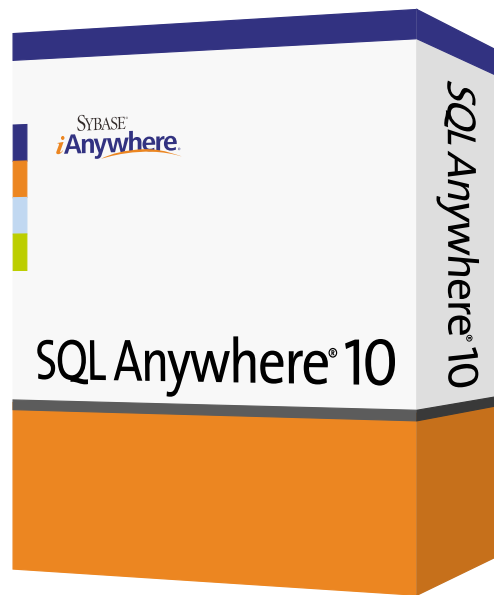


# Sybase iAnywhere White Paper



## Analytic Functions in SQL Anywhere

G. N. Paulley      B. Lucier

18 July 2007

Document Number 1037447

## Abstract

Analytic functions, recently added to the ANSI SQL 2003 standard as features T431, T611 and T612, offer the ability to perform complex data analysis within a single SQL statement. These analytic functions are sometimes referred to as OLAP, or On-Line Analytical Processing, functions. These SQL language enhancements include functions to compute variance, standard deviation, correlation and regression measures, and a new class of aggregate functions termed *window functions*. These functions, coupled with the **Group by** extensions **Cube**, **Rollup**, and **Grouping Sets** which were introduced in the 1999 ANSI standard, provide an efficient mechanism to compute common analyses such as ranking, percentiles, moving averages, and cumulative sums in a single SQL statement that previously would require self-joins, correlated subqueries, temporary tables, or some combination of all three. This iAnywhere Solutions' White Paper describes the OLAP functions available in the 10.0.1 release of SQL Anywhere.

## Contents

1	Introduction	5
2	Group by clause extensions	6
2.1	Group by Grouping Sets . . . . .	7
2.1.1	Using the GROUPING() function . . . . .	9
2.2	Group by Rollup and Cube . . . . .	11
2.2.1	Group by Rollup . . . . .	11
2.2.2	Equivalence of Rollup with Grouping Sets . . . . .	12
2.2.3	Group by Cube . . . . .	14
3	Statistical aggregate functions	15
3.1	Standard deviation and variance . . . . .	16
3.2	Correlation and linear regression . . . . .	17
3.2.1	Goodness-of-fit statistic . . . . .	17
3.2.2	Regression and correlation functions . . . . .	19
4	Window functions	21
4.1	Defining the window . . . . .	24
4.2	Ranking functions . . . . .	28
4.2.1	The DENSE_RANK() function . . . . .	29
4.2.2	Other ranking functions . . . . .	31
4.3	Numbering rows . . . . .	33
4.4	Window aggregate functions . . . . .	34
4.4.1	The FIRST_VALUE() and LAST_VALUE() functions . . . . .	40
A	BNF Grammar for OLAP Functions	42
	References	45

## List of Tables

1	Simple aggregate functions. . . . .	15
2	Statistical aggregate functions. . . . .	18
3	Additional regression measures. . . . .	21

## List of Figures

1	Schematic of a 3-row moving window with partitioned input. . . . .	26
2	Graphical plan for the query in Example 24. . . . .	35
3	Graphical plan for the main query block in Example 25. . . . .	37
4	Graphical plan for the subquery in Example 25. . . . .	38

- 5 Execution plan of the query from Example 26, using a window function. . . . . 39

## 1 Introduction

Extensions to the ANSI SQL standard [4] to include complex data analysis were first proposed as an amendment to the then-existing ANSI SQL standard [2] in November 1999 [3]. These extensions are now entrenched in the ANSI SQL 2003 standard as features T431, T611 and T612 (outside of ‘core’ SQL support). While iAnywhere Solutions has added portions of these SQL enhancements to previous releases of Adaptive Server Anywhere, the 9.0.1 and 9.0.2 releases of Adaptive Server Anywhere contain comprehensive support of these analytic functions. This technical white paper outlines the support for OLAP functionality in these 9.x releases of Adaptive Server Anywhere, and in the Jasper release of SQL Anywhere.

A summary of these SQL extensions are:

- *Extensions to the Group By clause.* With these extensions, which were first introduced in the 1999 ANSI standard [2], it is possible to write a complex SQL statement that can:
  1. partition the input rows in multiple ways, yielding a result set that concatenates all of the different groups together;
  2. create a ‘data cube’, providing a sparse, multi-dimensional result set for data mining analyses;
  3. create a result set that includes the original groups, but optionally also includes sub-total and grand-total rows.
- *Ranking functions.* These new functions enable application developers to compose single-statement SQL queries that answer questions such as ‘name the top 10 products shipped this year by total sales’, or ‘give the top 5% of salespersons who sold orders to at least 15 different companies’. These new ranking functions include RANK(), DENSE\_RANK(), PERCENT\_RANK(), and CUME\_DIST().
- *Window aggregate functions.* These functions provide moving average and cumulative measures in order to compute answers to queries such as ‘what is the quarterly moving average of the Dow Jones Industrial average’ or ‘list all employees and their cumulative salaries for each department’. Existing aggregate functions such as COUNT(), SUM(), MIN(), MAX(), AVG(), STDDEV(), and VARIANCE() and newer analytical functions such as COVAR\_SAMP() can be used as a window function with a new clause in an SQL query specification, Window, that conceptually creates a ‘moving window’ over a result set as it is processed.
- *Reporting aggregate functions.* Aggregate functions, including the basic aggregate functions COUNT(), SUM(), MIN(), MAX(), AVG(), can be used

to place the results of aggregate functions, possibly computed at different ‘levels’ in the statement, on the same row. This provides a means to compare aggregate values with detail rows within a group, avoiding the need for a join or a correlated subquery.

There are two primary benefits of these new OLAP functions. First, they provide the ability, in a concise way, to perform multidimensional data analysis, data mining, time series analyses, trend analysis, cost allocations, goal seeking, and exception alerting, often with a single SQL statement. Second, the window and reporting aggregate functions utilize a new relational operator, *window*, that can typically be executed much more efficiently than semantically equivalent queries that utilize self-joins or correlated subqueries.

## 2 Group by clause extensions

Recall the semantics of an ordinary query specification involving the clauses **Select - From - Where - Group by - Having** from the ANSI SQL standard (see grammar<sup>1</sup> rule 2):

1. Compute the Cartesian product of the table expressions present in the **From** clause.
2. Apply the predicates in the **Where** clause to this result. Rows that fail to satisfy the **Where** clause—that is, the **Where** clause does not evaluate to *true*—are rejected outright.
3. Except for aggregate functions, evaluate the expressions in the **Select** list and the list of expressions in the **Group by** clause for every row.
4. Partition the resulting rows by grouping them together based on distinct values of the expressions in the **Group by** clause, treating **Null** as simply a special value in each domain. The expressions in the **Group by** clause thus form a unique key for each partition.
5. For each partition, evaluate the aggregate functions present in the **Select** list or **Having** clause. Eliminate the detail rows from each partition, and form a result set consisting of the **Group by** expressions and the values of the aggregate functions computed for each partition.
6. Eliminate from this result set those rows (now partitions) that do not satisfy the **Having** clause.

---

<sup>1</sup> References to grammar rules in the text are to the Backus-Naur form of the relevant subset of SQL Anywhere’s SQL grammar as listed in Appendix A.

The first OLAP extension we describe is the ability to partition an intermediate result in *multiple ways* within the same query, concatenating all of the partitioning results together into a single result. A straightforward way to do this is by using a <GROUPING SETS SPECIFICATION> (see grammar rule 54) in the query's Group by clause.

## 2.1 Group by Grouping Sets

### EXAMPLE 1 (SPECIFYING MORE THAN ONE GROUPING CRITERIA)

Consider the following query over the schema of the SQL Anywhere `asademo` database<sup>2</sup>:

```
Select City, State, Company_name, Count(*) as Cnt
From Customer
Where State In ('MB', 'KS')
Group by Grouping Sets( (City, State), (Company_name ), ( ) )
```

returns the result  $R$

	<i>City</i>	<i>State</i>	<i>Company Name</i>	<i>Cnt</i>
1	(Null)	(Null)	'Cooper Inc.'	1
2	(Null)	(Null)	'Molly's'	1
3	(Null)	(Null)	'North Land Trading'	1
4	(Null)	(Null)	'Out of Town Sports'	1
5	(Null)	(Null)	'Overland Army Navy'	1
6	(Null)	(Null)	'The Ultimate'	1
7	(Null)	(Null)	'Toto's Active Wear'	1
8	(Null)	(Null)	(Null)	8
9	(Null)	(Null)	'Westend Dealers'	1
10	'Drayton'	'KS'	(Null)	3
11	'Pembroke'	'MB'	(Null)	4
12	'Petersburg'	'KS'	(Null)	1

The query specification above is semantically equivalent to the query expression

---

2 All of the example queries contained in this technical white paper are over the schema of the `asademo` sample database as shipped with the 9.0.2 release of Adaptive Server Anywhere.

```

Select Null, Null, Null, Count(*) as Cnt
From Customer
Where State In ('MB', 'KS')
  Union All
Select City, State, Null, Count(*) as Cnt
From Customer
Where State In ('MB', 'KS')
Group by City, State
  Union All
Select Null, Null, Company_name, Count(*) as Cnt
From Customer
Where State In ('MB', 'KS')
Group by Company_name

```

There are several points to be made about this example. Firstly, note that Null values are used as placeholders for Group by expression values that are not used for a particular grouping set. In this example, rows 1–7 and 9 are the rows generated by grouping over Company\_name, rows 10–12 are rows generated by grouping over the combination of City and State, and row 8 is the ‘grand total’ represented by the ‘empty’ grouping set, specified using a pair of matched parentheses ‘()’ (see Example 2 below). As a result of using placeholders for ‘missing’ <GROUP BY EXPRESSION>s, it is easy to confuse a row generated from a specific <GROUP BY TERM> from any Null values that are present in the input. These values can be differentiated by the use of the GROUPING() function, described in Section 2.1.1 below.

Secondly, the empty grouping set represents a single partition of all of the rows in the input to the Group by.

#### EXAMPLE 2 (EMPTY GROUPING SETS)

The following query specifies an empty grouping set:

```

Select Avg(Quantity)
From Product
Group by ()

```

and is semantically equivalent<sup>3</sup> to:

---

3 Recall that a <QUERY SPECIFICATION> that contains an aggregate function but does *not* contain a Group by clause will always yield a single row result, even if the result of the table expression being partitioned is empty. These semantics hold when an empty grouping set is used, contrary to the case where a <QUERY SPECIFICATION> does contain a Group by clause with any other <GROUP BY EXPRESSION>. In this latter case, an empty set generated by the query’s <FROM CLAUSE> and <WHERE CLAUSE> will yield an empty set for the query as well.

```
Select Avg(Quantity)
From Product
```

Thirdly, one can specify duplicate <GROUP BY TERM>s in a <GROUPING SETS SPECIFICATION> (see Example 3 below). In this case the result of the <QUERY SPECIFICATION> will contain identical rows<sup>4</sup>.

EXAMPLE 3 (DUPLICATE GROUPING SETS)

```
Select City, Count(*) as Cnt
From Customer
Where State In ('MB', 'KS')
Group by Grouping Sets( (City), (City) )
```

returns the result *R*

	<i>City</i>	<i>Cnt</i>
1	'Pembroke'	4
2	'Drayton'	3
3	'Petersburg'	1
4	'Pembroke'	4
5	'Drayton'	3
6	'Petersburg'	1

Fourthly, it is good practice to be explicit with the use of parentheses when using *Grouping Sets*, due to the ambiguity that arises with the SQL dialect of SQL Anywhere which permits any complex expression, including subqueries, to be parenthesized and used in virtually any context that a <COLUMN REFERENCE> can be used. For example, the syntax *Grouping Sets* (*X*, *Y*) does not yield a single group-by term, but two individual terms (*X*) and (*Y*). Hence an ordinary *Group by* clause that does not specify *Grouping Sets* is simply equivalent to a single grouping set (in the grammar, a single <GROUP BY TERM>) consisting of *n* <GROUP BY EXPRESSION>s in a single <SIMPLE GROUP BY TERM LIST>.

