SQL:2003 Has Been Published

Andrew Eisenberg
IBM, Westford, MA 01886
andrew.eisenberg@us.ibm.com

Jim Melton
Oracle Corp., Sandy, UT 84093
jim.melton@acm.org

Krishna Kulkarni
IBM, San Jose, CA 94151
krishnak@us.ibm.com

Jan-Eike Michels
IBM, San Jose, CA 94151
janeike@us.ibm.com

Fred Zemke
Oracle Corp., Redwood Shores, CA 94065
fred.zemke@oracle.com

Guest Introduction

SQL:2003 has finally achieved final publication as an International Standard, replacing SQL:1999. SQL:2003 is popularly believed to be largely a "bug-fix release" of the SQL standard — except, of course, for the SQL/XML work on which we have previously reported. However, as you will learn from this and future columns, there are many compelling new features in the 2003 edition of the SQL standard.

We are pleased that three of the more active SQL proposal writers have joined forces to present several of those new features in this month's column. Krishna Kulkarni is, among other responsibilities, the formal International Representative for the INCITS H2 Technical Committee for Database. Jan-Eike Michels is a frequent USA representative to the corresponding international group, ISO/IEC JTC1/SC32/WG3. Fred Zemke is widely acknowledged as a principle expert in many areas of the SQL standard and also a regular USA representative to WG3. In the coming months, we will provide information about even more of SQL:2003's new features.

Introduction

SQL:2003 makes revisions to all parts of SQL:1999 and adds a brand new part, Part 14: SQL/XML (XML-Related Specifications). In addition, there is some slight reorganization of the parts inherited from SQL:1999. A substantial chunk of SQL:1999's Part 2: SQL/Foundation that dealt with the Information Schema and Definition Schema is split off into its own part, Part 11: SQL/Schemata in SQL:2003. SQL:1999's Part 5: SQL/Bindings has been eliminated in SQL:2003 by merging all the material contained in that part into SQL:2003's Part 2: SQL/Foundation.

In this article, we focus on describing the following new features introduced into SQL:2003's Part 2: SQL/Foundation:

- New data types
- Enhancements to SQL-invoked routines
- Extensions to CREATE TABLE statement
- A new MERGE statement,
- A new schema object - the sequence generator,
- Two new sorts of columns - identity columns and generated columns.

Future articles will cover the remaining new features of SQL:2003's Part 2: SQL/Foundation, such as retrospective check constraints, OLAP (on-line analytical processing) extensions in the form of new built-in functions (both scalar functions and aggregate functions) and a new WINDOW clause in query expressions (published previously as an Amendment to SQL:1999), support for the use of sampled data for better performance, improved savepoint handling, enhanced diagnostics management, *etc*.

New Data Types

SQL:2003 retains all data types that existed in SQL:1999 with the exception of the `BIT` and `BIT VARYING` data types. Those two were removed from the standard due to the lack of support in existing SQL database engines, and the lack of expected support in the future. SQL:2003 introduces three new data types: `BIGINT`, `MULTISET`, and `XML`. Since the `XML` data type is part of SQL/XML [1], we will not elaborate on it here further. For further details on

this type, as well as related extensions in Part 14: SQL/XML, the interested reader is referred to [2] and to an upcoming article in a future issue of this publication.

The new data types are first class data types, meaning they can be used in all the contexts that any other (existing) SQL data type can be used; *e.g.*, as column types, parameter and return types of SQL-invoked routines, *etc*.

The `BIGINT` data type is a new numerical type similar to the existing `SMALLINT` and `INTEGER` types, just with a greater precision (or more precisely, a precision no smaller). Though a particular precision for an `INTEGER` type (or any of the other numerical data types) is not mandated in the SQL standard, actual implementations in general support 32-bit `INTEGER` values. Those implementations then usually support 64-bit `BIGINT` values. However, a conforming implementation could also choose any other precision (even the choice of decimal or binary precision is left open in the SQL standard). The `BIGINT` type supports the same arithmetic operations as the `INTEGER` type; *e.g.*, `+`, `-`, `ABS`, `MOD`, *etc*.

