SAP SQL Anywhere テクニカルホワイトペーパー

SAP SQL Anywhere 10 の分析関数

G. N. Paulley　　B. Lucier 18 July 2007

https://archive.sap.com/kmuuid2/b0a254c3-ef4c-3010-59ad-ff11d0d27029/Analytic%20Functions%20in%20SQL%20Anywhere.pdf

(*この文書は上記の日本語訳版です。Version 10 について書かれたもので以降のバージョンでは変更されている場合があります。)

要約

ANSI 標準 SQL 2003 で追加された分析関数 T431、T611、および T612 は、1 つの SQL 文で複雑なデータ分析を行うもので、これらの分析機能は OLAP または On-Line Analytical Processing と呼ばれています。強化された SQL 言語には、分散、標準偏差、相関、回帰測定を計算する関数、そして window 関数と呼ばれる新しい aggregate 関数のクラスなどがあります。これらの関数は、これまで 自己結合、相関サブクエリー、テンポラリーテーブル、あるいはこれら全ての組み合わせが必要であったランキング、パーセンタイル、移動平均、累積和などの一般的な分析計算を ANSI 標準 SQL 1999 に含められた **Group by** 拡張 **Cube**、**Rollup**、**Grouping Sets** とともに1つの SQL 文で計算するための効率的なメカニズムです。このホワイトペーパーでは、SAP SQL Anywhere version 10.0.1 で利用できる OLAP 関数について解説します。

目次

テーブルのリスト

図のリスト

1 はじめに

ANSI 標準 SQL [4] に複雑なデータ分析を含めるという拡張は、1999 11月 [3] の ANSI 標準 SQL [2] への修正提案が最初でした。これらの拡張関数は、(「コア」の SQL サポートとは別に) ANSI 標準 SQL 2003 の機能 T431、T611、T612 として盛り込まれています。SAP SQL Anywhere では、これらの SQL 強化の一部を SAP SQL Anywhere の以前のバージョンにすでに追加済みでしたが、9.0.1 および 9.0.2 では、これらの分析関数を完全にサポートしました。このテクニカルホワイトペーパーでは、version 9.0.1 および version 10 でサポートされた OLAP 機能の概要を解説します。

これらの拡張機能をまとめると以下のとおりです。

- **Group By** 句の拡張 **:** ANSI 標準 SQL 1999 [2] に最初に盛り込まれたこれらの拡張により、以下のことが可能な複雑な SQL 文を書くことができます。

  1. 様々な方法で入力行をパーティション化し、異なるグループ全てを一緒に連結する結果セットを出します。
  2. 「データキューブ」を作成し、スパース、データマイニング分析に多次元の結果セットを提供します。
  3. オリジナルのグループを含む結果セットを作成します。必要に応じて小計 (sub-total) や 合計 (grand-total) 行も含めることができます。

- Ranking 関数：これらの新しい関数により、アプリケーション開発者は、例えば「今年出荷された製品の上位 10 製品の名前を、売り上げごとに挙げる」や「少なくとも異なる企業 15 社から注文を得た営業のトップ 5 %を挙げる」などの問いに対して 1 つの SQL 文のクエリーで回答を得ることができます。このような新しいランキング関数には、**RANK()**、**DENSE_RANK()**、**PERCENT_RANK()**、**CUME_DIST()** などがあります。

- Window aggregate 関数：これらの関数により、例えば「ダウ平均株価の四半期ごとの移動平均は？」「全従業員をリストアップし、それぞれの部署ごとの累積給料を出す」などのクエリーに対して移動平均や累積測定による解答が可能になります。**COUNT()**、**SUM()**、**MIN()**、**MAX()**、**AVG()**、**STDDEV()**、**VARIANCE()** などの既存の aggregate 関数や、**COVAR_SAMP()** などの新しい分析関数を、Window 関数として、処理が進むにつれ、結果セットに対して概念的に「ムービングウィンドウ」を作成する SQL クエリー仕様 **Window** の中で新しい句とともに使用することが可能です。

- レポーティング aggregate 関数 ： 同じ行のステートメント内の異なる「レベル」で計算された、aggregate 関数の結果を配置するために基本的な aggregate 関数である **COUNT()**、**SUM()**、**MIN()**、**MAX()**、**AVG()** を含めた aggregate 関数を使用することができます。

これにより、join または相関サブクエリーの必要性を避けつつ、グループ内の詳細行と aggregate 値を比較する方法を提供します。

これらの新しい OLAP 関数には、主に 2 つのメリットがあります。1 つは、容易な方法で多次元分析、データマイニング、時系列分析、傾向分析、コストアロケーション、ゴールシーク、例外アラートなどができるようになります。単一の SQL 文でそれが可能なことも多々あります。2 つ目は、window や レポーティング aggregate 関数は、新しいリレーショナルオペレーターである window を利用し、self-join や相関サブクエリーを利用するセマンティクス的に同等のクエリーよりもずっと効率的に実行することができる点です。

2 Group by 句の拡張

ANSI 標準 SQL (文法 1 ルール 2 参照) からの句 **Select - From - Where - Group by – Having** が絡む通常のクエリー仕様のセマンティクスを考えてみます。

1. **FROM** 句内にあるテーブル式の Cartesian 幾何学製品を計算します。
2. **Where** 句の述部をこの結果に適用します。 **Where** 句を満たせないロー --- つまり、**Where** 句 が true と評価しないもの --- は即座にリジェクトされます。
3. aggregate 関数を除いて、各ローに対して **Group by** 句内の式のリストと **Select** リストにある式を評価します。
4. **NULL** をシンプルにそれぞれのドメインの特別な値として扱い、**Group by** 句内の式の distinct 値に基づきこれらをグルーピングすることで結果のローをパーティション化します。**Group by** 句内の式はそれぞれのパーティションにユニークなキーを形成します。
5. それぞれのパーティションで、**Select** リストまたは **Having** 句に現れる aggregate 関数を評価します。それぞれのパーティションから詳細ローを取り除き、**Group by** 式とそれぞれのパーティションに対して計算される aggregate 関数の値で構成される結果セットを形成します。.
6. **Having** 句を満たさないこれらのロー （ 現在はパーティション ） をこの結果セットから除外します。

---

1 テキスト内の文法ルールの参照は、バッカス・ナウア記法により Appendix A にリストされている SAP SQL Anywhere の SQL 文法の関連サブセットです。

最初に説明する OLAP 拡張は、同じクエリー内で、複数の方法で中間結果をパーティション化し、全てのパーティション結果を一つの結果に連結できることです。わかりやすい方法としては、クエリーの **Group by**句内の < GROUPING SETS SPECIFICATION > (文法ルール 54 参照) を使うことです。

2.1 Group by Grouping Sets

例 1 (2 つ以上のグルーピングクライテリアを指定)

SAP SQL Anywhere の **asademo** データベース[2] のスキーマへの以下のクエリーについて考えてみます。

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

結果 *R* を返します。

R

	City	*State*	*Company Name*	*Cnt*
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

上記のクエリー仕様はクエリー式に対してセマンティクス的に同等です。

---

2 このテクニカルホワイトペーパーに含まれるクエリーの例は全て、SAP SQL Anywhere 9.0.2 に含まれる **asademo** サンプルデータベースのスキーマに対してのものです。

```
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
```

この例について言及する点がいくつかあります。

最初に、特定の grouping セットに対して使われていない **Group by** 式値のプレースフォルダーとして **Null** 値が使用されていることに注意してください。この例では、ロー 1～7 と 9 は Company_ name でグルーピングされて生成されたローです。ロー 10～12 は、City と State のコンビネーションでグルーピングされて生成されたローです。ロー 8 は、「empty」grouping セットで表された「 grand total 」であり、マッチした挿入語句「()」のペアを使用して指定されています (下の 例 2 参照)。「 missing 」< GROUP BY EXPRESSION > にプレースホルダーを使用した結果、入力内に存在するあらゆる **Null** からの特定の < GROUP BY TERM > から生成されたローを容易に混乱させてしまうことになります。これらの値は、以降のセクション 2.1.1 で説明する **GROUPING()** 関数を使用することで区別することが可能です。

次に、空の grouping セットは、**Group by** への入力内の全てのローの 1 つのパーティションを表します。

例 2 (空の Grouping Sets)

以下のクエリーで空の grouping sets を指定します :

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

セマンティクス的には以下と同等です[3]:

3　パーティション化されたテーブル式の結果が空であっても、aggregate 関数を含みかつ **Group by** 句を含まない < QUERY SPECIFICATION > は、常に単一のロー結果をイールドすることを思い出してください。これらのセマンティクスは、空のグルーピングセットが発行された場合には、< QUERY SPECIFICATION > が **Group by** 句と他の < GROUP BY EXPRESSION > を含まない場合とは反対にホールドします。後者のケース

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

3 つ目に、< GROUPING SETS SPECIFICATION > 内の重複する < GROUP BY TERM > を指定することができます (下の例 3 参照)。この場合、
< QUERY SPECIFICATION > の結果は、同一のロー[4] を含みます。

例 3 (重複 Grouping Sets)

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

結果 *R* を返します。

R

	City	*Cnt*
1	'Pembroke'	4
2	'Drayton'	3
3	'Petersburg'	1
4	'Pembroke'	4
5	'Drayton'	3
6	'Petersburg'	1

