SQL99, SQL/MM, and SQLJ: An Overview of the SQL Standards

Nelson M. Mattos

with contributions from Kathryn Zeidenstein, Krishna Kulkarni, Stefan Dessloch, Hugh Darwen, Paul Cotton, Peter Pistor, Curt Cotner, Bob Lyle, Bill Bireley, and many others

IBM Database Common Technology

Table of Contents

Database Standards Organizations.................................................................13
Process..............................................................................................................16
Other Related Standards..................................................................................17
History of SQL Standard..................................................................................18
DBL Project History........................................................................................18
Progression of SQL Standards........................................................................19
SQL Conforming Products..............................................................................21
SQL 86..............................................................................................................22
Definition of Orthogonality............................................................................23
SQL 89..............................................................................................................24
SQL 92..............................................................................................................27
Entry Level.......................................................................................................29
Transitional Level............................................................................................30
Intermediate Level............................................................................................32
Full Level..........................................................................................................35
FIPS 127-2.......................................................................................................38
SQL99 (so-called SQL3)..................................................................................39
Overview..........................................................................................................39
SQL Framework...............................................................................................42
SQL Foundation...............................................................................................43
Overview..........................................................................................................43
Database Objects..............................................................................................50
Catalogs and Schemas....................................................................................51
Schema Manipulation Language.......................................................................52
# Table of Contents

SQL99 (so-called SQL3) continued

## SQL Foundation continued

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifiers</td>
<td>53</td>
</tr>
<tr>
<td>Information Schema Tables</td>
<td>54</td>
</tr>
<tr>
<td>Data Types</td>
<td>55</td>
</tr>
<tr>
<td>Overview</td>
<td>55</td>
</tr>
<tr>
<td>Predefined Types</td>
<td>56</td>
</tr>
<tr>
<td>Character String Data</td>
<td>58</td>
</tr>
<tr>
<td>Character Sets</td>
<td>61</td>
</tr>
<tr>
<td>Collations</td>
<td>63</td>
</tr>
<tr>
<td>Translations and Conversions</td>
<td>64</td>
</tr>
<tr>
<td>Boolean Type</td>
<td>65</td>
</tr>
<tr>
<td>Date, Time, Timestamp, Interval Types</td>
<td>66</td>
</tr>
<tr>
<td>Intervals</td>
<td>68</td>
</tr>
<tr>
<td>Operations</td>
<td>69</td>
</tr>
<tr>
<td>Data Conversions</td>
<td>70</td>
</tr>
<tr>
<td>Domains</td>
<td>71</td>
</tr>
<tr>
<td>SQL-invoked Routines</td>
<td>72</td>
</tr>
<tr>
<td>SQL Routines</td>
<td>75</td>
</tr>
<tr>
<td>External Routines</td>
<td>77</td>
</tr>
<tr>
<td>Routine Characteristics</td>
<td>81</td>
</tr>
<tr>
<td>Privilege Requirements</td>
<td>83</td>
</tr>
<tr>
<td>Routine Overloading</td>
<td>84</td>
</tr>
<tr>
<td>Specific Names</td>
<td>85</td>
</tr>
<tr>
<td>Routine Invocation</td>
<td>86</td>
</tr>
<tr>
<td>Subject Routine Determination</td>
<td>87</td>
</tr>
<tr>
<td>Dropping Routines</td>
<td>91</td>
</tr>
<tr>
<td>Altering Routines</td>
<td>92</td>
</tr>
</tbody>
</table>

---

## Object-Relational Support

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview and Motivation</td>
<td>94</td>
</tr>
<tr>
<td>Large Object Data Types</td>
<td>97</td>
</tr>
<tr>
<td>LOB Functions</td>
<td>102</td>
</tr>
<tr>
<td>Locators</td>
<td>104</td>
</tr>
<tr>
<td>User-defined Types</td>
<td>109</td>
</tr>
<tr>
<td>Distinct Types</td>
<td>112</td>
</tr>
<tr>
<td>Structured Types</td>
<td>115</td>
</tr>
<tr>
<td>Methods</td>
<td>118</td>
</tr>
<tr>
<td>Creating Structured Types</td>
<td>121</td>
</tr>
<tr>
<td>Uninstantiable Types</td>
<td>122</td>
</tr>
<tr>
<td>Manipulating Attributes</td>
<td>123</td>
</tr>
<tr>
<td>Dot Notation</td>
<td>125</td>
</tr>
<tr>
<td>Initializing Instances</td>
<td>126</td>
</tr>
<tr>
<td>Manipulating Structured Types</td>
<td>128</td>
</tr>
<tr>
<td>Subtyping and Inheritance</td>
<td>129</td>
</tr>
<tr>
<td>Value Substitutability</td>
<td>132</td>
</tr>
<tr>
<td>Structured Types used as Column Types</td>
<td>133</td>
</tr>
<tr>
<td>Structured Types used as Row types - Typed Tables</td>
<td>136</td>
</tr>
<tr>
<td>Reference Types</td>
<td>137</td>
</tr>
<tr>
<td>Subtables - Table Hierarchies</td>
<td>140</td>
</tr>
<tr>
<td>Substitutability</td>
<td>141</td>
</tr>
<tr>
<td>Substitutability: Type Predicate and ONLY</td>
<td>142</td>
</tr>
<tr>
<td>Path Expressions - &lt;dereference operator&gt;</td>
<td>143</td>
</tr>
<tr>
<td>Method Reference</td>
<td>144</td>
</tr>
<tr>
<td>Reference Resolution</td>
<td>145</td>
</tr>
</tbody>
</table>
## Table of Contents

