What’s new in SQL:2011
Fred Zemke, Oracle Corporation, fred.zemke@oracle.com

ABSTRACT

SQL:2011 was published in December 2011, replacing the former version (SQL:2008) as the most recent update to the SQL standard for relational databases. This paper surveys the new non-temporal features of SQL:2011.

1. INTRODUCTION

SQL (pronounced es-cue-el) is the relational database standard published jointly by ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission). In December 2011, ISO/IEC published the latest edition of SQL, SQL:2011.

Many readers are perhaps most familiar with SQL-86, SQL-89 and SQL-92 published by ANSI (the American National Standards Institute). Subsequently, the SQL standard has been developed internationally under the auspices of ISO/IEC. Since SQL-92, the major revisions of the SQL standard have been SQL:1999, SQL:2003, SQL:2008, and now SQL:2011. Articles in SIGMOD Record on prior versions of SQL may be found in references [10] - [12].

The SQL standard is published in multiple volumes, called parts. Currently there are nine parts, numbered 1, 2, 3, 4, 9, 10, 11, 13 and 14. (The gaps are for parts that have been withdrawn for various reasons; once a part number is issued, it is not recycled). Parts 3 and following were each published for the first time between the major release years (i.e., not 1992, 1999, 2003, 2008 or 2011) and subsequently “progressed” together with other extant parts. The complete list of the parts of SQL is found in the references [1] - [9].

By far the most important part is 2, Foundation [2], which is also the largest at 1470 pages (about 100 pages larger than in SQL:2008). Only five of the parts were revised in SQL:2011; for the other four parts, the version published in SQL:2008 remains in effect. This paper concentrates on the new features in SQL/Foundation:2011.

The SQL standard specifies mandatory and optional features of SQL. The mandatory features, known as Entry Level in SQL-92 and Core in the subsequent international versions, have not changed appreciably since SQL:1999. Thus the growth in the standard has been in the optional features.

Perhaps the most important new features in SQL:2011 are in the area of temporal databases. This article does not have enough space to adequately cover that topic, so it will be discussed in a future article.

The most important new non-temporal features of SQL:2011 are the following:
- DELETE in MERGE
- Pipelined DML
- CALL enhancements
- Limited fetch capability
- Collection type enhancements
- Non-enforced table constraints
- Window enhancements

Additionally, there are new DDL features to improve the usability of generated and identity columns, which are not discussed further in this article due to space limitations.

2. DELETE in MERGE

MERGE is a data manipulation command introduced in SQL:2003 and enhanced in SQL:2008. Here is an example that is permitted by SQL:2008. Suppose that Inventory (Part, Qty) is a table that lists parts and quantity on hand, and Changes (Part, Qty, Action) is a table of changes to be applied to Inventory. The Action column has two values, with the following meanings:
- 'Mod': add Changes.Qty to Inventory.Qty if the part already exists in Inventory.
- 'New': add a new row to Inventory using the values of Changes.Part and Changes.Qty.

In SQL:2008 this might be accomplished using the following statement:

```
MERGE INTO Inventory AS I
USING Changes AS C
ON I.Part = C.Part
WHEN MATCHED AND
  I.Action = 'Mod'
  THEN UPDATE
    SET Qty = I.Qty + C.Qty
WHEN NOT MATCHED AND
  I.Action = 'New'
  THEN INSERT (Part, Qty)
  VALUES (C.Part, C.Qty)
```

In the preceding example, the rows of Inventory
and Changes are matched using the join condition 
I.Part = C.Part. When a row of Changes 
matches a row of Inventory, and Action is Mod, 
then the row of Inventory is updated. When a row of 
Changes has no match in Inventory, and Action is New, 
then a new row is inserted in Inventory.

SQL:2011 has added the ability to perform DELETE 
within MERGE. This permits the following additional 
Action:
• 'Dis': delete the part from Inventory since it 
has been discontinued.

This additional value of Action can be supported 
with the following command:

```
MERGE INTO Inventory AS I
USING Changes AS C
ON I.Part = C.Part
WHEN MATCHED AND
  I.Action = 'Mod'
  THEN UPDATE
  SET Qty = Qty + C.Qty
WHEN MATCHED AND
  I.Action = 'Dis'
  THEN DELETE
WHEN NOT MATCHED AND
  I.Action = 'Mod'
  THEN INSERT
  VALUES (C.Part, C.Qty)
```

The new capability in this example is the underlined 
DELETE, which deletes a row from Inventory.

