5 Expressions

This chapter describes how to combine values, operators, and functions into **expressions**.

This chapter includes these sections:

About SQL Expressions

An **expression** is a combination of one or more values, operators, and SQL functions that evaluates to a value. An expression generally assumes the data type of its components.

This simple expression evaluates to 4 and has data type `NUMBER` (the same data type as its components):

```
2*2
```

The following expression is an example of a more complex expression that uses both functions and operators. The expression adds seven days to the current date, removes the time component from the sum, and converts the result to `CHAR` data type:

```
TO_CHAR(TRUNC(SYSDATE+7))
```

You can use expressions in:

- The select list of the `SELECT` statement

- A condition of the `WHERE` clause and `HAVING` clause
- The `CONNECT BY`, `START WITH`, and `ORDER BY` clauses
- The `VALUES` clause of the `INSERT` statement
- The `SET` clause of the `UPDATE` statement

For example, you could use an expression in place of the quoted string `'Smith'` in this `UPDATE` statement `SET` clause:

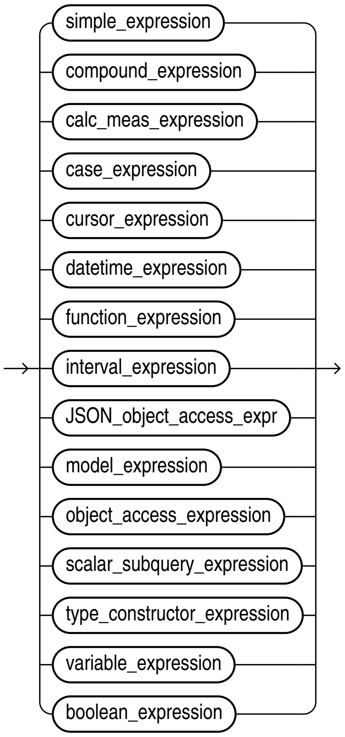
```
SET last_name = 'Smith';
```

This `SET` clause has the expression `INITCAP`(`last_name`) instead of the quoted string '`Smith`':

```
SET last_name = INITCAP(last_name);
```

Expressions have several forms, as shown in the following syntax:

***expr*::=**



simple_expression::=,,,,,,,boolean_expression::=

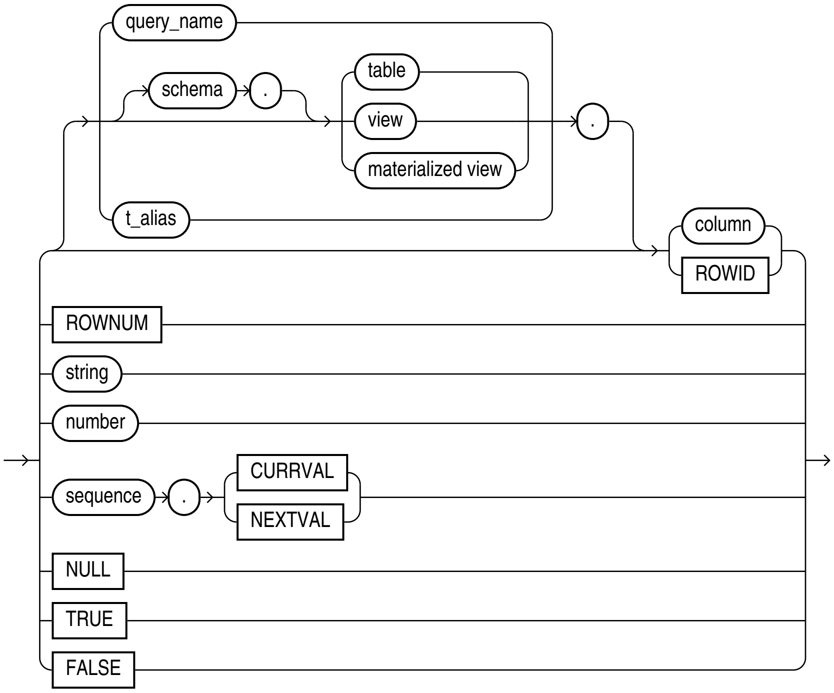
Oracle Database does not accept all forms of expressions in all parts of all SQL statements. Refer to the section devoted to a particular SQL statement in this book for information on restrictions on the expressions in that statement.

You must use appropriate expression notation whenever *`expr`* appears in conditions, SQL functions, or SQL statements in other parts of this reference. The sections that follow describe and provide examples of the various forms of expressions.

Simple Expressions

A simple expression specifies a column, pseudocolumn, constant, sequence number, or null.

***simple_expression*::=**



In addition to the schema of a user, *schema* can also be "PUBLIC" (double quotation marks required), in which case it must qualify a public synonym for a table, view, or materialized view. Qualifying a public synonym with "PUBLIC" is supported only in data manipulation language (DML) statements, not data definition language (DDL) statements.

You can specify ROWID only with a table, not with a view or materialized view. NCHAR and NVARCHAR2 are not valid pseudocolumn data types.

> **See Also:**
>
> Pseudocolumns for more information on pseudocolumns and subquery_factoring_clause for information on *query_name*

Some valid simple expressions are:

```
employees.last_name
'this is a text string'
10
N'this is an NCHAR string'
```

Analytic View Expressions

You can use analytic view expressions to create calculated measures within the definition of an analytic view or in a query that selects from an analytic view.

Analytic view expressions differ from other types of expressions in that they reference elements of hierarchies and analytic views rather than tables and columns.

An analytic view expression is one of the following:

- An *av_meas_expression*, which is based on a measure in an analytic view
- An *av_hier_expression*, which returns an attribute value of the related member

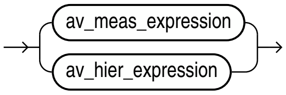
You use an analytic view expression as the *calc_meas_expression* parameter in a *calc_measure_clause* in a `CREATE ANALYTIC VIEW` statement and in the `WITH` or `FROM` clauses of a `SELECT` statement.

In defining a calculated measure, you may also use the following types of expression:

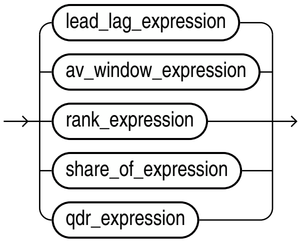
- Simple
- Case
- Compound
- Datetime
- Interval