### SQL99 (so-called SQL3) continued

#### SQL Foundation continued

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predicates</td>
<td>214</td>
</tr>
<tr>
<td>Extensions to BETWEEN</td>
<td>216</td>
</tr>
<tr>
<td>Extensions to SIMPLE match</td>
<td>217</td>
</tr>
<tr>
<td>DISTINCT</td>
<td>218</td>
</tr>
<tr>
<td>SIMILAR</td>
<td>219</td>
</tr>
<tr>
<td>Type Predicate</td>
<td>220</td>
</tr>
<tr>
<td>SET Operators</td>
<td>221</td>
</tr>
<tr>
<td>DML Orthogonality</td>
<td>222</td>
</tr>
<tr>
<td>CAST Specification</td>
<td>225</td>
</tr>
<tr>
<td>CASE Expression</td>
<td>226</td>
</tr>
<tr>
<td>Joined Tables</td>
<td>227</td>
</tr>
<tr>
<td>OUTER Join</td>
<td>228</td>
</tr>
<tr>
<td>Derived Tables (Table Expressions)</td>
<td>229</td>
</tr>
<tr>
<td>Common Table Expression</td>
<td>230</td>
</tr>
<tr>
<td>Recursive SQL</td>
<td>231</td>
</tr>
<tr>
<td>Overview</td>
<td>232</td>
</tr>
<tr>
<td>Bill of Material Queries</td>
<td>234</td>
</tr>
<tr>
<td>OLAP Extensions</td>
<td>240</td>
</tr>
<tr>
<td>ROLLUP</td>
<td>242</td>
</tr>
<tr>
<td>CUBE</td>
<td>244</td>
</tr>
<tr>
<td>GROUPING SETS</td>
<td>246</td>
</tr>
<tr>
<td>Grand Total</td>
<td>247</td>
</tr>
<tr>
<td>GROUPING Function</td>
<td>248</td>
</tr>
<tr>
<td>Selecting Nongrouped Columns</td>
<td>250</td>
</tr>
</tbody>
</table>

### SQL99 (so-called SQL3) continued

#### SQL Foundation continued

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predicates</td>
<td>214</td>
</tr>
<tr>
<td>Extensions to BETWEEN</td>
<td>216</td>
</tr>
<tr>
<td>Extensions to SIMPLE match</td>
<td>217</td>
</tr>
<tr>
<td>DISTINCT</td>
<td>218</td>
</tr>
<tr>
<td>SIMILAR</td>
<td>219</td>
</tr>
<tr>
<td>Type Predicate</td>
<td>220</td>
</tr>
<tr>
<td>SET Operators</td>
<td>221</td>
</tr>
<tr>
<td>DML Orthogonality</td>
<td>222</td>
</tr>
<tr>
<td>CAST Specification</td>
<td>225</td>
</tr>
<tr>
<td>CASE Expression</td>
<td>226</td>
</tr>
<tr>
<td>Joined Tables</td>
<td>227</td>
</tr>
<tr>
<td>OUTER Join</td>
<td>228</td>
</tr>
<tr>
<td>Derived Tables (Table Expressions)</td>
<td>229</td>
</tr>
<tr>
<td>Common Table Expression</td>
<td>230</td>
</tr>
<tr>
<td>Recursive SQL</td>
<td>231</td>
</tr>
<tr>
<td>Overview</td>
<td>232</td>
</tr>
<tr>
<td>Bill of Material Queries</td>
<td>234</td>
</tr>
<tr>
<td>OLAP Extensions</td>
<td>240</td>
</tr>
<tr>
<td>ROLLUP</td>
<td>242</td>
</tr>
<tr>
<td>CUBE</td>
<td>244</td>
</tr>
<tr>
<td>GROUPING SETS</td>
<td>246</td>
</tr>
<tr>
<td>Grand Total</td>
<td>247</td>
</tr>
<tr>
<td>GROUPING Function</td>
<td>248</td>
</tr>
<tr>
<td>Selecting Nongrouped Columns</td>
<td>250</td>
</tr>
</tbody>
</table>
# Table of Contents

**SQL99 (so-called SQL3) continued**

**SQL Foundation continued**
- Update through UNION and JOIN .............................................................. 251
- INSERT through JOIN ............................................................................. 253
- Named Expressions ................................................................................ 254
- Cursors .................................................................................................... 255
  - Scrollable .......................................................................................... 255
  - READ ONLY, FOR UPDATE, INSENSITIVE .................................... 257
  - Holdable ............................................................................................. 258
- ORDER BY Expressions and on Columns not in Select List .............. 261
- Temporary Tables ................................................................................. 262
- Roles ...................................................................................................... 264
- Error Handling ...................................................................................... 267
- Transactions .......................................................................................... 269
- Savepoints ............................................................................................. 272
- Connections .......................................................................................... 273
- SQL Flagger .......................................................................................... 274
- Module Language .................................................................................. 275

**SQL Persistent Stored Modules (PSM)** .................................................. 276

**SQL Procedural Language Extensions** .................................................. 279
- Compound Statement ............................................................................ 280
- Assignment Statement .......................................................................... 284
- LEAVE Statement ................................................................................ 285
- IF Statement ........................................................................................ 286
- CASE Statement ................................................................................... 287
- LOOP Statement ................................................................................... 288
- WHILE Statement ................................................................................ 289
- REPEAT Statement ............................................................................... 290
- FOR Statement ....................................................................................... 291
- ITERATE Statement ............................................................................. 292
- Condition Handling .............................................................................. 293
- Embedding Control Statements .............................................................. 296

**SQL Language Bindings** ........................................................................ 297
- Embedded SQL ...................................................................................... 298
- Dynamic SQL ........................................................................................ 299
- Direct SQL ............................................................................................. 302