4 番目に、SAP SQL Anywhere の SQL 方言は、サブクエリーを含めてあらゆる複雑な式をかっこに入れ、< COLUMN REFERENCE > が使用できるあらゆるコンテキストにバーチャルに使用されることを許容する曖昧さを持つため、**Grouping Sets** を使用した場合には、かっこを使用して明示的にするのも良い方法です。例えば、構文 **Grouping Sets (X,Y)** は、2 つの個別のターム **(X)** と **(Y)** を除いて、単一の group-by term をイールドしません。そのため、通常の **Grouping Sets** を指定しない **Group by** 句は、単純に、単一の < SIMPLE GROUP BY TERM LIST > 内に *n* < GROUP BY EXPRESSION > から構成される grouping set と同等です(文法上は、単一の < GROUP BY TERM >)。

2.1.1 GROUPING() 関数を使用する

GROUPING() 関数では、1つ以上の < GROUP BY EXPRESSION > に Null 値が含まれるグループ化された < QUERY SPECIFICATION > の結果ローのかっこを外す必要があります。この関数は、単一のパラメーターをとります。つまり、

---

4 SAP SQL Anywhere 10 は、ANSI 標準 SQL 2003 の機能 T434 をサポートしていません。これは、set quantifiers All (デフォルト) または **Group by** 句への **Distinct** を認めています。Distinct を指定することで、< GROUPING SET SPECIFICATION > から重複 < GROUP BY TERM > を除外することができます。

1 つ以上のグルーピングセット[5]で指定される < GROUP BY EXPRESSION > です。この関数の範囲は、シンプルに integer 値 0, 1 です。**GROUPING(X)** 関数は、**X** が **Null** の場合、値 1 を返します。なぜならば、これはそのためにこのローがアウトプットされたグルーピングセットのメンバーではないからです。さもなくば、0 を返します。

例 4 (grouping() 関数)

```
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
```

結果 *R* をイールドします。

R

	Employee	*Year*	*Orders*	*GE*	*GY*
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

この例では、結果のタプル (1) は、空の grouping set () の仕様により order (54) の「grand total」を表しています。この結果のタプル (2) は、subtotal ロー (GY 値 0 により)、つまり、**Null** year のゼロ order の sumです。この例では、**Null** year が存在しています。なぜならば、そのクエリーを qualify する employee のうち 5 人 は、(7) 経由の結果タプル (3) で示されるように order がありません。タプル (9) と (11) は、year 2000 と 2001 の employee 667 の order 数を示します。最後に、GY 値 0 と GE 値 1 で示されている subtotal タプル (8) と (10) は、2 年間の total order を示します。

---

5 SAP SQL Anywhere 10 は、ANSI 標準 SQL 2003 のマルチ引数 **GROUPING()** 関数を定義するANSI 標準 SQL 2003 の T433 機能はサポートしていません

各結果タプルで **Grouping** 関数の値をテストすることにより、detail、subtotal、grand total ローを区別することができます。

2.2 Group by Rollup と Cube

Grouping Sets の使用は、データの複数の異なるパーティンショニングを単一の結果セットに concatenate したい時に便利です。しかしながら、これは個々の < GROUP BY TERM > を < GROUPING SETS SPECIFICATION > に指定するには少々面倒です。幸い、容易に一般的なグルーピングパターンを指定する 2 つの重要なシンタックス的ショートカットが存在します。これらのパターンの 1 つは、*rollup*、そしてもう *1* つは、*cube* と呼ばれています。

2.2.1 Group by Rollup

あらゆる SQL クエリー仕様の複数のグルーピング属性を指定する場合、subtotal ローの計算を要望するのが普通です。多くのアプリケーションで典型的なある特定のパターンが、IBM の Cobol Report Writer を連想させる、左から右へ順番どおりに subtotal を計算するものです。このパターンは *hierarchy* として言及されることもあります。なぜならば、追加の subtotal 計算の導入は、より詳細な粒度のローを追加で作成することになります。ANSI SQL 構文において、hierarchy の grouping 属性を指定する方法は < ROLLUP TERM > (文法ルール 48) を指定する **Rollup** キーワードの使用を通じて行います。

例 5 (Group by Rollup)
以下のクエリーは、年間および四半期の注文を summarize します。

```
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
```

結果 *R* をイールドします。

R

	Quarter	*Year*	*Orders*	*GQ*	*GY*
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	2000	10	0	0

このクエリーは、各 quarter (ロー 3～6 と 8～10) ごとの total order、year (ロー 2 と 7) ごとの subtotal 、order の「grand total」ロー 648 (ロー 1) を返します。**GROUPING()** 関数よって、どのように値が返されるかは、grand total を含むローから subtotal ローを区別するために使用することが可能であることに注意してください。

Rollup を指定するもう 1 つの方法として、Microsoft SQL Server 互換の構文 **With Rollup** (文法ルール 53) を使用する方法があります。**With Rollup** を使用した場合には、シンプルな式のみが指定可能です。つまり、**Group by** 句 は < SINGLE GROUP BY TERM LIST > のみを含む可能性があることに注意してください。さもなくば、構文エラーでこのリクエストは失敗します。

例 6 (Group by With Rollup)

以下の Microsoft SQL Server と互換性のあるクエリーでは、上の例 5 と同じ結果がでます。

```
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 Rollup と Grouping Sets の同値性

Rollup キーワードの使用を通して指定された <ROLLUP TERM> (文法ルール 48) は、本質的に grouping sets の hierarchical series を作成する generating 関数です。言い換えれば、もし <ROLLUP TERM> が (X_1, X_2,…, X_n) 形式の n <GROUP BY EXPRESSION> を含む場合、<ROLLUP TERM> は以下のような $n+1$ grouping sets を生成します。

{(), (X_1), (X_1, X_2), (X_1, X_2, X_3),..., (X_1, X_2, X_3,..., X_n)}.

例 7 (Grouping Sets としての Rollup)
以下のクエリーはセマンティクス的には上の例 5 のクエリーと同等です。

```
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
```

しかしながら、<ROLLUP TERM> は <GROUPING SETS SPECIFICATION> 内のどこにでも現れる可能性があることに注意してください。そして、1 つの <GROUP BY CLAUSE> のどれにも複数の <ROLLUP TERM> が存在する可能性があります。<ROLLUP TERM> が他の<GROUP BY TERM> と組み合わさる場合、結果の grouping sets は、様々な terms によって指定または生成された全ての grouping sets の cross-product を計算し、重複を除外することで決定されます。

例 8 (Group by Rollup terms のコンビネーション)
以下の <GROUP BY CLAUSE>

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

は、以下と同等です。

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

どの grouping sets が暗黙的に生成されるかは、**Group by** 句内の <GROUP BY TERM> のシンタックス的な order には依存しません。上の例の場合、<GROUP BY>「C」が <ROLLUP TERM> で生成されたそれぞれの grouping sets にどのように concatenate されるのか注意してください。しかしながら、<GROUP BY EXPRESSION> のそれぞれの grouping set は、パーティション化された結果のキーを形成する属性のセットを特定します。そのため、様々な <GROUP BY TERM> のシンタックス的な order は、immaterial (非実体)です --- **Order by** 句のみが、結果の特定の order を保証することができます。これは、この例で示されるような複数の <ROLLUP TERM>の場合でも同じです。

例 9 (複数の Rollup Terms)
以下の **Group by** 句は

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

以下と同等です。

```
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

ANSI SQL OLAP 拡張でサポートされている典型的なグルーピングパターンの 2 つ目は、「キューブ」または「データキューブ」です。データキューブ の基本的な考え方は、可能なあらゆる <GROUP BY TERM> のコンビネーションを使用した入力の *n*-dimensional summarization を作成することです。**Rollup** のように、**Cube** は、可能なあらゆる grouping sets のコンビネーションを作成する generating 関数 --- 複雑なデータ分析に非常に便利になりうる入力の全次元のクロス集計です。

<CUBE TERM> 内に $(X_1, X_2, \ldots, X_n)$ 形式の *n* <GROUPING EXPRESSION> が存在する場合、**Cube** は、以下のような 2^n grouping sets を生成します。

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

例 10 (Group by Cube)

以下のクエリーは、sales order を year ごと、quarter ごと、year 内の quarter に summarize します。

```
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
```

結果 *R* をイールドします。

R

	Quarter	*Year*	*Orders*	*GQ*	*GY*
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	2000	10	0	0

Function	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)	σ_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)	σ_x	$\sqrt{\frac{\sum (x_i - \bar{x})^2}{n}}$
STDDEV(X)		identical to STDDEV_SAMP(X)

テーブル 1: シンプルな aggregate関数

上の例 5 内の Rollupクエリーで作られた結果ローに加え、ロー 2～5 はあらゆる year 内の calendar year ごとの sales orders を summarize します。クエリー仕様の <GROUP BY CLAUSE> に含まれる他の任意の <GROUP BY TERM> は、**Rollup** のそれと同じ方法で <CUBE TERM> と組み合わされています。SAP SQL Anywhere は、**Rollup** と同様、Microsoft SQL Server と互換の構文 **WithCube** をサポートしています。

Cube は指数関数的な数の grouping set を生成するため、**Cube** の使用を通して生成された結果セットは、非常に大きいものになりうることに注意してください。この理由のため、SAP SQL Anywhere では、1 つの <GROUP BY CLAUSE> 内に <GROUP BY CLAUSE> 64 よりも多くの <GROUP BY EXPRESSION> を含ませることを許容していません。ステートメントがこの制限を超えた場合には、SQLCODE -944 (SQLSTATE 42WA1) で失敗します。

3 統計 aggregate 関数

ANSI SQL OLAP 拡張は、数値データの統計分析を許容する複数の追加 aggregate 関数 (表 1 と 2 参照) を提供しています。これには、分散、標準偏差、相関、 線形回帰などを計算する関数のサポートが含まれます。