### 3. Pipelined DML

Pipelined DML gives the ability to perform data 
change commands (INSERT, UPDATE, DELETE, 
MERGE) within a SELECT command.

A data change command has one or two “delta 
tables” containing the specific rows that are touched. A 
DELETE has only an old delta table (the rows to be 
deleted). An INSERT has only a new delta table (the 
rows to be inserted). An UPDATE has both an old delta 
table and a new delta table; the old delta table contains 
the “before images” and the new delta table contains the 
“after images”. The delta table(s) of a MERGE are the 
unions of the old delta tables and the new delta tables of 
the INSERT, UPDATE and DELETE commands found 
within the MERGE.

Pipelined DML provides access to either the old 
delta table or the new delta table of a data manipulation 
command within a SELECT. For example,

```
SELECT Oldtable.Empno
FROM OLD TABLE (DELETE FROM Emp 
  WHERE Deptno = 2) AS Oldtable
```

In the preceding example, the FROM clause has a 
DELETE command nested within it. The key words 
OLD TABLE indicate that rows from the old delta table of 
this DELETE are desired. The DELETE is executed, 
and afterwards, the rows of the old delta table are used 
to create the result, which is a list of Empno of the 
deleted rows.

The key words NEW TABLE may be used to access 
the new delta table of an INSERT, UPDATE or MERGE, 
for example

```
SELECT Newtable.Empno
FROM NEW TABLE (UPDATE Emp 
  SET Salary = 0 
  WHERE Empno = 0 
  WHERE Empno > 100)
AS Newtable
```

The preceding example sets certain salaries to 0 and 
returns a result consisting of the Empno of those rows 
whose salary is set to 0.

When using NEW TABLE, the new delta table is 
computed by constructing the set of new candidate rows 
as indicated by the INSERT, UPDATE or MERGE state-
ment. New candidate rows may be modified by 
BEFORE triggers, after which they are applied to the 
target table. The new delta table is a snapshot of this 
point in query processing. There are later stages (cas-
caded referential actions and AFTER triggers) whose 
effects are not captured in the new delta table. Thus, if 
there are applicable cascaded referential actions or 
AFTER triggers, the final value of the target table may 
differ from the result of NEW TABLE. If the user is con-
cerned about these, the user can specify instead FINAL 
TABLE. There is no “final delta table”, so the FINAL 
TABLE option merely raises an exception if any cas-
caded referential action or AFTER trigger touches the 
target table.

### 4. CALL enhancements

The CALL statement, introduced in Part 4 PSM in 
1996 and subsequently incorporated in Part 2 Founda-
tion in 1999, is used to invoke an SQL-invoked proce-
dure. Here is an example of how to create an SQL-
invoked procedure using features prior to SQL:2011:

```
CREATE PROCEDURE P ( 
  IN A INTEGER, 
  OUT B INTEGER ) ... 
```

The ellipsis omits details of the syntax that specify 
such things as the host language of the procedure, the 
path to use to invoke it, etc. This example defines a pro-
cedure P with two integer parameters A and B; A is an 
input parameter and B is an output parameter. It is also 
possible to declare a parameter that is both input and 
output using the keyword INOUT.
Once \( P \) is defined, it can be invoked using a \texttt{CALL} statement, like this:
\[
\text{CALL } P \left( 1, \text{ :MyVar} \right)
\]
Here \texttt{MyVar} might be the name of an embedded variable that will receive the value assigned to the parameter \texttt{B} of \texttt{P}.

SQL:2011 provides two enhancements to SQL-invoked procedures:
- named arguments, and
- default input arguments.

Named arguments are illustrated in the following example:
\[
\text{CALL } P \left( \text{B} \Rightarrow \text{ :MyVar}, \text{ A} \Rightarrow 1 \right)
\]
This statement is equivalent to the first example of a \texttt{CALL} statement. Using named arguments, the user can specify the arguments in any order.

The new default input argument feature will be illustrated using the following procedure definition:
\[
\text{CREATE PROCEDURE } P \left( \right) \\
\text{ IN A INTEGER DEFAULT 2 } \\
\text{ OUT B INTEGER } \ldots
\]
The underlined phrase \texttt{DEFAULT 2} is new in SQL:2011. This syntax may be used to specify the default value of an input parameter (A in this example). Output parameters, including in-out parameters, do not support default values.