**SQL Call Level Interface (CLI)** ............................................................... 303
- Conformance .......................................................................................... 313
- Core ....................................................................................................... 314
- Packages ................................................................................................. 319
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL/MM</td>
<td>326</td>
</tr>
<tr>
<td>SQL/MM Full Text</td>
<td>328</td>
</tr>
<tr>
<td>SQL/MM Spatial</td>
<td>332</td>
</tr>
<tr>
<td>SQL/MM Still Image</td>
<td>340</td>
</tr>
<tr>
<td>SQLJ and JDBC</td>
<td>345</td>
</tr>
<tr>
<td>SQLJ Part 0 (Embedded SQL for Java)</td>
<td>346</td>
</tr>
<tr>
<td>SQLJ Syntax</td>
<td>348</td>
</tr>
<tr>
<td>SQLJ Versus JDBC</td>
<td>349</td>
</tr>
<tr>
<td>Result Set Iterators</td>
<td>352</td>
</tr>
<tr>
<td>Named Iterator</td>
<td>353</td>
</tr>
<tr>
<td>Positioned Iterator</td>
<td>354</td>
</tr>
<tr>
<td>Connection Contexts</td>
<td>355</td>
</tr>
<tr>
<td>Execution Contexts</td>
<td>356</td>
</tr>
<tr>
<td>Advanced Features</td>
<td>357</td>
</tr>
<tr>
<td>Compiling an SQLJ Application</td>
<td>358</td>
</tr>
<tr>
<td>Binary Portability</td>
<td>359</td>
</tr>
<tr>
<td>SQLJ Part 1 (Stored procedures and UDFs using Java)</td>
<td>360</td>
</tr>
<tr>
<td>Installing Java Classes in the DB</td>
<td>361</td>
</tr>
<tr>
<td>Creating Procedures and UDFs</td>
<td>362</td>
</tr>
<tr>
<td>Invoking SQLJ Routines</td>
<td>363</td>
</tr>
<tr>
<td>SQLJ Stored Procedures</td>
<td>364</td>
</tr>
<tr>
<td>Future Developments within SQL</td>
<td>383</td>
</tr>
<tr>
<td>SQL Management of External Data (MED)</td>
<td>386</td>
</tr>
<tr>
<td>SQL4</td>
<td>386</td>
</tr>
<tr>
<td>SQLJ and JDBC continued</td>
<td></td>
</tr>
<tr>
<td>SQLJ Part 1 (Stored procedures and UDFs for Java) continued</td>
<td></td>
</tr>
<tr>
<td>Error Handling</td>
<td>365</td>
</tr>
<tr>
<td>Additional Features</td>
<td>366</td>
</tr>
<tr>
<td>Conformance</td>
<td>367</td>
</tr>
<tr>
<td>JDBC 2.0 Extensions for SQL99 Types</td>
<td>368</td>
</tr>
<tr>
<td>&quot;Native&quot; Java Object Support</td>
<td>369</td>
</tr>
<tr>
<td>Mapping Java Objects to Structured Types</td>
<td>370</td>
</tr>
<tr>
<td>JDBC 2.0 Structured Type Support</td>
<td>371</td>
</tr>
<tr>
<td>Mapping Infrastructure</td>
<td>372</td>
</tr>
<tr>
<td>Object References</td>
<td>375</td>
</tr>
<tr>
<td>Manipulating Large Objects</td>
<td>376</td>
</tr>
<tr>
<td>Arrays</td>
<td>377</td>
</tr>
<tr>
<td>SQLJ Part 2 (SQL Types and Methods using Java)</td>
<td>378</td>
</tr>
<tr>
<td>Mapping Java Classes to SQL</td>
<td>379</td>
</tr>
<tr>
<td>Instance Update Methods</td>
<td>381</td>
</tr>
<tr>
<td>Future Developments within SQL</td>
<td></td>
</tr>
<tr>
<td>SQL Management of External Data (MED)</td>
<td></td>
</tr>
<tr>
<td>SQL4</td>
<td></td>
</tr>
</tbody>
</table>
Disclaimers

- The content of this presentation is not intended to represent the viewpoint of IBM, NCITS, or ISO DBL.
- This presentation does not cover all features of SQL99, SQL/MM, and SQLJ.
  - Not all options are presented
  - Examples do not include error handling, and may contain simplifications and/or inaccuracies
- Any problems caused by mistakes in this presentation are solely the responsibility of the user.

SQL Standard

- Goal: enable the portability of SQL applications across conforming products
- Side effect: Increases and stabilizes the database market
- Joint efforts between vendors and users
  - Computer Associates
  - IBM
  - Informix
  - Oracle
  - Sybase
  - Microsoft
  - etc.
- Joint effort among several countries
Database Standards Organizations: JTC1

- JTC1/SC32: Data Management and Interchange
  - WG1: Open EDI (Finland)
  - WG2: Metadata (USA)
  - WG3: Database Languages (Netherlands)
  - WG4: SQL Multimedia and Application Packages (Japan)
  - WG5: Remote Database Access (RDA) (United Kingdom)
  - RG1: Reference Model for Data Management (Maintenance) (United Kingdom)
  - RG2: Export /Import (Maintenance) (Canada)

Database Standards Organizations: JTC1

- JTC1/SC32/WG3 Projects (SQL3 only):
  - Part 1: Framework
  - Part 2: Foundation
  - Part 3: Call-Level Interface
  - Part 4: Persistent Stored Modules
  - Part 5: Language Bindings
  - Part 6: XA Specialization
  - Part 7: Temporal
  - Part 9: Management of External Data
  - Part 10: Object Language Bindings

- JTC1/SC32/WG4 Projects:
  - Part 1: SQL/MM Framework
  - Part 2: SQL/MM Full-Text
  - Part 3: SQL/MM Spatial
  - Part 4: SQL/MM General Purpose Facilities
  - Part 5: SQL/MM Still Image
**Database SQL Standards**