### 2.1.1 Using the GROUPING() function

The *GROUPING()* function is required to disambiguate result rows from a grouped <QUERY SPECIFICATION> where one or more of the <GROUP BY EXPRESSION>s contain Null values. The function takes a single parameter, that of

---

4 SQL Anywhere does not support feature T434 of the ANSI SQL 2003 standard, which permits the addition of the set quantifiers *All* (the default) or *Distinct* to the *Group by* clause. Specifying *Distinct* eliminates duplicate <GROUP BY TERM>s from a <GROUPING SET SPECIFICATION>. iAnywhere plans to include support for feature T434 in a future release of SQL Anywhere.

a `<GROUP BY EXPRESSION>` specified in one or more grouping sets<sup>5</sup>. The function's range is simply the integer values  $\{0, 1\}$ . The `GROUPING(X)` function returns the value 1 if, and only if, `X` is Null because it is not a member of the grouping set for which this row was output, and 0 otherwise.

EXAMPLE 4 (`GROUPING()` FUNCTION)

```
Select Employee.Emp_id As Employee,
       Year(Order_date) As Year,
       Count(Sales_order.ID) As Orders,
       Grouping(Employee) As GE,
       Grouping(Year) As GY
From Employee Left Outer Join Sales_order
     On Employee.Emp_id = Sales_order.Sales_rep
Where Employee.Sex In ('F')
     And Employee.State In ('TX', 'NY' )
Group by Grouping Sets ( (Year, Employee), (Year), ( ) )
Order by Year, Employee
```

yields the result  $R$

	<i>Employee</i>	<i>Year</i>	<i>Orders</i>	<i>GE</i>	<i>GY</i>
1	(Null)	(Null)	54	1	1
2	(Null)	(Null)	0	1	0
3	102	(Null)	0	0	0
4	390	(Null)	0	0	0
5	1062	(Null)	0	0	0
6	1090	(Null)	0	0	0
7	1507	(Null)	0	0	0
8	(Null)	2000	34	1	0
9	667	2000	34	0	0
10	(Null)	2001	20	1	0
11	667	2001	20	0	0

In this example, result tuple (1) represents the 'grand total' of orders (54), due to the specification of the empty grouping set `()`. Tuple (2) in the result is a subtotal row (indicated by a `GY` value of 0), a sum of zero orders for the Null year. In this example, a Null year exists because five of the employees that qualify for the query have no orders, as is shown by the result tuples (3) through (7). Tuples (9) and (11) show order counts for employee 667, for the years 2000 and 2001 respectively. Finally, subtotal tuples (8) and (10), indicated by a `GY` value of 0 and a `GE` value of 1, show the total orders for the

<sup>5</sup> SQL Anywhere does not support feature T433 of the 2003 ANSI SQL standard that defines a multi-argument `GROUPING()` function.

two years. Hence by testing the value of the `Grouping` function with each result tuple, one can differentiate between detail, subtotal, and grand total rows.

## 2.2 Group by Rollup and Cube

Using `Grouping Sets` is useful when one desires the concatenation of a number of disparate partitionings of the data into a single result set. However, it can be somewhat tedious to specify each individual `<GROUP BY TERM>` in a `<GROUPING SETS SPECIFICATION>`. Fortunately, two important syntactic shortcuts exist to concisely specify common grouping patterns. The first of these patterns is termed *rollup*, and the second is termed *cube*.

### 2.2.1 Group by Rollup

When specifying multiple grouping attributes for any SQL query specification, it is quite common to desire the computation of subtotal rows. One particular pattern that is typical of many applications is to compute subtotals of the grouping attributes from left-to-right in sequence, in a way reminiscent of IBM's COBOL Report Writer. Sometimes this pattern is referred to as a *hierarchy*, because the introduction of additional subtotal calculations produces additional rows with finer granularity of detail. In ANSI SQL syntax, the way in which one specifies a hierarchy of grouping attributes is through the use of the `Rollup` keyword that specifies a `<ROLLUP TERM>` (grammar rule 48).

#### EXAMPLE 5 (GROUP BY ROLLUP)

The following query summarizes the sales orders by year and quarter:

```
Select Quarter(Order_date) as Quarter,  
       Year(Order_date) As Year,  
       Count(*) As Orders,  
       Grouping(Quarter) As GQ,  
       Grouping(Year) As GY  
From Sales_order  
Group by Rollup(Year, Quarter)  
Order by Year, Quarter
```

yields the result *R*

	<i>Quarter</i>	<i>Year</i>	<i>Orders</i>	<i>GQ</i>	<i>GY</i>
1	(Null)	(Null)	648	1	1
2	(Null)	2000	380	1	0
3	1	2000	87	0	0
4	2	2000	77	0	0
5	3	2000	91	0	0
6	4	2000	125	0	0
7	(Null)	2001	268	1	0
8	1	2001	139	0	0
9	2	2001	119	0	0
10	3	2001	10	0	0

This query returns the total orders by each quarter (rows 3–6 and 8–10), along with subtotals by year (rows 2 and 7), and a ‘grand total’ row showing 648 orders (row 1). Note again how the values returned by `GROUPING()` function can be used to differentiate subtotal rows from the row that contains the grand total.

Another way to specify `Rollup` is to use the Microsoft SQL Server-compatible syntax `With Rollup` (grammar rule 53). Note that when using `With Rollup`, only simple expressions can be specified—that is, the `Group by` clause may contain only a `<SINGLE GROUP BY TERM LIST>`. Otherwise, the request will fail with a syntax error.

#### EXAMPLE 6 (GROUP BY WITH ROLLUP)

The following Microsoft SQL Server-compatible query produces the identical result to that of Example 5 above:

```
Select Quarter(Order_date) as Quarter,
       Year(Order_date) As Year,
       Count(* ) As Orders,
       Grouping(Quarter) As GQ,
       Grouping(Year) As GY
From Sales_order
Group by Year, Quarter With Rollup
Order by Year, Quarter
```

#### 2.2.2 Equivalence of Rollup with Grouping Sets

A `<ROLLUP TERM>` (grammar rule 48), specified through the use of the `Rollup` keyword, is essentially a generating function that produces a hierarchical series of grouping sets. In other words, if the `<ROLLUP TERM>` contains  $n$  `<GROUP BY EXPRESSION>`s of the form  $(X_1, X_2, \dots, X_n)$  then the `<ROLLUP TERM>` generates  $n + 1$  grouping sets as:

$$\{(), (X_1), (X_1, X_2), (X_1, X_2, X_3), \dots, (X_1, X_2, X_3, \dots, X_n)\}.$$

## EXAMPLE 7 (ROLLUP AS GROUPING SETS)

The following query is semantically equivalent to the query of Example 5 above:

```
Select Quarter(Order_date) as Quarter,
       Year(Order_date) As Year,
       Count(*) As Orders,
       Grouping(Quarter) As GQ,
       Grouping(Year) As GY
From Sales_order
Group by Grouping Sets ( (), (Year), (Year, Quarter) )
Order by Year, Quarter
```

Note, however, that a <ROLLUP TERM> may appear anywhere within a <GROUPING SETS SPECIFICATION>, and there may be multiple <ROLLUP TERM>s in any one <GROUP BY CLAUSE>. When a <ROLLUP TERM> is combined with other <GROUP BY TERM>s the resulting grouping sets are determined by computing the cross-product of all of the grouping sets specified or generated by the various terms, with duplicates eliminated.

## EXAMPLE 8 (COMBINATIONS OF GROUP BY ROLLUP TERMS)

The following <GROUP BY CLAUSE>:

```
Group by Rollup(A, B), C
```

is equivalent to

```
Group by Grouping Sets( (C), (A, C), (A, B, C) ).
```

Which grouping sets are implicitly generated does not depend on the syntactic order of the <GROUP BY TERM>s in the **Group by** clause. In the case of the example above, note how <GROUP BY EXPRESSION> ‘C’ is concatenated to each of the grouping sets generated by the <ROLLUP TERM>. However, each grouping set of <GROUP BY EXPRESSION>s specifies a set of attributes that form a key of the partitioned result, and so the syntactic ordering of the various <GROUP BY TERM>s is immaterial—only an **Order by** clause can guarantee a specific ordering of the result. This is true even in the case of multiple <ROLLUP TERM>s, as shown by this example:

## EXAMPLE 9 (MULTIPLE ROLLUP TERMS)

The following **Group by** clause

```
Group by Rollup(A, B), Rollup(C, D)
```

is equivalent to

```
Group by Grouping Sets(
  (), (A), (A, B), (A, C), (A, B, C),
  (A, C, D), (A, B, C, D)
).
```

**2.2.3** Group by Cube

The second typical grouping pattern supported by the ANSI SQL OLAP extensions is the ‘cube’, or ‘data cube’. The basic idea of a data cube is to create an  $n$ -dimensional summarization of the input using every possible combination of <GROUP BY TERM>s. Similar to **Rollup**, **Cube** is a generating function that produces every possible combination of grouping sets—a cross-tabulation of every dimension of the input that can be very useful for complex data analysis.

If there are  $n$  <GROUPING EXPRESSION>s of the form  $(X_1, X_2, \dots, X_n)$  in a <CUBE TERM> then **Cube** generates  $2^n$  grouping sets as:

$$\{(), (X_1), (X_1, X_2), (X_1, X_2, X_3), \dots, (X_1, X_2, X_3, \dots, X_n), \\ (X_2), (X_2, X_3), (X_2, X_3, X_4), \dots, (X_2, X_3, X_4, \dots, X_n), \dots, (X_n)\}.$$

**EXAMPLE 10 (GROUP BY CUBE)**

The following query summarizes sales orders by year, by quarter, and quarter within year:

```
Select Quarter(Order_date) as Quarter,
       Year(Order_date) As Year,
       Count(*) As Orders,
       Grouping(Quarter) As GQ,
       Grouping(Year) As GY
From Sales_order
Group by Cube (Year, Quarter)
Order by Year, Quarter
```

and yields the result  $R$

	<i>Quarter</i>	<i>Year</i>	<i>Orders</i>	<i>GQ</i>	<i>GY</i>
1	(Null)	(Null)	648	1	1
2	1	(Null)	226	0	1
3	2	(Null)	196	0	1
4	3	(Null)	101	0	1
5	4	(Null)	125	0	1
6	(Null)	2000	380	1	0
7	1	2000	87	0	0
8	2	2000	77	0	0
9	3	2000	91	0	0
10	4	2000	125	0	0
11	(Null)	2001	268	1	0
12	1	2001	139	0	0
13	2	2001	119	0	0
14	3	2001	10	0	0

<i>Function</i>	Symbol	Formula
SUM(X)		$\sum_{i=1}^n x_i$
MAX(X)		$x_i : x_i \geq x_j, i \neq j \forall i, j \in n$
MIN(X)		$x_i : x_i \leq x_j, i \neq j \forall i, j \in n$
AVG(X)	$\bar{x}$	$\frac{\sum x_i}{n}$
COUNT(*)		$n$
VAR_SAMP(X)	$s_x^2$	$\frac{\sum (x_i - \bar{x})^2}{(n-1)}$
VAR_POP(X)	$\sigma_x^2$	$\frac{\sum (x_i - \bar{x})^2}{n}$
VARIANCE(X)		identical to VAR_SAMP(X)
STDDEV_SAMP(X)	$s_x$	$\sqrt{\frac{\sum (x_i - \bar{x})^2}{(n-1)}}$
STDDEV_POP(X)	$\sigma_x$	$\sqrt{\frac{\sum (x_i - \bar{x})^2}{n}}$
STDDEV(X)		identical to STDDEV_SAMP(X)

TABLE 1: Simple aggregate functions.

In addition to the result rows that are produced by the Rollup query in Example 5 above, rows 2–5 summarize sales orders by calendar quarter *in any year*. Other arbitrary <GROUP BY TERM>s included in the query specification's <GROUP BY CLAUSE> are combined with a <CUBE TERM> in the identical manner to that of Rollup. As with Rollup, SQL Anywhere supports the Microsoft SQL Server-compatible syntax `With Cube`.

Note that the result set generated by through the use of Cube can be very large because Cube generates an exponential number of grouping sets. For this reason, SQL Anywhere will not permit a <GROUP BY CLAUSE> to contain more than 64 <GROUP BY EXPRESSION>s. If a statement exceeds this limit, it will fail with SQLCODE -944 (SQLSTATE 42WA1).

### 3 Statistical aggregate functions

The ANSI SQL OLAP extensions provide a number of additional aggregate functions (see Tables 1 and 2) that permit statistical analysis of numeric data. This support includes functions to compute variance, standard deviation, correlation, and linear regression.

