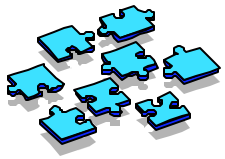
Overview of SQL:2003

Krishna Kulkarni

Silicon Valley Laboratory
IBM Corporation, San Jose
2003-11-06



Outline of the talk

- Overview of SQL-2003
- New features in SQL/Framework
- New features in SQL/Foundation
- New features in SQL/CLI
- New features in SQL/PSM
- New features in SQL/MED
- New features in SQL/OLB
- New features in SQL/Schemata
- New features in SQL/JRT
- Brief overview of SQL/XML

SQL:2003

- Replacement for the current standard, SQL:1999.
- FCD Editing completed in January 2003. New International Standard expected by December 2003.
- Bug fixes and enhancements to all 8 parts of SQL:1999.
- One new part (SQL/XML).
- No changes to conformance requirements - Products conforming to Core SQL:1999 should conform automatically to Core SQL:2003.

SQL:2003 (contd.)

- Structured as 9 parts:
  - Part 1: SQL/Framework
  - Part 2: SQL/Foundation
  - Part 3: SQL/CLI (Call-Level Interface)
  - Part 4: SQL/PSM (Persistent Stored Modules)
  - Part 9: SQL/MED (Management of External Data)
  - Part 10: SQL/OLB (Object Language Binding)
  - Part 11: SQL/Schemata
  - Part 13: SQL/JRT (Java Routines and Types)
  - Part 14: SQL/XML
- Parts 5, 6, 7, 8, and 12 do not exist

Part 1: SQL/Framework

- Structure of the standard and relationship between various parts
- Common definitions and concepts
- Conformance requirements statement
- Updates in SQL:2003/Framework reflect updates in all other parts.

Part 2: SQL/Foundation

- The largest and the most important part
- Specifies the "core" language
- SQL:2003/Foundation includes all of SQL:1999/Foundation (with lots of corrections) and plus a number of new features
  - Predefined data types
  - Type constructors
  - DDL (data definition language) for creating, altering, and dropping various persistent objects including tables, views, user-defined types, and SQL-invoked routines.
  - Scalar and table expressions
  - Predicates
  - DML (data manipulation language) for retrieving and updating persistent data
  - Host language bindings, dynamic SQL, and direct SQL

New Features in SQL/Foundation

- New data types
  - BIGINT
  - MULTISET
- Extensions to existing data types
  - Unbounded ARRAY
- Deletion of existing types
  - BIT
  - BIT VARYING
- New schema objects
  - Sequence generators

New Features in SQL/Foundation (contd.)

- Enhancements to existing schema objects
  - Identity columns for tables
  - Generated columns for tables
  - Major enhancements to CREATE TABLE LIKE
  - Base tables created from a query expression (aka Materialized tables)
  - Retrospective check constraints
  - ALTER functionaity for Transforms
  - SQL-invoked routines with Invoker's rights
  - "Table" functions
  - Dynamic and schema statements inside routine bodies

New Features in SQL/Foundation (contd.)

- New built-in scalar functions
  - LN (expr)
  - EXP (expr)
  - POWER (expr, expr)
  - SQRT (expr)
  - FLOOR (expr)
  - CEIL[ING] (expr)
  - WIDTH_BUCKET(expr, expr, expr, expr)
- New one-argument aggregate functions
  - STDDEV_POP (expr)
  - STDDEV_SAMP (expr)
  - VAR_POP (expr)
  - VAR_SAMP (expr)

New Features in SQL/Foundation (contd.)

- New two-argument aggregate functions
  - COVAR_POP (expr, expr)
  - COVAR_SAMP (expr, expr)
  - CORR (expr, expr)
  - REGR_SLOPE (expr, expr)
  - REGR_INTERCEPT (expr, expr)
  - REGR_COUNT (expr, expr)
  - REGR_R2 (expr, expr)
  - REGR_AVGX (expr, expr)
  - REGR_AVGY (expr, expr)
  - REGR_SXX (expr, expr)
  - REGR_SYY (expr, expr)
  - REGR_SXY (expr, expr)

New Features in SQL/Foundation (contd.)

- New windowed table functions
  - RANK () OVER ...
  - DENSE_RANK () OVER ...
  - PERCENT_RANK () OVER ...
  - CUME_DIST () OVER ...
  - ROW_NUMBER () OVER ...
- New inverse distribution functions ....
  - PERCENTILE_DISC (expr) WITHIN GROUP (ORDER BY <sort specification list>)
  - PERCENTILE_CONT (expr) WITHIN GROUP (ORDER BY <sort specification list>)

New Features in SQL/Foundation (contd.)

- New hypothetical aggregate functions
  - RANK (expr, expr ...) WITHIN GROUP (ORDER BY <sort specification list>)
  - DENSE_RANK (expr, expr ...) WITHIN GROUP (ORDER BY <sort specification list>)
  - PERCENT_RANK (expr, expr ...) WITHIN GROUP (ORDER BY <sort specification list>)
  - CUME_DIST (expr, expr ...) WITHIN GROUP (ORDER BY <sort specification list>)

New Features in SQL/Foundation (contd.)

- Extensions to scalar expressions
  - Extensions to CASE
  - Extensions to TREAT
- Extensions to query expressions
  - Extensions to GROUP BY clause
  - WINDOW clause
  - TABLESAMPLE clause
- Multiset expressions
- Extensions to routine invocations
  - Untyped dynamic parameter arguments in routine invocations
  - Ability to qualify parameters with routine name inside routine bodies

New Features in SQL/Foundation (contd.)

- New Predicates
  - Multiset predicates
  - NORMALIZE
- New DML statements
  - MERGE statement
  - SET COLLATION /SET NO COLLATION statement
  - Multiple column assignment
- Nested savepoints
- Improved diagnostic management
- Enhancements to PREPARE statement

Sequence Generators

- Mechanism for automatic generation of sequential values.
- Two kinds:
  - External: Explicit schema object created via CREATE statement.
  - Internal: Implicitly created when an identity column is created.
- Example:

```
CREATE SEQUENCE part_num AS INTEGER
        START WITH 1
        INCREMENT BY 1
        MAXVALUE 100000
        MINVALUE 1
        CYCLE
```

Sequence Generators (contd.)

- The data type of a sequence generator must be exact numeric with scale 0. If unspecified, an implementation-defined exact numeric type with scale 0 is implicit.
- If increment is not specified, then an increment of 1 is implicit.
- If maximum value is not specified or if NO MAXVALUE is specified, then an implementation-defined maximum value is implicit.
- If minimum value is not specified or if NO MINVALUE is specified, then an implementation-defined maximum value is implicit.

Sequence Generators (contd.)

