4
Operators

An **operator** manipulates data items and returns a result. Syntactically, an operator appears before or after an operand or between two operands.

This chapter contains these sections:

- About SQL Operators
- Arithmetic Operators
- COLLATE Operator
- Data Quality Operators
- Concatenation Operator
- GRAPH_TABLE Operator
- Hierarchical Query Operators
- Multiset Operators
- Set Operators
- SHARD_CHUNK_ID Operator
- User-Defined Operators

This chapter discusses nonlogical (non-Boolean) operators. These operators cannot by themselves serve as the condition of a `WHERE` or `HAVING` clause in queries or subqueries. For information on logical operators, which serve as conditions, refer to Conditions.

About SQL Operators

Operators manipulate individual data items called **operands** or **arguments**. Operators are represented by special characters or by keywords. For example, the multiplication operator is represented by an asterisk (*).

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

Unary and Binary Operators

The two general classes of operators are:

- **unary**: A unary operator operates on only one operand. A unary operator typically appears with its operand in this format:

  ```
  operator operand
  ```

- **binary**: A binary operator operates on two operands. A binary operator appears with its operands in this format:

  ```
  operand1 operator operand2
  ```

Other operators with special formats accept more than two operands. If an operator is given a null operand, then the result is always null. The only operator that does not follow this rule is concatenation (||).

Operator Precedence

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

Table 4-1 lists the levels of precedence among SQL operators from high to low. Operators listed on the same line have the same precedence.

Table 4-1 SQL Operator Precedence

Operator	Operation
`+, -` (as unary operators), `PRIOR`, `CONNECT_BY_ROOT`, `COLLATE`	Identity, negation, location in hierarchy
`*, /`	Multiplication, division
`+, -` (as binary operators), `||`	Addition, subtraction, concatenation
<-> is the Euclidian distance operator, <=> is the cosine distance operator, <#> is the negative dot product operator	Shorthand Operators for Distances
SQL conditions are evaluated after SQL operators	See "Condition Precedence"

Precedence Example

In the following expression, multiplication has a higher precedence than addition, so Oracle first multiplies 2 by 3 and then adds the result to 1.

```
1+2*3
```

You can use parentheses in an expression to override operator precedence. Oracle evaluates expressions inside parentheses before evaluating those outside.

SQL also supports set operators (`UNION`, `UNION ALL`, `INTERSECT`, and `MINUS`), which combine sets of rows returned by queries, rather than individual data items. All set operators have equal precedence.

> **See Also:**
>
> Hierarchical Query Operators and Hierarchical Queries for information on the `PRIOR` operator, which is used only in hierarchical queries

Arithmetic Operators

You can use an arithmetic operator with one or two arguments to negate, add, subtract, multiply, and divide numeric values. Some of these operators are also used in datetime and interval arithmetic. The arguments to the operator must resolve to numeric data types or to any data type that can be implicitly converted to a numeric data type.

Unary arithmetic operators return the same data type as the numeric data type of the argument. For binary arithmetic operators, Oracle determines the argument with the highest numeric precedence, implicitly converts the remaining arguments to that data type, and returns that data type. Table 4-2 lists arithmetic operators.

> **See Also:**
>
> Table 2-9 for more information on implicit conversion, Numeric Precedence for information on numeric precedence, and Datetime/Interval Arithmetic

Table 4-2 Arithmetic Operators

Operator	Purpose	Example

+ -	When these denote a positive or negative expression, they are unary operators.	`SELECT * FROM order_items WHERE quantity = -1 ORDER BY order_id, line_item_id, product_id;`
		
		`SELECT * FROM employees WHERE -salary < 0 ORDER BY employee_id;`

+ -	When they add or subtract, they are binary operators.	`SELECT hire_date FROM employees WHERE SYSDATE - hire_date > 365 ORDER BY hire_date;`
* /	Multiply, divide. These are binary operators.	`UPDATE employees SET salary = salary * 1.1;`

Do not use two consecutive minus signs (--) in arithmetic expressions to indicate double negation or the subtraction of a negative value. The characters -- are used to begin comments within SQL statements. You should separate consecutive minus signs with a space or parentheses. Refer to Comments for more information on comments within SQL statements.

COLLATE Operator

The `COLLATE` operator determines the collation for an expression. This operator enables you to override the collation that the database would have derived for the expression using standard collation derivation rules.

`COLLATE` is a postfix unary operator. It has the same precedence as other unary operators, but it is evaluated after all prefix unary operators have been evaluated.

You can apply this operator to expressions of type `VARCHAR2`, `CHAR`, `LONG`, `NVARCHAR`, or `NCHAR`.

The `COLLATE` operator takes one argument, *`collation_name`*, for which you can specify a named collation or pseudo-collation. If the collation name contains a space, then you must enclose the name in double quotation marks.

Table 4-3 describes the COLLATE operator.

Table 4-3 COLLATE Operator

Operator	Purpose	Example

COLLATE *collation_name*	Determines the collation for an expression	`SELECT last_name`
		`  FROM employees`
		`  ORDER BY last_name COLLATE GENERIC_M;`

> **See Also:**
>
> - Compound Expressions for information on using the COLLATE operator in a compound expression
> - *Oracle Database Globalization Support Guide* for more information on the COLLATE operator

Concatenation Operator

The concatenation operator manipulates character strings and CLOB data. Table 4-4 describes the concatenation operator.

Table 4-4 Concatenation Operator

Operator	Purpose	Example

||	Concatenates character strings and CLOB data.	`SELECT 'Name is ' || last_name`
		`  FROM employees`
		`  ORDER BY last_name;`

The result of concatenating two character strings is another character string. If both character strings are of data type CHAR, then the result has data type CHAR and is limited to 2000 characters. If either string is of data type VARCHAR2, then the result has data type VARCHAR2 and is limited to 32767 characters if the initialization parameter MAX_STRING_SIZE = EXTENDED and 4000 characters if MAX_STRING_SIZE = STANDARD. Refer to Extended Data Types for more information. If either argument is a CLOB, the result is a temporary CLOB. Trailing blanks in character strings are preserved by concatenation, regardless of the data types of the string or CLOB.

On most platforms, the concatenation operator is two solid vertical bars, as shown in Table 4-4. However, some IBM platforms use broken vertical bars for this operator. When moving SQL script files between systems having different character sets, such as between ASCII and EBCDIC, vertical bars might not be translated into the vertical bar required by the target Oracle Database environment. Oracle provides the CONCAT character function as an alternative to the vertical bar operator for cases when it is difficult or impossible to control translation performed by operating system or network utilities. Use this function in applications that will be moved between environments with differing character sets.

Although Oracle treats zero-length character strings as nulls, concatenating a zero-length character string with another operand always results in the other operand, so null can result

only from the concatenation of two null strings. However, this may not continue to be true in future versions of Oracle Database. To concatenate an expression that might be null, use the NVL function to explicitly convert the expression to a zero-length string.

> **See Also:**
>
> - Character Data Types for more information on the differences between the CHAR and VARCHAR2 data types
> - The functions CONCAT and NVL
> - *Oracle Database SecureFiles and Large Objects Developer's Guide* for more information about CLOBs
> - *Oracle Database Globalization Support Guide* for the collation derivation rules for the concatenation operator

Concatenation Example

This example creates a table with both CHAR and VARCHAR2 columns, inserts values both with and without trailing blanks, and then selects these values and concatenates them. Note that for both CHAR and VARCHAR2 columns, the trailing blanks are preserved.

```
CREATE TABLE tab1 (col1 VARCHAR2(6), col2 CHAR(6),
                   col3 VARCHAR2(6), col4 CHAR(6));

INSERT INTO tab1 (col1,  col2,     col3,     col4)
          VALUES ('abc', 'def   ', 'ghi   ', 'jkl');

SELECT col1 || col2 || col3 || col4 "Concatenation"
  FROM tab1;

Concatenation
------------------------
abcdef   ghi   jkl
```

Hierarchical Query Operators

Two operators, PRIOR and CONNECT_BY_ROOT, are valid only in hierarchical queries.

PRIOR

In a hierarchical query, one expression in the CONNECT BY *condition* must be qualified by the PRIOR operator. If the CONNECT BY *condition* is compound, then only one condition requires the PRIOR operator, although you can have multiple PRIOR conditions. PRIOR evaluates the immediately following expression for the parent row of the current row in a hierarchical query.

PRIOR is most commonly used when comparing column values with the equality operator. (The PRIOR keyword can be on either side of the operator.) PRIOR causes Oracle to use the value of the parent row in the column. Operators other than the equal sign (=) are theoretically possible in CONNECT BY clauses. However, the conditions created by these other operators can result in an infinite loop through the possible combinations. In this case Oracle detects the loop at run time and returns an error. Refer to Hierarchical Queries for more information on this operator, including examples.

CONNECT_BY_ROOT

CONNECT_BY_ROOT is a unary operator that is valid only in hierarchical queries. When you qualify a column with this operator, Oracle returns the column value using data from the root row. This operator extends the functionality of the CONNECT BY [PRIOR] condition of hierarchical queries.

Restriction on CONNECT_BY_ROOT

You cannot specify this operator in the START WITH condition or the CONNECT BY condition.

> **See Also:**
>
> CONNECT_BY_ROOT Examples

Set Operators

Set operators combine the results of two component queries into a single result. Queries containing set operators are called compound queries. Table 4-5 lists the SQL set operators. They are fully described with examples in The Set Operators.

Table 4-5 Set Operators

Operator	Returns
UNION	All distinct rows selected by either query
UNION ALL	All rows selected by either query, including duplicates
INTERSECT	All distinct rows selected by both queries
INTERSECT ALL	All rows selected by both queries including duplicates
MINUS	All distinct rows selected by the first query but not the second
MINUS ALL	All rows selected by the first query but not the second including duplicates
EXCEPT	All distinct rows selected by the first query but not the second
EXCEPT ALL	All rows selected by the first query but not the second including duplicates

Multiset Operators

Multiset operators combine the results of two nested tables into a single nested table.

The examples related to multiset operators require that two nested tables be created and loaded with data as follows:

First, make a copy of the oe.customers table called customers_demo:

```
CREATE TABLE customers_demo AS
  SELECT * FROM customers;
```

Next, create a table type called cust_address_tab_typ. This type will be used when creating the nested table columns.

```
CREATE TYPE cust_address_tab_typ AS
  TABLE OF cust_address_typ;
/
```

Now, create two nested table columns in the `customers_demo` table:

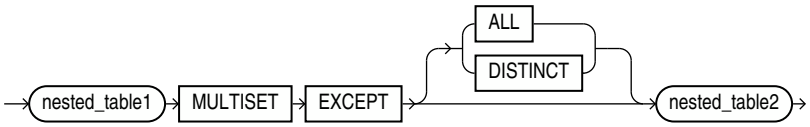
```
ALTER TABLE customers_demo
  ADD (cust_address_ntab cust_address_tab_typ,
       cust_address2_ntab cust_address_tab_typ)
    NESTED TABLE cust_address_ntab STORE AS cust_address_ntab_store
    NESTED TABLE cust_address2_ntab STORE AS cust_address2_ntab_store;
```

Finally, load data into the two new nested table columns using data from the `cust_address` column of the `oe.customers` table:

```
UPDATE customers_demo cd
  SET cust_address_ntab =
    CAST(MULTISET(SELECT cust_address
                  FROM customers c
                  WHERE c.customer_id =
                        cd.customer_id) as cust_address_tab_typ);

UPDATE customers_demo cd
  SET cust_address2_ntab =
    CAST(MULTISET(SELECT cust_address
                  FROM customers c
                  WHERE c.customer_id =
                        cd.customer_id) as cust_address_tab_typ);
```

MULTISET EXCEPT

`MULTISET EXCEPT` takes as arguments two nested tables and returns a nested table whose elements are in the first nested table but not in the second nested table. The two input nested tables must be of the same type, and the returned nested table is of the same type as well.



- The `ALL` keyword instructs Oracle to return all elements in *`nested_table1`* that are not in *`nested_table2`*. For example, if a particular element occurs *`m`* times in *`nested_table1`* and *`n`* times in *`nested_table2`*, then the result will have *`(m-n)`* occurrences of the element if *`m>n`* and 0 occurrences if *`m<=n`*. `ALL` is the default.
- The `DISTINCT` keyword instructs Oracle to eliminate any element in *`nested_table1`* which is also in *`nested_table2`*, regardless of the number of occurrences.
- The element types of the nested tables must be comparable. Refer to Comparison Conditions for information on the comparability of nonscalar types.

Example

The following example compares two nested tables and returns a nested table of those elements found in the first nested table but not in the second nested table:

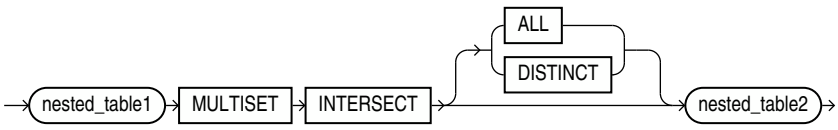
```
SELECT customer_id, cust_address_ntab
  MULTISET EXCEPT DISTINCT cust_address2_ntab multiset_except
  FROM customers_demo
```

```
  ORDER BY customer_id;

CUSTOMER_ID MULTISET_EXCEPT(STREET_ADDRESS, POSTAL_CODE, CITY, STATE_PROVINCE, COUNTRY_ID)
----------- -------------------------------------------------------------------------------
        101 CUST_ADDRESS_TAB_TYP()
        102 CUST_ADDRESS_TAB_TYP()
        103 CUST_ADDRESS_TAB_TYP()
        104 CUST_ADDRESS_TAB_TYP()
        105 CUST_ADDRESS_TAB_TYP()
. . .
```

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.

MULTISET INTERSECT

`MULTISET INTERSECT` takes as arguments two nested tables and returns a nested table whose values are common in the two input nested tables. The two input nested tables must be of the same type, and the returned nested table is of the same type as well.



- The `ALL` keyword instructs Oracle to return all common occurrences of elements that are in the two input nested tables, including duplicate common values and duplicate common `NULL` occurrences. For example, if a particular value occurs `m` times in *`nested_table1`* and `n` times in *`nested_table2`*, then the result would contain the element `min(m,n)` times. `ALL` is the default.
- The `DISTINCT` keyword instructs Oracle to eliminate duplicates from the returned nested table, including duplicates of `NULL`, if they exist.
- The element types of the nested tables must be comparable. Refer to Comparison Conditions for information on the comparability of nonscalar types.