The `MULTISET` data type is a new collection type similar to the existing `ARRAY` type, but without an implied ordering. Conceptually, a multiset is an unordered collection of elements, all of the same type (the *element type*), with duplicates permitted. The element type can be any other supported SQL data type. For example, `INTEGER MULTISET` denotes the type of a multiset value whose element type is `INTEGER` and whose actual value could be (5, 30, 100, 45, -8, 9). The element type could also be another collection type, which allows for deeply nested collections; though those are an advanced feature as far as the standard is concerned. A multiset is an unbounded collection, with no declared maximum cardinality; unlike arrays which have either a user-specified or, in absence of such, an implementation-defined maximum cardinality. This does not mean, however, that the user can insert elements into a multiset without limit, just that the standard does not mandate that there should be a limit. This is analogous to tables, which have no declared maximum number of rows. Values of a `MULTISET` type can be created either by enumerating the individual elements or by supplying the elements through a query expression; *e.g.*, `MULTISET[1, 2, 3, 4]` or `MULTISET(SELECT grades FROM courses)`. The latter example shows how a table or part of a table can be converted into a multiset. Conversely, a multiset value can be used as a table reference in the `FROM` clause using the `UNNEST` operator. Example 1 shows how this would work.

Example 1: Using UNNEST to reference a multiset in the FROM clause.

```
SELECT T.A, T.A*2 AS TIMES_TWO
FROM UNNEST(MULTISET[4, 3, 2, 1])
       AS T(A)
```

returns the following:

A	TIMES_TWO
4	8
3	6
2	4
1	2

The `MULTISET` type supports operations for casting a multiset into an array or another multiset with a compatible element type, for removing duplicates from the multiset, for returning the number of elements in a given multiset, and for returning the only element of a multiset that has exactly one element. Additionally, the union, intersection, and difference of two multiset values are supported, as well as three new aggregate functions to create a multiset from the value of the argument in each row of a group (COLLECT), to create the multiset union of a multiset value in all rows of a group (FUSION), and to create the multiset intersection of a multiset value in all rows of a group (INTERSECTION).

Example 2 illustrates the three multiset aggregate functions. Predicates are supported to test two multiset values for equality, inequality, and distinctness, to test whether a given value is a member of a multiset, to test whether a given multiset is a subset of another multiset, and to test whether a multiset contains duplicates. Host language programs access multisets through locators in the same way they access arrays.

Example 2: New multiset aggregate functions.

Given the this table FRIENDS:

FRIEND	HOBBIES
'John'	MULTISET['READING', 'POP-MUSIC', 'RUNNING']
'Susan'	MULTISET['MOVIES', 'OPERA', 'READING']
'James'	MULTISET['MOVIES', 'READING']

The following query:

```
SELECT COLLECT(FRIEND) AS
         ALL_FRIENDS,
       FUSION(HOBBIES) AS
         ALL_HOBBIES,
```

```
       INTERSECTION(HOBBIES) AS
         COMMON_HOBBIES
FROM FRIENDS
```

Returns:

ALL_ FRIENDS	ALL_HOBBIES	COMMON_ HOBBIES
MULTISET ['John', 'Susan', 'James']	MULTISET ['READING', 'READING', 'READING', 'POP-MUSIC', 'RUNNING', 'OPERA', 'MOVIES', 'MOVIES']	MULTISET ['READING']

Table Functions

Table functions are new in SQL:2003, though many users might already be familiar with them, since they have been available in SQL products for a quite some time. A table function is an SQL-invoked function that returns a "table". For specification purposes in the standard, the return type is equivalent to a multiset of rows (*i.e.*, a `MULTISET` type whose element type is a `ROW` type) and not a real table, but it can be queried just like a table. Table functions are useful in their own right and consequently, the standard does not mandate support of multisets in order to support table functions. It should be obvious where table functions get their name. Additionally, the syntaxes for defining and invoking table functions reflect this by requiring the `TABLE` keyword in various places. When a table function is defined, its `RETURNS` clause specifies the keyword `TABLE` followed by a list of column name/data type pairs.