Syntax

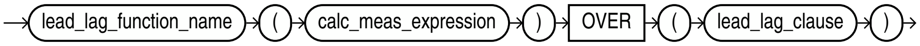
***av_expression*::=**



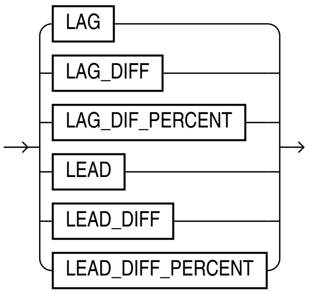
***av_meas_expression*::=**



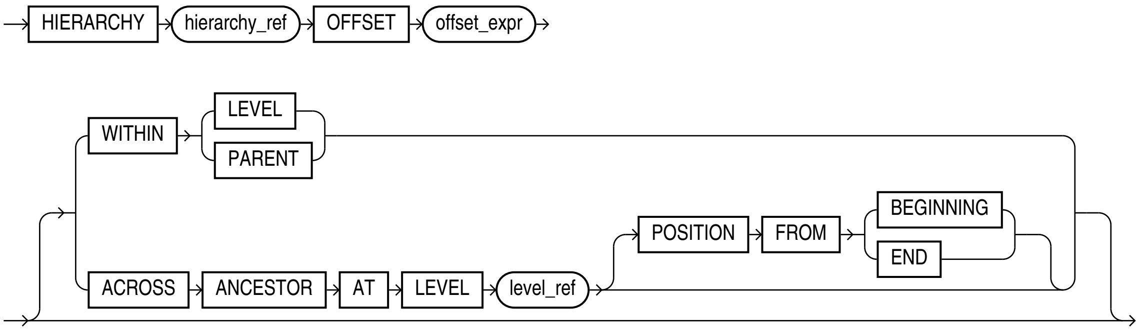
***lead_lag_expression*::=**



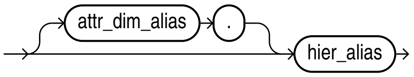
lead_lag_function_name::=



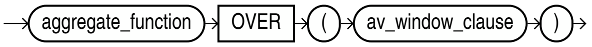
lead_lag_clause::=



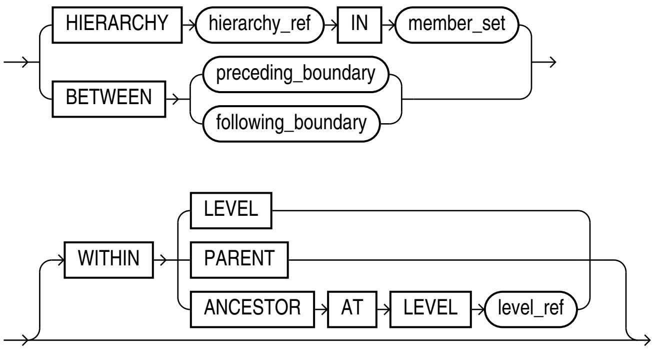
hierarchy_ref::=



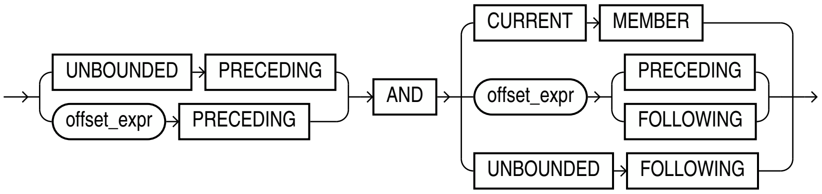
av_window_expression::=



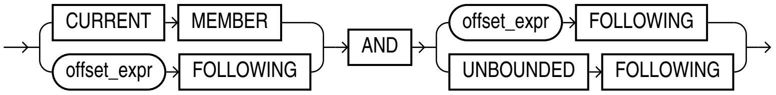
av_window_clause::=



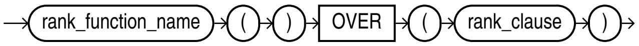
preceding_boundary ::=



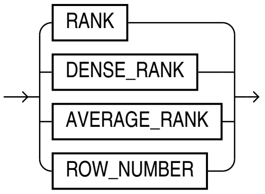
following_boundary::=



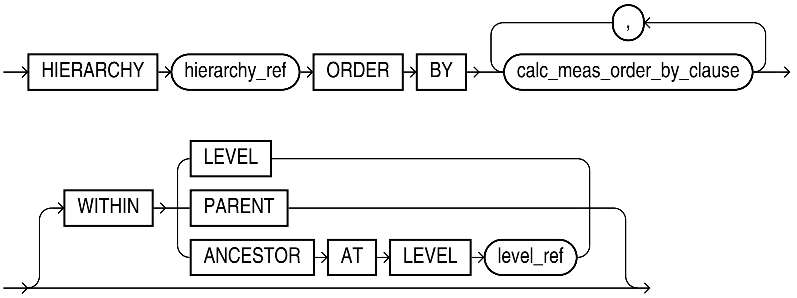
rank_expression::=



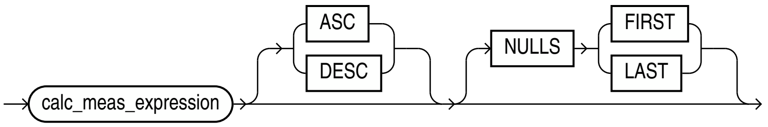
rank_function_name::=



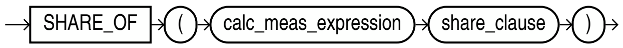
rank_clause::=



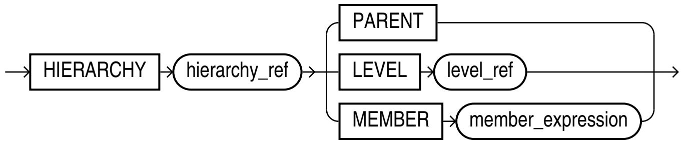
calc_meas_order_by_clause::=



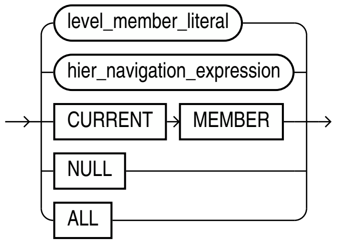
***share_of_expression*::=**



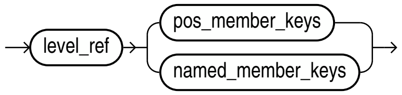
***share_clause*::=**



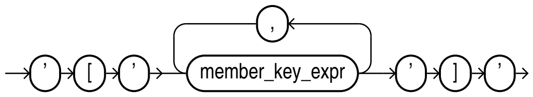
***member_expression*::=**



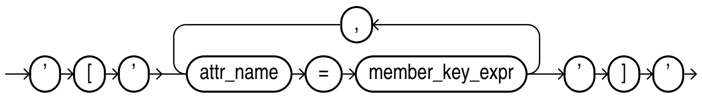
***level_member_literal*::=**



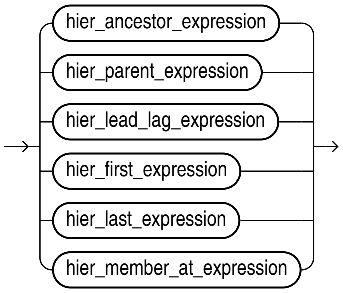
***pos_member_keys*::=**



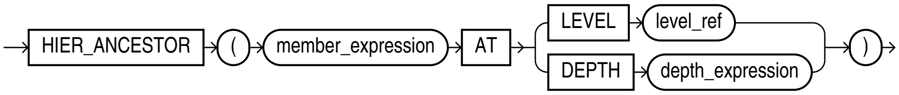
***named_member_keys*::=**



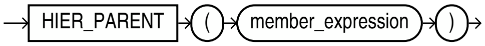
hier_navigation_expression::=



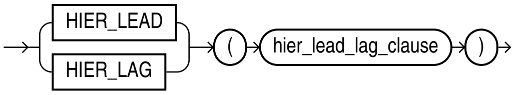
hier_ancestor_expression::=



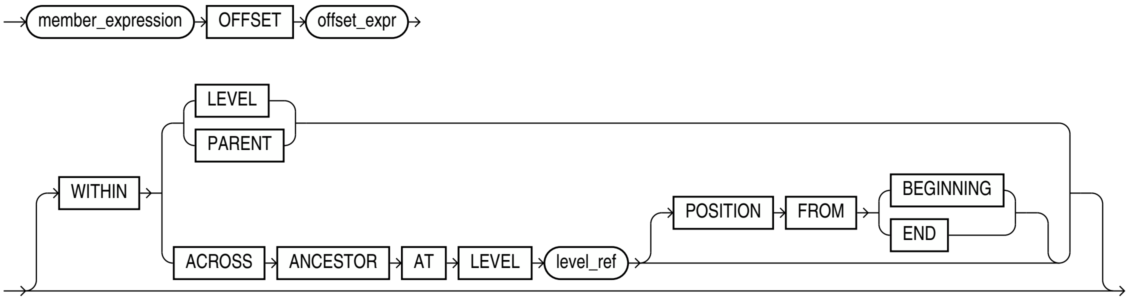
hier_parent_expression::=



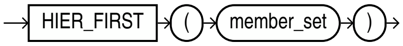
hier_lead_lag_expression::=



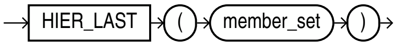
hier_lead_lag_clause::=



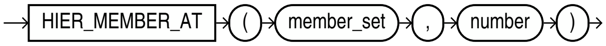
hier_first_expression::=



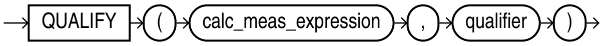
hier_last_expression::=



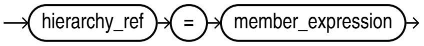
hier_member_at_expression::=



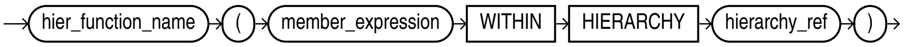
qdr_expression::=



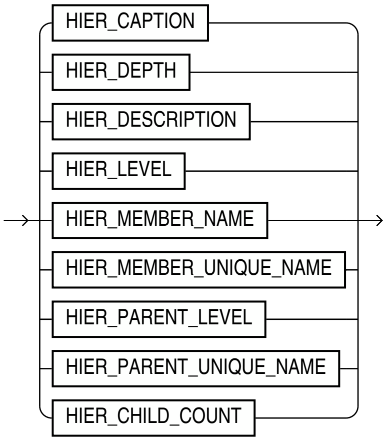
qualifier::=



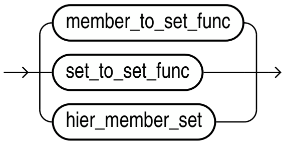
av_hier_expression::=



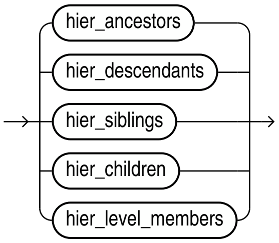
hier_function_name::=



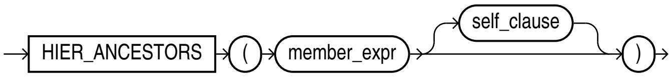
member_set::=



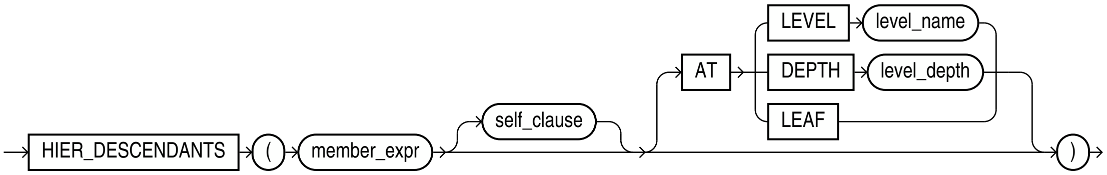
***member_to_set_func*::=**



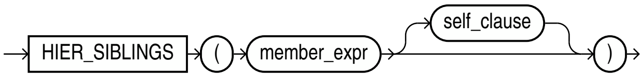
***hier_ancestors*::=**



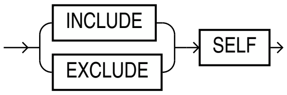
***hier_descendants*::=**



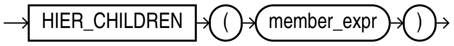
***hier_siblings*::=**



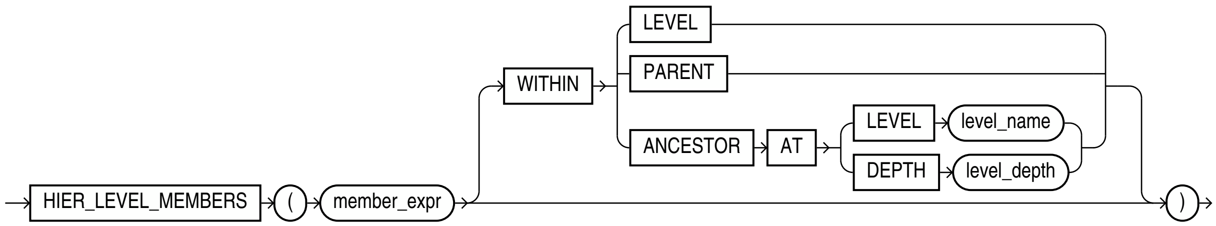
***self_clause* ::=**



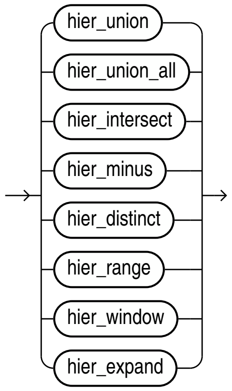
***hier_children*::=**



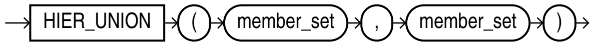
hier_level_members::=



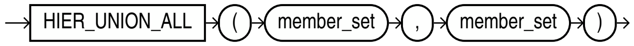
set_to_set_func::=



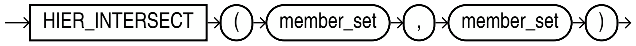
hier_union::=



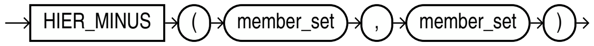
hier_union_all::=



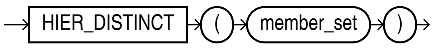
hier_intersect::=



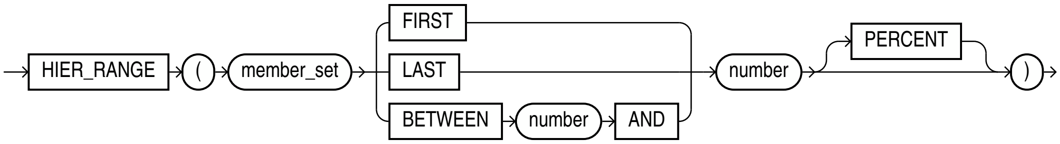
hier_minus::=



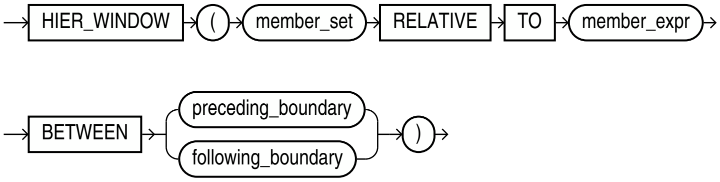
***hier_distinct*::=**



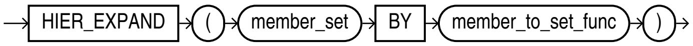
***hier_range*::=**



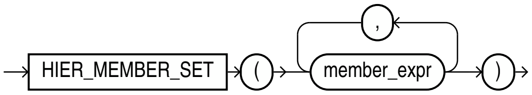
***hier_window*::=**



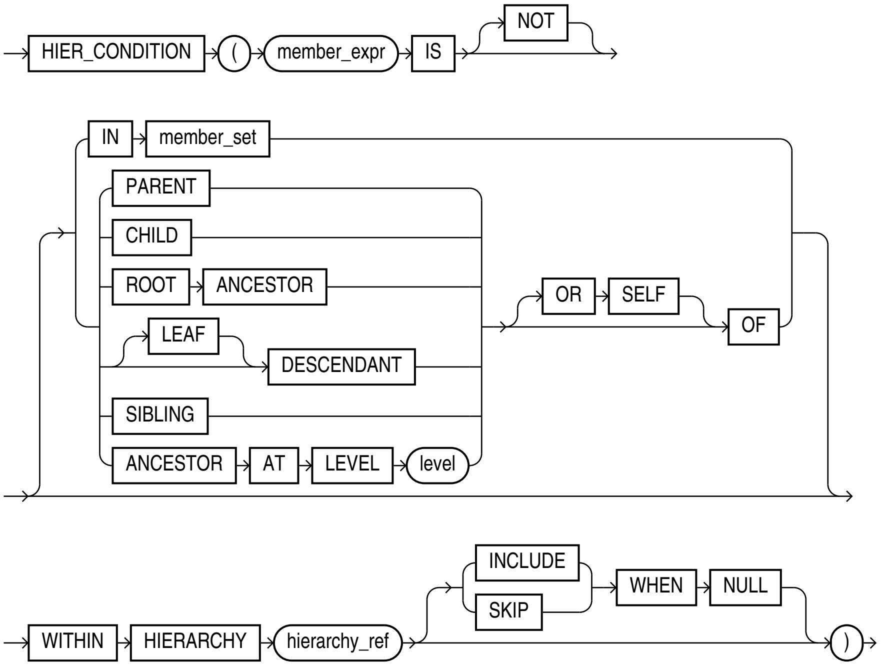
***hier_expand*::=**



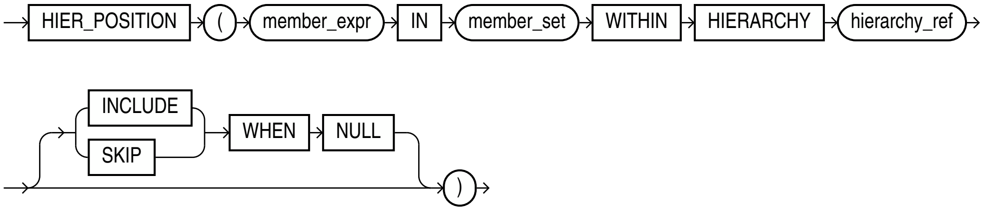
***hier_member_set*::=**



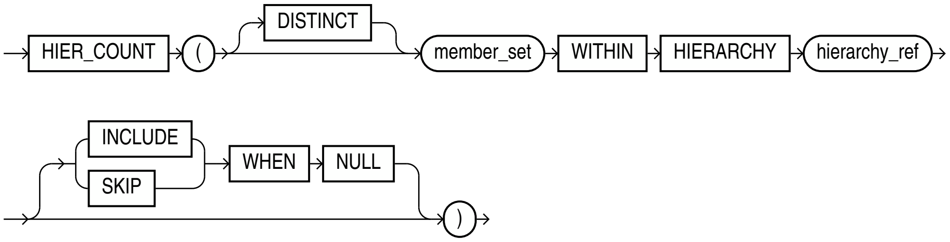
***hier_cond*::=**



***hier_position*::=**



***hier_count*::=**



Semantics

av_meas_expression

An expression that performs hierarchical navigation to locate related measure values.

lead_lag_expression

An expression that specifies a lead or lag operation that locates a related measure value by navigating forward or backward by some number of members within a hierarchy.

The *calc_meas_expression* parameter is evaluated in the new context created by the *lead_lag_expression*. This context has the same members as the outer context, except that the member of the specified hierarchy is changed to the related member specified by the lead or lag operation. The lead or lag function is run over the hierarchy members specified by the *lead_lag_clause* parameter.

lead_lag_function_name

The lead or lag function may be one of the following:

- `LAG` returns the measure value of an earlier member.
- `LAG_DIFF` returns the difference between the measure value of the current member and the measure value of an earlier member.
- `LAG_DIFF_PERCENT` returns the percent difference between the measure value of the current member and the measure value of an earlier member.
- `LEAD` returns the measure value of a later member.
- `LEAD_DIFF` returns the difference between the measure value of the current member and the measure value of a later member.
- `LEAD_DIFF_PERCENT` returns the percent difference between the measure value of the current member and the measure value of a later member.

lead_lag_clause

Specifies the hierarchy to evaluate and an offset value. The parameters of the *lead_lag_clause* are the following:

- `HIERARCHY` *hierarchy_ref* specifies the alias of a hierarchy as defined in the analytic view.
- `OFFSET` *offset_expr* specifies a *calc_meas_expression* that resolves to a number. The number specifies how many members to move either forward or backward from the current member. The ordering of members within a level is determined by the definition of the attribute dimension used by the hierarchy.
- `WITHIN LEVEL` specifies locating the related member by moving forward or backward by the offset number of members within the members that have the same level depth as the current member. The ordering of members within the level is determined by the definition of the attribute dimension used by the hierarchy.

  The `WITHIN LEVEL` operation is the default if neither the `WITHIN LEVEL` nor the `ACROSS ANCESTOR AT LEVEL` keywords are specified.
- `WITHIN PARENT` specifies locating the related member by moving forward or backward by the offset number of members within the members that have the same parent as the current member.
- `ACROSS ANCESTOR AT LEVEL` *level_ref* specifies locating the related member by navigating up to the ancestor (or to the member itself if no ancestor exists) of the current member at the level specified by *level_ref*, and noting the position of each ancestor member (including the member itself) within its parent. The *level_ref* parameter is the name of a level in the specified hierarchy.