- **Specification:**
  - Vendor extensions allowed
  - Implementation-defined behaviors exist

- **Players**
  - US: ANSI NCITS H2 Database Language Committee
    - Mix of vendors (18) and users (13)
  - International: ISO/JTC1 SC32/WG3 DBL Working Group (Database Languages)
    - 11 countries participating
  - Consortia: SQLJ
    - Major database vendors

- **Process**
  - Standards are produced by volunteers
  - Open process oriented towards achieving consensus
  - Proposals to change existing base document

- **Life cycle of an ISO standard:**
  - Working Draft
  - Proposals
  - Committee Draft
  - Proposals
  - Final Committee Draft
  - Technical Corrigendum
  - International Standard
  - YES/NO vote
  - Final Draft
  - International Standard
  - Review every 5 years to reaffirm, replace, or withdraw
### Other Related Standards

- **NDL:** (X3.133-1986)
  - Network Database Language
  - Has been reaffirmed for another 5 years
  - Cancelled as international standard

- **RDA:** (IS 9579-1: 1993, IS 9579-2:1993)
  - Remote Database Access
  - Defines client/server protocol
  - IS 9579-2, Information Technology - Remote Database Access - Part 2: SQL Specialization
  - RDA/SQL Amendm.1: Secure RDA (work in progress)
  - RDA/SQL Amendm.2: Distribution Schema for RDA (work in progress)
  - RDA/SQL Amendm.3: Encompassing Transaction (work in progress)
  - RDA/SQL Support for SQL3 (work in progress)

### DBL Project History

<table>
<thead>
<tr>
<th>Year</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>early 70's</td>
<td>Ted Codd's first papers on Relational Algebra</td>
</tr>
<tr>
<td>1975</td>
<td>CODASYL Database Specifications</td>
</tr>
<tr>
<td>1977</td>
<td>database Project Initiated in U.S.</td>
</tr>
<tr>
<td>1978</td>
<td>ANSI Database Project Approved</td>
</tr>
<tr>
<td>1979</td>
<td>ISO Database Project Initiated</td>
</tr>
<tr>
<td>1982</td>
<td>ANSI Project Split into NDL and SQL</td>
</tr>
<tr>
<td>1983</td>
<td>ISO Project Split into NDL and SQL</td>
</tr>
<tr>
<td>1986</td>
<td>ANSI SQL Published - December</td>
</tr>
<tr>
<td>1987</td>
<td>ISO/IEC 9075:1986 (SQL86)</td>
</tr>
</tbody>
</table>
Progression of SQL Standards

- SQL/86
- SQL/89 (FIPS 127-1)
- SQL/89 with Integrity Enhancement 120pp
- SQL/92 622pp July 92
  - Entry Level (FIPS 127-2)
  - Intermediate Level
  - Full Level
- SQL CLI 200pp Sept 95
- SQL PSM 250pp Nov 96
- SQL/3 (Work in Progress)
  - SQL Framework 20pp May 99
  - SQL Foundation 900pp May 99
  - SQL Call Level Interface (CLI) 100pp Dec 99
  - SQL Persistent Stored Modules (PSM) 150pp May 99
  - SQL Language Bindings 200pp May 99
  - SQL Management of External Data 112pp Jul 00
  - SQL Object Language Bindings 238pp Feb 00
- SQL/4 (Work to be defined soon)
  - All of the above, and...
  - XA
  - SQL Temporal

Progression of SQL Standards (cont.)

- SQL/MM (for SQL3)
  - Framework 007pp Oct 99
  - Full Text 208pp May 99
  - Spatial 343pp May 99
  - Still Image 045pp Oct 00

- SQL/MM "Later progression"
  - Full Text Sep 01
  - Spatial Sep 01
SQL Conforming Products

- **Validation**
  - Performed formerly by NIST. Discontinued in 1996. Other organizations are considering using the NIST test suite for certification.

- **SQL/89**
  - 11 validated products on 52 different platforms

- **SQL/92**
  - Over 10 validated products on over 100 different platforms
  - IBM DB2
  - Informix Online
  - Microsoft SQL Server
  - Oracle 7 and Rdb
  - Software AG ADABAS D
  - Sybase SQL Server
  - etc.

---


- The starting point: IBM's SQL implementation
  - SQL/86 became a subset of IBM's SQL implementation

- Criticized for lack of common features and orthogonality (described in next slide)

- Defined 3 ways to process DML
  - "Direct processing"
  - "Module language"
  - Embedded SQL

- Bindings to
  - Cobol
  - Fortan
  - Pascal
  - PL/1
Orthogonality: What does that mean?

"Orthogonality means independence. A language is orthogonal if independent concepts are kept independent and not mixed together in confusing ways."

"...desirable because the less orthogonal a language is, the more complicated it is...and the less powerful it is."


- Superset of SQL/86
- Replaced SQL/86
- C and ADA were added to existing language bindings
- DDL in a separate "schema definition language"
  CREATE TABLE
  CREATE VIEW
  GRANT PRIVILEGES
  (No DROP, ALTER, OR REVOKE)
SQL/89 with Integrity Enhancement

- **DEFAULT**
  - Default value for a column when omitted at INSERT time
- **UNIQUE** (column-list)
- **NOT NULL**
- **Views WITH CHECK OPTION**
  - Insertions to view are rejected if they don’t satisfy the view-definition
- **PRIMARY KEYs**
- **CHECK constraint**
  - Integrity constraint on values in a single row
- **Referential Integrity**
  
  ```
  CREATE TABLE T2
  .... FOREIGN KEY (COL3) REFERENCES T1 (COL2)
  ```
  - Any update that would violate referential integrity is rejected

SQL/89 Language Bindings

- **Database Language Embedded SQL**
  (X3.168-1989)
  - ANSI only, not needed in ISO
  - Necessary because embedding was defined in an appendix in SQL/86 and SQL/89
  - C and ADA language bindings (in addition to COBOL, Fortran, Pascal, and PL/I)
SQL/92: Overview

- Superset of SQL/89
  - Very few incompatibilities documented in an annex
- Not "least-common-denominator"
- Significantly larger than SQL/89 (579 versus 120 pages)
  - Greater orthogonality
  - Data type extensions (varchar, bit, character sets, date, time & interval)
  - Multiple join operators
  - Catalogs
  - "Domains"
  - Derived tables in FROM clause
  - Assertions
  - Temporary tables
  - Referential actions
  - Schema manipulation language
  - Dynamic SQL
  - Scrollable cursors
  - Connections
  - Information schema tables

SQL/92: Overview (cont.)

