# 3
# Pseudocolumns

A **pseudocolumn** behaves like a table column, but is not actually stored in the table. You can select from pseudocolumns, but you cannot insert, update, or delete their values. A pseudocolumn is also similar to a function without arguments (refer to Functions). However, functions without arguments typically return the same value for every row in the result set, whereas pseudocolumns typically return a different value for each row.

This chapter contains the following sections:



## Hierarchical Query Pseudocolumns

The hierarchical query pseudocolumns are valid only in hierarchical queries. The hierarchical query pseudocolumns are:



To define a hierarchical relationship in a query, you must use the `CONNECT BY` clause.

### CONNECT_BY_ISCYCLE Pseudocolumn

The `CONNECT_BY_ISCYCLE` pseudocolumn returns 1 if the current row has a child which is also its ancestor. Otherwise it returns 0.

You can specify `CONNECT_BY_ISCYCLE` only if you have specified the `NOCYCLE` parameter of the `CONNECT BY` clause. `NOCYCLE` enables Oracle to return the results of a query that would otherwise fail because of a `CONNECT BY` loop in the data.

> **See Also:**
>
> Hierarchical Queries for more information about the `NOCYCLE` parameter and Hierarchical Query Examples for an example that uses the `CONNECT_BY_ISCYCLE` pseudocolumn

# CONNECT_BY_ISLEAF Pseudocolumn

The `CONNECT_BY_ISLEAF` pseudocolumn returns 1 if the current row is a leaf of the tree defined by the `CONNECT BY` condition. Otherwise it returns 0. This information indicates whether a given row can be further expanded to show more of the hierarchy.

### CONNECT_BY_ISLEAF Example

The following example shows the first three levels of the `hr.employees` table, indicating for each row whether it is a leaf row (indicated by 1 in the `IsLeaf` column) or whether it has child rows (indicated by 0 in the `IsLeaf` column):

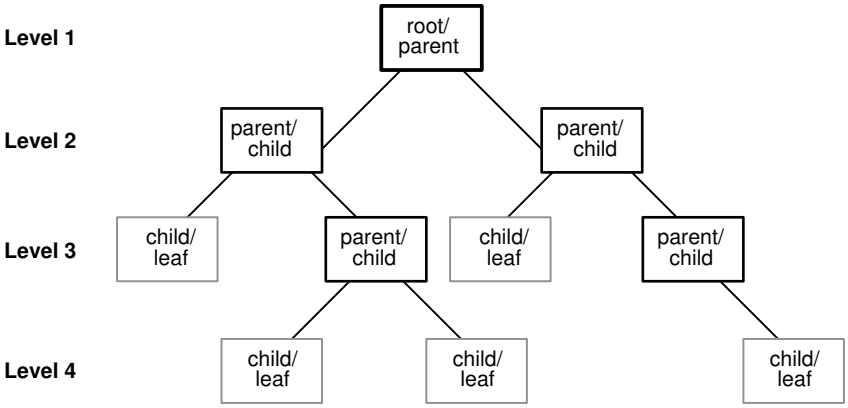
```
SELECT last_name "Employee", CONNECT_BY_ISLEAF "IsLeaf",
       LEVEL, SYS_CONNECT_BY_PATH(last_name, '/') "Path"
  FROM employees
  WHERE LEVEL <= 3 AND department_id = 80
  START WITH employee_id = 100
  CONNECT BY PRIOR employee_id = manager_id AND LEVEL <= 4
  ORDER BY "Employee", "IsLeaf";

Employee                       IsLeaf      LEVEL Path
------------------------- ---------- ---------- -------------------------
Abel                                1          3 /King/Zlotkey/Abel
Ande                                1          3 /King/Errazuriz/Ande
Banda                               1          3 /King/Errazuriz/Banda
Bates                               1          3 /King/Cambrault/Bates
Bernstein                           1          3 /King/Russell/Bernstein
Bloom                               1          3 /King/Cambrault/Bloom
Cambrault                           0          2 /King/Cambrault
Cambrault                           1          3 /King/Russell/Cambrault
Doran                               1          3 /King/Partners/Doran
Errazuriz                           0          2 /King/Errazuriz
Fox                                 1          3 /King/Cambrault/Fox
. . .
```

> **See Also:**
>
> Hierarchical Queries and SYS_CONNECT_BY_PATH

# LEVEL Pseudocolumn

For each row returned by a hierarchical query, the `LEVEL` pseudocolumn returns 1 for a root row, 2 for a child of a root, and so on. A **root row** is the highest row within an inverted tree. A **child row** is any nonroot row. A **parent row** is any row that has children. A **leaf row** is any row without children. Figure 3-1 shows the nodes of an inverted tree with their `LEVEL` values.

