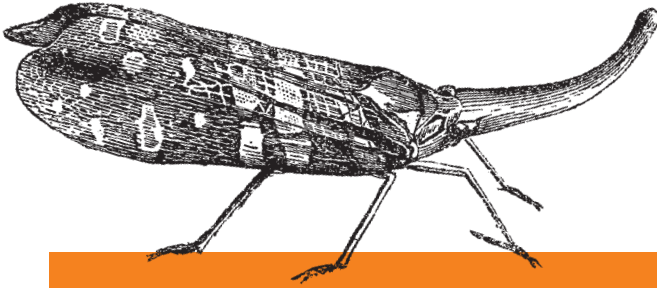


*Putting Oracle SQL to Work*

**2nd Edition**  
Covers Oracle Database 10g



# Mastering Oracle SQL



**O'REILLY®**

*Sanjay Mishra & Alan Beaulieu*

# Set Operations

There are situations when we need to combine the results from two or more SELECT statements. SQL enables us to handle these requirements by using set operations. The result of each SELECT statement can be treated as a set, and SQL set operations can be applied on those sets to arrive at a final result. Oracle SQL supports the following four set operations:

- UNION ALL
- UNION
- MINUS
- INTERSECT

SQL statements containing these set operators are referred to as *compound queries*, and each SELECT statement in a compound query is referred to as a *component query*. Two SELECTs can be combined into a compound query by a set operation only if they satisfy the following two conditions:

- The result sets of both the queries must have the same number of columns.
- The data type of each column in the second result set must match the data type of its corresponding column in the first result set.

These conditions are also referred to as *union compatibility* conditions. The term union compatibility is used even though these conditions apply to other set operations as well. Set operations are often called *vertical joins*, because the result combines data from two or more SELECTs based on columns instead of rows. The generic syntax of a query involving a set operation is:

```
component_query  
{UNION | UNION ALL | MINUS | INTERSECT}  
component_query
```

The keywords UNION, UNION ALL, MINUS, and INTERSECT are set operators. You can have more than two component queries in a composite query; you will always use one less set operator than the number of component queries.

There is an exception to the second union compatibility condition. Two data types do not need to be the same if they are in the same *data type group*. By data type group, we mean the general categories such as numbers, strings, and datetimes. For example, it is ok to have a column in the first component query of data type CHAR, that corresponds to a VARCHAR2 column in the second component query (or vice versa). Oracle performs implicit type conversion in such a case.

However, Oracle will not perform implicit type conversion if corresponding columns in the component queries belong to different data type groups. For example, if a column in the first component query is of data type DATE, and the corresponding column in the second component query is of data type CHAR, Oracle will not perform implicit conversion, and you will get an error as a result of violation of data type compatibility. This is illustrated in the following example:

```
SELECT TO_DATE('12-OCT-03') FROM DUAL
UNION
SELECT '13-OCT-03' FROM DUAL;
```

```
SELECT TO_DATE('12-OCT-03') FROM DUAL
*
```

ERROR at line 1:

ORA-01790: expression must have same datatype as corresponding expression

The following sections discuss syntax, examples, rules, and restrictions for the four set operations.

## Set Operators

The following list briefly describes the four set operations supported by Oracle SQL:

### *UNION ALL*

Combines the results of two SELECT statements into one result set.

### *UNION*

Combines the results of two SELECT statements into one result set, and then eliminates any duplicate rows from that result set.

### *MINUS*

Takes the result set of one SELECT statement, and removes those rows that are also returned by a second SELECT statement. Duplicate rows are eliminated.

### *INTERSECT*

Returns only those rows that are returned by each of two SELECT statements. Duplicate rows are eliminated.