Once the ancestor member is found, navigation moves either forward or backward the offset number of members within the members that have the same depth as the ancestor member. After locating the related ancestor, navigation proceeds back down the hierarchy from this member, matching the position within the parent as recorded on the way up (in reverse order). The position within the parent is either an offset from the first child or the last child depending on whether `POSITION FROM BEGINNING` or `POSITION FROM END` is specified. The default value is `POSITION FROM BEGINNING`. The ordering of members within the level is determined by the definition of the attribute dimension used by the hierarchy.

av_window_expression

An *`av_window_expression`* selects the set of members that are in the specified range starting from the current member and that are at the same depth as the current member. You can further restrict the selection of members by specifying a hierarchical relationship using a WITHIN phrase. Aggregation is then performed over the selected measure values to produce a single result for the expression.

The parameters for an *`av_window_expression`* are the following:

- *`aggregate_function`* is any existing SQL aggregate function except `COLLECT`, `GROUP_ID`, `GROUPING`, `GROUPING_ID`, `SYS_XMLAGG`, `XMLAGG`, and any multi-argument function. A user defined aggregate function is also allowed. The arguments to the aggregate function are *`calc_meas_expression`* expressions. These expressions are evaluated using the outer context, with the member of the specified hierarchy changed to each member in the related range. Therefore, each expression argument is evaluated once per related member. The results are then aggregated using the *`aggregate_function`*.
- `OVER (`*`av_window_clause`*`)` specifies the hierarchy to use and the boundaries of the window to consider.

> **See Also:**
>
> Aggregate Functions

av_window_clause

The *`av_window_clause`* parameter selects a range of members related to the current member. The range is between the members specified by the *`preceding_boundary`* or *`following_boundary`* parameters. The range is always computed over members at the same level as the current member.

Use `IN member_set` to specify an arbitrary member set to be used as the window for the window expression.

The parameters for a *`av_window_clause`* are the following:

- `HIERARCHY` *`hierarchy_ref`* specifies the alias of the hierarchy as defined in the analytic view.
- `BETWEEN` *`preceding_boundary`* or *`following_boundary`* defines the set of members to relate to the current member.
- `WITHIN LEVEL` selects the related members by applying the boundary clause to all members of the current level. This is the default when the `WITHIN` keyword is not specified.
- `WITHIN PARENT` selects the related members by applying the boundary clause to all members that share a parent with the current member.

- `WITHIN ANCESTOR AT LEVEL` selects the related members by applying the boundary clause to all members at the current depth that share an ancestor (or is the member itself) at the specified level with the current member. The value of the window expression is `NULL` if the current member is above the specified level. If the level is not in the specified hierarchy, then an error occurs.

preceding_boundary

The *`preceding_boundary`* parameter defines a range of members from the specified number of members backward in the level from the current member and forward to the specified end of the boundary. The following parameters specify the range:

- `UNBOUNDED PRECEDING` begins the range at the first member in the level.
- *`offset_expr`* `PRECEDING` begins the range at the *`offset_expr`* number of members backward from the current member. The *`offset_expr`* expression is a *`calc_meas_expression`* that resolves to a number. If the offset number is greater than the number of members from the current member to the first member in the level, than the first member is used as the start of the range.
- `CURRENT MEMBER` ends the range at the current member.
- *`offset_expr`* `PRECEDING` ends the range at the member that is *`offset_expr`* backward from the current member.
- *`offset_expr`* `FOLLOWING` ends the range at the member that is *`offset_expr`* forward from the current member.
- `UNBOUNDED FOLLOWING` ends the range at the last member in the level.

following_boundary

The *`following_boundary`* parameter defines a range of members from the specified number of members from the current member forward to the specified end of the range. The following parameters specify the range:

- `CURRENT MEMBER` begins the range at the current member.
- *`offset_expr`* `FOLLOWING` begins the range at the member that is *`offset_expr`* forward from the current member.
- *`offset_expr`* `FOLLOWING` ends the range at the member that is *`offset_expr`* forward from the current member.
- `UNBOUNDED FOLLOWING` ends the range at the last member in the level.

hierarchy_ref

A reference to a hierarchy of an analytic view. The *`hier_alias`* parameter specifies the alias of a hierarchy in the definition of the analytic view. You may use double quotes to escape special characters or preserve case, or both.

The optional *`attr_dim_alias`* parameter specifies the alias of an attribute dimension in the definition of the analytic view. You may use the *`attr_dim_alias`* parameter to resolve the ambiguity if the specified hierarchy alias conflicts with another hierarchy alias in the analytic view or if an attribute dimension is used more than once in the analytic view definition. You may use the *`attr_dim_alias`* parameter even when a name conflict does not exist.

rank_expression

Hierarchical rank calculations rank the related members of the specified hierarchy based on the order of the specified measure values and return the rank of the current member within those results.

Hierarchical rank calculations locate a set of related members in the specified hierarchy, rank all the related members based on the order of the specified measure values, and then return the rank of the current member within those results. The related members are a set of members at the same level as the current member. You may optionally restrict the set by some hierarchical relationship, but the set always includes the current member. The ordering of the measure values is determined by the *calc_meas_order_by_clause* of the *rank_clause*.

rank_function_name

Each hierarchical ranking function assigns an order number to each related member based on the *calc_meas_order_by_clause*, starting at 1. The functions differ in the way they treat measure values that are the same.

