INTERNATIONAL STANDARD

**ISO/IEC 19075-9**

First edition
2022-10

# Information technology — Guidance for the use of database language SQL —

## Part 9: Online analytic processing (OLAP) capabilities (Guide/OLAP)

*Langages de bases de données utilisés dans les technologies de l'information — Recommandations pour l'utilisation du langage de base de données SQL —*

*Partie 9: Capacités de traitement analytique en ligne (OLAP), (Guide/OLAP)*



Reference number
ISO/IEC 19075-9:2022(E)



ISO copyright office
CP 401 • Ch. de Blandonnet 8
CH-1214 Vernier, Geneva
Phone: +41 22 749 01 11
Email: copyright@iso.org
Website: www.iso.org

Published in Switzerland

# Contents

Page

Foreword . . . vii
Introduction . . . ix
**1 Scope . . . 1**
**2 Normative references . . . 2**
**3 Terms and definitions . . . 3**
**4 Example data . . . 4**
4.1 Introduction to example data . . . 4
4.2 Table sales history . . . 4
4.3 Table stock1 . . . 5
4.4 Table stocks . . . 6
4.5 Table homes . . . 6
**5 Windows . . . 8**
5.1 Introduction to windows . . . 8
5.2 Window definitions . . . 8
5.2.1 Introduction to window definitions . . . 8
5.2.2 Window partitioning . . . 9
5.2.3 Window ordering . . . 10
5.2.3.1 Introduction to window ordering . . . 10
5.2.3.2 Null ordering and treatment . . . 11
5.2.4 Window frames . . . 12
5.2.4.1 Introduction to window frames . . . 12
5.2.4.2 Physical window frames . . . 13
5.2.4.3 Logical window frames . . . 15
5.2.4.3.1 Introduction to logical window frames . . . 15
5.2.4.3.2 RANGE window frames . . . 15
5.2.4.3.3 GROUPS window frames . . . 16
5.2.4.4 Window frame exclusions . . . 17
5.3 Explicit vs. implicit window definitions . . . 18
5.4 Multiple window definitions . . . 19
**6 Window functions . . . 20**
6.1 Introduction to window functions . . . 20
6.2 Rank functions . . . 20
6.3 Distribution functions . . . 21
6.4 Row number function . . . 22
6.5 Window aggregate functions . . . 23
6.6 Ntile function . . . 26
6.7 LEAD and LAG functions . . . 27
6.8 FIRST_VALUE and LAST_VALUE functions . . . 29

6.9 NTH_VALUE function. . . . 30
6.10 Null treatment. . . . 32
**7 Nested window functions. . . . 33**
7.1 Introduction to nested window functions. . . . 33
7.2 Row markers. . . . 34
7.3 Offsets. . . . 35
7.4 FRAME_ROW. . . . 36
7.5 Nested ROW_NUMBER function. . . . 37
7.6 Effects of EXCLUDE. . . . 38
**8 Enhanced aggregate functions. . . . 40**
8.1 Introduction to enhanced aggregate functions. . . . 40
8.2 Unary statistical aggregate functions. . . . 40
8.3 Binary statistical aggregate functions. . . . 41
8.4 Hypothetical rank and distribution aggregate functions. . . . 44
8.5 Inverse distribution functions. . . . 45
**Bibliography. . . . 47**
**Index. . . . 48**



 

# Tables

## Table

**Page**


1 Table sales_history. . . . 4
2 Table stock1. . . . 5
3 Table stocks. . . . 6
4 Table homes. . . . 6
5 Result of window clause. . . . 10
6 Result of window clause ordering. . . . 11
7 Result of physical window frame, UNBOUNDED PRECEDING to CURRENT ROW. . . . 13
8 Result of physical window frame, 2 PRECEDING to 1 FOLLOWING. . . . 14
9 Result of logical window frame with RANGE. . . . 15
10 Result of logical window frame with GROUPS. . . . 16
11 Result of window frame exclusion 1. . . . 17
12 Result of window frame exclusion 2. . . . 18
13 Result of RANK and DENSE_RANK function. . . . 21
14 Result of PERCENT_RANK and CUME_DIST functions. . . . 22
15 Result of ROW_NUMBER function. . . . 23
16 Result of aggregate function (SUM) ordered. . . . 24
17 Result of aggregate function (SUM) unordered. . . . 24
18 Result of aggregate function (AVG). . . . 25
19 Result of aggregate function (NTILE) with partitioned query. . . . 26
20 Result of aggregate function (NTILE) in non-partitioned query. . . . 27
21 Result of aggregate function (LEAD). . . . 28
22 Result of aggregate function (LAG). . . . 28
23 Result of aggregate function (FIRST_VALUE). . . . 29
24 Result of aggregate function (LAST_VALUE). . . . 30
25 Result of aggregate function (NTH_VALUE). . . . 31
26 Result of row markers. . . . 34
27 Result of row markers (offsets). . . . 35
28 Result of window frames with row markers. . . . 36
29 Result of EXCLUDE. . . . 38
30 Result of hypothetical aggregate functions. . . . 44
31 Result of inverse distribution functions. . . . 45
32 Result of inverse distribution functions with ordering. . . . 46

# Examples

## Example

Page

1 Window clause. . . . 9
2 Window clause ordering. . . . 11
3 Physical window frame, UNBOUNDED PRECEDING to CURRENT ROW. . . . 13
4 Physical window frame, 2 PRECEDING to 1 FOLLOWING. . . . 14
5 Logical window frame with RANGE. . . . 15
6 Logical window frame with GROUPS. . . . 16
7 Window frame exclusion 1. . . . 17
8 Window frame exclusion 2. . . . 17
9 Explicit window definition. . . . 18
10 Implicit window definition. . . . 19
11 Multiple window definitions. . . . 19
12 Rank functions with explicit window. . . . 20
13 Rank functions with implicit window. . . . 21
14 Distribution functions. . . . 22
15 Row number function. . . . 23
16 Window aggregate function (SUM) ordered. . . . 23
17 Window aggregate function (SUM) unordered. . . . 24
18 Window aggregate moving average. . . . 25
19 NTILE in partitioned query. . . . 26
20 NTILE in non-partitioned query. . . . 26
21 LEAD function. . . . 28
22 LAG function. . . . 28
23 FIRST_VALUE function. . . . 29
24 LAST_VALUE function. . . . 30
25 NTH_VALUE function usage. . . . 31
26 Equivalent NTH_VALUE function usage. . . . 31
27 Null treatment with LEAD function. . . . 32
28 Null treatment with FIRST_VALUE function. . . . 32
29 Null treatment with LAST_VALUE function. . . . 32
30 Q1: CASE expression in a window query. . . . 33
31 Q2: Complex join. . . . 33
32 Q3: VALUE_OF function usage. . . . 34
33 Q4: frame_row and current_row in value_of function. . . . 37
34 Weight function. . . . 38
35 Hypothetical aggregate function. . . . 44
36 Inverse distribution function. . . . 45
37 Inverse distribution function ordered. . . . 46

# Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives or www.iec.ch/members_experts/refdocs).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents) or the IEC list of patent declarations received (see patents.iec.ch).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see www.iso.org/iso/foreword.html. In the IEC, see www.iec.ch/understanding-standards.

This document was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 32, *Data management and interchange*.

This first edition of ISO/IEC 19075-9 cancels and replaces ISO/IEC TR 19075-9:2020.

This document is intended to be used in conjunction with the following editions of the parts of the ISO/IEC 9075 series:

— ISO/IEC 9075-1, sixth edition or later;

— ISO/IEC 9075-2, sixth edition or later;

— ISO/IEC 9075-3, sixth edition or later;

— ISO/IEC 9075-4, seventh edition or later;

— ISO/IEC 9075-9, fifth edition or later;

— ISO/IEC 9075-10, fifth edition or later;

— ISO/IEC 9075-11, fifth edition or later;

— ISO/IEC 9075-13, fifth edition or later;

— ISO/IEC 9075-14, sixth edition or later;

— ISO/IEC 9075-15, second edition or later;

— ISO/IEC 9075-16, first edition or later.

A list of all parts in the ISO/IEC 19075 series can be found on the ISO and IEC websites.

Any feedback or questions on this document should be directed to the user's national standards body. A complete listing of these bodies can be found at www.iso.org/members.html and www.iec.ch/national-committees.

# Introduction

This document discusses the syntax and semantics for including online analytic processing (OLAP) capabilities in SQL, as defined in ISO/IEC 9075-2.

The organization of this document is as follows:

1) Clause 1, "Scope", specifies the scope of this document.

2) Clause 2, "Normative references", identifies standards that are referenced as part of requirements by this document.

3) Clause 3, "Terms and definitions", defines the terms and definitions used in this document.

4) Clause 5, "Windows", discusses Feature T611, "Elementary OLAP operations" and Feature T612, "Advanced OLAP operations", introducing the concept of a window in an SQL query.

5) Clause 6, "Window functions", further discusses Feature T611, "Elementary OLAP operations" and Feature T612, "Advanced OLAP operations", as well as Feature T614, "NTILE function", Feature T615, "LEAD and LAG functions", Feature T616, "Null treatment option for LEAD and LAG functions", Feature T617, "FIRST_VALUE and LAST_VALUE functions", and Feature T618, "NTH_VALUE function".

6) Clause 7, "Nested window functions", discusses the additional window functionality in Feature T619, "Nested window functions".

7) Clause 8, "Enhanced aggregate functions", discusses Feature T621, "Enhanced numeric functions" and its introduction of enhanced aggregate functions in SQL.

# Information technology — Guidance for the use of database language SQL —

Part 9:
# Online analytic processing (OLAP) capabilities (Guide/OLAP)

## 1 Scope

This document discusses the syntax and semantics for including online analytic processing (OLAP) capabilities in SQL, as defined in ISO/IEC 9075-2.

It discusses the following features regarding OLAP capabilities of the SQL language:

— Feature T611, “Elementary OLAP operations”,

— Feature T612, “Advanced OLAP operations”,

— Feature T614, “NTILE function”,

— Feature T615, “LEAD and LAG functions”,

— Feature T616, “Null treatment option for LEAD and LAG functions”,

— Feature T617, “FIRST_VALUE and LAST_VALUE functions”,

— Feature T618, “NTH_VALUE function”,

— Feature T619, “Nested window functions”,

— Feature T620, “WINDOW clause: GROUPS option”,

— Feature T621, “Enhanced numeric functions”

## 2 Normative references

There are no normative references in this document.

# 3 Terms and definitions

No terms and definitions are listed in this document.

ISO and IEC maintain terminological databases for use in standardization at the following addresses:

— IEC Electropedia: available at http://www.electropedia.org/

— ISO Online browsing platform: available at http://www.iso.org/obp

# 4 Example data

## 4.1 Introduction to example data

The examples in this document are based on several tables.

The order in which the rows of all sample tables are displayed is immaterial.