### 3.1 Standard deviation and variance

The most common measure of the spread of a normal distribution is the *standard deviation*, which measures the spread of values  $x_i$  from their mean value,  $\bar{x}$ . A *deviation* is simply this difference  $x_i - \bar{x}$ , with some of the differences being positive and some negative. To standardize this difference, the deviations are squared so that all differences are positive values. Larger deviations from the mean, when squared, have a more significant impact on the total amount of the deviation. The *average* squared deviation is termed the *variance*, and is defined as:

$$s^2 = \frac{1}{(n-1)} \sum (x_i - \bar{x})^2 \quad (1)$$

and the standard deviation  $s$  is simply the square root of this value. A more convenient, and efficient, ‘computing formula’ for variance can be derived with a little algebra which avoids the need to precompute the mean of the distribution:

$$s^2 = \frac{1}{(n-1)} \left[ \sum x_i^2 - \frac{1}{n} \left( \sum x_i \right)^2 \right] \quad (2)$$

With ANSI SQL, two versions of both the variance and standard deviation functions exist. The *sampling* versions of the functions in SQL Anywhere compute the formula as described above in Equation 2, and correspond to the VAR\_SAMP() and STDDEV\_SAMP() functions. These are correspondingly aliased as VARIANCE() and STDDEV(), since the typical usage of variance and standard deviation deals with samples, not populations. The *population* version of these functions uses  $n$  as the denominator for computing the ‘average’ squared deviation, rather than  $(n-1)$ :

$$\sigma^2 = \frac{1}{n} \left[ \sum x_i^2 - \frac{1}{n} \left( \sum x_i \right)^2 \right] \quad (3)$$

Population variance and standard deviation are computed by the VAR\_POP() and STDDEV\_POP() functions, respectively.

All six functions are true aggregate functions in that they can compute values for a partition of rows as determined by the query’s <GROUP BY CLAUSE>. As with other basic aggregate functions such as MAX() or MIN(), their computation also ignores Null values in the input. Also, regardless of the domain of the expression being analyzed, all variance and standard deviation computation is done using IEEE double-precision floating point. If the input to any variance or standard deviation function is the empty set, then each function returns Null as its result. If VAR\_SAMP() is computed for a single row, then it returns Null, while VAR\_POP() returns the value 0.

### 3.2 Correlation and linear regression

*Linear regression* is a technique used to summarize the dependence of a variable  $y$  (termed the *dependent variable*) on an ‘explanatory’ variable  $x$  (termed the *independent variable*). Graphically, linear regression models are used to approximate a relationship between pairs of  $(x, y)$  values in a scatter-plot with a single line whose linear equation is

$$y = a + bx. \quad (4)$$

Linear regression uses the *method of least squares* to fit the various points to a line with slope  $b$  and intercept  $a$ , that passes through the point  $(\bar{x}, \bar{y})$ . The line models the *expected* value  $\hat{y}$  from each corresponding  $x$ -value. The difference between the expected value and the actual  $y$ -value from each observed  $(x, y)$  pair is termed a *residual*. If either of values of  $x$  or  $y$  are Null, that pair is ignored. As with the computation for standard deviation, some of the residuals will be positive, and some negative. The linear regression method of least squares derives a line  $y = a + bx$  such that the sum of the squares of the residuals is as small as possible. Given  $n$  non-Null observations, the formulae for computing the values of  $a$  and  $b$  are:

$$b = \frac{\sum(x - \bar{x})(y - \bar{y})}{\sum(x - \bar{x})^2} \quad (5)$$

$$a = \bar{y} - b\bar{x} \quad (6)$$

Once again, a more convenient computing formula for solving Equation 5 can be derived with a little algebra that eliminates the need to precompute the sample means  $\bar{x}$  and  $\bar{y}$ :

$$b = \frac{n \sum xy - (\sum x)(\sum y)}{n \sum x^2 - (\sum x)^2}. \quad (7)$$

The accuracy of the predicted value  $\hat{y}$  with a linear approximation is highly variable. Regression is significantly affected by *outliers* in the values of the  $(x, y)$  pairs being observed, and, of course, it may be that the relationship between  $x$  and  $y$  is not at all linear, but may be quadratic or even exponential. Nonetheless, linear regression tests are very useful tools in data analysis.

A summary of the OLAP linear regression functions supported by SQL Anywhere appears in Table 2. In particular, the aggregate function REGR\_SLOPE() returns the slope ( $b$ ) of the linear regression equation that models the relationship between the dependent and independent variables, while the aggregate function REGR\_INTERCEPT() returns the intercept ( $a$ ) of that equation.

#### 3.2.1 Goodness-of-fit statistic

To measure whether or not a linear relationship exists between  $x$  and  $y$ , one can compute the data’s *correlation coefficient* which is denoted by  $r$  (sam-

COVAR_SAMP(Y,X)	<i>Co-variance</i>	$s_{xy} = \frac{\sum xy - \frac{(\sum x)(\sum y)}{n}}{(n-1)}$
COVAR_POP(Y,X)	<i>Co-variance</i>	$\sigma_{xy} = \frac{\sum xy - \frac{(\sum x)(\sum y)}{n}}{n}$
CORR(Y,X)	<i>Correlation Coefficient</i>	$r = \frac{\sum xy - \frac{1}{n}(\sum x)(\sum y)}{(n-1)s_x s_y}$
REGR_AVGX(Y,X)	<i>Independent mean</i>	$\bar{x}$
REGR_AVGY(Y,X)	<i>Dependent mean</i>	$\bar{y}$
REGR_SLOPE(Y,X)	<i>Regression Slope</i>	$b = r \frac{s_y}{s_x}$
REGR_INTERCEPT(Y,X)	<i>Regression Intercept</i>	$a = \bar{y} - b\bar{x}$
REGR_R2(Y,X)	<i>'Goodness-of-fit'</i>	$r^2$
REGR_COUNT(Y,X)	<i>Sample size</i>	$n$ (non-null (Y, X) pairs)
REGR_SXX(Y,X)	<i>Sum of squares (x)</i>	$\sum x^2 - \frac{(\sum x)^2}{n}$
REGR_SYY(Y,X)	<i>Sum of squares (y)</i>	$\sum y^2 - \frac{(\sum y)^2}{n}$
REGR_SXY(Y,X)	<i>Sum of products</i>	$\sum xy - \frac{(\sum y)(\sum x)}{n}$

TABLE 2: Statistical aggregate functions.

pling) and  $\rho$  (population). The range of  $r$ , which is returned by the aggregate function `CORR()`, is between -1 and 1. The higher the absolute value, the stronger the association: if positive, a strong *positive* association, and if negative, a strong *negative* association. The formula for computing the sampling correlation coefficient for  $n$  observations is

$$r = \frac{1}{n-1} \sum \left( \frac{x - \bar{x}}{s_x} \right) \left( \frac{y - \bar{y}}{s_y} \right) \quad (8)$$

where  $\bar{x}$  and  $s_x$  are the sample mean and standard deviation of the  $x$ -values in the  $n$  observed pairs, and correspondingly  $\bar{y}$  and  $s_y$  are the sample mean and standard deviation of the  $y$ -values. To put it another way, the correlation coefficient is equal to the *co-variance* of  $x$  and  $y$ , denoted  $s_{xy}$ , divided by the product of their standard deviations. As with variance, there are two definitions of co-variance. The *sampling* co-variance of  $x$  and  $y$  is

$$s_{xy} = \frac{1}{(n-1)} \sum (x - \bar{x})(y - \bar{y}) = \frac{1}{(n-1)} \sum xy - \frac{(\sum x)(\sum y)}{n} \quad (9)$$

and the *population* co-variance of  $(x, y)$  is

$$\sigma_{xy} = \frac{1}{n} \sum xy - \frac{(\sum x)(\sum y)}{n}. \quad (10)$$

These two values are returned by the SQL functions `COVAR_SAMP()` and `COVAR_POP()` respectively.

The computing formula for  $r$  is

$$r = \frac{1}{n-1} \frac{\sum xy - \frac{1}{n}(\sum x)(\sum y)}{s_x s_y}. \quad (11)$$

From this equation, it can be shown that the extreme values of  $r = 1$  and  $r = -1$  occur only in the case of perfect linear association, and that values close to 1 or  $-1$  indicate that the points lie close to a straight line. The *square*  $r^2$  of the correlation coefficient is defined as the fraction of the variation in the values of  $y$  that is explained by the least squares regression of  $y$  on  $x$ . For example, if  $r^2 = 0.988$  then one can state that linear regression explains<sup>6</sup> 98.8% of the observed variation in  $y$ -values with respect to  $x$ . The value of  $r^2$  is sometimes referred to as the ‘*goodness-of-fit*’ statistic, and is returned by the aggregate function `REGR_R2()`.

The concepts of correlation and regression are tightly intertwined; in fact,  $r$  can be used in an alternative method to compute the slope  $b$  of a linear regression line:

$$b = \rho \frac{\sigma_y}{\sigma_x} = r \frac{s_y}{s_x}. \quad (12)$$

The two formulae are equivalent because the  $(n-1)$  terms from the sampling formulae for co-variance and standard deviation cancel each other out, as do the  $n$  terms from the population variants of those functions.

### 3.2.2 Regression and correlation functions

SQL Anywhere supports a variety of statistical functions, in addition to the goodness-of-fit function `REGR_R2()`, whose results can be used to assist in analyzing the quality of a linear regression. The statistical formulae implemented by these functions are summarized in Table 2. The first argument of each function is the *dependent* expression (designated by  $Y$ ), and the second argument is the *independent* expression (designated by  $X$ ). Briefly, these functions are:

- `REGR_COUNT(Y,X)`: This function returns the number of non-Null pairs of  $(x, y)$  values in the input. Only if *both*  $x$  and  $y$  in a given pair are non-Null will that observation be used in any linear regression computation.
- `REGR_AVGX(Y,X)`: This function returns the mean of the  $x$ -values from all of the non-Null pairs of  $(x, y)$  values.

---

6 A comprehensive overview of statistics is well beyond the scope of this paper. However, we caution the reader that a strong correlation does not necessarily imply a *cause and effect* relationship. Analysis of causation requires additional context than simply analyzing pairs of related values.

- **REGR\_AVGY(Y,X)**: This function returns the mean of the  $y$ -values from all of the non-Null pairs of  $(x, y)$  values.

In addition to these functions, SQL Anywhere supports three additional linear regression functions as per the ANSI SQL 2003 standard:

- **REGR\_SXX(Y,X)**: This function returns the sum of squares of  $x$ -values of the  $(x, y)$  pairs using the formula

$$\sum x^2 - \frac{1}{n} \left( (\sum x)^2 \right). \quad (13)$$

This equation is equivalent to the numerator of the sample or population variance formulae (see Equations 2 and 3). Note, as with the other linear regression functions, that **REGR\_SXX()** ignores any pair of  $(x, y)$  values in the input where either  $x$  or  $y$  is Null.

- **REGR\_SYY(Y,X)**: This function returns the sum of squares of  $y$ -values of the  $(x, y)$  pairs using the formula

$$\sum y^2 - \frac{1}{n} \left( (\sum y)^2 \right), \quad (14)$$

and again is equivalent to the numerator of the variance equation for  $y$ -values.

- **REGR\_SXY(Y,X)**: This function returns the difference of two sum of products formulae over the set of  $(x, y)$  pairs:

$$\sum xy - \frac{(\sum y)(\sum x)}{n}. \quad (15)$$

These three additional functions can be useful to analyze the characteristics of a linear regression because they represent partial formulae used to compute some of the other regression functions. In particular, the slope  $b$  of a linear regression equation (see Equation 5 above) as returned by the **REGR\_SLOPE()** function is equivalent to

$$b = \frac{\text{REGR\_SXY}(Y,X)}{\text{REGR\_SXX}(Y,X)} \quad (16)$$

and the goodness-of-fit measure  $r^2$  is equivalent to

$$r^2 = \frac{(\text{REGR\_SXY}(Y,X))^2}{\text{REGR\_SXX}(Y,X) \times \text{REGR\_SYY}(Y,X)}. \quad (17)$$

Using these additional functions, one can easily compute a broader range of measures pertaining to the linear correlation of pairs of dependent ( $x$ ) and independent ( $y$ ) expressions. Some of these measures [1, 5, 6] are outlined in Table 3.