3.1 標準偏差と分散

最も一般的な正規分布の分布の測定は、平均値 $\bar{x}$ からの 値 x_i の分布を図る標準偏差です。偏差は、シンプルに正であったり負であったりするこの差である $x_i \bar{x}$ です。この差を標準化するため二乗され全ての差を正の値にします。二乗すると、平均値からより大きな偏差は、偏差の合計により重要なインパクトがあります。平均二乗偏差は、分散と呼ばれ、以下のように定義されています。

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

また、標準偏差は、単にこの値の平方根です。より便利で、効率的な分散のための「計算式」は、分布の平均値を事前計算せずに少量の代数で導くことができます。

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

ANSI SQL では、分散と標準偏差関数の両方のものが存在します。SAP SQL Anywhere におけるこの関数のサンプリング版は、この式を上記の方程式 2 で表されているようにこの式を計算し、**VAR_SAMP()** と **STDDEV_SAMP()** 関数と一致します。これらは、それぞれ **VARIANCE()** と **STDDEV()** としてエイリアスされます。 なぜならば、典型的な分散と標準偏差の利用は、サンプルで処理し、母集団ではないからです。これらの関数の母集団版は、「平均」偏差平方の計算に分母として (*n* – 1) よりも *n* を使用します。

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

母集団分散と標準偏差は、それぞれ **VAR_POP()** と **STDDEV_POP()**関数で計算されます。

6 つの関数は全てクエリーの <GROUP BY CLAUSE> によって決定されるローのパーティションの値を計算できるという点で true の aggregate 関数です。**MAX()** または **MIN()** のような他の基本的な aggregate 関数のように、これらの計算では入力内の **Null** 値を無視します。また、分析されている式のドメインに関わらず、全ての分散と標準偏差の計算は、IEEE 倍精度浮動小数点を使用して行われます。どの分散または標準偏差関数への入力も空集合の場合、それぞれの関数はその結果として **Null** を返します。**VAR_SAMP()** を単一のローのために計算した場合、これは **Null** を返し、**VAR_POP()** は値 0 を返します。

3.2 相関と線形回帰

線形回帰は、変数 *y* (従属変数と呼ばれている) の「説明」変数 *x* (独立変数と呼ばれている) への従属 の summarize に使用されているテクニックです。線形回帰モデルは、グラフィカルに散布図内の (*x, y*) のペアの値間の関係を線形方程式が以下である一本の線で近似させるために使用されています。

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

線形回帰は、様々な点を傾き b と切片 a で $(\bar{x}, \bar{y})$ を通る線にフィットさせるために最小二乗法を使用します。ラインは、それぞれの相応 x-値から期待された値 $\hat{y}$ をモデル化します。観測された (x, y) ペアからの期待値と実際の *y*- 値間の差は、残差と呼ばれています。*x* または *y* の値のどちらかが **Null** の場合、そのペアは無視されます。標準偏差の計算のように、残差によっては正の値であったり負の値であったりします。線形回帰最小二乗法は、残差の平方の合計ができる限り小さくなるような線 $y = a + bx$ を導きます。 *n* の non-**Null** の観測値が与えらえた場合、a と b の値を計算する公式は以下のとおりです。

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

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

繰り返しますが、方程式 5 を解くより便利な計算値は、サンプルの平均値 $\bar{x}$ と $\bar{y}$ を事前計算しなくとも、少量の代数で求めることが可能です。

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

線形近似で予測された値 $\hat{y}$ は、高い精度を得ることが可能です。回帰は、観測されている (*x,y*) ペアの値内の異常値によって大きく影響されます。そして、もちろん、*x* と *y* の関係性は全てが線形であるわけではなく、二次または指数関数的かもしれません。それにも関わらず、線形回帰テストは、データ分析において非常に便利なツールです。

SAP SQL Anywhere でサポートされている OLAP 線形回帰関数は、表 2 にまとめられています。特に aggregate 関数 **REGR_SLOPE()** は、独立変数と従属変数間の関係をモデル化する線形回帰方程式の傾き (*b*) を返します。一方、aggregate 関数 **REGR_INTERCEPT()** は、その方程式の切片 (*a*) を返します。

3.2.1 GOF(Goodness-of-fit） 統計

線形関係が *x* と *y* 間に存在するかを測定する方法として、*r* (サンプリング

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

表 2: 統計 aggregate 関数

) と ρ (母集団) で示されるデータの相関係数を計算することができます。aggregate 関数 **CORR()** で返される *r* の範囲は -1 と 1 の間です。絶対値が高ければ高いほど、関連は強くなります。正の場合は、正の強い関連、負の場合は負の強い関連です。n 観測値のサンプリング相関係数の計算式は、以下のとおりです。

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

ここでは $\bar{x}$ と s_x が n 観測値ペア内の *x*- 値のサンプル平均値と標準偏差であり、相応するように $\bar{y}$ と s_y が*y*-値のサンプル平均値と標準偏差です。別の言い方をすると、相関係数は、*x* と *y* の共分散、s_{xy} で表された、標準偏差の積で割ったものとイコールです。分散のように、共分散の定義は 2 つあります。*x* と *y* のサンプリング共分散は以下、

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

そして (*x, y*) の母集団共分散は以下のとおりです。

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

これら 2 つの値は、SQL 関数 **COVAR_SAMP()** と **COVAR_POP()** でそれぞれ返されます。*r* の計算式は、以下のとおりです。

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

この方程式から、*r* = 1 と *r* = 1 の極値は完全線形アソシエーションの場合のみ発生すると表すことができます。そして、1 に近い値または 1 は点が直線に近いところにあることを示しています。相関係数の平方 r^2 は、*y* on *x* の最小二乗回帰によって説明される y 値内の変動の分散として定義されます。例えば、r^2 = 0.988 の場合、線形回帰は、*x* に関して、y-値内の観察された変動の[6] 98.8 % を説明すると述べることができます。r^2 値は、GOF (Goodness-of-fit) 統計と呼ばれることもありますが、aggregate 関数 **REGR_R2()** によって返されます。

相関と回帰のコンセプトは、密接に絡み合っています。実際、*r* は、線形回帰線の傾き *b* を計算する代替方法として使用することができます。

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

2 つの公式は同等です。なぜならば、共分散のサンプリング公式からの (*n* 1) terms と標準偏差は、これらの関数の母集団変量がそうであるように、お互いに相殺し合うからです。

3.2.2 回帰と相関関数

SAP SQL Anywhere 10 は、線形回帰の品質分析をアシストするために使用することができる goodness-of-fit 関数 **REGR_R2()** に加え、様々な統計関数をサポートしています。これらの関数によって実装されている統計公式は、表 2 にまとめられています。それぞれの関数の最初の引数は、従属式 (**Y** で指定されている)、そして 2 つめの引数は独立式 (**X** で指定されている) です。短く言うと、これらの関数とは

- **REGR_COUNT(Y,X):** この関数は、入力内の (*x, y*) 値の non-**Null** のペアの数を返します。与えられたペアの x も y もどちらも non-**Null** の場合のみ、あらゆる線形回帰計算にその観測値が使用されます。
- **REGR_AVGX(Y,X)**: この関数は、(*x, y*) 値の全ての non-**Null** ペアからの *x*-値の平均値を返します。

6 この文書の対象範囲を超える統計に関する包括的な概要ですが、強い相関が必ずしも因果関係を暗示するものではないであることを読者に警告したいと思います。原因分析には、関連する値のペアの単純な分析よりさらなる文脈が必要となります。

- **REGR_AVGY(Y,X)**: この関数は、(*x, y*) 値のペアの全ての non-Null からの *y*-値の平均値を返します。

これらの関数に加えて、SAP SQL Anywhere 10 では、ANSI 標準 SQL 2003 による3つの線形回帰関数をサポートしています。

- **REGR_SXX(Y,X)**: この関数は、下の式を使用して (*x, y*) ペアの *x*-値の平方の sum を返します。

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

この方程式は、サンプルまたは母集団分散公式 (方程式 2 と 3 参照) の分子と同じです。他の線形回帰関数のように、**REGR_SXX()** は、入力の *x* または *y* が **Null** の (*x, y*) 値のペアを無視することに注意してください。

- **REGR_SYY(Y,X)**: この関数は、下の式を使用して、(*x, y*) ペアの *y*-値の平方の sum を返します。

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

そして、また、これはy 値の分散方程式の分子と同じです。

- **REGR SXY(Y,X)**: この関数は、(*x, y*) ペアのセットへの積公式の 2 つの sum の差を返します。

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

これらの 3 つの関数は、線形回帰の特徴を分析するには便利なこともあります。なぜならば、これらは他の回帰関数の計算にも使用される部分的な公式を表しているからです。特に、**REGR_SLOPE()** 関数で返される線形回帰方程式 (上記の方程式 5 参照) の傾き *b* は、下と同じです。

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

そして、goodness-of-fit 測定 r^2 は、下と同じです。

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

これらの追加関数を使用すると、従属変数 (x) と独立変数 (y) の式のペアの線形相関に付随する幅広い測定を容易に計算することができます。これらの測定 [1, 5, 6] のいくつかは、表 3 にまとめられています。

Standard Error (e) of y-values
SQRT((REGR_SYY()-(POWER(REGR_SXY(),2.0)/REGR_SXX()))/(REGR_COUNT()-2))
Standard Error of Slope (b)
e / SQRT(REGR_SXX())
Standard Error of Intercept (a)
e * SQRT((1/REGR_COUNT()) + (POWER(REGR_AVGX(),2) / REGR_SXX()))
Regression sum of squares
POWER(REGR_SXY(),2) / REGR_SXX()
Residual sum of squares
REGR_SYY() - (POWER(REGR_SXY(),2)/REGR_SXX())
t Statistic for Slope
REGR_SLOPE() * SQRT(REGR_SXX()) / *e*

