6
Conditions

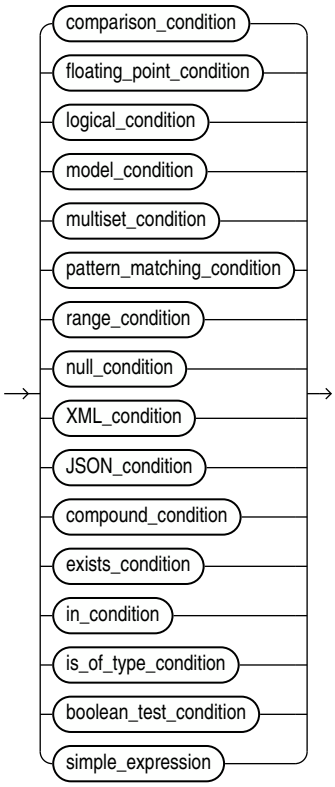
A **condition** specifies a combination of one or more expressions and logical (Boolean) operators and returns a value of `TRUE`, `FALSE`, or `UNKNOWN`.

This chapter contains the following sections:

About SQL Conditions

Conditions can have several forms, as shown in the following syntax.

***condition*::=**



If you have installed Oracle Text, then you can create conditions with the built-in operators that are part of that product, including `CONTAINS`, `CATSEARCH`, and `MATCHES`. For more information on these Oracle Text elements, refer to *Oracle Text Reference*.

The sections that follow describe the various forms of conditions. You must use appropriate condition syntax whenever *`condition`* appears in SQL statements.

You can use a condition in the `WHERE` clause of these statements:

- `DELETE`
- `SELECT`
- `UPDATE`

You can use a condition in any of these clauses of the `SELECT` statement:

- `WHERE`
- `START WITH`
- `CONNECT BY`
- `HAVING`

A condition could be said to be of a logical data type, although Oracle Database does not formally support such a data type.

The following simple condition always evaluates to `TRUE`:

```
1 = 1
```

The following more complex condition adds the `salary` value to the `commission_pct` value (substituting the value 0 for null) and determines whether the sum is greater than the number constant 25000:

```
NVL(salary, 0) + NVL(salary + (salary*commission_pct, 0) > 25000)
```

Logical conditions can combine multiple conditions into a single condition. For example, you can use the `AND` condition to combine two conditions:

```
(1 = 1) AND (5 < 7)
```

Here are some valid conditions:

```
name = 'SMITH'
employees.department_id = departments.department_id
hire_date > '01-JAN-08'
job_id IN ('SA_MAN', 'SA_REP')
salary BETWEEN 5000 AND 10000
commission_pct IS NULL AND salary = 2100
```

Oracle Database does not accept all conditions in all parts of all SQL statements. Refer to the section devoted to a particular SQL statement in this book for information on restrictions on the conditions in that statement.

Condition Precedence

Precedence is the order in which Oracle Database evaluates different conditions in the same expression. When evaluating an expression containing multiple conditions, Oracle evaluates conditions with higher precedence before evaluating those with lower precedence. Oracle evaluates conditions with equal precedence from left to right within an expression, with the following exceptions:

- Left to right evaluation is not guaranteed for multiple conditions connected using `AND`
- Left to right evaluation is not guaranteed for multiple conditions connected using `OR`

Table 6-1 lists the levels of precedence among SQL condition from high to low. Conditions listed on the same line have the same precedence. As the table indicates, Oracle evaluates operators before conditions.

Table 6-1 SQL Condition Precedence

Type of Condition	Purpose
SQL operators are evaluated before SQL conditions	See Operator Precedence
`=, !=, <, >, <=, >=,`	comparison
`IS [NOT] NULL| TRUE|FALSE , LIKE, [NOT] BETWEEN, [NOT] IN, EXISTS, IS OF` *type*	comparison
`NOT`	exponentiation, logical negation
`AND`	conjunction
`OR`	disjunction

Comparison Conditions

Comparison conditions compare one expression with another. The result of such a comparison can be `TRUE`, `FALSE`, or `UNKNOWN`.

Large objects (LOBs) are not supported in comparison conditions. However, you can use PL/SQL programs for comparisons on `CLOB` data.

When comparing numeric expressions, Oracle uses numeric precedence to determine whether the condition compares `NUMBER`, `BINARY_FLOAT`, or `BINARY_DOUBLE` values. Refer to Numeric Precedence for information on numeric precedence.

When comparing character expressions, Oracle uses the rules described in Data Type Comparison Rules . The rules define how the character sets of the expressions are aligned before the comparison, the use of binary or linguistic comparison (collation), the use of blank-padded comparison semantics, and the restrictions resulting from limits imposed on collation keys, including reporting of the error `ORA-12742: unable to create the collation key`.

Two objects of nonscalar type are comparable if they are of the same named type and there is a one-to-one correspondence between their elements. In addition, nested tables of user-defined object types, even if their elements are comparable, must have `MAP` methods defined on them to be used in equality or `IN` conditions.

> **See Also:**
>
> *Oracle Database Object-Relational Developer's Guide* for information on using `MAP` methods to compare objects

Table 6-2 lists comparison conditions.

Table 6-2 Comparison Conditions

Type of Condition	Purpose	Example
=	Equality test.	`SELECT * FROM employees WHERE salary = 2500 ORDER BY employee_id;`
!= ^= <>	Inequality test.	`SELECT * FROM employees WHERE salary != 2500 ORDER BY employee_id;`
> <	Greater-than and less-than tests.	`SELECT * FROM employees WHERE salary > 2500 ORDER BY employee_id;` `SELECT * FROM employees WHERE salary < 2500 ORDER BY employee_id;`
>= <=	Greater-than-or-equal-to and less-than-or-equal-to tests.	`SELECT * FROM employees WHERE salary >= 2500 ORDER BY employee_id;` `SELECT * FROM employees WHERE salary <= 2500 ORDER BY employee_id;`

Table 6-2 (Cont.) Comparison Conditions

Type of Condition	Purpose	Example

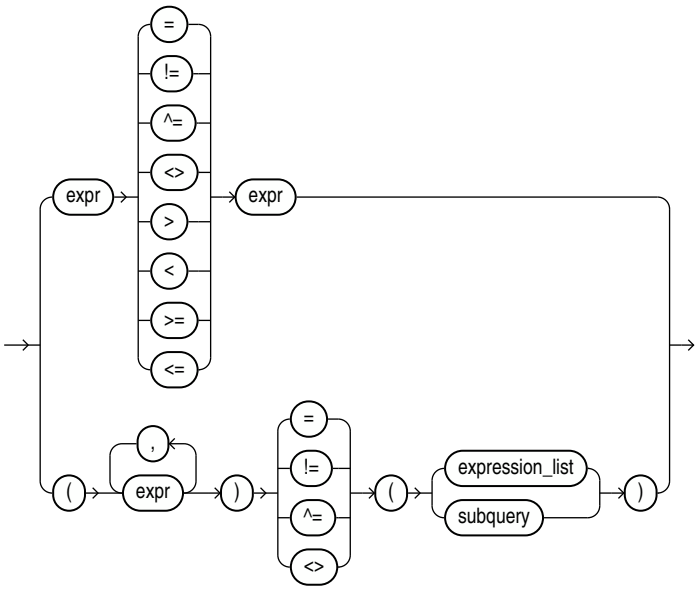
`op ANY`
`op SOME`
	"op" must be one of =, !=, >, <, <=, or >=.
	`op ANY` compares a value on the left side either to each value in a list, or to each value returned by a query, whichever is specified on the right side, using the condition `op`.
	If any of these comparisons returns TRUE, `op ANY` returns `TRUE`.
	If all of these comparisons return `FALSE`, or the subquery on the right side returns no rows, `op ANY` returns `FALSE`. Otherwise, the return value is `UNKNOWN`.
	`op ANY` and `op SOME` are synonymous.
		`SELECT * FROM employees`
		`  WHERE salary = ANY`
		`  (SELECT salary`
		`   FROM employees`
		`  WHERE department_id = 30)`
		`  ORDER BY employee_id;`

`op ALL`	"op" must be one of =, !=, >, <, <=, or >=.
	`op ALL` compares a value on the left side either to each value in a list, or to each value returned by a subquery, whichever is specified on the right side, using the condition `op`.
	If any of these comparisons returns `FALSE`, `op ALL` returns `FALSE`.
	If all of these comparisons return `TRUE`, or the subquery on the right side returns no rows, `op ALL` returns `TRUE` . Otherwise, the return value is `UNKNOWN`.
		`SELECT * FROM employees`
		`  WHERE salary >=`
		`  ALL (1400, 3000)`
		`  ORDER BY employee_id;`

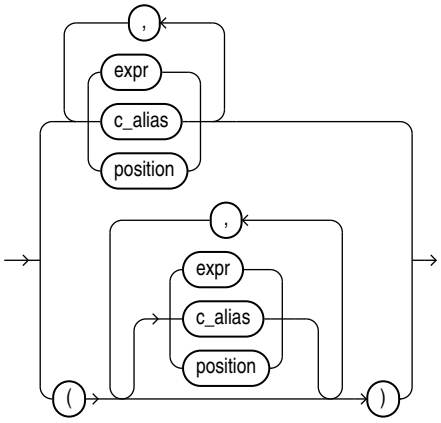
Simple Comparison Conditions

A simple comparison condition specifies a comparison with expressions or subquery results.

***simple_comparison_condition*::=**



***expression_list*::=**



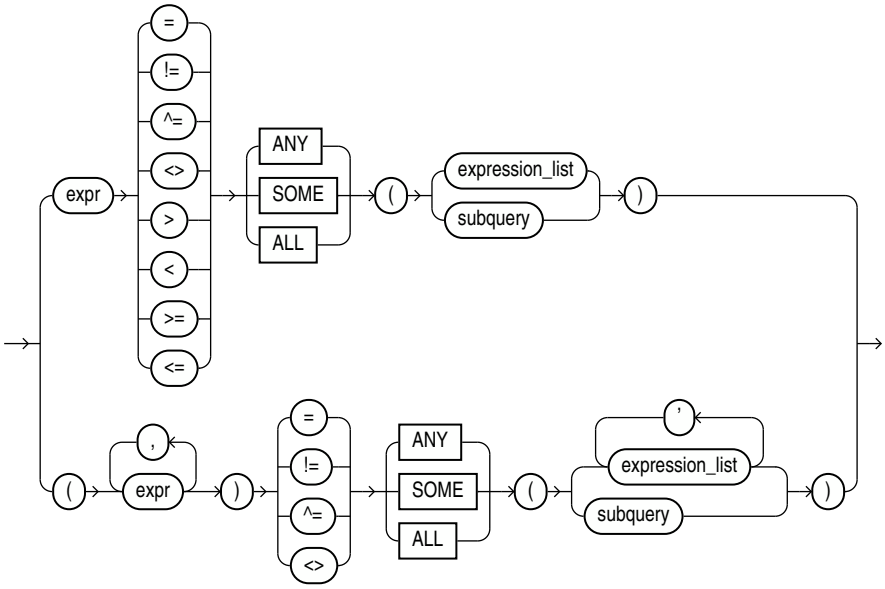
If you use the lower form of this condition with a single expression to the left of the operator, then you can use the upper or lower form of *expression_list*. If you use the lower form of this condition with multiple expressions to the left of the operator, then you must use the lower form of *expression_list*. In either case, the expressions in *expression_list* must match in number and data type the expressions to the left of the operator. If you specify *subquery*, then the values returned by the subquery must match in number and data type the expressions to the left of the operator.