## 4.2 Table sales history

Table 1, "Table sales_history", contains information on a business spread over several territories with total sales accumulated monthly in each territory. Table 1, "Table sales_history", shows sample data for Subclause 4.2, "Table sales history":

**Table 1 — Table sales_history**

| Territory | Month | Sales |
|---|---|---|
| East | 199812 | 11 |
| West | 199811 | 12 |
| West | 199901 | 11 |
| East | 199811 | 4 |
| East | 199810 | 10 |
| West | 199810 | 8 |
| East | 199902 | 10 |
| East | 199901 | 7 |
| West | 199812 | 7 |
| West | 199902 | 6 |



SQL to create and populate Subclause 4.2, "Table sales history".

```
CREATE TABLE Sales_History
(Territory CHARACTER (10),
 Month INTEGER,
 Sales INTEGER)

INSERT INTO Sales_History VALUES ('East', 199812, 11)
INSERT INTO Sales_History VALUES ('West', 199811, 12)
INSERT INTO Sales_History VALUES ('West', 199901, 11)
INSERT INTO Sales_History VALUES ('East', 199811, 4)
INSERT INTO Sales_History VALUES ('East', 199810, 10)
INSERT INTO Sales_History VALUES ('West', 199810, 8)
INSERT INTO Sales_History VALUES ('East', 199902, 10)
INSERT INTO Sales_History VALUES ('East', 199901, 7)
```

```
INSERT INTO Sales_History VALUES ('West', 199812, 7)
INSERT INTO Sales_History VALUES ('West', 199902, 6)
```

## 4.3 Table stock1

The next examples are two variants of a stock table containing information on stock transactions for a particular account. Columns in Table 2, "Table stock1", include transaction ID, trade day, and type, as well as the share amount and ticker symbol. Subclause 4.4, "Table stocks", covers the columns ticker, tradeday, and price.

**Table 2 — Table stock1**

| Acno | Tid | Tradeday | TType | Amount | Ticker |
|---|---|---|---|---|---|
| 123 | 1 | 1 | buy | 1000 | csco |
| 123 | 2 | 1 | buy | 400 | inpr |
| 123 | 3 | 2 | buy | 2000 | symc |
| 123 | 4 | 2 | buy | 1200 | csco |
| 123 | 5 | 2 | buy | 500 | inpr |
| 123 | 6 | 4 | buy | 200 | csco |
| 123 | 7 | 4 | buy | 100 | csco |
| 123 | 9 | 5 | buy | 400 | inpr |
| 123 | 10 | 5 | buy | 200 | goog |
| 123 | 11 | 5 | buy | 1000 | inpr |
| 123 | 12 | 5 | buy | 4000 | inpr |
| 123 | 13 | 8 | buy | 2000 | hpq |

SQL to create and populate Table 2, "Table stock1".

```
CREATE TABLE Stock1
(Acno INTEGER,
 Tid INTEGER,
 Tradeday INTEGER,
 TType CHARACTER (10),
 Amount INTEGER,
 Ticker CHARACTER (10))

INSERT INTO Stock1 VALUES (123, 1, 1, 'buy', 1000, 'csco')
INSERT INTO Stock1 VALUES (123, 2, 1, 'buy', 400, 'inpr')
INSERT INTO Stock1 VALUES (123, 3, 2, 'buy', 2000, 'symc')
INSERT INTO Stock1 VALUES (123, 4, 2, 'buy', 1200, 'csco')
INSERT INTO Stock1 VALUES (123, 5, 2, 'buy', 500, 'inpr')
INSERT INTO Stock1 VALUES (123, 6, 4, 'buy', 200, 'csco')
INSERT INTO Stock1 VALUES (123, 7, 4, 'buy', 100, 'csco')
INSERT INTO Stock1 VALUES (123, 9, 5, 'buy', 400, 'inpr')
INSERT INTO Stock1 VALUES (123, 10, 5, 'buy', 200, 'goog')
INSERT INTO Stock1 VALUES (123, 11, 5, 'buy', 1000, 'inpr')
INSERT INTO Stock1 VALUES (123, 12, 5, 'buy', 4000, 'inpr')
INSERT INTO Stock1 VALUES (123, 13, 8, 'buy', 2000, 'hpq')
```

## 4.4 Table stocks

**Table 3 — Table stocks**

| Ticker | Tradeday | Price |
|---|---|---|
| ZYX | 1 | 10 |
| ZYX | 2 | 11 |
| ZYX | 3 | 12 |
| ZYX | 4 | 12 |
| ZYX | 5 | 12 |
| ZYX | 6 | 11 |
| ZYX | 7 | 12 |
| ZYX | 8 | 12 |

SQL to create and populate Table 3, "Table stocks".

```
CREATE TABLE Stocks
(Ticker CHARACTER (10),
 Tradeday INTEGER,
 Price INTEGER)

INSERT INTO Stocks VALUES ('ZYX', 1, 10)
INSERT INTO Stocks VALUES ('ZYX', 2, 11)
INSERT INTO Stocks VALUES ('ZYX', 3, 12)
INSERT INTO Stocks VALUES ('ZYX', 4, 12)
INSERT INTO Stocks VALUES ('ZYX', 5, 12)
INSERT INTO Stocks VALUES ('ZYX', 6, 11)
INSERT INTO Stocks VALUES ('ZYX', 7, 12)
INSERT INTO Stocks VALUES ('ZYX', 8, 12)
```

## 4.5 Table homes

The final example is Table 4, "Table homes", containing data concerning house prices and locations.

**Table 4 — Table homes**

| Area | Address | Price |
|---|---|---|
| Uptown | 15 Peekaboo St. | 456000 |
| Uptown | 27 Primrose Path | 341000 |
| Uptown | 44 Shady Lane | 341000 |
| Uptown | 23301 Highway 61 | 244000 |
| Uptown | 34 Desolation Rd. | 244000 |
| Uptown | 77 Sunset Strip | 102000 |

| Area | Address | Price |
|---|---|---|
| Downtown | 72 Easy St. | 509000 |
| Downtown | 29 Wong Way | 201000 |
| Downtown | 45 Diamond Lane | 201000 |
| Downtown | 76 Blind Alley | 201000 |
| Downtown | 15 Tern Pike | 199000 |
| Downtown | 444 Kanga Rua | 102000 |

SQL to create and populate Table 4, "Table homes".

```
CREATE TABLE Homes
(Area CHARACTER (10),
 Address CHARACTER (20),
 Price INTEGER)

INSERT INTO Homes VALUES ('Uptown', '15 Peekaboo St.', 456000)
INSERT INTO Homes VALUES ('Uptown', '27 Primrose Path', 341000)
INSERT INTO Homes VALUES ('Uptown', '44 Shady Lane', 341000)
INSERT INTO Homes VALUES ('Uptown', '23301 Highway 61', 244000)
INSERT INTO Homes VALUES ('Uptown', '34 Desolation Rd.', 244000)
INSERT INTO Homes VALUES ('Uptown', '77 Sunset Strip', 102000)
INSERT INTO Homes VALUES ('Downtown', '72 Easy St.', 509000)
INSERT INTO Homes VALUES ('Downtown', '29 Wong Way', 201000)
INSERT INTO Homes VALUES ('Downtown', '45 Diamond Lane', 201000)
INSERT INTO Homes VALUES ('Downtown', '76 Blind Alley', 201000)
INSERT INTO Homes VALUES ('Downtown', '15 Tern Pike', 199000)
INSERT INTO Homes VALUES ('Downtown', '444 Kanga Rua', 102000)
```

# 5 Windows

## 5.1 Introduction to windows

SQL in Feature T611, "Elementary OLAP operations" and Feature T612, "Advanced OLAP operations" of ISO/IEC 9075-2 adds support for online analytical processing (OLAP). The extensions are parts of the SELECT command.

OLAP is concerned with data aggregation across grouping criteria to generate values such as subtotals and totals on multiple levels. Grouping criteria are often called dimensions. OLAP is based on the concept of multiple dimensions and navigation across the aggregation levels as well as the accumulated data. OLAP is used in applications such as analytics and reporting. It may be used in iterative fashions trying out different grouping criteria and different subsets of the data to be analyzed.

To deal with these requirements, the features mentioned above introduce a "WINDOW" facility that may present aggregated content as a rolling window, ordered and grouped by the specified criteria. See the example in Subclause 5.2.2, "Window partitioning", Table 5, "Result of window clause".

The data may be treated like other result sets, and/or may be used for further processing.

A query contains a select list and a table expression. The table expression produces a result set; call it *RT*. The select list is evaluated by applying its expressions to each row in *RT*. Without the OLAP features, the select list expressions may only "see" that one current row, making it impossible to compute values that rely on values from other rows in *RT*. Such computations may be simulated only by arranging for the necessary values to be included as additional columns in *RT*, which may be inconvenient or impossible.

The concept of the windowed table alleviates this limitation by adding windows to the result of the table expression. One way to think of a window is to imagine that it is a transient copy of *RT*, including an indication of the current row. (This is just a conceptual device; an SQL-implementation need not actually copy *RT*.) This transient copy may be logically re-arranged according to a sort ordering, a partitioning, or both; it may also be limited to a subset of rows, via window framing. These rearrangements of the window's rows take place without affecting *RT* itself. Multiple windows may exist, each with its own independently applied ordering, partitioning, and/or framing specifications. Windows may be defined either in the new window clause, or in-line in individual window function specifications in the select list.

Partitioning and/or ordering may be used to compute such results as ranking, Ntiles, and other analytic functions. The frame specifies which rows of a partition, relative to the current row, should participate in the calculation of an OLAP function. Through frames, windows support such important OLAP capabilities as cumulative sums and moving averages.

Ordering in windows is specified with the same sort specification list used by cursors and elsewhere in the SELECT statement, and with the same semantics. Ordering as enhanced for the OLAP capabilities of SQL also includes user specified control of the ordering of nulls. Although ordering of rows may be non-deterministic within a window, the same non-deterministic ordering is used in windows that have equivalent partitioning and ordering clauses specified.

## 5.2 Window definitions

### 5.2.1 Introduction to window definitions

The <window clause> is an additional syntax element of the query expression and, if specified, follows the <from clause>, <where clause>, <group by clause>, and <having clause>. As with the other clauses of the query expression, the window clause applies to the result of the preceding clauses. The window clause consists of a comma separated list of window definitions. Each of these has a name, for reference

by OLAP functions in the select list, and a window specification. For convenience, a window definition may also be coded inline in the OLAP function specification. Clause 6, "Window functions", describes the various OLAP functions available for use, and which combinations of window specification detail clauses are permitted for each.

The window specification may contain any or all of a partitioning, ordering, and frame definition. For example:

```
WINDOW tms AS
   (PARTITION BY territory -- window partitioning
    ORDER BY month, sales) -- window ordering
```

A window function may refer to the defined window by name, for instance:

```
SELECT RANK() OVER tms AS the_rank, ...
FROM ...
WINDOW tms AS ... as above ...
```

Or, implicitly by specifying the window definition directly in-line:

```
SELECT RANK() OVER (PARTITION BY territory
                    ORDER BY month, sales) AS the_rank, ...
FROM ...
```

which has the identical result.