Before moving on to the details on these set operators, let's look at the following two queries, which we'll use as component queries in our subsequent examples. The first query retrieves all the customers in region 5:

```
SELECT cust_nbr, name
FROM customer
WHERE region_id = 5;
```

```
CUST_NBR NAME
-----
1 Cooper Industries
2 Emblazon Corp.
3 Ditech Corp.
4 Flowtech Inc.
5 Gentech Industries
```

The second query retrieves all the customers with the sales representative 'MARTIN':

```
SELECT c.cust_nbr, c.name
FROM customer c
WHERE c.cust_nbr IN (SELECT o.cust_nbr
                     FROM cust_order o, employee e
                     WHERE o.sales_emp_id = e.emp_id
                     AND e.lname = 'MARTIN');
```

```
CUST_NBR NAME
-----
4 Flowtech Inc.
8 Zantech Inc.
```

If you look at the results returned by these two queries, you will notice that there is one common row (for Flowtech Inc.). The following sections discuss the effects of the various set operations between these two result sets.

## UNION ALL

The UNION ALL operator merges the result sets of two component queries. This operation returns rows retrieved by either of the component queries, without eliminating duplicates. The following example illustrates the UNION ALL operation:

```
SELECT cust_nbr, name
FROM customer
WHERE region_id = 5
UNION ALL
SELECT c.cust_nbr, c.name
FROM customer c
WHERE c.cust_nbr IN (SELECT o.cust_nbr
                     FROM cust_order o, employee e
                     WHERE o.sales_emp_id = e.emp_id
                     AND e.lname = 'MARTIN');
```

```

CUST_NBR NAME
-----
      1 Cooper Industries
      2 Emblazon Corp.
      3 Ditech Corp.
      4 Flowtech Inc.
      5 Gentech Industries
      4 Flowtech Inc.
      8 Zantech Inc.

```

7 rows selected.

As you can see from the result set, there is one customer, which is retrieved by both the SELECTs, and therefore appears twice in the result set. The UNION ALL operator simply merges the output of its component queries, without caring about any duplicates in the final result set.

## UNION

The UNION operator returns all distinct rows retrieved by two component queries. The UNION operation eliminates duplicates while merging rows retrieved by either of the component queries. The following example illustrates the UNION operation:

```

SELECT cust_nbr, name
FROM customer
WHERE region_id = 5
UNION
SELECT c.cust_nbr, c.name
FROM customer c
WHERE c.cust_nbr IN (SELECT o.cust_nbr
                     FROM cust_order o, employee e
                     WHERE o.sales_emp_id = e.emp_id
                     AND e.lname = 'MARTIN');

```

```

CUST_NBR NAME
-----
      1 Cooper Industries
      2 Emblazon Corp.
      3 Ditech Corp.
      4 Flowtech Inc.
      5 Gentech Industries
      8 Zantech Inc.

```

6 rows selected.

This query is a modification of the query from the preceding section; the keywords UNION ALL have been replaced with UNION. Now, the result set contains only distinct rows (no duplicates). To eliminate duplicate rows, a UNION operation needs to do some extra tasks as compared to the UNION ALL operation. These extra tasks include sorting and filtering the result set. If you observe carefully, you will notice that the result set of the UNION ALL operation is not sorted, whereas the

result set of the UNION operation is sorted. (The result set of a UNION is sorted on the combination of all the columns in the SELECT list. In the preceding example, the UNION result set will be sorted on the combination cust\_nbr and name.) These extra tasks introduce a performance overhead to the UNION operation. A query involving UNION will take more time than the same query with UNION ALL, even if there are no duplicates to remove.



Unless you have a valid need to retrieve only distinct rows, use UNION ALL instead of UNION for better performance.

## INTERSECT

INTERSECT returns only the rows retrieved by *both* component queries. Compare this with UNION, which returns the rows retrieved by *any* of the component queries. If UNION acts like “OR,” INTERSECT acts like “AND.” For example:

```
SELECT cust_nbr, name
FROM customer
WHERE region_id = 5
INTERSECT
SELECT c.cust_nbr, c.name
FROM customer c
WHERE c.cust_nbr IN (SELECT o.cust_nbr
                     FROM cust_order o, employee e
                     WHERE o.sales_emp_id = e.emp_id
                     AND e.lname = 'MARTIN');
```

```
CUST_NBR NAME
-----
      4 Flowtech Inc.
```

As you saw earlier, “Flowtech Inc.” was the only customer retrieved by both SELECT statements. Therefore, the INTERSECT operator returns just that one row.