- The maximum value must be greater than the minimum value.
- If increment value is negative, then the sequence generator is a *descending sequence generator*; otherwise, it is an *ascending sequence generator*.
- If start-value is not specified, then a start-value that is equal to the implicit or explicit minimum value is implicit for ascending sequence generators and a start-value that is equal to the implicit or explicit maximum value is implicit for descending sequence generators.

Sequence Generators (contd.)

- The start value must lie between the minimum and maximum value.
- Increment must not be 0.
- If CYCLE or NO CYCLE is not specified, then NO CYCLE is implicit.
- Every sequence generator has a “current base value”
  - Set to the start value at the time of creation.
- Generate next value of a sequence generator:

```
NEXT VALUE FOR seqname
```

(returns current base value + N * increment for some N > 0)

Sequence Generators (contd.)

- If next value > MAXVALUE (or < MINVALUE), then:
  - If NO CYCLE was specified, raise an exception.
  - Otherwise, reset to MINVALUE (or MAXVALUE) and compute new value for some N.
- Examples:

```
INSERT INTO orders (orderno, custno)
VALUES (NEXT VALUE FOR my_seq, 123456);

UPDATE orders
SET orderno = NEXT VALUE FOR my_seq
WHERE orderno = 123456;
```

Sequence Generators (contd.)

- All NEXT VALUE FOR expressions specifying the same sequence generator within a single statement evaluate to the same value for a given row.
- Changes to the current base value of a sequence generator are not under transaction control, so values returned by NEXT VALUE FOR may have gaps.
- ALTER statement can be used to alter the values of increment, maximum value, minimum value, cycle option or specify a new current base value.

```
ALTER SEQUENCE myseq
    RESTART WITH 500
    INCREMENT BY 2
```

- DROP SEQUENCE statement drops a sequence generator.

Identity columns

- At most one column of a base table can be designated as an **identity column**.
- Values for an identity column are assigned automatically every time a row is inserted into such tables.
- An internal sequence generator is assumed to be associated with every identity column (only conceptually - no need for implementations to use one).
- Same options as those for sequence generators - data type, start value, increment, maximum value, minimum value, and cycle option.

Identity Columns (contd.)

- Example:

```
CREATE TABLE employees (
        EMP_ID    INTEGER
                GENERATED ALWAYS AS IDENTITY
                START WITH 100
                INCREMENT 1
                MINVALUE 100
                NO MAXVALUE
                NO CYCLE,
        SALARY   DECIMAL(7,2),
... )
```

Generated columns

- Any number of columns of a base table can be designated as **generated columns**.
- Each generated column must be associated with a scalar expression. All column references in such expressions must be to columns of the base table containing that generated column.
- If a type is specified for a generated column, it must correspond to the type of expression. If unspecified, column type is the type of expression.
- Values for generated columns are computed and assigned automatically every time a row is inserted into such tables.

Generated columns (contd.)

- Example:

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

Tables created from a query

- The definition and/or content of a base table can be generated from a query expression.
- "Materialized view" - refresh mechanism not standardized yet.
- Example:

```
CREATE TABLE T1 AS
      (SELECT (C1+1) AS X, (C2+1) AS Y
        FROM T
        WHERE C1 = 1)
      WITH DATA
```

- If WITH NO DATA is specified, an empty table gets created.

Table functions

- A SQL-invoked function whose return type is MULTISET (ROW (...)).
- Example of an external table function:

```
CREATE FUNCTION DOCMATCH
(VARCHAR(30),VARCHAR(255))
        RETURNS TABLE(DOC_ID CHAR(16))
        LANGUAGE C
        NO SQL
        DETERMINISTIC
        EXTERNAL
        PARAMETER STYLE SQL
```

Table functions (contd.)

- Example of a SQL table function:

```
CREATE FUNCTION DEPTEMPLOYEES
                                    (DEPTNO CHAR(3))
    RETURNS TABLE ( EMPNO CHAR(6),
                                    NAME VARCHAR(40))
    READS SQL DATA
    DETERMINISTIC
    RETURN TABLE(SELECT EMPNO, NAME
                                FROM EMPLOYEE
                                WHERE
EMPLOYEE.WORKDEPT =
                                DEPTEMPLOYEES.DEPTNO)
```

Table functions (contd.)

- Table functions can appear in FROM clauses and other places where table references are allowed.
- Example:

```
SELECT T.AUTHOR, T.DOCTEXT
FROM DOCS AS T,
        TABLE(DOCMATCH('MATHEMATICS', 'ZORN''S
                LEMMA')) AS F(DOC_ID)
WHERE T.DOCID = F.DOC_ID
```

BIGINT type

- Exact numeric, scale 0
- Implementation-defined precision, but the precision of BIGINT >= precision of INTEGER >= precision of SMALLINT
- Must have same radix as SMALLINT and INTEGER
- All operations defined for values of INTEGER or SMALLINT are also applicable for values of BIGINT.

Multiset type constructor

- SQL:1999 offered only one collection type constructor: ARRAY
- SQL:200x offers another collection type constructor: MULTISET
- Varying- length, unordered collections of element having specified type
- No syntax to specify maximum cardinality
- Example:

```
CREATE TABLE (
    C1  INTEGER,
    C2  INTEGER MULTISET,
    C3 ROW ( F1 INT, F2 INT ) MULTISET)
```

Multiset value constructor

- Constructing an empty multiset
  - MULTISET ( )
  - Element type determined by the context
- Multiset constructed by explicit enumeration of elements:
  - MULTISET(2, 3, 5, 7)
- Multiset constructed from the result of a query:
  - MULTISET(SELECT col1
    FROM tbl1
    WHERE col2 > 10)

Multiset operations

- CARDINALITY (*value1*)
  - Type of *value1* must be multiset
  - Returns number of elements in *value1*
- SET (*value1*)
  - Type of value1 must be multiset
  - Returns *value1* with duplicate elements removed
- ELEMENT (*value1*)
  - Type of *value1* must be multiset
  - Cardinality of *value1* must be 1
  - Returns the single element in *value1*

Multiset operations (contd.)

- UNNEST(*value1*) AS *corr- name*
  - Type of *value1* must be multiset
  - " Unnests" *value1* and turns the elements into rows of a virtual table
- Example:

```
SELECT SUM (t.c)
FROM UNNEST (MULTISET (2, 3, 5, 7)) AS t(c)
```

produces the following result:

17

Multiset operations (contd.)

- *value1* MULTISET *setop quantifier value2*
  - Type of *value1* and *value2* must be multiset
  - *setop* — UNION or EXCEPT or INTERSECT
  - *quantifier* — ALL or DISTINCT
  - ALL is the default quantifier
- Example:

```
SELECT col1
      MULTISET INTERSECT DISTINCT col2
FROM tbl1
WHERE CARDINALITY( col2) > 50
```

- Close analogs to ordinary set operators UNION , EXCEPT, and INTERSECT

Multiset operations (contd.)