Example

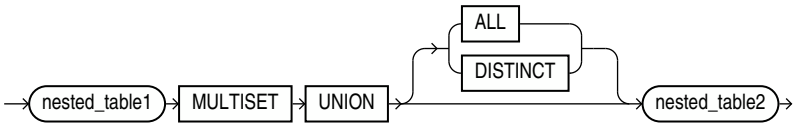
The following example compares two nested tables and returns a nested table of those elements found in both input nested tables:

```
SELECT customer_id, cust_address_ntab
  MULTISET INTERSECT DISTINCT cust_address2_ntab multiset_intersect
  FROM customers_demo
  ORDER BY customer_id;

CUSTOMER_ID MULTISET_INTERSECT(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 two nested table columns containing data. Refer to Multiset Operators to create this table and nested table columns.

MULTISET UNION

`MULTISET UNION` takes as arguments two nested tables and returns a nested table whose values are those of the two input nested tables. The two input nested tables must be of the same type, and the returned nested table is of the same type as well.



- The `ALL` keyword instructs Oracle to return all elements that are in the two input nested tables, including duplicate values and duplicate `NULL` occurrences. This is the default.
- The `DISTINCT` keyword instructs Oracle to eliminate duplicates from the returned nested table, including duplicates of `NULL`, if they exist.
- The element types of the nested tables must be comparable. Refer to Comparison Conditions for information on the comparability of nonscalar types.

Example

The following example compares two nested tables and returns a nested table of elements from both input nested tables:

```
SELECT customer_id, cust_address_ntab
  MULTISET UNION cust_address2_ntab multiset_union
  FROM customers_demo
  ORDER BY customer_id;

CUSTOMER_ID MULTISET_UNION(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'),
                                 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'),
                                 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'),
                                 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'),
                                 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'),
                                 CUST_ADDRESS_TYP('4019 W 3Rd St', '47404', 'Bloomington', 'IN', 'US'))
. . .
```

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.

SHARD_CHUNK_ID Operator

You can use the SQL operator `SHARD_CHUNK_ID` to get the chunk ID in a sharding environment. You must provide the table family ID and the sharding key as input.

This operator can be used in all three sharding types: system, user-defined, and composite. You can run the operator from the catalog and the shard.

Syntax

```
SELECT SHARD_CHUNK_ID( table_family, sharding_key1 [, sharding_key2 ...])
FROM table_name ...
```

Semantics

table_family

The first operand *table_family* refers to the identifier of the table family. It can be:

- The table family id that can be queried from the GSMADMIN_INTERNAL.TABLE_FAMILY table, or
- The name of the root table in the form of SCHEMA_NAME.TABLE_NAME .

If there is only one table family across the entire sharding environment, table_family can take NULL as input. This will default to the existing single table family.

sharding_key

The second operand *sharding_key* refers to a list of sharding keys. It can be a constant value or column name.

You must order the list of sharding keys as follows:

1. List of super-sharding keys in the order they are defined.
2. List of sharding keys in the order they are defined. For this refer to GSMADMIN_INTERNAL.SHARDKEY_COLUMNS .

In system and user-defined sharding environments, where super-sharding keys are not used, you only need to supply sharding keys.

Example

Given the composite sharded table customers defined as follows:

```
CREATE SHARDED TABLE customers (
    custno         NUMBER NOT NULL,
    name           VARCHAR2(50) NOT NULL,
    signup         DATE DEFAULT NULL,
    class          VARCHAR2(3) NOT NULL,
CONSTRAINT cust_pk PRIMARY KEY(custno,name))
PARTITIONSET BY LIST (class)
PARTITION BY CONSISTENT HASH (custno,name)
PARTITIONS AUTO
(PARTITIONSET gold VALUES ('gld') TABLESPACE SET tbs1,
 PARTITIONSET silver VALUES ('slv') TABLESPACE SET tbs2)
;
```

You can query it for the chunk ID with the following statement:

```
SELECT SHARD_CHUNK_ID(null, class, custno, name) FROM customers;
```

> **See Also:**
>
> - *Using Oracle Sharding*

User-Defined Operators

Like built-in operators, user-defined operators take a set of operands as input and return a result. However, you create them with the `CREATE OPERATOR` statement, and they are identified by user-defined names. They reside in the same namespace as tables, views, types, and standalone functions.

After you have defined a new operator, you can use it in SQL statements like any other built-in operator. For example, you can use user-defined operators in the select list of a `SELECT` statement, the condition of a `WHERE` clause, or in `ORDER BY` clauses and `GROUP BY` clauses. However, you must have `EXECUTE` privilege on the operator to do so, because it is a user-defined object.

> **See Also:**
>
> CREATE OPERATOR for an example of creating an operator and *Oracle Database Data Cartridge Developer's Guide* for more information on user-defined operators

Data Quality Operators

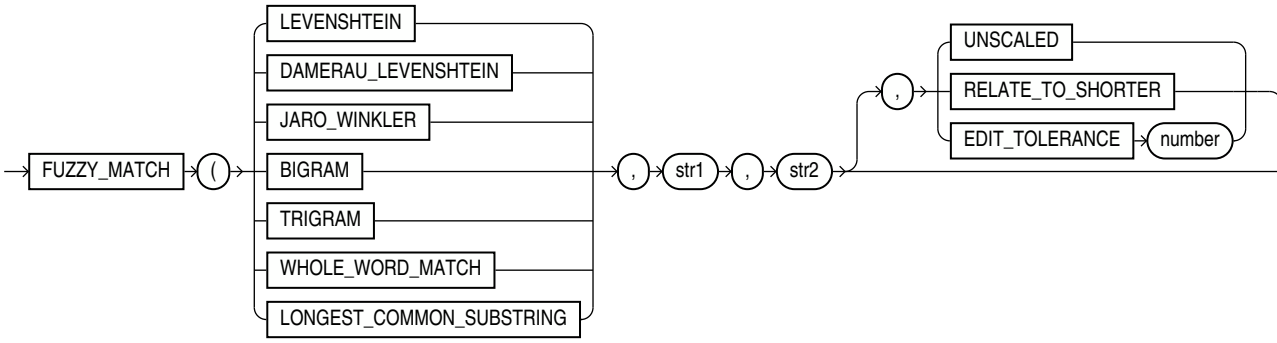
You can expand data quality capabilities within Oracle Database with string matching operators `PHONIC_ENCODE` and `FUZZY_MATCH`.

These operators can help you find near duplicate rows by matching strings that sounds alike or have small differences in spelling, for example:

- "Chris" and "Kris", in strings that sound alike
- "kitten" and "sitten", in strings that have small differences in spelling

FUZZY_MATCH

`FUZZY_MATCH` takes the algorithm to be used as the first argument, the strings to be processed as the second and third arguments, and some optional arguments that control the quality of the desired output.



The `UTL_MATCH` package evaluates byte by byte, while `FUZZY_MATCH` evaluates character by character. Therefore `UTL_MATCH` only works for comparison between single-byte strings while `FUZZY_MATCH` handles multi-byte charactersets.

When the `UNSCALED` option is specified, `FUZZY_MATCH` returns a measure in characters for the following algorithms: `LEVENSHTEIN` , `DAMERAU_LEVENSHTEIN` , `BIGRAM` , `TRIGRAM` , `LONGEST_COMMON_SUBSTRING` .

Arguments

The supported algorithms are:

- `LEVENSHTEIN` corresponds to `UTL_MATCH.EDIT_DISTANCE` or `UTL_MATCH.EDIT_SIMILARITY` and gives a measure of character edit distance or similarity.
- `DAMERAU_LEVENSHTEIN` distance differs from the classical `LEVENSHTEIN` distance by including transpositions among its allowable operations in addition to the three classical single-character edit operations (insertions, deletions and substitutions).
- `JARO_WINKLER` corresponds to `UTL_MATCH.JARO_WINKLER` (a percentage between 0-1) or `UTL_MATCH.JARO_WINKLER_SIMILARITY` (the same but scaled from 0-100).
- `BIGRAM` and `TRIGRAM` are instances of the N-gram matching technique, which counts the number of common contiguous sub-strings (grams) between the two strings.
- `WHOLE_WORD_MATCH` corresponds to Word Match Percentage or Count comparison in Oracle Enterprise Data Quality. It calculates the `LEVENSHTEIN` or edit distance of two phrases with words (instead of letters) as matching units.
- `LONGEST_COMMON_SUBSTRING` finds the longest common substring between the two strings.

Both *`str`* arguments can be any of the datatypes `CHAR`, `VARCHAR2`, `NCHAR`, `NVARCHAR2`.

UNSCALED

The keyword `UNSCALED` is optional. If you specify `UNSCALED` the return is one of :

- `LEVENSHTEIN` or edit distance
- `JARO_WINKLER` value in percentage
- N-grams, the number of common substrings
- LCS, the length of the longest common substring

RELATE_TO_SHORTER

The keyword `RELATE_TO_SHORTER` is optional. If you specify `RELATE_TO_SHORTER`, then the similarity measure is scaled by the length of the shorter input string. If you do not specify

RELATE_TO_SHORTER, then the default behavior is that the longer string length is used as the denominator.

EDIT_TOLERANCE

The keyword EDIT_TOLERANCE is optional. You can only specify EDIT_TOLERANCE with the WHOLE_WORD_MATCH algorithm. If you specify EDIT_TOLERANCE, the character error tolerance is the maximum percentage of the number of characters in a word that you allow to be different, while still considering each word as the same.

Returns

The operator returns NUMBER. By default, it is a similarity score normalized to be a percentage between 0-100.

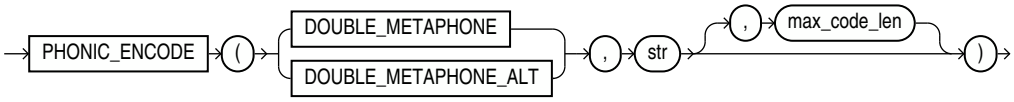
Examples

```
SQL> select fuzzy_match(LEVENSHTEIN, 'Mohamed Tarik', 'Mo Tariq') from dual;

FUZZY_MATCH(LEVENSHTEIN,'MOHAMEDTARIK','MOTARIQ')
-------------------------------------------------
                                               54

1 row selected.

SQL> select fuzzy_match(LEVENSHTEIN, 'Mohamed Tarik', 'Mo Tariq', unscaled) from dual;

FUZZY_MATCH(LEVENSHTEIN,'MOHAMEDTARIK','MOTARIQ',UNSCALED)
----------------------------------------------------------
                                                         6

1 row selected.


SQL> select fuzzy_match(DAMERAU_LEVENSHTEIN, 'Mohamed Tarik', 'Mo Tariq',
relate_to_shorter) from dual;

FUZZY_MATCH(DAMERAU_LEVENSHTEIN,'MOHAMEDTARIK','MOTARIQ',RELATE_TO_SHORTER)
---------------------------------------------------------------------------
                                                                         25

1 row selected.

SQL> select fuzzy_match(BIGRAM, 'Mohamed Tarik', 'Mo Tariq', unscaled) from dual;

FUZZY_MATCH(BIGRAM,'MOHAMEDTARIK','MOTARIQ',UNSCALED)
-----------------------------------------------------
                                                    5

1 row selected.

SQL> select fuzzy_match(LONGEST_COMMON_SUBSTRING, 'Mohamed Tarik', 'Mo Tariq', unscaled)
from dual;

FUZZY_MATCH(LONGEST_COMMON_SUBSTRING,'MOHAMEDTARIK','MOTARIQ',UNSCALED)
-----------------------------------------------------------------------
                                                                      5

1 row selected.

SQL> select fuzzy_match(WHOLE_WORD_MATCH,  'Mohamed Tarik', 'Mo Tariq') from dual;
```

```
FUZZY_MATCH(WHOLE_WORD_MATCH,'MOHAMEDTARIK','MOTARIQ')
------------------------------------------------------
                                                     0

1 row selected

SQL> select fuzzy_match(WHOLE_WORD_MATCH, 'Pawan Kumar Goel', 'Pavan Kumar G',
EDIT_TOLERANCE 60) from dual;

FUZZY_MATCH(WHOLE_WORD_MATCH,'PAWANKUMARGOEL','PAVANKUMARG',EDIT_TOLERANCE60)
-----------------------------------------------------------------------------
                                                                           67

1 row selected.
```

PHONIC_ENCODE

`PHONIC_ENCODE` takes the algorithm to be used as the first argument, the string to be processed as the second argument, and an optional `max_code_len` argument that controls the length of the desired output. `max_code_len` must be an integer between 1 and 12.



Arguments

`DOUBLE_METAPHONE` returns the primary code. `DOUBLE_METAPHONE_ALT` returns the alternative code if present. If the alternative code is not present, it returns the primary code.

`str` can be any of the datatypes `CHAR`, `VARCHAR2`, `NCHAR`, `NVARCHAR2`.

The optional argument *`max_code_len`* must be an integer. It allows codes longer than the default 4 characters to be returned for the original Metaphone algorithm.

Returns

The operator returns `VARCHAR2`.

Examples

```
SQL> select phonic_encode(DOUBLE_METAPHONE,     'smith') c1,
  2         phonic_encode(DOUBLE_METAPHONE_ALT, 'smith') c2 from dual;

C1            C2
------------- -------------
SM0           XMT

1 row selected.

SQL> select phonic_encode(DOUBLE_METAPHONE,     'Schmidt') c1,
  2         phonic_encode(DOUBLE_METAPHONE_ALT, 'Schmidt') c2 from dual;

C1            C2
------------- -------------
XMT           SMT

1 row selected.
```

```
SQL> select phonic_encode(DOUBLE_METAPHONE,     'phone') c1,
  2         phonic_encode(DOUBLE_METAPHONE_ALT, 'phone') c2 from dual;

C1            C2
------------- -------------
FN            FN

1 row selected.

SQL> select phonic_encode(DOUBLE_METAPHONE,     'George') c1,
  2         phonic_encode(DOUBLE_METAPHONE_ALT, 'George') c2 from dual;

C1            C2
------------- -------------
JRJ           KRK

1 row selected.