<i>Standard Error (e) of y-values</i>
SQRT((REGR_SYY()-(POWER(REGR_SXY(),2.0)/REGR_SXX()))/(REGR_COUNT()-2))
<i>Standard Error of Slope (b)</i>
$e / \text{SQRT}(\text{REGR\_SXX}())$
<i>Standard Error of Intercept (a)</i>
$e * \text{SQRT}((1/\text{REGR\_COUNT}()) + (\text{POWER}(\text{REGR\_AVGX}(),2) / \text{REGR\_SXX}() ))$
<i>Regression sum of squares</i>
$\text{POWER}(\text{REGR\_SXY}(),2) / \text{REGR\_SXX}()$
<i>Residual sum of squares</i>
$\text{REGR\_SYY}() - (\text{POWER}(\text{REGR\_SXY}(),2)/\text{REGR\_SXX}())$
<i>t Statistic for Slope</i>
$\text{REGR\_SLOPE}() * \text{SQRT}(\text{REGR\_SXX}()) / e$

TABLE 3: Additional regression measures.

#### 4 Window functions

A major feature of the ANSI SQL extensions for OLAP is a new relational algebra construct called a *window*.

The SQL/OLAP windowing extensions allow a user to divide the result set of a query into groups of rows called *partitions*. Logically, as part of the semantics of computing the result of a query specification, partitions are created after the groups defined by the **Group by** clause are created, but before the evaluation of the final **Select** list and the query's **Order by** clause. Consequently, the order of evaluation of the clauses within an SQL statement becomes:

From  $\rightarrow$  **Where**  $\rightarrow$  **Group by**  $\rightarrow$  **Having**  $\rightarrow$  **Window**  $\rightarrow$  **Distinct**  $\rightarrow$  **Order by**.

Because window partitioning follows a **Group by** operator, the result of any aggregate function, such as **SUM()**, **AVG()**, and **VARIANCE()**, is available to the computation done for a partition. Hence a window provides another opportunity to perform grouping and ordering operations in addition to a query's **Group by** and **Order by** clauses. However, any computation involving a window function's result—for example, using it in a predicate—will require a more complex statement construction. Typically that construction will require a derived table to compute the window function result(s), with the derived table's outer **SELECT** block containing the desired **Where** clause predicates.

A window's partition can consist of the entire input, a single row, or something in between. The advantage of a 'window' construct is that the rows within the partition can then be sorted to support the additional expressive power provided by *window functions*. With window functions, one can construct queries that compute aggregate functions over aggregate functions, for example, the maximum `SUM()` of a particular quantity. Three classes of window functions (see grammar rule 6) can be used with a window:

- *Ranking* functions, such as `RANK()` and `CUME_DIST()`, which return the rank of a row relative to the other rows in a partition;
- a *row numbering* function, `ROW_NUMBER()`, which can uniquely number the rows in a partition; and
- *window aggregate* functions, such as `SUM()` and `COUNT()`, which are a subset of all of the aggregate functions supported by SQL Anywhere but can be used with a window.

Windows are defined using a `<WINDOW SPECIFICATION>` (grammar rule 23). The concept of a window is flexible. For each row in a window partition, one can define a 'sliding' window, which may vary the specific range of rows used to perform any computation on the 'current' row of the partition. This 'current row' provides the reference point for determining the start and end points of the window. Window sizes can be based on either:

Varying the size of a window

- a physical number of rows, using a `<WINDOW SPECIFICATION>` that defines a `<WINDOW FRAME UNIT>` (grammar rule 35) of `ROWS`, or
- a logical interval of a *numeric* value, using a `<WINDOW SPECIFICATION>` that defines a `<WINDOW FRAME UNIT>` of `RANGE`.

Both the start and end points of the window can be dynamically altered. For example, computing a cumulative sum would involve a window whose start point is fixed at the first row of each partition, and an end point that would 'slide' along the rows of the partition to include the current row (see Figure 1 on page 26 below). As another example, both the start and end points of the window may be variable, but define a constant number of rows for the entire partition. This construction permits one to compose queries that compute 'moving' averages, for example: writing an SQL query that returns a moving 3-day average stock price is relatively straightforward when utilizing a sliding window.

#### EXAMPLE 11 (QUERIES WITH WINDOW FUNCTIONS)

Consider the following query which lists all products shipped in July and August 2001 and the cumulative shipped quantity by shipping date:

```

Select p.id, p.description, s.quantity, s.ship_date,
       Sum(s.quantity) Over (Partition by s.prod_id
                           Order by s.ship_date
                           Rows Between Unbounded Preceding
                               and Current Row) as cumulative_qty
From sales_order_items s Join product p On (s.prod_id = p.id)
Where s.ship_date Between '2001-07-01' and '2001-08-31'
Order by p.id

```

This query returns the result *R*

	<i>ID</i>	<i>Description</i>	<i>Quantity</i>	<i>Ship Date</i>	<i>Cumulative Qty</i>
<i>R</i>	1	V-neck	24	2001-07-16	24
	2	Crew Neck	60	2001-07-02	60
	3	Crew Neck	36	2001-07-13	96
	4	Cotton Cap	48	2001-07-05	48
	5	Cotton Cap	24	2001-07-19	72
	6	Wool cap	48	2001-07-09	48
	7	Cloth Visor	12	2001-07-22	12
	8	Plastic Visor	60	2001-07-07	60
	9	Plastic Visor	12	2001-07-12	72
	10	Plastic Visor	12	2001-07-22	84
	11	Zipped Sweatshirt	60	2001-07-19	60
	12	Cotton Shorts	24	2001-07-26	24

In Example 11, the computation of the `SUM()` window function occurs after the join of the two tables and the application of the query's `Where` clause. The query uses an inline `<WINDOW SPECIFICATION>` that specifies that the input rows from the join shall be processed as follows:

1. Partition (group) the input rows based on the value of the `prod_id` attribute;
2. Within each partition, sort the rows by the `ship_date` attribute;
3. For each row in the partition, evaluate the `SUM()` function over the `quantity` attribute, using a sliding window consisting of the first (sorted) row of each partition, up to and including the current row (see Figure 1).

An alternative construction for the query is to specify the window separate from the functions that utilize it. This is useful when more than one window function is specified that are based on the same window. In the case of the query in Example 11, a construction that uses the `Window` clause (declaring a window identified by 'cumulative') is:

```

Select p.id, p.description, s.quantity, s.ship_date,
       Sum(s.quantity) Over(cumulative
                           Rows Between Unbounded Preceding
                                   and Current Row
                           ) as cumulative_qty
From sales_order_items s Join product p On (s.prod_id = p.id)
Where s.ship_date Between '2001-07-01' and '2001-08-31'
Window cumulative as (Partition by s.prod_id Order by s.ship_date)
Order by p.id

```

Note how the Window clause appears before the Order by clause in a <QUERY SPECIFICATION>. When using a Window clause, the following restrictions apply:

Mixing named and inlined window specifications

- the inlined portion of a <WINDOW SPECIFICATION> cannot have a Partition by clause;
- the window specified within the Window clause cannot contain a <WINDOW FRAME CLAUSE> (see grammar rule 34);
- either the inlined <WINDOW SPECIFICATION>, or the <WINDOW SPECIFICATION> specified in the Window clause, can contain a <WINDOW ORDER CLAUSE> (grammar rule 33), but not both.

#### EXAMPLE 12 (UTILIZING A WINDOW WITH MULTIPLE FUNCTIONS)

It is possible to define a single (named) window and compute multiple function results over it, as the following example demonstrates.

```

Select Product.Id, Description, Quantity,
       Rank() Over Qty AS Rank_quantity,
       Cume_Dist() Over Qty AS Dist_cume,
       Row_Number() Over Qty as Qty_order
From Product
Window Qty As (Order by Quantity Asc)
Order by Qty_order

```

### 4.1 Defining the window

A <WINDOW SPECIFICATION> permits one to define:

- a *partitioning* of the input rows through the use of a <WINDOW PARTITION CLAUSE> (grammar rule 30). If a <WINDOW PARTITION CLAUSE> is not specified, then the input is treated as a single partition.
- an *ordering* of the rows in each partition through the use of a <WINDOW ORDER CLAUSE> (grammar rule 33). If a <WINDOW ORDER CLAUSE> is not specified, then the input rows will be processed in an arbitrary order. Note, however, that the use of a <WINDOW FRAME UNIT> of Range

requires the existence of a `<WINDOW ORDER CLAUSE>`; otherwise, SQLSTATE '42WA9' will result. In the case of `Range`, the `<WINDOW ORDER CLAUSE>` may only consist of a single expression. Also, note that the ranking functions require a `<WINDOW ORDER CLAUSE>` because they are defined only over sorted input. As with an `Order by` clause in a `<QUERY SPECIFICATION>`, the default sort sequence is ascending (`Asc`).

- the *characteristics* of the window used to process all of the rows in the partition. These characteristics are specified through the use of the `<WINDOW FRAME CLAUSE>`. The `<WINDOW FRAME CLAUSE>` defines the beginning and end of the 'window', relative to the current row, that is used in the computation of window functions. If a `<WINDOW FRAME CLAUSE>` is not specified, the default window frame depends on whether or not a `<WINDOW ORDER CLAUSE>` is specified:
  - If the window specification contains a `<WINDOW ORDER CLAUSE>`, the window's start point is `Unbounded Preceding`, and the end point is `Current Row`—thus defining a varying-size window suitable for computing cumulative values.
  - If the window specification *does not* contain a `<WINDOW ORDER CLAUSE>`, the window's start point is `Unbounded Preceding`, and the end point is `Unbounded Following`—thus defining a window of fixed size, regardless of the current row.

Default window size

Note that a `<WINDOW FRAME CLAUSE>` cannot be used with a ranking function or the `ROW_NUMBER()` function.

#### EXAMPLE 13 (USING AN UNBOUNDED WINDOW)

The following query produces a result set consisting of all of the products accompanied by the total quantity of all products:

```
Select id, description, quantity,
       Sum(quantity) Over () as total
From product
```

As mentioned above, there are two `<WINDOW FRAME UNIT>`s that can be specified:

- One can specify the size of the window based on the physical number of rows relative to the current row in the window. Common examples are:
  - `Rows Between Unbounded Preceding and Current Row`. This window frame specifies a window whose start point is the beginning of each partition, and the end point of the window is the current row. This `<WINDOW FRAME CLAUSE>` is often used to construct windows that compute cumulative results, such as cumulative sums.

Typical window frames based on 'Rows'

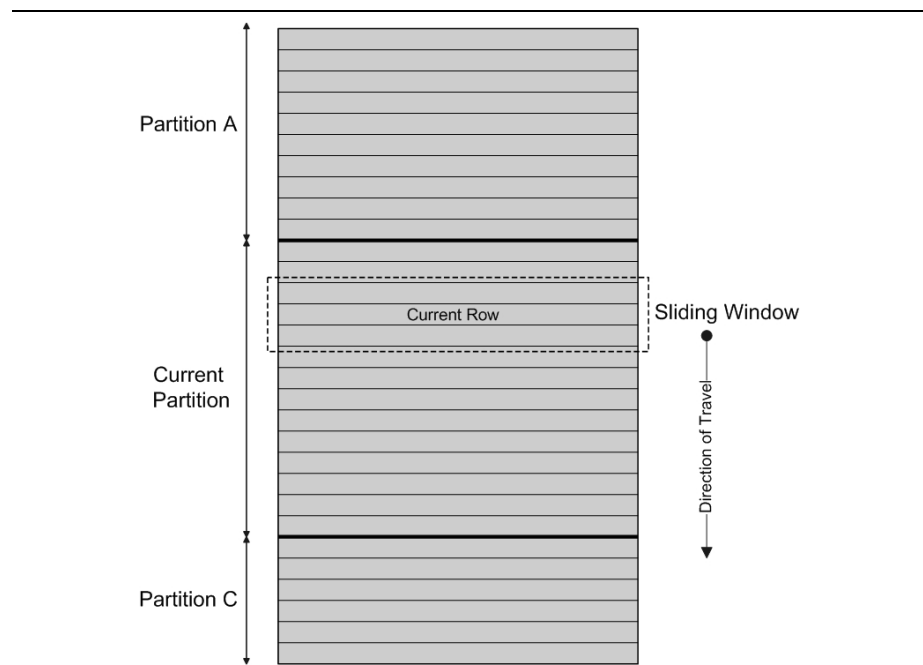


FIGURE 1: Schematic of a 3-row moving window with partitioned input.