## MINUS

MINUS returns all rows from the first SELECT that are not also returned by the second SELECT.



Oracle’s use of MINUS does not follow the ANSI/ISO SQL standard. The corresponding ANSI/ISO SQL keyword is EXCEPT.

The following example illustrates how MINUS works:

```
SELECT cust_nbr, name
FROM customer
WHERE region_id = 5
```

```

MINUS
SELECT c.cust_nbr, c.name
FROM customer c
WHERE c.cust_nbr IN (SELECT o.cust_nbr
                     FROM cust_order o, employee e
                     WHERE o.sales_emp_id = e.emp_id
                     AND e.lname = 'MARTIN');

```

```

CUST_NBR NAME
-----
1 Cooper Industries
2 Emblazon Corp.
3 Ditech Corp.
5 Gentech Industries

```

You might wonder why you don't see "Zantech Inc." in the output. An important thing to note here is that the execution order of component queries in a set operation is from top to bottom. The results of UNION, UNION ALL, and INTERSECT will not change if you alter the ordering of component queries. However, the result of MINUS will be different if you alter the order of the component queries. If you rewrite the previous query by switching the positions of the two SELECTs, you get a completely different result:

```

SELECT c.cust_nbr, c.name
FROM customer c
WHERE c.cust_nbr IN (SELECT o.cust_nbr
                     FROM cust_order o, employee e
                     WHERE o.sales_emp_id = e.emp_id
                     AND e.lname = 'MARTIN')

```

```

MINUS
SELECT cust_nbr, name
FROM customer
WHERE region_id = 5;

```

```

CUST_NBR NAME
-----
8 Zantech Inc.

```

In the second MINUS example, the first component query adds "Flowtech Inc." and "Zantech Inc." to the result set while the second component query removes "Flowtech Inc.", leaving "Zantech Inc." as the sole remaining row.



In a MINUS operation, rows may be returned by the second SELECT that are not also returned by the first. These rows are not included in the output.

# Precedence of Set Operators

If more than two component queries are combined using set operators, then Oracle evaluates the set operators from left to right. In the following example, the UNION is evaluated before the INTERSECT:

```
SELECT cust_nbr, name
FROM customer
WHERE region_id = 5
UNION
SELECT c.cust_nbr, c.name
FROM customer c
WHERE c.cust_nbr IN (SELECT o.cust_nbr
                     FROM cust_order o, employee e
                     WHERE o.sales_emp_id = e.emp_id
                     AND e.lname = 'MARTIN')

INTERSECT
SELECT cust_nbr, name
FROM customer
WHERE region_id = 6;

CUST_NBR NAME
-----
      8 Zantech Inc.
```

To influence a particular order of evaluation of the set operators, you can use parentheses. Looking at the preceding example, if you want the INTERSECT to be evaluated before the UNION, you should introduce parentheses into the query such that the component queries involving the INTERSECT are enclosed in parentheses, as shown in the following example:

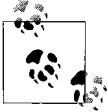
```
SELECT cust_nbr, name
FROM customer
WHERE region_id = 5
UNION
(
SELECT c.cust_nbr, c.name
FROM customer c
WHERE c.cust_nbr IN (SELECT o.cust_nbr
                     FROM cust_order o, employee e
                     WHERE o.sales_emp_id = e.emp_id
                     AND e.lname = 'MARTIN')

INTERSECT
SELECT cust_nbr, name
FROM customer
WHERE region_id = 6
);

CUST_NBR NAME
-----
      1 Cooper Industries
      2 Emblazon Corp.
      3 Ditech Corp.
```

- 4 Flowtech Inc.
- 5 Gentech Industries
- 8 Zantech Inc.

The operation within the parentheses is evaluated first. The result is then combined with the component queries outside the parentheses.



The ANSI/ISO SQL standard gives higher precedence to the INTERSECT operator. However, Oracle, at least through Oracle Database 10g, doesn't implement that higher precedence. All set operations currently have equal precedence.

In the future, Oracle may change the precedence of INTERSECT to comply with the standard. To prepare for that possibility, we recommend using parentheses to control the order of evaluation of set operators whenever you use INTERSECT in a query with any other set operator.