**Figure 3-1 Hierarchical Tree**




> **See Also:**
>
> Hierarchical Queries for information on hierarchical queries in general and IN Condition for restrictions on using the `LEVEL` pseudocolumn

# Sequence Pseudocolumns

A **sequence** is a schema object that can generate unique sequential values. These values are often used for primary and unique keys. You can refer to sequence values in SQL statements with these pseudocolumns:

- `CURRVAL`: Returns the current value of a sequence
- `NEXTVAL`: Increments the sequence and returns the next value

You must qualify `CURRVAL` and `NEXTVAL` with the name of the sequence:

```
sequence.CURRVAL
sequence.NEXTVAL
```

To refer to the current or next value of a sequence in the schema of another user, you must have been granted either `SELECT` object privilege on the sequence or `SELECT ANY SEQUENCE` system privilege, and you must qualify the sequence with the schema containing it:

```
schema.sequence.CURRVAL
schema.sequence.NEXTVAL
```

To refer to the value of a sequence on a remote database, you must qualify the sequence with a complete or partial name of a database link:

```
schema.sequence.CURRVAL@dblink
schema.sequence.NEXTVAL@dblink
```

A sequence can be accessed by many users concurrently with no waiting or locking.

> **See Also:**
>
> References to Objects in Remote Databases for more information on referring to database links

## Where to Use Sequence Values

You can use CURRVAL and NEXTVAL in the following locations:

- The select list of a SELECT statement that is not contained in a subquery, materialized view, or view
- The select list of a subquery in an INSERT statement
- The VALUES clause of an INSERT statement
- The SET clause of an UPDATE statement

**Restrictions on Sequence Values**

You cannot use CURRVAL and NEXTVAL in the following constructs:

- A subquery in a DELETE, SELECT, or UPDATE statement
- A query of a view or of a materialized view
- A SELECT statement with the DISTINCT operator
- A SELECT statement with a GROUP BY clause or ORDER BY clause
- A SELECT statement that is combined with another SELECT statement with the UNION, INTERSECT, or MINUS set operator
- The WHERE clause of a SELECT statement
- The condition of a CHECK constraint

Within a single SQL statement that uses CURRVAL or NEXTVAL, all referenced LONG columns, updated tables, and locked tables must be located on the same database.

## How to Use Sequence Values

When you create a sequence, you can define its initial value and the increment between its values. The first reference to NEXTVAL returns the initial value of the sequence. Subsequent references to NEXTVAL increment the sequence value by the defined increment and return the new value. Any reference to CURRVAL always returns the current value of the sequence, which is the value returned by the last reference to NEXTVAL.

Before you use CURRVAL for a sequence in your session, you must first initialize the sequence with NEXTVAL. Refer to CREATE SEQUENCE for information on sequences.

Within a single SQL statement containing a reference to NEXTVAL, Oracle increments the sequence once:

- For each row returned by the outer query block of a SELECT statement. Such a query block can appear in the following places:
  - A top-level SELECT statement