The functions and the differences between them are the following:

- RANK, which assigns the same rank to identical measure values. The rank after a set of tied values is the number of tied values plus the tied order value; therefore, the ordering may not be consecutive numbers.
- DENSE_RANK, which assigns the same minimum rank to identical measure values. The rank after a set of tied values is always one more than the tied value; therefore, the ordering always has consecutive numbers.
- AVERAGE_RANK, assigns the same average rank to identical values. The next value after the average rank value is the number of identical values plus 1, that sum divided by 2, plus the average rank value. For example, for the series of five values 4, 5, 10, 5, 7, AVERAGE_RANK returns 1, 1.5, 1.5, 3, 4. For the series 2, 12, 10, 12, 17, 12, the returned ranks are 1, 2, 3, 3, 3, 5.
- ROW_NUMBER, which assigns values that are unique and consecutive across the hierarchy members. If the *calc_meas_order_by_clause* results in equal values then the results are non-deterministic.

rank_clause

The rank_clause locates a range of hierarchy members related to the current member. The range is some subset of the members in the same level as the current member. The subset is determined from the WITHIN clause.

Valid values for the WITHIN clause are:

- WITHIN LEVEL, which specifies that the related members are all the members of the current level. This is the default subset if the WITHIN keyword is not specified.
- WITHIN PARENT, which specifies that the related members all share a parent with the current member
- WITHIN ANCESTOR AT LEVEL, which specifies that the related members are all of the members of the current level that share an ancestor (or self) at the specified level with the current member.

share_of_expression

A *share_of_expression* expression calculates the ratio of an expression's value for the current context over the expression's value at a related context. The expression is a *calc_meas_expression* that is evaluated at the current context and the related context. The *share_clause* specification determines the related context to use.

share_clause

A *share_clause* modifies the outer context by setting the member for the specified hierarchy to a related member.

The parameters of the share clause are the following:

- `HIERARCHY` *hierarchy_ref* specifies the name of the hierarchy that is the outer context for the *share_of_expression* calculations.
- `PARENT` specifies that the related member is the parent of the current member.
- `LEVEL` *level_ref* specifies that the related member is the ancestor (or is the member itself) of the current member at the specified level in the hierarchy. If the current member is above the specified level, then `NULL` is returned for the share expression. If the level is not in the hierarchy, then an error occurs.
- `MEMBER` *member_expression* specifies that the related member is the member returned after evaluating the *member_expression* in the current context. If the value of the specified member is `NULL`, then `NULL` is returned for the share expression.

member_expression

A *member_expression* a member expression is an expression that returns a single member in a hierarchy. A member set contains multiple members (possibly including duplicates), and may be empty. A multiple member expression is an expression that returns a member set.

The hierarchy can be determined from the outer expression (enforced by the syntax).

A *member_expression* can be one of the following:

- *level_member_literal* expression specifies a particular member contained within a particular level. The member is identified by specifying a key value.
- *hier_navigation_expr* is an expression that relates one member of the hierarchy to another member.
- `CURRENT MEMBER` indicates that the function should operate on the current member of the hierarchy, typically the starting point of a function, used in the innermost function when nesting. For example, `HIER_PARENT(HIER_PARENT(CURRENT MEMBER))` returns the grandparent of the current member.

  When used within a hierarchical window expression, for example, the current member is the one in which the window is currently operating. The current member can also be provided by some member set functions as well as in `QUALIFY`.
- The `NULL` keyword is simply a placeholder for a member that is not in the hierarchy, called an empty member. This can be specified explicitly, but can also be the result of a function. For example, `HIER_PARENT` on the `ALL` member of a hierarchy will result in the empty member. The empty member should not be confused with `NULL` members that are true hierarchy members of `SKIP WHEN NULL` levels.

- The ALL keyword specifies the ALL member, the ultimate ancestor of every other member in the hierarchy. Every hierarchy has an implicit ALL member contained within an implicit ALL level.

level_member_literal

A level member expression specifies a particular member contained within a particular level. The member is identified by specifying a key value. If the attribute name is not specified, it is assumed to be the primary key attribute. Typically, just a single attribute needs qualification.

In the case of a level with either a multi-column key or a SKIP WHEN NULL level, multiple attributes need to be qualified in order to uniquely identify a member. If the key attribute is specified, ordering is not important. If not specified, the ordering is assumed to be the ordering as defined in the xxx_HIER_LEVEL_ID_ATTRS data dictionary view.

pos_member_keys

The *member_key_expr* expression resolves to the key value for the member. When specified by position, all components of the key must be given in the order found in the ALL_HIER_LEVEL_ID_ATTRS dictionary view. For a hierarchy in which the specified level is not determined by the child level, then all member key values of all such child levels must be provided preceding the current level's member key or keys. Duplicate key components are only specified the first time they appear.

The primary key is used when *level_member_literal* is specified using the *pos_member_keys* phrase. You can reference an alternate key by using the *named_member_keys* phrase.

named_member_keys

The *member_key_expr* expression resolves to the key value for the member. The *attr_name* parameter is an identifier for the name of the attribute. If all of the attribute names do not make up a key or alternate key of the specified level, then an error occurs.

When specified by name, all components of the key must be given and all must use the attribute *name* = *value* form, in any order. For a hierarchy in which the specified level is not determined by the child level, then all member key values of all such child levels must be provided, also using the named form. Duplicate key components are only specified once.

hier_navigation_expression

A *hier_navigation_expression* expression navigates from the specified member to a different member in the hierarchy.

hier_ancestor_expression

Returns the ancestor of the specified member at the given level. The level can either be specified by name or depth. If the member has no ancestor at the specified level, the empty member is returned.

The depth is specified as an expression that must resolve to a number. If the member is at a level or depth above the specified member, or the member is NULL, then NULL is returned for the expression value. If the specified level is not in the context hierarchy, then an error occurs.

hier_parent_expression

Returns the parent of the specified member, or the empty member if it has no parent (i.e. is the ALL member).

hier_first_expression

Returns the first element in the specified member set. If the member set is empty, the empty member is returned.

hier_last_expression

Returns the last element in the specified member set. If the member set is empty, the empty member is returned.

hier_member_at_expression

Returns the member in the specified member set at the position identified by the given expression representing the position, where positions are 1-based. If the specified position is greater than the number of elements in the member set, the empty member is returned. The expression must be coercible to a numeric type, and will be rounded to the nearest integer. If the expression resolves to an integer less than 1, the empty member is returned.

hier_lead_lag_expression

Navigates from the specified member to a related member by moving forward or backward some number of members within the context hierarchy. The `HIER_LEAD` keyword returns a later member. The `HIER_LAG` keyword returns an earlier member.

hier_lead_lag_clause

Navigates the *`offset_expr`* number of members forward or backward from the specified member. The ordering of members within a level is specified in the definition of the attribute dimension.

The optional parameters of *`hier_lead_lag_clause`* are the following:

- `WITHIN LEVEL` locates the related member by moving forward or backward *`offset_expr`* members within the members that have the same depth as the current member. The ordering of members within the level is determined by the definition of the attribute dimension. The `WITHIN LEVEL` operation is the default if neither the `WITHIN` nor the `ACROSS` keywords are used.
- `WITHIN PARENT` locates the related member by moving forward or backward *`offset_expr`* members within the members that have the same depth as the current member, but only considers members that share a parent with the current member. The ordering of members within the level is determined by the definition of the attribute dimension.
- `WITHIN ACROSS ANCESTOR AT LEVEL` locates the related member by navigating up to the ancestor of the current member (or to the member itself) at the specified level, noting the position of each ancestor member (including the member itself) within its parent. Once the ancestor member is found, navigation moves forward or backward *`offset_expr`* members within the members that have the same depth as the ancestor member.

  After locating the related ancestor, navigation moves back down the hierarchy from that member, matching the position within the parent as recorded on the way up (in reverse order). The position within the parent is either an offset from the first child or the last child depending on whether `POSITION FROM BEGINNING` or `POSITION FROM END` is specified, defaulting to `POSITION FROM BEGINNING`. The ordering of members within the level is determined by the definition of the attribute dimension.

qdr_expression

A *qdr_expression* is a qualified data reference that evaluates the specified *calc_meas_expression* in a new context and sets the hierarchy member to the new value.

qualifier

A qualifier modifies the outer context by setting the member for the specified hierarchy to the member resulting from evaluating *member_expression*. If *member_expression* is NULL, then the result of the *qdr_expression* selection is NULL.

av_hier_expression

An *av_hier_expression* performs hierarchy navigation to locate an attribute value of the related member. An *av_hier_expression* may be a top-level expression, whereas a *hier_navigation_expression* may only be used as a *member_expression* argument.

For example, in the following query HIER_MEMBER__NAME is an *av_hier_expression* and HIER_PARENT is a *hier_navigation_expression*.

```
HIER_MEMBER_NAME(HIER_PARENT(CURRENT MEMBER) WITHIN HIERARCHY product_hier))
```

hier_function_name

The *hier_function_name* values are the following:

- HIER_CAPTION, which returns the caption of the related member in the hierarchy.
- HIER_DEPTH, which returns one less than the number of ancestors between the related member and the ALL member in the hierarchy. The depth of the ALL member is 0.
- HIER_DESCRIPTION, which returns the description of the related member in the hierarchy.
- HIER_LEVEL, which returns as a string value the name of the level to which the related member belongs in the hierarchy.
- HIER_MEMBER_NAME, which returns the member name of the related member in the hierarchy.
- HIER_MEMBER_UNIQUE_NAME, which returns the member unique name of the related member in the hierarchy.

member_set

The primary purpose of member sets is to allow them to be used within hierarchical functions. A member set is the result of either a member to set function or a set to set function.

member_to_set_func

All member to set functions take a member expression as input and produce a member set in hierarchy order. The variants that have a self_clause can specify whether or not the member specified in the given member expression itself should be included in the resulting member set, with the default being that it is excluded. If the given member is the empty member, all functions return an empty set even when INCLUDE SELF is specified.

hier_ancestors

Returns a member set consisting of all ancestors of the specified member, optionally including the member itself. If the member has no ancestors (i.e. is the `ALL` member) and self is excluded, an empty set is returned.

hier_descendants

Returns a member set consisting of all descendants of the specified member, optionally including the member itself. If the `AT` clause is specified, the set of descendants are filtered to only include members at the specified level or depth. If the member has no descendants (i.e. is a leaf) optionally filtered to the given level and self is excluded, an empty set is returned.

hier_siblings

Returns a member set consisting of all siblings of the specified member, optionally including the member itself. A sibling is defined as any member whose parent is equal to the parent of the given member. If the member has no siblings and self is excluded, an empty set is returned.

hier_children

Returns a member set consisting of all children of the specified member. If the member has no children, an empty set is returned.

hier_level_members

Returns a member set consisting of members at the same level as the given member that have a common ancestor as defined by the `WITHIN` clause. This function always includes self. `WITHIN PARENT` returns all members that are children of the given member’s parent. `WITHIN ANCESTOR AT` returns all members at the same level as the given member that have the same ancestor at the specified level. `WITHIN LEVEL` returns all members at the same level as the given member. If the `WITHIN` clause is omitted, the default is `WITHIN LEVEL`.