> **See Also:**
>
> Expression Lists for more information about combining expressions and SELECT for information about subqueries

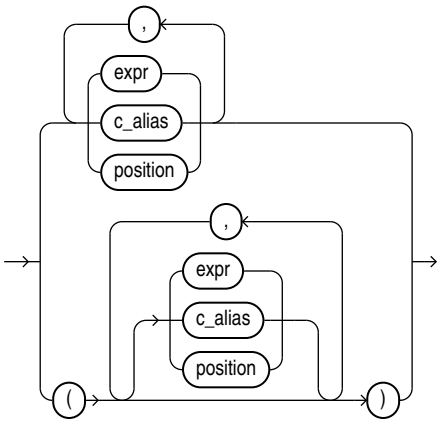
Group Comparison Conditions

A group comparison condition specifies a comparison with any or all members in a list or subquery.

***group_comparison_condition*::=**



***expression_list*::=**



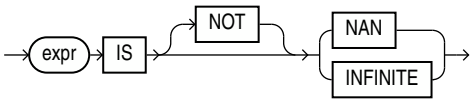
If you use the upper form of this condition (with a single expression to the left of the operator), then you must use the upper form of *expression_list*. If you use the lower form of this condition (with multiple expressions to the left of the operator), then you must use the lower form of *expression_list*, and the expressions in each *expression_list* must match in number and data type the expressions to the left of the operator. If you specify *subquery*, then the values returned by the subquery must match in number and data type the expressions to the left of the operator.

> **See Also:**
>
> - Expression Lists
> - SELECT
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules for comparison conditions

Floating-Point Conditions

The floating-point conditions let you determine whether an expression is infinite or is the undefined result of an operation (is not a number or `NaN`).

***floating_point_condition*::=**



In both forms of floating-point condition, *`expr`* must resolve to a numeric data type or to any data type that can be implicitly converted to a numeric data type. Table 6-3 describes the floating-point conditions.

Table 6-3 Floating-Point Conditions

Type of Condition	Operation	Example
`IS [NOT] NAN`	Returns `TRUE` if *`expr`* is the special value `NaN` when `NOT` is not specified. Returns `TRUE` if *`expr`* is not the special value `NaN` when `NOT` is specified.	`SELECT COUNT(*) FROM employees WHERE commission_pct IS NOT NAN;`
`IS [NOT] INFINITE`	Returns `TRUE` if *`expr`* is the special value `+INF` or `-INF` when `NOT` is not specified. Returns `TRUE` if *`expr`* is neither `+INF` nor `-INF` when `NOT` is specified.	`SELECT last_name FROM employees WHERE salary IS NOT INFINITE;`

> **See Also:**
>
> - Floating-Point Numbers for more information on the Oracle implementation of floating-point numbers
> - Implicit Data Conversion for more information on how Oracle converts floating-point data types

Logical Conditions

A logical condition combines the results of two component conditions to produce a single result based on them or to invert the result of a single condition. Table 6-4 lists logical conditions.

Table 6-4 Logical Conditions

Type of Condition	Operation	Examples

NOT	Returns TRUE if the following condition is FALSE. Returns FALSE if it is TRUE. If it is UNKNOWN, then it remains UNKNOWN.	`SELECT *`
		`  FROM employees`
		`  WHERE NOT (job_id IS NULL)`
		`  ORDER BY employee_id;`
		`SELECT *`
		`  FROM employees`
		`  WHERE NOT`
		`  (salary BETWEEN 1000 AND 2000)`
		`  ORDER BY employee_id;`

AND	Returns TRUE if both component conditions are TRUE. Returns FALSE if either is FALSE. Otherwise returns UNKNOWN.	`SELECT *`
		`  FROM employees`
		`  WHERE job_id = 'PU_CLERK'`
		`  AND department_id = 30`
		`  ORDER BY employee_id;`

OR	Returns TRUE if either component condition is TRUE. Returns FALSE if both are FALSE. Otherwise returns UNKNOWN.	`SELECT *`
		`  FROM employees`
		`  WHERE job_id = 'PU_CLERK'`
		`  OR department_id = 10`
		`  ORDER BY employee_id;`

Table 6-5 shows the result of applying the NOT condition to an expression.

Table 6-5 NOT Truth Table

--	TRUE	FALSE	UNKNOWN
NOT	FALSE	TRUE	UNKNOWN

Table 6-6 shows the results of combining the AND condition to two expressions.

Table 6-6 AND Truth Table

AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

For example, in the WHERE clause of the following SELECT statement, the AND logical condition is used to ensure that only those hired before 2004 and earning more than $2500 a month are returned:

```
SELECT * FROM employees
WHERE hire_date < TO_DATE('01-JAN-2004', 'DD-MON-YYYY')
  AND salary > 2500
  ORDER BY employee_id;
```

Table 6-7 shows the results of applying OR to two expressions.

Table 6-7 OR Truth Table

OR	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

For example, the following query returns employees who have a 40% commission rate or a salary greater than $20,000:

```
SELECT employee_id FROM employees
   WHERE commission_pct = .4 OR salary > 20000
   ORDER BY employee_id;
```

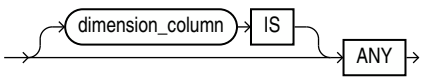
Model Conditions

Model conditions can be used only in the MODEL clause of a SELECT statement.

IS ANY Condition

The IS ANY condition can be used only in the *model_clause* of a SELECT statement. Use this condition to qualify all values of a dimension column, including NULL.

***is_any_condition*::=**



The condition always returns a Boolean value of TRUE in order to qualify all values of the column.

> **See Also:**
>
> *model_clause* and Model Expressions for information

Example

The following example sets sales for each product for year 2000 to 0:

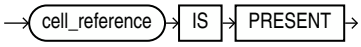
```
SELECT country, prod, year, s
  FROM sales_view_ref
  MODEL
    PARTITION BY (country)
    DIMENSION BY (prod, year)
    MEASURES (sale s)
    IGNORE NAV
    UNIQUE DIMENSION
    RULES UPSERT SEQUENTIAL ORDER
    (
      s[ANY, 2000] = 0
    )
  ORDER BY country, prod, year;

COUNTRY      PROD                                    YEAR          S
----------   ----------------------------------   --------   ---------
France       Mouse Pad                               1998     2509.42
France       Mouse Pad                               1999     3678.69
France       Mouse Pad                               2000           0
France       Mouse Pad                               2001     3269.09
France       Standard Mouse                          1998     2390.83
France       Standard Mouse                          1999     2280.45
France       Standard Mouse                          2000           0
France       Standard Mouse                          2001     2164.54
Germany      Mouse Pad                               1998     5827.87
Germany      Mouse Pad                               1999     8346.44
Germany      Mouse Pad                               2000           0
Germany      Mouse Pad                               2001     9535.08
Germany      Standard Mouse                          1998     7116.11
Germany      Standard Mouse                          1999     6263.14
Germany      Standard Mouse                          2000           0
Germany      Standard Mouse                          2001     6456.13

16 rows selected.
```

The preceding example requires the view `sales_view_ref`. Refer to The MODEL clause: Examples to create this view.

IS PRESENT Condition

***is_present_condition*::=**

The `IS PRESENT` condition can be used only in the *`model_clause`* of a `SELECT` statement. Use this condition to test whether the cell referenced is present prior to the execution of the *`model_clause`*.



The condition returns `TRUE` if the cell exists prior to the execution of the *`model_clause`* and `FALSE` if it does not.

> **See Also:**
>
> *model_clause* and Model Expressions for information

Example

In the following example, if sales of the Mouse Pad for year 1999 exist, then sales of the Mouse Pad for year 2000 is set to sales of the Mouse Pad for year 1999. Otherwise, sales of the Mouse Pad for year 2000 is set to 0.

```
SELECT country, prod, year, s
  FROM sales_view_ref
  MODEL
    PARTITION BY (country)
    DIMENSION BY (prod, year)
    MEASURES (sale s)
    IGNORE NAV
    UNIQUE DIMENSION
    RULES UPSERT SEQUENTIAL ORDER
    (
      s['Mouse Pad', 2000] =
        CASE WHEN s['Mouse Pad', 1999] IS PRESENT
             THEN s['Mouse Pad', 1999]
             ELSE 0
        END
    )
  ORDER BY country, prod, year;

COUNTRY       PROD                                    YEAR          S
----------    ----------------------------------      --------   ---------
France        Mouse Pad                                  1998     2509.42
France        Mouse Pad                                  1999     3678.69
France        Mouse Pad                                  2000     3678.69
France        Mouse Pad                                  2001     3269.09
France        Standard Mouse                             1998     2390.83
France        Standard Mouse                             1999     2280.45
France        Standard Mouse                             2000     1274.31
France        Standard Mouse                             2001     2164.54
Germany       Mouse Pad                                  1998     5827.87
Germany       Mouse Pad                                  1999     8346.44
Germany       Mouse Pad                                  2000     8346.44
Germany       Mouse Pad                                  2001     9535.08
Germany       Standard Mouse                             1998     7116.11
Germany       Standard Mouse                             1999     6263.14
Germany       Standard Mouse                             2000     2637.31
Germany       Standard Mouse                             2001     6456.13
16 rows selected.
```

The preceding example requires the view `sales_view_ref`. Refer to The MODEL clause: Examples to create this view.

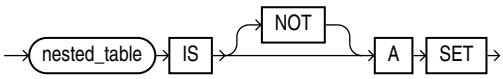
Multiset Conditions

Multiset conditions test various aspects of nested tables.

IS A SET Condition

Use `IS A SET` conditions to test whether a specified nested table is composed of unique elements. The condition returns `UNKNOWN` if the nested table is `NULL`. Otherwise, it returns `TRUE` if the nested table is a set, even if it is a nested table of length zero, and `FALSE` otherwise.