Given the preceding procedure definition, it might be invoked like this:
\[
\text{CALL } P \left( \text{B} \Rightarrow \text{ :MyVar} \right)
\]
Note that this invocation does not specify the argument \texttt{A}. Since \texttt{A} is omitted, the default value 2 will be used, so the example is equivalent to
\[
\text{CALL } P \left( \text{B} \Rightarrow \text{ :MyVar}, \text{ A} \Rightarrow 2 \right)
\]
or
\[
\text{CALL } P \left( 2, \text{ :MyVar} \right)
\]

5. Limited fetch capability

A common application requirement is to fetch a subset of a query. For example, in a table of scored data, it might be desired to fetch only the top three results. Or during application development, it may be desired to fetch just ten arbitrary rows as a sample. Or in a deployed application, it may be desired to fetch only as many rows as fit in a limited display space. SQL:2008 introduced syntax to support such scenarios; this syntax has been enhanced in SQL:2011.

An example supported by SQL:2008 is
\[
\begin{align*}
\text{SELECT Name, Salary} \\
\text{FROM Emp} \\
\text{ORDER BY Salary DESCENDING} \\
\text{FETCH FIRST 10 ROWS ONLY}
\end{align*}
\]
The preceding example will obtain the top ten wage earners from \texttt{Emp}. If there is a tie for the ninth, tenth and eleventh place, it is nondeterministic which two of the three ties will be returned. New in SQL:2011, one can write
\[
\begin{align*}
\text{SELECT Name, Salary} \\
\text{FROM Emp} \\
\text{ORDER BY Salary DESCENDING} \\
\text{FETCH FIRST 10 ROWS WITH TIES}
\end{align*}
\]
The underlined key words \texttt{WITH TIES} (replacing the last key word \texttt{ONLY} in the first example) indicate that any rows tied with the tenth row should also be returned, making the result deterministic.

Another new feature is the ability to fetch a percentage of rows, as in this example:
\[
\begin{align*}
\text{SELECT Name, Salary} \\
\text{FROM Emp} \\
\text{ORDER BY Salary DESCENDING} \\
\text{FETCH FIRST 10 PERCENT ROWS ONLY}
\end{align*}
\]
The \texttt{PERCENT} keyword may also be used with the \texttt{WITH TIES} option.

The final new feature is the ability to start the retrieval at a fixed offset, as in this example
\[
\begin{align*}
\text{SELECT Name, Salary} \\
\text{FROM Emp} \\
\text{ORDER BY Salary DESCENDING} \\
\text{OFFSET 10 ROWS}
\end{align*}
\]
The underlined phrase \texttt{OFFSET 10 ROWS} specifies to skip the first ten rows; thus this example will retrieve the second ten highest wage earners from \texttt{Emp}. The underlined noise word \texttt{NEXT} is actually a synonym for \texttt{FIRST} (seen in prior examples) which the user might prefer for readability when fetching at a positive offset.

6. Collection type enhancements

SQL has two kinds of collection types: arrays (introduced in SQL:1999) and multisets (introduced in SQL:2003). Collection types are used to represent homogenous collections of elements (every element of a collection has the same data type, called the element type of the collection).

An array is an ordered collection of values, the elements of the array. The cardinality of an array is the number of elements in the array. An SQL array has variable cardinality, from zero up to a declared maximum cardinality. Thus if \texttt{C} is a column of array type, the cardinality of \texttt{C} may vary from row to row. An array may be atomically null, in which case its cardinality is regarded as null. A null array (cardinality null) is distinguished from both an empty array (cardinality 0) and an array whose every element is null (cardinality > 0).
If an array has cardinality less than the declared maximum, then the unused cells of the array are non-existent (they are not treated as implicitly null). Individual elements of an array can be referenced using square brackets to enclose a subscript; for example, $C[5]$ references the fifth element of $C$.

Since SQL:1999, the cardinality of an array can be learned using the `CARDINALITY` function. New in SQL:2011, the maximum cardinality of an array can be learned using the `ARRAY_MAX_CARDINALITY` function. This is useful, for example, in writing general-purpose routines, avoiding the need to hard-code the maximum cardinality.

SQL:1999 allows assignment to a subscripted element of an array, which will either replace an existing element of the array, or increase the cardinality of the array. Any other change to an array value is done by assigning to the array as a whole, i.e., replacing the entire array. This meant that there was no way easy to remove elements from an array. New in SQL:2011, the function `TRIM_ARRAY` can be used to remove elements from the end of an array.