  - An INSERT ... SELECT statement (either single-table or multitable). For a multitable insert, the reference to NEXTVAL must appear in the VALUES clause, and the sequence is updated once for each row returned by the subquery, even though NEXTVAL may be referenced in multiple branches of the multitable insert.
  - A CREATE TABLE ... AS SELECT statement
  - A CREATE MATERIALIZED VIEW ... AS SELECT statement
- For each row updated in an UPDATE statement
- For each INSERT statement containing a VALUES clause
- For each INSERT ... [ALL | FIRST] statement (multitable insert). A multitable insert is considered a single SQL statement. Therefore, a reference to the NEXTVAL of a sequence will increase the sequence only once for each input record coming from the SELECT portion of the statement. If NEXTVAL is specified more than once in any part of the INSERT ... [ALL | FIRST ] statement, then the value will be the same for all insert branches, regardless of how often a given record might be inserted.
- For each row merged by a MERGE statement. The reference to NEXTVAL can appear in the *merge_insert_clause* or the *merge_update_clause* or both. The NEXTVALUE value is incremented for each row updated and for each row inserted, even if the sequence number is not actually used in the update or insert operation. If NEXTVAL is specified more than once in any of these locations, then the sequence is incremented once for each row and returns the same value for all occurrences of NEXTVAL for that row.
- For each input row in a multitable INSERT ALL statement. NEXTVAL is incremented once for each row returned by the subquery, regardless of how many occurrences of the *insert_into_clause* map to each row.

If any of these locations contains more than one reference to NEXTVAL, then Oracle increments the sequence once and returns the same value for all occurrences of NEXTVAL.

If any of these locations contains references to both CURRVAL and NEXTVAL, then Oracle increments the sequence and returns the same value for both CURRVAL and NEXTVAL.

### Finding the next value of a sequence: Example

This example selects the next value of the employee sequence in the sample schema hr:

```
SELECT employees_seq.nextval
  FROM DUAL;
```

### Inserting sequence values into a table: Example

This example increments the employee sequence and uses its value for a new employee inserted into the sample table hr.employees:

```
INSERT INTO employees
  VALUES (employees_seq.nextval, 'John', 'Doe', 'jdoe', '555-1212',
          TO_DATE(SYSDATE), 'PU_CLERK', 2500, null, null, 30);
```

### Reusing the current value of a sequence: Example

This example adds a new order with the next order number to the master order table. It then adds suborders with this number to the detail order table:

```
INSERT INTO orders (order_id, order_date, customer_id)
  VALUES (orders_seq.nextval, TO_DATE(SYSDATE), 106);

INSERT INTO order_items (order_id, line_item_id, product_id)
```

```
  VALUES (orders_seq.currval, 1, 2359);

INSERT INTO order_items (order_id, line_item_id, product_id)
  VALUES (orders_seq.currval, 2, 3290);

INSERT INTO order_items (order_id, line_item_id, product_id)
  VALUES (orders_seq.currval, 3, 2381);
```

# Version Query Pseudocolumns

The version query pseudocolumns are valid only in Oracle Flashback Version Query, which is a form of Oracle Flashback Query. The version query pseudocolumns are:

- `VERSIONS_STARTSCN` and `VERSIONS_STARTTIME`: Starting System Change Number (SCN) or `TIMESTAMP` when the row version was created. This pseudocolumn identifies the time when the data first had the values reflected in the row version. Use this pseudocolumn to identify the past target time for Oracle Flashback Table or Oracle Flashback Query. If this pseudocolumn is `NULL`, then the row version was created before start.
- `VERSIONS_ENDSCN` and `VERSIONS_ENDTIME`: SCN or `TIMESTAMP` when the row version expired. If the pseudocolumn is `NULL`, then either the row version was current at the time of the query or the row corresponds to a `DELETE` operation.
- `VERSIONS_XID`: Identifier (a `RAW` number) of the transaction that created the row version.
- `VERSIONS_OPERATION`: Operation performed by the transaction: `I` for insertion, `D` for deletion, or `U` for update. The version is that of the row that was inserted, deleted, or updated; that is, the row after an `INSERT` operation, the row before a `DELETE` operation, or the row affected by an `UPDATE` operation.

  For user updates of an index key, Oracle Flashback Version Query might treat an `UPDATE` operation as two operations, `DELETE` plus `INSERT`, represented as two version rows with a `D` followed by an `I` `VERSIONS_OPERATION`.

> **See Also:**
>
> - *flashback_query_clause* for more information on version queries
> - *Oracle Database Development Guide* for more information on using Oracle Flashback Version Query
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules for values of the `VERSIONS_OPERATION` pseudocolumn

# COLUMN_VALUE Pseudocolumn

When you refer to an `XMLTable` construct without the `COLUMNS` clause, or when you use the `TABLE` collection expression to refer to a scalar nested table type, the database returns a virtual table with a single column. This name of this pseudocolumn is `COLUMN_VALUE`.