- One new aggregate function that produces a value of multiset type:
  - COLLECT
    - Transform the values in a group into a multiset
- Two new aggregate functions on columns of multiset type:
  - FUSION
    - Form a union of the multisets in a group
    - number of duplicates of a given value in the result is the sum of the number of duplicates in the multisets in the rows of the group
  - INTERSECTION
    - Form an intersection of the multisets in a group
    - number of duplicates of a given value in the result is the minimum of the number of duplicates in the multisets in the rows of the group

Multiset operations (contd.)

- = and <> are the only comparison operations allowed on multisets.
- Three new predicates defined for values of multisets.
- MEMBER predicate:
  value1 [NOT] MEMBER [OF] *value2*
  - *value2* should be a multiset and *value1* should be comparable to the element type of *value2*.
  - If *value1* is equal to some element of *value2*, returns *true*
  - If *value2* is empty, returns *false*
  - If some element of *value2* is null, returns unknown.

Multiset operations (contd.)

- SUBMULTISET predicate:

  value1 [NOT] SUBMULTISET [OF] *value2*

  - Both *value1* and *value2* should be multisets and their element types must be comparable
  - If *value1* is empty or if every value in *value1* has a corresponding value in *value2*, then returns true

- SET predicate:

  value1 IS [NOT] A SET

  - *value1* must be a multiset
  - If there are no duplicate values in *value1*, returns *true*
  - Maximum of 1 null value in *value1*

Windowed Table functions

- Windowed table functions provide facilities for calculating ranks, moving sums, moving averages, etc.
- A **windowed table function** operates on a window of a table and returns a value for every row in that window. The value is calculated by taking into consideration values from the set of rows in that window.
- 5 new windowed table functions
  - RANK () OVER ...
  - DENSE_RANK () OVER ...
  - PERCENT_RANK () OVER ...
  - CUME_DIST () OVER ...
  - ROW_NUMBER () OVER ...

Rank, Dense_Rank, and Rownumber

- RANK
  - Returns the relative position within an ordered list.
  - Requires ordering.
  - Ties have the same rank.
- DENSE_RANK
  - Like RANK, but no gaps in rankings in the case of ties.
- ROW_NUMBER
  - Ties are nondeterministically numbered.
  - If no ordering is specified, each row is nondeterministically numbered.

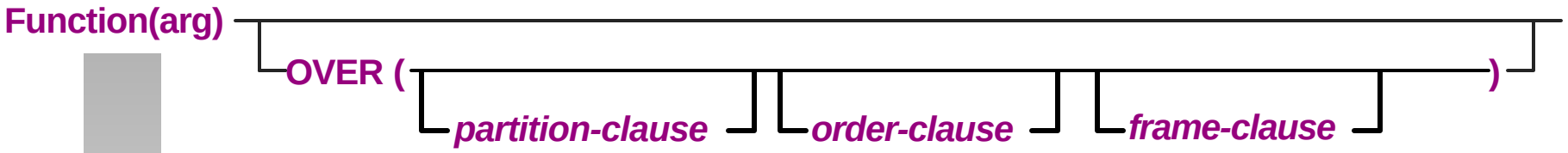
Rank, Dense_Rank, and Rownumber

- Rank the employees in descending order of their salary.

```
SELECT name, salary,
rank() over (ORDER BY salary DESC) as rank,
denserank() over (ORDER BY salary DESC) as dense_rank,
rownumber() over (ORDER BY salary DESC) as rownum
FROM emp;
```

name	salary	rank	dense_rank	rownum
Chris	8000	1	1	1
Sally	7900	2	2	2
Jim	7900	2	2	3
Frank	6600	4	3	4
Lynn	6600	4	3	5
Sara	6000	6	4	6

Windowed Table Functions - Syntax



- Windows are defined using the ***window-clause.***
- Window clause consists of
  - A partition clause - The partition clause allows the rows in a table to be grouped into partitions.
  - An order clause - The order clause allows the rows in a partition to be ordered.
  - A frame clause - The frame clause allows the specification of the range of rows relative to the current row that would participate in the function evaluation.

Aggregate Functions as Windowed Table Functions

- Existing aggregate functions can also be used as windowed table functions:
  - SUM (...) OVER ...
  - AVG (...) OVER ...
  - MAX (...) OVER ...
  - MIN (...) OVER ...
  - COUNT (...) OVER ...
  - EVERY (...) OVER ...
  - ANY (...) OVER ...
  - SOME (...) OVER ...
- Allows calculation of moving and cumulative aggregate values.

Hypothetical Aggregate Functions

- Hypothetical aggregate functions evaluate the aggregate over the window extended with a new row derived from the specified values.
- 4 new hypothetical aggregate functions
  - RANK (expr, expr ...) WITHIN GROUP (ORDER BY <sort specification list>)
  - DENSE_RANK (expr, expr ...) WITHIN GROUP (ORDER BY <sort specification list>)
  - PERCENT_RANK (expr, expr ...) WITHIN GROUP (ORDER BY <sort specification list>)
  - CUME_DIST (expr, expr ...) WITHIN GROUP (ORDER BY <sort specification list>)

Inverse Distribution Functions

- 2 new inverse distribution functions ....
  - PERCENTILE_DISC (expr) WITHIN GROUP (ORDER BY <sort specification list>)
  - PERCENTILE_CONT (expr) WITHIN GROUP (ORDER BY <sort specification list>)
- Argument must evaluate to a value between 0 and 1.
- Return the values of expressions specified in <sort specification list> that correspond to the specified percentile value.

Queries over Sampled data

- Permits evaluation of aggregates on samples derived from stored data.
- Better performance when database is huge and absolutely exact results are not necessary.
- Two forms of sampling:
  - BERNOULLI: the probability of a given row of the original table appearing in the sample table is independent of every other row.
  - SYSTEM: the probability of a given row of the original table appearing in the sample table may depend on rows alreday included in the sample.

Queries over Sampled data (contd.)

- Example: Retrieve an approximate estimate of the total salary of employees in each department:

```
SELECT dept, SUM(salary) * 10
  FROM employee TABLESAMPLE BERNOULLI (10)
  REPEATABLE (5)
  GROUP BY dept
```

- REPEATABLE clause serves the same purpose as a random number seed for sampling - If specified, repeated executions return identical result table for the same argument, provided certain implementation-defined conditions are satisfied.

MERGE statement

- Combination of insert and update operations in a single statement.
- Rows in the input table are divided into two groups based on a predicate: *insert source table* if the predicate evaluates to false or unknown and *update source table* if the predicate evaluates to true.
- Insert source table is inserted into the target table.
- Every row in target table that has a matching row in update source table is updated. It is an error if a given row in target table matches with more than one row in update source table.