The other kind of collection type is multiset, which is an unordered homogenous collection that permits duplicates among the elements. There is no declared maximum cardinality for a multiset, though implementations will have physical or architectural limits. Since a multiset is unordered, there is no subscript notation to address an individual element.

New in SQL:2011, it is possible to define distinct types that are sourced from collection types. Previously, distinct types as introduced in SQL:1999 are user-defined types sourced from a predefined type. For example, a user might define `Shoesize` as a distinct type sourced from `INTEGER`. A value of `Shoesize` is an `INTEGER` value, but without the built-in semantics; for example, addition is not defined on `Shoesize`. Instead, the user can define the semantics of `Shoesize` by defining methods or other user-defined routines to manipulate values of type `Shoesize`.

In SQL:1999, the only source types for distinct types were predefined types. SQL:2011 now permits collection types as source types. For example, under SQL:1999, the user could create a column or an SQL variable that is an array of `Shoesize`, but could not create a distinct type that is array of `INTEGER`. There is no difference between an array of `Shoesize` and an array of `INTEGER` at the storage level, but semantically, SQL:1999 had no way to define methods on the array as a whole, only on the elements of the array. SQL:2011 has filled this gap by allowing distinct types sourced from collection types.

### 7. Non-enforced table constraints

Table constraints are declared restrictions on the possible values in rows of a table. There are three kinds: unique constraints (requiring the value in a column or set of columns to be unique across the rows of the table), referential constraints (to enforce parent-child relationships between tables) and check constraints (to enforce a Boolean condition within each row, for example, that hire date must be greater than birth date). Table constraints have been part of SQL since its inception in SQL-86.

Under most circumstances, the user wishes constraints to be enforced, since they are vital to maintaining the integrity of the data. However, there are situations in which a user wishes to temporarily turn off one or more constraint checks, such as bulk loads or replications. SQL-92 provided the ability to defer constraint checking to the end of a transaction. However, this capability does not really address the bulk load scenario, since the user will want to commit periodically during a large data transfer as a precaution against a system failure.

SQL:2011 has addressed this problem by providing syntax to alter a constraint to be either enforced or not enforced. By default, a constraint is enforced, but the user can set it to be not enforced, for example, during a bulk load. A non-enforced constraint is not checked, not even at commit time. However, when a non-enforced constraint is subsequently set to be enforced, then the constraint will be checked on all the data. Generally, such enforcement is more efficient than doing it incrementally during the load.

### 8. Window enhancements

Windows and window functions were first introduced via an amendment in 2000 to SQL:1999, and were incorporated directly in SQL/Foundation:2003 and subsequent editions of the standard.

To review, a window allows a user to optionally partition a data set, optionally order the rows of each partition, and finally specify a collection of rows (called the window frame) that is associated with each row. The window frame of a row $R$ is some subset of the window partition of $R$. For example, the window frame may consist of all rows from the beginning of the partition up through and including $R$, according to the window ordering.

A window function is a function that computes a value for a row $R$ using the collection of rows in the window frame of $R$. For example, an aggregate such as `SUM` might be computed over a window, as in this example:
SELECT Acctno, TransDate, SUM (Amount) OVER
  ( PARTITION BY Acctno
    ORDER BY TransDate
    ROWS BETWEEN 8A%2@%#@@#312+--K%1C#%@+'#%+@%B8$)%#_,#01%1C+1%21%
FROM Accounts

In the preceding example, Accounts is a table containing data including Acctno, TransDate and Amount. The OVER clause specifies the window, which is partitioned by Acctno, ordered by TransDate, and finally, for each row \( R \), the window frame consists of all rows from the beginning of the partition through \( R \). Thus this query might be used to provide running account balances for each account number, in order of transaction date.

SQL:2011 has added the following window enhancements:
- NTILE
- Navigation within a window
- Nested navigation in window functions
- GROUPS option

These new features are described in the following subsections.