***is_a_set_condition*::=**



Example

The following example selects from the table `customers_demo` those rows in which the `cust_address_ntab` nested table column contains unique elements:

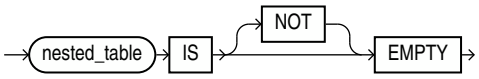
```
SELECT customer_id, cust_address_ntab
  FROM customers_demo
  WHERE cust_address_ntab IS A SET
  ORDER BY customer_id;

CUSTOMER_ID CUST_ADDRESS_NTAB(STREET_ADDRESS, POSTAL_CODE, CITY, STATE_PROVINCE, COUNTRY_ID)
-----------------------------------------------------------------------------------------
        101 CUST_ADDRESS_TAB_TYP(CUST_ADDRESS_TYP('514 W Superior St', '46901', 'Kokomo', 'IN', 'US'))
        102 CUST_ADDRESS_TAB_TYP(CUST_ADDRESS_TYP('2515 Bloyd Ave', '46218', 'Indianapolis', 'IN', 'US'))
        103 CUST_ADDRESS_TAB_TYP(CUST_ADDRESS_TYP('8768 N State Rd 37', '47404', 'Bloomington', 'IN', 'US'))
        104 CUST_ADDRESS_TAB_TYP(CUST_ADDRESS_TYP('6445 Bay Harbor Ln', '46254', 'Indianapolis', 'IN', 'US'))
        105 CUST_ADDRESS_TAB_TYP(CUST_ADDRESS_TYP('4019 W 3Rd St', '47404', 'Bloomington', 'IN', 'US'))
```

The preceding example requires the table `customers_demo` and a nested table column containing data. Refer to "Multiset Operators " to create this table and nested table column.

IS EMPTY Condition

Use the `IS [NOT] EMPTY` conditions to test whether a specified nested table is empty. A nested table that consists of a single value, a `NULL`, is not considered an empty nested table.

***is_empty_condition*::=**



The condition returns a Boolean value: `TRUE` for an `IS EMPTY` condition if the collection is empty, and `TRUE` for an `IS NOT EMPTY` condition if the collection is not empty. If you specify `NULL` for the nested table or varray, then the result is `NULL`.

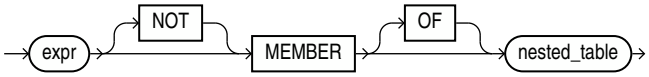
Example

The following example selects from the sample table `pm.print_media` those rows in which the `ad_textdocs_ntab` nested table column is not empty:

```
SELECT product_id, TO_CHAR(ad_finaltext) AS text
   FROM print_media
   WHERE ad_textdocs_ntab IS NOT EMPTY
   ORDER BY product_id, text;
```

MEMBER Condition

***member_condition*::=**



A *member_condition* is a membership condition that tests whether an element is a member of a nested table. The return value is TRUE if *expr* is equal to a member of the specified nested table or varray. The return value is NULL if *expr* is null or if the nested table is empty.

- *expr* must be of the same type as the element type of the nested table.
- The OF keyword is optional and does not change the behavior of the condition.
- The NOT keyword reverses the Boolean output: Oracle returns FALSE if *expr* is a member of the specified nested table.
- The element types of the nested table must be comparable. Refer to Comparison Conditions for information on the comparability of nonscalar types.

Example

The following example selects from the table customers_demo those rows in which the cust_address_ntab nested table column contains the values specified in the WHERE clause:

```
SELECT customer_id, cust_address_ntab
  FROM customers_demo
  WHERE cust_address_typ('8768 N State Rd 37', 47404,
                         'Bloomington', 'IN', 'US')
  MEMBER OF cust_address_ntab
  ORDER BY customer_id;

CUSTOMER_ID CUST_ADDRESS_NTAB(STREET_ADDRESS, POSTAL_CODE, CITY, STATE_PROVINCE, COUNTRY_ID)
----------- --------------------------------------------------------------------------------
        103 CUST_ADDRESS_TAB_TYP(CUST_ADDRESS_TYP('8768 N State Rd 37', '47404', 'Bloomington', 'IN', 'US'))
```

The preceding example requires the table customers_demo and a nested table column containing data. Refer to Multiset Operators to create this table and nested table column.

SUBMULTISET Condition

The SUBMULTISET condition tests whether a specified nested table is a submultiset of another specified nested table.

The operator returns a Boolean value. TRUE is returned when *nested_table1* is a submultiset of *nested_table2*. *nested_table1* is a submultiset of *nested_table2* when one of the following conditions occur:

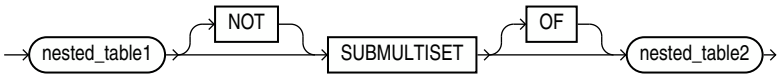
- *nested_table1* is not null and contains no rows. TRUE is returned even if *nested_table2* is null since an empty multiset is a submultiset of any non-null replacement for *nested_table2*.
- *nested_table1* and *nested_table2* are not null, *nested_table1* does not contain a null element, and there is a one-to-one mapping of each element in *nested_table1* to an equal element in *nested_table2*.

NULL is returned when one of the following conditions occurs:

- *nested_table1* is null.
- *nested_table2* is null, and *nested_table1* is not null and not empty.
- *nested_table1* is a submultiset of *nested_table2* after modifying each null element of *nested_table1* and *nested_table2* to some non-null value, enabling a one-to-one mapping of each element in *nested_table1* to an equal element in *nested_table2*.

If none of the above conditions occur, then FALSE is returned.

***submultiset_condition*::=**



- The OF keyword is optional and does not change the behavior of the operator.
- The NOT keyword reverses the Boolean output: Oracle returns FALSE if *nested_table1* is a subset of *nested_table2*.
- The element types of the nested table must be comparable. Refer to Comparison Conditions for information on the comparability of nonscalar types.

Example

The following example selects from the customers_demo table those rows in which the cust_address_ntab nested table is a submultiset of the cust_address2_ntab nested table:

```
SELECT customer_id, cust_address_ntab
  FROM customers_demo
  WHERE cust_address_ntab SUBMULTISET OF cust_address2_ntab
  ORDER BY customer_id;
```

The preceding example requires the table customers_demo and two nested table columns containing data. Refer to Multiset Operators to create this table and nested table columns.

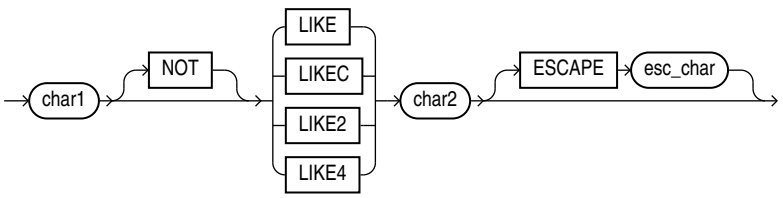
Pattern-matching Conditions

The pattern-matching conditions compare character data.

LIKE Condition

The LIKE conditions specify a test involving pattern matching. Whereas the equality operator (=) exactly matches one character value to another, the LIKE conditions match a portion of one character value to another by searching the first value for the pattern specified by the second. LIKE calculates strings using characters as defined by the input character set. LIKEC uses Unicode complete characters. LIKE2 uses UCS2 code points. LIKE4 uses UCS4 code points.

***like_condition*::=**



In this syntax:

- *char1* is a character expression, such as a character column, called the **search value**.
- *char2* is a character expression, usually a literal, called the **pattern**.
- *esc_char* is a character expression, usually a literal, called the **escape character**.

The LIKE condition is the best choice in almost all situations. Use the following guidelines to determine whether any of the variations would be helpful in your environment:

- Use LIKE2 to process strings using UCS-2 semantics. LIKE2 treats a Unicode supplementary character as two characters.
- Use LIKE4 to process strings using UCS-4 semantics. LIKE4 treats a Unicode supplementary character as one character.
- Use LIKEC to process strings using Unicode complete character semantics. LIKEC treats a composite character as one character.

For more on character length see the following:

- *Oracle Database Globalization Support Guide*
- *Oracle Database SecureFiles and Large Objects Developer's Guide*

If *esc_char* is not specified, then there is no default escape character. If any of *char1*, *char2*, or *esc_char* is null, then the result is unknown. Otherwise, the escape character, if specified, must be a character string of length 1.

All of the character expressions (*char1*, *char2*, and *esc_char*) can be of any of the data types CHAR, VARCHAR2, NCHAR, or NVARCHAR2. If they differ, then Oracle converts all of them to the data type of *char1*.

The pattern can contain special pattern-matching characters:

- An underscore (_) in the pattern matches exactly one character (as opposed to one byte in a multibyte character set) in the value.
- A percent sign (%) in the pattern can match zero or more characters (as opposed to bytes in a multibyte character set) in the value. The pattern '%' cannot match a null.

You can include the actual characters % or _ in the pattern by using the ESCAPE clause, which identifies the escape character. If the escape character precedes the character % or _ in the pattern, then Oracle interprets this character literally in the pattern rather than as a special pattern-matching character. You can also search for the escape character itself by repeating it. For example, if @ is the escape character, then you can use @@ to search for @.

> **Note:**
>
> Only ASCII-equivalent underscore (_) and percent (%) characters are recognized as pattern-matching characters. Their full-width variants, present in East Asian character sets and in Unicode, are treated as normal characters.

Table 6-8 describes the LIKE conditions.

Table 6-8 LIKE Condition

Type of Condition	Operation	Example
`x [NOT] LIKE y [ESCAPE 'z']`	`TRUE` if *x* does [not] match the pattern *y*. Within *y*, the character `%` matches any string of zero or more characters except null. The character _ matches any single character. Any character can follow `ESCAPE` except percent (%) and underbar (_). A wildcard character is treated as a literal if preceded by the escape character.	`SELECT last_name FROM employees WHERE last_name LIKE '%A\_B%' ESCAPE '\' ORDER BY last_name;`

To process the `LIKE` conditions, Oracle divides the pattern into subpatterns consisting of one or two characters each. The two-character subpatterns begin with the escape character and the other character is %, or _, or the escape character.

Let P_1, P_2, ..., P_n be these subpatterns. The like condition is true if there is a way to partition the search value into substrings S_1, S_2, ..., S_n so that for all *i* between 1 and *n*:

- If P_i is _, then S_i is a single character.
- If P_i is %, then S_i is any string.
- If P_i is two characters beginning with an escape character, then S_i is the second character of P_i.
- Otherwise, $P_i = S_i$.