- Many (but not all) features are available in existing products
- Divided into 3 levels:
  - Entry level (much the same as SQL/89 with Integrity Enhancement)
  - Intermediate level
  - Full level
- Features are assigned to level
  - Full is a superset of Intermediate
  - Intermediate is superset of Entry
- FIPS 127-2 defines a Transitional Level:
  - Level between Entry and Intermediate
  - Subset of Intermediate
  - Superset of Entry
SQL/92 Entry Level

- SQL/89 plus a small set of new features:
  - SQLSTATE
    - Carries more feedback information than SQLCODE
  - Delimited identifiers
    CREATE TABLE "SELECT"...
  - Named expressions in SELECT - list:

```sql
SELECT name, sal+comm AS pay
FROM employee
ORDER BY pay
```

SQL/92: Transitional Level

- Defined by FIPS 127-2
- Subset of SQL/92: Intermediate Level
- Data types and operators
  - DATE, TIME, TIMESTAMP, INTERVAL (with arithmetic)
  - CHAR VARYING(n)
  - LENGTH, SUBSTR, TRIM, and || (concatenate) operators
- Referential integrity with cascading delete
- New types of join
  - NATURAL JOIN
  - LEFT and RIGHT OUTER JOIN
- Dynamic SQL
  - PREPARE
  - EXECUTE
  - DESCRIBE
**SQL/92: Transitional Level (cont.)**

- Schema evolution
  - ALTER TABLE
  - DROP TABLE
  - REVOKE PRIVILEGE
- CAST (expression AS type)
  - Conversions among
    - Numeric types
    - Numeric <-> Character
    - Character <-> Date and time
- Standard Catalogs
  - TABLES VIEWS COLUMNS
  - PRIVILEGE
- Views containing UNION
- Multiple schemas (collection of tables and other objects) per user
- Transaction isolation levels
  - READ UNCOMMITTED
  - READ COMMITTED
  - REPEATABLE READ
  - SERIALIZABLE

**SQL/92: Intermediate Level**

- Scrollable cursors
- FULL OUTER JOIN
- Domains
  - "Macro" facility for data type, default, value, nullability, and CHECK constraint
  - No strong typing (type checking based on underlying data type)
  - Not the same as Codd’s notion of domains
- Online DDL
- Implicit casting
  - Scalar-valued subquery can be used in place of any scalar
SQL/92: Intermediate Level (cont.)

- Set operations between query blocks:
  - INTERSECT
  - EXCEPT
  - CORRESPONDING (allows operators to apply to like-named columns of tables)

- CASE expression
  ```sql
  SELECT CASE (sex)
  WHEN "F" THEN "female"
  WHEN "M" THEN "male"
  END
  ...
  ```

- COALESCE
  - Returns the first non-null value
  ```sql
  COALESCE (EMP.AGE, "Age is null")
  ```

SQL/92: Intermediate Level (cont.)

- UNIQUE predicate
  ```sql
  UNIQUE <subquery>
  ```
  - Returns true if the subquery returns no duplicates; otherwise, false

- 128-character identifiers

- Multiple character sets (including double-byte)

- SET statement to change authorization-ID

- More comprehensive catalog information
  - Constraints
  - Usage
  - Domains
  - Assertions

- Date and time arithmetic with time zones

- SQL FLAGGER
  - Extensions
  - Conforming language being processed in a non-conforming way
SQL/92: Full Level

- Derived tables
  - table-expressions in FROM-clause
- Referential integrity with CASCADE UPDATE and SET NULL
- Integrity assertions
  - Stand-alone assertions that apply to entire tables or multiple tables
  - Subqueries in CHECK clause
  - Deferred checking of constraints (including assertions)
- Enhanced predicates
  - Multiple-column matching:
    - WHERE (X,Y) MATCH (SELECT A, B FROM T2)
  - Comparison by high-order and low-order columns:
    - WHERE (X, Y) > (A,B)

SQL/92: Full Level (cont.)

- More types of join
  - CROSS JOIN
  - UNION JOIN
- New data types
  - BIT (n)
  - BIT VARYING (n)
- Temporary tables (vanish at end of transaction or session)
- Implementation-defined collating sequences
- More character-string operators:
  - UPPER
  - LOWER
  - POSITION
- INSERT privilege on individual columns
SQL/92: Full Level (cont.)