SQL> -- PNNT / PKNNT
SQL> select phonic_encode(DOUBLE_METAPHONE,     'poignant') c1,
  2         phonic_encode(DOUBLE_METAPHONE_ALT, 'poignant') c2,
  3         phonic_encode(DOUBLE_METAPHONE_ALT, 'poignant', 10) c3 from dual;

C1            C2            C3
------------- ------------- -------------
PNNT          PKNN          PKNNT
```

GRAPH_TABLE Operator

Purpose

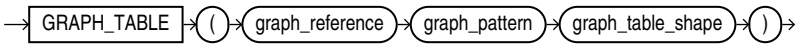
The `GRAPH_TABLE` operator can be used as a table expression in a `FROM` clause. It takes a graph as input against which it matches a specified graph pattern. It then outputs a set of solutions in tabular form.

This topic consists of the following sub-topics:

- Graph Reference
- Graph Pattern
- Graph Table Shape
- Value Expressions for GRAPH_TABLE

Syntax

***graph_table*::=**



(graph_reference ::=, graph_pattern ::=, graph_table_shape ::=)

Semantics

The`GRAPH_TABLE` operator starts with the keyword `GRAPH_TABLE` and consists of the following three parts that are placed between parentheses:

- *graph_reference*: a reference to a graph to perform the pattern matching on. Note that any graph first needs to be created through a `CREATE PROPERTY GRAPH` statement before it can be referenced in a `GRAPH_TABLE`.
- *graph_pattern*: a graph pattern consisting of vertex and edge patterns together with search conditions. The pattern is matched against the graph to obtain a set of solutions.
- *graph_table_shape*: a `COLUMNS` clause that projects the solutions into a tabular form.

A `FROM` clause in SQL may contain any number of `GRAPH_TABLE` operators as well as other types of table expressions. This allows for joining data from multiple graphs or for joining graph data with tabular, JSON, XML, or other types of data.

Examples

Setting Up Sample Data

This example creates a property graph, `students_graph`, using `persons`,`university`, `friendships`, and `students` as the underlying database tables for the graph.

The following statements first create the necessary tables and fill them with sample data:

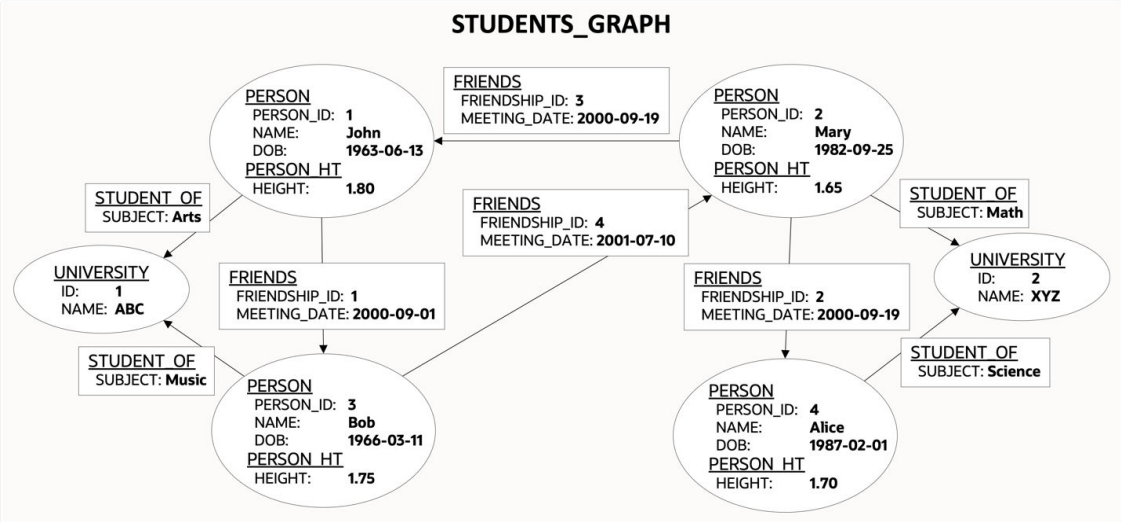
```
CREATE TABLE university (
    id NUMBER GENERATED ALWAYS AS IDENTITY (START WITH 1 INCREMENT BY 1),
    name VARCHAR2(10),
    CONSTRAINT u_pk PRIMARY KEY (id));
INSERT INTO university (name) VALUES ('ABC');
INSERT INTO university (name) VALUES ('XYZ');


CREATE TABLE persons (
     person_id NUMBER GENERATED ALWAYS AS IDENTITY (START WITH 1 INCREMENT
     BY 1),
     name VARCHAR2(10),
     birthdate DATE,
     height FLOAT DEFAULT ON NULL 0,
     person_data JSON,
     CONSTRAINT person_pk PRIMARY KEY (person_id));

INSERT INTO persons (name, height, birthdate, person_data)
       VALUES ('John', 1.80, to_date('13/06/1963', 'DD/MM/YYYY'),
'{"department":"IT","role":"Software Developer"}');

INSERT INTO persons (name, height, birthdate, person_data)
       VALUES ('Mary', 1.65, to_date('25/09/1982', 'DD/MM/YYYY'),
'{"department":"HR","role":"HR Manager"}');

INSERT INTO persons (name, height, birthdate, person_data)
       VALUES ('Bob', 1.75, to_date('11/03/1966', 'DD/MM/YYYY'),
'{"department":"IT","role":"Technical Consultant"}');

INSERT INTO persons (name, height, birthdate, person_data)
       VALUES ('Alice', 1.70, to_date('01/02/1987', 'DD/MM/YYYY'),
'{"department":"HR","role":"HR Assistant"}');


CREATE TABLE students (
      s_id NUMBER GENERATED ALWAYS AS IDENTITY (START WITH 1 INCREMENT BY 1),
      s_univ_id NUMBER,
      s_person_id NUMBER,
      subject VARCHAR2(10),
```

```
      CONSTRAINT stud_pk PRIMARY KEY (s_id),
      CONSTRAINT stud_fk_person FOREIGN KEY (s_person_id) REFERENCES persons(person_id),
      CONSTRAINT stud_fk_univ FOREIGN KEY (s_univ_id) REFERENCES university(id)
    );


INSERT INTO students(s_univ_id, s_person_id,subject) VALUES (1,1,'Arts');
INSERT INTO students(s_univ_id, s_person_id,subject) VALUES (1,3,'Music');
INSERT INTO students(s_univ_id, s_person_id,subject) VALUES (2,2,'Math');
INSERT INTO students(s_univ_id, s_person_id,subject) VALUES (2,4,'Science');


CREATE TABLE friendships (
    friendship_id NUMBER GENERATED ALWAYS AS IDENTITY (START WITH 1 INCREMENT BY 1),
    person_a NUMBER,
    person_b NUMBER,
    meeting_date DATE,
    CONSTRAINT fk_person_a_id FOREIGN KEY (person_a) REFERENCES persons(person_id),
    CONSTRAINT fk_person_b_id FOREIGN KEY (person_b) REFERENCES persons(person_id),
    CONSTRAINT fs_pk PRIMARY KEY (friendship_id)
);

INSERT INTO friendships (person_a, person_b, meeting_date) VALUES (1, 3,
to_date('01/09/2000', 'DD/MM/YYYY'));
INSERT INTO friendships (person_a, person_b, meeting_date) VALUES (2, 4,
to_date('19/09/2000', 'DD/MM/YYYY'));
INSERT INTO friendships (person_a, person_b, meeting_date) VALUES (2, 1,
to_date('19/09/2000', 'DD/MM/YYYY'));
INSERT INTO friendships (person_a, person_b, meeting_date) VALUES (3, 2,
to_date('10/07/2001', 'DD/MM/YYYY'));
```

The following statement creates a graph on top of the tables:

```
CREATE PROPERTY GRAPH students_graph
  VERTEX TABLES (
    persons KEY (person_id)
      LABEL person
        PROPERTIES (person_id, name, birthdate AS dob)
      LABEL person_ht
        PROPERTIES (height),
    university KEY (id)
  )
  EDGE TABLES (
    friendships AS friends
      KEY (friendship_id)
      SOURCE KEY (person_a) REFERENCES persons(person_id)
      DESTINATION KEY (person_b) REFERENCES persons(person_id)
      PROPERTIES (friendship_id, meeting_date),
    students AS student_of
      SOURCE KEY (s_person_id) REFERENCES persons(person_id)
      DESTINATION KEY (s_univ_id) REFERENCES university(id)
      PROPERTIES (subject)
  );
```

This creates the following graph:



Example: GRAPH_TABLE Query

The following query matches a pattern on graph `students_graph` to find friends of a person named John:

```
SELECT *
FROM GRAPH_TABLE ( students_graph
  MATCH (a IS person) -[e IS friends]- (b IS person)
  WHERE a.name = 'John'
  COLUMNS (b.name)
);
```

In the query:

- `(a IS person)` is a vertex pattern that matches vertices labeled person and binds the solutions to a variable `a`.
- `-[e IS friends]-` is an edge pattern that matches either incoming or outgoing edges labeled friends and binds the solutions to a variable `e`.
- `(b IS person)` is another vertex pattern that matches vertices labeled person and binds the solutions to a variable `b`.
- `WHERE a.name = 'John'` is a search condition that accesses the property `name` from vertices bound to variable a to compare against the value `John`.
- `COLUMNS (b.name)` specifies to return the property `name` of vertex `b` as part of the output table.

The output is:

```
NAME
----------
Mary
Bob
```

> **See Also:**
>
> - *SQL Property Graph*
> - For property graph definitions and terminology , see *CREATE PROPERTY GRAPH*.

Graph Reference

Purpose

Each GRAPH_TABLE starts with a graph reference that references the graph to perform the pattern matching on.

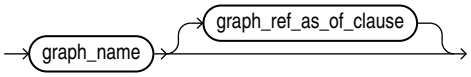
Prerequisites

To query a property graph, you must have `READ` or `SELECT` object privilege on the graph. Note that you do not require `READ` or `SELECT` object privilege on the tables or materialized views that underlie the graph.

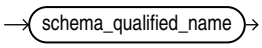
To issue an Oracle Flashback Query using the *`graph_ref_as_of_clause`* in `GRAPH_TABLE`, you must additionally have `FLASHBACK` object privilege on the tables and materialized views that underlie the graph. This is needed only for those tables and views that are accessed by the query, based on the specified graph pattern and label expressions used therein. Alternatively, you must have `FLASHBACK ANY TABLE` system privilege.

Syntax

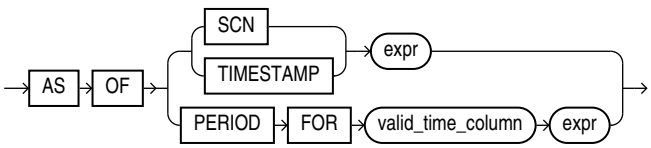
***graph_reference*::=**



***graph_name*::=**



***graph_ref_as_of_clause*::=**



Semantics

A graph name may be qualified with a schema name to allow for querying graphs created by other users. Furthermore, you can specify the *`graph_ref_as_of_clause`* clause to retrieve the

result of the graph query at a particular change number (`SCN`) or timestamp. If you specify `SCN`, then *`expr`* must evaluate to a number. If you specify `TIMESTAMP`, then *`expr`* must evaluate to a timestamp value. In either case, *`expr`* cannot evaluate to NULL.

Example 1

The following query counts the number of persons in the students_graph owned by user scott:

```
SELECT COUNT(*)
FROM GRAPH_TABLE ( scott.students_graph
  MATCH (a IS person)
  COLUMNS (a.name)
);
```

The output is:

```
  COUNT(*)
----------
         4
```

Example 2

The following example queries a graph at two different timestamps. It first inserts a new row into the university table that underlies the `students_graph`. It then queries versions of the graph before and after the insertion.

```
INSERT INTO university (name) VALUES ('u3');


SELECT COUNT(*)
FROM GRAPH_TABLE (
  students_graph
  MATCH (u IS university)
  COLUMNS (u.*)
);


SELECT COUNT(*)
FROM GRAPH_TABLE (
  students_graph AS OF TIMESTAMP (SYSTIMESTAMP - INTERVAL '2' MINUTE)
  MATCH (u IS university)
  COLUMNS (u.*)
);


DELETE FROM university WHERE name = 'u3';
```

The output of the first query is:

```
  COUNT(*)
----------
         3
```

The output of the second query is:

```
   COUNT(*)
----------
         2
```

Note: this example assumes that the second `SELECT` query is run at least two minutes after the graph was created and within two minutes after running the `INSERT` statement, otherwise the output is different.

Graph Pattern

Purpose

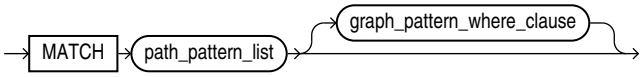
A graph pattern consists of a set of vertex and edge patterns together with search conditions. A graph pattern is matched against a graph to obtain a set of solutions containing bindings for each vertex and edge variable in the pattern.

This topic has the following sub-topics:

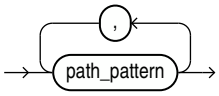
- Path Pattern
- Element Pattern
- Quantified Path Pattern
- Parenthesized Path Pattern
- Graph Pattern WHERE Clause

Syntax

***graph_pattern*::=**



***path_pattern_list*::=**



Semantics

A graph pattern contains the following parts:

- `MATCH` keyword.
- *`path_pattern_list`*: a list containing one or more comma-separated path patterns.
- *`graph_pattern_where_clause`*: an optional `WHERE` clause defining a search condition that may reference vertices and edges from the pattern.

Two path patterns inside the same `GRAPH_TABLE` may share vertex and edge variables to allow for creating more complex, non-linear patterns. Variables may also be repeated within a single path pattern to create a cyclic pattern. If multiple vertex or edge patterns share a variable then all the label expressions and element pattern `WHERE` clauses in those patterns must satisfy for binding to the element variable to occur.

If there are no shared variables between two path patterns, then the solution set is a cross product of the solutions of the individual path patterns.

Restrictions

A vertex variable may not have the same name as an edge variable.

Examples

Example 1

The following query finds cyclic paths from `Mary` via two other persons back to `Mary`. Only incoming edges are matched (`<-[..]-`).