- **Rows Between Unbounded Preceding and Unbounded Following.** This window frame specifies a fixed window (regardless of the current row) over the entire partition. Hence the value of a window aggregate function will be identical in each row of the partition.
- **Rows Between 1 Preceding and 1 Following.** This window frame specifies a fixed-sized ‘moving’ window over three adjacent rows (one each before and after the current row). One would typically use this `<WINDOW FRAME CLAUSE>` when, for example, computing a 3-day or 3-month moving average. However, one must be careful to avoid problems due to ‘gaps’ in the input to the window function. If the set of values is not continuous, it may make more sense to use `Range` rather than `Rows` because a window’s bounds based on `Range` will automatically handle adjacent rows with duplicate values, and will also not include other rows when there exists a ‘gap’ in the range.

Moving windows and  
boundary rows

Note that in the case of a moving window, it is assumed that rows containing `Null` values exist before the first row, and after the last row, in the input. What this means is that with a 3-row moving window, the computation for the last row in the input—the ‘current’ row—will include the immediately preceding row and a `Null` value.

- **Rows Between Current Row and Current Row.** One can specify this `<WINDOW FRAME CLAUSE>` to restrict the window to the current row only.
- **Rows Between 1 Preceding and 1 Preceding.** This window frame specifies a single row window consisting only of the preceding row (with respect to the current row). In combination with another window function that computes a value based on the current row only, this construction makes it possible to easily compute ‘deltas’, or differences in value, between adjacent rows (see Example 14 below).
- One can also define the window based on the *values* of a specific attribute in the input. This is specified using a `<WINDOW FRAME UNIT>` of `Range`. When using `Range`, the attribute(s) specified in the window’s `<WINDOW ORDER SPECIFICATION>` must have a numeric domain, as the server will simply add the `<UNSIGNED VALUE SPECIFICATION>` from a `<WINDOW FRAME PRECEDING>` or `<WINDOW FRAME FOLLOWING>` to the value of the attribute. The window size is based on the *values* of the attributes in the preceding and following rows relative to the current row. Using `Range` requires that the `<WINDOW ORDER SPECIFICATION>` in the window consist of only a single column.

Window size based on  
‘Range’

#### EXAMPLE 14 (COMPUTING DELTAS BETWEEN ADJACENT ROWS)

As mentioned above, using two windows—one window over the current row, the other over the previous row—provides a straightforward way of computing ‘deltas’, or changes, between adjacent rows. Consider the following query:

```
Select emp_id as employee, emp_lname as lastname,
       Sum(salary) Over (Order by birth_date
                        Rows Between Current Row
                        and Current Row) as curr,
       Sum(salary) Over (Order by birth_date
                        Rows Between 1 Preceding
                        and 1 Preceding) as prev,
       (curr - prev) as delta
From Employee
Where State in ('NY')
```

This query returns the following result *R*

	<i>Employee</i>	<i>Last name</i>	<i>Curr</i>	<i>Prev</i>	<i>Delta</i>	
<i>R</i>	1	913	Martel	55700.000	(Null)	(Null)
	2	1062	Blaikie	54900.000	55700.000	-800.000
	3	249	Guevara	42998.000	54900.000	-11902.000
	4	390	Davidson	57090.000	42998.000	14092.000
	5	102	Whitney	45700.000	57090.000	-11390.000
	6	1507	Wetherby	35745.000	45700.000	-9955.000
	7	1751	Ahmed	34992.000	35745.000	-753.000
	8	1157	Soo	39075.000	34992.000	4083.000

Note that although the window function `SUM()` is used, because of how the window is specified the sum will contain only the salary value of either the current or previous row. Note as well that the ‘prev’ value of the first row in the result is Null since it has no predecessor, and hence the ‘delta’ is Null as well.

In each of the examples above, the function used with the `Over` clause is the `SUM()` aggregate function. In fact, there are three classes of functions that can be used with a window: ranking functions, the row numbering function, and window aggregate functions. We now describe each of these function classes in turn, beginning with the ranking functions.

#### 4.2 Ranking functions

The ranking functions supported by SQL Anywhere (see grammar rule 7) are `RANK()`, `DENSE_RANK()`, `PERCENT_RANK()`, and `CUME_DIST()`. Ranking functions are not considered aggregate functions as they do not compute a result from multiple input rows in the same manner as, say, the `SUM()` aggregate function. Rather, each of these functions computes the *rank*, or relative ordering, of a row within a partition based on the value of a particular expression. Each set of rows within a partition are ranked independently; if the `Over` clause does not contain a `Partition by` clause, the entire input is treated as a single partition. Consequently one may not specify a `<WINDOW FRAME CLAUSE>` for a window utilized by a ranking function. It is possible to compose a query containing multiple ranking functions, each of which partition or sort the input rows differently.

All ranking functions require a `<WINDOW ORDER CLAUSE>` to specify the sortedness of the input rows upon which the ranking functions depend. If the `Order by` clause includes multiple expressions, the second and subsequent expressions will be used to break ties if the first expression has the same value in adjacent rows. Null values in SQL Anywhere are always sorted before any other value (in ascending sequence)<sup>7</sup>.

<sup>7</sup> At present, SQL Anywhere does not support the `Nulls First` and `Nulls Last` clauses that are defined in the ANSI SQL 2003 standard.

Often, use of a ranking function requires the use of a derived table that can restrict the ranked rows to the subset desired by the application, or re-sort the output rows in a different sequence. This technique is illustrated by the following example.

EXAMPLE 15 (RANK() FUNCTION)

Consider the following query, which determines the three most expensive products in the database:

```
Select Top 3 *
      From (Select description, quantity, unit_price,
                Rank() Over (Order by unit_price Desc) as Rank
            From product) as DT
Order by Rank
```

which returns the result *R*

	<i>Description</i>	<i>Quantity</i>	<i>Unit Price</i>	<i>Rank</i>	
<i>R</i>	1	Hooded Sweatshirt	39	24.00	1
	2	Zipped Sweatshirt	32	24.00	1
	3	Cotton Shorts	80	15.00	3

Note that a descending sort sequence was specified in the <WINDOW ORDER CLAUSE> so that the most expensive products had the lowest rank (rankings start at 1). Note as well that adjacent rows (1,2) that have the same value of the ordered expression ‘quantity’ have the same rank. With the RANK() function, the rank value ‘jumps’ in the case of ties—this is the difference between RANK() and its sister function, DENSE\_RANK().

**4.2.1** The DENSE\_RANK() function

The DENSE\_RANK() function

The DENSE\_RANK() function returns a series of ranks that are monotonically increasing with no ‘gaps’, or ‘jumps’.

EXAMPLE 16 (DENSE\_RANK() FUNCTION)

The following query is identical to that of Example 15 but instead uses the DENSE\_RANK() function:

```
Select Top 3 *
      From (Select description, quantity, unit_price,
                Dense_Rank() Over (Order by unit_price Desc) as Rank
            From product) as DT
Order by Rank
```

which yields the result *R*

	<i>Description</i>	<i>Quantity</i>	<i>Unit Price</i>	<i>Rank</i>	
<i>R</i>	1	Hooded Sweatshirt	39	24.00	1
	2	Zipped Sweatshirt	32	24.00	1
	3	Cotton Shorts	80	15.00	2

Because windows are evaluated after a query's **Group by** clause, one can specify complex requests that determine rankings based on the value of an aggregate function.

EXAMPLE 17 (USING RANKING WITH AGGREGATION)

The following query produces the top three salespeople in each region by their total sales within that region, along with the total sales for each region:

```
Select *
From (Select o.sales_rep, o.region,
        Sum(s.quantity * p.unit_price) as total_sales,
        Dense_rank() Over (Partition by o.region, Grouping(o.sales_rep)
                          Order by total_sales Desc) as sales_rank
      From product p, sales_order_items s, sales_order o
      Where p.id = s.prod_id and s.id = o.id
      Group by Grouping Sets( (o.sales_rep, o.region), o.region )) as DT
Where sales_rank <= 3
Order by region, sales_rank
```

This query returns the result *R*

	<i>Sales Rep</i>	<i>Region</i>	<i>Total Sales</i>	<i>Rank</i>	
<i>R</i>	1	(Null)	Canada	24768.00	1
	2	299	Canada	9312.00	1
	3	1596	Canada	3564.00	2
	4	856	Canada	2724.00	3
	5	(Null)	Central	134568.00	1
	6	299	Central	32592.00	1
	7	856	Central	14652.00	2
	8	467	Central	14352.00	3
	9	(Null)	Eastern	142038.00	1
	10	299	Eastern	21678.00	1
	11	902	Eastern	15096.00	2
	12	690	Eastern	14808.00	3
	13	(Null)	South	45262.00	1
	14	1142	South	6912.00	1
	15	667	South	6480.00	2
	16	949	South	5782.00	3
	17	(Null)	Western	37632.00	1
	18	299	Western	5640.00	1
	19	1596	Western	5076.00	2
	20	667	Western	4068.00	3

This query combines multiple groupings through the use of **Grouping Sets**. Hence the `<WINDOW PARTITION CLAUSE>` for the window uses the `GROUPING()` function to distinguish between detail rows that represent particular salespeople and the subtotal rows that list the total sales for an entire region. The subtotal rows by region, which have the value `Null` for the `sales_rep` attribute, each have the ranking value of 1 because the result's ranking order is restarted with each partition of the input; this ensures that the detail rows are ranked correctly starting at 1.

Finally, note in this example that the `DENSE_RANK()` function ranks the input over the aggregation of the total sales. An aliased `Select` list item is used as a shorthand in the `<WINDOW ORDER CLAUSE>`, an SQL vendor extension supported only by SQL Anywhere.

#### 4.2.2 Other ranking functions

SQL Anywhere also supports two other ranking functions. The first is the cumulative distribution function `CUME_DIST()`, sometimes defined as the inverse of percentile. `CUME_DIST()` computes the normalized position of a specific value relative to the entire set of values. The range of the function is between 0 and 1. For a specific  $i$ th value  $x_i$  in an ordered set of  $x$ -values of size  $n$ , the `CUME_DIST()` function is defined as

$$\text{CUME\_DIST}(x_i) = \frac{j}{n} \text{ such that } i \leq j \leq n \forall j : x_i \leq x_j \quad (18)$$

That is, the value returned by `CUME_DIST()` is the highest position within the ordered set of values equal to  $x_i$ , divided by the number  $n$  of values in the set. The semantics of the `CUME_DIST()` function is similar to that of the other ranking functions; `Null` values are treated like any other value in the domain.

#### EXAMPLE 18 (CUME\_DIST() FUNCTION)

```
Select description, quantity, unit_price,
       Cume_Dist() Over (Order by unit_price Asc) as CD
From product
Order by CD
```

which returns the result  $R$

	<i>Description</i>	<i>Quantity</i>	<i>Unit Price</i>	<i>CD</i>
1	Plastic Visor	28	7.00	0.2
2	Cloth Visor	36	7.00	0.2
3	Tank Top	28	9.00	0.4
4	Cotton Cap	112	9.00	0.4
5	Wool cap	12	10.00	0.5
6	Crew Neck	75	14.00	0.7
7	V-neck	54	14.00	0.7
8	Cotton Shorts	80	15.00	0.8
9	Zipped Sweatshirt	32	24.00	1.0
10	Hooded Sweatshirt	39	24.00	1.0

*R*

### Computing medians

CUME\_DIST() provides a simple method to determine the median of a set of values. CUME\_DIST() can be used to compute the median value successfully in the face of ties and whether the input contains an even or odd number of rows. Essentially, one need only determine the first row with a CUME\_DIST() value of greater than or equal to 0.5.

#### EXAMPLE 19 (COMPUTING THE MEDIAN)

The following query returns the product information for the product with the median unit price:

```
Select First *
From   (Select description, quantity, unit_price,
        Cume_dist() Over (Order by unit_price Asc) as CDist
        From product) as DT
Where  CDist >= 0.5
Order  by CDist
```

Finally, SQL Anywhere also supports the ANSI SQL PERCENT\_RANK() function, which is quite similar to CUME\_DIST(). However, rather than use the *position* of the value in the ordered set, PERCENT\_RANK() uses the result of the RANK() function in the numerator—recall that RANK() gives the same value in the case of ties. Hence PERCENT\_RANK() is defined as

$$\text{PERCENT\_RANK}(x_i) = \frac{\text{RANK}() \text{ of } x_i \text{ in its partition} - 1}{(n - 1)}. \quad (19)$$

As with CUME\_DIST(), the range of PERCENT\_RANK() is between 0 and 1. Row(s) with a RANK() of 1 will have a PERCENT\_RANK() of zero.