### 5.2.2 Window partitioning

The optional "window partition clause" specifies a partitioning of the result set generated by the preceding from, where, group by and having clauses. Like the group by clause, the window partition clause is a comma-separated list of column references used to group rows for subsequent processing. However, unlike the group by clause, each input row to a window partitioning is retained in the result set. This permits the introduction of analytical functions that operate on the individual rows of a partition. The "collate clause" option allows character columns to be partitioned based on a named collation. If there is no window partition clause, then the entire result set of the containing query constitutes a single partition.

Notice that although the window partition clause is similar to the group by clause, it is not the same thing. The difference is that the grouping specified by a group by clause collapses each group to a single row in its result set. The partitioning specified by a window partition clause does not collapse the partitions to a single row. Rather, the window partition of a row *R* is the collection of rows that are not distinct from *R*, for all columns enumerated in the window partitioning clause.

Example 1, "Window clause", shows the effect of partitioning Sales_history using the Territory column in the definition.

**Example 1 — Window clause**

```
SELECT Territory, Month, Sales FROM Sales_history
     WINDOW w_eg1 AS (PARTITION BY Territory)
```

— `WINDOW w_eg1` identifies the window definition so that it may be referenced in one or more OLAP functions elsewhere in the query (`w_eg1` is the window's name).

— `PARTITION BY` introduces the partitioning. A partitioning is simply a list of one or more columns on which the data is partitioned.

**Table 5 — Result of window clause**

| Territory | Month | Sales | |
|---|---|---|---|
| East | 199901 | 7 | ⎫ |
| East | 199811 | 4 | ⎪ |
| East | 199810 | 10 | ⎬ "East" partition |
| East | 199812 | 11 | ⎪ |
| East | 199902 | 10 | ⎭ |
| West | 199811 | 12 | ⎫ |
| West | 199810 | 8 | ⎪ |
| West | 199902 | 6 | ⎬ "West" partition |
| West | 199901 | 11 | ⎪ |
| West | 199812 | 7 | ⎭ |

The rows of Table 5, "Result of window clause", are clustered to show the effect of the partitioning, but in reality, the result rows may still appear in any sequence. If it is desired that a result set be ordered on one or more columns, the containing query expression states that intent through the incorporation of an order by clause.

### 5.2.3 Window ordering

#### 5.2.3.1 Introduction to window ordering

The next optional element of a window definition is the "window order clause". It consists of a "sort specification list" that is syntactically the same as the one found in an "order by" clause of the query expression.

Whether in a window or a query expression, a sort specification list specifies an ordering of rows. The difference is that, in a query expression, the ordering determines the sequence of rows in the result set of the query expression. In a window, the ordering helps to determine the value of order-dependent OLAP functions such as ranking. Another difference is that the query expression ordering applies to all result rows, whereas a window ordering is applied to each partition separately.

As the rows of each partition are ordered, it may turn out that multiple rows are peers; that is, they have the same values in each of the elements of the sort specification list according to ordering semantics. A window ordering group is a maximal set of rows (perhaps just one) in a partition that are peers according to the window ordering. Some analytic functions (RANK, for instance) operate on all rows in an ordering group or set of ordering groups, whereas others (such as NTH_VALUE) may operate on individual rows regardless of their participation in an ordering group. The ordering of rows within a single window ordering group is implementation-dependent, hence introducing the potential for non-determinism in some OLAP functions. Distinct ordering groups within a partition are of course ordered according to the sort specification list of the window order clause.

If there is no window order clause, then each partition contains a single window ordering group consisting of all the rows in the partition and all rows are peers.

Example 2, “Window clause ordering”, shows the effect of ordering Sales_history using the Sales column in combination with the partitioning of Example 1, “Window clause”.

**Example 2 — Window clause ordering**

```
SELECT Territory, Month, Sales FROM Sales_history
       WINDOW w_eg2 AS (PARTITION  BY Territory
                        ORDER BY Month ASC)
```

— `w_eg2` is the name of the window defined by this example.

— In addition to the partitioning, `ORDER BY` introduces the ordering of rows within each partition. The syntax is identical to the order by clause of a query expression, including the optional null ordering specification.

**Table 6 — Result of window clause ordering**

| Territory | Month | Sales | |
|---|---|---|---|
| East | 199810 | 10 | ⎫ |
| East | 199811 | 4 | ⎪ |
| East | 199812 | 11 | ⎬ “East” partition |
| East | 199901 | 7 | ⎪ |
| East | 199902 | 10 | ⎭ |
| West | 199810 | 8 | ⎫ |
| West | 199811 | 12 | ⎪ |
| West | 199812 | 7 | ⎬ “West” partition |
| West | 199901 | 11 | ⎪ |
| West | 199902 | 6 | ⎭ |

Once again, the rows of Table 6, “Result of window clause ordering”, are organized to show the effect of the window specification and how they are operated on by the various OLAP functions. However, the result of the application of the window specification and subsequent OLAP functions does not assure any particular result row sequence. If it is desired that a result set be ordered on one or more columns, the containing query expression states that intent through the incorporation of an order by clause.

### 5.2.3.2 Null ordering and treatment

Feature T611, “Elementary OLAP operations” also introduced the “null ordering” option to the sort specification list of both the window order clause and the existing order by clause. Without it, null values may be sorted ahead of all valued instances of a column, or after all valued instances. Which option is chosen by default is implementation-defined. With the null ordering option, the user may explicitly request using NULLS FIRST or NULLS LAST that null values of an ordering column be sorted ahead or following the valued instances. In some cases the ability to explicitly define the null ordering is necessary for producing the correct result of an analytical function.

Unlike ordinary aggregate functions, window functions do not necessarily ignore NULL values. The ROW_NUMBER function, for instance, is fundamentally a row counting function over a window and as

such does not care about NULL values. Some window functions, such as LAG, LEAD, FIRST_VALUE, LAST_VALUE, and NTH_VALUE, permit explicit choice of null treatment: one may ignore nulls or respect nulls, as required by the desired outcome.

### 5.2.4 Window frames

#### 5.2.4.1 Introduction to window frames

The window frame of a row *R* in a window partition is a multiset of rows, defined relative to *R* in the ordering of the rows of *R*'s partition. Window frames are used to specify multisets of rows on which to perform window or aggregate functions, such as SUM or AVG or one of several other OLAP-specific functions.

There are three ways to specify the scope of a window frame, given a current row *R*: row counting, value offset, or ordering group counting.

1) The keyword ROWS says to include the specified number of rows before or after *R*. This defines a physical window frame.

2) The keyword RANGE requires that the partition be ordered on a single ordering key, of a type that may have a value added or subtracted (thus, numeric, datetime, or interval); one specifies a sort key value offset, and the window frame extends to any ordering groups containing values in the range between that offset and the sort key value in row *R*. This defines a logical window frame.

3) The keyword GROUPS requires that the partition be ordered (possibly on multiple sort keys), and says to include the specified number of ordering groups before or after the ordering group containing *R*. This also defines a logical window frame.

Thus there are two components in the specification of a frame, both mandatory:

1) The choice of the keyword ROWS, RANGE, or GROUPS, to indicate how the extent of the window frame is to be determined.

2) An indication of the starting row and ending row of the frame. Note that the ending row of the frame may follow the row currently being processed. This enables functions to be computed over "look ahead" data, a very valuable property for OLAP processing. The window frame extent syntax may consist of any of the following elements:

— UNBOUNDED PRECEDING to request all rows, ranges or groups preceding the current row.

— *n* PRECEDING to request the *n* rows, ranges, or groups preceding the current row.

— *n* FOLLOWING to request the *n* rows, ranges, or groups following the current row.

— UNBOUNDED FOLLOWING to request all rows, ranges, or groups following the current row.

— BETWEEN *start* AND *end* to request all rows, ranges, or groups between the start and end designations.

— *start* or *end* may also be CURRENT ROW to start or end the frame with the current row.

If the frame specification would extend the frame past the beginning or end of a partition, the frame will be truncated to the partition boundary.

If there is neither a window frame clause nor a window ordering clause, then there is an implicit window frame consisting of all rows in the partition. If there is no window frame clause, but a window ordering clause is present, then there is an implicit window frame that contains all the rows of the window partition of *R* that precede *R* or are peers of *R* in the window ordering of the window partition defined by the window ordering clause.

### 5.2.4.2 Physical window frames

A physical window frame is defined by counting some number of rows from the current row. Physical window frames are defined by the keyword ROWS. For physical framing to produce a sensible result from many window functions, there has to be a known and predictable number of input rows for each set of values of interest. Gaps or unexpected multiple rows would throw off the result or make it non-deterministic.

For instance, suppose one wanted a moving average of 3 calendar months from the sales_history example table. One could define a frame as "ROWS BETWEEN 2 PRECEDING AND CURRENT ROW", but that will only produce a meaningful running average if there is exactly one sales_history row for each calendar month. If there are missing months, the calendar month moving average will be incorrect.

Similarly, physical framing applied to data with multiple rows with the same ordering value may produce non-deterministic results. Rows within an ordering group are ordered in an implementation-dependent way, so a physical frame that does not cover the entire ordering group is not guaranteed to return deterministic results.

On the other hand, physical framing does not place any restrictions on window ordering (it does not even require window ordering), and may be the simplest way to frame data that has a well defined progression of values.

This Subclause contains two examples of the use of physical window frames. The first, Example 3, "Physical window frame, UNBOUNDED PRECEDING to CURRENT ROW", illustrates the use of a physical window frame that includes the current row and all preceding rows of the window partition; the results of the first example are seen in Table 7, "Result of physical window frame, UNBOUNDED PRECEDING to CURRENT ROW".

**Example 3 — Physical window frame, UNBOUNDED PRECEDING to CURRENT ROW**

```
SELECT Territory, Month, Sales FROM Sales_history
       WINDOW w_eg3 AS (PARTITION BY Territory
                        ORDER BY Month ASC
                        ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
```

— `w_eg3` is the window name of this example.

— For each row of each partition, the window frame syntax uses the keyword `ROWS` to designate the frames as physical. Each frame consists of the current row and all preceding rows in the partition.

**Table 7 — Result of physical window frame, UNBOUNDED PRECEDING to CURRENT ROW**

| Territory | Month | Sales | |
|---|---|---|---|
| East | 199810 | 10 | ⎫ |
| East | 199811 | 4 | ⎬ frame for "East", 199901 |
| East | 199812 | 11 | ⎪ |
| East | 199901 | 7 | ⎭ |
| East | 199902 | 10 | |
| West | 199810 | 8 | ⎫ |
| West | 199811 | 12 | ⎭ frame for "West", 199811 |

| Territory | Month | Sales | |
|---|---|---|---|
| West | 199812 | 7 | |
| West | 199901 | 11 | |
| West | 199902 | 6 | |

The second example, Example 4, "Physical window frame, 2 PRECEDING to 1 FOLLOWING", illustrates the use of a physical window frame that includes the current row of the window partition, two rows before the current row and one row following the current row; the results of this example is seen in Table 8, "Result of physical window frame, 2 PRECEDING to 1 FOLLOWING".