表 3: 追加回帰測定

4 Window 関数

OLAP のための ANSI SQL 拡張の主な機能は、window と呼ばれる新たな関係代数 construct です。

SQL/OLAP windowing 拡張では、クエリーの結果セットをパーティションと呼ばれるローのグループに分割することが可能です。パーティションは、論理的には、クエリー仕様の結果を計算するセマンティクスの一部として、最後の **Select** リストとそのクエリーの **Order by** 句の評価の前に、そして、**Group by** 句によって定義されたグループが作成された後に、作られます。そのため、SQL 文内の句の評価の順は、以下のようになります。

From → Where → Group by → Having → Window → Distinct → Order by

window パーティションは **Group by** 演算子に続くため、**SUM()**、**AVG()**、**VARIANCE()** のようなあらゆる aggregate 関数の結果を、パーティションのための計算で利用することができます。そのため window はクエリーの **Group by** と **Order by** 句に加えて grouping と ordering オペレーションを実行する別の機会を提供します。しかしながら、window 関数の結果が絡むどの計算でも --- 例えば、これを述部に使用する --- より複雑な文の construction が必要になります。通常、この construction には window 関数の結果を計算する派生テーブルと、その派生テーブルの外部の **SELECT** ブロック(望ましい **Where** 句述部が含まれるもの）が必要になります。

window のパーティションは、入力全体、単一のロー、またはその中間で構成することが可能です。「window」construct のメリットは、パーティション内のローは、*window* 関数によって提供される追加の表現力をサポートするためにソートすることが可能という点です。window 関数では、aggregate 関数 over aggregate 関数を計算するクエリーを construct することがが可能です。例えば、ある特定の数量の最大 **SUM()** などwindows では、3 つの window 関数 (文法ルール 6 参照) のクラスを使用できます。

- *Ranking* 関数 (**RANK()** 、**CUME_DIST()** など) : パーティション内の他のローに関連するローのランクを返します。
- *Row numbering* 関数(**ROW NUMBER()**) : パーティション内のローに固有の番号をふります。
- *window aggregate* 関数 (**SUM()** 、**COUNT()** など) : SAP SQL Anywhere 10 でサポートされている全ての aggregate 関数のサブセットですが、window とともに使用することも可能です。

window のサイズ変化

window は、<WINDOW SPECIFICATION> (文法ルール 23) を使用して定義します。window のコンセプトは柔軟です。window パーティション内のそれぞれのローに対して、「sliding」window を定義することができます。これは、パーティションの「current」ロー上のあらゆる計算を実行するために使用されているローの特定の範囲を変更する可能性があります。この「current ロー」は、window の開始点と終了点を決定するための参照点を提供します。Window サイズは、以下のどちらかをベースにすることができます。

- 物理ロー数 ---- **ROWS** の <WINDOW FRAME UNIT> (文法ルール 35) を定義する <WINDOW SPECIFICATION> を使用
- *numeric* 値の論理的インターバル ---- **RANGE** の <WINDOW FRAME UNIT> を定義する <WINDOW SPECIFICATION>

window の開始点も終了点も動的に変更できます。例えば、累計を計算するには、開始点が各パーティションの最初のローで固定され、終了点は current ローが含まれるようにパーティションのローに沿って「slide」する window が関与します (下の図 1 参照) 。他の例を挙げると、window の開始点も終了点も可変する可能性がありますが、パーティション全体に対して一定のロー数を定義します。この construction は、「moving」平均を計算するクエリーを composeすることを許します。例えば、sliding window を活用すると、移動する 3-day 平均株価を返す SQL クエリーを書くのは、比較的容易です。

例 11 (クエリーと window 関数)

2001 年 7 月と 8 月に出荷された全ての製品と出荷日ごとの累積出荷数をリストする以下のクエリーを考えてみます。

```
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
```

このクエリーは、結果 *R* を返します。

R

	ID	*Description*	*Quantity*	*Ship Date*	*Cumulative Qty*
1	301	V-neck	24	2001-07-16	24
2	302	Crew Neck	60	2001-07-02	60
3	302	Crew Neck	36	2001-07-13	96
4	400	Cotton Cap	48	2001-07-05	48
5	400	Cotton Cap	24	2001-07-19	72
6	401	Wool cap	48	2001-07-09	48
7	500	Cloth Visor	12	2001-07-22	12
8	501	Plastic Visor	60	2001-07-07	60
9	501	Plastic Visor	12	2001-07-12	72
10	501	Plastic Visor	12	2001-07-22	84
11	601	Zipped Sweatshirt	60	2001-07-19	60
12	700	Cotton Shorts	24	2001-07-26	24

例 11 では、**SUM()** window 関数 の計算は、2 つのテーブルの join とクエリーの **Where** 句のアプリケーションの後に発生します。このクエリーは、join からの入力ローは以下のように処理されるということを指定するインライン <WINDOW SPECIFICATION> を使用します。

1. **Prod_id** 属性の値に基づき入力ローをパーティション (group) する
2. パーティション内で、**ship_date** 属性によってローをソートする
3. 各パーティションの最初の (ソートされた) ローで構成される sliding window を使用してパーティション内のそれぞれのローで、**quantity** 属性に対して**SUM()** 関数を current ローまで (current ローを含む ） 評価する (図 1 参照)

このクエリーの代替 construction は、それを利用する関数とは別の window を指定することです。これは、同じ window に基づく 2 つ以上の window が指定されている場合に便利です。例 11 内のクエリーの場合、（「cumulative」によって識別された window を宣言しながら）**Window** 句を使用する construction は、以下のとおりです。

```
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
```

<QUERY SPECIFICATION> 内の **Order by** 句の前にどのように表れるか注意してください。

Window 句を使用する場合、以下の制限が適用されます。

named および インラインされた window のミックス specifications

- <WINDOW SPECIFICATION> のインラインされた部分は、**Partition by** 句を持つことができません。
- Window句内で指定された **window** は <WINDOW FRAME CLAUSE> を含めることができません (文法ルール 34 参照)。
- インラインされた <WINDOW SPECIFICATION> または **Window**句内で指定された <window specification> のどちらかに <WINDOW ORDER CLAUSE> (文法ルール 33) を含むことができますが、両方を含むことはできません。

例 12 (window と multiple 関数を利用する)