```
SELECT *
FROM GRAPH_TABLE ( students_graph
  MATCH (a IS person) <-[e1 IS friends]- (b IS person)
            <-[e2 IS friends]- (c IS person)
               <-[e3 is friends]- (a IS person)
  WHERE a.name= 'Mary'
  COLUMNS (a.name AS person_a, b.name AS person_b, c.name AS person_c)
);
```

Here, the graph pattern consists of a single path pattern that has four vertex patterns and three edge patterns. The first vertex pattern shares a variable `a` with the last vertex pattern so that the pattern matches cyclic paths.

Only a single path matches the pattern:

```
PERSON_A   PERSON_B   PERSON_C
---------- ---------- ----------
Mary       Bob        John
```

Here, the output shows that a path was matched that starts in `Mary` with an incoming edge to `Bob`, followed by an incoming edge to `John`, followed by an incoming edge back to `Mary`.

The same query may also be expressed by breaking up the single path pattern into multiple path patterns as follows:

```
SELECT *
FROM GRAPH_TABLE ( students_graph
  MATCH (a IS person) <-[e1 IS friends]- (b IS person),
        (b) <-[e2 IS friends]- (c IS person),
        (c) <-[e3 is friends]- (a IS person)
  WHERE a.name= 'Mary'
  COLUMNS (a.name AS person_a, b.name AS person_b, c.name AS person_c)
);
```

Here, the first path pattern shares variable `b` with the second path pattern, the second path pattern shares variable `c` with the third path pattern, and the third path pattern shares variable `a` with the first path pattern.

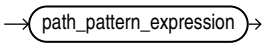
Path Pattern

Purpose

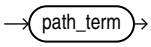
A path pattern specifies a linear pattern that matches a string of vertices and edges. Path patterns are made up of the concatenation of one or more vertex and edge patterns. Vertex and edge patterns may be quantified as well as parenthesized.

Syntax

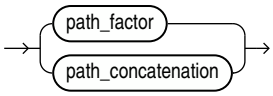
***path_pattern*::=**



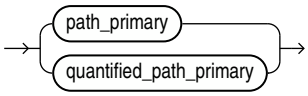
***path_pattern_expression*::=**



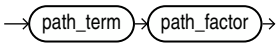
***path_term*::=**



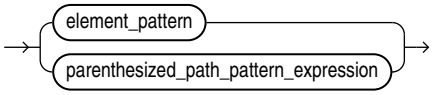
***path_factor*::=**



***path_concatenation*::=**



***path_primary*::=**



Semantics

Syntactically, a path pattern is the concatenation of one or more element patterns, which are either vertex patterns or edge patterns.

The element patterns of a path pattern are not required to alternate between vertex and edge patterns; there may be two consecutive edge patterns or two consecutive vertex patterns. These topologically inconsistent patterns are understood during pattern matching as follows:

- Two consecutive vertex patterns bind to the same vertex.
- Two consecutive edge patterns conceptually have an implicit vertex pattern between them.

Restricitons

Path patterns have the following restrictions:

- A path pattern may only contain two consecutive vertex patterns if one of the vertex patterns is contained in a parenthesized path pattern while the other one is not.
- A parenthesized path pattern must be quantified.

Examples

Example 1

The following query counts the number of vertices in the graph:

```
SELECT COUNT(*)
FROM GRAPH_TABLE ( students_graph
  MATCH (v)
  COLUMNS (1 AS dummy)
);
```

Note that the `COLUMNS` clause needs to contain at least one expression, hence a dummy value is projected but it is not returned from the query.

The result is:

```
  COUNT(*)
----------
         6
```

Example 2

The following query counts the number of edges in the graph:

```
SELECT COUNT(*)
FROM GRAPH_TABLE ( students_graph
  MATCH -[e]->
  COLUMNS (1 AS dummy)
);
```

The result is:

```
  COUNT(*)
----------
         8
```

Example 3

The following query finds persons that are two friend hops away from `Mary`, following either incoming or outgoing friends edges:

```
SELECT *
FROM GRAPH_TABLE ( students_graph
  MATCH (n IS person) -[IS friends]- () -[IS friends]- (m IS person)
  WHERE n.name = 'Mary' AND m.name <> n.name
```

```
  COLUMNS (m.name AS fof)
);
```

In the path pattern above:

- (`n IS person`) is a vertex pattern that has a variable `n` and a label expression `IS person`.
- `-[IS friends]-` is an any-directed edge pattern that has an implicit variable and a label expression `IS friends`.
- `()` is a vertex pattern that has an implicit variable and no label expression such that it matches vertices having any label(s).
- `-[IS friends]-` is again an any-directed edge pattern that has an implicit variable and a label expression `IS friends`.
- (`n IS person`) is a vertex pattern that has a variable `n` and a label expression `IS person`.

The result is:

```
FOF
----------
Bob
John
```

Note that the query above can also be expressed as:

```
SELECT *
FROM GRAPH_TABLE ( students_graph
  MATCH (n IS person) -[IS friends]- -[IS friends]- (m IS person)
  WHERE n.name = 'Mary' AND m.name <> n.name
  COLUMNS (m.name AS fof)
);
```

Here, the vertex pattern between the two edge patterns is implicit.

The same query can be expressed using a quantifier to avoid the repeated specification of the same edge pattern:

```
SELECT *
FROM GRAPH_TABLE ( students_graph
  MATCH (n IS person) -[IS friends]-{2}(m IS person)
  WHERE n.name = 'Mary' AND m.name <> n.name
  COLUMNS (m.name AS fof)
);
```

Quantified path patterns may be parenthesized:

```
SELECT *
FROM GRAPH_TABLE ( students_graph
  MATCH (n IS person) (-[IS friends]-){2}(m IS person)
  WHERE n.name = 'Mary' AND m.name <> n.name
  COLUMNS (m.name AS fof)
);
```

Note that each of the syntax variations above gives the same result:

```
FOF
----------
```

```
Bob
John
```

Element Pattern

Purpose

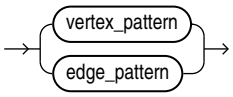
An element pattern is either a vertex pattern or an edge pattern. The result of matching an element pattern is the binding of vertices or edges to the implicitly or explicitly declared variable of the element pattern.

This section comprises the following sections:

- Vertex Pattern
- Edge Pattern
- Element Pattern Filler
- Element Variable
- Label Expression
- Element Pattern WHERE Clause

Syntax

***element_pattern*::=**



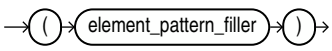
Vertex Pattern

Purpose

A vertex pattern is a pattern that matches vertices in a graph. The result of such matching is the binding of a set of vertices to the implicitly or explicitly declared variable of the vertex pattern.

Syntax

***vertex_pattern*::=**



Semantics

Visually, a vertex pattern has two parentheses ( ) to mimic a circle since vertices are typically represented by circles in visualizations of graphs.

Examples

Example 1

The following query counts the number of vertices in the graph:

```
SELECT COUNT(*)
FROM GRAPH_TABLE ( students_graph
  MATCH (v)
  COLUMNS (1 AS dummy)
);
```

Note that the `COLUMNS` clause needs to contain at least one expression, hence a dummy value is projected but it is not returned from the query.

The result is:

```
  COUNT(*)
----------
       6
```

Example 2

The following query matches all persons with a date of birth greater than 1 January 1980:

```
SELECT name, birthday
FROM GRAPH_TABLE ( students_graph
  MATCH (p IS person WHERE p.dob > DATE '1980-01-01')
  COLUMNS (p.name, p.dob AS birthday)
)
ORDER BY birthday;
```

The result is:

```
NAME       BIRTHDAY
---------- ---------
Mary       25-SEP-82
Alice      01-FEB-87
```

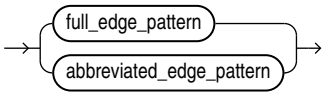
Edge Pattern

Purpose

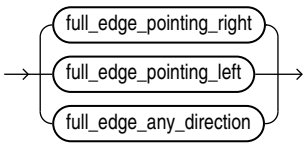
An edge pattern is a pattern that matches edges in a graph. The result of such matching is the binding of a set of edges to the implicitly or explicitly declared variable of the edge pattern.

Syntax

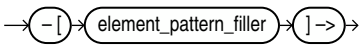
***edge_pattern*::=**



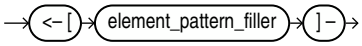
***full_edge_pattern*::=**



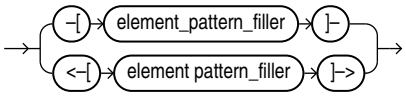
***full_edge_pointing_right*::=**



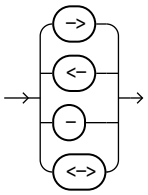
***full_edge_pointing_left*::=**



***full_edge_any_direction*::=**



***abbreviated_edge_pattern*::=**



Semantics

Visually, an edge pattern mimics an arrow since edges are typically represented by arrows in visualizations of graphs. For example, <-[]- or <- are incoming edge patterns because they look like incoming arrows, while -[]-> or -> are outgoing edge patterns because they look like outgoing arrows.

An edge_pattern is either a full_edge_pattern or an abbreviated_edge_pattern. The full edge pattern has an element_pattern_filler with optional element pattern variable, label expression and element pattern WHERE clause, while the abbreviated edge pattern provides syntactic sugar in case none of the three optional filler parts are needed.

The following table summarizes the options:

Table 4-6 Summary of Edge Patterns

Directionality	Full Edge Pattern	Abbreviated Edge Pattern
Directed pointing to the right	-[ ] ->	->
Directed pointing to the left	<-[ ]-	<-
Any-Directed: pointing to the right or the left	-[ ]- or <-[ ]->	-

Note that since the abbreviated syntax does not allow for providing a variable name, a label expression, or an element pattern `WHERE` clause, abbreviated edge patterns match with all edges in the graph that have the specified direction.

Examples

Example 1

The following query counts the number of edges in the graph:

```
SELECT COUNT(*)
FROM GRAPH_TABLE ( students_graph
  MATCH ->
  COLUMNS (1 AS dummy)
);
```

The result is:

```
  COUNT(*)
----------
         8
```

Example 2

The following query matches all `friends` edges that have a property `meeting_date` with a value greater than `DATE '2000-01-01'`:

```
SELECT *
FROM GRAPH_TABLE ( students_graph
  MATCH -[e IS friends WHERE e.meeting_date > DATE '2001-01-01']->
  COLUMNS (e.meeting_date)
);
```

The result is:

```
MEETING_D
---------
10-JUL-01
```

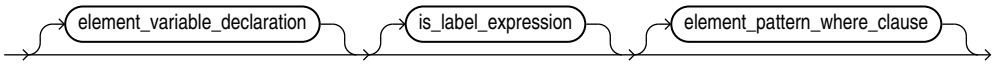
Element Pattern Filler

Purpose

Vertex patterns and full edge patterns have a filler for providing an optional variable declaration, an optional label expression, and an optional `WHERE` clause.

Syntax

***element_pattern_filler*::=**



Semantics

Vertex patterns and full edge patterns and have a filler containing the following parts:

- An optional *element_variable_declaration* for providing a variable name for the element pattern so that the element can be referenced elsewhere, for example in `WHERE` and `COLUMNS` clauses. If no variable name is specified, a variable is implicit and cannot be referenced.
- An optional *is_label_expression* for defining a label expression. Vertices and edges only match if they satisfy the specified label expression.
- An optional *element_pattern_where_clause* for defining an in-lined search condition. Vertices and edges only match if they satisfy the specified search condition.

Examples

Example 1

The following query finds persons that are two friend hops away from `Mary`, following either incoming or outgoing friends edges:

```
SELECT *
FROM GRAPH_TABLE ( students_graph
  MATCH (n IS person WHERE n.name = 'Mary')
          -[e IS friends WHERE e.meeting_date > DATE '2001-01-01']-
            () -[IS friends]- (m IS person)
  WHERE m.name <> n.name
  COLUMNS (m.name, e.meeting_date)
);
```

In the path pattern above:

- (`n IS person WHERE n.name = 'Mary'`) is a vertex pattern that has a variable `n`, a label expression `IS person` and an element pattern `WHERE` clause `WHERE n.name = 'Mary'`.
- `-[e IS friends WHERE e.meeting_date > DATE '2001-01-01']-` is an any-directed edge pattern that has a variable `e`, a label expression `IS friends` and an element pattern `WHERE` clause `WHERE e.meeting_date > DATE '2001-01-01'`.
- `()` is a vertex pattern that has an implicit variable and neither has a label expression nor an element pattern `WHERE` clause.
- `-[IS friends]-` is an any-directed edge pattern that has an implicit variable, a label expression `IS friends` but no element pattern `WHERE` clause.
- (`n IS person`) is a vertex pattern that has a variable `n`, a label expression `IS person` but no element pattern `WHERE` clause.

The result is:

```
NAME       MEETING_D
---------- ---------
John       10-JUL-01
```

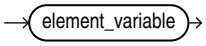
Element Variable

Purpose

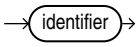
Element variables are either vertex or edge variables. During pattern matching, the variables will bind to sets of vertices or edges in the graph. Element variables can be referenced from other places in the query to access data of vertices and edges, such as their property values.

Syntax

***element_variable_declaration*::=**



***element_variable*::=**



Semantics

Syntactically, an *element_variable_declaration* is an identifier and can thus be either double quoted or unquoted. Declaring an element variable is optional and if no element variable is declared then the element pattern has an implicit variable with an (implicit) unique name. Implicit variables cannot be referenced elsewhere in the query.

Multiple vertex patterns may declare the same element variable and multiple edge patterns may also declare the same element variable. In such cases, there are not multiple variables but there is a single variable that is shared by the different vertex or edge patterns.

Declared variables are visible within the `GRAPH_TABLE` in which they are declared. They may be referenced in `WHERE` and `COLUMNS` clauses defined in the same `GRAPH_TABLE`.

If an element variable is declared in a quantified path pattern, then it may bind to more than one vertex or edge within a single solution to the pattern. References are interpreted contextually: if the reference occurs outside the quantified path pattern, then the reference is to the complete list of graph elements that are bound to the element variable. In this circumstance, the element variable is said to have group degree of reference. However, if the reference does not cross a quantifier, then the reference has singleton degree of reference.

For example, in `(X) -[E WHERE E.P > 1]->{1,10} (Y) WHERE SUM(E.P) < 100` the edge variable `E` is referenced twice: once in the edge pattern and once outside the edge pattern. Within the edge pattern, `E` has singleton degree of reference and the property reference `E.P` references a property of a single edge. On the other hand, the reference within the `SUM` aggregate has group degree of reference (because of the quantifier `{1,10}`) and references the list of edges that are bound to `E`.

Restrictions

- A vertex pattern may not declare a variable with the same name as an edge pattern.
- A quantified path pattern may not declare a variable with the same name as an element variable declared outside of the quantified path pattern.

Examples

Example 1

The following query finds friends of friends of John following incoming or outgoing edges that have a property `meeting_date` with a value greater than `DATE '2000-09-015'`:

```
SELECT DISTINCT name
FROM GRAPH_TABLE ( students_graph
  MATCH (a IS person) -[e IS friends WHERE e.meeting_date > DATE '2000-09-15']-{2} ("b"
IS person)
  WHERE a.name = 'John' AND a.name <> "b".name
  COLUMNS ("b".name)
);
```

In the query above, `a` and `"b"` are vertex variables, `e` is an edge variable and `e.meeting_date`, `a.name` and `"b".name` are property references that access a property value of the referenced vertex or edge.

The result shows that John has two such friends of friends:

```
NAME
----------
Bob
Alice
```

Example 2

The following query finds friends of Mary and the universities that Mary and her friends went to:

```
SELECT *
FROM GRAPH_TABLE ( students_graph
  MATCH (p1 IS person) -[e1 IS friends]- (p2 IS person)
      , (p1) -[IS student_of]-> (u1 IS university)
      , (p2) -[IS student_of]-> (u2 IS university)
  WHERE p1.name = 'Mary'
  COLUMNS (p1.name, p2.name AS friend, e1.meeting_date, u1.name AS univ_1, u2.name AS
univ_2)
);
```

In the query above, `p1`, `p2`, `u1` and `u2` are vertex variables, while `e1` is an edge variable. The pattern `-[IS student_of]->` appears twice and implicitly declares two unique variables that cannot be referenced. Furthermore, there are two vertex patterns that share variable `p1` and there are two vertex patterns that share variable `p2`. Vertices will only bind to such variable if both vertex patterns match.

The result shows that Mary has three friends, one of which goes to the same university `XYZ`, while two other friends go to a different university `ABC`:

```
NAME       FRIEND     MEETING_D UNIV_1     UNIV_2
```

```
---------- ---------- --------- ---------- ----------
Mary       John       19-SEP-00 XYZ        ABC
Mary       Bob        10-JUL-01 XYZ        ABC
Mary       Alice      19-SEP-00 XYZ        XYZ
```

Example 3

The following query finds all paths that have a length between 2 and 5 edges (`{2,5}`), starting from a person named Alice and following both incoming and outgoing edges labeled `friends`. Edges along paths should not be traversed twice (`COUNT(e.friendship_id)` = `COUNT(DISTINCT e.friendship_id)`). The query returns all friendship IDs along paths as well as the length of each path.

```
SELECT *
FROM GRAPH_TABLE ( students_graph
       MATCH (p IS person) -[e IS friends]-{2,5} (friend IS person)
       WHERE p.name = 'Alice' AND
             COUNT(e.friendship_id) = COUNT(DISTINCT e.friendship_id)
       COLUMNS (LISTAGG(e.friendship_id, ', ') AS friendship_ids,
                COUNT(e.friendship_id) AS path_length));
```

Note that in the element pattern `WHERE` clause of the query above, `p.name` references a property of a single edge, while `e.friendship_id` within the `COUNT` aggregate accesses a list of property values since the edge variable `e` is enclosed by the quantifier `{2,5}`. Similarly, the two property references in the `COLUMNS` clause both access a list of property values.

The result is:

```
FRIENDSHIP_IDS    PATH_LENGTH
----------------- -----------
2, 3              2
2, 4              2
2, 3, 1           3
2, 4, 1           3
2, 3, 1, 4        4
2, 4, 1, 3        4
```

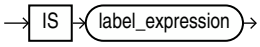
Label Expression

Purpose

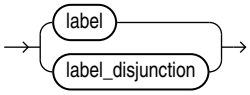
Label expressions are used to limit the search to only vertices or edges of a specific type.

Syntax

***is_label_declaration*::=**

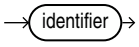


***label_expression*::=**



***label_disjunction*::=**

***label*::=**



Semantics

Syntactically, an `is_label_declaration` starts with the keyword `IS` followed by a `label_expression`, which is either a `label` or a `label_disjunction` denoted by a vertical bar |. A label itself is an identifier and can thus be double quoted or unquoted.

An element pattern matches only vertices and edges that satisfy the label expression. If the label expression is omitted, then all vertices and edges are matched irrespective of their labels.

Examples

Example 1

The following query matches all vertices labeled person or university and retrieves their name and date of birth properties:

```
SELECT *
FROM GRAPH_TABLE ( students_graph
  MATCH (x IS person|university)
  COLUMNS (x.name, x.dob)
)
ORDER BY name;
```

The result is:

```
NAME       DOB
---------- ---------
ABC
Alice      01-FEB-87
Bob        11-MAR-66
John       13-JUN-63
Mary       25-SEP-82
XYZ
```

Above, since universities do not have a date of birth, a null value is returned and shows up as empty string in the `DOB` column.

Example 2

The following query matches outgoing edges labeled `student_of` or `friends` from a person named Mary to a vertex `m` that is labeled `university` or `"PERSON"`:

```
SELECT *
FROM GRAPH_TABLE ( students_graph
```

```
  MATCH (n IS person) -[e IS student_of|friends]-> (m IS university|"PERSON")
  WHERE n.name = 'Mary'
  COLUMNS (e.subject, e.meeting_date, m.name)
)
ORDER BY subject, meeting_date, name;
```

The result is:

```
SUBJECT    MEETING_D NAME
---------- --------- ----------
Math                 XYZ
           19-SEP-00 Alice
           19-SEP-00 John
```

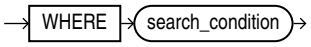
Element Pattern WHERE Clause

Purpose

The element pattern `WHERE` clause specifies a search condition that is syntactically placed inside a vertex or an edge pattern and that needs to be satisfied by the vertex or edge for the pattern to match.

Syntax

***element_pattern_where_clause*::=**



Semantics

Syntactically, the *element_pattern_where_clause* starts with the keyword `WHERE` and is followed by a *search_condition*, which is an arbitrary boolean value expression.

The element pattern `WHERE` clause may reference any graph element variable in the graph pattern. If the variable has group degree of reference, then the reference must be inside the arguments of an aggregate function. See Aggregation in GRAPH_TABLE. There is no requirement that the search condition must reference the variable of the element pattern itself, but for improved query readability it is generally recommended that it always does such that any search condition that does not reference the element variable is placed in the graph pattern `WHERE` clause instead.

Examples

Example 1

The following query finds all friends of John whom he met after `15 September 2000`:

```
SELECT Gt.name
FROM GRAPH_TABLE ( students_graph
  MATCH (a IS person WHERE a.name = 'John')
          -[e IS friends WHERE e.meeting_date > DATE '2000-09-15']-
            (b IS person)
  COLUMNS (b.name)
) GT;
```

The example above contains two element pattern `WHERE` clauses:

- `WHERE a.name = 'John'`
- `WHERE e.meeting_date > DATE '2000-09-15'.`

The result is:

```
NAME
----------
Mary
```

Quantified Path Pattern

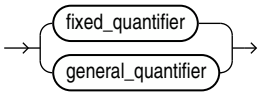
Purpose

Quantified path patterns allow for repeated matching of a path pattern, typically for the purpose of matching variable-length paths. The specified quantifier determines a minimum and maximum for the number of times to match the path pattern.

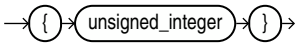
Syntax

***quantified_path_primary*::=**

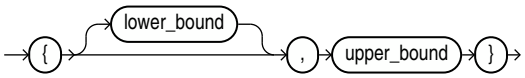
***graph_pattern_quantifier*::=**



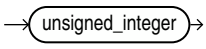
***fixed_quantifier*::=**



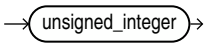
***general_quantifier*::=**



***lower_bound*::=**



***upper_bound*::=**



Semantics

A *quantified_path_primary* is a path_primary together with a quantifier. Here, the path primary must be either an edge pattern or a parenthesized path pattern.

A graph_pattern_quantifier is either:

- A *fixed_quantifier*, which is an unsigned integer placed between curly braces. The integer value specifies an exact number of times the pattern should be matched. In other words, the lower bound on the number of times to match the pattern is the same as the upper bound.
- A *general_quantifier*, which has an optional *lower_bound*, a comma (,) and a mandatory *upper_bound*, all of which are placed between curly braces. Lower and upper bound are unsigned integers and specify a minimum and a maximum number of times to match the path pattern. If no lower bound is specified, then the lower bound is zero (0).

The following table summarizes the options:

Table 4-7 Quantifier Table

Quantifier	Meaning
`{ n }`	Exactly `n`
`{ n, m }`	Between `n` and `m` (inclusive)
`{ , m }`	Between zero (0) and `m` (inclusive)

Restrictions

The following restrictions apply to quantified path patterns:

- The path primary that is quantified must be either an edge pattern or a parenthesized path pattern. For example, vertex patterns cannot be quantified unless they appear together with at least one edge pattern inside a parenthesized path pattern.
- The lower bound should be 0 or greater, while the upper bound should be 1 or greater and should also be greater than or equal to the lower bound.
- Nested quantifiers are not allowed.

Examples

Example 1

The following query finds friends of friends of John following incoming or outgoing edges that have a property `meeting_date` with a value greater than `DATE '2000-090-15'`:

```
SELECT DISTINCT name
FROM GRAPH_TABLE ( students_graph
  MATCH (a IS person)
          -[e IS friends WHERE e.meeting_date > DATE '2000-09-15']-{2}
            (b IS person)
```

```
  WHERE a.name = 'John' AND a.name <> b.name
  COLUMNS (b.name)
);
```

In the query above, the path pattern `-[e IS friends WHERE e.meeting_date > DATE '2000-09-15']-` is quantified with the fixed quantifier `{2}` to indicate that the edge pattern should match exactly twice.

The result is:

```
NAME
----------
Bob
Alice
```

The same query may be written using a parenthesized path pattern too. The following are all syntactic alternatives, the latter two use a parenthesized path pattern:

- `-[e IS friends WHERE e.meeting_date > DATE '2000-09-15']-{2}`
- `(-[e IS friends WHERE e.meeting_date > DATE '2000-09-15']-){2}`
- `(-[e IS friends]- WHERE e.meeting_date > DATE '2000-09-15'){2}`

Example 2

The following query finds persons that can be reached from Mary within three hops, following only persons that are taller than Mary.

```
SELECT DISTINCT name, height
FROM GRAPH_TABLE ( students_graph
  MATCH (a IS person|person_ht)
           (-[e IS friends]- (x IS person_ht) WHERE x.height > a.height) {,3}
             (b IS person|person_ht)
  WHERE a.name = 'Mary'
  COLUMNS (b.name, b.height)
)
ORDER BY height;

The result is:
NAME           HEIGHT
---------- ----------
Mary             1.65
Alice             1.7
Bob              1.75
John              1.8
```

Note that the reason Mary is included in the result is because the specified quantifier `{,3}` has a lower bound of zero such that the quantified pattern is allowed to match zero times in which case variables `a` and `b` bind to the same vertex corresponding to Mary.

Example 3

The following query finds all paths between university `ABC` and university `XYZ` such that paths have a length of up to 3 edges (`{,3}`). For each path, a JSON array is returned such that the array contains the `friendship_id` value for edges labeled `friends`, and the subject value for edges labeled `student_of`. Note that the `friendship_id` property is cast to `VARCHAR(100)` to make it type-compatible with the subject property.

```
SELECT *
```

```
FROM GRAPH_TABLE ( students_graph
       MATCH (u1 IS university) -[e]-{,3} (u2 IS university)
       WHERE u1.name = 'ABC' AND u2.name = 'XYZ'
       COLUMNS (JSON_ARRAYAGG(CASE WHEN e.subject IS NOT NULL THEN e.subject
                              ELSE CAST(e.friendship_id AS VARCHAR(100)) END) AS path));
```

The result is:

```
PATH
----------------------
["Arts","3","Math"]
["Music","4","Math"]
```

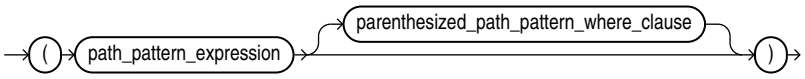
Parenthesized Path Pattern

Purpose

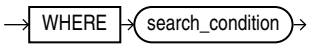
Parenthesized path patterns allow for defining more complex quantified path pattern expressions.

Syntax

***parenthesized_path_pattern_expression*::=**



***parenthesized_path_pattern_where_clause*::=**



Semantics

A *parenthesized_path_pattern_expression* is a *path_pattern_expression* together with an optional *parenthesized_path_pattern_where_clause*, placed in between parentheses.

Parenthesized path patterns allow for the quantification of any path pattern expression that contains at least one edge pattern. Without parentheses, only a single edge pattern can be quantified.

The parenthesized path pattern `WHERE` clause may reference vertex and edge variables declared in the parenthesized path pattern itself as well as vertex and edge variables declared outside of the parenthesized path pattern. If the variable has group degree of reference, then the reference must be inside the arguments of an aggregate function. See Aggregation in GRAPH_TABLE.

Restrictions

The following restrictions apply to parenthesized path pattern expressions:

- Each parenthesized path pattern needs to be quantified.
- There can only be a single level of parentheses. Nesting of parenthesized path patterns is not allowed.

Examples

Example 1

The following query finds persons that can be reached from Bob within one to three hops (`{1,3}`) such that for each consecutive pair of persons along the path, the first person has a date of birth that is smaller than the date of birth of the second person.

```
SELECT DISTINCT name, birthday
FROM GRAPH_TABLE ( students_graph
  MATCH
   (a IS person)
     ( (x) -[e IS friends]- (y IS person)
       WHERE x.dob < y.dob ){1,3}
         (b IS person)
  WHERE a.name = 'Bob'
  COLUMNS (b.name, b.dob AS birthday)
)
ORDER BY birthday;
```

The result is:

```
NAME       BIRTHDAY
---------- ---------
Mary       25-SEP-82
Alice      01-FEB-87
```

Example 2

The following query finds all paths that have a length between 2 and 3 edges ({2,3}), starting from a person named John and following only outgoing edges labeled friends and vertices labeled person. Vertices along paths should not have the same person_id as John (WHERE p.person_id <> friend.person_id).

