME 'Calculate sales figures';

SELECT SUM(amt) INTO daily\_sales FROM sales

WHERE dte = SYSDATE;

SELECT SUM(amt) INTO weekly\_sales FROM sales

WHERE dte > SYSDATE - 7;

SELECT SUM(amt) INTO monthly\_sales FROM sales

WHERE dte > SYSDATE - 30;

COMMIT; -- ends read-only transaction

END;

The SET TRANSACTION statement must be the first SQL statement in a read-only transaction and can only appear once in a transaction. If you set a transaction to READ ONLY, subsequent queries see only changes committed before the transaction began. The use of READ ONLY does not affect other users or transactions.

#### Restrictions on SET TRANSACTION

Only the SELECT INTO, OPEN, FETCH, CLOSE, LOCK TABLE, COMMIT, and ROLLBACK statements are allowed in a read-only transaction. Queries cannot be FOR UPDATE.

### Overriding Default Locking

By default, Oracle locks data structures for you automatically, which is a major strength of the Oracle database: different applications can read and write to the same data without harming each other's data or coordinating with each other.

You can request data locks on specific rows or entire tables if you need to override default locking. Explicit locking lets you deny access to data for the duration of a transaction.:

* With the LOCK TABLE statement, you can explicitly lock entire tables.
* With the SELECT FOR UPDATE statement, you can explicitly lock specific rows of a table to make sure they do not change after you have read them. That way, you can check which or how many rows will be affected by an UPDATE or DELETE statement before issuing the statement, and no other application can change the rows in the meantime.

**Using FOR UPDATE**

When you declare a cursor that will be referenced in the CURRENT OF clause of an UPDATE or DELETE statement, you must use the FOR UPDATE clause to acquire exclusive row locks. An example follows:

DECLARE

CURSOR c1 IS SELECT empno, sal FROM emp

WHERE job = 'SALESMAN' AND comm > sal

FOR UPDATE NOWAIT;

The SELECT ... FOR UPDATE statement identifies the rows that will be updated or deleted, then locks each row in the result set. This is useful when you want to base an update on the existing values in a row. In that case, you must make sure the row is not changed by another user before the update.

The optional keyword NOWAIT tells Oracle not to wait if requested rows have been locked by another user. Control is immediately returned to your program so that it can do other work before trying again to acquire the lock. If you omit the keyword NOWAIT, Oracle waits until the rows are available.

All rows are locked when you open the cursor, not as they are fetched. The rows are unlocked when you commit or roll back the transaction. Since the rows are no longer locked, you cannot fetch from a FORUPDATE cursor after a commit. (For a workaround, see ["Fetching Across Commits"](https://docs.oracle.com/cd/B12037_01/appdev.101/b10807/06_ora.htm#i3160).)

When querying multiple tables, you can use the FOR UPDATE clause to confine row locking to particular tables. Rows in a table are locked only if the FOR UPDATE OF clause refers to a column in that table. For example, the following query locks rows in the emp table but not in the dept table:

DECLARE

CURSOR c1 IS SELECT ename, dname FROM emp, dept

WHERE emp.deptno = dept.deptno AND job = 'MANAGER'

FOR UPDATE OF sal;

As the next example shows, you use the CURRENT OF clause in an UPDATE or DELETE statement to refer to the latest row fetched from a cursor:

DECLARE

CURSOR c1 IS SELECT empno, job, sal FROM emp FOR UPDATE;

BEGIN

OPEN c1;

LOOP

FETCH c1 INTO ...

UPDATE emp SET sal = new\_sal WHERE CURRENT OF c1;

END LOOP;

**Using LOCK TABLE**

You use the LOCK TABLE statement to lock entire database tables in a specified lock mode so that you can share or deny access to them.. Row share locks allow concurrent access to a table; they prevent other users from locking the entire table for exclusive use. Table locks are released when your transaction issues a commit or rollback.

LOCK TABLE emp IN ROW SHARE MODE NOWAIT;

The lock mode determines what other locks can be placed on the table. For example, many users can acquire row share locks on a table at the same time, but only one user at a time can acquire an exclusivelock. While one user has an exclusive lock on a table, no other users can insert, delete, or update rows in that table. For more information about lock modes, see [**Oracle Database Application Developer's Guide - Fundamentals**](https://docs.oracle.com/cd/B12037_01/appdev.101/b10795/toc.htm).

A table lock never keeps other users from querying a table, and a query never acquires a table lock. Only if two different transactions try to modify the same row will one transaction wait for the other to complete.