**Example 4 — Physical window frame, 2 PRECEDING to 1 FOLLOWING**

```
SELECT Territory, Month, Sales FROM Sales_history
       WINDOW w_eg4 AS (PARTITION  BY Territory
                        ORDER BY Month ASC
                        ROWS BETWEEN 2 PRECEDING AND 1 FOLLOWING)
```

— `w_eg4` is the window name of this example.

— For each row of each partition, the window frame syntax uses the keyword `ROWS` to designate the frames as physical. Each frame consists of the current row, the two rows that precede it and the row that follows it.

**Table 8 — Result of physical window frame, 2 PRECEDING to 1 FOLLOWING**

| Territory | Month | Sales | |
|---|---|---|---|
| East | 199810 | 10 | ⎫ |
| East | 199811 | 4 | ⎪ |
| East | 199812 | 11 | ⎬ frame for "East", 199812 |
| East | 199901 | 7 | ⎭ |
| East | 199902 | 10 | |
| West | 199810 | 8 | ⎫ |
| West | 199811 | 12 | ⎬ frame for "West", 199811 |
| West | 199812 | 7 | ⎭ |
| West | 199901 | 11 | |
| West | 199902 | 6 | |

Note that the frame for "West", 199811 only contains three rows since there is only one actual preceding row in the partition.

### 5.2.4.3 Logical window frames

#### 5.2.4.3.1 Introduction to logical window frames

Logical window frames do not count rows, rather, the frame is defined based on values in the data. A logical framing is either looking for a value range, or counting ordering groups. As such, logical window frames require a window ordering. Framing by value range uses the keyword RANGE, while framing by ordering groups uses the keyword GROUPS.

#### 5.2.4.3.2 RANGE window frames

A logical window frame defined by the RANGE keyword may have only a single sort key in its ordering clause. An application specifies a quantity that is added to and subtracted from the sort key to define the starting or ending point of a frame. A frame is then defined by value offsets preceding and following the current row. This means that the data type shall be numeric, datetime, or interval, and that the offset shall be a numeric literal or a compatible interval literal.

Consider a similar Sales_history table, with some months missing from the data, and the Month column stored as a date value containing the first day of each month. Example 5, "Logical window frame with RANGE", demonstrates the benefits of RANGE window frames. The results are shown in Table 9, "Result of logical window frame with RANGE".

**Example 5 — Logical window frame with RANGE**

```
SELECT Territory, Month, Sales FROM Similar_sales_history
       WINDOW w_eg5 AS (PARTITION BY Territory
                        ORDER BY Month ASC
                        RANGE INTERVAL '02' MONTH PRECEDING)
```

— `w_eg5` is the window name in this example.

— For each row of each partition, the window frame syntax uses the keyword `RANGE` to designate the frame as logical defined by a range of values in the ordering column (there may only be one ordering column). `INTERVAL '02' MONTH` is the literal that defines the range.

— The absence of a following clause means that each frame will include the CURRENT ROW.

**Table 9 — Result of logical window frame with RANGE**

| Territory | Month | Sales | |
|---|---|---|---|
| East | 1998-10-01 | 10 | ⎫ frame for "East", 1998-12-01 |
| East | 1998-12-01 | 11 | ⎭ |
| East | 1999-01-01 | 7 | |
| East | 1999-02-01 | 10 | |
| West | 1998-10-01 | 8 | ⎫ |
| West | 1998-11-01 | 12 | ⎬ frame for "West", 1998-12-01 |
| West | 1998-12-01 | 7 | ⎭ |
| West | 1999-02-01 | 6 | |

Note that the frame for “East”, 1998-12-01 only contains 2 rows since there is no row for “East”, 1998-11-01.

#### 5.2.4.3.3 GROUPS window frames

A logical window frame defined by the GROUPS keyword may have any number of sort keys in its ordering clause. Likewise, there is no limitation on the data types of the sort keys. A frame is defined by counting some number of ordering groups from the ordering group that contains the current row. The use of GROUPS syntax overcomes the weaknesses of physical window frames in that the ordering data does not have to be dense and the results are deterministic. Rows within an ordering group may have a non-deterministic order, but the groups as a whole are deterministically ordered. Results of any functions executed on grouped windows are then also deterministic. Likewise, they overcome the weaknesses of RANGE defined logical windows by permitting multiple sort keys with no type restrictions.

Example 6, “Logical window frame with GROUPS”, the results of which are seen in Table 10, “Result of logical window frame with GROUPS”, demonstrates the use of GROUPS window frames.

**Example 6 — Logical window frame with GROUPS**

```
SELECT Acno, Tid, Tradeday, TType, Amount, Ticker FROM Stock1
       WINDOW w_eg6 AS (PARTITION BY Acno
                        ORDER BY Tradeday
                        GROUPS BETWEEN 2 PRECEDING AND CURRENT ROW)
```

— `w_eg6` is the window name in this example.

— For each row of each partition, the window frame syntax uses the keyword `GROUPS` to designate the frame as logical defined by three ordering groups — the one containing the current row and the two preceding ordering groups.

— Note that, despite the `CURRENT ROW` designation, rows that are in the same ordering group as the current row but have yet to be processed are considered to be in the frame.

**Table 10 — Result of logical window frame with GROUPS**

| Acno | Tid | Tradeday | TType | Amount | Ticker | |
|---|---|---|---|---|---|---|
| 123 | 1 | 1 | buy | 1000 | csco | ⎫ |
| 123 | 2 | 1 | buy | 400 | inpr | ⎪ |
| 123 | 3 | 2 | buy | 2000 | symc | ⎬ frame for Tradeday 2 (same frame for Tid’s 3, 4 and 5) |
| 123 | 4 | 2 | buy | 1200 | csco | ⎪ |
| 123 | 5 | 2 | buy | 500 | inpr | ⎭ |
| 123 | 6 | 4 | buy | 200 | csco | ⎫ |
| 123 | 7 | 4 | buy | 100 | csco | ⎪ |
| 123 | 9 | 5 | buy | 400 | inpr | ⎪ |
| 123 | 10 | 5 | buy | 200 | goog | ⎬ frame for Tradeday 8 |
| 123 | 11 | 5 | buy | 1000 | inpr | ⎪ |

| Acno | Tid | Tradeday | TType | Amount | Ticker | |
|---|---|---|---|---|---|---|
| 123 | 12 | 5 | buy | 4000 | inpr | ⎜ |
| 123 | 13 | 8 | buy | 2000 | hpq | ⎝ |

### 5.2.4.4 Window frame exclusions

Syntax also allows the identification of rows to be specifically excluded from window frames in which they would otherwise participate. Options may be used to exclude the current row, the current ordering group or all rows that are ties of the current row. This adds a degree of fine tuning to the specification of window frames.

Example 7, "Window frame exclusion 1", and Example 8, "Window frame exclusion 2", show the effect of window frame exclusion syntax. Their results are available in Table 11, "Result of window frame exclusion 1", and Table 12, "Result of window frame exclusion 2", respectively.

**Example 7 — Window frame exclusion 1**

```
SELECT Territory, Month, Sales FROM Sales_history
       WINDOW w_eg7 AS (PARTITION BY Territory
                        ORDER BY Month ASC
                        RANGE INTERVAL '02' MONTH PRECEDING
                        EXCLUDE CURRENT ROW)
```

— `w_eg7` is the window name in this example,

— This is the same as the earlier example defining `w_eg5`, with the exception that the current row is excluded from each frame.

**Table 11 — Result of window frame exclusion 1**

| Territory | Month | Sales | |
|---|---|---|---|
| East | 1998-10-01 | 10 | -- frame for "East", 199812 |
| East | 1998-12-01 | 11 | ⎫ frame for "East", 199902 |
| East | 1999-01-01 | 7 | ⎭ |
| East | 1999-02-01 | 10 | |
| West | 1998-10-01 | 8 | ⎫ frame for "West", 199812 |
| West | 1998-11-01 | 12 | ⎭ |
| West | 1998-12-01 | 7 | |
| West | 1999-02-01 | 6 | |

**Example 8 — Window frame exclusion 2**

```
SELECT Acno, Tid, Tradeday, TType, Amount, Ticker FROM Stock1
       WINDOW w_eg8 AS (PARTITION BY acno
                        ORDER BY Tradeday
```

```
                        GROUPS BETWEEN 2 PRECEDING AND CURRENT ROW
                        EXCLUDE TIES)
```

— `w_eg8` is the window name in this example.

— This example is the same as earlier example defining `w_eg6` with the exception that only the current row is included in the frame from the ordering group that contains it (Tid's 4 and 5 are omitted from the frame for Tid 3 and Tid's 10, 11, and 12 are omitted from the frame for Tid 9).

**Table 12 — Result of window frame exclusion 2**

| Acno | Tid | Tradeday | TType | Amount | Ticker | |
|---|---|---|---|---|---|---|
| 123 | 1 | 1 | buy | 1000 | csco | ⎫ |
| 123 | 2 | 1 | buy | 400 | inpr | ⎬ frame for Tid 3 |
| 123 | 3 | 2 | buy | 2000 | symc | ⎫ ⎭ |
| 123 | 4 | 2 | buy | 1200 | csco | ⎪ |
| 123 | 5 | 2 | buy | 500 | inpr | ⎬ frame for Tid 9 |
| 123 | 6 | 4 | buy | 200 | csco | ⎪ |
| 123 | 7 | 4 | buy | 100 | csco | ⎪ |
| 123 | 9 | 5 | buy | 400 | inpr | ⎭ |
| 123 | 10 | 5 | buy | 200 | goog | |
| 123 | 11 | 5 | buy | 1000 | inpr | |
| 123 | 12 | 5 | buy | 4000 | inpr | |
| 123 | 13 | 8 | buy | 2000 | hpq | |

The EXCLUDE GROUP option is used to exclude the entire ordering group of the current row from a frame, and applies to any type of window frame (ROWS, RANGE, or GROUPS).

## 5.3 Explicit vs. implicit window definitions

The window definitions discussed thus far have been explicitly declared using the window definition clause in the table expression. Analytic functions may reference a defined window explicitly by name or may include the elements of the window definition syntax in the function specification. Such windows are implicitly defined and the only difference in their specification is the absence of a window name. Either form of window definition may be used, although explicit window definitions are a useful feature when more than one analytic function is to be executed on the same window. In either case, the semantics of function execution are the same.

Example 9, "Explicit window definition", demonstrates the use of an explicit window definition.