以下の例で示すように単一の (named) window を定義し、それに対して複数の関数結果を計算することが可能です。

```
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 window の定義

<WINDOW SPECIFICATION> は、以下を定義することが可能です。

- <WINDOW PARTITION CLAUSE> (文法ルール 30) の使用を通じた 入力 ロー のパーティション --- <WINDOW PARTITION CLAUSE> が指定されていない場合、入力 は単一のパーティションとして扱われます。
- <WINDOW ORDER CLAUSE> (文法ルール 33) の使用を通した各パーティション内のローの順序 --- <WINDOW ORDER CLAUSE> が指定されていない場合、入力 ロー は任意の順番で処理されます。**Range** の <WINDOW FRAME UNIT> の使用には <WINDOW ORDER CLAUSE> が存在している必要があることに注意してください。

さもなくば、結果は SQLSTATE '42WA9' になります。**Range** の場合、<WINDOW ORDER CLAUSE> は、単一の式のみで構成されている可能性があります。また、ranking 関数には、<WINDOW ORDER CLAUSE> が必要なことに注意してください。なぜならば、ソートされた 入力 に対してのみに定義されているからです。<QUERY SPECIFICATION> 内の **Order by** 句 のように、デフォルトのソートのシーケンスは、昇順 **(Asc)** です。

- パーティション内の全てのローの処理に使用されている window の特徴。これらの特徴は、<WINDOW FRAME CLAUSE> の使用を通じて指定されています。<WINDOW FRAME CLAUSE> は、current ローと関連して、window 関数の計算に使用されている「window」 の開始と終了を定義します。<WINDOW FRAME CLAUSE>が指定されていない場合、デフォルトの window フレームは、<WINDOW ORDER CLAUSE>が指定されているかどうかに依存します。

  デフォルト window サイズ

  – window 仕様に <WINDOW ORDER CLAUSE> が含まれる場合、window の 開始点は **Unbounded Preceding** で、終了点は **Current Row** ー です --- したがって累積値の計算に適した varying-size window を定義します。

  – window 仕様が <WINDOW ORDER CLAUSE> を含まない場合、window の開始点は **Unbounded Preceding**で、終了点は、**Unbounded Following** です --- したがって、current ローに関わらず、固定サイズの window を定義します。

  <WINDOW FRAME CLAUSE> は、ranking 関数 または **ROW NUMBER()** 関数とは使用できないことに注意してください。

例 13 (unbounded window の使用)
以下のクエリーは、全ての製品の合計数に付随する全ての製品で構成される結果セットを作成します。

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

上で述べたように、指定できる <WINDOW FRAME UNIT> は 2 つあります。

- Window 内の current ローに関連するローの物理数をベースに window のサイズを指定することが可能です。よくある例は、
  - ***Rows Between Unbounded Preceding and Current Row***
    この window フレームは、開始点がそれぞれのパーティションの始めであり、window の終了点が current ローである window を指定します。この <WINDOW FRAME CLAUSE> は、しばしば累積 sum などの累積結果を計算する window を construct するために使用されます。

ローをベースにした通常の window フレーム

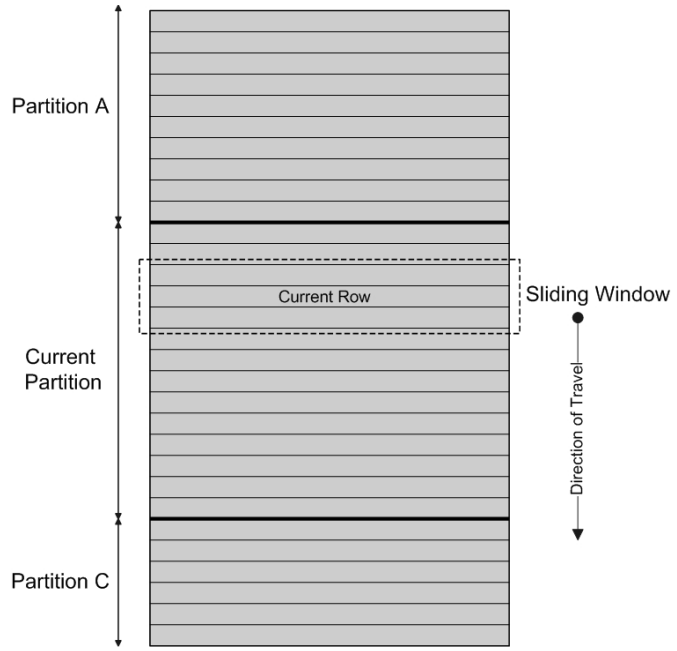

図 1 : 3 行のローの移動 window とパーティションされた入力の図解

Rows Between Unbounded Preceding and Unbounded Following

この window フレームは、パーティション全体に対して (current ローに関わらず) 固定の window を指定します。そのため window aggregate 関数の値は、パーティションのそれぞれのローと同じになります。

- ***Rows Between 1 Preceding and 1 Following***

この window フレームでは、サイズが固定された「moving」window を隣接した 3 行のロー (current ローの前後各 1 行) に対して指定します。

通常、例えば 3 日間や 3 か月間の移動平均を計算する場合などには <WINDOW FRAME CLAUSE> を使用します。しかしながら、window 関数への入力における「ギャップ」による問題を避けるよう注意する必要があります。

値のセットが継続するものでない場合、**Rows** よりも **Range** を使用する方が良いかもしれません。なぜならば、window の **Range** を元にした bounds は、自動的に隣接するローと重複値を (handle row with duplicate value) 扱い、範囲内に「ギャップ」が存在する場合、他のローを含まないからです。

Moving windows と boundary ロー

moving window の場合、入力内で **Null** 値を含むローが最初のローより前と最後のローの後に存在すると仮定されることに注意してください。

これが意味するのは、3 行のローの moving window では、入力の最後のローの計算 --- 「current」ロー --- は、即時に preceding するローと **Null** 値を含むということです。

- ***Rows Between Current Row and Current Row.***
  この<window frame clause> を指定することで window を current ローのみに限定することができます。
- ***Rows Between 1 Preceding and 1 Preceding***.
  この window フレームは、（ current ローに関して ） preceding ローのみで構成される単一のローを指定します。Current ローのみに基づく値を計算する他の window 関数とのコンビネーションで、この construction は隣接するロー間の「deltas」や値の違いを簡単に計算できるようにします。(下の例 14 参照)

「Range」ベースの window サイズ

- 入力内の特定の属性の値に基づく window の定義も可能です。
  これは、**Range** の <WINDOW FRAME UNIT> を使用して指定します。
  Range を使用すると、window の <WINDOW ORDER SPECIFICATION> 内で指定されている属性は、<WINDOW ORDER SPECIFICATION> 数値のドメインを持たなければなりません。なぜならば、サーバーは <WINDOW FRAME PRECEDING> または <WINDOW FRAME FOLLOWING> からの <UNSIGNED VALUE SPECIFICATION> をその属性の値へ単純に追加するからです。
  windowサイズは、current ローに関連する preceding と following ローの属性の値に基づいています。**Range** の使用には、window 内の <WINDOW ORDER SPECIFICATION> が単一のカラムのみで構成されている必要があります。

例 14 (近接ロー間のデルタの計算)
上で述べているように、2つの windows を使用するのは --- current ローに 1 window、もう 1 つを previous ローに --- は、隣接するロー間の「deltas」または変更を計算するための straight forward な方法を提供します。以下のクエリーを考えてみます。

```
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')
```

このクエリーは、以下の結果 *R* を返します。

R

	Employee	*Last name*	*Curr*	*Prev*	*Delta*
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

window 関数 **SUM()** が使用されても、window の指定方法が理由で、sum は current または previous のどちらかの salary 値のみを含むということに注意してください。また、predecessor がないため、さらに「delta」も **Null** のため、結果の最初のローの「prev」値は **Null** だということに注意してください。上の例のそれぞれで、**Over** 句 とともに使用されている関数は、**SUM()** aggregate 関数です。実際、window とともに使用できる 3 つのクラスの関数があります。ranking 関数、ロー numbering 関数、window aggregate 関数です。これらの関数について、ranking 関数から順に説明します。

4.2 Ranking 関数

SAP SQL Anywhere 10 でサポートされている ranking 関数 (文法ルール 7 参照) は、**RANK()**、**DENSE_RANK()**、**PERCENT_RANK()**、そして **CUME_DIST()** です。Ranking 関数は、複数の入力ローからの結果を同じような方法、つまり **SUM()** aggregate 関数で計算しないため、aggregate 関数とは考えられていません。むしろ、これらの関数それぞれがパーティションのローの *rank*、または関連 ordering を特定の式の値に基づいて計算します。パーティション内のそれぞれのローのセットは、独立してランク付けされます。**Over** 句 が、**Partition by** 句を含んでいない場合、入力全体が単一のパーティションとして扱われます。その結果、ranking 関数によって利用された window のための <WINDOW FRAME CLAUSE> は指定しない可能性があります。それぞれが入力ローを異なるようにパーティションするまたはソートする複数の ranking 関数を含むクエリーを構成するのは、可能です。

全ての ranking 関数は、<WINDOW ORDER CLAUSE> が ranking 関数が依存する入力ローの sortedness を指定するために <WINDOW ORDER CLAUSE> を必要とします。**Order by** 句 が複数の式を含んでいる場合、2番目および subsequent な式を使用して、break ties します。最初の式が隣接するローと同じ値を持つ場合、SAP SQL Anywhere 10 における **Null** 値は、常に他のどの値 (in ascending sequence)[7] よりも前にソートされます。

7 SAP SQL Anywhere 10 では ANSI 標準 SQL 2003 で定義されている **Nulls First** と **Nulls Last** 句はサポートしていません。

ranking 関数の使用では、ランクされたローをアプリケーションが望むサブセットに制限する、または、アウトプットローを異なるシーケンスに再ソートすることのできる派生テーブルの使用を必要とすることがよくあります。このテクニックを以下の例で説明します。

例 15 (rank() 関数)

データベース内で最も高価な製品を決める以下のクエリーについて考えてみます。

```
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
```

これは結果 *R* を返します。

	Description	*Quantity*	*Unit Price*	*Rank*
1	Hooded Sweatshirt	39	24.00	1
2	Zipped Sweatshirt	32	24.00	1
3	Cotton Shorts	80	15.00	3

最も高価な製品が最も低いランク（ ランキングは 1 からスタート ）になるよう <window order clause> 内で降順にソートするシーケンスが指定されていることに、注意してください。また、ordered expression「quantity」の値が同じである隣接するロー (1,2) が同じランクなことにも注意してください。**RANK()** 関数では、タイの場合には、rank 値は「ジャンプ」します --- これが、**RANK()** 関数と、姉妹関数である **DENSE_RANK()** との違いです。

4.2.1 DENSE_RANK() 関数

DENSE_RANK() 関数

DENSE_RANK() 関数 は、「ギャプ」なし、「ジャンプ」なしで単調に増加する一連のランクを返します。

例 16 (DENSE_RANK() 関数)

以下のクエリーは、例 15と同じですが、代わりに**DENSE_RANK()** 関数を使用しています。

```
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
```

これは結果 *R* をイールドします。

R

	Description	Quantity	Unit Price	Rank
1	Hooded Sweatshirt	39	24.00	1
2	Zipped Sweatshirt	32	24.00	1
3	Cotton Shorts	80	15.00	2

クエリーの **Group by** 句の後に window が評価されるため、aggregate 関数の値を基にしたランキングを決定する複雑なリクエストを指定することができます。

例 17 (ランキングを aggregation とともに使用する)

以下のクエリーは、各 Region の総売上と各地区内の各自の Total Sales トップ 3 の Sales Rep を算出します。