With the `LIKE` conditions, you can compare a value to a pattern rather than to a constant. The pattern must appear after the `LIKE` keyword. For example, you can issue the following query to find the salaries of all employees with names beginning with `R`:

```
SELECT salary
    FROM employees
    WHERE last_name LIKE 'R%'
    ORDER BY salary;
```

The following query uses the = operator, rather than the `LIKE` condition, to find the salaries of all employees with the name 'R%':

```
SELECT salary
    FROM employees
    WHERE last_name = 'R%'
    ORDER BY salary;
```

The following query finds the salaries of all employees with the name 'SM%'. Oracle interprets 'SM%' as a text literal, rather than as a pattern, because it precedes the `LIKE` keyword:

```
SELECT salary
    FROM employees
    WHERE 'SM%' LIKE last_name
    ORDER BY salary;
```

Collation and Case Sensitivity

The `LIKE` condition is collation-sensitive. Oracle Database compares the subpattern Pi to the substring Si in the processing algorithm above using the collation determined from the derived

collations of *char1* and *char2*. If this collation is case-insensitive, the pattern-matching is case-insensitive as well.

> **See Also:**
>
> *Oracle Database Globalization Support Guide* for more information on case- and accent-insensitive collations and on collation determination rules for the `LIKE` condition

Pattern Matching on Indexed Columns

When you use `LIKE` to search an indexed column for a pattern, Oracle can use the index to improve performance of a query if the leading character in the pattern is not `%` or _. In this case, Oracle can scan the index by this leading character. If the first character in the pattern is `%` or _, then the index cannot improve performance because Oracle cannot scan the index.

LIKE Condition: General Examples

This condition is true for all `last_name` values beginning with `Ma`:

```
last_name LIKE 'Ma%'
```

All of these `last_name` values make the condition true:

```
Mallin, Markle, Marlow, Marvins, Mavris, Matos
```

Case is significant, so `last_name` values beginning with `MA`, `ma`, and `mA` make the condition false.

Consider this condition:

```
last_name LIKE 'SMITH_'
```

This condition is true for these `last_name` values:

```
SMITHE, SMITHY, SMITHS
```

This condition is false for `SMITH` because the special underscore character (_) must match exactly one character of the `last_name` value.

ESCAPE Clause Example

The following example searches for employees with the pattern `A_B` in their name:

```
SELECT last_name
    FROM employees
    WHERE last_name LIKE '%A\_B%' ESCAPE '\'
    ORDER BY last_name;
```

The `ESCAPE` clause identifies the backslash (\) as the escape character. In the pattern, the escape character precedes the underscore (_). This causes Oracle to interpret the underscore literally, rather than as a special pattern matching character.

Patterns Without % Example

If a pattern does not contain the `%` character, then the condition can be true only if both operands have the same length. Consider the definition of this table and the values inserted into it:

```
CREATE TABLE ducks (f CHAR(6), v VARCHAR2(6));
INSERT INTO ducks VALUES ('DUCK', 'DUCK');
SELECT '*'||f||'*' "char",
   '*'||v||'*' "varchar"
   FROM ducks;

char     varchar
-------- --------
*DUCK  * *DUCK*
```

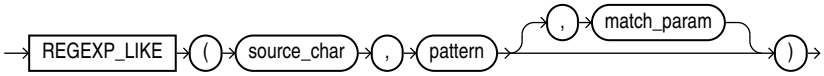
Because Oracle blank-pads `CHAR` values, the value of `f` is blank-padded to 6 bytes. `v` is not blank-padded and has length 4.

REGEXP_LIKE Condition

`REGEXP_LIKE` is similar to the `LIKE` condition, except `REGEXP_LIKE` performs regular expression matching instead of the simple pattern matching performed by `LIKE`. This condition evaluates strings using characters as defined by the input character set.

This condition complies with the POSIX regular expression standard and the Unicode Regular Expression Guidelines. For more information, refer to Oracle Regular Expression Support.

***regexp_like_condition*::=**



- *source_char* is a character expression that serves as the search value. It is commonly a character column and can be of any of the data types `CHAR`, `VARCHAR2`, `NCHAR`, `NVARCHAR2`, `CLOB`, or `NCLOB`.
- *pattern* is the **regular expression**. It is usually a text literal and can be of any of the data types `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2`. It can contain up to 512 bytes. If the data type of *pattern* is different from the data type of *source_char*, Oracle converts *pattern* to the data type of *source_char*. For a listing of the operators you can specify in *pattern*, refer to Oracle Regular Expression Support.
- *match_param* is a character expression of the data type `VARCHAR2` or `CHAR` that lets you change the default matching behavior of the condition.

  The value of *match_param* can include one or more of the following characters:

  – `'i'` specifies case-insensitive matching, even if the determined collation of the condition is case-sensitive.

  – `'c'` specifies case-sensitive and accent-sensitive matching, even if the determined collation of the condition is case-insensitive or accent-insensitive.

  – `'n'` allows the period (.), which is the match-any-character wildcard character, to match the newline character. If you omit this parameter, then the period does not match the newline character.

- `'m'` treats the source string as multiple lines. Oracle interprets `^` and `$` as the start and end, respectively, of any line anywhere in the source string, rather than only at the start or end of the entire source string. If you omit this parameter, then Oracle treats the source string as a single line.
- `'x'` ignores whitespace characters. By default, whitespace characters match themselves.

If the value of *match_param* contains multiple contradictory characters, then Oracle uses the last character. For example, if you specify `'ic'`, then Oracle uses case-sensitive and accent-sensitive matching. If the value contains a character other than those shown above, then Oracle returns an error.

If you omit *match_param*, then:

- The default case and accent sensitivity are determined by the determined collation of the `REGEXP_LIKE` condition.
- A period (.) does not match the newline character.
- The source string is treated as a single line.

Similar to the `LIKE` condition, the `REGEXP_LIKE` condition is collation-sensitive.

> **See Also:**
>
> - LIKE Condition
> - REGEXP_INSTR , REGEXP_REPLACE , and REGEXP_SUBSTR for functions that provide regular expression support
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules for the `REGEXP_LIKE` condition

Examples

The following query returns the first and last names for those employees with a first name of Steven or Stephen (where `first_name` begins with `Ste` and ends with `en` and in between is either `v` or `ph`):

```
SELECT first_name, last_name
FROM employees
WHERE REGEXP_LIKE (first_name, '^Ste(v|ph)en$')
ORDER BY first_name, last_name;

FIRST_NAME           LAST_NAME
-------------------- -------------------------
Steven               King
Steven               Markle
Stephen              Stiles
```

The following query returns the last name for those employees with a double vowel in their last name (where `last_name` contains two adjacent occurrences of either `a`, `e`, `i`, `o`, or `u`, regardless of case):

```
SELECT last_name
FROM employees
WHERE REGEXP_LIKE (last_name, '([aeiou])\1', 'i')
ORDER BY last_name;
```

```
LAST_NAME
-------------------------
De Haan
Greenberg
Khoo
Gee
Greene
Lee
Bloom
Feeney
```

Null Conditions

A `NULL` condition tests for nulls. This is the only condition that you should use to test for nulls.

***null_condition*::=**

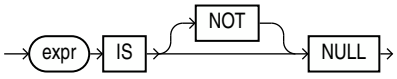


Table 6-9 lists the null conditions.

Table 6-9 Null Condition

Type of Condition	Operation	Example

`IS [NOT] NULL`	Tests for nulls.
	See Also: Nulls
		`SELECT last_name`
		`  FROM employees`
		`  WHERE commission_pct`
		`  IS NULL`
		`  ORDER BY last_name;`

XML Conditions

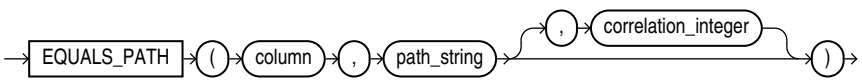
XML conditions determine whether a specified XML resource can be found in a specified path.

EQUALS_PATH Condition

The `EQUALS_PATH` condition determines whether a resource in the Oracle XML database can be found in the database at a specified path.

Use this condition in queries to `RESOURCE_VIEW` and `PATH_VIEW`. These public views provide a mechanism for SQL access to data stored in the XML database repository. `RESOURCE_VIEW` contains one row for each resource in the repository, and `PATH_VIEW` contains one row for each unique path in the repository.

***equals_path_condition*::=**



This condition applies only to the path as specified. It is similar to but more restrictive than UNDER_PATH.

For *path_string*, specify the (absolute) path name to resolve. This can contain components that are hard or weak resource links.

The optional *correlation_integer* argument correlates the EQUALS_PATH condition with its ancillary functions DEPTH and PATH.

> **See Also:**
>
> UNDER_PATH Condition , DEPTH , and PATH

Example

The view RESOURCE_VIEW computes the paths (in the any_path column) that lead to all XML resources (in the res column) in the database repository. The following example queries the RESOURCE_VIEW view to find the paths to the resources in the sample schema oe. The EQUALS_PATH condition causes the query to return only the specified path:

```
SELECT ANY_PATH FROM RESOURCE_VIEW
   WHERE EQUALS_PATH(res, '/sys/schemas/OE/www.example.com')=1;

ANY_PATH
------------------------------------------------
/sys/schemas/OE/www.example.com
```

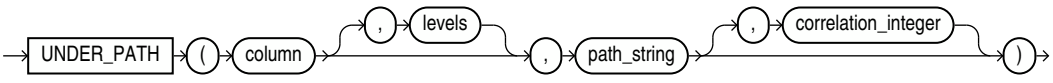
Compare this example with that for UNDER_PATH Condition .

UNDER_PATH Condition

The UNDER_PATH condition determines whether resources specified in a column can be found under a particular path specified by *path_string* in the Oracle XML database repository. The path information is computed by the RESOURCE_VIEW view, which you query to use this condition.

Use this condition in queries to RESOURCE_VIEW and PATH_VIEW. These public views provide a mechanism for SQL access to data stored in the XML database repository. RESOURCE_VIEW contains one row for each resource in the repository, and PATH_VIEW contains one row for each unique path in the repository.

***under_path_condition*::=**



The optional *levels* argument indicates the number of levels down from *path_string* Oracle should search. For *levels*, specify any nonnegative integer.

The optional *correlation_integer* argument correlates the UNDER_PATH condition with its ancillary functions PATH and DEPTH.