#### EXAMPLE 20 (PERCENT\_RANK() FUNCTION)

Consider the query from Example 18, this time rewritten to use PERCENT\_RANK():

```
Select description, quantity, unit_price,
       Percent_rank() Over (Order by unit_price Asc) as PctRank
From product
Order by PctRank
```

which returns the result *R*

	<i>Description</i>	<i>Quantity</i>	<i>Unit Price</i>	<i>PctRank</i>	
<i>R</i>	1	Plastic Visor	28	7.00	0.0
	2	Cloth Visor	36	7.00	0.0
	3	Tank Top	28	9.00	0.2222222222222222
	4	Cotton Cap	112	9.00	0.2222222222222222
	5	Wool cap	12	10.00	0.4444444444444444
	6	Crew Neck	75	14.00	0.5555555555555556
	7	V-neck	54	14.00	0.5555555555555556
	8	Cotton Shorts	80	15.00	0.7777777777777778
	9	Zipped Sweatshirt	32	24.00	0.8888888888888888
	10	Hooded Sweatshirt	39	24.00	0.8888888888888888

### 4.3 Numbering rows

It is often necessary for an SQL statement to uniquely number the rows in its result. For several releases, SQL Anywhere has provided a special vendor extension, the `NUMBER(*)` builtin function, that can be used, under various restrictions, to provide a unique number for each row. The ANSI SQL 2003 standard, however, defines a `ROW_NUMBER()` function that can be used instead of `NUMBER(*)` and, moreover, is not subject to the limitations of `NUMBER(*)`.

`ROW_NUMBER()` versus  
`NUMBER(*)`

As with the ranking functions, `ROW_NUMBER()` requires a `<WINDOW ORDER CLAUSE>`. As well, `ROW_NUMBER()`, like the ranking functions, can return non-deterministic results when the window's `Order by` clause is over non-unique expressions—row order is unpredictable in the case of ties. `ROW_NUMBER()` is designed to work only over the entire partition, hence a `<WINDOW FRAME CLAUSE>` cannot be specified with a `ROW_NUMBER()` function.

#### EXAMPLE 21 (NUMBERING ROWS IN A RESULT)

One can use `ROW_NUMBER()` in any situation in which one can use a ranking function. This makes it possible to use `ROW_NUMBER()` in a derived table, so that additional restrictions, even joins, can be made over the `ROW_NUMBER()` values:

```
Select *
From (Select description, quantity,
       Row_Number() Over (Order by id Asc) as rownum
     From product) as DT
Where rownum <= 3
Order by rownum
```

#### 4.4 Window aggregate functions

As demonstrated above, window aggregate functions can be used to perform aggregation within a window. The window aggregate functions supported by SQL Anywhere include `COUNT()`, `SUM()`, `MIN()`, `MAX()`, `AVG()`, `FIRST_VALUE()`, `LAST_VALUE()`, and the six variance and standard deviation functions (see grammar rule 8). Note in particular that SQL Anywhere's `LIST()` aggregate function, a vendor extension, cannot be used with a `<WINDOW SPECIFICATION>`.

Because one can define a window with varying start and end points, window aggregate functions can support moving sum, moving average, moving minimum or maximum, cumulative sum, and various statistical functions. In addition, complex data analysis often requires multiple levels of aggregation. Window partitioning and ordering, in addition to, or instead of, a `Group by` clause, offers application developers considerable flexibility in the composition of complex SQL queries.

##### EXAMPLE 22 (USING A WINDOW FUNCTION)

The following query returns a result set that shows a list of the products that sold higher than the average number of products sold per annum:

```
Select *
From (Select Year(order_date) as Year, prod_id,
           Sum(quantity) as Q,
           Avg(Sum(quantity)) Over (Partition by Year) as Average
      From sales_order Key Join sales_order_items
      Group by Year(order_date), prod_id) as DT
Where DT.Q > DT.Average
Order by DT.Year
```

##### EXAMPLE 23 (COMPUTING A MOVING AVERAGE)

In this example, `AVG()` is used as a window function to compute the moving average of all product sales, by month, in the year 2000. Note that the `<WINDOW SPECIFICATION>` uses a `RANGE` clause, which causes the window bounds to be computed based on the *month value*, and not simply by an adjacent row as with the `ROWS` clause. Using `ROWS` would yield different results if, for example, some or all of the products happened to have no sales at all in a particular month.

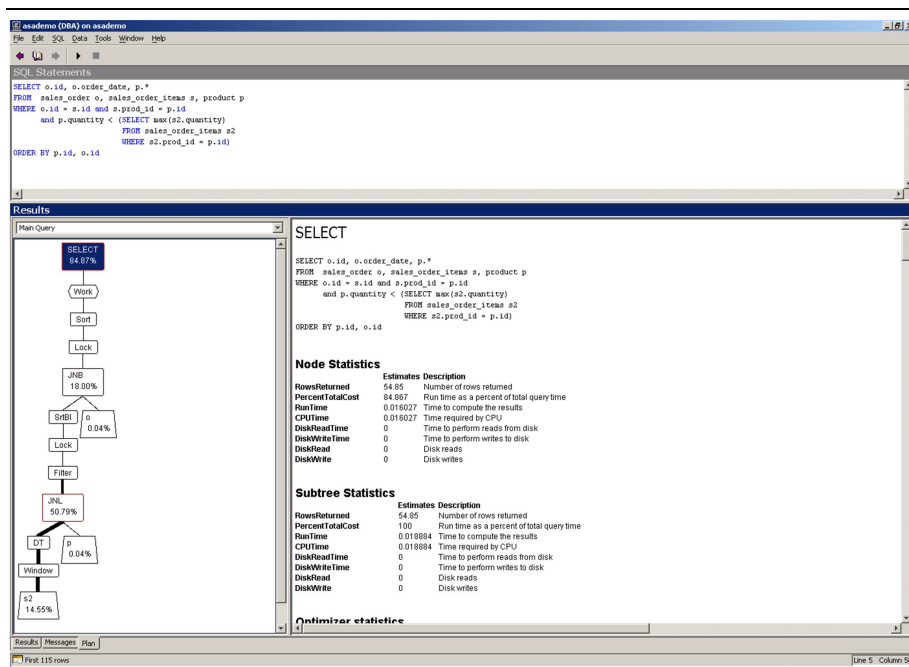


FIGURE 2: Graphical plan for the query in Example 24.

```
Select *
From (Select s.prod_id, Month(o.order_date) as julian_month,
Sum(s.quantity) as sales,
Avg(Sum(s.quantity))
Over (Partition by s.prod_id
Order by Month(o.order_date) Asc
Range Between 1 Preceding and 1 Following)
as average_sales
From sales_order_items s Key Join sales_order o
Where Year(o.order_date) = 2000
Group by s.prod_id, Month(o.order_date)) as DT
Order by 1,2
```

## EXAMPLE 24 (ELIMINATING CORRELATED SUBQUERIES)

In many situations one requires the ability to compare a particular column value with a maximum or minimum value. Often one composes these queries as nested queries involving a correlated attribute (sometimes known as an outer reference). As one example, consider the following query, which lists all orders, including product information, where the product quantity-on-hand cannot cover the maximum single order for that product:

```

Select o.id, o.order_date, p.*
From sales_order o, sales_order_items s, product p
Where o.id = s.id and s.prod_id = p.id
      and p.quantity < (Select max(s2.quantity)
                        From sales_order_items s2
                        Where s2.prod_id = p.id)
Order by p.id, o.id

```

The graphical plan for this query is shown in Figure 2. Note how the query optimizer has transformed this nested query to a join of the `product` and `sales_order` tables with a derived table, denoted by the correlation name ‘DT’, that contains a window function. The SQL Anywhere optimizer performs such transformations on a cost basis, after evaluating the expected cost of utilizing different execution strategies including naive nested iteration.

Rather than relying on SQL Anywhere’s query optimizer to transform the correlated subquery into a join with a derived table—which can only be done for straightforward cases due to the complexity of the semantic analysis—one can compose such queries using a window function:

```

Select order_qty.id, o.order_date, p.*
From (Select s.id, s.prod_id,
           Max(s.quantity) Over (Partition by s.prod_id
                               Order by s.prod_id) as max_q
      From sales_order_items s) as order_qty,
      product p, sales_order o
Where p.id = prod_id and o.id = order_qty.id and p.quantity < max_q
Order by p.id, o.id

```

#### EXAMPLE 25 (COMPLEX ANALYTICS)

Consider the following query, which lists the top salespeople (defined by total sales) for each product in the database:

```

Select s.prod_id as product, o.sales_rep,
       Sum(s.quantity) as total_quantity,
       Sum(s.quantity * p.unit_price) as total_sales
From sales_order o Key Join sales_order_items s Key Join product p
Group by s.prod_id, o.sales_rep
Having total_sales = (Select First Sum(s2.quantity *
                                     p2.unit_price) as sum_sales
                    From sales_order o2 Key Join
                        sales_order_items s2 Key Join product p2
                    Where s2.prod_id = s.prod_id
                    Group by o2.sales_rep
                    Order by sum_sales Desc)
Order by s.prod_id

```

This query returns the result:

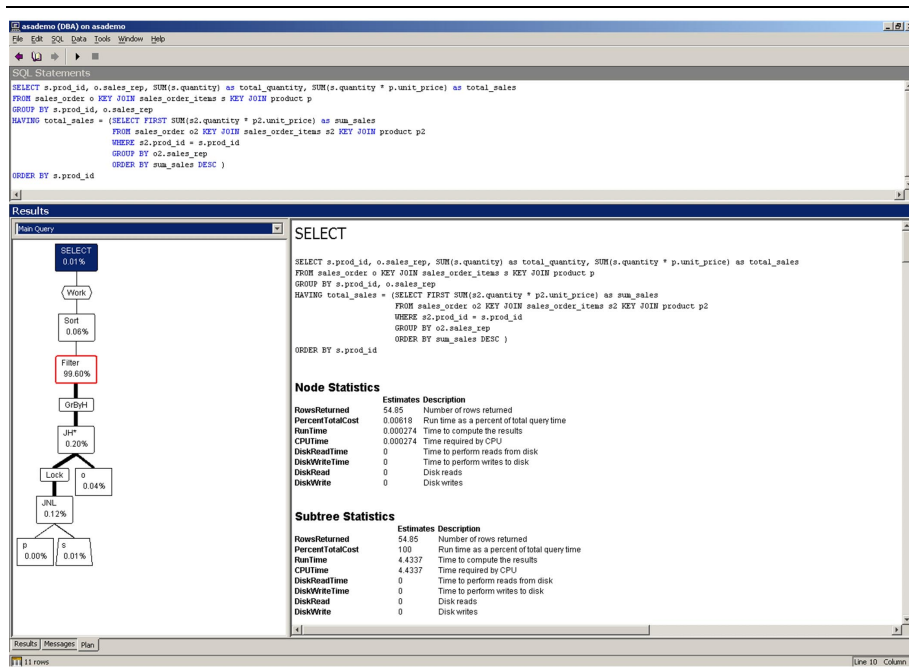


FIGURE 3: Graphical plan for the main query block in Example 25.

$R$

	<i>Product</i>	<i>Sales Rep</i>	<i>Total Qty</i>	<i>Total Sales</i>
1	300	299	660	5940.00
2	301	299	516	7224.00
3	302	299	336	4704.00
4	400	299	458	4122.00
5	401	902	360	3600.00
6	500	949	360	2520.00
7	501	690	360	2520.00
8	501	949	360	2520.00
9	600	299	612	14688.00
10	601	299	636	15264.00
11	700	299	1008	15120.00

The original query is composed using a correlated subquery which determines the highest sales for any particular product, as `prod_id` is the subquery’s correlated outer reference. Using a nested query, however, is often an expensive option, as in this case (see the graphical query plans in Figures 3 and 4). This is because the subquery involves not only a `Group by` clause, but also uses an `Order by` clause within it—a feature of SQL Anywhere—that makes it impossi-

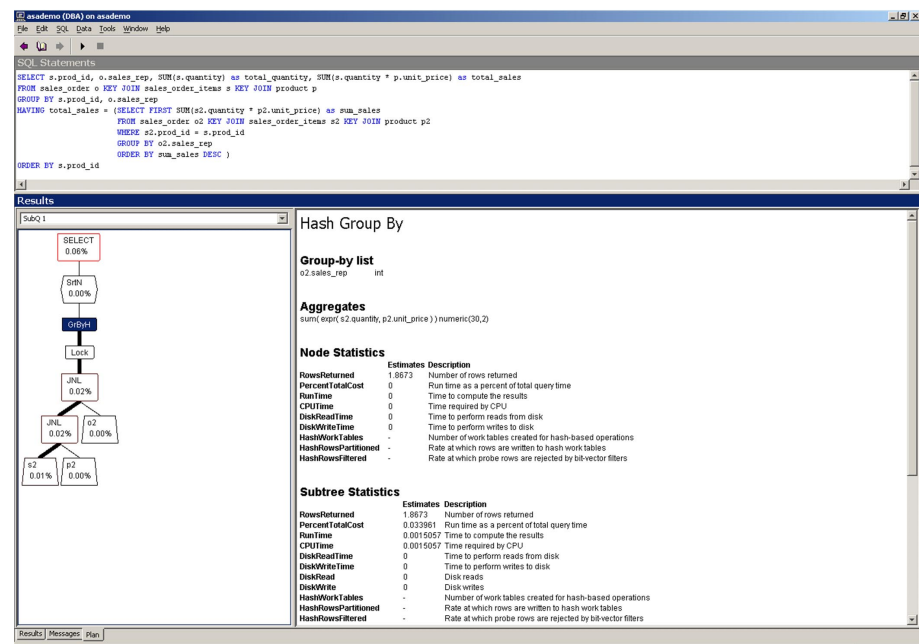


FIGURE 4: Graphical plan for the subquery in Example 25.

ble for SQL Anywhere’s query optimizer to rewrite this nested query as a join while retaining the same semantics (as was done in Example 24).