MERGE statement (contd.)

- Example:

```
MERGE INTO inventory AS in
USING (SELECT partno, description, count
          FROM shipment
           WHERE shipment.partno IS NOT NULL) AS sh
ON (in.partno = sh.partno)
WHEN MATCHED THEN
      UPDATE SET description = sh.description,
                          quantity = in.quantity + sh.count
WHEN NOT MATCHED THEN
      INSERT (partno, description, quantity)
            VALUES (sh.partno, sh.description, sh.count)
```

MERGE statement (contd.)

- MATCHED and NOT MATCHED clauses permitted once each, in any order.
- The order of inserts and updates depends on the order of MATCHED and NOT MATCHED clauses.
- No MERGE triggers - UPDATE/INSERT triggers fire as necessary.

Multiple assignment

- In SQL:1999's UPDATE statements, a value can be assigned to a single target only.
- In SQL:2003's UPDATE and MERGE statements, multiple targets can be updated by assigning a value from a list of values to the corresponding target in a list of targets.
- All values are evaluated before any assignment is executed.

Part 3: SQL/CLI

- A Call-Level Interface for dynamically invoking SQL from application programs
- Consists of over 60 routine specifications
  - Control connections to SQL-servers
  - Allocate and deallocate resources
  - Execute SQL statements
  - Control transaction termination
  - Obtain information about the implementation
- Provided for vendors of truly portable "shrink wrapped" software
  - CLI does not require pre-compilation of the application programs
  - Application programs can be delivered in "shrink wrapped", object-code form

New Features in SQL/CLI

- No new functionality in SQL:2003, mainly bug fixes

Part 4: SQL/PSM

- Procedural language constructs similar to those found in block-structured languages
  - **Improve performance in centralized and client/server environments**
    - Multiple SQL statements in a single EXEC SQL
    - Multi-statement procedures, functions, and methods
  - **Gives great power to DBMS**
    - Several, new control statements (procedural language extension) (begin/end block, assignment, call, case, if, loop, for, singal/resignal, variables, exception handling)
  - SQL-only implementation of complex functions
    - Without worrying about security ("firewall")
    - Without worrying about performance ("local call")
  - SQL-only implementation of class libraries

New Features in SQL/PSM

- Extensions to CASE statement
- Ability to qualify variables with beginning labels

Part 9: SQL/MED

- MED - Management of External Data
- Two distinct techniques:
  - Datalinks
  - Wrapper Interface
- Datalinks
  - External data coordinated by SQL-server
  - External data accessed through native interface
  - External data kept consistent with SQL-data
- Wrapper interface
  - External data not controlled by SQL-server
  - External data accessed through SQL-server
  - Data access from multiple sources transparently

New Features in SQL/MED

- Enhancements to Datalinks
  - Update-in-place - Linked file can be updated while it is linked provided the datalink column was defined with WRITE PERMISSION ADMIN ...

New Features in SQL/MED (contd.)

- Enhancements to the Wrapper interface
  - The ability for an SQL-server to communicate complex requests to foreign-data wrappers.
    - Requests containing WHERE clauses
    - Requests containing multiple table references in FROM clauses
    - Requests containing complex value expressions in SELECT and WHERE clauses
    - Requests containing user-defined functioninvocations in value expressions
  - The ability for an SQL-server to communicate the query context (that is, information to identify requests belonging to the same query) to foreign-data wrappers.
  - The ability for an SQL-server to ask foreign-data wrappers for request execution "costs".

New Features in SQL/MED (contd.)

- In SQL:1999, all foreign server requests were required to be of the form:

```
SELECT *
    FROM foreign_table
```

- In SQL:2003, foreign server requests could be of the form:

```
SELECT c1+1, c2 || c3, fun1 (c4)
    FROM foreign_table1, foreign_table2
    WHERE c4 = 1 AND c5 = 2 AND c6 = ?
```

Part 10: SQL/OLB

- OLB - Object Language Bindings
- Embedding of SQL statements in Java programs
- Many differences from the traditional host language bindings:
  - specification in terms of JDBC, but static compilation
  - provides typed cursors and better exception handling
  - platform independence (binary portability)
- Close relationship between SQL/OLB and JDBC
- SQL/OLB advantages over JDBC:
  - SQL/OLB provides higher level of abstraction than JDBC
  - SQL/OLB uses static precompiled SQL while JDBC uses dynamic SQL.

New Features in SQL/OLB

- Reference to JDBC revised to JDBC 3.0
- Support for ARRAY and DATALINK types
- Support for Savepoints
- Support for multiple result sets returned from an SQL-invoked procedure

Part 11: SQL/Schemata

- Specification of over 85 views that describe the metadata
  - TABLES view
  - COLUMNS view
  - USER_DEFINED_TYPES view, etc.
- These views exist in a special schema named INFORMATION_SCHEMA
- Can be queried by users - users see only those objects that are either owned or have some privilege on
- Cannot be updated by users
- Underlying base tables created in a hypothetical "Definition schema" - auomatically updated everytime a DDL statement executes

New Features in SQL/Schemata

- New views for SQL:2003 schema objects
  - SEQUENCES
  - ROUTINE_SEQUENCE_USAGE
  - TRIGGER_SEQUENCE_USAGE
- Additional views for SQL:1999 schema objects
  - COLUMN_COLUMN_USAGE
  - CHECK_CONSTRAINT_ROUTINE_USAGE
  - ROUTINE_ROUTINE_USAGE
  - TRIGGER_ROUTINE_USAGE
  - VIEW_ROUTINE_USAGE
  - COLLATION_CHARACTER_SET_APPLICABILITY
  - SQL_PARTS
- Additional columns for SQL:1999 views

Part 13: SQL/JRT

- SQL Routines and types using the Java™ programming language
- Extension to <SQL-invoked routine> syntax to create SQL-invoked routines based on Java static methods.
- Extension to <user-defined type definition> syntax to create SQL user-defined types based on Java classes.

New Features in SQL/JRT

- Support for table functions
- Support for DATALINK type as a parameter or result type.

Part 14: SQL/XML

- New part in SQL:2003
- A new built-in type, XML.
- 4 built-in operators:
  - XMLPARSE: returns a value of XML type given an SQL character string expression
  - XMLSERIALIZE: returns a value of character string type given an XML expression
  - XMLROOT: modifies the root information item of an XML value and returns the modified value.
  - XMLCONCAT: concats two or more XML values and returns the resulting value.
- A predicate, IS DOCUMENT, to test whether an XML value has a single root element.

Part 14: SQL/XML (contd.)

- 5 "publishing functions" that generate values of XML type from SQL expressions:
  - XMLELEMENT
  - XMLFOREST
  - XMLATTRIBUTE
  - XMLNAMESPACES
  - XMLAGG
- Host language bindings for values of XML type.