> **See Also:**
>
> The related condition EQUALS_PATH Condition and the ancillary functions DEPTH and PATH

Example

The view `RESOURCE_VIEW` computes the paths (in the `any_path` column) that lead to all XML resources (in the `res` column) in the database repository. The following example queries the `RESOURCE_VIEW` view to find the paths to the resources in the sample schema `oe`. The query returns the path of the XML schema that was created in XMLType Table Examples:

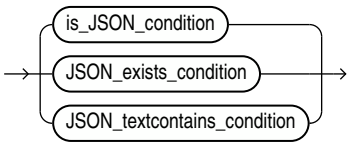
```
SELECT ANY_PATH FROM RESOURCE_VIEW
   WHERE UNDER_PATH(res, '/sys/schemas/OE/www.example.com')=1;

ANY_PATH
--------------------------------------------
/sys/schemas/OE/www.example.com/xwarehouses.xsd
```

SQL For JSON Conditions

SQL for JSON conditions allow you to test JavaScript Object Notation (JSON) data as follows:

- IS JSON Condition lets you test whether an expression is syntactically correct JSON data.
- JSON_EQUAL Condition tests whether two JSON values are the same.
- JSON_EXISTS Condition lets you test whether a specified JSON value exists in JSON data.
- JSON_TEXTCONTAINS Condition lets you test whether a specified character string exists in JSON property values.

***JSON_condition*::=**



IS JSON Condition

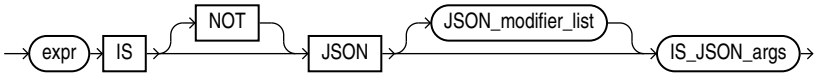
Use this SQL JSON condition to test whether an expression is syntactically correct, well-formed, JSON data.

- If the data tested is syntactically correct and keyword `VALIDATE` is not present, then `IS JSON` returns true, and `IS NOT JSON` returns false.
- If keyword `VALIDATE` is present, then the data is tested to ensure that it is both well-formed and valid with respect to the specified JSON schema. Keyword `VALIDATE` (optionally followed by keyword `USING`) must be followed by a SQL string literal that is the JSON schema to validate against.
- If an error occurs during parsing or validating, and the data is considered to not be well-formed or not valid, then `IS JSON` returns false and `IS NOT JSON` returns true. Parsing and

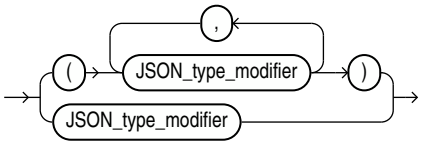
validation errors are handled by the condition itself returning true or false. Other errors that are neither from parsing or validation, these errors are raised.

- You can use `IS JSON` and `IS NOT JSON` in a `CASE` expression or the `WHERE` clause of a `SELECT` statement. You can use `IS JSON` in a check constraint.

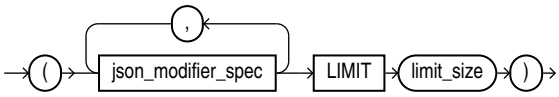
***is_JSON_condition*::=**



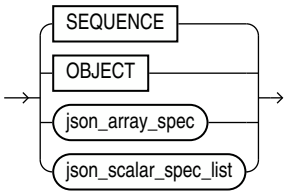
***JSON_modifier_list*::=**



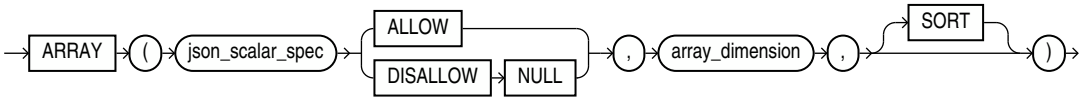
***JSON_type_modifier*::=**



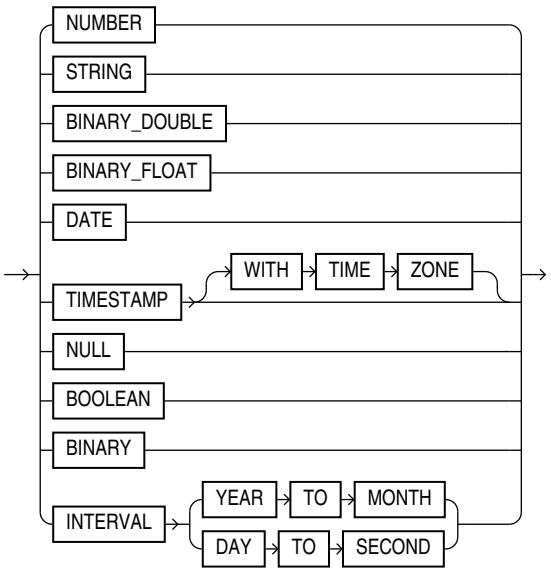
***json_modifier_spec*::=**



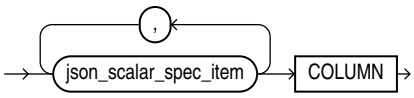
***json_array_spec*::=**



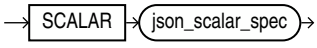
***JSON_scalar_spec*::=**



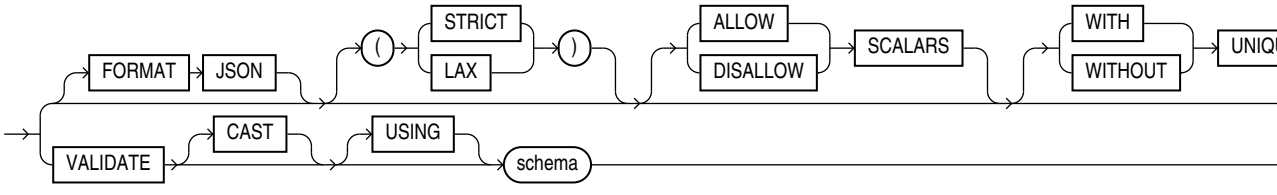
***json_scalar_spec_list*::=**



***json_scalar_spec_item*::=**



***is_json_args*::=**



- Use *expr* to specify the JSON data to be evaluated. Specify an expression that evaluates to a text literal. If *expr* is a column, then the column must be of data type VARCHAR2, CLOB, or BLOB. If *expr* evaluates to null or a text literal of length zero, then this condition returns UNKNOWN.
- LIMIT in *json_modifier_spec* applies to the entire JSON modifier specification.

- Specify `ALLOW NULL` in *`json_array_spec`* to allow a JSON single type scalar value of NULL.
- Specify `DISALLOW NULL` in *`json_array_spec`* to disallow a JSON single type scalar value of NULL. This is the default.
- Specify `SORT` in *`json_array_spec`* to sort the JSON array elements in ascending order.

  For more information see SQL/JSON Conditions IS JSON and IS NOT JSON of the *JSON Developer's Guide*.

is_json_args

- You must specify `FORMAT JSON` if *`expr`* is a column of data type `BLOB`.
- If you specify `STRICT`, then this condition considers only strict JSON syntax to be well-formed JSON data. If you specify `LAX`, then this condition also considers lax JSON syntax to be well-formed JSON data. The default is `LAX`.

  For a full discussion of `STRICT` and `LAX` syntax see *About Strict and Lax JSON Syntax*, and *TYPE Clause for SQL Functions and Conditions*
- If you specify `WITH UNIQUE KEYS`, then this condition considers JSON data to be well-formed only if key names are unique within each object. If you specify `WITHOUT UNIQUE KEYS`, then this condition considers JSON data to be well-formed even if duplicate key names occur within an object. A `WITHOUT UNIQUE KEYS` test performs faster than a `WITH UNIQUE KEYS` test. The default is `WITHOUT UNIQUE KEYS`.
- Specify the optional keyword `VALIDATE` to test that the data is also valid with respect to a given JSON schema.
- To enforce that a JSON type value is a certain type, you can use JSON type modifiers. See SQL/JSON Conditions IS JSON and IS NOT JSON *of the JSON Developer's Guide.*

JSON Schema Validation

A JSON schema typically specifies the allowed structure and data typing of other JSON documents. You can therefore use a JSON schema to validate JSON data. You can validate JSON data against a JSON schema in the following ways:

- Use condition `IS JSON` (or `IS NOT JSON`) with keyword `VALIDATE` and the name of a JSON schema, to test whether targeted data is valid (or invalid) against that schema. The schema can be provided as a literal string or a usage domain. (Keyword `VALIDATE` can optionally be followed by keyword `USING`.)

  You can use `VALIDATE` with condition is json anywhere you can use that condition. This includes use in a `WHERE` clause, or as a check constraint to ensure that only valid data is inserted in a column.

  When used as a check constraint for a JSON-type column, you can alternatively omit is json, and just use keyword `VALIDATE` directly. These two table creations are equivalent, for a JSON-type column:

```
CREATE TABLE tab (jcol JSON VALIDATE '{"type" : "object"}');

CREATE TABLE tab (jcol JSON CONSTRAINT jchk
  CHECK (jcol IS JSON VALIDATE '{"type" : "object"}'));
```

- Use a usage domain as a check constraint for JSON type data. For example:

```
CREATE DOMAIN jd AS JSON CONSTRAINT jchkd CHECK (jd IS JSON VALIDATE '{"type" :
"object"});

CREATE TABLE jtab(jcol JSON DOMAIN jd);
```

When creating a domain from a schema, you can alternatively omit the constraint and is json, and just use keyword `VALIDATE` directly. This domain creation is equivalent to the previous one:

```
CREATE DOMAIN jd AS JSON VALIDATE '{"type" : "object"};
```

- For databases that support a binary JSON format, data can be encoded on the client. In such cases, the database does not have to convert textual JSON to its binary representation and hence validation using an extended data type can be performed.

  If textual JSON is sent to the database, it is followed by an encoding process to a binary representation (server-side encoding). In such circumstances, the schema validator can operate in `CAST` mode. That is, the binary encoder can use the value specified for the extended data type keyword in the JSON schema and encode the scalar field to its binary representation. Only scaler types are eligible for casting.

```
CREATE TABLE jtab (
  id    NUMBER(9) PRIMARY KEY,
  jcol  JSON CHECK(jcol IS JSON VALIDATE CAST USING '{
                  "type": "object",
                  "properties": {
                       "firstName": {
                        "extendedType": "string",
                        "maxLength": 50
                      },
                      "birthDate" : {
                        "extendedType": "date"
                      }
                    },
                    "required": ["firstName", "birthDate"]
                  }'
  )
);
```

  The following textual JSON is a valid document per the above schema:

```
{
  "firstName": "Scott",
  "birthDate": "1990-04-02"
}
```

- Use PL/SQL functions explained fully in *JSON Schema* of the *JSON Developer's Guide*.

Static dictionary views `DBA_JSON_SCHEMA_COLUMNS`, `ALL_JSON_SCHEMA_COLUMNS`, and `USER_JSON_SCHEMA_COLUMNS` describe a JSON schema that you is used as a check constraint.