```
SELECT *
FROM GRAPH_TABLE ( students_graph
       MATCH (p IS person) ( -[e IS friends]-> (friend IS person)
                             WHERE p.person_id <> friend.person_id){2,3}
       WHERE p.name = 'John'
       COLUMNS (COUNT(e.friendship_id) AS path_length,
                LISTAGG(friend.name, ', ') AS names,
                LISTAGG(e.meeting_date, ', ') AS meeting_dates ));
```

Above, the `COLUMNS` clause contains three aggregates, the first to compute the length of each path, the second to create a comma-separated list of person names along paths, and the third to create a comma-separate list of meeting dates along paths.

The result of the query is:

```
PATH_LENGTH NAMES
MEETING_DATES
----------- -------------------
-----------------------------------
          2 Bob, Mary           01-SEP-00, 10-
JUL-01
          3 Bob, Mary, Alice    01-SEP-00, 10-JUL-01, 19-SEP-00
```

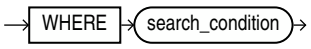
Graph Pattern WHERE Clause

Purpose

The graph pattern `WHERE` clause specifies a search condition that is syntactically placed at the end of the graph pattern and that needs to be satisfied by the complete graph pattern in order for the graph pattern to match.

Syntax

***graph_pattern_where_clause*::=**



Semantics

Syntactically, the graph pattern `WHERE` clause starts with the keyword `WHERE` and is followed by a *`search_condition`*, which is an arbitrary boolean value expression.

The graph pattern `WHERE` clause may reference any element variables in the graph pattern. If the variable has group degree of reference, then the reference must be inside the arguments of an aggregate function. See Aggregation in GRAPH_TABLE .

Examples

Example 1

The following query finds all friends of John whom he met after 15 September 2000:

```
SELECT Gt.name
FROM GRAPH_TABLE ( students_graph
  MATCH (a IS person) -[e IS friends]- (b IS person)
  WHERE a.name = 'John' AND e.meeting_date > DATE '2000-09-15'
  COLUMNS (b.name)
) GT;
```

Note that the two conditions are placed together in the graph pattern `WHERE` clause to form a single search that needs to be satisfied by the pattern: `WHERE a.name = 'John' AND e.meeting_date > DATE '2000-09-15'`.

The result is:

```
NAME
----------
Mary
```

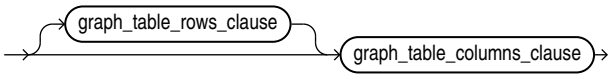
Graph Table Shape

Purpose

A graph table shape defines how the result of pattern matching should be transformed into tabular form. This is done through the *`graph_table_rows_clause`* and *`graph_table_columns_clause`* clauses.

Syntax

***graph_table_shape*::=**



COLUMNS Clause

Rows Clause

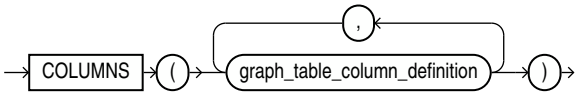
COLUMNS Clause

Purpose

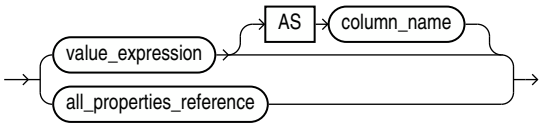
The `COLUMNS` clause allows for defining a projection that transforms the result of graph pattern matching into a regular table that no longer contains graph objects like vertices and edges but instead regular data values only.

Syntax

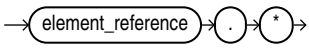
***graph_table_columns_clause*::=**



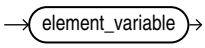
***graph_table_column_definition*::=**



***all_properties_reference*::=**



***element_reference*::=**



Semantics

Syntactically, the `COLUMNS` clause starts with the keyword `COLUMNS` and is followed by an opening parenthesis, a comma-separated list of one or more *`graph_table_column_definition`* and a closing parenthesis.

A *`graph_table_column_definition`* defines either:

- A single output column via an arbitrary value expression. The value expression may contain references to vertices and edges in the graph pattern, for example to access property values of vertices and edges. An optional alias, `AS` *`column_name`* provides a name for the column. The alias can only be omitted if the value expression is a property reference, in which case the alias defaults to the property name.
- An *`all_properties_reference`* that expands to the set of all valid properties based on the element type (vertex or edge) and any label expression specified for the element. The set of properties is the union of properties of the vertex (or edge) labels belonging to tables that have at least one label that satisfies the label expression. In case some of these matching tables define a property while other tables do not, then NULL values will be returned for those tables that do not define the property.

An optional alias, `AS` *`column_name`* provides a name for the column. The alias can only be omitted if the value expression is a property reference, in which case the alias defaults to the property name.

Examples

Example 1

The following example returns the name of each person as well as the height in feet by multiplying the height in meters by 3.281:

```
SELECT *
FROM GRAPH_TABLE ( students_graph
  MATCH (n IS person|person_ht)
  COLUMNS (n.name, n.height * 3.281 AS height_in_feet)
)
ORDER BY name;
```

In the query above, the `COLUMNS` clause defines two columns. Note that `n.name` is short for `n.name AS name`.

The result is:

```
NAME       HEIGHT_IN_FEET
---------- --------------
Alice              5.5777
```

Example 2

The following query matches all `FRIENDS` edges between two persons `P1` and `P2` and uses all properties references `P1.*` and `E.*` to retrieve all the properties of vertex `P1` as well as all properties of edge `E`:

```
SELECT *
FROM GRAPH_TABLE ( students_graph
  MATCH (p1 IS person) -[e IS friends]-> (p2 IS person)
  COLUMNS ( p1.*, p2.name AS p2_name, e.* )
```

```
)
ORDER BY 1, 2, 3, 4, 5;
```

The result is:

```
PERSON_ID NAME DOB       HEIGHT P2_NAME FRIENDSHIP_ID MEETING_D
--------- ---- --------- ------ ------- ------------- ---------
        1 John 13-JUN-63   1.8   Bob             1 01-SEP-00
        2 Mary 25-SEP-82   1.65  Alice              2 19-SEP-00
        2 Mary 25-SEP-82   1.65  John               3 19-SEP-00
        3 Bob  11-MAR-66   1.75  Mary               4 10-JUL-01
```

Note that the result for `P1.*` includes properties `PERSON_ID`, `NAME` and `DOB` of label `PERSON` as well as property `HEIGHT` of label `PERSON_HT`. Furthermore, the result for `E.*` includes properties `FRIENDSHIP_ID` and `MEETING_DATE` of label `FRIENDS`.

Example 3

The following query matches all vertices in the graph and retrieves all their properties:

```
SELECT *
FROM GRAPH_TABLE ( students_graph
  MATCH (v)
  COLUMNS ( v.* )
)
ORDER BY 1, 2, 3, 4, 5;
```

The result is:

```
 PERSON_ID NAME       DOB       HEIGHT     ID
---------- ---------- --------- ---------- ----------
         1 John       13-JUN-63        1.8
         2 Mary       25-SEP-82       1.65
         3 Bob        11-MAR-66       1.75
         4 Alice      01-FEB-87        1.7
           ABC                                      1
           XYZ                                      2
```

Note that since `PERSON` vertices do not have an `ID` property, `NULL` values (empty strings) are returned. Similarly, `UNIVERSITY` vertices do not have `PERSON_ID`, `DOB` and `HEIGHT` properties so again `NULL` values (empty strings) are returned.

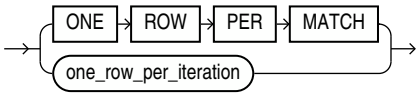
Rows Clause

Purpose

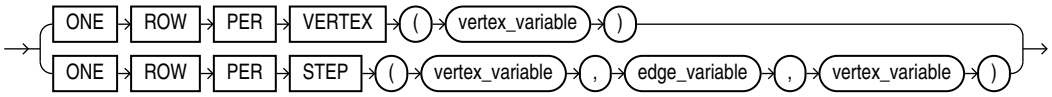
The `GRAPH_TABLE` rows clause is used to specify how many rows should be returned from `GRAPH_TABLE`, based on the number of matches to the graph pattern or the number of vertices or steps in such matches.

Syntax

***graph_table_rows_clause*::=**



***one_row_per_iteration*::=**



graph_table_rows_clause

- `ONE ROW PER MATCH`, the default, specifies that one row is returned per match to the graph pattern.
- *`one_row_per_iteration`* declares one or more iterator variables and returns one row per iteration. In particular:
  - `ONE ROW PER VERTEX` declares a single iterator vertex variable. It iterates through the vertices in paths and binds the iterator variable to different vertices in different iterations. For each path, it creates as many rows as there are vertices in the path. For example, if a pattern matches two paths, one with 3 vertices and another with 5 vertices, then a total of 8 rows are returned.
  - `ONE ROW PER STEP` declares an iterator vertex variable, an iterator edge variable, and another iterator vertex variable. It iterates through the steps of the different paths. A step is a vertex-edge-vertex triple. If a path is non-empty and thus contains at least one edge and two vertices, then there are as many steps as there are edges and each iteration binds the iterator variables to the next edge and its source and destination in the path. However, if a path is empty and consists of a single vertex only, then the path has a single step and the first iterator vertex variable binds to that vertex, while the iterator edge variable and the second iterator vertex variable are not bound to any graph element.

When an *`all_properties_reference`* contains a reference to an iterator variable, then depending on the type of the iterator variable, it expands to either all vertex properties or to all edge properties in the graph. Note that label expressions for elements in the graph pattern are not considered when expanding the properties of an iterator variable.

See #unique_387/unique_387_Connect_42_GUID-252808EF-CF93-420C-8DE9-73E5625B4577 *all_properties_clause* of `COLUMNS`.

Restrictions

The *`graph_table_rows_clause`* clause is subject to the following restrictions:

- If *`one_row_per_iteration`* is used then the graph pattern must consist of exactly one path pattern.

- Iterator element variables cannot be multiply declared. This means that an iterator variable may not be declared with the same name as an element variable declared in the graph pattern, or as another iterator variable.
- Iterator variables may only be referenced in the `COLUMNS` clause but not in the graph pattern or in the graph pattern `WHERE` clause.

Examples

Example 1

The following query finds all friends path with length between 0 and 3 starting from a person named John. It outputs one row per vertex.

```
SELECT *
FROM GRAPH_TABLE ( students_graph
       MATCH (n IS person) -[e1 IS friends]->{0,3} (IS person)
       WHERE n.name = 'John'
       ONE ROW PER VERTEX (v)
       COLUMNS (
        LISTAGG(e1.friendship_id, ', ') AS friendship_ids,
        v.name)
     );
```

The results are:

```
FRIENDSHIP_IDS       NAME
-------------------- ---------------
                     John
1                    John
1                    Bob
1, 4                 John
1, 4                 Bob
1, 4                 Mary
1, 4, 3              John
1, 4, 3              Bob
1, 4, 3              Mary
1, 4, 3              John
1, 4, 2              John
1, 4, 2              Bob
1, 4, 2              Mary
1, 4, 2              Alice
```

The results above show data from five paths that were matched:

- The empty path (zero *friendship_ids*) contains a single person named John.
- The path with *friendship_ids* 1 contains two persons named John and Bob.
- The path with *friendship_ids* 1, 4 contains three persons named John, Bob and Mary.
- The path with *friendship_ids* 1, 4, 3 contains four persons named John, Bob, Mary and John (this is a cycle).
- The path with *friendship_ids* 1, 4, 2 contains four persons named John, Bob, Mary and Alice.

Example 2

The following query again finds all friends path with length between 0 and 3 starting from a person named John. This time it outputs one row per step.

```
SELECT *
FROM GRAPH_TABLE ( students_graph
       MATCH (n IS person) -[e1 IS friends]->{0,3} (IS person)
       WHERE n.name = 'John'
       ONE ROW PER STEP (src, e2, dst)
       COLUMNS (
         LISTAGG(e1.friendship_id, ', ') AS friendship_ids,
         src.name AS src_name,
         e2.friendship_id,
         dst.name AS dst_name)
     );
```

The results are:

```
FRIENDSHIP_IDS       SRC_NAME   FRIENDSHIP_ID DST_NAME
-------------------- ---------- ------------- ----------
                     John
1                    John       1             Bob
1, 4                 John       1             Bob
1, 4                 Bob        4             Mary
1, 4, 3              John       1             Bob
1, 4, 3              Bob        4             Mary
1, 4, 3              Mary       3             John
1, 4, 2              John       1             Bob
1, 4, 2              Bob        4             Mary
1, 4, 2              Mary       2             Alice
```

The results above show data from five paths that were matched:

- The empty path (no *friendship_ids*) has a single step in which iterator vertex variable `src` is bound to the vertex corresponding to the person named John, while iterator edge variable `e2` and iterator vertex variable dst are not bound, resulting in NULL values for `FRIENDSHIP_ID` and `DST_NAME`.
- The path with *friendship_ids* 1 has a single step since it has a single edge. In this step, iterator vertex variable `src` is bound to the vertex corresponding to John, iterator edge variable `e2` is bound to the edge with *friendship_ids* 1, and iterator vertex variable `dst` is bound to the vertex corresponding to Bob.
- The path with *friendship_ids* 1, 4 has two steps since it has two edges.
- The path with *friendship_ids* 1, 4, 3 has three steps since it has three edges.
- The path with *friendship_ids* 1, 4, 2 again has three steps since it has three edges.

Example 3

The following query matches paths between universities ABC and XYZ such that paths consist of an incoming student_of edge, followed by one or two friends edges, followed by an outgoing student_of edge. The query returns one row per vertex and for each row it returns the match number, the element number, the type of the vertex (either person or university), as well as the name of the university or the person.