Part 14: SQL/XML (contd.)

- Rules for mapping SQL "things" to XML "things"
  - Rules for mapping SQL identifiers to XML names
  - Rules for mapping XML names to SQL identifiers
  - Rules for mapping SQL types to XML Schema types
  - Rules for mapping SQL values to XML values
  - Rules for mapping SQL tables, schemas, and catalogs to XML values
- Conformance requirements

XML type

- A new SQL built-in type
  - Can be used wherever a SQL data type is allowed - as the type of a column of a table, parameter of a routine, attribute of an UDT, field of a row, or a SQL variable.
  - Strongly-typed - values of XML type are distinct from their textual representation.
  - Semantics of operations on values of XML type is specified by assuming a tree-based internal representation based on the XML Information Set Recommendation (Infoset).
  - The Infoset model is modified in one significant way: the document information item of Infoset is replaced by a new kind of information item, XML root information item.

XML type (contd.)

- An *SQL/XML information item* is either an
  - XML root information item (as defined by SQL/XML)
  - XML attribute information item (as defined by Infoset)
  - XML character information item (as defined by Infoset)
  - XML comment information item (as defined by Infoset)
  - XML document type declaration information item (as defined by Infoset)
  - XML element information item (as defined by Infoset)
  - XML namespace information item (as defined by Infoset)
  - XML notation information item (as defined by Infoset)
  - XML processing instruction information item (as defined by Infoset)
  - XML unexpanded entity reference information item (as defined by Infoset)
  - XML unparsed entity information item (as defined by Infoset)

XML type (contd.)

- XML root information item is similar to document information item of Infoset with one difference.
- While document information item allows exactly one child element information item, XML root information item allows zero or more child element information items.
- An XML value is either
  - the null value, or
  - a collection of SQL/XML information items that consists of exactly one XML root information item and every SQL/XML information item that can be reached recursively by traversing the properties of the SQL/XML information items.

XMLPARSE

- XMLPARSE produces an XML value given an SQL string expression:

  XMLPARSE ({ DOCUMENT | CONTENT }
  <string value expression>
  [{ PRESERVE | STRIP } WHITESPACE])

- If DOCUMENT is specified, <string value expression> must evaluate to a character string that conforms to XML 1.0 as modified by the Namespaces recommendation; otherwise an exception is raised.

XMLPARSE (contd.)

- If CONTENT is specified, <string value expression> must evaluate to a character string that conforms to a grammar obtained from the grammar of XML 1.0 as modified by the Namespaces recommendation with a new "start symbol", defined as follows:

  XMLvalue ::= XMLDecl? content

  Otherwise, an exception is raised.

- If the argument value contains in-line DTDs, XMLPARSE will:
  - replace all entity references defined in an in-line DTD by their expanded form,
  - apply the default values defined by an internal DTD.

XMLPARSE (contd.)

- Examples:

```
INSERT INTO employees ( id, xvalue)
VALUES (1001,
XMLPARSE (DOCUMENT '<Emp> John Smith
</Emp>'))

INSERT INTO employees ( id, xvalue)
VALUES (1001,
XMLPARSE (CONTENT 'John Smith'))

INSERT INTO employees ( id, xvalue)
VALUES (1001,
XMLPARSE (CONTENT '<First> John </First> <Last>
Smith </Last>'))
```

XMLPARSE (contd.)

- Whitespace handling
  - If an element specification includes the attribute

    xml:space='preserve'

    then, XMLPARSE preserves all whitespace contained in the element.
  - If an element specification includes the attribute

    xml:space='default'

    or has no such attribute, the behavior of XMLPARSE is governed by whether the user has specified PRESERVE WHITESPACE or STRIP WHITESPACE.
  - If the user has not specified either of the two options, STRIP WHITESPACE is implicit.

XMLPARSE (contd.)

- If PRESERVE WHITESPACE is specified, then XMLPARSE preserves all whitespace contained in the argument string.
- If STRIP WHITESPACE is specified or implied, then XMLPARSE will drop all whitespace between two adjacent element tags unless there is at least one non-whitespace character.

XMLPARSE (contd.)

- In the following example,

  INSERT INTO employees (id, xvalue)

  VALUES (1001,

  XMLPARSE (CONTENT :hv STRIP WHITESPACE))

  assume :hv contains the following value:

```
<?xml          standalone='1.0'       ?>

<well/>

Hello

<a attr='  '>
  <b>       </b>
  Dolly
</a>
```

XMLPARSE (contd.)

- XMLPARSE will treat the value in :hv as equivalent to the following value

```
<?xml standalone='1.0'?>
<well/>

Hello

<a attr='  '><b></b>
  Dolly
</a>
```

XMLSERIALIZE

- XMLSERIALIZE produces an SQL string value given an XML value expression:
  XMLSERIALIZE ({ DOCUMENT | CONTENT }
  <XML value expression> AS <data type>
- <data type> must be a character string type (CHAR, VARCHAR or CLOB).
- If DOCUMENT is specified, <XML value expression> must evaluate to an XML value that has exactly one top-level element; otherwise an exception is raised.
- SQL/XML does not mandate the exact content of the resulting string, but the resulting string, passed as an argument to XMLPARSE using the specified DOCUMENT or CONTENT option and PRESERVE WHITESPACE option, must yield the same XML value produced by <XML value expression>.

XMLSERIALIZE (contd.)

- Example:

```
XMLSERIALIZE( DOCUMENT
  XMLPARSE (DOCUMENT '<Emp> John Smith
</Emp>')
  AS VARCHAR(100))
```

may produce the character string

```
<Emp> John Smith </Emp>
```

or

```
<?xml encoding="UTF-8" version="1.0"?> <Emp> John
Smith </Emp>
```

or

```
<?xml encoding="UTF-8" version="1.0"?>
      <Emp> John Smith </Emp>
```

XMLROOT

- XMLROOT produces an XML value with specified version and standalone properties given an expression of XML type:

  XMLROOT (<XML value expression> ,
  VERSION {<string value expression>| NO VALUE }
  [, STANDALONE { YES | NO | NO VALUE }]

- Example:

  INSERT INTO employees (id, xvalue)
  VALUES (1001,
  XMLROOT (XMLPARSE (DOCUMENT '<Emp> John Smith </Emp>'), VERSION '1.0', STANDALONE YES)

IS DOCUMENT

- IS DOCUMENT predicate checks whether the specified XML value is an XML document, i.e., whether it has exactly one top-level element:

  <XML value expression> IS [NOT] DOCUMENT

- Example:

```
SELECT XMLSERIALIZE (DOCUMENT xvalue AS
CLOB)
FROM employees
WHERE xvalue IS DOCUMENT;
```

XMLELEMENT

- XMLELEMENT produces an XML value given
  - an SQL identifier that acts as the name of an element
  - an optional namespace declaration
  - an optional list of named expressions that provides names and values of its attributes, and
  - an optional list of expressions that provides the element content.

```
SELECT e.id,
            XMLELEMENT (NAME "Emp",
                        e.fname || ' ' || e.lname) AS xvalue
FROM employees e WHERE ...

==>
ID              XVALUE
1001      <Emp> John Smith </Emp>
...
```

XMLELEMENT (contd.)