- Row and table constructors:
  
  ```sql
  ( (1, 'OPERATOR', 'JONES'),
     (2, 'PROGRAMMER', 'SMITH'),
     (3, 'MGR', 'MATTOS')
  )
  ```

- Explicit Tables
  - TABLE EMP can be a subquery

- DISTINCT applies to expression:
  ```sql
  SELECT COUNT (DISTINCT SAL+COMM)
  ```

- Cursors declared SENSITIVE (see updates after OPEN) or INSENSITIVE
- Updates via scrollable or ordered cursors
- UPDATE and DELETE with subqueries on the same table

FIPS SQL

- NIST (National Institute of Standards and Technology)
  - Publishes FIPS (Federal Information Processing Standards)
- A FIPS provides guidelines for purchases by U.S. federal agencies:
  - FIPS 127 for SQL/86
  - FIPS 127-1 for SQL/89
  - FIPS 127-2 for SQL/92
- FIPS requires a FIPS flagger to detect extensions to the standard
- NIST develops test suites
  - FIPS 127-1: close to 200 test cases
  - FIPS 127-2 for Entry Level of SQL/92: over 400 test cases
  - Performing validation tests
    - Conforming implementation placed on Validated Products List
    - Certificates of conformance issued
SQL99 Overview

- Superset of SQL/92
  - Completely upward compatible
- Significantly larger than SQL/92
  - Object-Relational extensions
    - User-defined data types
    - Reference types
    - Collection types (e.g., arrays)
    - Large object support (LOBs)
    - Table hierarchies
  - Triggers
  - Stored procedures and user-defined functions
  - Recursive queries
  - OLAP extensions (CUBE and ROLLUP)
  - SQL procedural constructs
  - Expressions in ORDER BY
  - Savepoints
  - Update through unions and joins

- Existing Standard
  - Development began before publication of SQL/92
  - Published in 1999
  - Identical ANSI and ISO standards
  - Other countries republish or translate
    - the ISO standard (e.g. Japan, Brazil)

- Many contributions from ...
  - Australia
  - Brazil
  - France
  - Canada
  - Germany
  - Italy
  - Japan
  - Netherlands
  - Spain
  - UK
  - USA
  - ...
SQL99 Overview

Multipart standard:

- **SQL/Framework** (Part 1)
  - Overview and conformance clause

- **SQL/Foundation** (Part 2)
  - The basics: types, schemas, tables, views, query and update statements, expressions, security model, predicates, assignment rules, transaction management and so forth

- **SQL/CLI** (Call Level Interface) (Part 3)
  - No preprocessing of SQL statements necessary

- **SQL/PSM** (Persistent Stored Modules) (Part 4)
  - Extensions to SQL to make it procedural

- **SQL/Bindings** (Part 5)
  - Dynamic, embedded, direct invocation

SQL99 Framework Overview

- **Overview**
  - Provides an overview of the complete standard

- **Conformance**
  - Contains conformance clause
SQL99 Foundation Overview

- All of SQL/92 functionality
  - Schemas
  - Different kinds of joins
  - Temporary tables
  - CASE expressions
  - Scrollable cursors
  - ...

- New built-in data types for increased modeling power
  - Boolean
  - Large objects (LOBs)

- Enhanced update capabilities
  - Increase expressive powers
    - Update/delete through unions
    - Update/delete through joins

- Other relational extensions to increase modeling and expressive power
  - Additional predicates (FOR ALL, FOR SOME, SIMILAR TO)
  - Extensions to cursors (sensitive cursor, holdable cursor)
  - Extensions to referential integrity (RESTRICT)
  - Extensions to joins

SQL99 Foundation Overview...

- Triggers
  - Enhances integrity mechanism (active DBMS)
    - Different triggering events: update/delete/insert
    - Optional condition
    - Activation time: before or after
    - Multi-statement action
    - Several triggers per table
    - Condition and multi-statement action per each row or per statement

- Roles
  - Enhanced security mechanisms
    - GRANT/REVOKE privileges to roles
    - GRANT/REVOKE roles to users and other roles
SQL99 Foundation Overview

- Recursion
  - Increase expressive power
  - Linear (both direct and mutual) recursion
  - Stop conditions
  - Different search strategies (depth first, breadth first)

- Savepoints
  - Enhances user-controlled integrity
  - Savepoint definition
  - Roll back to savepoint
  - Nesting

- OLAP extensions
  - Enhances query capabilities
    - CUBE
    - ROLLUP
    - Expressions in ORDER BY

Object-relational Extensions

- Extensibility: application specific data types "understandable" by DBMS
- Increase modeling power (complex objects): increase the range of applications
- Reusability: sharing existing type libraries
- Integration: enable integration of OO and relational concepts in a single language

- User-defined types
  - Distinct types
    - Strong typing
    - Type-specific behavior
  - Structured types
    - Strong typing
    - Type-specific behaviors
    - encapsulation
    - Value substitutability
    - Polymorphic routines
    - Dynamic binding (run-time function dispatch)
    - Compile-time type checking
SQL99 Foundation Overview

- Collection types
  - Arrays
- Row types
  - Like record structures in programming languages
  - Type of rows in tables
  - Nesting (rows with row-valued fields)
- Reference types
  - Support "object identity"
  - Navigational access (path expressions)

SQL99 Foundation Overview

- User-defined functions
  - SQL and external functions
  - Overloaded functions
  - User-defined paths
  - Compile time type checking
  - Static binding
- User-defined procedures
  - SQL and external procedures
  - NO overloading
  - Input and output parameters
  - Result sets
  - Static binding
- User-defined methods
  - Describe a user-defined type behavior
  - SQL and external methods
  - Overloading and overriding
  - Compile time checking
  - Late binding (dynamic dispatch)
**SQL99 Foundation Overview**

- Subtables (table hierarchies)
  - Increase modeling power and expressive power of queries
  - Means to model collection hierarchies or object extents
    - CREATE/DROP subtable
    - CREATE/DROP subview
    - Object "identity" by means of references
    - Queries on a table operate on subtables as well
    - "Object-like" manipulation through references and path expressions
  - Extensions to authorization model to support "object-like" manipulation

**Database Objects**

- SQL environment
- catalog
- client module
- privilege
- schema
- role
- trigger
- user-defined type
- assertion
- domain
- constraint
- viewed table
- table
- routine
- translation
- translation
- character set
- collation
- implementation
- authorization identifier
- procedure
- function
- method
Catalogs and Schemas