Example 3 and Example 4 show the definitions of an external and an SQL-bodied table function, respectively. External table functions allow the incorporation of data that is not stored in tables but comes from outside the database into queries against the database. Example 3 returns a set of rows representing cities and their current weather conditions. Several options can be specified for external functions that govern the expected behavior of the function.

Here, the actual function implementation (for brevity, not shown) is written in C (indicated by the `LANGUAGE C` clause) – which is important to know for the parameter passing mechanism used, the function itself does not call back to the SQL engine to execute SQL statements (indicated by `NO SQL`) – which is important to know for transaction management, different invocations with the same input values (none in this case) return different results (indicated by `NOT DETERMINISTIC`) – which is important to know for potential optimizations, and each input and output parameter has a null indicator associated with it (indicated by `PARAMETER STYLE SQL`).

Example 3: Definition of an external table function.

```
CREATE FUNCTION weather()
  RETURNS TABLE (
    CITY VARCHAR(25),
    TEMP_IN_F INTEGER,
    HUMIDITY INTEGER,
    WIND VARCHAR(5),
    FORECAST CHAR(25) )
  NOT DETERMINISTIC
  NO SQL
  LANGUAGE C
  EXTERNAL
  PARAMETER STYLE SQL;
```

SQL-bodied table functions, on the other hand, allow for so called "parameterized views" (as a reminder, regular SQL views are fixed at the time they are created). In Example 4, the only input parameter of the function — by virtue of being used in the predicate of the WHERE — clause determines the subset of rows that is returned; *i.e.*, for different input values (department numbers) the function returns different sets of rows (the employees in the department identified by the department number).

Example 4: Definition of an SQL-bodied table function.

```
CREATE FUNCTION DEPTEMPS
                (DEPTNO CHAR(3))
  RETURNS TABLE (
    EMPNO CHAR(6),
    LNAME VARCHAR(15),
    FNAME VARCHAR(12))
  LANGUAGE SQL
  READS SQL DATA
  DETERMINISTIC
  RETURN TABLE(
    SELECT EMPNO, LASTNAME, FIRSTNME
    FROM EMPLOYEE
    WHERE EMPLOYEE.WORKDEPT =
          DEPTEMPS.DEPTNO)
```

Again, options can be specified that govern the behavior of the function. `LANGUAGE SQL` indicates that the body of the function is written in SQL (in this example it consists of only one statement, the `RETURN` statement). `READS SQL DATA` indicates that it accesses data stored in the database in a read-only fashion. `DETERMINISTIC` indicates that the

function returns the same result given the same input values and the same database state.

When a table function is invoked in the FROM clause, it is preceded by the keyword TABLE, either in addition to other table references or as the only table reference, as can be seen in Example 5.

Example 5: Invocation of a table function.

```
SELECT W.CITY, W.TEMP_IN_F,
       W.FORECAST
FROM TABLE(weather()) AS W
 WHERE W.TEMP_IN_F > 65
```

The SQL standard specifies the exact rules of how an external table function is invoked by the SQL engine. Simplified, it consists of three phases. In the first phase, the function is invoked once with a special value for one of the parameters that indicates that the function is invoked for the first time (the "open call"). This allows the function to set up any data structures it needs in subsequent invocations. The second phase consists of as many invocations (each one a "fetch call") as are needed to transmit all rows to the SQL engine. One row is transmitted per invocation.

When there are no more rows to supply, the external function indicates this using a special value for one of the return parameters. The third and last phase consists again of a single invocation of the function (the "close call") with a special value for one of the parameters indicating to the function that the SQL engine acknowledges that there are no more rows to fetch and allowing the function to dispose of all resources it may have allocated to satisfy the previous requests.