Each row of these views contains the name of the table, the JSON column, and the constraint defined by the JSON schema, as well as the JSON schema itself and an indication of whether the cast mode is specified for the JSON schema. Views `DBA_JSON_SCHEMA_COLUMNS` and `ALL_JSON_SCHEMA_COLUMNS` also contain the name of the table owner.

Examples

IS JSON VALIDATE

The following exampe creates a schema `jsontab1` with a JSON constraint `jtlisj` that has a JSON validate check:

```
CREATE TABLE jsontab1(
    id NUMBER(4),
    j  JSON CONSTRAINT jt1isj CHECK (j IS JSON VALIDATE USING
```

```
     '{
       "type":"object",
       "minProperties": 2
      }')
    );
```

The following example shows the error when you try to insert values other than a JSON object:

```
INSERT INTO jsontab1(j) VALUES ('["a", "b"]');
INSERT INTO jsontab1(j) VALUES ('["a", "b"]')
*
ERROR at line 1:
ORA-02290: check constraint (SYS.JT1ISJ) violated
```

The following two examples shows a row added with valid input :

```
INSERT INTO jsontab1(j) VALUES ('{"a": "a", "b": "b"}');

   1 row created.


INSERT INTO jsontab1(jschd) VALUES (json('"a json string"'));

1 row created.
```

The following example adds another constraint `jschdsv` to table `jsontab1`:

```
ALTER TABLE jsontab1
ADD jschd JSON CONSTRAINT jschdsv
                  CHECK (jschd IS JSON VALIDATE USING '{"type":"string"}');

Table altered.


SQL> INSERT INTO jsontab1(jschd) VALUES (json('3.1415'));
INSERT INTO jsontab1(jschd) VALUES (json('3.1415'))
*
ERROR at line 1:
ORA-02290: check constraint (SYS.JSCHDSV) violated
```

IS JSON VALIDATE in WHERE Clause

```
SELECT COUNT(1) FROM jsontab1 WHERE j IS JSON
VALIDATE
         '{"type" : "object",
              "properties" : {
                  "id" : {
                      "type" : "number"
                   }
               }
           }';
```

Testing for STRICT or LAX JSON Syntax: Example

The following statement creates table `t` with column `col1`:

```
CREATE TABLE t (col1 VARCHAR2(100));
```

The following statements insert values into column `col1` of table `t`:

```
INSERT INTO t VALUES ( '[ "LIT192", "CS141", "HIS160" ]' );
INSERT INTO t VALUES ( '{ "Name": "John" }' );
```

```
INSERT INTO t VALUES ( '{ "Grade Values" : { A : 4.0, B : 3.0, C : 2.0 } }');
INSERT INTO t VALUES ( '{ "isEnrolled" : true }' );
INSERT INTO t VALUES ( '{ "isMatriculated" : False }' );
INSERT INTO t VALUES (NULL);
INSERT INTO t VALUES ('This is not well-formed JSON data');
```

The following statement queries table `t` and returns `col1` values that are well-formed JSON data. Because neither the `STRICT` nor `LAX` keyword is specified, this example uses the default `LAX` setting. Therefore, this query returns values that use strict or lax JSON syntax.

```
SELECT col1
  FROM t
  WHERE col1 IS JSON;

COL1
--------------------------------------------------
[ "LIT192", "CS141", "HIS160" ]
{ "Name": "John" }
{ "Grade Values" : { A : 4.0, B : 3.0, C : 2.0 } }
{ "isEnrolled" : true }
{ "isMatriculated" : False }
```

The following statement queries table `t` and returns `col1` values that are well-formed JSON data. This example specifies the `STRICT` setting. Therefore, this query returns only values that use strict JSON syntax.

```
SELECT col1
  FROM t
  WHERE col1 IS JSON STRICT;

COL1
--------------------------------------------------
[ "LIT192", "CS141", "HIS160" ]
{ "Name": "John" }
{ "isEnrolled" : true }
```

The following statement queries table `t` and returns `col1` values that use lax JSON syntax, but omits `col1` values that use strict JSON syntax. Therefore, this query returns only values that contain the exceptions allowed in lax JSON syntax.

```
SELECT col1
  FROM t
  WHERE col1 IS NOT JSON STRICT AND col1 IS JSON LAX;

COL1
--------------------------------------------------
{ "Grade Values" : { A : 4.0, B : 3.0, C : 2.0 } }
{ "isMatriculated" : False }
```

Testing for Unique Keys: Example

The following statement creates table `t` with column `col1`:

```
CREATE TABLE t (col1 VARCHAR2(100));
```

The following statements insert values into column `col1` of table `t`:

```
INSERT INTO t VALUES ('{a:100, b:200, c:300}');
INSERT INTO t VALUES ('{a:100, a:200, b:300}');
INSERT INTO t VALUES ('{a:100, b : {a:100, c:300}}');
```

The following statement queries table t and returns `col1` values that are well-formed JSON data with unique key names within each object:

```
SELECT col1 FROM t
  WHERE col1 IS JSON WITH UNIQUE KEYS;

COL1
--------------------------
{a:100, b:200, c:300}
{a:100, b : {a:100, c:300}}
```

The second row is returned because, while the key name `a` appears twice, it is in two different objects.

The following statement queries table `t` and returns `col1` values that are well-formed JSON data, regardless of whether there are unique key names within each object:

```
SELECT col1 FROM t
  WHERE col1 IS JSON WITHOUT UNIQUE KEYS;

COL1
--------------------------
{a:100, b:200, c:300}
{a:100, a:200, b:300}
{a:100, b : {a:100, c:300}}
```

Using IS JSON as a Check Constraint: Example

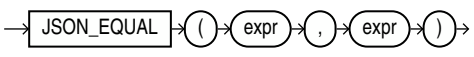
The following statement creates table `j_purchaseorder`, which will store JSON data in column `po_document`. The statement uses the `IS JSON` condition as a check constraint to ensure that only well-formed JSON is stored in column `po_document`.

```
CREATE TABLE j_purchaseorder
  (id RAW (16) NOT NULL,
   date_loaded TIMESTAMP(6) WITH TIME ZONE,
   po_document CLOB CONSTRAINT ensure_json CHECK (po_document IS JSON));
```

> **See Also:**
>
> *Conditions IS JSON and IS NOT JSON* of the *JSON Developer's Guide*.

JSON_EQUAL Condition

Syntax



Purpose

The Oracle SQL condition `JSON_EQUAL` compares two `JSON` values and returns true if they are equal. It returns false if the two values are not equal. The input values must be valid `JSON` data.

The comparison ignores insignificant whitespace and insignificant object member order. For example, `JSON` objects are equal, if they have the same members, regardless of their order.

If either of the two compared inputs has one or more duplicate fields, then the value returned by `JSON_EQUAL` is unspecified.

`JSON_EQUAL` supports `ERROR ON ERROR`, `FALSE ON ERROR`, and `TRUE ON ERROR`. The default is `FALSE ON ERROR`. A typical example of an error is when the input expression is not valid `JSON`.

Examples

The following statements return TRUE:

```
JSON_EQUAL('{}', '{ }')

JSON_EQUAL('{a:1, b:2}', '{b:2 , a:1 }')
```

The following statement return FALSE:

```
JSON_EQUAL('{a:"1"}', '{a:1 }') -> FALSE
```

The following statement results in a `ORA-40441 JSON` syntax error

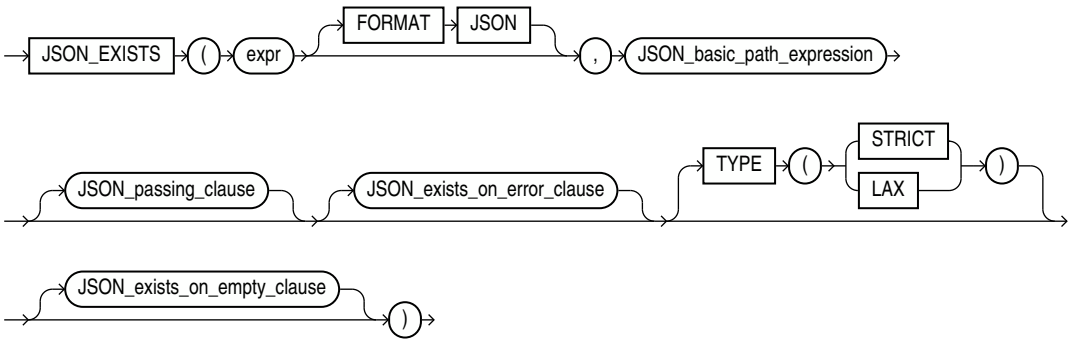
```
JSON_EQUAL('[1]', '[}' ERROR ON ERROR)
```

> **See Also:**
>
> - *Oracle Database JSON Developer's Guide* for more information.

JSON_EXISTS Condition

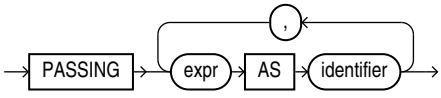
Use the SQL/JSON condition `JSON_EXISTS` to test whether a specified JSON value exists in JSON data. This condition returns `TRUE` if the JSON value exists and `FALSE` if the JSON value does not exist.

***JSON_exists_condition*::=**

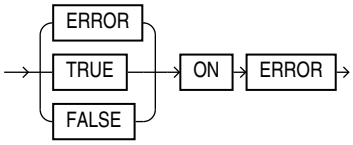


(*JSON_basic_path_expression*: See *Oracle Database JSON Developer's Guide*)

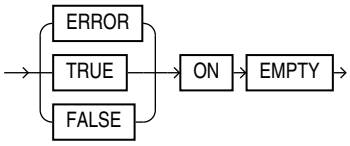
***JSON_passing_clause*::=**



***JSON_exists_on_error_clause*::=**



***JSON_exists_on_empty_clause*::=**



expr

Use this clause to specify the JSON data to be evaluated. For *expr*, specify an expression that evaluates to a text literal. If *expr* is a column, then the column must be of data type VARCHAR2, CLOB, or BLOB. If *expr* evaluates to null or a text literal of length zero, then the condition returns UNKNOWN.

If *expr* is not a text literal of well-formed JSON data using strict or lax syntax, then the condition returns FALSE by default. You can use the *JSON_exists_on_error_clause* to override this default behavior. Refer to the *JSON_exists_on_error_clause*.

FORMAT JSON

You must specify FORMAT JSON if *expr* is a column of data type BLOB.