**Example 9 — Explicit window definition**

```
SELECT Territory, Month, Sales,
       SUM (Sales) OVER Win1 AS Sums
```

```
FROM Sales_history
     WINDOW Win1 AS (PARTITION BY Territory ORDER BY Month)
```

Example 10, "Implicit window definition", exemplifies same windows definition, but as an *implicit* window definition:

**Example 10 — Implicit window definition**

```
SELECT Territory, Month, Sales,
       SUM (Sales) OVER (PARTITION BY Territory ORDER BY Month) AS Sums
FROM Sales_history
```

## 5.4 Multiple window definitions

A given query may have any number of window definitions, explicit and/or implicit. Moreover, one window definition may reference another explicit window definition, with the resulting window containing a combination of the specifications of each. This is true of purely explicit window definitions or of mixtures of explicit and implicit window definitions. So, a window function may reference an explicit window definition, then fill in some of the window properties as they apply to the particular function. More specifically, if a window definition *X* references an existing window definition *Y*:

— *X* shall not include a window partition specification.

— If *Y* includes a window ordering specification, *X* shall not include a window ordering specification.

— *Y* shall not include a window frame specification.

This capability permits the easier definition of multiple windows that have some common elements. Example 11, "Multiple window definitions", shows how this is done.

**Example 11 — Multiple window definitions**

```
SELECT Territory, Month, Sales FROM Sales_history
       WINDOW w_eg9a AS (PARTITION BY Territory),
              w_eg9b AS (w_eg9a ORDER BY Month),
              w_eg9c AS (w_eg9b ROWS 2 PRECEDING),
              w_eg9d AS (w_eg9b GROUPS 1 PRECEDING EXCLUDE TIES)
```

— Any of windows `w_eg9a`, `w_eg9b`, `w_eg9c`, and/or `w_eg9d` may be used in different functions in the containing query.

— `w_egc` and `w_egd` use the same partitioning and ordering, but define different window frames.

— Another window might also be defined from `w_eg9a` with a different ordering and subsequently, different frame specifications, as well.

# 6 Window functions

## 6.1 Introduction to window functions

In addition to the window definition, Feature T611, "Elementary OLAP operations" and Feature T612, "Advanced OLAP operations" of ISO/IEC 9075-2 introduce new functions that exploit the window definition to produce results that are commonly used in OLAP applications. This clause discusses these functions, in addition to those introduced by Feature T614, "NTILE function", Feature T615, "LEAD and LAG functions", Feature T616, "Null treatment option for LEAD and LAG functions", Feature T617, "FIRST_VALUE and LAST_VALUE functions", and Feature T618, "NTH_VALUE function". Syntax and examples are used to explain the functions, as well as the features of window definitions that are relevant to their execution.

A window function is one of the following:

— rank function.

— distribution function.

— row number function.

— window aggregate function.

— Ntile function.

— lead or lag function.

— first, last, or Nth value function.

## 6.2 Rank functions

The rank functions compute the ordinal rank of each row *R* within a partition as defined by a window. The window definition shall include a window order clause but no window frame clause. There are two rank functions included in OLAP support:

— RANK() assigns an ordinal rank to each row in a partition according to its position in the ordering of the partition. If a row *R* in a partition has peers according to the ordering, the peers will be assigned the same rank as *R* and there will be a gap in the assignment of rank values to the rows of the partition.

— DENSE_RANK() also assigns an ordinal rank to each row in a partition according to its position in the ordering of the partition. If a row *R* has peers according to the ordering, the peers will be assigned the same rank as *R*. However, for DENSE_RANK(), there are no gaps in the assignment of rank values.

Example 12, "Rank functions with explicit window", selects rows of sales_history with RANK and DENSE_RANK computed for sales within each territory:

**Example 12 — Rank functions with explicit window**

```
SELECT territory, month, sales,
       RANK() OVER w_eg13 AS rank,
       DENSE_RANK() OVER w_eg13 AS dense_rank
FROM sales_history WINDOW w_eg13 AS (PARTITION BY territory
                                     ORDER BY sales DESC)
```

The computed values is seen below. Both `RANK()` and `DENSE_RANK()` show ties, but `RANK()` also shows gaps when ties are present, unlike `DENSE_RANK()`.

Example 13, "Rank functions with implicit window", shows the use of a single explicitly defined window `w_eg13`. The functions could also be specified as

**Example 13 — Rank functions with implicit window**

```
SELECT territory, month, sales,
       RANK() OVER (PARTITION BY Territory
                    ORDER BY Sales DESC) AS rank,
       DENSE_RANK() OVER (PARTITION BY Territory
                    ORDER BY Sales DESC) AS dense_rank
FROM Sales_history
```

Clearly the abbreviated form is easier both to write and to understand.

As with numerous examples, the results of this query (see Table 13, "Result of RANK and DENSE_RANK function") are ordered for the reader's convenience. An order by clause for the entire query is required to guarantee the result set ordering.

**Table 13 — Result of RANK and DENSE_RANK function**

| Territory | Month | Sales | Rank | Dense_Rank |
|---|---|---|---|---|
| East | 199812 | 11 | 1 | 1 |
| East | 199810 | 10 | 2 | 2 |
| East | 199902 | 10 | 2 | 2 |
| East | 199901 | 7 | 4 | 3 |
| East | 199811 | 4 | 5 | 4 |
| West | 199811 | 12 | 1 | 1 |
| West | 199901 | 11 | 2 | 2 |
| West | 199810 | 8 | 3 | 3 |
| West | 199812 | 7 | 4 | 4 |
| West | 199902 | 6 | 5 | 5 |



## 6.3 Distribution functions

The distribution functions compute a relative rank of each row within a partition as defined by a window. The window definition shall include a window order clause but no window frame clause. The rank computed by a distribution function is an approximate numeric ratio between 0.0 and 1.0, inclusive. There are two distribution functions included in OLAP support:

— PERCENT_RANK() is defined the same as the function of the same name in popular spreadsheet applications. Its value for each row in a partition is defined as the rank of the row minus 1 divided by the number of rows in the partition minus 1. This results in values ranging from 0.0 to 1.0 in each partition.

— CUME_DIST() is defined as the statistical cumulative distribution function. It is computed as the number of rows prior to or peer with the current row divided by the total number of rows in the partition.

The use of RANK, PERCENT_RANK, and CUME_DIST is demonstrated in Example 14, "Distribution functions", with the results shown in Table 14, "Result of PERCENT_RANK and CUME_DIST functions".

**Example 14 — Distribution functions**

```
SELECT territory, month, sales,
       RANK() OVER w_eg14 AS rank,
       PERCENT_RANK() OVER w_eg14 AS percent_rank,
       CUME_DIST() OVER w_eg14 AS cume_dist
FROM sales_history
       WINDOW w_eg14 AS (PARTITION BY territory
                         ORDER BY sales DESC)
```

**Table 14 — Result of PERCENT_RANK and CUME_DIST functions**

| Territory | Month | Sales | Rank | Percent_Rank | Cume_Dist |
|---|---|---|---|---|---|
| East | 199812 | 11 | 1 | 0.0 | 0.2 |
| East | 199810 | 10 | 2 | 0.25 | 0.4 |
| East | 199902 | 10 | 2 | 0.25 | 0.4 |
| East | 199901 | 7 | 4 | 0.75 | 0.8 |
| East | 199811 | 4 | 5 | 1.00 | 1.0 |
| West | 199811 | 12 | 1 | 0.00 | 0.2 |
| West | 199901 | 11 | 2 | 0.25 | 0.4 |
| West | 199810 | 8 | 3 | 0.50 | 0.6 |
| West | 199812 | 7 | 4 | 0.75 | 0.8 |
| West | 199902 | 6 | 5 | 1.00 | 1.0 |

As with numerous examples, the results of this query are ordered for the reader's convenience. An order by clause for the entire query is required to guarantee the result set ordering.

## 6.4 Row number function

The row number function computes a sequential number (starting with 1 for the first row) for each row *R* within a partition as defined by a window according to the window ordering. A window order clause is not required for the row number function, but no window frame clause is permitted.

Example 15, "Row number function", illustrates the use of ROW_NUMBER(). The result of that query appears in Table 15, "Result of ROW_NUMBER function".

**Example 15 — Row number function**

```
SELECT Territory, Month, Sales,
       ROW_NUMBER() OVER (ORDER BY Month, Territory) AS row_number
FROM sales_history
```

**Table 15 — Result of ROW_NUMBER function**

| Territory | Month | Sales | Row_Number |
|---|---|---|---|
| East | 199810 | 10 | 1 |
| West | 199810 | 8 | 2 |
| East | 199811 | 4 | 3 |
| West | 199811 | 12 | 4 |
| East | 199812 | 11 | 5 |
| West | 199812 | 7 | 6 |
| East | 199901 | 7 | 7 |
| West | 199901 | 11 | 8 |
| East | 199902 | 10 | 9 |
| West | 199902 | 6 | 10 |

As with numerous examples, the results of this query are ordered for the reader's convenience. An order by clause for the entire query is required to guarantee the result set ordering.

## 6.5 Window aggregate functions

Window aggregate functions perform the standard aggregations on frames of rows within a partition of a window. Such "moving" aggregations are commonplace in OLAP applications. These functions are the major beneficiaries of window frames since the aggregations are performed on the rows within a defined frame. Aggregate functions may also be executed on windows without an implicit or explicit frame specification. (Recall that a window ordering without explicit frame specification implies a frame from the start of the partition through the current row and all its ordering peers.) In this case, the aggregate is computed over the entire partition and the same value is the result for each row in the partition.

Example 16, "Window aggregate function (SUM) ordered", demonstrates the use of an aggregate function in an ordered query. It produces results such as those in Table 16, "Result of aggregate function (SUM) ordered".

**Example 16 — Window aggregate function (SUM) ordered**

```
SELECT Territory, Month, Sales,
       SUM (Sales) OVER (PARTITION BY Territory) AS sums
FROM Sales_history
ORDER BY Territory, Sales DESC, Month
```

This is an example of an aggregate without a window frame. Each row in a partition computes the same sum over all the rows in the partition.

**Table 16 — Result of aggregate function (SUM) ordered**

| Territory | Month | Sales | Sums |
|---|---|---|---|
| East | 199812 | 11 | 42 |
| East | 199810 | 10 | 42 |
| East | 199902 | 10 | 42 |
| East | 199901 | 7 | 42 |
| East | 199811 | 4 | 42 |
| West | 199811 | 12 | 44 |
| West | 199901 | 11 | 44 |
| West | 199810 | 8 | 44 |
| West | 199812 | 7 | 44 |
| West | 199902 | 6 | 44 |

In this case, the ordering of the results is guaranteed by the query ORDER BY clause.

The code in Example 17, "Window aggregate function (SUM) unordered", has been altered with the addition of a window ordering clause, which implies a window frame, producing cumulative results over each partition. Table 17, "Result of aggregate function (SUM) unordered", shows the results of that example.