hier_member_set

Returns a member set consisting of explicitly specified members, in the order specified. This function is in its own category as it is not really performing a navigation, but simply building a set from some number of given members. Duplicate members are allowed. Any empty members in the given set are ignored, as a member set will never include the empty member.

set_to_set_func

The functions in this section all operate on a member set. They perform standard set operations and further hierarchical navigation.

hier_union

Returns the distinct union of members among the two given sets by taking all distinct members of the first set followed by all members in the second set that are not in the first set.

hier_union_all

Returns all members in the first set followed by all members in the second set, retaining duplicates.

hier_intersect

Returns all distinct members in order from the first set that also appear in the second set.

hier_minus

Returns all distinct members in order from the first set that do not appear in the second set.

hier_distinct

Returns the distinct members in order from the given set.

hier_range

Returns members in order from the set that fall within the specified range. In all cases, *`number`* is an expression that is coercible to a number. When `PERCENT` is not specified, the expression must evaluate to a positive integer. `FIRST` will return the first `N`members in the set. If `N` is greater than the number of elements in the set, all elements are returned. `LAST` will return the last `N` members in the set. If `N` is greater than the number of elements in the set, all elements are returned. `BETWEEN` will return all elements whose position in the set is >= the start position and <= the given end position, with positions being 1-based. If the `PERCENT` keyword is specified, the *`number`* arguments all represent percentages and must evaluate to a number between 0 and 100.

hier_window

Returns all members in order from the given set which fall within the specified boundary relative to the given member. If the given member is not in the given set, an empty set is returned.

hier_expand

For each member in the given set, applies the specified member to set function. References to `CURRENT MEMBER` in the member to set function refer to the current member in the set to which it is being applied. The member sets produced for the members are combined using the semantics of `HIER_UNION_ALL` (i.e. retaining duplicates).

hier_cond

Use `IN` `member_set` to specify an arbitrary member set to use for the comparison.

hier_position

Returns the numeric 1-based position of the first occurrence of the member identified by mbr_expr in the specified member set, with references to `CURRENT MEMBER` referring to the current member in the set to which it is being applied. If the member does not appear in the set, `NULL` is returned. This could be useful if a user wanted to order the output of a query based on the set order.

hier_count

Returns the number of members in the member set. If the `DISTINCT` keyword is included, returns the number of distinct members in the member set.

Examples of Analytic View Expressions

This topic contains examples that show calculated measures defined in the MEASURES clause of an analytic view and in the ADD MEASURES clause of a SELECT statement.

The examples are the following:

- Examples of LAG Expressions
- Example of a Window Expression
- Examples of SHARE OF Expressions
- Examples of QDR Expressions
- Example of an Added Measure Using the RANK Function

For more examples, see the tutorials on analytic views at the SQL Live website at https://livesql.oracle.com/apex/livesql/file/index.html.

Examples of LAG Expressions

These calculated measures different LAG operations.

```
-- These calculated measures are from the measures_clause of the
-- sales_av analytic view.
MEASURES
 (sales FACT sales,                         -- A base measure
  units FACT units,                         -- A base measure
  sales_prior_period AS                     -- Calculated measures
    (LAG(sales) OVER (HIERARCHY time_hier OFFSET 1)),
  sales_year_ago AS
    (LAG(sales) OVER (HIERARCHY time_hier OFFSET 1
     ACROSS ANCESTOR AT LEVEL year)),
  chg_sales_year_ago AS
    (LAG_DIFF(sales) OVER (HIERARCHY time_hier OFFSET 1
     ACROSS ANCESTOR AT LEVEL year)),
  pct_chg_sales_year_ago AS
    (LAG_DIFF_PERCENT(sales) OVER (HIERARCHY time_hier OFFSET 1
     ACROSS ANCESTOR AT LEVEL year)),
  sales_qtr_ago AS
    (LAG(sales) OVER (HIERARCHY time_hier OFFSET 1
     ACROSS ANCESTOR AT LEVEL quarter)),
  chg_sales_qtr_ago AS
    (LAG_DIFF(sales) OVER (HIERARCHY time_hier OFFSET 1
     ACROSS ANCESTOR AT LEVEL quarter)),
  pct_chg_sales_qtr_ago AS
    (LAG_DIFF_PERCENT(sales) OVER (HIERARCHY time_hier OFFSET 1
     ACROSS ANCESTOR AT LEVEL quarter))
 )
```

Example of a Window Expression

This calculated measure uses a window operation.

```
MEASURES
 (sales FACT sales,
  units FACT units,
  sales_qtd AS
    (SUM(sales) OVER (HIERARCHY time_hier
     BETWEEN UNBOUNDED PRECEDING AND CURRENT MEMBER
     WITHIN ANCESTOR AT LEVEL QUARTER)),
  sales_ytd AS
    (SUM(sales) OVER (HIERARCHY time_hier
     BETWEEN UNBOUNDED PRECEDING AND CURRENT MEMBER
     WITHIN ANCESTOR AT LEVEL YEAR))
 )
```

Examples of SHARE OF Expressions

These calculated measures use `SHARE OF` expressions.

```
MEASURES
 (sales FACT sales,
  units FACT units,
 sales_shr_parent_prod AS
   (SHARE_OF(sales HIERARCHY product_hier PARENT)),
 sales_shr_parent_geog AS
   (SHARE_OF(sales HIERARCHY geography_hier PARENT)),
 sales_shr_region AS
   (SHARE_OF(sales HIERARCHY geography_hier LEVEL REGION))
 )
```

Examples of QDR Expressions

These calculated measures use the `QUALIFY` keyword to specify qualified data reference expressions.

```
MEASURES
 (sales FACT sales,
  units FACT units,
  sales_2011 AS
    (QUALIFY (sales, time_hier = year['11'])),
  sales_pct_chg_2011 AS
    ((sales - (QUALIFY (sales, time_hier = year['11']))) /
    (QUALIFY (sales, time_hier = year['11'])))
 )
```

Example of an Added Measure Using the RANK Function

In this example, the units_geog_rank_level measure uses the `RANK` function to rank geography hierarchy members within a level based on units.

```
SELECT geography_hier.member_name AS "Region",
       units AS "Units",
       units_geog_rank_level AS "Rank"
  FROM ANALYTIC VIEW (
    USING sales_av HIERARCHIES (geography_hier)
    ADD MEASURES (
      units_geog_rank_level AS (
        RANK() OVER (
          HIERARCHY geography_hier
          ORDER BY units desc nulls last
          WITHIN LEVEL))
    )
  )
  WHERE geography_hier.level_name IN ('REGION')
  ORDER BY units_geog_rank_level;
```

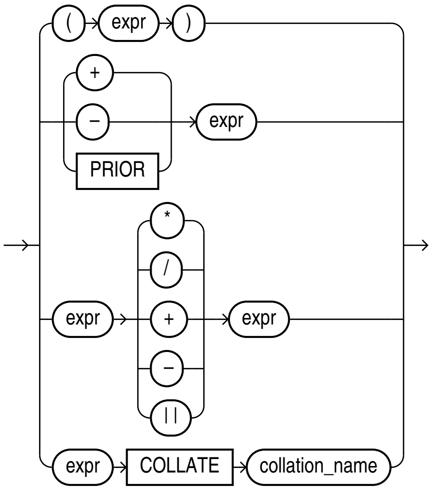
The following is the result of the query.

```
Regions             Units  Rank
-------------  ---------  ----
Asia            56017849     1
South America   23904155     2
North America   20523698     3
Africa          12608308     4
Europe           8666520     5
Oceania           427664     6
```

Compound Expressions

A compound expression specifies a combination of other expressions.

***compound_expression*::=**



You can use any built-in function as an expression (Function Expressions). However, in a compound expression, some combinations of functions are inappropriate and are rejected. For example, the `LENGTH` function is inappropriate within an aggregate function.

The `PRIOR` operator is used in `CONNECT BY` clauses of hierarchical queries.

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

See Also:

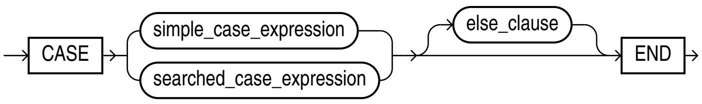
- Operator Precedence
- Hierarchical Queries
- COLLATE Operator

Some valid compound expressions are:

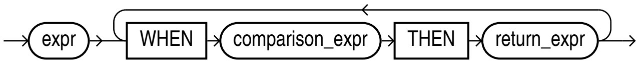
```
('CLARK' || 'SMITH')
LENGTH('MOOSE') * 57
SQRT(144) + 72
my_fun(TO_CHAR(sysdate,'DD-MMM-YY'))
name COLLATE BINARY_CI
```

CASE Expressions

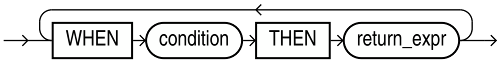
`CASE` expressions let you use `IF` ... `THEN` ... `ELSE` logic in SQL statements without having to invoke procedures. The syntax is:



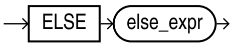
***simple_case_expression*::=**



***searched_case_expression*::=**



***else_clause*::=**



In a simple `CASE` expression, Oracle Database searches for the first `WHEN` ... `THEN` pair for which *`expr`* is equal to *`comparison_expr`* and returns *`return_expr`*. If none of the `WHEN` ... `THEN` pairs meet this condition, and an `ELSE` clause exists, then Oracle returns *`else_expr`*. Otherwise, Oracle returns null.

In a searched CASE expression, Oracle searches from left to right until it finds an occurrence of *condition* that is true, and then returns *return_expr*. If no *condition* is found to be true, and an ELSE clause exists, then Oracle returns *else_expr*. Otherwise, Oracle returns null.

Oracle Database uses **short-circuit evaluation**. For a simple CASE expression, the database evaluates each *comparison_expr* value only before comparing it to *expr*, rather than evaluating all *comparison_expr* values before comparing any of them with *expr*. Consequently, Oracle never evaluates a *comparison_expr* if a previous *comparison_expr* is equal to *expr*. For a searched CASE expression, the database evaluates each *condition* to determine whether it is true, and never evaluates a *condition* if the previous *condition* was true.

For a simple CASE expression, the *expr* and all *comparison_expr* values must either have the same data type (CHAR, VARCHAR2, NCHAR, or NVARCHAR2, NUMBER, BINARY_FLOAT, or BINARY_DOUBLE) or must all have a numeric data type. If all expressions have a numeric data type, then Oracle determines the argument with the highest numeric precedence, implicitly converts the remaining arguments to that data type, and returns that data type.

For both simple and searched CASE expressions, all of the *return_expr*s must either have the same data type (CHAR, VARCHAR2, NCHAR, or NVARCHAR2, NUMBER, BINARY_FLOAT, or BINARY_DOUBLE) or must all have a numeric data type. If all return expressions have a numeric data type, then Oracle determines the argument with the highest numeric precedence, implicitly converts the remaining arguments to that data type, and returns that data type.