JSON_basic_path_expression

Use this clause to specify a SQL/JSON path expression. The condition uses the path expression to evaluate *expr* and determine if a JSON value that matches, or satisfies, the path expression exists. The path expression must be a text literal, but it can contain variables whose values are passed to the path expression by the *JSON_passing_clause*. See *Oracle Database JSON Developer's Guide* for the full semantics of *JSON_basic_path_expression*.

JSON_passing_clause

Use this clause to pass values to the path expression. For *expr*, specify a value of data type VARCHAR2, NUMBER, BINARY_DOUBLE, DATE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE. The result of evaluating *expr* is bound to the corresponding identifier in the *JSON_basic_path_expression*.

JSON_exists_on_error_clause

Use this clause to specify the value returned by this condition when *expr* is not well-formed JSON data.

You can specify the following clauses:

- ERROR ON ERROR - Returns the appropriate Oracle error when *expr* is not well-formed JSON data.

- TRUE ON ERROR - Returns TRUE when *expr* is not well-formed JSON data.
- FALSE ON ERROR - Returns FALSE when *expr* is not well-formed JSON data. This is the default.

TYPE Clause

For a full discussion of STRICT and LAX syntax see *About Strict and Lax JSON Syntax*, and *TYPE Clause for SQL Functions and Conditions*

JSON_exists_on_empty_clause

Use this clause to specify the value returned by this function if no match is found when the JSON data is evaluated using the SQL/JSON path expression.

You can specify the following clauses:

- ERROR ON EMPTY - Returns the appropriate Oracle error when *expr* is not well-formed JSON data.
- TRUE ON EMPTY - Returns TRUE when *expr* is not well-formed JSON data.
- FALSE ON EMPTY - Returns FALSE when expr is not well-formed JSON data. This is the default.

Examples

The following statement creates table t with column name:

```
CREATE TABLE t (name VARCHAR2(100));
```

The following statements insert values into column name of table t:

```
INSERT INTO t VALUES ('[{first:"John"}, {middle:"Mark"}, {last:"Smith"}]');
INSERT INTO t VALUES ('[{first:"Mary"}, {last:"Jones"}]');
INSERT INTO t VALUES ('[{first:"Jeff"}, {last:"Williams"}]');
INSERT INTO t VALUES ('[{first:"Jean"}, {middle:"Anne"}, {last:"Brown"}]');
INSERT INTO t VALUES (NULL);
INSERT INTO t VALUES ('This is not well-formed JSON data');
```

The following statement queries column name in table t and returns JSON data that consists of an array whose first element is an object with property name first. The ON ERROR clause is not specified. Therefore, the JSON_EXISTS condition returns FALSE for values that are not well-formed JSON data.

```
SELECT name FROM t
  WHERE JSON_EXISTS(name, '$[0].first');

NAME
--------------------------------------------------
[{first:"John"}, {middle:"Mark"}, {last:"Smith"}]
[{first:"Mary"}, {last:"Jones"}]
[{first:"Jeff"}, {last:"Williams"}]
[{first:"Jean"}, {middle:"Anne"}, {last:"Brown"}]
```

The following statement queries column name in table t and returns JSON data that consists of an array whose second element is an object with property name middle. The ON ERROR clause is not specified. Therefore, the JSON_EXISTS condition returns FALSE for values that are not well-formed JSON data.

```
SELECT name FROM t
  WHERE JSON_EXISTS(name, '$[1].middle');

NAME
--------------------------------------------------------------------------------
[{first:"John"}, {middle:"Mark"}, {last:"Smith"}]
[{first:"Jean"}, {middle:"Anne"}, {last:"Brown"}]
```

The following statement is similar to the previous statement, except that the `TRUE ON ERROR` clause is specified. Therefore, the `JSON_EXISTS` condition returns `TRUE` for values that are not well-formed JSON data.

```
SELECT name FROM t
  WHERE JSON_EXISTS(name, '$[1].middle' TRUE ON ERROR);

NAME
--------------------------------------------------------------------------------
[{first:"John"}, {middle:"Mark"}, {last:"Smith"}]
[{first:"Jean"}, {middle:"Anne"}, {last:"Brown"}]
This is not well-formed JSON data
```

The following statement queries column `name` in table `t` and returns JSON data that consists of an array that contains an element that is an object with property name `last`. The wildcard symbol (`*`) is specified for the array index. Therefore, the query returns arrays that contain such an object, regardless of its index number in the array.

```
SELECT name FROM t
  WHERE JSON_EXISTS(name, '$[*].last');

NAME
--------------------------------------------------
[{first:"John"}, {middle:"Mark"}, {last:"Smith"}]
[{first:"Mary"}, {last:"Jones"}]
[{first:"Jeff"}, {last:"Williams"}]
[{first:"Jean"}, {middle:"Anne"}, {last:"Brown"}]
```

The following statement performs a filter expression using the passing clause. The SQL/JSON variable *`$var1`* in the comparison predicate *`(@.middle == $var1)`* gets its value from the bind variable *`var1`* of the PASSING clause.

Using bind variables for value comparisons avoids query re-compilation.

```
SELECT name FROM t

  WHERE JSON_EXISTS(name, '$[1]?(@.middle == $var1)' PASSING 'Anne' as "var1");

NAME

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

[{first:"Jean"}, {middle:"Anne"}, {last:"Brown"}]
```

> **See Also:**
>
> Condition JSON_Exists

JSON_TEXTCONTAINS Condition

Use the SQL/JSON condition `JSON_TEXTCONTAINS` to test whether a specified character string exists in JSON property values. You can use this condition to filter JSON data on a specific word or number.

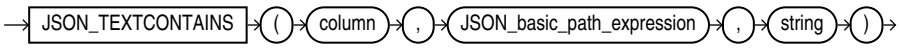
This condition takes the following arguments:

- A table or view column that contains JSON data. A JSON search index, which is an Oracle Text index designed specifically for use with JSON data, must be defined on the column. Each row of JSON data in the column is referred to as a **JSON document**.
- A SQL/JSON path expression. The path expression is applied to each JSON document in an attempt to match a specific JSON object within the document. The path expression can contain only JSON object steps; it cannot contain JSON array steps.
- A character string. The condition searches for the character string in all of the string and numeric property values in the matched JSON object, including array values. The string must exist as a separate word in the property value. For example, if you search for 'beth', then a match will be found for string property value "beth smith", but not for "elizabeth smith". If you search for '10', then a match will be found for numeric property value 10 or string property value "10 main street", but a match will not be found for numeric property value 110 or string property value "102 main street".

This condition returns `TRUE` if a match is found, and `FALSE` if a match is not found.

> **See Also:**
>
> JSON Full text search queries

***JSON_textcontains_condition*::=**



(*JSON_basic_path_expression*: See *Oracle Database JSON Developer's Guide*)

column

Specify the name of the table or view column containing the JSON data to be tested. The column must be of data type `VARCHAR2`, `CLOB`, or `BLOB`. A JSON search index, which is an Oracle Text index designed specifically for use with JSON data, must be defined on the column. If a column value is a null or a text literal of length zero, then the condition returns `UNKNOWN`.

If a column value is not a text literal of well-formed JSON data using strict or lax syntax, then the condition returns `FALSE`.

JSON_basic_path_expression

Use this clause to specify a SQL/JSON path expression. The condition uses the path expression to evaluate *column* and determine if a JSON value that matches, or satisfies, the path expression exists. The path expression must be a text literal. See *Oracle Database JSON Developer's Guide* for the full semantics of *JSON_basic_path_expression*.

string

The condition searches for the character string specified by *string*. The string must be enclosed in single quotation marks.

Examples

The following statement creates table `families` with column `family_doc`:

```
CREATE TABLE families (family_doc VARCHAR2(200));
```

The following statement creates a JSON search index on column `family_doc`:

```
CREATE INDEX ix
  ON families(family_doc)
  INDEXTYPE IS CTXSYS.CONTEXT
  PARAMETERS ('SECTION GROUP CTXSYS.JSON_SECTION_GROUP SYNC (ON COMMIT)');
```

The following statements insert JSON documents that describe families into column `family_doc`:

```
INSERT INTO families
VALUES ('{family : {id:10, ages:[40,38,12], address : {street : "10 Main Street"}}}');

INSERT INTO families
VALUES ('{family : {id:11, ages:[42,40,10,5], address : {street : "200 East Street", apt : 20}}}');

INSERT INTO families
VALUES ('{family : {id:12, ages:[25,23], address : {street : "300 Oak Street", apt : 10}}}');
```

The following statement commits the transaction:

```
COMMIT;
```

The following query returns the JSON documents that contain `10` in any property value in the document:

```
SELECT family_doc FROM families
  WHERE JSON_TEXTCONTAINS(family_doc, '$', '10');

FAMILY_DOC
--------------------------------------------------------------------------------
{family : {id:10, ages:[40,38,12], address : {street : "10 Main Street"}}}
{family : {id:11, ages:[42,40,10,5], address : {street : "200 East Street", apt : 20}}}
{family : {id:12, ages:[25,23], address : {street : "300 Oak Street", apt : 10}}}
```

The following query returns the JSON documents that contain 10 in the `id` property value:

```
SELECT family_doc FROM families
  where json_textcontains(family_doc, '$.family.id', '10');

FAMILY_DOC
--------------------------------------------------------------------------------
{family : {id:10, ages:[40,38,12], address : {street : "10 Main Street"}}}
```

The following query returns the JSON documents that have a 10 in the array of values for the `ages` property:

```
SELECT family_doc FROM families
  WHERE JSON_TEXTCONTAINS(family_doc, '$.family.ages', '10');

FAMILY_DOC
--------------------------------------------------------------------------------
{family : {id:11, ages:[42,40,10,5], address : {street : "200 East Street", apt : 20}}}
```

The following query returns the JSON documents that have a 10 in the `address` property value:

```
SELECT family_doc FROM families
  WHERE JSON_TEXTCONTAINS(family_doc, '$.family.address', '10');

FAMILY_DOC
--------------------------------------------------------------------------------
{family : {id:10, ages:[40,38,12], address : {street : "10 Main Street"}}}
{family : {id:12, ages:[25,23], address : {street : "300 Oak Street", apt : 10}}}
```

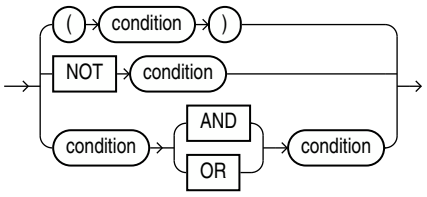
The following query returns the JSON documents that have a 10 in the `apt` property value:

```
SELECT family_doc FROM families
  WHERE JSON_TEXTCONTAINS(family_doc, '$.family.address.apt', '10');

FAMILY_DOC
--------------------------------------------------------------------------------
{family : {id:12, ages:[25,23], address : {street : "300 Oak Street", apt : 10}}}
```

Compound Conditions

A compound condition specifies a combination of other conditions.