**Example 17 — Window aggregate function (SUM) unordered**

```
SELECT Territory, Month, Sales,
       SUM (Sales) OVER (PARTITION BY Territory
                         ORDER BY Month) AS sums
FROM Sales_history
```

The window ordering clause defines an implicit window frame that for some row *R*, consists of all rows preceding *R* in the partition, plus all rows in *R*'s ordering group.

**Table 17 — Result of aggregate function (SUM) unordered**

| Territory | Month | Sales | Sums |
|---|---|---|---|
| East | 199812 | 11 | 25 |
| East | 199810 | 10 | 10 |
| East | 199902 | 10 | 42 |
| East | 199901 | 7 | 32 |
| East | 199811 | 4 | 14 |

| Territory | Month | Sales | Sums |
|---|---|---|---|
| West | 199811 | 12 | 20 |
| West | 199901 | 11 | 38 |
| West | 199810 | 8 | 8 |
| West | 199812 | 7 | 27 |
| West | 199902 | 6 | 44 |

This example has no query ORDER BY clause, so the ordering of the output rows is not guaranteed.

An explicit window framing might be used to calculate a moving average. Example 18, "Window aggregate moving average", illustrates this usage, and Table 18, "Result of aggregate function (AVG)", shows the results of that query.

**Example 18 — Window aggregate moving average**

```
SELECT Territory, Month, Sales,
       AVG (Sales)
         OVER (PARTITION BY Territory
               ORDER BY Month
               ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) AS avgs
FROM Sales_history
```

— This example computes the moving average of the sales column taking into account the current row, preceding row, and following row based on the specified ordering. This is very common and useful in analytical applications.

— As with numerous examples, the results of this query are ordered for the reader's convenience. An order by clause for the entire query is required to guarantee the result set ordering.

**Table 18 — Result of aggregate function (AVG)**

| Territory | Month | Sales | Avgs |
|---|---|---|---|
| East | 199810 | 10 | 7.0 |
| East | 199811 | 4 | 8.3 |
| East | 199812 | 11 | 7.3 |
| East | 199901 | 7 | 9.3 |
| East | 199902 | 10 | 8.5 |
| West | 199810 | 8 | 10.0 |
| West | 199811 | 12 | 9.0 |
| West | 199812 | 7 | 10.0 |
| West | 199901 | 11 | 8.0 |
| West | 199902 | 6 | 8.5 |

## 6.6 Ntile function

The NTILE function assigns a quantile ranking to the rows of a partition (or all rows of a query, in the absence of a partitioning), according to its ordering. The sole parameter of the function is an exact numeric literal or dynamic parameter that produces a positive value indicating the number of quantiles into which the ordered rows are to be distributed. If a partition contains fewer rows than the requested number of quantiles, the function simply returns values 1, 2, ... The window definition for a NTILE function shall include a window ordering, but shall not include a window frame.

Example 19, "NTILE in partitioned query", contains an example of NTILE used in a partitioned query, which produces the results shown in Table 19, "Result of aggregate function (NTILE) with partitioned query".

**Example 19 — NTILE in partitioned query**

```
SELECT Territory, Month, Sales,
       NTILE(3)
         OVER (PARTITION BY Territory
               ORDER BY Sales DESC) AS quantile
FROM Sales_history
```

— This `NTILE` function requests that the rows be divided into three quantiles according to their Sales rank.

**Table 19 — Result of aggregate function (NTILE) with partitioned query**

| Territory | Month | Sales | Quantile |
|---|---|---|---|
| East | 199812 | 11 | 1 |
| East | 199810 | 10 | 1 |
| East | 199902 | 10 | 2 |
| East | 199901 | 7 | 2 |
| East | 199811 | 4 | 3 |
| West | 199811 | 12 | 1 |
| West | 199901 | 11 | 1 |
| West | 199810 | 8 | 2 |
| West | 199812 | 7 | 2 |
| West | 199902 | 6 | 3 |

By contrast, Example 20, "NTILE in non-partitioned query", shows an examples of NTILE used in a non-partitioned query, the results of which are shown in Table 20, "Result of aggregate function (NTILE) in non-partitioned query".

**Example 20 — NTILE in non-partitioned query**

```
SELECT Territory, Month, Sales,
       NTILE(3)
```

```
        OVER (ORDER BY Sales DESC) AS quantile
FROM Sales_history
```

— This NTILE function omits the window partition specification so that the quantile ranking is done over the entire table.

— In this query (as in the previous), some rows are placed in different quantiles, in spite of having the same Sales values (see 'East', 199810 and 'East', 199902). Such rows may legitimately be assigned to either quantile. Multiple rows that contain the same values in the ordering columns result in non-deterministic results from the NTILE function.

**Table 20 — Result of aggregate function (NTILE) in non-partitioned query**

| Territory | Month | Sales | Quantile |
|---|---|---|---|
| West | 199811 | 12 | 1 |
| East | 199812 | 11 | 1 |
| West | 199901 | 11 | 1 |
| East | 199810 | 10 | 1 |
| East | 199902 | 10 | 2 |
| West | 199810 | 8 | 2 |
| East | 199901 | 7 | 2 |
| West | 199812 | 7 | 3 |
| West | 199902 | 6 | 3 |
| East | 199811 | 4 | 3 |



As with numerous examples, the results of these queries are ordered for the reader's convenience. An order by clause for the entire query is required to guarantee the result set ordering.

## 6.7 LEAD and LAG functions

The LEAD and LAG functions provide the capability to reference rows preceding or following the current row being processed. Both functions take three parameters: the first is a value expression, the second is an exact numeric literal with a positive value, and the third is another value expression whose result data type is the same as the first. The result of LEAD is the value of the expression from the first argument evaluated on the row that follows the current row by as many rows as specified by the second argument. If there is no row that corresponds to the specified offset, the result is the value of the third argument. The result of LAG is the same, except that the row on which the first argument is evaluated is the row that precedes the current row by as many rows as specified by the second argument. As with LEAD, if there is no row that corresponds to the specified offset, the result is the value of the third argument. The window definition for LEAD or LAG functions shall include a window ordering, but shall not include a window frame.

Example 21, "LEAD function", and Example 22, "LAG function", demonstrate the use of LEAD and LAG, respectively. The results of the two examples are respectively seen in Table 21, "Result of aggregate function (LEAD)", and Table 22, "Result of aggregate function (LAG)".

**Example 21 — LEAD function**

```
SELECT Territory, Month, Sales,
       LEAD (Sales, 1, CAST (NULL AS INTEGER))
         OVER (PARTITION BY Territory
               ORDER BY Month) AS next_sales
FROM Sales_history
```

— This query displays Sales for the current and next month.

— The last row of each partition returns the NULL value for the `LEAD()` function. Note the NULL value is cast to the same type as Sales.

**Table 21 — Result of aggregate function (LEAD)**

| Territory | Month | Sales | Next_Sales |
|---|---|---|---|
| East | 199810 | 10 | 4 |
| East | 199811 | 4 | 11 |
| East | 199812 | 11 | 7 |
| East | 199901 | 7 | 10 |
| East | 199902 | 10 | |
| West | 199810 | 8 | 12 |
| West | 199811 | 12 | 7 |
| West | 199812 | 7 | 11 |
| West | 199901 | 11 | 6 |
| West | 199902 | 6 | |



**Example 22 — LAG function**

```
SELECT Territory, Month, Sales,
       LAG (Sales, 1, CAST (NULL AS INTEGER))
         OVER (PARTITION BY Territory
               ORDER BY Month) AS prev_sales
FROM Sales_history
```

— This query displays Sales for the current and previous month.

— The first row of each partition returns the NULL value for the `LAG()` function. Note the NULL value is cast to the same type as Sales.

**Table 22 — Result of aggregate function (LAG)**

| Territory | Month | Sales | Prev_Sales |
|---|---|---|---|
| East | 199810 | 10 | |
| East | 199811 | 4 | 10 |

| Territory | Month | Sales | Prev_Sales |
|---|---|---|---|
| East | 199812 | 11 | 4 |
| East | 199901 | 7 | 11 |
| East | 199902 | 10 | 7 |
| West | 199810 | 8 | |
| West | 199811 | 12 | 8 |
| West | 199817 | 7 | 12 |
| West | 199901 | 11 | 7 |
| West | 199902 | 6 | 11 |

As with numerous examples, the results of these queries are ordered for the reader's convenience. An order by clause for the entire query is required to guarantee the result set ordering.

## 6.8 FIRST_VALUE and LAST_VALUE functions

FIRST_VALUE and LAST_VALUE are used to compute an expression using the first or last row of a window frame in a given partition. If there is no row in the window frame, then the result of the function is NULL. If the window on which the FIRST_VALUE or LAST_VALUE function is being executed has no window frame specification, the function uses the first or last row of the entire partition.

Example 23, "FIRST_VALUE function", and Example 24, "LAST_VALUE function", demonstrate the use of FIRST_VALUE and LAST_VALUE, respectively. The two examples produce, respectively, the results seen in Table 23, "Result of aggregate function (FIRST_VALUE)", and Table 24, "Result of aggregate function (LAST_VALUE)".

**Example 23 — FIRST_VALUE function**

```
SELECT Territory, Month, Sales,
       FIRST_VALUE (Sales)
         OVER (PARTITION BY Territory
               ORDER BY Month) AS first_sales
FROM Sales_history
```

This query displays Sales beside the first Sales value of each partition.

**Table 23 — Result of aggregate function (FIRST_VALUE)**

| Territory | Month | Sales | First_Sales |
|---|---|---|---|
| East | 199810 | 10 | 10 |
| East | 199811 | 4 | 10 |
| East | 199812 | 11 | 10 |
| East | 199901 | 7 | 10 |
| East | 199902 | 10 | 10 |

| Territory | Month | Sales | First_Sales |
|---|---|---|---|
| West | 199810 | 8 | 8 |
| West | 199811 | 12 | 8 |
| West | 199817 | 7 | 8 |
| West | 199901 | 11 | 8 |
| West | 199902 | 6 | 8 |

**Example 24 — LAST_VALUE function**

```
SELECT Territory, Month, Sales,
       LAST_VALUE (Sales)
         OVER (PARTITION BY Territory
               ORDER BY Month
               ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) AS last_sales
FROM Sales_history
```

This query displays Sales beside the last Sales value taken from the frame consisting of the preceding, current, and following row being processed.

**Table 24 — Result of aggregate function (LAST_VALUE)**

| Territory | Month | Sales | Last_Sales |
|---|---|---|---|
| East | 199810 | 10 | 4 |
| East | 199811 | 4 | 11 |
| East | 199812 | 11 | 7 |
| East | 199901 | 7 | 10 |
| East | 199902 | 10 | 10 |
| West | 199810 | 8 | 12 |
| West | 199811 | 12 | 7 |
| West | 199817 | 7 | 11 |
| West | 199901 | 11 | 6 |
| West | 199902 | 6 | 6 |



As with numerous examples, the results of these queries are ordered for the reader's convenience. An order by clause for the entire query is required to guarantee the result set ordering.