Other Enhancements to SQL-Invoked Functions

Besides the major new functionality of table functions, several "smaller" enhancements for SQL-invoked routines were introduced. It is now possible to successfully invoke an SQL-invoked function with an untyped dynamic parameter marker (commonly known as a "?") as long as exactly one executable function can be determined using the standard subject routine determination algorithm. It is also possible to execute SQL-schema and SQL-dynamic statements within external as well as SQL-bodied routines. Besides being executed with the privileges of its definer, a routine can now alternatively be defined to be executed with the privileges of the invoker. Furthermore, it is possible to qualify a parameter name inside the routine body with the name of the routine to avoid a potential naming conflict (`DEPTEMPS.DEPTNO` in Example 4).

CREATE TABLE LIKE Extensions

Without exaggeration, the most important DDL statement is the CREATE TABLE statement that lets the users define the tables to store their data. Simplified, tables consist of columns, which have a name and an associated SQL data type (*e.g.*, INTEGER, VARCHAR, DATE, BLOB, or any of the new data types mentioned earlier). Constraints within a table (such as unique and/or primary keys, and check constraints) and relationships between tables (using foreign keys) can be expressed. Default values can be assigned to columns that are used if the user does not explicitly specify a value for the column in, for example, an INSERT statement.

New in SQL:2003 are the possibility to declare a column as an identity column or as a generated column (described later in this article). In either case, the value for such a column is automatically generated and supplied whenever a new row is inserted into the table. New are also two enhancements to the CREATE TABLE statement that are particularly valuable for database designers/administrators when they design or prototype new tables that are based on or similar in structure to existing tables.

Already in SQL:1999, a user could specify that a new table should look *like* one or more existing tables. This was done by a special clause (the LIKE clause) that allows copying the structure of one or more existing tables into the new table in addition to zero or more new columns. However, the copying was restricted to the column names and data types of the existing table(s). Example 6 shows what was possible in SQL:1999. Given table T1 as defined in the first `CREATE TABLE` statement, the definition of table T2 is equivalent to the definition of table T3. Since several tens or even hundreds of columns in tables in real-world applications are not the exception but rather the norm, it can be easily seen what kind of benefits this functionality can bring in the development and testing process.

Example 6: Existing CREATE TABLE LIKE functionality

```
CREATE TABLE T1 (
  C1 INTEGER GENERATED ALWAYS
    AS IDENTITY (START WITH 1,
                 INCREMENT BY 2),
  C2 VARCHAR(100) NOT NULL
                  DEFAULT 'test',
  C3 CHAR(30));

CREATE TABLE T2 (
  LIKE T1,
```

```
  C4 CHAR(50));

CREATE TABLE T3 (
  C1 INTEGER,
  C2 VARCHAR(100),
  C3 CHAR(30),
  C4 CHAR(50));
```

As can also be seen in Example 6, the structure of table T1 contains more information than table T2; *e.g.*, the information that C1 was an identity column is lost. Thus, in SQL:2003 additional (optional) options for the `LIKE` clause were introduced that allow for copying more information (such as identity column options, the expressions used for generated columns, and the default values). Example 7 shows the new options for copying the identity information and default values. Table T4 has now exactly the same structure as table T1. If no options are specified, then the same semantics as in SQL:1999 apply with the exception that if the column that a new column is based on is known to be not null (i.e., it was declared with `NOT NULL`), then the new column is known to be not null too.

Example 7: New options of the CREATE TABLE LIKE.

```
CREATE TABLE T4 (
  LIKE T1
    INCLUDING COLUMN DEFAULTS
    INCLUDING IDENTITY);
```

It is worthwhile pointing out that the CREATE TABLE LIKE statement does not create a dependency between the new table and the table(s) used in the LIKE clause(s).

CREATE TABLE AS Extensions

The CREATE TABLE LIKE extensions presented above are useful if the user wants to create a new table that can copy the complete structure of one or more existing tables. There are, however, also circumstances when it would be useful to copy only a subset of the structures of one or more existing tables or more generally any query expression. For these cases the `CREATE TABLE` statement has an extension (commonly referred to as `CREATE TABLE AS`) that allows the creation of a table with the structure (*i.e.*, the column names, data types, and nullability characteristics) of a query expression. This extension also allows to rename the columns, if needed, and to insert the rows that this query expression yields into the newly created table. Example 8 shows how to create a new table and populate it in the same step.