- Attribute specifications must be bracketed by XMLATTRIBUTES keyword and must appear as the third argument if namespace declaration is specified, second argument otherwise.
- Each attribute can be named implicitly or explicitly.

```
SELECT e.id,
            XMLELEMENT (NAME "Emp",
                XMLATTRIBUTES (e.id AS "empid"),
                              e.fname || ' ' || e.lname)
                              ) AS xvalue
FROM employees e WHERE ...

==>
ID                  XVALUE
1001      <Emp empid="1001"> John Smith </Emp>
...
```

XMLELEMENT (contd.)

- XMLELEMENT can produce nested element structure.

```
SELECT e.id,
            XMLELEMENT (NAME "Emp",
                XMLELEMENT (NAME "name",
                            e.fname || ' ' || e.lname ),
                XMLELEMENT (NAME "hiredate", e.hire)
                            ) AS xvalue
FROM employees e WHERE ...

==>
ID                  XVALUE
1001      <Emp>
                <name> John Smith </name>
                <hiredate> 2000-05-24 </hiredate>
            </Emp>

...
```

XMLELEMENT (contd.)

- XMLELEMENT can produce mixed content.

```
SELECT e.id,
            XMLELEMENT (NAME "Emp", 'Employee '
                XMLELEMENT (NAME "name",
                e.fname || ' ' || e.lname ), 'was hired on '
                XMLELEMENT (NAME "hiredate", e.hire)
                                ) AS xvalue
FROM employees e WHERE ...

==>
ID                  XVALUE
1001        <Emp>
                Employee <name> John Smith </name>
                was hired on <hiredate> 2000-05-24 </hiredate>
            </Emp>
...
```

XMLELEMENT (contd.)

- XMLELEMENT can take scalar subqueries as arguments.

```
SELECT e.id,
  XMLELEMENT (NAME "Emp",
      XMLELEMENT (NAME "name", e.fname || ' ' || e.lname ),
      XMLELEMENT (NAME "dependents",
          (SELECT COUNT (*) FROM dependents d
              WHERE d.parent = e.id ) ) AS xvalue
FROM employees e WHERE ...

==>
ID                  XVALUE
1001      <Emp>
              <name> John Smith </name>
              <dependents> 3 </dependents>
          </Emp>
...
```

XMLELEMENT (contd.)

- Namespace declarations can be specified in XMLELEMENT; must appear before attribute specifications.
- Lexical scoping rules apply for nested elements.
- Syntax of namespace declarations:

<XML namespace declaration> ::=
    XMLNAMESPACES (<XML namespace> [, ..])
<XML namespace> ::=
    <URI> AS <prefix>
    | DEFAULT <URI>
    | NO DEFAULT
<URI> ::= <character string literal>
<prefix> ::= <identifier>

XMLELEMENT (contd.)

- Example with namespace declaration:

```
SELECT e.id,
            XMLELEMENT (NAME "IBM:Emp",
                XMLNAMESPACES ('http://a.b.c' AS IBM,
                XMLATTRIBUTES (e.id AS "empid"),
                                e.fname || ' ' || e.lname)
                                ) AS xvalue
FROM employees e WHERE ...

==>
ID                  XVALUE
1001      <IBM:Emp xmlns:IBM="http://a.b.c" empid="1001">
                                John Smith </IBM:Emp>

...
```

XMLFOREST

- XMLFOREST produces a sequence of XML elements given an optional namespace declaration and a list of named expressions as arguments.
- Each element can be named implicitly or explicitly.
- If any of the expressions evaluate to the null value, it is ignored. If all the expressions evaluate to the null value, result is the null value.

XMLFOREST (contd.)

- Example:

```
SELECT e.id,
   XMLELEMENT (NAME "Emp",
         XMLFOREST (e.fname || ' ' || e.lname AS "name", e.hire)
                        ) AS xvalue
FROM employees e
WHERE ...

==>
ID                    XVALUE
1001        <Emp>
                <name> John Smith </name>
                <HIRE> 200-05-24 </HIRE>
            </Emp>
....
```

XMLFOREST (contd.)

- Namespace declaration, if present, must be the first argument.

```
SELECT e.id,
   XMLELEMENT (NAME "Emp",
        XMLFOREST (
             XMLNAMESPACES (DEFAULT 'http://a.b.c',
                                    'http://d.e.f' AS "xx"),
             e.fname || ' ' || e.lname AS "xx:name", e.hire)
                  ) AS xvalue
FROM employees e WHERE ...

==>
ID                  XVALUE
1001      <Emp>
               <xx:name xmlns:xx="http://d.e.f"> John Smith
               </xx:name>
               <HIRE xmlns="http:a.b.c"> 200-05-24 </HIRE>
          </Emp>
```

XMLCONCAT

- XMLCONCAT produces an XML value given given two or more expressions of XML type.
- If any of the expressions evaluate to the null value, it is ignored. If all the expressions evaluate to the null value, the result is the null value.

```
SELECT e.id,
  XMLCONCAT (XMLELEMENT (NAME "name",
                                    e.fname || ' ' || e.lname),
                    XMLELEMENT (NAME "hiredate", e.hire)
                    ) AS xvalue
FROM employees e WHERE ...

==>
ID                  XVALUE
1001          <name> John Smith </name>
                    <hiredate> 200-05-24 <hiredate>
```

XMLAGG

- XMLAGG is an <aggregate function>, similar to SUM, AVG, etc.
- The argument for XMLAGG must be an expression of XML type.
- For each row in a group *G*, the expression is evaluated and the resulting XML values are concatenated to produce a single XML value as the result for *G*.
- An ORDER BY clause can be specified to order the results of the argument expression before concatenating.
- All null values are dropped before concatenating.
- If all inputs to concatenation are null or if the group is empty, the result is the null value.

XMLAGG (contd.)

- Example:

```
SELECT XMLELEMENT
  (NAME "Department",
        XMLATTRIBUTES (e.dept AS "name"),
        XMLAGG (XMLELEMENT (NAME "emp", e.lname) )
  ) AS xvalue
FROM employees e
GROUP BY e.dept;

==>
XVALUE
<Department name="Accounting">
  <emp> Smith </emp>
  <emp> Martin </emp>
</Department>
....
```

XMLAGG (contd.)