In the context of `XMLTable`, the value returned is of data type `XMLType`. For example, the following two statements are equivalent, and the output for both shows `COLUMN_VALUE` as the name of the column being returned:

```
SELECT *
  FROM XMLTABLE('<a>123</a>');

COLUMN_VALUE
----------------------------------------
<a>123</a>

SELECT COLUMN_VALUE
  FROM (XMLTable('<a>123</a>'));

COLUMN_VALUE
----------------------------------------
<a>123</a>
```

In the context of a `TABLE` collection expression, the value returned is the data type of the collection element. The following statements create the two levels of nested tables illustrated in Creating a Table: Multilevel Collection Example to show the uses of `COLUMN_VALUE` in this context:

```
CREATE TYPE phone AS TABLE OF NUMBER;
/
CREATE TYPE phone_list AS TABLE OF phone;
/
```

The next statement uses `COLUMN_VALUE` to select from the `phone` type:

```
SELECT t.COLUMN_VALUE
  FROM TABLE(phone(1,2,3)) t;

COLUMN_VALUE
------------
           1
           2
           3
```

In a nested type, you can use the `COLUMN_VALUE` pseudocolumn in both the select list and the `TABLE` collection expression:

```
SELECT t.COLUMN_VALUE
  FROM TABLE(phone_list(phone(1,2,3))) p, TABLE(p.COLUMN_VALUE) t;

COLUMN_VALUE
------------
           1
           2
           3
```

The keyword `COLUMN_VALUE` is also the name that Oracle Database generates for the scalar value of an inner nested table without a column or attribute name, as shown in the example that follows. In this context, `COLUMN_VALUE` is not a pseudocolumn, but an actual column name.

```
CREATE TABLE my_customers (
    cust_id       NUMBER,
    name          VARCHAR2(25),
    phone_numbers phone_list,
    credit_limit  NUMBER)
  NESTED TABLE phone_numbers STORE AS outer_ntab
  (NESTED TABLE COLUMN_VALUE STORE AS inner_ntab);
```

**See Also:**

- XMLTABLE for information on that function
- *table_collection_expression*::= for information on the `TABLE` collection expression
- `ALTER TABLE` examples in Nested Tables: Examples
- Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules for values of the `COLUMN_VALUE` pseudocolumn

# OBJECT_ID Pseudocolumn

The `OBJECT_ID` pseudocolumn returns the object identifier of a column of an object table or view. Oracle uses this pseudocolumn as the primary key of an object table. `OBJECT_ID` is useful in `INSTEAD OF` triggers on views and for identifying the ID of a substitutable row in an object table.

**Note:**

In earlier releases, this pseudocolumn was called `SYS_NC_OID$`. That name is still supported for backward compatibility. However, Oracle recommends that you use the more intuitive name `OBJECT_ID`.

**See Also:**

*Oracle Database Object-Relational Developer's Guide* for examples of the use of this pseudocolumn

# OBJECT_VALUE Pseudocolumn

The `OBJECT_VALUE` pseudocolumn returns system-generated names for the columns of an object table, `XMLType` table, object view, or `XMLType` view. This pseudocolumn is useful for identifying the value of a substitutable row in an object table and for creating object views with the `WITH OBJECT IDENTIFIER` clause.

**Note:**

In earlier releases, this pseudocolumn was called `SYS_NC_ROWINFO$`. That name is still supported for backward compatibility. However, Oracle recommends that you use the more intuitive name `OBJECT_VALUE`.

> **See Also:**
>
> - *object_table* and *object_view_clause* for more information on the use of this pseudocolumn
> - *Oracle Database Object-Relational Developer's Guide* for examples of the use of this pseudocolumn

# ORA_ROWSCN Pseudocolumn

`ORA_ROWSCN` reflects the system change-number (SCN) of the most recent change to a row. This change can be at the level of a block (coarse) or at the level of a row (fine-grained). The latter is provided by row-level dependency tracking. Refer to `CREATE TABLE` ... NOROWDEPENDENCIES | ROWDEPENDENCIES for more information on row-level dependency tracking. In the absence of row-level dependencies, `ORA_ROWSCN` reflects block-level dependencies.