- SQL objects (i.e., tables, views, ...) are contained in schemas
- Schemas are contained in catalogs
- Each schema has a single owner
- Objects can be referenced with explicit or implicit catalog and schema name

```
FROM people        --unqualified name
FROM sample.people --partially qualified name
FROM cat1.sample.people --fully qualified name
```

Schema Manipulation Language

- Syntax for creating objects
- Syntax for dropping or revoking with two behaviors
  - RESTRICT disallows the operation if database objects exist that reference the object being dropped or revoked
  - CASCADE propagates the change (in some form) to database objects that may reference the object being dropped or revoked
    - DROP TABLE => drop assertion that references the table
    - DROP DOMAIN => columns that reference the domain take on the data type, constraints, and the default value of the domain
- Syntax for altering objects
  - Table
    - Add/drop column
    - Alter column default and scope
    - Add/drop constraints
  - Domain
    - Set/drop default
    - Add/drop constraint
  - User-defined type
    - Add/drop attribute
    - Add/drop method
  - SQL-invoked routines
    - Alter routine characteristics
**Identifiers**

- Up to 128 characters
- Lower case characters may be used in identifiers and key words
  - These lower case characters are considered to be their upper case counterparts
- Delimited identifiers are case sensitive
  - Allow wider range of characters

  ```sql
  CREATE TABLE "People Hobbies"
  ...
  ```

**Information Schema Tables**

- A set of views describing the metadata contained in a catalog
  - Exist in the INFORMATION_SCHEMA schema
  - Are fully defined (column names, data types, and semantics)
  - May be queried by users
  - Are read-only
  - Reflect database objects that the user owns or for which the user has some privilege

  TABLES
  COLUMNS
  VIEWS
  DOMAINS
  etc
Data Types

- Predefined types
  - Numeric
  - String
  - BLOB
  - Boolean
  - Datetime
  - Interval
- Constructed atomic types
  - Reference
- Constructed composite types
  - Collection: Array
  - Row
- User-defined types
  - Distinct type
  - Structured type

Predefined Types

- Numeric
- String
- Datetime
- Interval
- Boolean
- BLOB
- CLOB
- Varying
- Fixed
- Approximate
- Real
- Float
- Double
- Exact
- Smallint
- Integer
- Decimal
- Numeric
- Bit
- Character
- Date
- Time
- Timestamp
# Constructed Types

- **Atomic**
  - Currently, only one: *reference type*

- **Composite**

More collection types likely in SQL4

---

# Varying Length Character String

CHARACTER VARYING (150)
VARCHAR (150)

- The number of characters in a value may vary, from 0 to some implementation-defined maximum
- Additional functions

  - CHARACTER_LENGTH
  - OCTET_LENGTH

- Fully compatible with fixed-length character strings
  - *Comparison* is allowed between character strings, regardless of whether they are varying or fixed
  - *Assignment* is allowed between character strings, regardless of whether they are varying or fixed
- Character string literals are fixed-length character strings (i.e., CHAR)
**Bit Strings**

- **BIT (32)**
- **BIT VARYING (1024)**

- String of binary digits, very much like character strings
  - Literals
    - B'101010'
    - X'123456789ABCDEF' -- hex digits
  - Function
    - BIT_LENGTH (bs)

- Fixed and varying-length bit strings are fully compatible with one another
  - Comparison is allowed between bit strings, regardless of whether they are varying or fixed
  - Assignment is allowed between bit strings, regardless of whether they are varying or fixed

**String Operations**

- **Concatenation**
  - 'abc' || 'xyz' 'abcxyz'
  - b'10' || b'01' b'1001'

- **Position**
  - POSITION ('bc' IN 'abcd') 2

- **Substring**
  - SUBSTRING ('Alexandre' FROM 4 FOR 1) 'x'
  - SUBSTRING (b'1011' FROM 2 FOR 2) b'01'

- **Upper/lower case transformation**
  - UPPER ('Hello') 'HELLO'
  - LOWER ('HI') 'hi'

- **Elimination of blanks and other characters**
  - TRIM (LEADING ' ' FROM 'NELSON') 'NELSON'
  - TRIM (TRAILING ' ' FROM 'NELSON') 'NELSON'
  - TRIM (BOTH ' ' FROM 'NELSON') 'NELSON'
  - TRIM (BOTH 'N' FROM 'NELSON') 'ELSO'
Character Sets

- May be defined by a standard, by an implementation, or (in a limited fashion) by a user
  - Must have the space character
  - Comparisons and string operators require operands with the same character set
- Character set may be specified for literals, or for characteristics of CHAR and VARCHAR types
- Character set for identifiers: SQL_IDENTIFIER

```sql
SPANISH ¿Como Esta? -- character string
name CHAR(20) -- column data type
CHARACTER SET BRAZILIAN
```  

- In SQL92, user could specify character set for identifiers
- CREATE TABLE -- identifier
  ```sql
  _GERMAN Bücher
  ```

  This feature was removed in SQL99

SQL99-defined Character Sets

- SQL_CHARACTER
  - 52 upper/lower case simple characters
  - 10 digits
  - 21 special characters
- GRAPHIC_IRV
  - 95 characters of ISO 646:1991
- LATIN1 (aka ISO 8859-1)
- ISO8BIT (aka ACII_FULL)
- 3 Unicode character sets
  - ISO10646 UTF16, UTF8, and UCS2
- SQL_Text
  - Union of all supported character sets
- SQL_IDENTIFIER
  - Subset of SQL_Text
  - Implicitly used for identifiers
Collations