Consequently, during query execution the subquery will be evaluated for each (derived) row computed in the outer block. Note the expensive ‘Filter’ predicate in the graphical plan: the optimizer estimates that 99% of the query’s execution cost is due to this plan operator. The plan for the subquery, shown in Figure 4, clearly illustrates why the filter operator for the main block is so expensive: the subquery involves two nested-loop joins, a hashed Group by operation, and a sort.

#### EXAMPLE 26 (COMPLEX QUERY USING A WINDOW FUNCTION)

A rewriting of the query in Example 25, shown below, computes the identical result much more efficiently by using a ranking function.

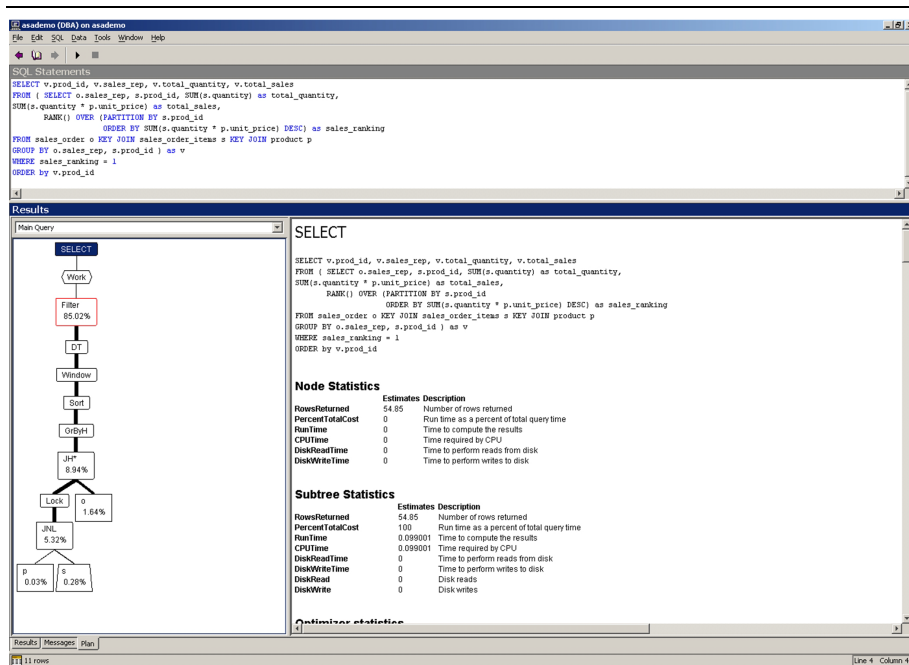


FIGURE 5: Execution plan of the query from Example 26, using a window function.

```

Select v.prod_id, v.sales_rep, v.total_quantity, v.total_sales
From ( Select o.sales_rep, s.prod_id,
        Sum(s.quantity) as total_quantity,
        Sum(s.quantity * p.unit_price) as total_sales,
        Rank() Over (Partition by s.prod_id
                    Order by Sum(s.quantity *
                                p.unit_price) Desc) as sales_ranking
From sales_order o Key Join
    sales_order_items s Key Join product p
Group by o.sales_rep, s.prod_id) as v
Where sales_ranking = 1
Order by v.prod_id

```

Recall that a window operator is computed after the processing of a Group by clause and prior to the evaluation of the Select list items and the query's Order by clause. As seen in the graphical plan, after the join of the three tables, the joined rows are grouped by the combination of the `sales_rep` and `prod_id` ('product identifier') attributes. Consequently, the `Sum()` aggregate functions of `total_quantity` and `total_sales` can be computed for each combination of

sales representative and product identifier.

Following the evaluation of the **Group by** clause, the **Rank()** function is then computed to rank the rows in the intermediate result in descending sequence by total sales, using a window. Note that the **<WINDOW SPECIFICATION>** involves a **Partition by** clause—by doing so, the result of the **Group by** clause is *repartitioned* (grouped) again, but this time only by product identifier. Hence the **Rank()** function ranks the rows for each product—in descending order of total sales—but for *all* sales representatives that have sold that product. With this ranking, determining the top salespersons simply requires restricting the derived table's result to reject those rows where the rank is not one. In the case of ties (rows 7 and 8 in the result *R* above), **Rank()** returns the same value. Hence both salespeople 690 and 949 appear in the final result, as desired.

#### 4.4.1 The **FIRST\_VALUE()** and **LAST\_VALUE()** functions

The functions **FIRST\_VALUE()** and **LAST\_VALUE()** return values from the first and last rows of a window. This allows a query to access values from multiple rows at once without the need for a self-join.

These two functions are different from the other window aggregate functions in that they can only be used with a window. Also, unlike the other window aggregate functions, **FIRST\_VALUE()** and **LAST\_VALUE()** allow the **IGNORE NULLS** clause. If **IGNORE NULLS** is specified, the first or last non-null value of the desired expression is returned. Otherwise, the first or last value is returned whether or not it is null.

##### EXAMPLE 27 (FIRST ENTRY IN A GROUP)

The **FIRST\_VALUE()** function can be used to retrieve the first entry in an ordered group of values. The following query returns, for each order, the product identifier of the order's first item; that is, the **product\_id** of the item with the smallest **line\_id** for each order.

```
Select Distinct id,
       First_value( prod_id ) over ( Partition by id order by line_id )
From sales_order_items
Order by id
```

The query uses the **Distinct** keyword to remove duplicates. Without **Distinct**, a duplicate row would be returned for each item in each order.

##### EXAMPLE 28 (PERCENTAGE OF HIGHEST SALES)

A common use of the **FIRST\_VALUE()** function is to compare a value in each row with the maximum or minimum value within the current group. The following query computes the total sales for each sales representative, as well as its relationship with the maximum total sales for the same product, expressed as a percentage.

```

Select s.prod_id as prod_id,
       o.sales_rep as sales_rep,
       Sum(s.quantity * p.unit_price) as total_sales,
       100 * total_sales /
         ( First_value(Sum(s.quantity * p.unit_price))
           Over Sales_Window ) as total_sales_percentage
FROM sales_order o Key Join
sales_order_items s Key Join product p
Window SalesWindow as (Partition by s.prod_id
                        Order by Sum(s.quantity *
                                      p.unit_price) Desc)
Group by o.sales_rep, s.prod_id
Order by s.prod_id

```

## EXAMPLE 29 (DATA DENSIFICATION)

The `FIRST_VALUE` and `LAST_VALUE` functions are useful for data densification when used with the `IGNORE NULLS` clause. For example, suppose the sales representative with the highest total sales each day wins the distinction of ‘Representative of the Day’. The following query lists the winning sales representatives for the first week of April, 2001:

```

Select v.order_date, v.sales_rep as rep_of_the_day
From ( Select o.sales_rep, o.order_date,
            Rank() Over (Partition by o.order_date
                        Order by Sum(s.quantity *
                                      p.unit_price) Desc) as sales_ranking
      From sales_order o Key Join
            sales_order_items s Key Join
            product p
      Group by o.sales_rep, o.order_date) as v
Where v.sales_ranking = 1
and v.order_date between '2001-04-01' and '2001-04-07'
Order by v.order_date

```

This query is very similar to the one in Example 26, except that partitioning in the derived table `v` occurs by `order_date` rather than by `part_id`. The results of the query are shown below.

	<i>order_date</i>	<i>rep_of_the_day</i>	
<i>R</i>	1	2000-01-01	949
	2	2000-01-02	856
	3	2000-01-05	902
	4	2000-01-06	467
	5	2000-01-07	299

Note that no results are returned for days in which no sales were made. This result set can be made dense by declaring that on days in which no sales were

made, the winning representative from the previous day will remain the Representative of the Day. This data densification is achieved by the following query.

```

Select d.order_date,
      Last_value( v.sales_rep Ignore Nulls )
      Over ( Order by d.order_date )
      As rep_of_the_day
From ( Select o.sales_rep, o.order_date,
          Rank() Over (Partition by o.order_date
                      Order by Sum(s.quantity *
                                p.unit_price) Desc) as sales_ranking
      From sales_order o Key Join
          sales_order_items s Key Join product p
      Group by o.sales_rep, o.order_date) as v
Right Outer Join ( Select dateadd( day, row_num-1, '2001-04-01' )
                  as order_date
                  From rowgenerator
                  Where row_num <= 7 ) as d
On v.order_date = d.order_date and sales_ranking = 1
Order by d.order_date

```

The query returns the following results:

	<i>order_date</i>	<i>rep_of_the_day</i>
1	2000-01-01	949
2	2000-01-02	856
3	2000-01-03	856
4	2000-01-04	856
5	2000-01-05	902
6	2000-01-06	467
7	2000-01-07	299

The derived table *v* from the previous query is joined to a derived table *d*, which contains all the dates under consideration. This yields a row for each desired day, but this outer join will contain NULL in the *sales\_rep* column for dates in which no sales were made. The use of the LAST\_VALUE function solves this problem by defining *rep\_of\_the\_day* for a given row to be the last non-null value of *sales\_rep* leading up to the corresponding day.

## **A** BNF Grammar for OLAP Functions

The following Backus-Naur Form grammar outlines the specific syntactic support for the various ANSI SQL analytic functions implemented in SQL Anywhere.

- (1)  $\langle \text{SELECT LIST EXPRESSION} \rangle ::=$   
 $\langle \text{EXPRESSION} \rangle$   
 $| \langle \text{GROUP BY EXPRESSION} \rangle$   
 $| \langle \text{AGGREGATE FUNCTION} \rangle$