## Comparing Two Tables

Developers, and even DBAs, occasionally need to compare the contents of two tables to determine whether the tables contain the same data. The need to do this is especially common in test environments, as developers may want to compare a set of data generated by a program under test with a set of “known good” data. Comparison of tables is also useful for automated testing purposes, when you have to compare actual results with a given set of expected results. SQL's set operations provide an interesting solution to this problem of comparing two tables.

The following query uses both MINUS and UNION ALL to compare two tables for equality. The query depends on each table having either a primary key or at least one unique index.

```
(SELECT * FROM customer_known_good
MINUS
SELECT * FROM customer_test)
UNION ALL
(SELECT * FROM customer_test
MINUS
SELECT * FROM customer_known_good);
```

You can look at this query as the union of two compound queries. The parentheses ensure that both MINUS operations take place first before the UNION ALL operation is performed. The result of the first MINUS query will be those rows in customer\_known\_good that are not also in customer\_test. The result of the second MINUS query will be those rows in customer\_test that are not also in customer\_known\_good. The UNION ALL operator simply combines these two result sets for convenience. If no rows are returned by this query, then we know that both tables have identical rows. Any rows returned by this query represent differences between the customer\_test and customer\_known\_good tables.

If the possibility exists for one or both tables to contain duplicate rows, you must use a more general form of this query to test the two tables for equality. This more general form uses row counts to detect duplicates:

```
(SELECT c1.*,COUNT(*)
 FROM customer_known_good
 GROUP BY c1.cust_nbr, c1.name...)
MINUS
SELECT c2.*, COUNT(*)
 FROM customer_test c2
 GROUP BY c2.cust_nbr, c2.name...)
UNION ALL
(SELECT c3.*,COUNT(*)
 FROM customer_test c3
 GROUP BY c3.cust_nbr, c3.name...)
MINUS
SELECT c4.*, COUNT(*)
 FROM customer_known_good c4
 GROUP BY c4.cust_nbr, c4.name...)
```

This query is getting complex! The GROUP BY clause (see Chapter 4) for each SELECT must list *all* columns for the table being selected. Any duplicate rows will be grouped together, and the count will reflect the number of duplicates. If the number of duplicates is the same in both tables, the MINUS operations will cancel those rows out. If any rows are different, or if any occurrence counts are different, the resulting rows will be reported by the query.

Let's look at an example to illustrate how this query works. We'll start with the following tables and data:

**DESC customer\_known\_good**

Name	Null?	Type
CUST_NBR	NOT NULL	NUMBER(5)
NAME	NOT NULL	VARCHAR2(30)

**SELECT \* FROM customer\_known\_good;**

CUST_NBR	NAME
1	Sony
1	Sony
2	Samsung
3	Panasonic
3	Panasonic
3	Panasonic

6 rows selected.

**DESC customer\_test**

Name	Null?	Type
CUST_NBR	NOT NULL	NUMBER(5)
NAME	NOT NULL	VARCHAR2(30)

```
SELECT * FROM customer_test;
```

CUST_NBR	NAME
1	Sony
1	Sony
2	Samsung
2	Samsung
3	Panasonic

As you can see the `customer_known_good` and `customer_test` tables have the same structure, but different data. Also notice that none of these tables has a primary or unique key; there are duplicate records in both. The following SQL will compare these two tables effectively:

```
(SELECT c1.*, COUNT(*)
FROM customer_known_good c1
GROUP BY c1.cust_nbr, c1.name
MINUS
SELECT c2.*, COUNT(*)
FROM customer_test c2
GROUP BY c2.cust_nbr, c2.name)
UNION ALL
(SELECT c3.*, COUNT(*)
FROM customer_test c3
GROUP BY c3.cust_nbr, c3.name
MINUS
SELECT c4.*, COUNT(*)
FROM customer_known_good c4
GROUP BY c4.cust_nbr, c4.name);
```

CUST_NBR	NAME	COUNT(*)
2	Samsung	1
3	Panasonic	3
2	Samsung	2
3	Panasonic	1

These results indicate that one table (`customer_known_good`) has one record for “Samsung,” whereas the second table (`customer_test`) has two records for the same customer. Also, one table (`customer_known_good`) has three records for “Panasonic,” whereas the second table (`customer_test`) has one record for the same customer. Both the tables have the same number of rows (two) for “Sony,” and therefore “Sony” doesn’t appear in the output.