```
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
```

このクエリーは、結果 *R* を返します。

R

	Sales Rep	Region	Total Sales	Rank
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

このクエリーは **GroupingSets** の使用をとおして複数のグルーピングを組み合わせます。window の <WINDOW PARTITION CLAUSE> は、**GROUPING()** 関数を使って特定の sales people と region 全体の total sales をリストする subtotal ローを表す詳細ローとを区別します。**sales_rep** 属性で **Null** 値を持つ region ごとの subtotal ロー のそれぞれは、ランキングの値 1 を持ちます。なぜならば、結果のランキングの order は、入力の各パーティションで再スタートされるからです。これにより、詳細ローが正確に1からスタートしてランクされるようにします。

最後に、この例では、**DENSE_RANK()** 関数が total sales の aggregation の入力に対してランク付けすることに注意してください。aliased **Select** リスト項目は、<WINDOW ORDER CLAUSE> 内で shorthand として使用されており、これは、SAP SQL Anywhere でのみサポートされているベンダー拡張 SQL です。

4.2.2 他の ranking 関数

SAP SQL Anywhere 10では、他にも2つのranking関数をサポートしています。1つは、累積分布関数 **CUME_DIST()** で、逆のパーセンタイルとして定義されることもあります。**CUME_DIST()**は、値の全体のセットに関連する特定の値の正規化された位置を計算します。関数の範囲は、0と1の間です。サイズ n の x-値の ordered set 内の特定の ith 値 x_i のため、**CUME_DIST()**関数は、以下のように定義されます。

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

つまり、**CUME_DIST()** によって返される値は、x_i, と同じ値の ordered set 内の最高の位置をセット内の値の数字 n で割ったものです。**CUME_DIST()**関数のセマンティクスは、他の ranking 関数のそれと似ています。**Null**値は、ドメイン内の他の値と同様に扱われます。

例 18 (CUME_DIST() 関数)

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

これは結果 *R* を返します。

R

	Description	*Quantity*	*Unit Price*	*CD*
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

中央値の計算

CUME_DIST() は、値のセットの中央値を決定するシンプルな方法を提供します。**CUME_DIST()** は、入力に奇数と偶数のどちらのローが含まれようと、タイの時にうまく中央値を計算するのに使用できます。本質的には、0.5 以上の **CUME_DIST()** 値で最初のローだけ決める必要があります。

例 19 (中央値の計算)
以下のクエリーは、単価の中央値の製品の製品情報を返します。

```
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
```

最後に、SAP SQL Anywhere は **CUME_DIST()** と非常によく似た ANSI SQL **PERCENT_RANK()** 関数もサポートしています。**PERCENT_RANK()** は、ordered セット内の値の位置を使用するよりも、分子内の **PERCENT_RANK()** の結果を使用します。— タイの場合は、**RANK()** が同じ値を与えることを思い出してください。そのため、**PERCENT_RANK()** は、以下として定義されます。

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

CUME_DIST() のように、**PERCENT_RANK()** の範囲は 0 と 1 の間です。1 の **RANK()** のローは、ゼロの **PERCENT_RANK()** になります。

例 20 (PERCENT_RANK() 関数)
例 18 のクエリーを考えてみます。今回は、**PERCENT_RANK()** を使用するようリライトしています。

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

これは結果 *R* を返します。

R

	Description	*Quantity*	*Unit Price*	*PctRank*
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 ロー

ROW NUMBER() v.s. NUMBER(*)

結果のローにユニークな番号をつける SQL 文が必要になることがよくあります。SAP SQL Anywhere では、様々な制限下で使用できる各ローにユニークな番号をつける **NUMBER**(*) ビルトイン関数という特別なベンダー拡張を以前のリリースで提供してきました。しかしながら、ANSI 標準 SQL では、**NUMBER(*)** ではなく **ROW NUMBER()** 関数を定義しています。さらに、これは **NUMBER(*)** の制限を受けません。

ranking 関数のように、**ROW NUMBER()** は <WINDOW ORDER CLAUSE> を必要とします。同様に、**ROW NUMBER()** は、ranking 関数のように window の **Order by** 句が over ユニークでない式にoverな場合でも非決定性の結果を返すことが可能です --- ローの順番は、ties の場合には予測不可能です。**ROW NUMBER()** は、全体のパーティションに対してのみ機能するようにデザインされています。そのため、<WINDOW FRAME CLAUSE> は、**ROW NUMBER()** 関数で特え定することができません。

例 21 (結果内の Numbering ロー)

ROW NUMBER() は ranking 関数が使用できるあらゆる状況で使用することが可能です。これによって、派生テーブル内で **ROW NUMBER()** を使用することができ、追加の制限は、join でさえも、**ROW NUMBER()** 値で可能です。

```
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 関数

上記のとおり、window aggregate 関数は、window 内で aggregation を実行するために使用することが可能です。SAP SQL Anywhere でサポートされている window aggregate 関数には、**COUNT()**、**SUM()**、**MIN()**、**MAX()**、**AVG()**、**FIRST VALUE()**、**LAST VALUE()**、6 つの分散と標準偏差関数 (文法ルール 8 参照) などがあります。特に、SAP SQL Anywhere のベンダー拡張である **LIST()** aggregate 関数は、<WINDOW SPECIFICATION> とともに使用することはできないことに注意してください。window を変化する開始点と終了点で定義することが可能なため、window aggregate 関数は、移動平均、移動最小または最大、累積和、様々な統計関数をサポートすることが可能です。さらに、複雑なデータ分析にはしばしば複数のレベルの aggregation が必要になります。アプリケーション開発者にとっては、window パーティショニングは、**Group by** 句に追加またはそのかわりとして、複雑な SQL クエリーの構成に対してかなりの柔軟性が得られることになります。

例 22 (window 関数の使用)

以下のクエリーは、1 年ごとの平均販売数よりも多い製品のリストを示す結果セットを返します。

```
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
```

例 23 (移動平均の計算)

この例では、**AVG()** は、window 関数として2000年の全 product の月ごとの sales の移動平均を計算するために使用されています。<WINDOW SPECIFICATION> は、**RANGE** 句を使用することに注意してください。これは、window bounds が month 値をもとに計算し、**ROWS** 句のようにシンプルに隣接するローで計算しないことに注意してください。**ROWS** を使用していれば、別の結果を yield している可能性があります。例えば、特定の月において一部あるいは全ての product の sales が全くない場合などです。

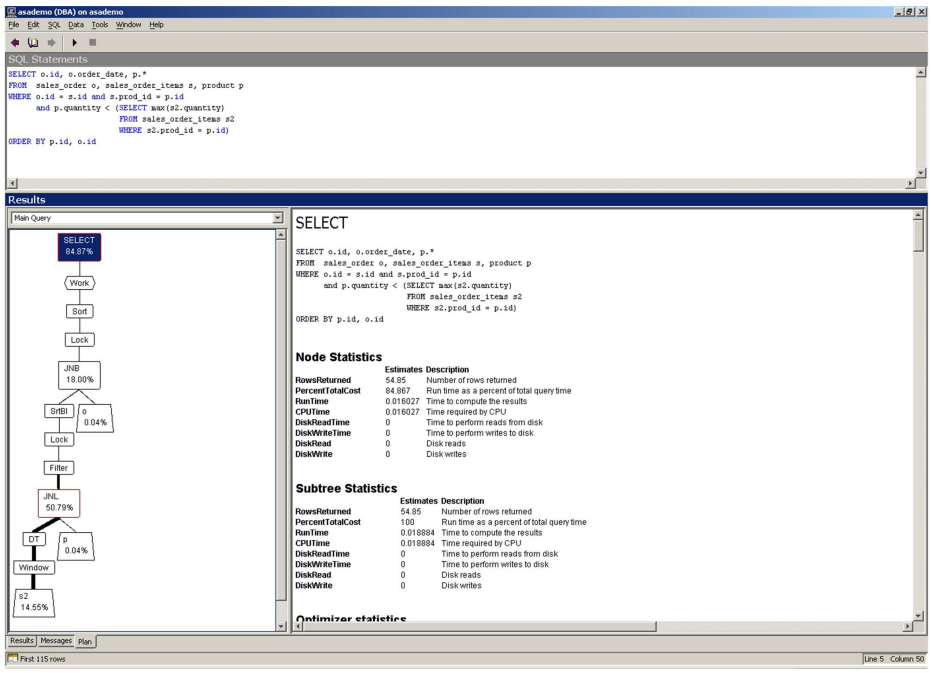

図 2 : 例 24 のクエリーのグラフィカルなプラン

```
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
```

例 24 (相関するサブクエリーの除外)

多くの場合、特定のカラム値と最大または最小値の比較ができることが求められます。しばしば、これらのクエリーをネストされたクエリーとして相関属性を含めて（ outer reference とも言われます）構成します。1 つの例として、以下のクエリーを考えてみてください。これは、product information を含め全ての order をリストしています。Product の quantity-on-hand はその製品の単体での最大 order をカバーすることができません。

```
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
```

このクエリーのグラフィカルなプランを図 2 に示します。クエリーオプティマイザーがこのネストされたクエリーを **product** や **sales_order** テーブルと派生テーブルの join に対してどのように変換したか注意してください。(相関名「**DT**」として示され、window 関数を含む)
SAP SQL Anywhere のオプティマイザーは、このような変換を、ネイティブのネストされたイテレーションなどの異なる実行戦略を利用して予測されるコストを評価した後、コストベースで行います。
相関クエリーを派生テーブルとの join に変換する場合、SAP SQL Anywhere のクエリーオプティマイザーに依存するよりも --- これはセマンティックな分析は複雑なため、シンプルなケースでのみ行われます --- このようなクエリーを window 関数を使用して構成することが可能です。

```
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
```

例 25 (複雑な分析)
以下のクエリーを考えてみます。データベース内のそれぞれの製品のトップ salespeople （ 総 sales で定義 ） をリストする以下のクエリーを考えてみます。

```
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
```