Example 8: CREATE TABLE AS

```
CREATE TABLE T5 (D1, D2, D3, D4) AS
  (SELECT T1.C1, T1.C2, T2.C3, T2.C4
   FROM T1, T2
   WHERE T1.C2 = T2.C2) WITH DATA
```

While the syntax of the `CREATE TABLE AS` statement is similar to what is known as *Materialized Query Tables (MQTs)* or *Materialized Views* in many commercial products, the semantics are different. The standard's `CREATE TABLE AS` statement does not create a dependency of the new table on the underlying query expression and, after the table is initially populated, updates to the tables in the query expression will not automatically be reflected in the new table. Because SQL products form a tight linkage between an MQT and its underlying query expression , the use of MQTs allows optimization when a user's query can be (at least partly) answered by using a populated MQT rather than by re-evaluating the complete query expression (which, in general, is not as trivial as in the example above).

MERGE statement

Prior to SQL:2003, SQL provided three statements for updating the database, popularly known as `INSERT`, `UPDATE`, and `DELETE` statements. An `INSERT` statement adds one or more new rows to an existing table or view. An `UPDATE` statement replaces values of one or more columns of one or more rows in an existing table or view. A `DELETE` statement removes one or more rows from an existing table or view. SQL:2003 adds a fourth statement, `MERGE`.

A frequent requirement that arises in database applications is to be able to transfer a set of rows from a "transaction table" (for example a shipment table or trades table) to a master table maintained by the database (for example a parts table, or accounts table). Typically, the transaction table contains updates to the existing rows in the master table and/or new rows that should be inserted into the master table. Contents of the transaction table can be transferred to the master table in two separate steps, executing an `UPDATE` statement for those rows that have a matching counterpart in the master table and an `INSERT` statement for those rows that do not have a matching counterpart in the master table. The `MERGE` statement introduced in SQL:2003 effectively combines the two steps into a one-step process, making it more efficient, as well as easier for the user to specify.

For example, assume the INVENTORY table lists the master list of all parts in a company and the

SHIPMENT table lists parts that were received on any given day. Further assume that both tables have a column PARTNUM that acts as the primary key. For illustration purpose, assume that contents of INVENTORY and SHIPMENT tables are as shown in Table 1 and Table 2, respectively.

Table 1 — INVENTORY table

PARTNUM	DESCRIPTION	QUANTITY
1	Cool Part	10
2	Another Cool Part	15
3	Really Cool Part	20

Table 2 — SHIPMENT table

PARTNUM	DESCRIPTION	QUANTITY
2	Another Cool Part	5
4	Yet Another Cool Part	15
1	Cool Part	10

Note that the first and third rows in SHIPMENT table each have a counterpart in INVENTORY table while the second row in SHIPMENT table does not. Here is how the new MERGE statement would be used to update the matching rows in INVENTORY table at the same time as adding the new row:

```
MERGE INTO INVENTORY AS INV
  USING (SELECT PARTNUM,
         DESCRIPTION,
         QUANTITY FROM SHIPMENT)
    AS SH
  ON (INV.PARTNUM = SH.PARTNUM)
WHEN MATCHED THEN UPDATE
  SET QUANTITY = INV.QUANTITY +
                 SH.QUANTITY
WHEN NOT MATCHED THEN INSERT
   (PARTNUM, DESCRIPTION, QUANTITY)
    VALUES (SH.PARTNUM,
            SH.DESCRIPTION,
            SH.QUANTITY)
```

Table 3 lists the contents of INVENTORY table after the execution of MERGE statement.

Table 3 — INVENTORY table after MERGE

PARTNUM	DESCRIPTION	QUANTITY
1	Cool Part	20
2	Another Cool Part	20
3	Really Cool Part	20
4	Yet Another Cool Part	15

Note that the SHIPMENT table does not have to have the same format as the INVENTORY table. It merely has to have one or more columns for matching purpose and whatever columns are required in the INSERT and UPDATE parts of the MERGE statement.