8.1 NTILE

NTILE is a window function that apportions an ordered window partition into some positive integer number \( n \) of buckets, numbering the buckets from 1 through \( n \). If the number of rows \( m \) in the partition is not evenly divisible by \( n \), then the extra rows are handled by making the first \( r \) buckets one row larger, where \( r \) is the remainder of the integer division \( m/n \). For example,

\[
\text{SELECT Name, NTILE(3) OVER (ORDER BY Salary ASC) AS Bucket FROM Emp}
\]

In this example, suppose there are 5 employees. The query asks to place them into 3 buckets. The remainder of 5/3 is 2; therefore the first two buckets will have 2 rows and the last bucket will have 1 row. Suppose the employees, in ascending order of Salary, are named Joe, Mary, Tom, Alice, and Frank. Then Joe and Mary are assigned to bucket 1, Tom and Alice to bucket 2, and Frank to bucket 3.

8.2 Navigation within a window

Five window functions have been added to evaluate an expression in a row \( R2 \) at interesting places in the window frame of a current row \( R1 \): LAG, LEAD, NTH_VALUE, FIRST_VALUE, and LAST_VALUE.

8.2.1 LAG and LEAD

LAG and LEAD are window functions that provide access to a row \( R2 \) at some offset from the current row \( R1 \) within the window frame of \( R1 \). For example, given a time series of prices, suppose you wish to display a Price and the Price immediately prior in the time series. This can be done with this query:

\[
\text{SELECT Price AS CurPrice, LAG (Price) OVER (ORDER BY Tstamp) AS PrevPrice FROM Data}
\]

In this example, the Price values in Data have been ordered by the timestamp Tstamp. For each row of Data, the result has two columns, CurPrice and PrevPrice. CurPrice is the current row’s value of Price and PrevPrice is the immediately preceding value of Price.

The default offset, as in the preceding example, is 1 row. Other offsets may be specified as the second argument to LAG using an unsigned integer literal, for example

\[
\text{SELECT Price AS CurPrice, LAG (Price, 2) OVER (ORDER BY Tstamp) AS PrevPrice2 FROM Data}
\]

The first \( n \) rows, where \( n \) is the offset, will have no predecessor, and the LAG function will result in null by default. A third optional argument may be used to specify a different default, like this:

\[
\text{SELECT Price AS CurPrice, LAG (Price, 2, 0) OVER (ORDER BY Tstamp) AS PrevPrice2a FROM Data}
\]

In the preceding example, in the first two rows, the value of PrevPrice2a is 0.

The final option on LAG is to compress out nulls before offsetting. This is illustrated in the following:

\[
\text{SELECT Price AS CurPrice, LAG (Price, 3, 0) IGNORE NULLS OVER (ORDER BY Tstamp) AS PrevPrice3 FROM Data}
\]

The preceding example counts backwards through three rows of non-null Price; if there are not that many, the value of PrevPrice3 is 0. If desired, the keywords RESPECT NULLS may be used for the default behavior, which is to retain null data before counting rows backwards from the current row.

LEAD is essentially the same as LAG, except that it looks forward in the ordered window partition rather than backwards.
8.2.2 NTH_VALUE

LAG and LEAD evaluate an expression in a row R2 that is located relative to the current row R1. The
NTH_VALUE window function is similar, except that it navigates to a row R2 that is at an offset from either the
first or last row of the window frame of R1. For example,

```
SELECT Price AS CurPrice,
       NTH_VALUE (Price, 1)
FROM First
IGNORE NULLS
OVER ( ORDER BY Tstamp
       ROWS BETWEEN 3 PRECEDING
             AND 3 FOLLOWING )
AS EarlierPrice
FROM Data
```

In this example, EarlierPrice is evaluated as follows:
- Form the window frame of the current row R1.
- Evaluate the expression Price in each row of the
window frame.
- Since IGNORE NULLS is specified, remove any
nulls from the collection of values.
- Starting at the first remaining value, move forward
(because FROM FIRST is specified) by one row
(because the offset in the second argument is 1)
- The value of EarlierPrice is the value of
Price in the chosen row.

Instead of FROM FIRST, one specifies FROM
LAST in order to offset from the last row of the window
frame. The offset is still a positive integer, though it is
used to offset backwards through the window frame.

Instead of IGNORE NULLS, one can specify
RESPECT NULLS to retain nulls in the set of candidate
rows prior to offsetting.

8.2.3 FIRST_VALUE and LAST_VALUE

The FIRST_VALUE and LAST_VALUE window
functions are special cases of NTH_VALUE, in which
the offset is always 0. FIRST_VALUE is equivalent to
NTH_VALUE using the FROM FIRST option with an
offset of 0, while LAST_VALUE is equivalent to
NTH_VALUE using FROM LAST with an option of 0.
Both FIRST_VALUE and LAST_VALUE support the
choice of IGNORE NULLS or RESPECT NULLS.