- | <GROUPING FUNCTION>  
 | <TABLE COLUMN>  
 | <WINDOWED TABLE FUNCTION>
- (2) <QUERY SPECIFICATION> ::=
- <FROM CLAUSE>  
 [ <WHERE CLAUSE> ]  
 [ <GROUP BY CLAUSE> ]  
 [ <HAVING CLAUSE> ]  
 [ <WINDOW CLAUSE> ]  
 [ <ORDER BY CLAUSE> ]
- (3) <ORDER BY CLAUSE> ::= <ORDER SPECIFICATION>
- (4) <GROUPING FUNCTION> ::=
- GROUPING** <LEFT PAREN> <GROUP BY EXPRESSION> <RIGHT PAREN>
- (5) <WINDOWED TABLE FUNCTION> ::=
- <WINDOWED TABLE FUNCTION TYPE> **OVER** <WINDOW NAME OR SPECIFICATION>
- (6) <WINDOWED TABLE FUNCTION TYPE> ::=
- <RANK FUNCTION TYPE> <LEFT PAREN> <RIGHT PAREN>  
 | **ROW\_NUMBER** <LEFT PAREN> <RIGHT PAREN>  
 | <WINDOW AGGREGATE FUNCTION>
- (7) <RANK FUNCTION TYPE> ::=
- RANK** | **DENSE\_RANK** | **PERCENT\_RANK** | **CUME\_DIST**
- (8) <WINDOW AGGREGATE FUNCTION> ::=
- <SIMPLE WINDOW AGGREGATE FUNCTION>  
 | <STATISTICAL AGGREGATE FUNCTION>
- (9) <AGGREGATE FUNCTION> ::=
- <DISTINCT AGGREGATE FUNCTION>  
 | <SIMPLE AGGREGATE FUNCTION>  
 | <STATISTICAL AGGREGATE FUNCTION>  
 | <VALUE AGGREGATE FUNCTION>
- (10) <DISTINCT AGGREGATE FUNCTION> ::=
- <BASIC AGGREGATE FUNCTION TYPE> <LEFT PAREN> **DISTINCT** <EXPRESSION> <RIGHT PAREN>  
 | **LIST** <LEFT PAREN> **DISTINCT** <EXPRESSION> [ <COMMA> <DELIMITER> ]  
 [ <ORDER SPECIFICATION> ] <RIGHT PAREN>
- (11) <BASIC AGGREGATE FUNCTION TYPE> ::=
- SUM** | **MAX** | **MIN** | **AVG** | **COUNT**
- (12) <SIMPLE AGGREGATE FUNCTION> ::=
- <SIMPLE AGGREGATE FUNCTION TYPE> <LEFT PAREN> <EXPRESSION> <RIGHT PAREN>  
 | **LIST** <LEFT PAREN> <EXPRESSION> [ <COMMA> <DELIMITER> ]  
 [ <ORDER SPECIFICATION> ] <RIGHT PAREN>
- (13) <SIMPLE AGGREGATE FUNCTION TYPE> ::= <SIMPLE WINDOW AGGREGATE FUNCTION TYPE>
- (14) <SIMPLE WINDOW AGGREGATE FUNCTION> ::=
- <SIMPLE WINDOW AGGREGATE FUNCTION TYPE> <LEFT PAREN> <EXPRESSION> <RIGHT PAREN>  
 | <GROUPING FUNCTION>
- (15) <SIMPLE WINDOW AGGREGATE FUNCTION TYPE> ::=
- <BASIC AGGREGATE FUNCTION TYPE>  
 | **STDDEV** | **STDDEV\_POP** | **STDDEV\_SAMP**  
 | **VARIANCE** | **VAR\_POP** | **VAR\_SAMP**
- (16) <STATISTICAL AGGREGATE FUNCTION> ::=
- <STATISTICAL AGGREGATE FUNCTION TYPE> <LEFT PAREN>  
 <DEPENDENT EXPRESSION> <COMMA> <INDEPENDENT EXPRESSION> <RIGHT PAREN>
- (17) <STATISTICAL AGGREGATE FUNCTION TYPE> ::=
- CORR** | **COVAR\_POP** | **COVAR\_SAMP** | **REGR\_R2** |  
**REGR\_INTERCEPT** | **REGR\_COUNT** | **REGR\_SLOPE** | **REGR\_SXX** |  
**REGR\_SXY** | **REGR\_SYY** | **REGR\_AVGY** | **REGR\_AVGX**

- (18)  $\langle \text{VALUE AGGREGATE FUNCTION} \rangle ::=$   
 $\langle \text{VALUE AGGREGATE FUNCTION TYPE} \rangle \langle \text{LEFT PAREN} \rangle \langle \text{EXPRESSION} \rangle$   
 $[\text{ IGNORE NULLS } ] \langle \text{RIGHT PAREN} \rangle$
- (19)  $\langle \text{VALUE AGGREGATE FUNCTION TYPE} \rangle ::=$   
**FIRST\_VALUE** | **LAST\_VALUE**
- (20)  $\langle \text{WINDOW NAME OR SPECIFICATION} \rangle ::=$   
 $\langle \text{WINDOW NAME} \rangle$  |  $\langle \text{IN-LINE WINDOW SPECIFICATION} \rangle$
- (21)  $\langle \text{WINDOW NAME} \rangle ::= \langle \text{IDENTIFIER} \rangle$
- (22)  $\langle \text{IN-LINE WINDOW SPECIFICATION} \rangle ::= \langle \text{WINDOW SPECIFICATION} \rangle$
- (23)  $\langle \text{WINDOW CLAUSE} \rangle ::= \text{WINDOW} \langle \text{WINDOW DEFINITION LIST} \rangle$
- (24)  $\langle \text{WINDOW DEFINITION LIST} \rangle ::=$   
 $\langle \text{WINDOW DEFINITION} \rangle [ \{ \langle \text{COMMA} \rangle \langle \text{WINDOW DEFINITION} \rangle \} \dots ]$
- (25)  $\langle \text{WINDOW DEFINITION} \rangle ::=$   
 $\langle \text{NEW WINDOW NAME} \rangle \text{ AS } \langle \text{WINDOW SPECIFICATION} \rangle$
- (26)  $\langle \text{NEW WINDOW NAME} \rangle ::= \langle \text{WINDOW NAME} \rangle$
- (27)  $\langle \text{WINDOW SPECIFICATION} \rangle ::=$   
 $\langle \text{LEFT PAREN} \rangle \langle \text{WINDOW SPECIFICATION DETAILS} \rangle \langle \text{RIGHT PAREN} \rangle$
- (28)  $\langle \text{WINDOW SPECIFICATION DETAILS} \rangle ::=$   
 $[ \langle \text{EXISTING WINDOW NAME} \rangle ]$   
 $[ \langle \text{WINDOW PARTITION CLAUSE} \rangle ]$   
 $[ \langle \text{WINDOW ORDER CLAUSE} \rangle ]$   
 $[ \langle \text{WINDOW FRAME CLAUSE} \rangle ]$
- (29)  $\langle \text{EXISTING WINDOW NAME} \rangle ::= \langle \text{WINDOW NAME} \rangle$
- (30)  $\langle \text{WINDOW PARTITION CLAUSE} \rangle ::=$   
**PARTITION BY**  $\langle \text{WINDOW PARTITION EXPRESSION LIST} \rangle$
- (31)  $\langle \text{WINDOW PARTITION EXPRESSION LIST} \rangle ::=$   
 $\langle \text{WINDOW PARTITION EXPRESSION} \rangle$   
 $[ \{ \langle \text{COMMA} \rangle \langle \text{WINDOW PARTITION EXPRESSION} \rangle \} \dots ]$
- (32)  $\langle \text{WINDOW PARTITION EXPRESSION} \rangle ::= \langle \text{EXPRESSION} \rangle$
- (33)  $\langle \text{WINDOW ORDER CLAUSE} \rangle ::= \langle \text{ORDER SPECIFICATION} \rangle$
- (34)  $\langle \text{WINDOW FRAME CLAUSE} \rangle ::=$   
 $\langle \text{WINDOW FRAME UNIT} \rangle$   
 $\langle \text{WINDOW FRAME EXTENT} \rangle$
- (35)  $\langle \text{WINDOW FRAME UNIT} \rangle ::= \text{ROWS} \mid \text{RANGE}$
- (36)  $\langle \text{WINDOW FRAME EXTENT} \rangle ::= \langle \text{WINDOW FRAME START} \rangle \mid \langle \text{WINDOW FRAME BETWEEN} \rangle$
- (37)  $\langle \text{WINDOW FRAME START} \rangle ::=$   
**UNBOUNDED PRECEDING**  
 $| \langle \text{WINDOW FRAME PRECEDING} \rangle$   
 $| \text{CURRENT ROW}$
- (38)  $\langle \text{WINDOW FRAME PRECEDING} \rangle ::= \langle \text{UNSIGNED VALUE SPECIFICATION} \rangle \text{ PRECEDING}$
- (39)  $\langle \text{WINDOW FRAME BETWEEN} \rangle ::=$   
**BETWEEN**  $\langle \text{WINDOW FRAME BOUND 1} \rangle$  **AND**  $\langle \text{WINDOW FRAME BOUND 2} \rangle$
- (40)  $\langle \text{WINDOW FRAME BOUND 1} \rangle ::= \langle \text{WINDOW FRAME BOUND} \rangle$
- (41)  $\langle \text{WINDOW FRAME BOUND 2} \rangle ::= \langle \text{WINDOW FRAME BOUND} \rangle$
- (42)  $\langle \text{WINDOW FRAME BOUND} \rangle ::=$   
 $\langle \text{WINDOW FRAME START} \rangle$   
 $| \text{UNBOUNDED FOLLOWING}$   
 $| \langle \text{WINDOW FRAME FOLLOWING} \rangle$
- (43)  $\langle \text{WINDOW FRAME FOLLOWING} \rangle ::= \langle \text{UNSIGNED VALUE SPECIFICATION} \rangle \text{ FOLLOWING}$
- (44)  $\langle \text{GROUP BY EXPRESSION} \rangle ::= \langle \text{EXPRESSION} \rangle$
- (45)  $\langle \text{SIMPLE GROUP BY TERM} \rangle ::=$   
 $\langle \text{GROUP BY EXPRESSION} \rangle$   
 $| \langle \text{LEFT PAREN} \rangle \langle \text{GROUP BY EXPRESSION} \rangle \langle \text{RIGHT PAREN} \rangle$   
 $| \langle \text{LEFT PAREN} \rangle \langle \text{RIGHT PAREN} \rangle$

- (46)  $\langle \text{SIMPLE GROUP BY TERM LIST} \rangle ::=$   
 $\langle \text{SIMPLE GROUP BY TERM} \rangle [ \{ \langle \text{COMMA} \rangle \langle \text{SIMPLE GROUP BY TERM} \rangle \} \dots ]$
- (47)  $\langle \text{COMPOSITE GROUP BY TERM} \rangle ::=$   
 $\langle \text{LEFT PAREN} \rangle \langle \text{SIMPLE GROUP BY TERM} \rangle$   
 $[ \{ \langle \text{COMMA} \rangle \langle \text{SIMPLE GROUP BY TERM} \rangle \} \dots ]$   
 $\langle \text{RIGHT PAREN} \rangle$
- (48)  $\langle \text{ROLLUP TERM} \rangle ::= \text{ROLLUP} \langle \text{COMPOSITE GROUP BY TERM} \rangle$
- (49)  $\langle \text{CUBE TERM} \rangle ::= \text{CUBE} \langle \text{COMPOSITE GROUP BY TERM} \rangle$
- (50)  $\langle \text{GROUP BY TERM} \rangle ::=$   
 $\langle \text{SIMPLE GROUP BY TERM} \rangle$   
 $| \langle \text{COMPOSITE GROUP BY TERM} \rangle$   
 $| \langle \text{ROLLUP TERM} \rangle$   
 $| \langle \text{CUBE TERM} \rangle$
- (51)  $\langle \text{GROUP BY TERM LIST} \rangle ::=$   
 $\langle \text{GROUP BY TERM} \rangle [ \{ \langle \text{COMMA} \rangle \langle \text{GROUP BY TERM} \rangle \} \dots ]$
- (52)  $\langle \text{GROUP BY CLAUSE} \rangle ::= \text{GROUP BY} \langle \text{GROUPING SPECIFICATION} \rangle$
- (53)  $\langle \text{GROUPING SPECIFICATION} \rangle ::=$   
 $\langle \text{GROUP BY TERM LIST} \rangle$   
 $| \langle \text{SIMPLE GROUP BY TERM LIST} \rangle \text{ WITH ROLLUP}$   
 $| \langle \text{SIMPLE GROUP BY TERM LIST} \rangle \text{ WITH CUBE}$   
 $| \langle \text{GROUPING SETS SPECIFICATION} \rangle$
- (54)  $\langle \text{GROUPING SETS SPECIFICATION} \rangle ::=$   
 $\text{GROUPING SETS} \langle \text{LEFT PAREN} \rangle \langle \text{GROUP BY TERM LIST} \rangle \langle \text{RIGHT PAREN} \rangle$
- (55)  $\langle \text{ORDER SPECIFICATION} \rangle ::= \text{ORDER BY} \langle \text{SORT SPECIFICATION LIST} \rangle$

## References

- [1] Jay Devore and Roxy Peck. *Statistics: The Exploration and Analysis of Data*. West Publishing Company, St. Paul, Minnesota, 1986.
- [2] International Standards Organization. (ANSI/ISO) 9075-2, *SQL Foundation*, September 1999.
- [3] International Standards Organization. (ANSI/ISO) 9075:1999-AM1, *On-Line Analytical Processing SQL/OLAP Amendment*, November 1999.
- [4] International Standards Organization. (ANSI/ISO) 9075-2, *SQL Foundation*, September 2003.
- [5] David S. Moore and George P. McCabe. *Introduction to the Practice of Statistics*. W. H. Freeman and Company, New York, New York, 1989.
- [6] Eric Weisstein. Mathworld encyclopedia of mathematics, September 2005. <http://mathworld.wolfram.com>.