Whether at the block level or at the row level, the `ORA_ROWSCN` should not be considered to be an exact SCN. For example, if a transaction changed row R in a block and committed at SCN 10, it is not always true that the `ORA_ROWSCN` for the row would return 10. While a value less than 10 would never be returned, any value greater than or equal to 10 could be returned. That is, the `ORA_ROWSCN` of a row is not always guaranteed to be the exact commit SCN of the transaction that last modified that row. However, with fine-grained `ORA_ROWSCN`, if two transactions T1 and T2 modified the same row R, one after another, and committed, a query on the `ORA_ROWSCN` of row R after the commit of T1 will return a value lower than the value returned after the commit of T2. If a block is queried twice, then it is possible for the value of `ORA_ROWSCN` to change between the queries even though rows have not been updated in the time between the queries. The only guarantee is that the value of `ORA_ROWSCN` in both queries is greater than the commit SCN of the transaction that last modified that row.

You cannot use the `ORA_ROWSCN` pseudocolumn in a query to a view. However, you can use it to refer to the underlying table when creating a view. You can also use this pseudocolumn in the `WHERE` clause of an `UPDATE` or `DELETE` statement.

`ORA_ROWSCN` is not supported for Flashback Query. Instead, use the version query pseudocolumns, which are provided explicitly for Flashback Query. Refer to the `SELECT` ... *flashback_query_clause* for information on Flashback Query and Version Query Pseudocolumns for additional information on those pseudocolumns.

**Restriction on ORA_ROWSCN:** This pseudocolumn is not supported for external tables.

### Example

The first statement below uses the `ORA_ROWSCN` pseudocolumn to get the system change number of the last operation on the `employees` table. The second statement uses the pseudocolumn with the `SCN_TO_TIMESTAMP` function to determine the timestamp of the operation:

```
SELECT ORA_ROWSCN, last_name
  FROM employees
  WHERE employee_id = 188;

SELECT SCN_TO_TIMESTAMP(ORA_ROWSCN), last_name
  FROM employees
  WHERE employee_id = 188;
```

> **See Also:**
>
> SCN_TO_TIMESTAMP

# ORA_SHARDSPACE_NAME Pseudocolumn

You can use the `ORA_SHARDSPACE_NAME` pseudocolumn to run queries across shards instead of a sharding key.

Before you can run cross-shard queries from the catalog, you must create users in the catalog with shared DDL enabled. Then you must grant these users access to the privately sharded tables.

The queries referencing the privately sharded tables will run across the shards in the catalog using the pseudocolumn `ORA_SHARDSPACE_NAME` associated to them. To run a cross shard query on a given shard, you must filter the query with the predicate `ORA_SHARDSPACE_NAME = <shardspace_name_belonging_to_name>`.

**Examples**

```
SELECT CUST_NAME, CUST_ID FROM CUSTOMER WHERE ORA_SHARDSPACE_NAME = 'EUROPE'
```

This query will run on one of the shards belonging to the shardspace named Europe. The query will run on the primary shard of the sharspace Europe or on one of its standbys, depending on the value of the parameter `MULTISHARD_QUERY_DATA_CONSISTENCY`.

A query like:

```
SELECT CUST_NAME, CUST_ID FROM CUSTOMER
```

where the table `CUSTOMER` is marked as privately sharded, will run on all shards.

# ROWID Pseudocolumn

For each row in the database, the `ROWID` pseudocolumn returns the address of the row. Oracle Database rowid values contain information necessary to locate a row:

- The data object number of the object
- The data block in the data file in which the row resides
- The position of the row in the data block (first row is 0)
- The data file in which the row resides (first file is 1). The file number is relative to the tablespace.

Usually, a rowid value uniquely identifies a row in the database. However, rows in different tables that are stored together in the same cluster can have the same rowid.

Values of the `ROWID` pseudocolumn have the data type `ROWID` or `UROWID`. Refer to Rowid Data Types and UROWID Data Type for more information.

Rowid values have several important uses:

- They are the fastest way to access a single row.
- They can show you how the rows in a table are stored.

- They are unique identifiers for rows in a table.