- ORDER BY can be used to order the result of aggregation.

```
SELECT XMLELEMENT
  (NAME "Department",
       XMLATTRIBUTES (e.dept AS "name"),
       XMLAGG (XMLELEMENT (NAME "emp", e.lname)
                    ORDER BY e.lname)
  ) AS xvalue
FROM employees e
GROUP BY e.dept;

==>
XVALUE
<Department name="Accounting">
  <emp> Martin </emp>
  <emp> Smith </emp>
</Department>
....
```

Statement-level Namespace Declarations

- Namespace declarations can be attached to query expressions and to certain SQL statements; such namespaces will take effect for all publishing functions in scope by default.
- Namespace declarations for query expressions:

<query expression> ::= [<with clause>] <query exp body>

<with clause> ::=
WITH [RECURSIVE] <XML namespace declaration>
[<with list>]

Statement-level Namespace Declarations (contd.)

- Namespace declarations in DML statements: can be specified in DELETE, UPDATE, MERGE, and INSERT statements
- Example syntax:

<delete statement:searched> ::=
DELETE
[WITH <XML namespace declaration>]
FROM <target table> [WHERE <search condition>]

Statement-level Namespace Declarations (contd.)

- Namespace declarations for DDL statements: can be specified only for the <generation clause> in column definitions, check constraint definitions and assertion definitions
- Example syntax:

<check constraint definition> ::=
CHECK (
[WITH <XML namespace declaration>]
<search condition>)

Statement-level Namespace Declarations (contd.)

- Namespace declarations for compound statements in PSM:

<compound statement> ::=
[<beginning label> <colon>]
BEGIN [[NOT] ATOMIC]
[DECLARE <XML namespace declaration>;]
[<local declaration list>]
[<local handler declaration list>]
[<SQL statement list>]
END [<ending label>]

Mapping SQL identifiers to XML Names

- Issues
  - SQL identifiers use an implementation-defined character set while XML Names use UCS.
  - SQL regular identifiers are case-insensitive, while XML Names are case-sensitive.
  - SQL delimited identifiers allow any legal characters while XML restricts the characters that can be used in XML Names.
- Solutions
  - Require an implementation-defined mapping from SQL character set to UCS.
  - Use special escape sequences to deal with characters that are illegal in XML Names.

Mapping SQL identifiers to XML Names (contd.)

- Rules for mapping regular identifiers
  - Each character in SQL names is mapped to its upper case equivalent.
    - *employee* from SQL is mapped to *EMPLOYEE* in XML.
  - What if SQL names start with XML?
    - Option 1: "Partially escaped" mode - do nothing special
      - *XMLTEXT* from SQL mapped to *XMLTEXT* in XML.
    - Option 2: "Fully escaped mode" - map initial X to _x0058_
      - *XMLTEXT* from SQL mapped to *_x0058_MLTEXT* in XML.

Mapping SQL identifiers to XML Names (contd.)

- Rules for mapping delimited identifiers
  - Each character in SQL names retains its case
    - *"Employee"* from SQL mapped to *Employee* in XML.
    - *"Work_address"* from SQL mapped to *Work_address* in XML.
  - What if SQL names contain chracters that are illegal in XML Names?
    - Map illegal characters to _xNNNN_ or _xNNNNNN_ , where N is a hex digit and NNNN or NNNNN is Unicode representation of the character.
      - *work@home"* from SQL mapped to *work_x0040_home* in XML.
      - *"last.name"* from SQL mapped to *last_x002E_name* in XML.
  - What if SQL names contain _x?
    - Escape the _ in _x:
      - *"Emp_xid" from SQL mapped to Emp_005F_xid* in XML.

Mapping SQL identifiers to XML Names (contd.)

- Rules for mapping delimited identifiers (contd.)
  - What if SQL names started with xml in any case combinations?
    - Option 1: "Partially escaped" mode - do nothing special
    - Option 2: "Fully escaped" mode - map initial x to _x0078_ or X to _x0058_:
      - *"xmlText"* from SQL mapped to *_x0078_mlText* in XML.
  - What if SQL names included a : ?
    - Option 1: "Partially escaped" mode - map only the leading colon to _x003A_
      - *":ab:cd"* from SQL mapped to *_x003A_ab:cd* in XML.
    - Option 2: "Fully escaped mode" - map every colon to _x003A_
      - *":ab:cd" from SQL mapped to _x003A_ab_x003A_cd* in XML

Mapping XML Names to SQL identifiers

- Rules
  - Map all sequences of _xNNNN_ and _xNNNNNN_ to the corresponding Unicode character; if there is no corresponding Unicode character, map to a sequence of implementation-defined characters.
  - Put double quotes around the result to make an SQL delimited identifier; double each contained double quote.
    - *employee* from XML is mapped to *"employee"* in SQL.
    - *EMPLOYEE* from XML is mapped to *"EMPLOYEE"* in SQL.
    - *work_x0040_home* from XML mapped to *"work@home"* in SQL.
  - Map the resulting string to SQL_TEXT character set using implementation-defined mapping rules - raise an exception if the mapping is not possible.

Mapping SQL types to XML Schema types

- SQL predefined data types
  - SMALLINT, INTEGER, BIGINT, DECIMAL, NUMERIC
  - FLOAT, REAL, DOUBLE PRECISION
  - CHARACTER, CHARACTER VARYING, CHARACTER LARGE OBJECT
  - BINARY LARGE OBJECT
  - BOOLEAN
  - DATE, TIME, TIMESTAMP, INTERVAL
  - XML
- SQL User-defined types
  - Distinct types
  - Structured types
- SQL Constructed types
  - ROW
  - ARRAY
  - MULTISET
  - REF

Mapping SQL types to XML Schema types (contd.)

- XML Schema types
  - Numbers: decimal, float, double, 12 additional types derived from decimal
  - Strings: string, 12 additional derived types
  - Boolean: boolean
  - Datetime: duration, date, dateTime, 6 more primitive types
  - Other: 5 additional primitive types
  - List types
  - Union types
  - Simple types
  - Complex types

Mapping SQL types to XML Schema types (contd.)

- Map each SQL predefined type to its closest analog in XML Schema.
- Appropriate XML Schema facets are used to constrain the range of values of XML Schema types to match the range of values of SQL types.
- Annotations are used to capture the distinctions among SQL types (e.g., CHARACTER VARYING and CHARACTER LARGE OBJECT cannot be distinguished in XML Schema).
  - Though the standard specifies the contents of annotations, it is implementation-dependent whether these are generated.
- In addition to the predefined types, mapping for row types, collection types, and distinct types is provided.

Mapping SQL types to XML Schema types (contd.)