## 6.9 NTH_VALUE function

The NTH_VALUE function is a generalization of the FIRST_VALUE and LAST_VALUE functions. The first parameter of NTH_VALUE is the expression to be evaluated using the specified row. The second parameter is an exact numeric literal whose value is used as an offset from the first or last row of the frame (or the

entire partition if there is no window frame specification) to identify the row to be used in evaluating the first argument. The value of the second parameter shall be positive.

NTH_VALUE also includes an optional direction specification to indicate whether the offset is to be taken from the first row of the frame or the last row of the frame. The direction specification immediately follows the NTH_VALUE function. The option values are FROM FIRST and FROM LAST, with FROM FIRST being the default.

As is the case with the FIRST_VALUE and LAST_VALUE functions, if there is no row in the window frame, then the result of the function is NULL.

Example 25, "NTH_VALUE function usage", and Example 26, "Equivalent NTH_VALUE function usage", demonstrate the use of NTH_VALUE with different argument values. The two examples are equivalent in this case, and the results appear in Table 25, "Result of aggregate function (NTH_VALUE)".

**Example 25 — NTH_VALUE function usage**

```
SELECT Territory, Month, Sales,
       NTH_VALUE (Sales, 2)
         OVER (PARTITION BY Territory
               ORDER BY Month) AS third_sales
FROM Sales_history
```

This query displays Sales beside the third Sales value in each partition.

Because the partitions each have 5 rows, the same result may be achieved by:

**Example 26 — Equivalent NTH_VALUE function usage**

```
SELECT Territory, Month, Sales,
       NTH_VALUE (Sales, 4) FROM LAST
         OVER (PARTITION BY Territory
               ORDER BY Month) AS third_sales
FROM Sales_history
```

**Table 25 — Result of aggregate function (NTH_VALUE)**

| Territory | Month | Sales | Third_Sales |
|---|---|---|---|
| East | 199810 | 10 | 4 |
| East | 199811 | 4 | 4 |
| East | 199812 | 11 | 4 |
| East | 199901 | 7 | 4 |
| East | 199902 | 10 | 4 |
| West | 199810 | 8 | 12 |
| West | 199811 | 12 | 12 |
| West | 199812 | 7 | 12 |
| West | 199901 | 11 | 12 |
| West | 199902 | 6 | 12 |

## 6.10 Null treatment

LEAD and LAG (and the FIRST_VALUE, LAST_VALUE, and NTH_VALUE functions) also include an option to describe how to handle NULL values in the selected row. The null treatment options are RESPECT NULLS and IGNORE NULLS and follow the function specification. If neither RESPECT NULLS nor IGNORE NULLS is specified, RESPECT NULLS is implicit.

When RESPECT NULLS is in effect, the entire window frame of the current row is used to evaluate the function. If IGNORE NULLS is in effect, only those rows from the window frame of the current row for which the FIRST_VALUE, LAST_VALUE, or NTH_VALUE expression evaluates to a non-null value are considered in determining the result.

Example 27, "Null treatment with LEAD function", Example 28, "Null treatment with FIRST_VALUE function", and Example 29, "Null treatment with LAST_VALUE function" are syntax examples of null treatment specification:

**Example 27 — Null treatment with LEAD function**

```
SELECT Territory, Month, Sales,
       LEAD (Sales, 1, CAST (NULL AS INTEGER))
         IGNORE NULLS OVER (PARTITION BY Territory
                            ORDER BY Month)
FROM Sales_history
```

Example 27, "Null treatment with LEAD function", only displays non-null Sales values. However, if all Sales values in a given partition are null, the third argument of the function will still return a null value.

**Example 28 — Null treatment with FIRST_VALUE function**

```
SELECT Territory, Month, Sales,
       FIRST_VALUE (Sales)
         IGNORE NULLS OVER (PARTITION BY Territory
                            ORDER BY Month)
FROM Sales_history
```

Example 28, "Null treatment with FIRST_VALUE function", displays the first non-null value of Sales in each partition.

**Example 29 — Null treatment with LAST_VALUE function**

```
SELECT Territory, Month, Sales,
       LAST_VALUE (Sales)
         RESPECT NULLS OVER (PARTITION BY Territory
                             ORDER BY Month
                             ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)
FROM Sales_history
```

Example 29, "Null treatment with LAST_VALUE function", displays the last value of Sales in each partition, regardless of whether it is null.

RESPECT NULLS is the default specification and may be omitted in this example.

# 7 Nested window functions

## 7.1 Introduction to nested window functions

Feature T619, "Nested window functions" introduces additional functionality to the windowed OLAP functions described in Clause 6, "Window functions", above.

Window definitions and functions as described in Clause 5, "Windows", and Clause 6, "Window functions", above provide a powerful mechanism whereby ranking, cumulative, moving, and reporting aggregate calculations may be specified succinctly in SQL and evaluated efficiently by the RDBMS. Using this specification, the user or an analytic application may define a window frame (fixed or variable in size) for each row and have an aggregate applied on all rows in the window frame.

For example, assuming the table Stocks (Ticker, TradeDay, Price), a query to find, for each stock ticker and each trade day, the number of times a stock price has exceeded $30 in the past 30 days of trading can be written like Example 30, "Q1: CASE expression in a window query".

**Example 30 — Q1: CASE expression in a window query**

```
SELECT Ticker, TradeDay, Price,
       SUM(CASE WHEN Price > 30 THEN 1 ELSE 0 END)
         OVER (PARTITION BY Ticker
               ORDER BY TradeDay
               ROWS BETWEEN 30 PRECEDING AND 1 PRECEDING) AS Freq
FROM Stocks
```

Though the window specification is powerful and allows efficient evaluation of ranking, moving, cumulative, and reporting aggregates by the RDBMS, it lacks the ability to solve what could be called "comparative window function" calculations. These are calculations wherein the window function value for the current row is the result of a comparative analysis involving one or more rows in the window frame.

For example, window functions as described in Clause 5, "Windows", and Clause 6, "Window functions" are not useful in answering queries like "find the number of times the stock price has exceeded a day's price in the preceding 30 days", instead of a fixed $30 value as in the above example. SQL to answer this query resorts to self-joins and becomes difficult to write, understand, and optimize. See Example 31, "Q2: Complex join".

**Example 31 — Q2: Complex join**

```
SELECT s1.Ticker, s1.TradeDay, s1.Price,
       SUM(CASE WHEN s2.Price > s1.Price THEN 1 ELSE 0 END) AS Freq
FROM Stocks s1, Stocks s2
WHERE s1.Ticker = s2.Ticker
AND s2.TradeDay <= s1.TradeDay-1
AND s2.TradeDay >= s1.TradeDay-30
GROUP BY s1.Ticker, s1.TradeDay, s1.Price
```

To address queries like Q2, Feature T619, "Nested window functions" extends window aggregates, making them more powerful and suitable for "comparative analysis". The extensions include:

— Row markers that give access to certain interesting rows.

— Offsets that provide ability to access a row at any offset from the marker rows.

— A row marker that moves across the rows in the window frame during calculation of a window aggregate.

## 7.2 Row markers

Row markers give access to interesting rows within a partition and frame for comparative analysis. The row markers that are defined in Feature T619, "Nested window functions" are:

— BEGIN_PARTITION — the first row of the window partition

— BEGIN_FRAME — first row of the window frame

— CURRENT_ROW — the row for which a window frame is created

— END_FRAME — the last row of the window frame

— END_PARTITION — the last row of the window partition

— FRAME_ROW — the variable row that ranges over all rows of a window frame (between BEGIN_FRAME and END_FRAME) when computing a window aggregate.

For example, consider the data in the Stocks table, partitioned by Ticker, ordered by TradeDay, and with the window framing clause ROWS BETWEEN 4 PRECEDING AND 1 PRECEDING. Assume that "(ZYX, 6, 11)" is the row for which we are currently computing the window function. Table 26, "Result of row markers", illustrates where the row markers other than FRAME_ROW point.

**Table 26 — Result of row markers**

| Ticker | Tradeday | Price | |
|---|---|---|---|
| ZYX | 1 | 10 | — BEGIN_PARTITION |
| ZYX | 2 | 11 | — BEGIN_FRAME |
| ZYX | 3 | 12 | |
| ZYX | 4 | 12 | |
| ZYX | 5 | 12 | — END_FRAME |
| ZYX | 6 | 11 | — CURRENT_ROW |
| ZYX | 7 | 12 | |
| ZYX | 8 | 12 | — END_PARTITION |



To access individual columns from the marker rows, the function VALUE_OF has been introduced. This function is only permitted nested in an aggregated argument of a window aggregate. For example "VALUE_OF (Price AT CURRENT_ROW)" gives the value of column "Price" from the row indicated by the row marker CURRENT_ROW. Using the CURRENT_ROW marker, the self-join query Q2 may be rewritten elegantly as shown in Example 32, "Q3: VALUE_OF function usage".

**Example 32 — Q3: VALUE_OF function usage**

```
SELECT Ticker, TradeDay, Price,
       SUM (CASE WHEN Price > VALUE_OF (Price AT CURRENT_ROW)
                 THEN 1
                 ELSE 0 END)
         OVER (PARTITION BY Ticker
               ORDER BY TradeDay
```

```
                ROWS BETWEEN 30 PRECEDING AND 1 PRECEDING) AS Freq
FROM Stocks;
```

Note that Q3 uses `ROWS BETWEEN 30 PRECEDING AND 1 PRECEDING`, whereas the preceding sample data showed `ROWS BETWEEN 4 PRECEDING AND 1 PRECEDING` for brevity.

The `CASE` expression in Q3 illustrates the use of `VALUE_OF`, nested in the window aggregate `SUM`. The `SUM` is adding a collection of 1's and 0's; i.e., counting the number of times the `WHEN` condition in the `CASE` expression is true. The `WHEN` condition involves a comparison of the Price column in two rows. The first Price is found in the row that varies throughout the window frame in the course of evaluating the aggregate. The second Price, in `VALUE_OF (Price AT CURRENT_ROW)`, is the value of Price in a fixed row, the row that "owns" the window frame and serves as the basis for the window frame's formation via `BETWEEN 30 PRECEDING AND 1 PRECEDING`. Thus, Q3 computes the number of times in the preceding 30 days that the value of Price has exceeded the current row's Price. By eliminating the self-join, the query Q3 is easier to write and understand. Moreover, the query should also be easier to optimize by the RDBMS.

For convenience, expressions may be the first argument of VALUE_OF function. For example, instead of having to write multiple functions as in:

```
VALUE_OF(sal AT CURRENT_ROW) +
VALUE_OF(commission AT CURRENT_ROW) -
VALUE_OF(revenue AT CURRENT_ROW),
```

one may code:

```
VALUE_OF(sal + commission - revenue AT CURRENT_ROW)
```

## 7.3 Offsets

Positive and negative integer offsets are also supported to access rows at a specified offset from the marker rows. For example one may say "VALUE_OF (Price AT BEGIN_FRAME + 1)" to get the value of column "Price" from the second row in the window frame. Similarly, "VALUE_OF(Price AT END_FRAME – 1)" gives access to Price from the next to last row in the window frame.