Duplicate rows are not possible in tables that have a primary key or at least one unique index. Use the short form of the table comparison query for such tables.

## Using NULLs in Compound Queries

We discussed union compatibility conditions at the beginning of this chapter. The union compatibility issue gets interesting when NULLs are involved. As you know, NULL doesn't have a data type, and NULL can be used in place of a value of any data type. If you purposely select NULL as a column value in a component query, Oracle no longer has two data types to compare to see whether the two component queries are compatible. This is particularly an issue with older Oracle releases. Oracle9i Database, and also later releases of Oracle, are "smart enough" to know which flavor of NULL to use in a compound query. The following examples, generated from an Oracle9i database, demonstrate this:

```
SELECT 1 num, 'DEFINITE' string FROM DUAL
UNION
SELECT NULL num, 'UNKNOWN' string FROM DUAL;
```

```
NUM STRING
-----
1 DEFINITE
  UNKNOWN
```

```
SELECT 1 num, SYSDATE dates FROM DUAL
UNION
SELECT 2 num, NULL dates FROM DUAL;
```

```
NUM DATES
-----
1 06-JAN-02
2
```

If you are using Oracle8i or prior, these queries may cause errors. The examples in the rest of this section are executed against an Oracle8i database.

When your set operation includes a character column that corresponds to a NULL literal, you won't have any problems from the use of NULL. All releases of Oracle handle this case. For example, from an Oracle8i installation:

```
SELECT 1 num, 'DEFINITE' string FROM DUAL
UNION
SELECT 2 num, NULL string FROM DUAL;
```

```
NUM STRING
-----
1 DEFINITE
2
```

Notice that Oracle8i considers the character string 'DEFINITE' from the first component query to be compatible with the NULL value supplied for the corresponding column in the second component query.

However, if a NUMBER or a DATE column of a component query is set to NULL, you must explicitly tell Oracle what “flavor” of NULL to use. Otherwise, you’ll encounter errors. For example:

```
SELECT 1 num, 'DEFINITE' string FROM DUAL
UNION
SELECT NULL num, 'UNKNOWN' string FROM DUAL;
SELECT 1 num, 'DEFINITE' string FROM DUAL
*
```

ERROR at line 1:

ORA-01790: expression must have same datatype as corresponding expression

Note that the use of NULL in the second component query causes a data type mismatch between the first column of the first component query, and the first column of the second component query. Using NULL for a DATE column causes the same problem, as in the following example:

```
SELECT 1 num, SYSDATE dates FROM DUAL
UNION
SELECT 2 num, NULL dates FROM DUAL;
SELECT 1 num, SYSDATE dates FROM DUAL
*
```

ERROR at line 1:

ORA-01790: expression must have same datatype as corresponding expression

In these cases, you need to cast the NULL to a suitable data type to fix the problem, as in the following examples:

```
SELECT 1 num, 'DEFINITE' string FROM DUAL
UNION
SELECT TO_NUMBER(NULL) NUM, 'UNKNOWN' string FROM DUAL;
```

```
NUM STRING
-----
1 DEFINITE
  UNKNOWN
```

```
SELECT 1 num, SYSDATE dates FROM DUAL
UNION
SELECT 2 num, TO_DATE(NULL) dates FROM DUAL;
```

```
NUM DATES
-----
1 06-JAN-02
2
```

Remember, you’ll only encounter these problems of union compatibility when using literal NULL values in Oracle8i and earlier releases. The problems go away beginning with the Oracle9i Database release.

# Rules and Restrictions on Set Operations

Other than the union compatibility conditions discussed at the beginning of the chapter, there are some other rules and restrictions that apply to the set operations. These rules and restrictions are described in this section.