You should not use `ROWID` as the primary key of a table. If you delete and reinsert a row with the Import and Export utilities, for example, then its rowid may change. If you delete a row, then Oracle may reassign its rowid to a new row inserted later.

Although you can use the `ROWID` pseudocolumn in the `SELECT` and `WHERE` clause of a query, these pseudocolumn values are not actually stored in the database. You cannot insert, update, or delete a value of the `ROWID` pseudocolumn.

**Example**

This statement selects the address of all rows that contain data for employees in department 20:

```
SELECT ROWID, last_name
  FROM employees
  WHERE department_id = 20;
```

# ROWNUM Pseudocolumn

> **Note:**
>
> - The `ROW_NUMBER` built-in SQL function provides superior support for ordering the results of a query. Refer to ROW_NUMBER for more information.
> - The *`row_limiting_clause`* of the `SELECT` statement provides superior support for limiting the number of rows returned by a query. Refer to *row_limiting_clause* for more information.

For each row returned by a query, the `ROWNUM` pseudocolumn returns a number indicating the order in which Oracle selects the row from a table or set of joined rows. The first row selected has a `ROWNUM` of 1, the second has 2, and so on.

You can use `ROWNUM` to limit the number of rows returned by a query, as in this example:

```
SELECT *
  FROM employees
  WHERE ROWNUM < 11;
```

If an `ORDER BY` clause follows `ROWNUM` in the same query, then the rows will be reordered by the `ORDER BY` clause. The results can vary depending on the way the rows are accessed. For example, if the `ORDER BY` clause causes Oracle to use an index to access the data, then Oracle may retrieve the rows in a different order than without the index. Therefore, the following statement does not necessarily return the same rows as the preceding example:

```
SELECT *
  FROM employees
  WHERE ROWNUM < 11
  ORDER BY last_name;
```

If you embed the `ORDER BY` clause in a subquery and place the `ROWNUM` condition in the top-level query, then you can force the `ROWNUM` condition to be applied after the ordering of the rows. For example, the following query returns the employees with the 10 smallest employee numbers. This is sometimes referred to as **top-N reporting**:

```
SELECT *
  FROM (SELECT * FROM employees ORDER BY employee_id)
  WHERE ROWNUM < 11;
```

In the preceding example, the `ROWNUM` values are those of the top-level `SELECT` statement, so they are generated after the rows have already been ordered by `employee_id` in the subquery.

Conditions testing for `ROWNUM` values greater than a positive integer are always false. For example, this query returns no rows:

```
SELECT *
  FROM employees
  WHERE ROWNUM > 1;
```

The first row fetched is assigned a `ROWNUM` of 1 and makes the condition false. The second row to be fetched is now the first row and is also assigned a `ROWNUM` of 1 and makes the condition false. All rows subsequently fail to satisfy the condition, so no rows are returned.

You can also use `ROWNUM` to assign unique values to each row of a table, as in this example:

```
UPDATE my_table
  SET column1 = ROWNUM;
```

Refer to the function ROW_NUMBER for an alternative method of assigning unique numbers to rows.

> **Note:**
>
> Using `ROWNUM` in a query can affect view optimization.

# XMLDATA Pseudocolumn

Oracle stores `XMLType` data either in LOB or object-relational columns, based on XMLSchema information and how you specify the storage clause. The `XMLDATA` pseudocolumn lets you access the underlying LOB or object relational column to specify additional storage clause parameters, constraints, indexes, and so forth.

### Example

The following statements illustrate the use of this pseudocolumn. Suppose you create a simple table of `XMLType` with one `CLOB` column:

```
CREATE TABLE xml_lob_tab of XMLTYPE
  XMLTYPE STORE AS CLOB;
```

To change the storage characteristics of the underlying LOB column, you can use the following statement:

```
ALTER TABLE xml_lob_tab
  MODIFY LOB (XMLDATA) (STORAGE (MAXSIZE 2G) CACHE);
```

Now suppose you have created an XMLSchema-based table like the `xwarehouses` table created in Using XML in SQL Statements . You could then use the `XMLDATA` column to set the properties of the underlying columns, as shown in the following statement:

```
ALTER TABLE xwarehouses
  ADD (UNIQUE(XMLDATA."WarehouseId"));
```