**Fetching Across Commits**

PL/SQL raises an exception if you try to fetch from a FOR UPDATE cursor after doing a commit. The FOR UPDATE clause locks the rows when you open the cursor, and unlocks them when you commit.

DECLARE

CURSOR c1 IS SELECT ename FROM emp FOR UPDATE OF sal;

BEGIN

FOR emp\_rec IN c1 LOOP -- FETCH fails on the second iteration

INSERT INTO temp VALUES ('still going');

COMMIT; -- releases locks

END LOOP;

END;

If you want to fetch across commits, use the ROWID pseudocolumn to mimic the CURRENT OF clause. Select the rowid of each row into a UROWID variable, then use the rowid to identify the current row during subsequent updates and deletes:

DECLARE

CURSOR c1 IS SELECT ename, job, rowid FROM emp;

my\_ename emp.ename%TYPE;

my\_job emp.job%TYPE;

my\_rowid UROWID;

BEGIN

OPEN c1;

LOOP

FETCH c1 INTO my\_ename, my\_job, my\_rowid;

EXIT WHEN c1%NOTFOUND;

UPDATE emp SET sal = sal \* 1.05 WHERE rowid = my\_rowid;

-- this mimics WHERE CURRENT OF c1

COMMIT;

END LOOP;

CLOSE c1;

END;

Because the fetched rows are not locked by a FOR UPDATE clause, other users might unintentionally overwrite your changes. The extra space needed for read consistency is not released until the cursor is closed, which can slow down processing for large updates.

The next example shows that you can use the %ROWTYPE attribute with cursors that reference the ROWID pseudocolumn:

DECLARE

CURSOR c1 IS SELECT ename, sal, rowid FROM emp;

emp\_rec c1%ROWTYPE;

BEGIN

OPEN c1;

LOOP

FETCH c1 INTO emp\_rec;

EXIT WHEN c1%NOTFOUND;

IF ... THEN

DELETE FROM emp WHERE rowid = emp\_rec.rowid;

END IF;

END LOOP;

CLOSE c1;

END;

## Doing Independent Units of Work with Autonomous Transactions

An **autonomous transaction** is an independent transaction started by another transaction, the **main transaction**. Autonomous transactions do SQL operations and commit or roll back, without committing or rolling back the main transaction. For example, if you write auditing data to a log table, you want to commit the audit data even if the operation you are auditing later fails; if something goes wrong recording the audit data, you do not want the main operation to be rolled back.

[Figure 6-1](https://docs.oracle.com/cd/B12037_01/appdev.101/b10807/06_ora.htm#i28274) shows how control flows from the main transaction (MT) to an autonomous transaction (AT) and back again.

**Figure 6-1 Transaction Control Flow**

[Description of the illustration lnpls028.gif](https://docs.oracle.com/cd/B12037_01/appdev.101/b10807/img_text/lnpls028.htm)

### Advantages of Autonomous Transactions

Once started, an autonomous transaction is fully independent. It shares no locks, resources, or commit-dependencies with the main transaction. You can log events, increment retry counters, and so on, even if the main transaction rolls back.

More important, autonomous transactions help you build modular, reusable software components. You can encapsulate autonomous transactions within stored procedures. A calling application does not need to know whether operations done by that stored procedure succeeded or failed.

### Defining Autonomous Transactions

To define autonomous transactions, you use the pragma (compiler directive) AUTONOMOUS\_TRANSACTION. The pragma instructs the PL/SQL compiler to mark a routine as autonomous (independent). In this context, the term routine includes

* Top-level (not nested) anonymous PL/SQL blocks
* Local, standalone, and packaged functions and procedures
* Methods of a SQL object type
* Database triggers

You can code the pragma anywhere in the declarative section of a routine. But, for readability, code the pragma at the top of the section. The syntax follows:

PRAGMA AUTONOMOUS\_TRANSACTION;

In the following example, you mark a packaged function as autonomous:

CREATE PACKAGE banking AS

...

FUNCTION balance (acct\_id INTEGER) RETURN REAL;

END banking;

CREATE PACKAGE BODY banking AS

...

FUNCTION balance (acct\_id INTEGER) RETURN REAL IS

PRAGMA AUTONOMOUS\_TRANSACTION;

my\_bal REAL;

BEGIN

...

END;

END banking;

**Restriction:** You cannot use the pragma to mark all subprograms in a package (or all methods in an object type) as autonomous. Only individual routines can be marked autonomous.