8.3 Nested navigation in window functions

The window functions LAG, LEAD, NTH_VALUE,
FIRST_VALUE and LAST_VALUE enable the user to
evaluate an expression at a row R2 at some point relative
to the window frame of the current row R1. However, these functions cannot be nested within other
window functions. Consider, for example, the following
query: how many times in the past 30 trades has the
Price been greater than the current Price? This
query can be answered with a self-join, which users fre-
frequently find difficult to write, and then the DBMS finds
difficult to optimize. Instead of using a self-join, it
would be desirable to use a window to survey the past
30 trades. Then the problem reduces to counting the
number of rows in the window frame in which the
Price exceeds the current Price. Using new features
in SQL:2011, the query can be expressed as follows:

```
SELECT Tstamp,
       SUM ( CASE WHEN Price >
                VALUE_OF (Price AT
                           CURRENT_ROW)
            THEN 1 ELSE 0 )
OVER ( ORDER BY Tstamp
       ROWS BETWEEN 30 PRECEDING
            AND CURRENT ROW )
FROM Data
```

In the preceding query, the SUM is a window aggre-
gate over a window that surveys the preceding 30 trades.
The SUM is performed on a collections of 1’s and 0’s, so
it is effectively a count of the number of 1’s that are
summed. There is a 1 for every Price in the window
frame that exceeds the Price at the current row. To
obtain the Price at the current row, the VALUE_OF
function, new in SQL:2011, is used. The VALUE_OF
function specifies an expression to evaluate and a row of
the window frame at which to perform the evaluation.
In this example, the keyword CURRENT_ROW, called a
row marker, indicates the current row. Other row mark-
ers can be used to indicate the beginning or end of the
window partition or the window frame. Additionally, a
row marker may have an integer offset that is added or
subtracted.

8.4 GROUPS option

The window frame of a row R consists of a set of
rows in the same window partition as R, defined by a
starting and an ending position. The starting position
UNBOUNDED PRECEDING is absolute, as is the ending
position UNBOUNDED FOLLOWING. CURRENT ROW
can be used as either a starting or an ending position,
and is simply the position of R. It is also possible to
specify relative starting or ending positions using an
offset from R. For example

```
SELECT Acctno, TransDate,
       SUM (Amount) OVER
             ( PARTITION BY Acctno
               ORDER BY TransDate
               ROWS BETWEEN
               3 PRECEDING
```
AND 3 FOLLOWING )
FROM Accounts
In the preceding example, the window frame is measures in ROWS and consists of up to 7 rows (3 before \( R \), \( R \) itself and 3 after \( R \)). Alternatively, the window frame might be measured quantitatively, as in this example
SELECT Acctno, TransDate,
SUM (Amount) OVER
( PARTITION BY Acctno
ORDER BY TransDate
RANGE BETWEEN
INTERVAL '1' MONTH PRECEDING
AND
INTERVAL '1' MONTH FOLLOWING )
FROM Accounts
The preceding example uses RANGE to specify the window frame by offsetting the value of the sort column Transdate plus or minus 1 month from \( R \).

The options ROWS and RANGE have their respective advantages and disadvantages. RANGE can only be used with a single sort key, and the sort key must be of a data type that supports addition and subtraction. ROWS will work with any number or data types of sort keys, but results can be non-deterministic since counting by rows may bisect a contiguous group of rows that are identical on the sort keys.

SQL:2011 introduced a third option, GROUPS, which combines some of the features of both ROWS and RANGE. GROUPS operates by counting groups of rows that are identical on the sort keys. Thus GROUPS can work with any number and data types of sort keys, and still give a deterministic result. For example,
SELECT Acctno, TransDate,
SUM (Amount) OVER
( PARTITION BY Acctno
ORDER BY TransDate
GROUPS BETWEEN
3 PRECEDING
AND 3 FOLLOWING )
FROM Accounts
The window frame of a row \( R \) consists of up to 7 groups of rows (3 groups before \( R \), the group of \( R \) itself, and 3 groups after \( R \)), where a group is a set of rows that have identical TransDate.

9. Acknowledgements
The author thanks his colleagues on the ANSI SQL committee Jim Melton (Oracle), Krishna Kulkarni (IBM) and Jan-Eike Michels (IBM) for reviewing this article.

10. References