Column names for the result set are derived from the first SELECT:

```
SELECT cust_nbr "Customer ID", name "Customer Name"
FROM customer
WHERE region_id = 5
UNION
SELECT c.cust_nbr "ID", c.name "Name"
FROM customer c
WHERE c.cust_nbr IN (SELECT o.cust_nbr
                     FROM cust_order o, employee e
                     WHERE o.sales_emp_id = e.emp_id
                     AND e.lname = 'MARTIN');
```

```
Customer ID Customer Name
-----
1 Cooper Industries
2 Emblazon Corp.
3 Ditech Corp.
4 Flowtech Inc.
5 Gentech Industries
8 Zantech Inc.
```

6 rows selected.

Although both SELECTs use column aliases, the result set takes the column names from the first SELECT. The same thing happens when you create a view based on a set operation. The column names in the view are taken from the first SELECT:

```
CREATE VIEW v_test_cust AS
SELECT cust_nbr "Customer_ID", name "Customer_Name"
FROM customer
WHERE region_id = 5
UNION
SELECT c.cust_nbr "ID", c.name "Name"
FROM customer c
WHERE c.cust_nbr IN (SELECT o.cust_nbr
                     FROM cust_order o, employee e
                     WHERE o.sales_emp_id = e.emp_id
                     AND e.lname = 'MARTIN');
```

View created.

```
DESC v_test_cust
```

Name	Null?	Type
Customer_ID		NUMBER
Customer_Name		VARCHAR2(45)

If you want to use ORDER BY in a query involving set operations, you must place the ORDER BY at the end of the entire statement. The ORDER BY clause can appear only once at the end of the compound query. The component queries can't have individual ORDER BY clauses. For example:

```
SELECT cust_nbr, name
FROM customer
WHERE region_id = 5
UNION
SELECT emp_id, lname
FROM employee
WHERE lname = 'MARTIN'
ORDER BY cust_nbr;
```

```
  CUST_NBR NAME
-----
         1 Cooper Industries
         2 Emblazon Corp.
         3 Ditech Corp.
         4 Flowtech Inc.
         5 Gentech Industries
       7654 MARTIN
```

6 rows selected.

Note that the column name used in the ORDER BY clause of this query is taken from the first SELECT. You couldn't order these results by emp\_id. If you attempt to ORDER BY emp\_id, you will get an error, as in the following example:

```
SELECT cust_nbr, name
FROM customer
WHERE region_id = 5
UNION
SELECT emp_id, lname
FROM employee
WHERE lname = 'MARTIN'
ORDER BY emp_id;
ORDER BY EMP_ID
      *
```

```
ERROR at line 8:
ORA-00904: invalid column name
```

The ORDER BY clause doesn't recognize the column names of the second SELECT. To avoid confusion over column names, it is a common practice to ORDER BY column positions:

```
SELECT cust_nbr, name
FROM customer
WHERE region_id = 5
UNION
SELECT emp_id, lname
FROM employee
WHERE lname = 'MARTIN'
ORDER BY 1;
```

```

CUST_NBR NAME
-----
      1 Cooper Industries
      2 Emblazon Corp.
      3 Ditech Corp.
      4 Flowtech Inc.
      5 Gentech Industries
7654 MARTIN

```

6 rows selected.

For better readability and maintainability of your queries, we recommend that you explicitly use identical column aliases in all the component queries, and then use these column aliases in the ORDER BY clause.



Unlike ORDER BY, you can use GROUP BY and HAVING clauses in component queries.

The following list summarizes some simple rules, restrictions, and notes that don't require examples:

- Set operations are not permitted on columns of type BLOB, CLOB, BFILE, and VARRAY, nor are set operations permitted on nested table columns.
- Since UNION, INTERSECT, and MINUS operators involve sort operations, they are not allowed on LONG columns. However, UNION ALL is allowed on LONG columns.
- Set operations are not allowed on SELECT statements containing TABLE collection expressions.
- SELECT statements involved in set operations can't use the FOR UPDATE clause.
- The number and size of columns in the SELECT list of component queries are limited by the block size of the database. The total bytes of the columns SELECTed can't exceed one database block.