The next example marks a standalone procedure as autonomous:

CREATE PROCEDURE close\_account (acct\_id INTEGER, OUT balance) AS

PRAGMA AUTONOMOUS\_TRANSACTION;

my\_bal REAL;

BEGIN ... END;

The following example marks a PL/SQL block as autonomous:

DECLARE

PRAGMA AUTONOMOUS\_TRANSACTION;

my\_empno NUMBER(4);

BEGIN ... END;

**Restriction:** You cannot mark a nested PL/SQL block as autonomous.

The example below marks a database trigger as autonomous. Unlike regular triggers, autonomous triggers can contain transaction control statements such as COMMIT and ROLLBACK.

CREATE TRIGGER parts\_trigger

BEFORE INSERT ON parts FOR EACH ROW

DECLARE

PRAGMA AUTONOMOUS\_TRANSACTION;

BEGIN

INSERT INTO parts\_log VALUES(:new.pnum, :new.pname);

COMMIT; -- allowed only in autonomous triggers

END;

#### Comparison of Autonomous Transactions and Nested Transactions

Although an autonomous transaction is started by another transaction, it is not a nested transaction:

* It does not share transactional resources (such as locks) with the main transaction.
* It does not depend on the main transaction. For example, if the main transaction rolls back, nested transactions roll back, but autonomous transactions do not.
* Its committed changes are visible to other transactions immediately. (A nested transaction's committed changes are not visible to other transactions until the main transaction commits.)
* Exceptions raised in an autonomous transaction cause a transaction-level rollback, not a statement-level rollback.

#### Transaction Context

The main transaction shares its context with nested routines, but not with autonomous transactions. When one autonomous routine calls another (or itself recursively), the routines share no transaction context. When an autonomous routine calls a non-autonomous routine, the routines share the same transaction context.

#### Transaction Visibility

Changes made by an autonomous transaction become visible to other transactions when the autonomous transaction commits. These changes become visible to the main transaction when it resumes, if its isolation level is set to READ COMMITTED (the default).

If you set the isolation level of the main transaction to SERIALIZABLE, changes made by its autonomous transactions are not visible to the main transaction when it resumes:

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

### Controlling Autonomous Transactions

The first SQL statement in an autonomous routine begins a transaction. When one transaction ends, the next SQL statement begins another transaction. All SQL statements executed since the last commit or rollback make up the current transaction. To control autonomous transactions, use the following statements, which apply only to the current (active) transaction:

* COMMIT
* ROLLBACK [TO savepoint\_name]
* SAVEPOINT savepoint\_name
* SET TRANSACTION

**Note:** Transaction properties set in the main transaction apply only to that transaction, not to its autonomous transactions, and vice versa.

**Entering and Exiting**

When you enter the executable section of an autonomous routine, the main transaction suspends. When you exit the routine, the main transaction resumes.

To exit normally, you must explicitly commit or roll back all autonomous transactions. If the routine (or any routine called by it) has pending transactions, an exception is raised, and the pending transactions are rolled back.

**Committing and Rolling Back**

COMMIT and ROLLBACK end the active autonomous transaction but do not exit the autonomous routine. When one transaction ends, the next SQL statement begins another transaction. A single autonomous routine could contain several autonomous transactions, if it issued several COMMIT statements.

**Using Savepoints**

The scope of a savepoint is the transaction in which it is defined. Savepoints defined in the main transaction are unrelated to savepoints defined in its autonomous transactions. In fact, the main transaction and an autonomous transaction can use the same savepoint names.

You can roll back only to savepoints marked in the current transaction. In an autonomous transaction, you cannot roll back to a savepoint marked in the main transaction. To do so, you must resume the main transaction by exiting the autonomous routine.

When in the main transaction, rolling back to a savepoint marked before you started an autonomous transaction does not roll back the autonomous transaction. Remember, autonomous transactions are fully independent of the main transaction.

**Avoiding Errors with Autonomous Transactions**

To avoid some common errors, keep the following points in mind:

* If an autonomous transaction attempts to access a resource held by the main transaction, a deadlock can occur. Oracle raises an exception in the autonomous transaction, which is rolled back if the exception goes unhandled.
* The Oracle initialization parameter TRANSACTIONS specifies the maximum number of concurrent transactions. That number might be exceeded because an autonomous transaction runs concurrently with the main transaction.
* If you try to exit an active autonomous transaction without committing or rolling back, Oracle raises an exception. If the exception goes unhandled, the transaction is rolled back.