```
SELECT *
FROM GRAPH_TABLE ( students_graph
       MATCH (u1 IS university)
               <-[IS student_of]- (p1 IS person)
               -[IS friends]-{1,2} (p2 IS person)
               -[IS student_of]-> (u2 IS university)
       WHERE u1.name = 'ABC' AND u2.name = 'XYZ'
```

```
        ONE ROW PER VERTEX (v)
        COLUMNS (MATCHNUM() AS matchnum,
                 ELEMENT_NUMBER(v) AS element_number,
                 CASE WHEN v.person_id IS NOT NULL
                   THEN 'person'
                   ELSE 'university'
                   END AS label,
                 v.name))
ORDER BY matchnum, element_number;
```

The results are:

```
MATCHNUM ELEMENT_NUMBER        LABEL       NAME
---------- -------------- ---------- ----------
         1              1 university         ABC
         1              3 person             John
         1              5 person             Mary
         1              7 university         XYZ
         2              1 university         ABC
         2              3 person             Bob
         2              5 person             John
         2              7 person             Mary
         2              9 university         XYZ
         3              1 university         ABC
         3              3 person             Bob
         3              5 person             Mary
         3              7 university         XYZ
         4              1 university         ABC
         4              3 person             John
         4              5 person             Mary
         4              7 person             Alice
         4              9 university         XYZ
         6              1 university         ABC
         6              3 person             John
         6              5 person             Bob
         6              7 person             Mary
         6              9 university         XYZ
         8              1 university         ABC
         8              3 person             Bob
         8              5 person             Mary
         8              7 person             Alice
         8              9 university         XYZ
```

Note that a total of 6 paths were matched with match numbers `1`, `2`, `3`, `4`, `6` and `8`. Each path has university `ABC` as the first vertex and university `XYZ` as the last vertex. Furthermore, paths with match numbers `1` and `3` contain two person vertices while the other paths (match numbers `2`, `4`, `6` and `8`) contain three person vertices.

Value Expressions for GRAPH_TABLE

Purpose

Value expressions in `WHERE` and `COLUMNS` clauses inside `GRAPH_TABLE` inherit all the functionality supported in value expressions outside of `GRAPH_TABLE`. Additionally, inside `GRAPH_TABLE`, the following value expressions are available:

- Property Reference
- Vertex and Edge ID Functions

- Vertex and Edge Equal Predicates
- SOURCE and DESTINATION Predicates
- Aggregation in GRAPH_TABLE
- JSON Object Access Expressions for Property Graphs

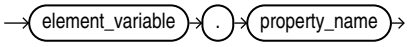
Property Reference

Purpose

Property references allow for accessing property values of vertices and edges.

Syntax

***property_reference*::=**



***property_name*::=**

Semantics

Syntactically, a property access is an element variable followed by a dot (.) and the name of the property. A property name is an identifier and may thus be either double quoted or unquoted.

The label expression specified for an element pattern determines which properties can be referenced:

- If no label expression is specified, then depending on the type of element variable, either all vertex properties or all edge properties in the graph can be referenced.
- Otherwise, if a label expression is specified, then the set of properties that can be referenced is the union of the properties of labels belonging to vertex (or edge) tables that have at least one label that satisfies the label expression.

If multiple labels satisfy the label expression but they define the same property but of a different data type, then such properties may only be referenced if the data types are union compatible. The resulting value will then have the union compatible data type.

If multiple labels satisfy the label expression while some labels have a particular property that other labels do not, then such properties can still be referenced. The property reference will result in null values for any vertex or edge that does not have the property.

Furthermore, if the element variable is not bound to a graph element, then the result is the null value. Note that the only way an element variable is optionally bound is when the element variable is an iterator variable declared in `ONE ROW PER STEP`. Specifically, the edge variable and the second vertex variable declared in `ONE ROW PER STEP` will not be bound to a graph element when the path pattern matches an empty path, for example because a quantifier iterated zero times.

Examples

Example 1

The following query lists the date of birth of all persons and universities in the graph:

```
SELECT GT.name, GT.birthday
FROM GRAPH_TABLE ( students_graph
  MATCH (p IS person|university)
  COLUMNS (p.name, p.dob AS birthday)
) GT
ORDER BY GT.birthday, GT.name;
```

Note that since only persons `John`, `Bob`, `Mary`, `Alice` have dates of birth while universities (`ABC` and `XYZ`) do not, null values are returned for universities. These appear as empty strings in the output:

```
NAME       BIRTHDAY
---------- ---------
John       13-JUN-63
Bob        11-MAR-66
Mary       25-SEP-82
Alice      01-FEB-87
ABC
XYZ
```

Example 2

The following query matches all `PERSON` vertices and returns their `NAME` and `HEIGHT`:

```
SELECT *
FROM GRAPH_TABLE ( students_graph
  MATCH (n IS person)
  COLUMNS ( n.name, n.height )
)
ORDER BY height;
```

The result is:

```
NAME       HEIGHT
---------- -------
Mary          1.65
Alice         1.7
Bob           1.75
John          1.8
```

Here, even though label `PERSON` does not have property `HEIGHT`, the property can still be referenced because vertex table `PERSONS` has labels `PERSON` and `PERSON_HT` and since label `PERSON` matches the label expression, the set of properties that can be referenced is the union of the properties of labels `PERSON` and `PERSON_HT`, which includes the property `HEIGHT` of label `PERSON_HT`.

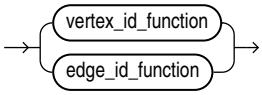
Vertex and Edge ID Functions

Purpose

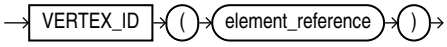
Vertex and edge ID functions allow for obtaining unique identifiers for graph elements.

Syntax

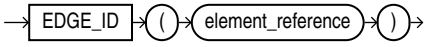
***element_id_function*::=**



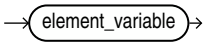
***vertex_id_function*::=**



***edge_id_function*::=**



***element_reference*::=**



Semantics

Syntactically, the `VERTEX_ID` and `EDGE_ID` functions take an element reference, which should be a vertex reference in case of `VERTEX_ID` and an edge reference in case of `EDGE_ID`. The two functions generate identifiers for graph elements that are globally unique within a database.

Content-wise, vertex and edge identifiers are JSON object that contains the following information:

- Owner of the graph that the vertex or edge is part of.
- Name of the graph that the vertex or edge is part of.
- Element table that the vertex or edge is defined in.
- Key value of the vertex or edge.

If the referenced element variable is not bound to a graph element, then the functions return the null value.

Examples

Example 1

The following query lists the vertex identifiers of friends of Mary:

```
SELECT CAST(p2_id AS VARCHAR2(200)) AS p2_id
FROM GRAPH_TABLE ( students_graph
  MATCH (p1 IS person) -[e1 IS friends]- (p2 IS person)
  WHERE p1.name = 'Mary'
  COLUMNS (vertex_id(p2) AS p2_id)
)
ORDER BY p2_id;
```

The result is:

```
P2_ID
--------------------------------------------------------------------------------------------------
---------------
{"GRAPH_OWNER":"SCOTT","GRAPH_NAME":"STUDENTS_GRAPH","ELEM_TABLE":"PERSONS","KEY_VALUE":
{"PERSON_ID":1}}
{"GRAPH_OWNER":"SCOTT","GRAPH_NAME":"STUDENTS_GRAPH","ELEM_TABLE":"PERSONS","KEY_VALUE":
{"PERSON_ID":3}}
{"GRAPH_OWNER":"SCOTT","GRAPH_NAME":"STUDENTS_GRAPH","ELEM_TABLE":"PERSONS","KEY_VALUE":
{"PERSON_ID":4}}
```

Example 2

The following query uses JSON dot-notation syntax to obtain a set of JSON objects representing the vertex keys of vertices corresponding to friends of Mary:

```
SELECT GT.p2_id.KEY_VALUE
FROM GRAPH_TABLE ( students_graph
  MATCH (p1 IS person) -[e1 IS friends]- (p2 IS person)
  WHERE p1.name = 'Mary'
  COLUMNS (vertex_id(p2) AS p2_id)
) GT
ORDER BY key_value;
```

The result is:

```
KEY_VALUE
----------------------------------------
{"PERSON_ID":1}
{"PERSON_ID":3}
{"PERSON_ID":4}
```

Example 3

The following query uses the `JSON_VALUE` function to obtain all the element table names of edges in the graph:

```
SELECT DISTINCT json_value(e_id, '$.ELEM_TABLE') AS elem_table
FROM GRAPH_TABLE ( students_graph
  MATCH -[e]-
  COLUMNS (edge_id(e) AS e_id)
)
ORDER BY elem_table;
```

The result is:

```
ELEM_TABLE
----------------------------------------
FRIENDS
STUDENT_OF
```

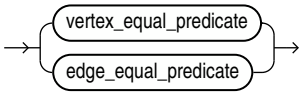
Vertex and Edge Equal Predicates

Purpose

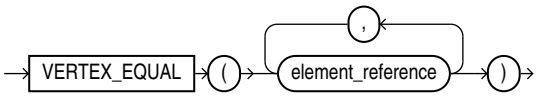
The vertex and edge equal predicates allow for specifying that two vertex variables (or two edge variables) should or should not bind to the same vertex (or edge).

Syntax

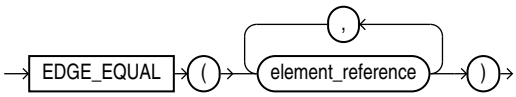
***element_equal_predicate*::=**



***vertex_equal_predicate*::=**



***edge_equal_predicate*::=**



Semantics

If at least one of the referenced element variables is not bound to a graph element, then the predicates evaluate to the null value. Otherwise, they evaluate to `TRUE` or `FALSE`.

Examples

Example 1

The following query finds friends of friends of Mary. Here, the *`vertex_equal predicate`* is used to make sure Mary herself is not included in the result.

```
SELECT name
FROM GRAPH_TABLE ( students_graph
  MATCH (p IS person)
          -[IS friends]- (friend IS person)
            -[IS friends]- (friend_of_friend IS person)
  WHERE p.name = 'Mary' AND NOT vertex_equal(p, friend_of_friend)
  COLUMNS (friend_of_friend.name)
```

```
)
ORDER BY name;
```

The result is:

```
NAME
----------
Bob
John
```

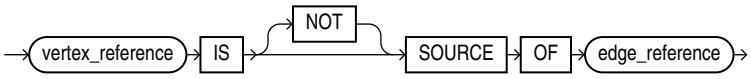
SOURCE and DESTINATION Predicates

Purpose

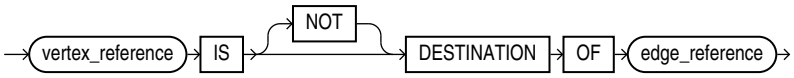
The `SOURCE` and `DESTINATION` predicates allow for testing if a vertex is the source or the destination of an edge. They are useful, for example, for determining the direction of edges that are matched via any-directed edge patterns.

Syntax

***source_predicate*::=**



***destination_predicate*::=**



Semantics

The `SOURCE` predicate takes a vertex and an edge as input and returns `TRUE` or `FALSE` depending on whether the vertex is (not) the source of the edge.

The `DESTINATION` predicate also takes a vertex and an edge as input and returns `TRUE` or `FALSE` depending on whether the vertex is (not) the destination of the edge.

If at least one of the referenced element variables is not bound to a graph element, then the predicates evaluate to the null value. Otherwise, they evaluate to `TRUE` or `FALSE`.

Examples

Example 1

The following query matches `FRIENDS` edges that are either incoming or outgoing from Mary. For each edge, it return the `NAME` property for the source of the edge as well as the `NAME` property of the destination of the edge.

```
SELECT *
FROM GRAPH_TABLE ( students_graph
       MATCH (p1 IS person) -[e IS friends]- (p2 IS person)
       WHERE p1.name = 'Mary'
```

```
       COLUMNS (e.friendship_id,
                e.meeting_date,
                CASE WHEN p1 IS SOURCE OF e THEN p1.name ELSE p2.name END AS from_person,
                CASE WHEN p1 IS DESTINATION OF e THEN p1.name ELSE p2.name END AS
to_person))
ORDER BY friendship_id;

FRIENDSHIP_ID MEETING_DATE FROM_PERSON TO_PERSON
------------- ------------ ----------- ---------
            2 19-SEP-00    Mary        Alice
            3 19-SEP-00    Mary        John
            4 10-JUL-01    Bob         Mary
```

Example 2

The following query find friends of friends of John such that the two `FRIENDS` edges are either both incoming or outgoing.

```
SELECT *
FROM GRAPH_TABLE ( students_graph
       MATCH (p1 IS person) -[e1 IS friends]- (p2 IS person)
               -[e2 IS friends]- (p3 IS person)
       WHERE p1.name = 'John'
         AND ((p1 IS SOURCE OF e1 AND p2 IS SOURCE OF e2) OR
              (p1 IS DESTINATION OF e1 AND p2 IS DESTINATION OF e2))
       COLUMNS (p1.name AS person_1,
                CASE WHEN p1 IS SOURCE OF e1
                  THEN 'Outgoing' ELSE 'Incoming'
                  END AS e1_direction,
                p2.name AS person_2,
                CASE WHEN p2 IS SOURCE OF e2
                  THEN 'Outgoing' ELSE 'Incoming'
                  END AS e2_direction,
                p3.name AS person_3))
ORDER BY 1, 2, 3;