The maximum number of arguments in a CASE expression is 65535. All expressions count toward this limit, including the initial expression of a simple CASE expression and the optional ELSE expression. Each WHEN ... THEN pair counts as two arguments. To avoid exceeding this limit, you can nest CASE expressions so that the *return_expr* itself is a CASE expression.

The comparison performed by the simple CASE expression is collation-sensitive if the compared arguments have a character data type (CHAR, VARCHAR2, NCHAR, or NVARCHAR2). The collation determination rules determine the collation to use.

See Also:

- Table 2-9 for more information on implicit conversion
- Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation and determination rules for the CASE expression
- Numeric Precedence for information on numeric precedence
- COALESCE and NULLIF for alternative forms of CASE logic
- *Oracle Database Data Warehousing Guide* for examples using various forms of the CASE expression

Simple CASE Example

For each customer in the sample oe.customers table, the following statement lists the credit limit as "Low" if it equals $100, "High" if it equals $5000, and "Medium" if it equals anything else.

```
SELECT cust_last_name,
   CASE credit_limit WHEN 100 THEN 'Low'
   WHEN 5000 THEN 'High'
   ELSE 'Medium' END AS credit
   FROM customers
   ORDER BY cust_last_name, credit;

CUST_LAST_NAME       CREDIT
-------------------- ------
Adjani               Medium
Adjani               Medium
Alexander            Medium
Alexander            Medium
Altman               High
Altman               Medium
. . .
```

Searched CASE Example

The following statement finds the average salary of the employees in the sample table `oe.employees`, using $2000 as the lowest salary possible:

```
SELECT AVG(CASE WHEN e.salary > 2000 THEN e.salary
   ELSE 2000 END) "Average Salary" FROM employees e;

Average Salary
--------------
    6461.68224
```

Column Expressions

A column expression, which is designated as *`column_expression`* in subsequent syntax diagrams, is a limited form of *`expr`*. A column expression can be a simple expression, compound expression, function expression, boolean expression, or expression list, but it can contain only the following forms of expression:

- Columns of the subject table — the table being created, altered, or indexed
- Constants (strings or numbers)
- Deterministic functions — either SQL built-in functions or user-defined functions

No other expression forms described in this chapter are valid. In addition, compound expressions using the `PRIOR` keyword are not supported, nor are aggregate functions.

You can use a column expression for these purposes:

- To create a function-based index.
- To explicitly or implicitly define a virtual column. When you define a virtual column, the defining *`column_expression`* must refer only to columns of the subject table that have already been defined, in the current statement or in a prior statement.

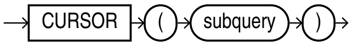
The combined components of a column expression must be deterministic. That is, the same set of input values must return the same set of output values.

See Also:

Simple Expressions , Compound Expressions , Function Expressions , and Expression Lists for information on these forms of *`expr`*

CURSOR Expressions

A CURSOR expression returns a nested cursor. This form of expression is equivalent to the PL/SQL REF CURSOR and can be passed as a REF CURSOR argument to a function.



A nested cursor is implicitly opened when the cursor expression is evaluated. For example, if the cursor expression appears in a select list, a nested cursor will be opened for each row fetched by the query. The nested cursor is closed only when:

- The nested cursor is explicitly closed by the user
- The parent cursor is reexecuted
- The parent cursor is closed
- The parent cursor is cancelled
- An error arises during fetch on one of its parent cursors (it is closed as part of the clean-up)

Restrictions on CURSOR Expressions

The following restrictions apply to CURSOR expressions:

- If the enclosing statement is not a SELECT statement, then nested cursors can appear only as REF CURSOR arguments of a procedure.
- If the enclosing statement is a SELECT statement, then nested cursors can also appear in the outermost select list of the query specification or in the outermost select list of another nested cursor.
- Nested cursors cannot appear in views.
- You cannot perform BIND and EXECUTE operations on nested cursors.

Examples

The following example shows the use of a CURSOR expression in the select list of a query:

```
SELECT department_name, CURSOR(SELECT salary, commission_pct
   FROM employees e
   WHERE e.department_id = d.department_id)
   FROM departments d
   ORDER BY department_name;
```

The next example shows the use of a CURSOR expression as a function argument. The example begins by creating a function in the sample OE schema that can accept the REF CURSOR argument. (The PL/SQL function body is shown in italics.)

```
CREATE FUNCTION f(cur SYS_REFCURSOR, mgr_hiredate DATE)
   RETURN NUMBER IS
   emp_hiredate DATE;
   before number :=0;
   after number:=0;
begin
  loop
    fetch cur into emp_hiredate;
    exit when cur%NOTFOUND;
    if emp_hiredate > mgr_hiredate then
      after:=after+1;
    else
      before:=before+1;
    end if;
  end loop;
  close cur;
  if before > after then
    return 1;
  else
    return 0;
  end if;
end;
/
```

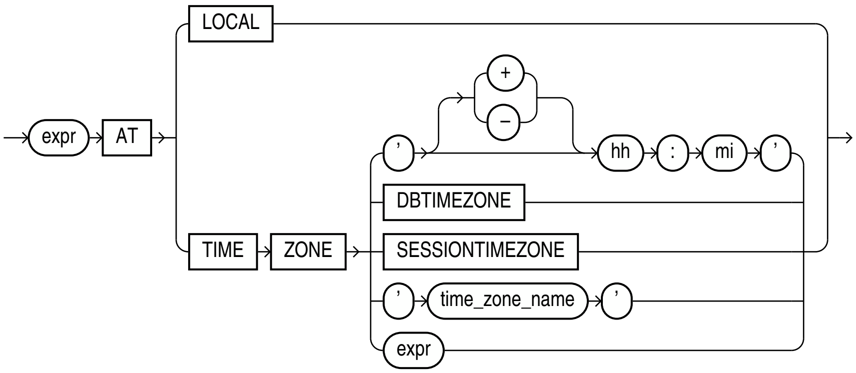
The function accepts a cursor and a date. The function expects the cursor to be a query returning a set of dates. The following query uses the function to find those managers in the sample `employees` table, most of whose employees were hired before the manager.

```
SELECT e1.last_name FROM employees e1
   WHERE f(
   CURSOR(SELECT e2.hire_date FROM employees e2
   WHERE e1.employee_id = e2.manager_id),
   e1.hire_date) = 1
   ORDER BY last_name;

LAST_NAME
-------------------------
Cambrault
Higgins
Hunold
Kochhar
Mourgos
Zlotkey
```

Datetime Expressions

A datetime expression yields a value of one of the datetime data types.

***datetime_expression*::=**



The initial *expr* is any expression, except a scalar subquery expression, that evaluates to a value of data type `TIMESTAMP`, `TIMESTAMP WITH TIME ZONE`, or `TIMESTAMP WITH LOCAL TIME ZONE`.

The `DATE` data type is not supported. If this *`expr`* is itself a *`datetime_expression`*, then it must be enclosed in parentheses.

Datetimes and intervals can be combined according to the rules defined in Table 2-5. The three combinations that yield datetime values are valid in a datetime expression.

If you specify `AT LOCAL`, then Oracle uses the current session time zone.

The settings for `AT TIME ZONE` are interpreted as follows:

- The string `'[+|-]`*`hh`*`:`*`mi`*`'` specifies a time zone as an offset from UTC. For *`hh`*, specify the number of hours. For *`mi`*, specify the number of minutes.
- `DBTIMEZONE`: Oracle uses the database time zone established (explicitly or by default) during database creation.
- `SESSIONTIMEZONE`: Oracle uses the session time zone established by default or in the most recent `ALTER SESSION` statement.
- *`time_zone_name`*: Oracle returns the *`datetime_value_expr`* in the time zone indicated by *`time_zone_name`*. For a listing of valid time zone region names, query the `V$TIMEZONE_NAMES` dynamic performance view.

  Note:

  Time zone region names are needed by the daylight saving feature. These names are stored in two types of time zone files: one large and one small. One of these files is the default file, depending on your environment and the release of Oracle Database you are using. For more information regarding time zone files and names, see *Oracle Database Globalization Support Guide*.

See Also:

- *Oracle Database Globalization Support Guide* for a complete listing of the time zone region names in both files
- *Oracle Database Reference* for information on the dynamic performance views

- *`expr`*: If *`expr`* returns a character string with a valid time zone format, then Oracle returns the input in that time zone. Otherwise, Oracle returns an error.

Example

The following example converts the datetime value of one time zone to another time zone:

```
SELECT FROM_TZ(CAST(TO_DATE('1999-12-01 11:00:00',
      'YYYY-MM-DD HH:MI:SS') AS TIMESTAMP), 'America/New_York')
   AT TIME ZONE 'America/Los_Angeles' "West Coast Time"
   FROM DUAL;

West Coast Time
------------------------------------------------
01-DEC-99 08.00.00.000000 AM AMERICA/LOS_ANGELES
```

Function Expressions

You can use any built-in SQL function or user-defined function as an expression. Some valid built-in function expressions are:

```
LENGTH('BLAKE')
ROUND(1234.567*43)
SYSDATE
```

> **See Also:**
>
> About SQL Functions ' and Aggregate Functions for information on built-in functions

A user-defined function expression specifies a call to:

- A function in an Oracle-supplied package (see *Oracle Database PL/SQL Packages and Types Reference*)
- A function in a user-defined package or type or in a standalone user-defined function (see About User-Defined Functions)
- A user-defined function or operator (see CREATE OPERATOR , CREATE FUNCTION , and *Oracle Database Data Cartridge Developer's Guide*)

Some valid user-defined function expressions are:

```
circle_area(radius)
payroll.tax_rate(empno)
hr.employees.comm_pct@remote(dependents, empno)
DBMS_LOB.getlength(column_name)
my_function(a_column)
```

In a user-defined function being used as an expression, positional, named, and mixed notation are supported. For example, all of the following notations are correct:

```
CALL my_function(arg1 => 3, arg2 => 4) ...

CALL my_function(3, 4) ...

CALL my_function(3, arg2 => 4) ...
```

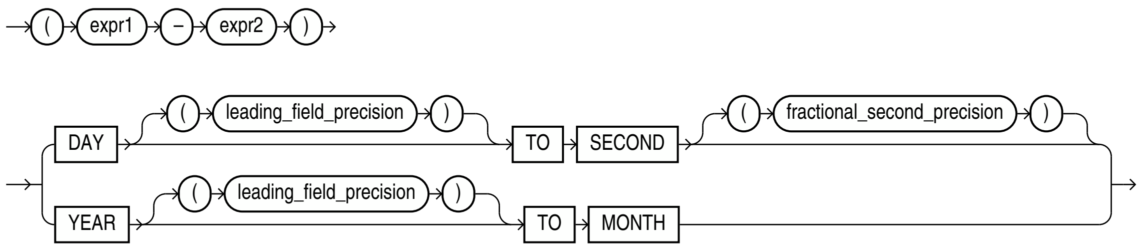
Restriction on User-Defined Function Expressions

You cannot pass arguments of object type or `XMLType` to remote functions and procedures.