このクエリーは以下の結果を返します。

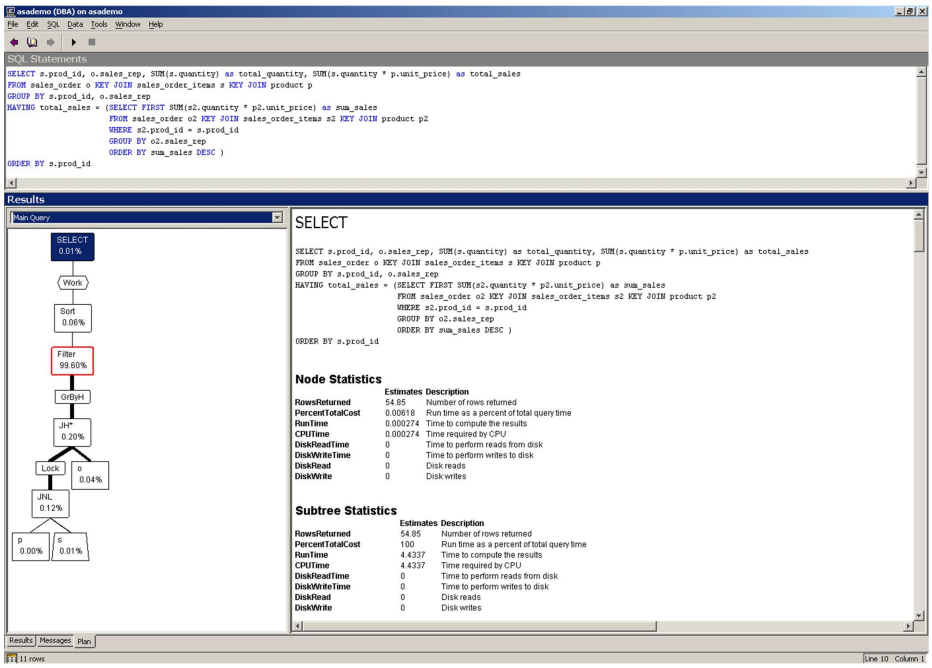

図 3 : 例 25 のメインクエリーブロックのグラフィカルなプラン

R

	Product	*Sales Rep*	*Total Qty*	*Total Sales*
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

オリジナルのクエリーは、**prod_id** はサブクエリーの相関 outer reference のため、特定の製品の最高の sales を決定する相関のサブクエリーを使用して構成されています。しかしながら、このケース (図 3 と 4 のグラフィカルなクエリープラン参照) のように、ネストされたクエリーを使用するのは、expensive なオプションであることがよくあります。なぜならば、サブクエリーは **Group by** 句だけが絡むのではなく、その中で **Order by** 句 --- SAP SQL Anywhere の機能 --- も使用するからです。このため、SAP SQL Anywhere のクエリーオプティマイザーは、(例 24 で行ったように) 同じセマンティクスを保ちながらネストされたクエリーを join としてリライトすることができません。

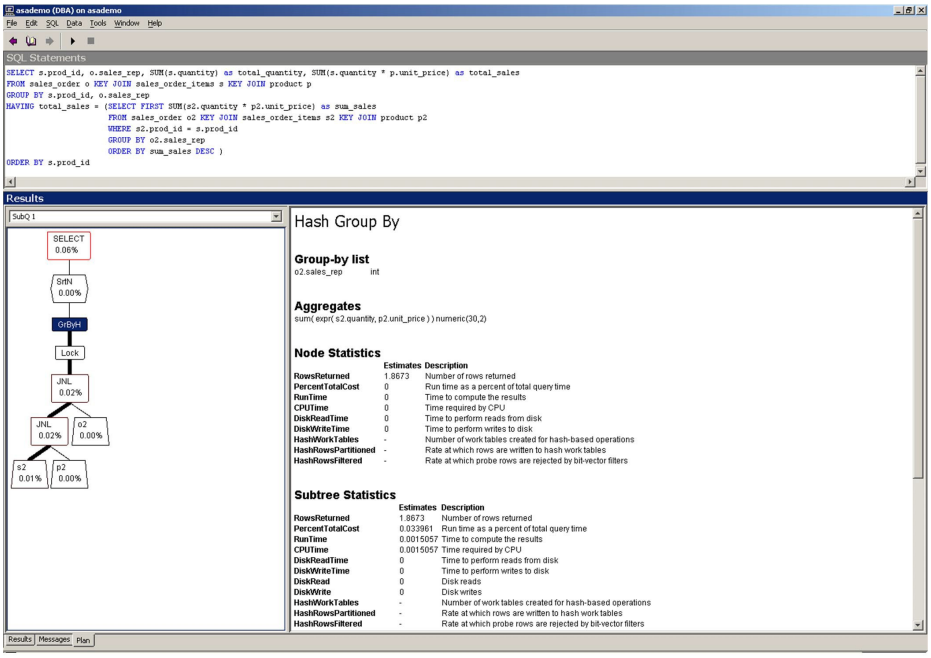

図 4: 例 25 のサブクエリーのグラフィカルなプラン

その結果、クエリー実行中にはサブクエリーは、outer block 内で計算されたそれぞれの derived ローで評価されます。グラフィカルなプラン内の expensive な「Filter」述部に注意してください。オプティマイザーは、クエリーの実行コストの 99 % はこのプラン演算子によるものだと見積もります。図 4 に示されるこのサブクエリーのプランは、なぜ main block のフィルター演算子がこれほどまでに expensive なのかを明確に示しています。このサブクエリーには、2 つの nested-loop joins が絡みます。hashed **Groupby** operation と sort です。

例 26 (window 関数を使用した複雑なクエリー)

以下に示される例 25 内のクエリーのリライトでは、ranking 関数を使用して同じ結果をより効率的に計算します。

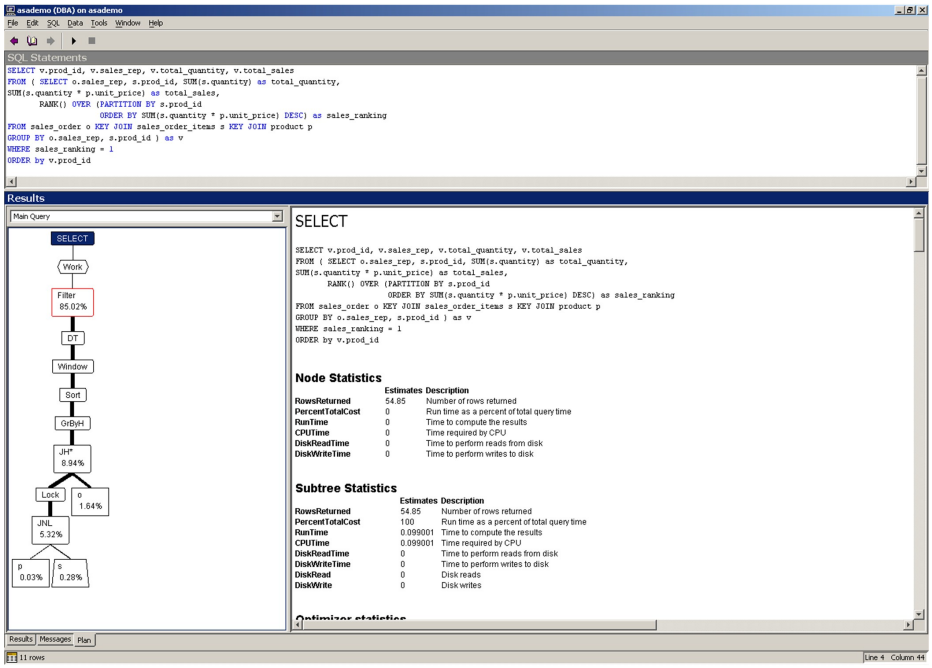

図5：window 関数を使用した例 26 からのクエリーの実行プラン

```
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
```

window 演算子は **Group by** 句の処理の後、そして **Select** リストアイテムの評価とクエリーの **Order by** 句の前に計算されることを思い出してください。グラフィカルなプランで見られるように、3 つのテーブルの join の後、join されたローは **sales_rep** と **prod_id**（「製品識別子」）属性の組み合わせでグループ化されます。その結果、**total_quantity** と **total_sales** の **Sum()** aggregate 関数は、sales representative と product 識別子それぞれの組み合わせで計算することが可能です。

Group by 句の評価に続いて、window を使用して total sales の中間結果のローを降順でランクづけするために **Rank()** 関数が計算されます。<window specification> は、**Partition by** 句が絡むことに注意してください。--- そうすることで、**Group by** 句 は再度パーティション化 （ グループ化 ） されますが、今回は product 識別子でのみです。 **Rank()**関数はそれぞれの product のローをランクづけします --- total sales を降順で --- しかし、product を販売したことのある全ての sales representative に対して行います。この ranking では、トップの sales person を決定するには、単純にランクが 1 ではない派生テーブルの結果のローを reject するように制限することが必要です。(上の結果 R 内のロー7と8) の場合には、**Rank()** は同じ値を返します。そして、要望されたように sales people 690 と 949 のどちらも最終結果に現れます。

4.4.1 FIRST VALUE() と LAST VALUE() 関数

FIRST VALUE() と **LAST VALUE()** 関数は、window の最初と最後のローから値を返します。これによって、クエリーは self-join の必要なく一度に複数のローからの値にアクセスすることが可能になります。

これらの 2 つの関数は、他の window aggregate 関数とは以下の点において異なります。これらは、window と一緒でのみ使用することが可能です。また、他の関数とは異なり、**FIRST VALUE()** と **LAST VALUE()** は、**IGNORE NULLS** 句を許しています。**IGNORE NULLS** が特定された場合、望まれた式の最初または最後の non-null 値が返されます。さもなくば、null であるかに関わらず最初または最後の値は Null で返されます。

例 27 (グループ内への最初のエントリー)

値の順番がつけられたグループ内の最初のエントリーを retrieve するために **FIRST VALUE()** 関数 を使用することが可能です。以下のクエリーは、それぞれの order で、order の最初のアイテムの product 識別子を返します。つまりそれぞれの order の最小の **line_id** のアイテムの **product_id** です。