PERSON_1   E1_DIRECTION PERSON_2   E2_DIRECTION PERSON_3
---------- ------------ ---------- ------------ ----------
John       Incoming     Mary       Incoming     Bob
John       Outgoing     Bob        Outgoing     Mary
```

Notice how the path from John via Mary to Alice is not part of the result since it has an incoming edge followed by an outgoing edge and thus not two edges in the same direction.

Aggregation in GRAPH_TABLE

Purpose

Aggregations in `GRAPH_TABLE` are used to compute one or more values for a set of vertices or edges in a variable-length path. This is done using the same Aggregate Functions that are also available for non-graph queries.

Syntax

All the aggregate functions that are available for non-graph queries are also available for graph queries. See Aggregate Functions for the syntax of these functions.

Aggregate functions can be used in `WHERE` and `COLUMNS` clauses in `GRAPH_TABLE`, with the restriction that `WHERE` clauses within quantified patterns may not contain aggregate functions.

Syntactically, the value expressions in the aggregations must contain references to vertices and edges in the graph pattern, rather than to columns of tables like in case of regular (non-graph) SQL queries.

Semantics

See Aggregate Functions for the semantics of aggregate functions.

The arguments of the aggregate function together must reference exactly one group variable. In addition, they can reference any number of singleton variables. Note that an element variable is said to have group degree of reference when the variable is declared in a quantified path pattern while the reference occurs outside the quantified path pattern. On the other hand, if the reference does not cross a quantifier then the reference has singleton degree of reference. Singleton variables may be element pattern variables declared in the graph pattern or iterator variables declared in the Rows Clause. Also see Element Variable for more details on the contextual interpretation of graph element references.

The order in which values are aggregated in case of `LISTAGG`, `JSON_ARRAYAGG` and `XMLAGG` is non-deterministic unless an `ORDER BY` clause is specified. For example: `LISTAGG(edge1.property1 ORDER BY edge1.property1))`. There is currently no way to explicitly order by path order in such a way that elements are ordered in the same order as the vertices or edges in the path. However, when omitting the `ORDER BY` clause, the current implementation nevertheless implicitly orders by path order, but it should not be relied upon as this behavior may change over time.

Restrictions

- Only `WHERE` clauses that are not within a quantified pattern may contain aggregations. For example, the graph pattern `WHERE` clause as well as non-quantified element pattern `WHERE` clauses may contain aggregations, while parenthesized path pattern `WHERE` clauses may not contain aggregations since parenthesized path patterns currently have a restriction that they must always be quantified.
- The arguments of an aggregate function in `GRAPH_TABLE` together must reference exactly one group variable. In addition, they may reference any number of singleton variables. For example, `MATCH -[e1]-> WHERE SUM(e1.prop) > 10` is not allowed since variable `e1` has singleton degree of reference within the `SUM` aggregate, while `MATCH -[e2]->{1,10} WHERE SUM(e2.prop) > 10` and `MATCH -[e3]->{1,1} WHERE SUM(e3.prop) > 10` are allowed since variables `e2` and `e3` have group degree of reference within the `SUM` aggregates.
- Variable references must be inside property references, vertex or edge ID functions, or JSON dot-notation expressions. For example, `vertex_equal`, `edge_equal`, `IS SOURCE OF` and `IS DESTINATION OF` cannot be used in aggregate functions. For example, `COUNT(edge1)` is not allowed but `COUNT(edge_id(edge1))` and `COUNT(edge1.some_property))` are allowed.
- The arguments of an aggregate function in `GRAPH_TABLE` cannot reference anything other than a vertex or edge declared within the graph pattern of the `GRAPH_TABLE`. For example, it is not possible to reference a column that is passed from an outer query.
- In case of `LISTAGG`, `JSON_ARRAYAGG` and `XMLAGG` there is no way to specify that the order of elements in the result should be in the order of the vertices or edges in the path, although the current implement nevertheless implicitly orders by path order.

Examples

Example 1

The following query finds all paths that have a length between 2 and 5 edges (`{2,5}`), starting from a person named Alice and following both incoming and outgoing edges labeled friends. Edges along paths should not be traversed twice (`COUNT(edge_id(e) = COUNT(DISTINCT edge_id(e))`). The query returns all friendship IDs along paths as well as the length of each path.

```
SELECT *
FROM GRAPH_TABLE ( students_graph
       MATCH (p IS person) -[e IS friends]-{2,5} (friend IS person)
       WHERE p.name = 'Alice' AND
             COUNT(edge_id(e)) = COUNT(DISTINCT edge_id(e))
       COLUMNS (LISTAGG(e.friendship_id, ', ') AS friendship_ids,
                COUNT(edge_id(e)) AS path_length))
ORDER BY path_length, friendship_ids;
```

Note that in the element pattern `WHERE` clause of the query above, p.name references a property of a single edge, while `edge_id(e)` within the `COUNT` aggregates accesses a list of element IDs since the edge variable e is enclosed by the quantifier `{2,5}`. Similarly, the two property references in the `COLUMNS` clause access a list of property values and edge ID values.

The result is:

```
FRIENDSHIP_IDS    PATH_LENGTH
----------------- -----------
2, 3              2
2, 4              2
2, 3, 1           3
2, 4, 1           3
2, 3, 1, 4        4
2, 4, 1, 3        4
```

Example 2

The following query finds all paths between university `ABC` and university `XYZ` such that paths have a length of up to 3 edges (`{,3}`). For each path, a JSON array is returned such that the array contains the `friendship_id` value for edges labeled friends, and the subject value for edges labeled `student_of`. Note that the `friendship_id` property is cast to `VARCHAR(100)` to make it type-compatible with the subject property.

```
SELECT *
FROM GRAPH_TABLE ( students_graph
       MATCH (u1 IS university) -[e]-{,3} (u2 IS university)
       WHERE u1.name = 'ABC' AND u2.name = 'XYZ'
       COLUMNS (JSON_ARRAYAGG(CASE WHEN e.subject IS NOT NULL THEN e.subject
                                   ELSE CAST(e.friendship_id AS VARCHAR(100)) END) AS path))
ORDER BY path;
The result is:
PATH
----------------------
["Arts","3","Math"]
["Music","4","Math"]
```

Example 3

Example 3 The following query finds all paths that have a length between 2 and 3 edges (`{2,3}`), starting from a person named John and following only outgoing edges labeled friends and vertices labeled person. Vertices along paths should not have the same person_id as John (`WHERE p.person_id <> friend.person_id`).

```
SELECT *
FROM GRAPH_TABLE ( students_graph
       MATCH (p IS person) ( -[e IS friends]-> (friend IS person)
                             WHERE p.person_id <> friend.person_id){2,3}
       WHERE p.name = 'John'
       COLUMNS (COUNT(edge_id(e)) AS path_length,
                LISTAGG(friend.name, ', ') AS names,
                LISTAGG(e.meeting_date, ', ') AS meeting_dates ))
ORDER BY path_length;
```

Above, the `COLUMNS` clause contains three aggregates, the first to compute the length of each path, the second to create a comma-separated list of person names along paths, and the third to create a comma-separate list of meeting dates along paths.

The result of the query is:

```
PATH_LENGTH NAMES
MEETING_DATES
----------- --------------------
-----------------------------------
          2 Bob, Mary           01-SEP-00, 10-
JUL-01
          3 Bob, Mary, Alice    01-SEP-00, 10-JUL-01, 19-SEP-00
```

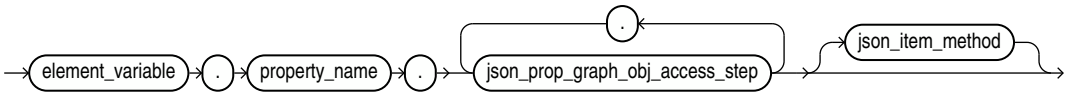
JSON Object Access Expressions for Property Graphs

Purpose

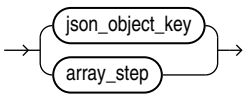
JSON dot notation for property graphs allows for easy access to JSON data exposed as vertex or edge property values. It provides a simple syntax for common use cases, while SQL/JSON functions json_value and json_query can be used for more complex queries against property graphs containing JSON data.

Syntax

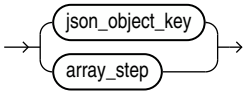
***json_property_graph_object_access_expr*::=**



***json_prop_graph_obj_access_step*::=**



***array_step*::=**



Semantics

JSON dot notation for property graphs supports the same functionality as JSON Dot Notation for columns of JSON data. Please refer to JSON Object Access Expressions.

Examples

The following example creates a new graph on top of the `persons` table from the sample data. This graph will have a vertex property `person_data` of type JSON since the persons table has `person_data` column of type JSON. Then, a `GRAPH_TABLE` query that uses JSON dot notation is issued against this graph to obtain the role of all persons in the HR department.

```
CREATE PROPERTY GRAPH persons_graph VERTEX TABLES ( persons );


SELECT *
FROM GRAPH_TABLE ( persons_graph
       MATCH (n)
       WHERE n.person_data.department = 'HR'
       COLUMNS (n.name, n.person_data.role.string() AS role)
     );

The output of above SELECT query is:

NAME            ROLE
--------------- ---------------
Mary            HR Manager
Alice           HR Assistant
```

Note how item method `string()` is used in the `COLUMNS` clause to return a `VARCHAR2(4000)`. Without the item method it would have returned a JSON string and the result would have been double quoted.

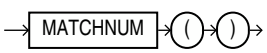
> **See Also:**
>
> *Simple Dot Notation Access JSON Data* of the *JSON Developer's Guide*.

MATCHNUM

Purpose

The `MATCHNUM` function returns a number that uniquely identifies a match in a set of matches.

Syntax



Semantics

The `MATCHNUM` function returns a number that uniquely identifies a match in a set of matches. The numbers are not necessarily consecutive, and gaps may appear for example when matches were filtered out. Rows returned from `GRAPH_TABLE` have unique match numbers

unless `ONE ROW PER VERTEX` or `ONE ROW PER STEP` is specified, in which case the same match number is returned for different iterations within a match.

Restricitons

`MATCHNUM` can only be used in the `COLUMNS` clause.

Examples

Example 1

The following query matches all person vertices and for each match returns a unique match number as well as the name of the person.

```
SELECT *
FROM GRAPH_TABLE ( students_graph
       MATCH (p IS person)
       COLUMNS (MATCHNUM() AS matchnum,
                p.name))
ORDER BY matchnum;
```

The results are:

```
  MATCHNUM NAME
---------- -------
         1 John
         2 Mary
         3 Bob
         4 Alice
```

Example 2

The following query finds paths connecting John and Mary either directly or indirectly via a common friend. For each match, the query returns one row per vertex, which means one row per person along the friendship path. Each result contains a match number, the element number of the person vertex, and the name of the person.

```
SELECT *
FROM GRAPH_TABLE ( students_graph
       MATCH (p1 IS person) -[IS friends]-{1,2} (p2 IS person)
       WHERE p1.name = 'John' AND p2.name = 'Mary'
       ONE ROW PER VERTEX (v)
       COLUMNS (MATCHNUM() AS matchnum,
                ELEMENT_NUMBER(v) AS element_number,
                v.name))
ORDER BY matchnum, element_number;
```

The results are:

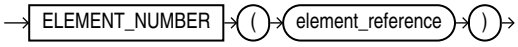
```
MATCHNUM   ELEMENT_NUMBER NAME
---------  -------------- ----------
        1              1  John
        1              3  Mary
        2              1  John
        2              3  Bob
        2              5  Mary
```

ELEMENT_NUMBER

Purpose

The `ELEMENT_NUMBER` function returns the sequential element number of the graph element that an iterator variable currently binds to.

Syntax



Semantics

The `ELEMENT_NUMBER` function can be used in a `COLUMNS` clause if `ONE ROW PER VERTEX` or `ONE ROW PER STEP` is specified. The function references an iterator variable and returns the sequential element number that the iterator variable currently binds to. Since paths always start with a vertex and alternate between vertices and edges, the first element is a vertex with element number 1, the second element is an edge with element number 2, the third element is a vertex with element number 3, etc. Vertices thus always have odd element numbers while edges have even element numbers. If a path is empty and thus only has a single vertex and no edges, and `ONE ROW PER STEP` is specified, then `ELEMENT_NUMBER` returns NULL when the iterator edge variable or the second iterator vertex variable is referenced. Note that empty paths result in single steps in which only the first iterator (vertex) variable is bound.

Restricitons

- `ELEMENT_NUMBER` can only be used in the `COLUMNS` clause.
- `ELEMENT_NUMBER` can only be used if `ONE ROW PER VERTEX` or `ONE ROW PER STEP` is specified.
- `ELEMENT_NUMBER` cannot reference any other type of variable than an iterator variable.

Examples

Example 1

The following query finds paths connecting John and Mary either directly or indirectly via a common friend. For each match, the query returns one row per step. Each result contains a match number, the element number of the friends edge in the step, the `friendship_id` and the names of the two persons in the step.

```
SELECT *
FROM GRAPH_TABLE ( students_graph
       MATCH (p1 IS person) -[IS friends]-{1,2} (p2 IS person)
       WHERE p1.name = 'John' AND p2.name = 'Mary'
       ONE ROW PER STEP (v1, e, v2)
       COLUMNS (MATCHNUM() AS matchnum,
                ELEMENT_NUMBER(e) AS element_number,
                v1.name AS name1,
                e.friendship_id,
                v2.name AS name2))
ORDER BY matchnum, element_number;
```

The results are:

```
The results are:
  MATCHNUM ELEMENT_NUMBER NAME1      FRIENDSHIP_ID NAME2
---------- -------------- ---------- ------------- ----------
         1              2 John                   3 Mary
         2              2 John                   1 Bob
         2              4 Bob                    4 Mary
```

Example 2

The following query finds all people connected to John via 0 or 1 friends edges. For each match, the query returns one row per step. Each result contains a match number, the element number of the friends edge in the step, the `friendship_id` and the names of the two persons in the step.

```
SELECT *
FROM GRAPH_TABLE ( students_graph
       MATCH (p1 IS person) -[IS friends]-{0,1} (p2 IS person)
       WHERE p1.name = 'John'
       ONE ROW PER STEP (v1, e, v2)
       COLUMNS (MATCHNUM() AS matchnum,
                ELEMENT_NUMBER(e) AS element_number,
                v1.name AS name1,
                e.friendship_id,
                v2.name AS name2))
ORDER BY matchnum, element_number;
```

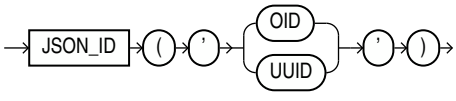
The results are:

```
The results are:
  MATCHNUM ELEMENT_NUMBER   NAME1      FRIENDSHIP_ID   NAME2
---------- --------------   ---------- -------------   ----------
         1                       John
         2              2         John             3        Mary
         4              2         John             1        Bob
```

Here, three paths were matched. The path with match number 1 has one vertex and zero edges. Thus, there is a single step in which iterator vertex variable v1 is bound but iterator edge variable e and iterator vertex variable v2 are not bound, leading to the NULL values in the ELEMENT_NUMBER, FRIENDSHIP_ID and NAME2 columns. The other two paths (with match numbers 2 and 4) also have a single step but since these paths do contain an edge as well as a second vertex, all three iterator variables are bound, and no NULL values are returned.

JSON_ID Operator

Syntax



Purpose

`JSON_ID` takes a single argument, one of `'OID'` or `'UUID'` to create a value for a document-identifier field that you provide.

`JSON_ID` returns a value of SQL type `RAW` that is globally unique. The value returned is determined by the argument that you provide. With string `'OID'`, a 12-byte `RAW` value is returned; with string `'UUID'`, a 16-byte `RAW` value is returned.

> **See Also:**
>
> JSON Collections of the *JSON Developer's Guide*.