Interval Expressions

An interval expression yields a value of `INTERVAL YEAR TO MONTH` or `INTERVAL DAY TO SECOND`.

interval_expression::=



The expressions `expr1` and `expr2` can be any expressions that evaluate to values of data type `DATE`, `TIMESTAMP`, `TIMESTAMP WITH TIME ZONE`, or `TIMESTAMP WITH LOCAL TIME ZONE`.

Datetimes and intervals can be combined according to the rules defined in Table 2-5. The six combinations that yield interval values are valid in an interval expression.

Both *`leading_field_precision`* and *`fractional_second_precision`* can be any integer from 0 to 9. If you omit the *`leading_field_precision`* for either `DAY` or `YEAR`, then Oracle Database uses the default value of 2. If you omit the *`fractional_second_precision`* for second, then the database uses the default value of 6. If the value returned by a query contains more digits that the default precision, then Oracle Database returns an error. Therefore, it is good practice to specify a precision that you know will be at least as large as any value returned by the query.

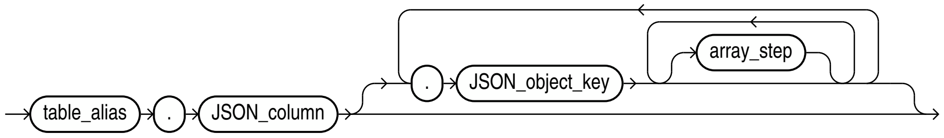
For example, the following statement subtracts the value of the `order_date` column in the sample table `orders` (a datetime value) from the system timestamp (another datetime value) to yield an interval value expression. It is not known how many days ago the oldest order was placed, so the maximum value of 9 for the `DAY` leading field precision is specified:

```
SELECT (SYSTIMESTAMP - order_date) DAY(9) TO SECOND FROM orders
   WHERE order_id = 2458;
```

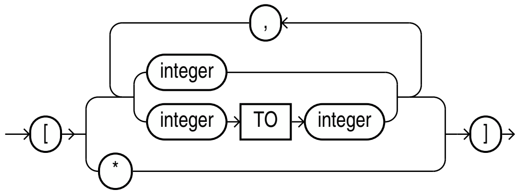
JSON Object Access Expressions

A JSON object access expression is used only when querying a column of JSON data. It yields a character string that contains one or more JSON values found in that data. The syntax for this type of expression is called dot-notation syntax.

json_object_access_expr::=



array_step::=



- For *table_alias*, specify the alias for the table that contains the column of JSON data. This table alias is required and must be assigned to the table elsewhere in the SQL statement.
- For *JSON_column*, specify the name of the column of JSON data. The column must be of data type VARCHAR2, CLOB, BLOB, or JSON.

  Columns can have data of JSON data type if they are the result of JSON generation functions, of JSON_QUERY, or TREAT .

  To identify non JSON type data types you can define the IS JSON check constraint on the column .
- You can optionally specify one or more JSON object keys. The object keys allow you to target specific JSON values in the JSON data. The first *JSON_object_key* must be a case-sensitive match to the key (property) name of an object member in the top level of the JSON data. If the value of that object member is another JSON object, then you can specify a second *JSON_object_key* that matches the key name of a member of that object, and so on. If a JSON array is encountered during any of these iterations, and you do not specify an *array_step*, then the array is implicitly unwrapped and the elements of the array are evaluated using the *JSON_object_key*.
- If the JSON value is an array, then you can optionally specify one or more *array_step* clauses. This allows you to access specific elements of the JSON array.
  - Use *integer* to specify the element at index *integer* in a JSON array. Use *integer* TO *integer* to specify the range of elements between the two index *integer* values, inclusive. If the specified elements exist in the JSON array being evaluated, then the array step results in a match to those elements. Otherwise, the array step does not result in a match. The first element in a JSON array has index 0.
  - Use the asterisk wildcard symbol (*) to specify all elements in a JSON array. If the JSON array being evaluated contains at least one element, then the array step results in a match to all elements in the JSON array. Otherwise, the array step does not result in a match.

A JSON object access expression yields a character string of data type VARCHAR2(4000), which contains the targeted JSON value(s) as follows:

- For a single targeted value, the character string contains that value, whether it is a JSON scalar value, object, or array.
- For multiple targeted values, the character string contains a JSON array whose elements are those values.

If you omit *JSON_object_key*, then the expression yields a character string that contains the JSON data in its entirety. In this case, the character string is of the same data type as the column of JSON data being queried.

A JSON object access expression cannot return a value larger than 4K bytes. If the value surpasses this limit, then the expression returns null. To obtain the actual value, instead use the JSON_QUERY function or the JSON_VALUE function and specify an appropriate return type with the RETURNING clause.

The collation derivation rules for the JSON object access expression are the same as for the JSON_QUERY function.

See Also:

Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules for the JSON_QUERY function

Examples

The following examples use the j_purchaseorder table, which is created in Creating a Table That Contains a JSON Document: Example. This table contains a column of JSON data called po_document. These examples return JSON values from column po_document.

The following statement returns the value of the property with key name PONumber. The value returned, 1600, is a SQL number.

```
SELECT po.po_document.PONumber.number()
  FROM j_purchaseorder po;

PONumber
--------
1600
```

The following statement first targets the property with key name ShippingInstructions, whose value is a JSON object. The statement then targets the property with key name Phone within that object. The statement returns the value of Phone, which is a JSON array.

```
SELECT po.po_document.ShippingInstructions.Phone
  FROM j_purchaseorder po;

SHIPPINGINSTRUCTIONS
-----------------------------------------------------------------------------------
[{"type":"Office","number":"909-555-7307"},{"type":"Mobile","number":"415-555-1234"}]
```

The following statement first targets the property with key name LineItems, whose value is a JSON array. The expression implicitly unwraps the array and evaluates its elements, which are JSON objects. Next, the statement targets the properties with key name Part, within the unwrapped objects, and finds two objects. The statement then targets the properties with key name Description within those two objects and finds string values. Because more than one value is returned, the values are returned as elements of a JSON array.

```
SELECT po.po_document.LineItems.Part.Description
  FROM j_purchaseorder po;

LINEITEMS
-----------------------------------
[One Magic Christmas,Lethal Weapon]
```

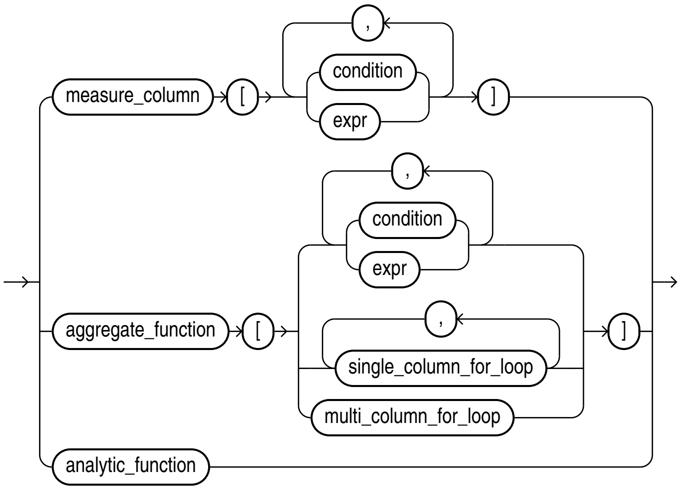
See Also:

Oracle Database JSON Developer’s Guide for more information on querying JSON data using dot-notation syntax

Model Expressions

A model expression is used only in the `model_clause` of a `SELECT` statement and then only on the right-hand side of a model rule. It yields a value for a cell in a measure column previously defined in the `model_clause`. For additional information, refer to *model_clause*.

***model_expression*::=**



When you specify a measure column in a model expression, any conditions and expressions you specify must resolve to single values.

When you specify an aggregate function in a model expression, the argument to the function is a measure column that has been previously defined in the `model_clause`. An aggregate function can be used only on the right-hand side of a model rule.

Specifying an analytic function on the right-hand side of the model rule lets you express complex calculations directly in the `model_clause`. The following restrictions apply when using an analytic function in a model expression:

- Analytic functions can be used only in an `UPDATE` rule.
- You cannot specify an analytic function on the right-hand side of the model rule if the left-hand side of the rule contains a `FOR` loop or an `ORDER BY` clause.
- The arguments in the `OVER` clause of the analytic function cannot contain an aggregate.
- The arguments before the `OVER` clause of the analytic function cannot contain a cell reference.

> **See Also:**
>
> The MODEL clause: Examples for an example of using an analytic function on the right-hand side of a model rule

When `expr` is itself a model expression, it is referred to as a **nested cell reference**. The following restrictions apply to nested cell references:

- Only one level of nesting is allowed.
- A nested cell reference must be a single-cell reference.
- When AUTOMATIC ORDER is specified in the *model_rules_clause*, a nested cell reference can be used on the left-hand side of a model rule only if the measures used in the nested cell reference remain static.

The model expressions shown below are based on the *model_clause* of the following SELECT statement:

```
SELECT country,prod,year,s
  FROM sales_view_ref
  MODEL
    PARTITION BY (country)
    DIMENSION BY (prod, year)
    MEASURES (sale s)
    IGNORE NAV
    UNIQUE DIMENSION
    RULES UPSERT SEQUENTIAL ORDER
    (
      s[prod='Mouse Pad', year=2000] =
        s['Mouse Pad', 1998] + s['Mouse Pad', 1999],
      s['Standard Mouse', 2001] = s['Standard Mouse', 2000]
    )
  ORDER BY country, prod, year;
```

The following model expression represents a single cell reference using symbolic notation. It represents the sales of the Mouse Pad for the year 2000.

```
s[prod='Mouse Pad',year=2000]
```

The following model expression represents a multiple cell reference using positional notation, using the CV function. It represents the sales of the current value of the dimension column prod for the year 2001.

```
s[CV(prod), 2001]
```

The following model expression represents an aggregate function. It represents the sum of sales of the Mouse Pad for the years between the current value of the dimension column year less two and the current value of the dimension column year less one.

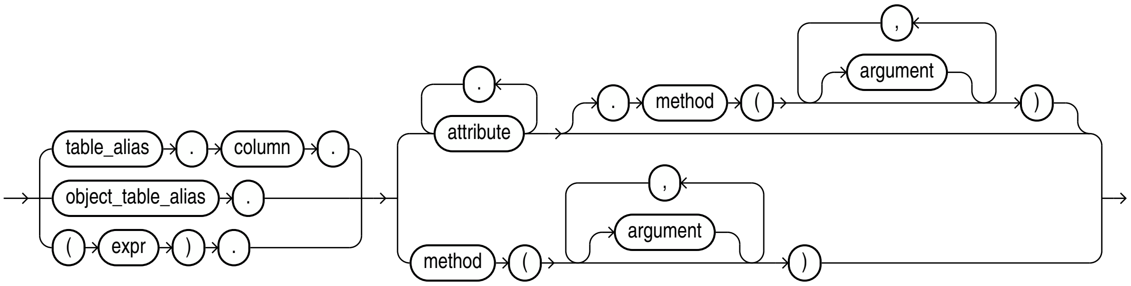
```
SUM(s)['Mouse Pad',year BETWEEN CV()-2 AND CV()-1]
```

> **See Also:**
>
> CV and *model_clause*

Object Access Expressions

An object access expression specifies attribute reference and method invocation.