Sequence Generators

Consider the PARTNUM column of the SHIPMENT table used in MERGE example. Clearly, the PARTNUM value for each part must be unique since it serves as the primary key. Either some one in the company must take on the task of coming up with unique PARTNUM values or one can get SQL to generate unique values automatically. SQL:2003 provides a new feature, *sequence generators*, for this purpose.

A sequence generator is a new kind of database object with an associated time-varying exact numeric value. A sequence generator comes into existence when a CREATE SEQUENCE statement is executed. As part of the CREATE SEQUENCE statement, users can specify a minimum value, a maximum value, a start value, an increment, and a cycle option for the sequence generator they are creating.

The following example illustrates the creation of a sequence generator called PARTSEQ:

```
CREATE SEQUENCE PARTSEQ AS INTEGER
  START WITH 1
  INCREMENT BY 1
  MINVALUE 1
  MAXVALUE 10000
  NO CYCLE
```

A sequence generator has a time-varying *current base value* and a cycle that consists of all the possible values between the minimum value and the maximum value that are expressible as (current base value + *M* * increment), where *M* is a non-negative number. When created, the current base value of a sequence generator is initialized to the start value. SQL:2003 provides a new built-in function, NEXT VALUE FOR, which, when applied on a sequence generator, modifies the current base value of that sequence generator to a value *V* taken from the sequence generator's current cycle such that *V* is expressible as (current base value + *N* * increment) where *N* is a non-negative number and returns the new current base value to the invoker. For example,

repeated applications of NEXT VALUE FOR function on PARTSEQ may yield values 1, 2, 3, 4, ...

The following example illustrates how a sequence generator can be used to assign unique PARTNUM values:

```
INSERT INTO SHIPMENT (
  PARTNUM, DECRIPTION, QUANTITY )
  VALUES ( NEXT VALUE FOR PARTSEQ,
           'Display', 20);
```

While creating a sequence generator, users can specify a CYCLE or NO CYCLE for that sequence generator. If NO CYCLE is specified, then an exception is raised when an application of NEXT VALUE on a sequence generator, *SG*, attempts to return a value that does not lie in the interval bounded by the minimum and maximum value of *SG*. On the other hand, if CYCLE is specified and if an application of NEXT VALUE on a sequence generator, *SG*, attempts to return a value that does not lie in the interval bounded by the minimum and maximum value of *SG*, then the actual result value returned is the minimum value of *SG* (if the increment of *SG* is a positive number) or the maximum value for of *SG* (if the increment of *SG* is a negative number).

SQL also provides an ALTER SEQUENCE statement to alter the properties of a sequence generator and a DROP SEQUENCE statement to drop a sequence generator. Using an ALTER SEQUENCE statement, users can alter the increment, maximum value, minimum value, and the cycle option of a sequence generator by using the ALTER SEQUENCE statement. In addition, users can specify a restart value using the ALTER SEQUENCE ... RESTART WITH ... A value so specified is returned as the result of the NEXT VALUE FOR function applied immediately following the ALTER SEQUENCE operation. This option is useful whenever users want the sequence generator to generate future values starting from a specified value rather than starting from the result of previous application of the NEXT VALUE FOR function.

Identity Columns

While sequence generators put SQL in charge of generating unique values, users are still burdened with tasks such as creating a sequence generator and invoking the NEXT VALUE FOR function at appropriate times. SQL:2003 provides another new feature, identity columns, that provides a more convenient mechanism by making it unnecessary for users to perform these additional tasks.

Identity columns are columns designated with the special keyword IDENTITY, as shown below:

```
CREATE TABLE PARTS (
  PARTNUM INTEGER GENERATED ALWAYS
    AS IDENTITY (START WITH 1
                 INCREMENT BY 1
                 MINVALUE 1
                 MAXVALUE 10000
                 NO CYCLE),
  DESCRIPTION VARCHAR (100),
  QUANTITY INTEGER )
```

As can be seen from the above example, identity columns share the same attributes as sequence generators. This is because a sequence generator that inherits the identity column attributes gets associated conceptually with each identity column. Note that at most one column in a table can be designated as an identity column.