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

このクエリーは **Distinct** キーワードを使用して、重複を除外しています。**Distinct** なしでは、それぞれの order のそれぞれの item で重複したローが返される可能性があります。

例 28 (highest sales のパーセンテージ)

FIRST VALUE() 関数の一般的な使い方は、current グループ内の最大または最小の値でそれぞれのロー内の値を比較することです。以下のクエリーは、それぞれの sales representative の total sales 計算します。As well as 同一 product の最大 total sales の関係と同様、パーセンテージで表されています。

```
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
```

例 29 (データの高密度化)

FIRST VALUE と **LAST VALUE** 関数 は、**IGNORE NULLS** 句とともに使用された場合のデータの高密度化に便利です。例えば、毎日の総 sales が最大の sales representative が「Representative of the Day」を獲得すると仮定します。以下のクエリーは、2001年4月の最初の週に win した sales representative をリストアップします。

```
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
```

このクエリーは、派生テーブル v 内のパーティショニングが、**part_id** ではなく **order_date** で発生することを除き、例 26 のものにとても似ています。結果のクエリーを下に示します。

R

	order_date	*rep_of_the_day*
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

Sales がなかった日については結果が返されなかったことに注意してください。この結果セットは、sales がなかった day では、それ以前の日から win している representative は継続して Representative of the Day であると宣言することによって高密度化できます。

このようなデータの高密度化は、以下のクエリーで行うことができます。

```
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
```

このクエリーは以下の結果を返します。

R

	order_date	*rep_of_the_day*
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

以前のクエリーからの派生テーブル v は、派生テーブル **d** に join されています。これには、検討中の全ての date が含まれます。これは、望まれる各 day のローを yield します。しかし、この outer join は、sale のない date の sales_rep カラムに **NULL** を含みます。LAST_VALUE 関数を使用し、ある特定のローの **rep_of_the_day** を対応する日に至るまでの最後の non-null 値の **sales_rep** となるよう定義することで、この問題を解決することができます。

A OLAP 関数の BNF 文法

以下のバッカス・ナウア記法(Backus-Naur Form ） 文法は、SAP SQL Anywhere 10 に実装されている様々な ANSI SQL 分析関数特定のシンタックスのサポートの概要を示したものです。

(1) 〈SELECT LIST EXPRESSION〉 ::=
〈EXPRESSION〉
| 〈GROUP BY EXPRESSION〉
| 〈AGGREGATE FUNCTION〉

| ⟨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) 〈VALUE AGGREGATE FUNCTION〉 ::=
〈VALUE AGGREGATE FUNCTION TYPE〉 〈LEFT PAREN〉 〈EXPRESSION〉
[**IGNORE NULLS**] 〈RIGHT PAREN〉

(19) 〈VALUE AGGREGATE FUNCTION TYPE〉 ::=
FIRST_VALUE | **LAST_VALUE**

(20) 〈WINDOW NAME OR SPECIFICATION〉 ::=
〈WINDOW NAME〉 | 〈IN-LINE WINDOW SPECIFICATION〉

(21) 〈WINDOW NAME〉 ::= 〈IDENTIFIER〉

(22) 〈IN-LINE WINDOW SPECIFICATION〉 ::= 〈WINDOW SPECIFICATION〉

(23) 〈WINDOW CLAUSE〉 ::= **WINDOW** 〈WINDOW DEFINITION LIST〉

(24) 〈WINDOW DEFINITION LIST〉 ::=
〈WINDOW DEFINITION〉 [{ 〈COMMA〉 〈WINDOW DEFINITION〉 } . . .]

(25) 〈WINDOW DEFINITION〉 ::=
〈NEW WINDOW NAME〉 **AS** 〈WINDOW SPECIFICATION〉

(26) 〈NEW WINDOW NAME〉 ::= 〈WINDOW NAME〉

(27) 〈WINDOW SPECIFICATION〉 ::=
〈LEFT PAREN〉 〈WINDOW SPECIFICATION DETAILS〉 〈RIGHT PAREN〉

(28) 〈WINDOW SPECIFICATION DETAILS〉 ::=
[〈EXISTING WINDOW NAME〉]
[〈WINDOW PARTITION CLAUSE〉]
[〈WINDOW ORDER CLAUSE〉]
[〈WINDOW FRAME CLAUSE〉]

(29) 〈EXISTING WINDOW NAME〉 ::= 〈WINDOW NAME〉

(30) 〈WINDOW PARTITION CLAUSE〉 ::=
PARTITION BY 〈WINDOW PARTITION EXPRESSION LIST〉

(31) 〈WINDOW PARTITION EXPRESSION LIST〉 ::=
〈WINDOW PARTITION EXPRESSION〉
[{ 〈COMMA〉 〈WINDOW PARTITION EXPRESSION〉 } . . .]

(32) 〈WINDOW PARTITION EXPRESSION〉 ::= 〈EXPRESSION〉

(33) 〈WINDOW ORDER CLAUSE〉 ::= 〈ORDER SPECIFICATION〉

(34) 〈WINDOW FRAME CLAUSE〉 ::=
〈WINDOW FRAME UNIT〉
〈WINDOW FRAME EXTENT〉

(35) 〈WINDOW FRAME UNIT〉 ::= **ROWS** | **RANGE**

(36) 〈WINDOW FRAME EXTENT〉 ::= 〈WINDOW FRAME START〉 | 〈WINDOW FRAME BETWEEN〉

(37) 〈WINDOW FRAME START〉 ::=
UNBOUNDED PRECEDING
| 〈WINDOW FRAME PRECEDING〉
| **CURRENT ROW**

(38) 〈WINDOW FRAME PRECEDING〉 ::= 〈UNSIGNED VALUE SPECIFICATION〉 PRECEDING

(39) 〈WINDOW FRAME BETWEEN〉 ::=
BETWEEN 〈WINDOW FRAME BOUND 1〉 **AND** 〈WINDOW FRAME BOUND 2〉

(40) 〈WINDOW FRAME BOUND 1〉 ::= 〈WINDOW FRAME BOUND〉

(41) 〈WINDOW FRAME BOUND 2〉 ::= 〈WINDOW FRAME BOUND〉

(42) 〈WINDOW FRAME BOUND〉 ::=
〈WINDOW FRAME START〉
| **UNBOUNDED FOLLOWING**
| 〈WINDOW FRAME FOLLOWING〉

(43) 〈WINDOW FRAME FOLLOWING〉 ::= 〈UNSIGNED VALUE SPECIFICATION〉 **FOLLOWING**

(44) 〈GROUP BY EXPRESSION〉 ::= 〈EXPRESSION〉

(45) 〈SIMPLE GROUP BY TERM〉 ::=
〈GROUP BY EXPRESSION〉
| 〈LEFT PAREN〉 〈GROUP BY EXPRESSION〉 〈RIGHT PAREN〉
| 〈LEFT PAREN〉 〈RIGHT PAREN〉

(46) 〈SIMPLE GROUP BY TERM LIST〉 ::=
(47) 〈SIMPLE GROUP BY TERM〉 [{ 〈COMMA〉 〈SIMPLE GROUP BY TERM〉 } .. .]
(48) 〈COMPOSITE GROUP BY TERM〉 ::=
(49) 〈LEFT PAREN〉 〈SIMPLE GROUP BY TERM〉
(50) [{ 〈COMMA〉 〈SIMPLE GROUP BY TERM〉 } .. .]
(51) 〈RIGHT PAREN〉
(52) 〈ROLLUP TERM〉 ::= ROLLUP 〈COMPOSITE GROUP BY TERM〉
(53) 〈CUBE TERM〉 ::= CUBE 〈COMPOSITE GROUP BY TERM〉
(54) 〈GROUP BY TERM〉 ::=
(55) 〈SIMPLE GROUP BY TERM〉
(56) | 〈COMPOSITE GROUP BY TERM〉
(57) | 〈ROLLUP TERM〉
(58) | 〈CUBE TERM〉
(59) 〈GROUP BY TERM LIST〉 ::=
(60) 〈GROUP BY TERM〉 [{ 〈COMMA〉 〈GROUP BY TERM〉} .. .]
(61) 〈GROUP BY CLAUSE〉 ::= GROUP BY 〈GROUPING SPECIFICATION〉
(62) 〈GROUPING SPECIFICATION〉 ::=
(63) 〈GROUP BY TERM LIST〉
(64) | 〈SIMPLE GROUP BY TERM LIST〉 WITH ROLLUP
(65) | 〈SIMPLE GROUP BY TERM LIST〉 WITH CUBE
(66) | 〈GROUPING SETS SPECIFICATION〉
(67) 〈GROUPING SETS SPECIFICATION〉 ::=
GROUPING SETS 〈LEFT PAREN〉 〈GROUP BY TERM LIST〉 〈RIGHT PAREN〉
(68) 〈ORDER SPECIFICATION〉 ::= ORDER BY 〈SORT SPECIFICATION LIST〉

参考

[1] Jay Devore and Roxy Peck. *Statistics: The Exploration and Analysis of Data*. WestPublishing Company, St. Paul, Minnesota, 1986.

[2] International Standards Organization. (ansi/iso) *9075-2,* sql *Foundation*, September1999.

[3] International Standards Organization. (ansi/iso) *9075:1999-AM1, On-Line Analytical Processing*sql/olap*Amendment*,November1999.

[4] International Standards Organization. (ansi/iso) *9075-2,* sql *Foundation*, September2003.

[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.

www.sap.com/contactsap

THE BEST RUN SAP