### Using Autonomous Triggers

Among other things, you can use database triggers to log events transparently. Suppose you want to track all inserts into a table, even those that roll back. In the example below, you use a trigger to insert duplicate rows into a shadow table. Because it is autonomous, the trigger can commit changes to the shadow table whether or not you commit changes to the main table.

-- create a main table and its shadow table

CREATE TABLE parts (pnum NUMBER(4), pname VARCHAR2(15));

CREATE TABLE parts\_log (pnum NUMBER(4), pname VARCHAR2(15));

-- create an autonomous trigger that inserts into the

-- shadow table before each insert into the main table

CREATE TRIGGER parts\_trig

BEFORE INSERT ON parts FOR EACH ROW

DECLARE

PRAGMA AUTONOMOUS\_TRANSACTION;

BEGIN

INSERT INTO parts\_log VALUES(:new.pnum, :new.pname);

COMMIT;

END;

-- insert a row into the main table, and then commit the insert

INSERT INTO parts VALUES (1040, 'Head Gasket');

COMMIT;

-- insert another row, but then roll back the insert

INSERT INTO parts VALUES (2075, 'Oil Pan');

ROLLBACK;

-- show that only committed inserts add rows to the main table

SELECT \* FROM parts ORDER BY pnum;

PNUM PNAME

------- ---------------

1040 Head Gasket

-- show that both committed and rolled-back inserts add rows

-- to the shadow table

SELECT \* FROM parts\_log ORDER BY pnum;

PNUM PNAME

------- ---------------

1040 Head Gasket

2075 Oil Pan

Unlike regular triggers, autonomous triggers can execute DDL statements using native dynamic SQL (discussed in [Chapter 7, " Performing SQL Operations with Native Dynamic SQL"](https://docs.oracle.com/cd/B12037_01/appdev.101/b10807/11_dynam.htm#CACDDACH)). In the following example, trigger bonus\_trig drops a temporary database table after table bonus is updated:

CREATE TRIGGER bonus\_trig

AFTER UPDATE ON bonus

DECLARE

PRAGMA AUTONOMOUS\_TRANSACTION; -- enables trigger to perform DDL

BEGIN

EXECUTE IMMEDIATE 'DROP TABLE temp\_bonus';

END;

For more information about database triggers, see [Oracle Database Application Developer's Guide - Fundamentals](https://docs.oracle.com/cd/B12037_01/appdev.101/b10795/toc.htm).

### Calling Autonomous Functions from SQL

A function called from SQL statements must obey certain rules meant to control side effects. (See ["Controlling Side Effects of PL/SQL Subprograms"](https://docs.oracle.com/cd/B12037_01/appdev.101/b10807/08_subs.htm#i22204).) To check for violations of the rules, you can use the pragma RESTRICT\_REFERENCES. The pragma asserts that a function does not read or write database tables or package variables. (For more information, See [Oracle Database Application Developer's Guide - Fundamentals](https://docs.oracle.com/cd/B12037_01/appdev.101/b10795/toc.htm).)

However, by definition, autonomous routines never violate the rules "read no database state" (RNDS) and "write no database state" (WNDS) no matter what they do. This can be useful, as the example below shows. When you call the packaged function log\_msg from a query, it inserts a message into database table debug\_output without violating the rule "write no database state."

-- create the debug table

CREATE TABLE debug\_output (msg VARCHAR2(200));

-- create the package spec

CREATE PACKAGE debugging AS

FUNCTION log\_msg (msg VARCHAR2) RETURN VARCHAR2;

PRAGMA RESTRICT\_REFERENCES(log\_msg, WNDS, RNDS);

END debugging;

-- create the package body

CREATE PACKAGE BODYq debugging AS

FUNCTION log\_msg (msg VARCHAR2) RETURN VARCHAR2 IS

PRAGMA AUTONOMOUS\_TRANSACTION;

BEGIN

-- the following insert does not violate the constraint

-- WNDS because this is an autonomous routine

INSERT INTO debug\_output VALUES (msg);

COMMIT;

RETURN msg;

END;

END debugging;

-- call the packaged function from a query

DECLARE

my\_empno NUMBER(4);

my\_ename VARCHAR2(15);

BEGIN

...

SELECT debugging.log\_msg(ename) INTO my\_ename FROM emp

WHERE empno = my\_empno;

-- even if you roll back in this scope, the insert

-- into 'debug\_output' remains committed because

-- it is part of an autonomous transaction

IF ... THEN

ROLLBACK;

END IF;

END;