Users do not need to specify a value for an identity column whenever a new row is inserted into a table containing that identity column. The value for such a column is generated automatically by invoking the NEXT VALUE FOR function implicitly under the covers. For example, the following INSERT statement:

```
INSERT INTO PARTS
  (DESCRIPTION, QUANTITY)
VALUES ('WIDGET', 30)
```

adds a new part named WIDGET to the PARTS table. The value for the PARTNUM column is generated automatically, following the same rules that are used for generating values of sequence generators. That is, the value of PARTNUM column for the first row would be the START WITH value specified for that column, and the values for subsequent rows would follow the formula we described previously for sequence generators.

What we said above is true if the user had specified GENERATED ALWAYS for the identity column, which is the case for PARTNUM column in our example. SQL:2003 provides another option, GENERATED BY DEFAULT. If the user chooses this option, automatic generation takes place only when values are not provided in the VALUES clause. This feature is very useful for making copies of tables with identity columns.

Generated Columns

It is well known that the performance of applications, particularly in the area of data warehousing, can be improved greatly if commonly used expressions are

evaluated once and their results stored for future use. This is especially true for those applications that involve computationally expensive expressions. SQL:2003 provides a new feature known as generated columns aimed at such applications.

Users can designate one or more columns of the table as generated columns. Each generated column is associated with a scalar expression. Whenever a row is inserted into a table that contains a generated column, the expression associated with the generated column is evaluated and the resulting value is assigned as the value of that column.

For example, consider the following `CREATE TABLE` statement:

```
CREATE TABLE EMPLOYEES (
  EMP_ID INTEGER,
  SALARY DECIMAL(7,2),
  BONUS DECIMAL(7,2),
  TOTAL_COMP GENERATED ALWAYS
    AS (SALARY + BONUS) )
```

TOTAL_COMP is a generated column of the EMPLOYEES table. The data type of the TOTAL_COMP is the data type of the expression (SALARY_BONUS). Users may optionally specify a data type for a generated column, in which case the specified data type must match with the data type of the associated expression.

The following INSERT statement:

```
INSERT INTO EMPLOYEES
  (EMP_ID, SALARY, BONUS)
VALUES (501, 65000.00, 5000.00)
```

would automatically generate a value for the TOTAL_COMP column by evaluating the expression (SALARY + BONUS) and insert the row (501, 65000.00, 5000.00, 70000.00) into EMPLOYEES table.

If a generated column is included in the column list of an `INSERT` statement, the corresponding entry in the `VALUES` clause must be the keyword `DEFAULT`; it is an error to specify anything else. For example, the following statement is legal:

```
INSERT INTO EMPLOYEES
  (EMP_ID, SALARY,
   BONUS, TOTAL_COMP)
VALUES (501, 65000.00,
        5000.00, DEFAULT)
```

but the following statement is not:

```
INSERT INTO EMPLOYEES
  (EMP_ID, SALARY,
   BONUS, TOTAL_COMP)
VALUES (501, 65000.00,
        5000.00, 100000.00)
```

Whenever the value of any column referenced in the expression associated with a generated column is updated, the value of generated column is automatically reevaluated and the value of the generated column is assigned the new value. A generated column could be the target of an UPDATE statement provided the keyword DEFAULT is specified as the source; it is an error if anything else is specified as the source.

Note that all column references in an expression associated with a generated column must be to columns of the base table containing that generated column. In addition, expressions associated with generated columns are not allowed to reference other generated columns. Generated columns can be added to an existing table via the usual `ALTER TABLE ADD COLUMN` statement.

Generated columns can lead to higher performance not only because of reduced computation, but also because implementations may allow indexing on such columns. For example, if it is frequently required to present a query result in descending order of TOTAL_COMP, that result might appear on the screen much faster if an index has been created on the TOTAL_COMP column.

References

[1] *Database Languages – SQL*, ISO/IEC 9075-*:2003

[2] *Andrew Eisenberg* and *Jim Melton*: SQL/XML is Making Good Progress. In: ACM SIGMOD Record, Vol. 31, No. 2, June 2002.