- INTEGER type from SQL maps to the following XML Schema type:

```
<xsd:simpleType>
    <xsd:annotation>
        <xsd:appinfo>
            <sqlxml:sqltype kind="predefined"
                    name="INTEGER"/>
        </xsd:appinfo>
    </xsd:annotation>
    <xsd:restriction base="xsd:integer"/>
        <xsd:maxInclusive value="IMP_MAX"/>
        <xsd:minInclusive value="IMP_MIN"/>
    </xsd:restriction>
</xsd:simpleType>
```

Mapping SQL types to XML Schema types (contd.)

- SQL type *CHAR (10) CHARACTER SET LATIN1 COLLATION deutsch* maps to the following XML Schema type:

```
<xsd:simpleType>
        <xsd:annotation>
            <xsd:appinfo>
                <sqlxml:sqltype kind="predefined"
                                name="CHAR" length="10"
                                characterSetName="LATIN1"
                                collation="deutsch"/>
            </xsd:appinfo>
        </xsd:annotation>
        <xsd:restriction base="xsd:string"/>
            <xsd:length value="10"/>
        </xsd:restriction>
</xsd:simpleType>
```

Mapping SQL values to XML

- Data type of values is mapped to corresponding XML schema types.
- Values of predefined types are first cast to a character string and then the resulting string is mapped to the string representation of the corresponding XML value.
- Values of numeric types with no fractional part are mapped with no periods.
- NULLs are mapped to either xsi:nil="true" or to absent elements, except for values of collection types whose NULLs are always mapped to xsi:nil="true".
- For character values, characters such as <, >, &, carriage return in the resulting string are replaced by their entitized forms.

Mapping SQL values to XML (contd.)

- Examples

SQL Value	XML Value
'Smith'	Smith
10	10
99.05	99.05
99.0	99
TIME '12:30:00'	12:30:00
TIMESTAMP '2002-10-14 11:00:00'	2002-10-14T11:00:00
INTERVAL '2:30' HOUR TO MINUTE	PT02H30M

Mapping SQL tables, schemas, and catalogs to XML documents

- Two documents are generated - a XML schema document and a XML document.
- Users can control whether a table is mapped to a single element or a sequence of elements.
- In a single element option:
  - The table name serves as the element name.
  - Each row is mapped to a nested element with each element named as "row".
  - Each column is mapped to a nested element with column name serving as the element name.
- In a sequence of elements option:
  - Each row is mapped to an element with the table name serving as the element name.
  - Each column is mapped to a nested element with column name serving as the element name.

Mapping SQL tables to XML

- EMPLOYEE table in single-element option:

```
<employee>
    <row>
        <id> 1001 </emp_id>
        <lastname> Smith </lastname>
        ...
    </row>
    <row>
        <id> 1206 </emp_id>
        <lastname> Martin </lastname>
        ...
    </row>
</employee>
```

Mapping SQL tables to XML (contd.)

- EMPLOYEE table in sequence-of-elements option:

```
<employee>
        <id> 1001 </emp_id>
        <lastname> Smith </lastname>
        ...
  </employee>
  <employee>
        <id> 1206 </emp_id>
        <lastname> Martin </lastname>
        ...
</employee>
```

Host Language Binding

- To transfer XML values from the SQL environment to host language environment, XML values are converted into character strings by implicit invocation of XMLSERIALIZE operator.
- To transfer XML values from the host language environment to SQL environment, XML values in the form of character strings are converted into XML values by implicit invocation of XMLPARSE operator.

Host Language Binding (contd.)

- Host variable declarations must indicate both the XML type and the intended character string type (VARCHAR or CLOB) and either the DOCUMENT or CONTENT option:

```
EXEC SQL BEGIN DECLARE SECTION;
    SQL TYPE IS XML DOCUMENT AS CLOB (1M)
        CHARACTER SET IS UTF8 xml_hv;
EXEC SQL END DECLARE SECTION;
```

- It is the responsibility of SQL implementations to ensure that the XML documents shipped to the host language program are in the character encoding specified in the host variable declaration.

Host Language Binding (contd.)

- SET XML OPTION statement is provided to set the DOCUMENT or CONTENT option for use by dynamic SQL statements during assignments to dynamic parameters:

```
SET XML OPTION DOCUMENT;
```

SQL/XML Feature Taxonomy

- SQL/XML functionality is divided into 44 features:
  - X010: XML type
  - X011: Arrays of XML type
  - X012: Multisets of XML type
  - X013: Distinct types of XML type
  - X014: Attributes of XML type
  - X015: Fields of XML type
  - X016: Persistent XML values
  - X020: XML Concatenation
  - X031: XMLElement
  - X032: XMLForest
  - X033: XMLRoot

Feature Taxonomy (contd.)

- X034: XMLAgg
- X035: XMLAGG: ORDER BY option
- X041: Basic table mapping: null absent
- X042: Basic table mapping: null as nil
- X043: Basic table mapping: table as forest
- X044: Basic table mapping: table as element
- X045: Basic table mapping: with target namespace
- X046: Basic table mapping: data mapping
- X047: Basic table mapping: metadata mapping
- X051: Advanced table mapping: null absent
- X052: Advanced table mapping: null as nil

Feature Taxonomy (contd.)

- X053: Advanced table mapping: table as forest
- X054: Advanced table mapping: table as element
- X055: Advanced table mapping: with target namespace
- X056: Advanced table mapping: data mapping
- X057: Advanced table mapping: metadata mapping
- X060: XMLParse: CONTENT option
- X061: XMLParse: DOCUMENT option
- X062: XMLParse: explicit WHITESPACE option
- X070: XMLSerialize: CONTENT option
- X071: XMLSerialize: DOCUMENT option
- X080: Namespaces in XML publishing

Feature Taxonomy (contd.)

- X081: Query-level XML namespace declarations
- X082: XML namespace declarations in DML
- X083: XML namespace declarations in DDL
- X084: XML namespace declarations in compound statements
- X090: XML document predicate
- X100: Host language support for XML: CONTENT option
- X101: Host language support for XML: DOCUMENT option
- X110: Host language support for XML: VARCHAR mapping
- X111: Host language support for XML: CLOB mapping
- X120: XML parameters in SQL routines
- X121: XML parameters in external routines

SQL/XML conformance

- To conform, implementations must support
  - X010: XML type (transient values only)
  - X031: XMLElement
  - X032: XMLForest
  - X034: XMLAgg
  - At least one of:
    - X070: XMLSerialize: CONTENT option
    - X071: XMLSerialize: DOCUMENT option
    - X100: Host language support for XML: CONTENT option and X110: Host language support for XML: VARCHAR mapping
    - X101: Host language support for XML: DOCUMENT option X110: Host language support for XML: VARCHAR mapping
    - X100: Host language support for XML: CONTENT option and X111: Host language support for XML: CLOB mapping
    - X101: Host language support for XML: DOCUMENT option X111: Host language support for XML: CLOB mapping