***object_access_expression*::=**



The column parameter can be an object or `REF` column. If you specify *`expr`*, then it must resolve to an object type.

When a type's member function is invoked in the context of a SQL statement, if the `SELF` argument is null, Oracle returns null and the function is not invoked.

Examples

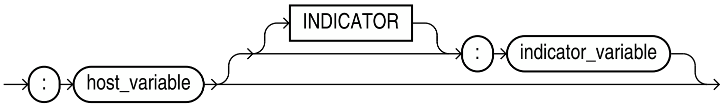
The following example creates a table based on the sample `oe.order_item_typ` object type, and then shows how you would update and select from the object column attributes.

```
CREATE TABLE short_orders (
   sales_rep VARCHAR2(25), item order_item_typ);

UPDATE short_orders s SET sales_rep = 'Unassigned';

SELECT o.item.line_item_id, o.item.quantity FROM short_orders o;
```

Placeholder Expressions

A placeholder expression provides a location in a SQL statement for which a third-generation language bind variable will provide a value. You can specify the placeholder expression with an optional indicator variable. This form of expression can appear only in embedded SQL statements or SQL statements processed in an Oracle Call Interface (OCI) program.

***placeholder_expression*::=**



Some valid placeholder expressions are:

```
:employee_name INDICATOR :employee_name_indicator_var
:department_location
```

See Also:

Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules for the placeholder expression with a character data type

Scalar Subquery Expressions

A scalar subquery expression is a subquery that returns exactly one column value from one row. The value of the scalar subquery expression is the value of the select list item of the subquery. If the subquery returns 0 rows, then the value of the scalar subquery expression is `NULL`. If the subquery returns more than one row, then Oracle returns an error.

You can use a scalar subquery expression in most syntax that calls for an expression (*`expr`*). In all cases, a scalar subquery must be enclosed in its own parentheses, even if its syntactic location already positions it within parentheses (for example, when the scalar subquery is used as the argument to a built-in function).

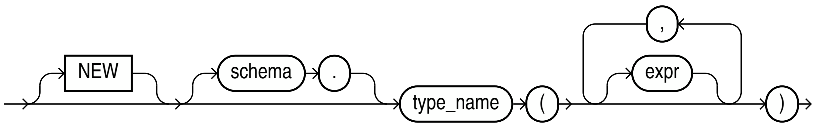
Scalar subqueries are not valid expressions in the following places:

- As default values for columns
- As hash expressions for clusters
- In the `RETURNING` clause of DML statements
- As the basis of a function-based index
- In `CHECK` constraints
- In `GROUP BY` clauses
- In statements that are unrelated to queries, such as `CREATE PROFILE`

Type Constructor Expressions

A type constructor expression specifies a call to a constructor method. The argument to the type constructor is any expression. Type constructors can be invoked anywhere functions are invoked.

***type_constructor_expression*::=**



The `NEW` keyword applies to constructors for object types but not for collection types. It instructs Oracle to construct a new object by invoking an appropriate constructor. The use of the `NEW` keyword is optional, but it is good practice to specify it.

If *`type_name`* is an **object type**, then the expressions must be an ordered list, where the first argument is a value whose type matches the first attribute of the object type, the second argument is a value whose type matches the second attribute of the object type, and so on. The total number of arguments to the constructor must match the total number of attributes of the object type.

If *type_name* is a **varray** or **nested table type**, then the expression list can contain zero or more arguments. Zero arguments implies construction of an empty collection. Otherwise, each argument corresponds to an element value whose type is the element type of the collection type.

Restriction on Type Constructor Invocation

In an invocation of a type constructor method, the number of parameters (*expr*) specified cannot exceed 999, even if the object type has more than 999 attributes. This limitation applies only when the constructor is called from SQL. For calls from PL/SQL, the PL/SQL limitations apply.

> **See Also:**
>
> *Oracle Database Object-Relational Developer's Guide* for additional information on constructor methods and *Oracle Database PL/SQL Language Reference* for information on PL/SQL limitations on calls to type constructors

Expression Example

This example uses the `cust_address_typ` type in the sample `oe` schema to show the use of an expression in the call to a constructor method (the PL/SQL is shown in italics):

```
CREATE TYPE address_book_t AS TABLE OF cust_address_typ;
DECLARE
   myaddr cust_address_typ := cust_address_typ(
     '500 Oracle Parkway', 94065, 'Redwood Shores', 'CA','USA');
   alladdr address_book_t := address_book_t();
BEGIN
   INSERT INTO customers VALUES (
      666999, 'Joe', 'Smith', myaddr, NULL, NULL, NULL, NULL,
      NULL, NULL, NULL, NULL, NULL, NULL, NULL);
END;
/
```

Subquery Example

This example uses the `warehouse_typ` type in the sample schema `oe` to illustrate the use of a subquery in the call to the constructor method.

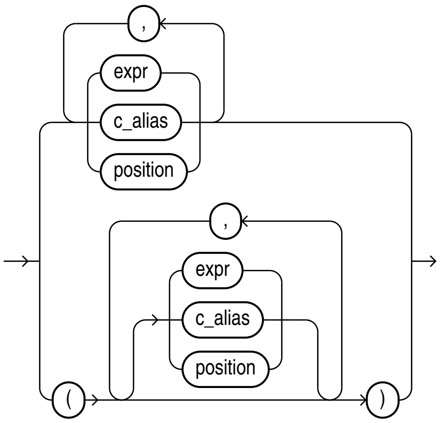
```
CREATE TABLE warehouse_tab OF warehouse_typ;

INSERT INTO warehouse_tab
   VALUES (warehouse_typ(101, 'new_wh', 201));

CREATE TYPE facility_typ AS OBJECT (
   facility_id NUMBER,
   warehouse_ref REF warehouse_typ);

CREATE TABLE buildings (b_id NUMBER, building facility_typ);

INSERT INTO buildings VALUES (10, facility_typ(102,
   (SELECT REF(w) FROM warehouse_tab w
      WHERE warehouse_name = 'new_wh')));

SELECT b.b_id, b.building.facility_id "FAC_ID",
   DEREF(b.building.warehouse_ref) "WH" FROM buildings b;
```

```
      B_ID     FAC_ID WH(WAREHOUSE_ID, WAREHOUSE_NAME, LOCATION_ID)
---------- ---------- --------------------------------------------
        10        102 WAREHOUSE_TYP(101, 'new_wh', 201)
```

Expression Lists

An expression list is a combination of other expressions.

***expression_list*::=**



Expression lists can appear in comparison and membership conditions and in `GROUP BY` clauses of queries and subqueries. An expression lists in a comparision or membership condition is sometimes referred to as a **row value constructor** or **row constructor**.

Comparison and membership conditions appear in the conditions of `WHERE` clauses. They can contain either one or more comma-delimited expressions or one or more sets of expressions where each set contains one or more comma-delimited expressions. In the latter case (multiple sets of expressions):

- Each set is bounded by parentheses
- Each set must contain the same number of expressions
- The number of expressions in each set must match the number of expressions before the operator in the comparison condition or before the `IN` keyword in the membership condition.

A comma-delimited list of expressions can contain no more than 65,535 expressions. A comma-delimited list of sets of expressions can contain any number of sets, but each set can contain no more than 1000 expressions.

The following are some valid expression lists in conditions:

```
(10, 20, 40)
('SCOTT', 'BLAKE', 'TAYLOR')
( ('Guy', 'Himuro', 'GHIMURO'),('Karen', 'Colmenares', 'KCOLMENA') )
```

In the third example, the number of expressions in each set must equal the number of expressions in the first part of the condition. For example:

```
SELECT * FROM employees
  WHERE (first_name, last_name, email) IN
  (('Guy', 'Himuro', 'GHIMURO'),('Karen', 'Colmenares', 'KCOLMENA'))
```

See Also:

Comparison Conditions and IN Condition conditions

In a simple GROUP BY clause, you can use either the upper or lower form of expression list:

```
SELECT department_id, MIN(salary) min, MAX(salary) max FROM employees
   GROUP BY department_id, salary
   ORDER BY department_id, min, max;

SELECT department_id, MIN(salary) min, MAX(salary) max FROM employees
   GROUP BY (department_id, salary)
   ORDER BY department_id, min, max;
```

In ROLLUP, CUBE, and GROUPING SETS clauses of GROUP BY clauses, you can combine individual expressions with sets of expressions in the same expression list. The following example shows several valid grouping sets expression lists in one SQL statement:

```
SELECT
prod_category, prod_subcategory, country_id, cust_city, count(*)
   FROM  products, sales, customers
   WHERE sales.prod_id = products.prod_id
   AND sales.cust_id=customers.cust_id
   AND sales.time_id = '01-oct-00'
   AND customers.cust_year_of_birth BETWEEN 1960 and 1970
GROUP BY GROUPING SETS
  (
   (prod_category, prod_subcategory, country_id, cust_city),
   (prod_category, prod_subcategory, country_id),
   (prod_category, prod_subcategory),
    country_id
  )
ORDER BY prod_category, prod_subcategory, country_id, cust_city;
```

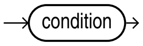
See Also:

SELECT

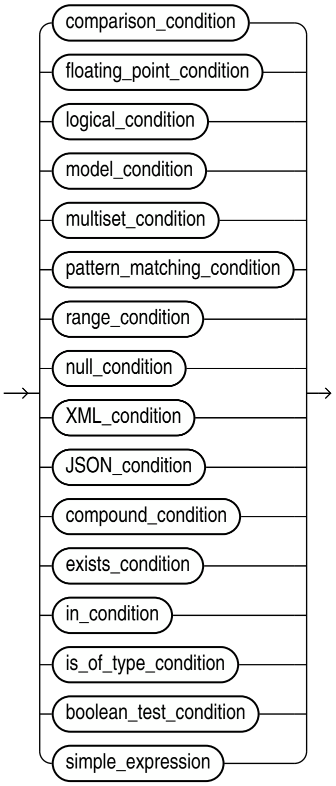
BOOLEAN Expressions

You can now use boolean value expressions within SQL expressions wherever an expression appears in SQL syntax.

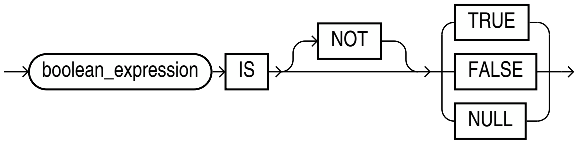
***boolean_expression*::=**



condition::=



boolean_test_condition::=



Use `boolean_expression` to evalute the input and return one of the following boolean values :

- `IS TRUE`
- `IS NOT TRUE`
- `IS FALSE`
- `IS NOT FALSE`
- `IS NULL`
- `IS NOT NULL`

See Also:

About SQL Expressions