When an offset specifies a row outside of the window partition (i.e., before BEGIN_PARTITION or after END_PARTITION), then VALUE_OF will return NULL. To illustrate the semantics, consider a window frame that is "BETWEEN 5 PRECEDING AND 2 PRECEDING" operating on the following data (Ticker, TradeDay, Price). Assume that the CURRENT_ROW is at (ZYX, 7, 12) and the Price column is being accessed using the VALUE_OF function. The results are those contained in Table 27, "Result of row markers (offsets)".

**Table 27 — Result of row markers (offsets)**

| Ticker | Tradeday | Price | |
|---|---|---|---|
| ZYX | 1 | 10 | — BEGIN_PARTITION |
| ZYX | 2 | 11 | — BEGIN_FRAME |
| ZYX | 3 | 12 | |
| ZYX | 4 | 12 | |
| ZYX | 5 | 12 | — END_FRAME |
| ZYX | 6 | 11 | |
| ZYX | 7 | 12 | — CURRENT_ROW |

| Ticker | Tradeday | Price | |
|---|---|---|---|
| ZYX | 8 | 12 | — END_PARTITION |

Then:

```
VALUE_OF (Price AT BEGIN_FRAME+1) = 12
VALUE_OF (Price AT BEGIN_FRAME-1) = 10
VALUE_OF (Price AT BEGIN_FRAME-2) IS NULL
VALUE_OF (Price AT CURRENT_ROW) = 12
```

Of course, the user may wish a different default than NULL when a row outside the window partition is referenced; for that an optional third parameter with the same data type as the first is supported (the second parameter is the row marker defined location of the desired row):

```
VALUE_OF (Price AT BEGIN_FRAME-2, 5) = 5
VALUE_OF (Price AT CURRENT_ROW+2, 6) = 6
```

The VALUE_OF function is permitted to be nested within itself, which means that the default for an outer VALUE_OF is computed by an inner VALUE_OF. For example:

```
VALUE_OF (Price AT CURRENT_ROW - 5, VALUE_OF (Price AT BEGIN_PARTITION))
```

In this example, the user wants the Price in the row 5 rows prior to the current row. This row might fall outside the partition, in which case the default applies. The default is another VALUE_OF expression. In this example, the default is the Price in the row at the beginning of the partition.

## 7.4 FRAME_ROW

Given a particular current row, most of the row markers have fixed positions within the window partition. FRAME_ROW, on the other hand, is a variable row that moves from BEGIN_FRAME to END_FRAME in the window frame as the window aggregate is computed. In Table 28, "Result of window frames with row markers", using a window frame BETWEEN 4 PRECEDING AND 1 PRECEDING, FRAME_ROW moves from row (ZYX, 2, 11) to (ZYX, 5, 12), when the CURRENT_ROW is at (ZYX, 6, 11).

**Table 28 — Result of window frames with row markers**

| Ticker | Tradeday | Price | |
|---|---|---|---|
| ZYX | 1 | 10 | |
| ZYX | 2 | 11 | — BEGIN_FRAME — FRAME_ROW |
| ZYX | 3 | 12 | — FRAME_ROW |
| ZYX | 4 | 12 | — FRAME_ROW |
| ZYX | 5 | 12 | — END_FRAME — FRAME_ROW |
| ZYX | 6 | 11 | — CURRENT_ROW |
| ZYX | 7 | 12 | |
| ZYX | 8 | 12 | — END_PARTITION |

As with other markers, VALUE_OF notation is used to get column values or more general complex expressions. Note that VALUE_OF (value expression AT FRAME_ROW+1) gives access to the row after FRAME_ROW and VALUE_OF (value expression AT FRAME_ROW-1) gives the row previous to FRAME_ROW. It may be helpful to note that VALUE_OF (value expression AT FRAME_ROW) is just the same as value

expression. For example, if W is a window name, then AVG (VALUE_OF (Price AT FRAME_ROW)) OVER W is just the same as AVG (Price) OVER W.

This is because AVG (Price) is computed as the average of Price as the moving row varies from BEGIN_FRAME through END_FRAME. Thus, the benefit of FRAME_ROW is the ability to access rows that are offset from the FRAME_ROW. However, for consistency with other row markers, VALUE_OF is also supported at FRAME_ROW with no offset. Using FRAME_ROW, queries like "for each day D, how many times in the preceding 30-day period, a stock price has exceeded D's value and maintained (either stayed the same or has increased) on the next day" may be answered efficiently using window function. The SQL for this query is shown in Example 33, "Q4: frame_row and current_row in value_of function".

**Example 33 — Q4: frame_row and current_row in value_of function**

```
SELECT Ticker, TradeDay, Price,
       SUM(CASE WHEN VALUE_OF(Price AT FRAME_ROW) >
                     VALUE_OF(Price AT CURRENT_ROW)
                 AND VALUE_OF(Price AT FRAME_ROW+1) >=
                     VALUE_OF(Price AT FRAME_ROW)
                THEN 1 ELSE 0 END)
         OVER (PARTITION BY Ticker
               ORDER BY TradeDay
               ROWS BETWEEN 30 PRECEDING AND 1 PRECEDING) AS Freq
FROM Stocks;
```

In Q4, the first comparison in the CASE expression looks for a `FRAME_ROW` whose Price is greater than the Price in `CURRENT_ROW`. And the second comparision looks for a `FRAME_ROW` whose Price is equaled or exceeded in the next row. Both conditions being satisfied indicates a "hit" and adds 1 (one) to the sum. The final result is the number of pairs (*R1*, *R2*) in which the first row *R1* beats the current row's Price, and the second row *R2* has a Price at least as high as *R1*.

## 7.5 Nested ROW_NUMBER function

Subclause 7.3, "Offsets", describes an optional third parameter to VALUE_OF to override the default null value when a row marker plus or minus offset that points to a non-existent row (outside the window partition) is used in VALUE_OF. This will handle most requirements to deal with boundary situations. However, the user may wish more sophisticated control, for example, to specify different handling at the beginning of the window partition as opposed to the end of the window partition. To cover such cases, the ROW_NUMBER function may be nested in the window aggregate function with a row marker argument. The syntax is:

```
ROW_NUMBER (row marker)
```

Unlike VALUE_OF, offsets are not permitted for nested ROW_NUMBER, since the ROW_NUMBER function commutes with addition and subtraction, *e.g.*,, `ROW_NUMBER (CURRENT_ROW + 1) = ROW_NUMBER (CURRENT_ROW) + 1`. Nested ROW_NUMBER provides complete information about where the row markers are within the window partition. Specifically:

```
ROW_NUMBER (BEGIN_PARTITION) = 1
ROW_NUMBER (BEGIN_FRAME) = row number of the first row of the window frame
ROW_NUMBER (CURRENT_ROW) = the row number of the current row
ROW_NUMBER (END_FRAME) = the row number of the last row of the window frame
ROW_NUMBER (END_PARTITION) = the number of rows in the window partition
ROW_NUMBER (FRAME_ROW) = the row number of the moving row (varies between BEGIN_FRAME and
END_FRAME).
```

As mentioned above, one use for nested ROW_NUMBER will be to supplement the default capability of VALUE_OF when a row is requested outside the window partition. For example, the user may require some more complex logic than a simple default provides. Using nested ROW_NUMBER, the user may do his own boundary testing.

Another use is to test if a row is outside the window frame, though perhaps still within the window partition. For example, suppose the user wants to compare an average over a window frame, and another

average taken over the rows that are three prior to ones in the window frame (i.e., at FRAME_ROW - 3). For the latter average, some of the rows will fall outside the window frame, and the user wants to substitute the value on the first row of the window frame for those rows. This may be done as follows:

```
AVG (Price) OVER w,
AVG (CASE WHEN ROW_NUMBER (FRAME_ROW) - 3 < ROW_NUMBER(BEGIN_FRAME)
          THEN VALUE_OF (Price AT BEGIN_FRAME)
          ELSE VALUE_OF (Price AT FRAME_ROW-3) END) OVER w
```

Another possible use for nested ROW_NUMBER is weighted aggregates. Window aggregates may be used to provide moving averages, a kind of "curve smoothing" that compensates for outlier values in a data set. A disadvantage of moving averages is that they have a "cliff" at the edge of a window frame: as long as an outlier is in the window frame, it is fully counted in the moving average, but when the outlier moves outside the window frame, the outlier is completely removed from the average.

An approach to mitigate the cliff effect in curve smoothing is to use weighted averages. The user might wish to weight values in an aggregate based on how far the rows are from CURRENT_ROW. The distance in rows between the FRAME_ROW and the CURRENT_ROW is ABS (ROW_NUMBER (FRAME_ROW) – ROW_NUMBER (CURRENT_ROW)). (The signed distance obtained by omitting ABS might also be of interest to the user.) To weight distant rows less than close rows, one could use weights that divide by the distance plus one, or the distance squared plus one. It would be convenient to define a weighting function, such as the one illustrated in Example 34, "Weight function".

**Example 34 — Weight function**

```
CREATE FUNCTION Weight (IN d INTEGER )
RETURNS REAL
LANGUAGE SQL
DETERMINISTIC
CONTAINS SQL
RETURNS NULL ON NULL INPUT
RETURN (1.0/((d * d + 1.0))
```

or such a weight function might be written in an external language. Then, using the weight function, one computes a weighted average as follows:

```
SUM (Weight (ROW_NUMBER (FRAME_ROW) - ROW_NUMBER (CURRENT_ROW)) * Val ) OVER w
    / SUM (Weight (ROW_NUMBER (FRAME_ROW) - ROW_NUMBER (CURRENT_ROW)) OVER w
```

Another potential use is to learn the number of rows in the window frame, which is ROW_NUMBER (END_FRAME) – ROW_NUMBER (BEGIN_FRAME). Distance, signed distance, frame size, and partition size are all calculated using the nested ROW_NUMBER.

## 7.6 Effects of EXCLUDE

The window exclude clause has the options EXCLUDE CURRENT ROW, EXCLUDE GROUP, EXCLUDE TIES, and EXCLUDE NO OTHERS. The last of these is the default case and actually excludes nothing. The other three are used to remove rows from the window frame (the current row, the current row and its peers under the ordering, or the peers of the current row while retaining the current row, respectively). Use of these options may make a window frame discontinuous, in the sense that there may be gaps in the numbering of rows within the window. For example, using:

```
WINDOW w1 AS ( PARTITION BY Ticker ORDER BY TradeDay ),
w2 AS ( w1 ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING EXCLUDE CURRENT ROW )
```

with the following data, and CURRENT_ROW as indicated in Table 29, "Result of EXCLUDE".

**Table 29 — Result of EXCLUDE**

| Ticker | Tradeday | Price | |
|---|---|---|---|
| ZYX | 1 | 10 | — BEGIN_PARTITION |