***compound_condition*::=**

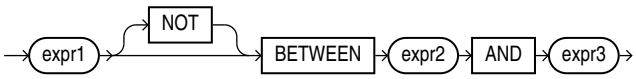


> **See Also:**
>
> Logical Conditions for more information about `NOT`, `AND`, and `OR` conditions

BETWEEN Condition

A `BETWEEN` condition determines whether the value of one expression is in an interval defined by two other expressions.

***between_condition*::=**



All three expressions must be numeric, character, or datetime expressions. In SQL, it is possible that *expr1* will be evaluated more than once. If the BETWEEN expression appears in PL/SQL, *expr1* is guaranteed to be evaluated only once. If the expressions are not all the same data type, then Oracle Database implicitly converts the expressions to a common data type. If it cannot do so, then it returns an error.

> **See Also:**
>
> Implicit Data Conversion for more information on SQL data type conversion

The value of

```
expr1 NOT BETWEEN expr2 AND expr3
```

is the value of the expression

```
NOT (expr1 BETWEEN expr2 AND expr3)
```

And the value of

```
expr1 BETWEEN expr2 AND expr3
```

is the value of the boolean expression:

```
expr2 <= expr1 AND expr1 <= expr3
```

If *expr3* < *expr2*, then the interval is empty. If *expr1* is NULL, then the result is NULL. If *expr1* is not NULL, then the value is FALSE in the ordinary case and TRUE when the keyword NOT is used.

The boolean operator AND may produce unexpected results. Specifically, in the expression x AND y, the condition x IS NULL is not sufficient to determine the value of the expression. The second operand still must be evaluated. The result is FALSE if the second operand has the value FALSE and NULL otherwise. See Logical Conditions for more information on AND.

Table 6-10 BETWEEN Condition

Type of Condition	Operation	Example
[NOT] BETWEEN x AND y	[NOT] (*expr2* less than or equal to *expr1* AND *expr1* less than or equal to *expr3*)	SELECT * FROM employees WHERE salary BETWEEN 2000 AND 3000 ORDER BY employee_id;

EXISTS Condition

An `EXISTS` condition tests for existence of rows in a subquery.

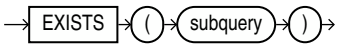


Table 6-11 shows the `EXISTS` condition.

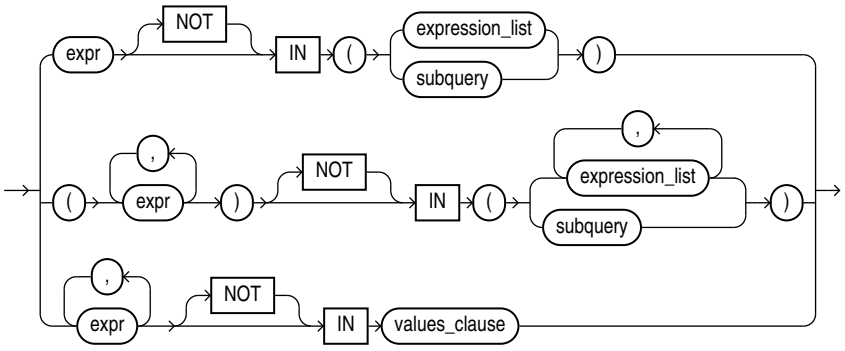
Table 6-11 EXISTS Condition

Type of Condition	Operation	Example
`EXISTS`	`TRUE` if a subquery returns at least one row.	`SELECT department_id FROM departments d WHERE EXISTS (SELECT * FROM employees e WHERE d.department_id = e.department_id) ORDER BY department_id;`

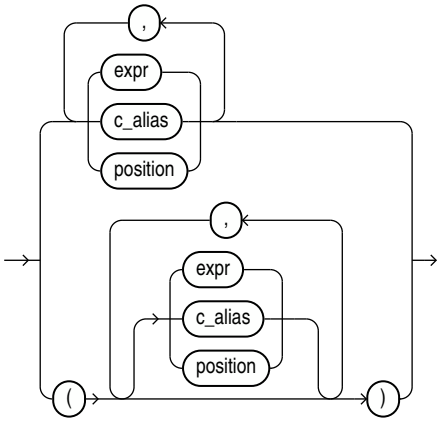
IN Condition

An *`in_condition`* is a membership condition. It tests a value for membership in a list of values or subquery

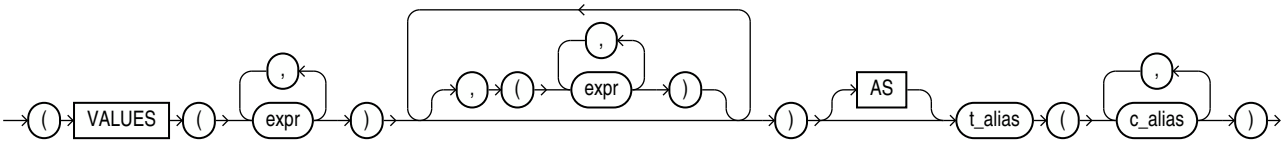
***in_condition*::=**



***expression_list*::=**



***values_clause*::=**



If you use the upper form of the *in_condition* condition (with a single expression to the left of the operator), then you must use the upper form of *expression_list*. If you use the lower form of this condition (with multiple expressions to the left of the operator), then you must use the lower form of *expression_list*, and the expressions in each *expression_list* must match in number and data type the expressions to the left of the operator. You can specify up to 65535 expressions in *expression_list*.

Oracle Database does not always evaluate the expressions in an *expression_list* in the order in which they appear in the `IN` list. However, expressions in the select list of a subquery are evaluated in their specified order.

> **See Also:**
>
> Expression Lists

Table 6-12 lists the form of `IN` condition.

Table 6-12 IN Condition

Type of Condition	Operation	Example
IN	Equal-to-any-member-of test. Equivalent to =ANY.	SELECT * FROM employees WHERE job_id IN ('PU_CLERK','SH_CLERK') ORDER BY employee_id; SELECT * FROM employees WHERE salary IN (SELECT salary FROM employees WHERE department_id =30) ORDER BY employee_id;
NOT IN	Equivalent to !=ALL. Evaluates to FALSE if any member of the set is NULL.	SELECT * FROM employees WHERE salary NOT IN (SELECT salary FROM employees WHERE department_id = 30) ORDER BY employee_id; SELECT * FROM employees WHERE job_id NOT IN ('PU_CLERK', 'SH_CLERK') ORDER BY employee_id;

values_clause

For semantics of the *values_clause* please see the *values_clause* of the SELECT statement *values_clause* .

If any item in the list following a NOT IN operation evaluates to null, then all rows evaluate to FALSE or UNKNOWN, and no rows are returned. For example, the following statement returns the string 'True' for each row:

```
SELECT 'True' FROM employees
   WHERE department_id NOT IN (10, 20);
```

However, the following statement returns no rows:

```
SELECT 'True' FROM employees
    WHERE department_id NOT IN (10, 20, NULL);
```

The preceding example returns no rows because the WHERE clause condition evaluates to:

```
department_id != 10 AND department_id != 20 AND department_id != null
```

Because the third condition compares department_id with a null, it results in an UNKNOWN, so the entire expression results in FALSE (for rows with department_id equal to 10 or 20). This behavior can easily be overlooked, especially when the NOT IN operator references a subquery.

Moreover, if a NOT IN condition references a subquery that returns no rows at all, then all rows will be returned, as shown in the following example:

```
SELECT 'True' FROM employees
   WHERE department_id NOT IN (SELECT 0 FROM DUAL WHERE 1=2);
```

For character arguments, the `IN` condition is collation-sensitive. The collation determination rules determine the collation to use.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules for the `IN` condition

Restriction on LEVEL in WHERE Clauses

In a [`NOT`] `IN` condition in a `WHERE` clause, if the right-hand side of the condition is a subquery, you cannot use `LEVEL` on the left-hand side of the condition. However, you can specify `LEVEL` in a subquery of the `FROM` clause to achieve the same result. For example, the following statement is not valid:

```
SELECT employee_id, last_name FROM employees
   WHERE (employee_id, LEVEL)
      IN (SELECT employee_id, 2 FROM employees)
   START WITH employee_id = 2
   CONNECT BY PRIOR employee_id = manager_id;
```

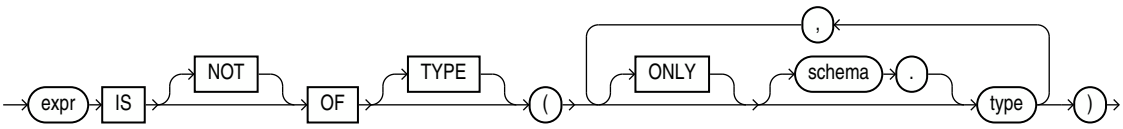
But the following statement is valid because it encapsulates the query containing the `LEVEL` information in the `FROM` clause:

```
SELECT v.employee_id, v.last_name, v.lev FROM
      (SELECT employee_id, last_name, LEVEL lev
      FROM employees v
      START WITH employee_id = 100
      CONNECT BY PRIOR employee_id = manager_id) v
   WHERE (v.employee_id, v.lev) IN
      (SELECT employee_id, 2 FROM employees);
```

IS OF *type* Condition

Use the `IS OF` *`type`* condition to test object instances based on their specific type information.

***is_of_type_condition*::=**



You must have `EXECUTE` privilege on all types referenced by *`type`*, and all *`type`*s must belong to the same type family.

This condition evaluates to null if *`expr`* is null. If *`expr`* is not null, then the condition evaluates to true (or false if you specify the `NOT` keyword) under either of these circumstances:

- The most specific type of *`expr`* is the subtype of one of the types specified in the *`type`* list and you have not specified `ONLY` for the type, or
- The most specific type of *`expr`* is explicitly specified in the *`type`* list.

The *`expr`* frequently takes the form of the `VALUE` function with a correlation variable.

The following example uses the sample table `oe.persons`, which is built on a type hierarchy in Substitutable Table and Column Examples. The example uses the `IS OF type` condition to restrict the query to specific subtypes:

```
SELECT * FROM persons p
   WHERE VALUE(p) IS OF TYPE (employee_t);

NAME                        SSN
--------------------------------
Joe                       32456
Tim                        5678

SELECT * FROM persons p
   WHERE VALUE(p) IS OF (ONLY part_time_emp_t);

NAME                        SSN
--------------------------------
Tim                       5678
```

BOOLEAN Test Condition

Syntax