- Set of rules for ordering character strings
  - A character set has a default collation
  - Additional collations may be defined by the implementation or by the user
  - Rules exist to cover the case where operands of a comparison or operator have different collations
  - The use of collations is pervasive
    - Comparison predicate
    - DISTINCT
    - ORDER BY
    - GROUP BY

```
SELECT lname, COUNT(*)
FROM people
GROUP BY lname COLLATE latin1_insensitive;
```

Translations and Conversions

- TRANSLATE built-in function is used to change input characters to characters of another character set

```
TRANSLATE (lname USING german)
```

- CONVERT built-in function is used to change input characters to a different "form-of-use," where form-of-use is defined as "an encoding for representing characters (e.g., fixed length vs. variable length)"

```
CONVERT (lname USING utf8toutf16)
```
**Boolean Data Type**

- Comprises distinct truth values *true* and *false*
- *unknown* if nulls are allowed
  - SOME and EVERY are functions valid for boolean expressions and boolean result data types
  ```sql
  SELECT DEPT#, EVERY ( salary > 20000 ) AS all_rich,
      SOME (salary > 20000) AS some_rich
  FROM  EMP
  GROUP BY DEPT#;
  ```

<table>
<thead>
<tr>
<th>DEPT#</th>
<th>all_rich</th>
<th>some_rich</th>
</tr>
</thead>
<tbody>
<tr>
<td>J64</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>Q05</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>M05</td>
<td>false</td>
<td>true</td>
</tr>
</tbody>
</table>

- Boolean comparison:
  ```sql
  SELECT cname, storename
  FROM stores s, customer c
  WHERE within(s.zone, c.location) AND  --boolean
      overlaps (s.zone, 'California')
  ```

**Date, Time, and Timestamp**

<table>
<thead>
<tr>
<th>TYPE</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE</td>
<td>YEAR&lt;br&gt;MONTH&lt;br&gt;DAY</td>
</tr>
<tr>
<td>TIME [WITH TIME ZONES]</td>
<td>HOUR&lt;br&gt;MINUTE&lt;br&gt;SECOND (+ fractional digits)</td>
</tr>
<tr>
<td>TIMESTAMP [precision] [WITH TIMEZONE]</td>
<td>YEAR&lt;br&gt;MONTH&lt;br&gt;DAY&lt;br&gt;HOUR&lt;br&gt;MINUTE&lt;br&gt;SECOND (+ fractional digits)</td>
</tr>
</tbody>
</table>

- Comparisons are only allowed between the same types
### Date, Time, Timestamp

- Coordinated universal time (UTC) used to store TIME and TIMESTAMP values

- WITH TIME ZONE can be specified
  
  Each session has a time zone, which is used if no time zone is explicitly specified

- Additional functions
  
  - CURRENT_DATE
  - CURRENT_TIME
  - CURRENT_TIMESTAMP (3)

- Literals
  
  - DATE ‘1992-06-03’
  - TIME ‘13:00:00’
  - TIME ‘13:00:00.5+08:00’
  - TIMESTAMP ‘1992-06-03 13:00:00’

### Intervals

<table>
<thead>
<tr>
<th>TYPE</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>year - month</td>
<td>YEAR</td>
</tr>
<tr>
<td></td>
<td>MONTH</td>
</tr>
<tr>
<td>day - time</td>
<td>DAY</td>
</tr>
<tr>
<td></td>
<td>HOUR</td>
</tr>
<tr>
<td></td>
<td>MINUTE</td>
</tr>
<tr>
<td></td>
<td>SECOND ( + fractional digits)</td>
</tr>
</tbody>
</table>

- INTERVAL YEAR TO MONTH
- INTERVAL HOUR
- INTERVAL HOUR TO MINUTE
- INTERVAL MINUTE TO SECOND (1)

- May be positive or negative
- Interval qualifier determines the specific fields to be used
- Literals:
  
  - INTERVAL +1-3’ YEAR TO MONTH
  - INTERVAL ‘15:15,15’ MINUTE TO SECOND (2)

- Comparisons cannot be performed between the two types of intervals
Operations on Datetime and Interval Values

- The following operations are supported:

<table>
<thead>
<tr>
<th>1st operand</th>
<th>operator</th>
<th>2nd operand</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATETIME</td>
<td>-</td>
<td>DATETIME</td>
<td>INTERVAL</td>
</tr>
<tr>
<td>DATETIME</td>
<td>+</td>
<td>INTERVAL</td>
<td>DATETIME</td>
</tr>
<tr>
<td>DATETIME</td>
<td>-</td>
<td>INTERVAL</td>
<td>DATETIME</td>
</tr>
<tr>
<td>INTERVAL</td>
<td>+</td>
<td>DATETIME</td>
<td>DATETIME</td>
</tr>
<tr>
<td>INTERVAL</td>
<td>+</td>
<td>INTERVAL</td>
<td>INTERVAL</td>
</tr>
<tr>
<td>INTERVAL</td>
<td>-</td>
<td>INTERVAL</td>
<td>INTERVAL</td>
</tr>
<tr>
<td>INTERVAL</td>
<td>*</td>
<td>number</td>
<td>INTERVAL</td>
</tr>
<tr>
<td>INTERVAL</td>
<td>/</td>
<td>number</td>
<td>INTERVAL</td>
</tr>
<tr>
<td>number</td>
<td>*</td>
<td>INTERVAL</td>
<td>INTERVAL</td>
</tr>
</tbody>
</table>

Data Type Conversions

- Implicit conversions
  - Dynamic SQL
  - UNION and similar operators

  VALUES ( (10.5), (5.E2), (2) )
  - results in data type approximate numeric

- Explicit conversions by means of CAST specification
**Domains**

- Persistent (named) definition of
  - A data type
  - An optional default value
  - An optional set of constraints
  - An optional collating sequence
- Used in place of a data type
- Do not provide strong typing
  - Not true “relational domains”

```sql
CREATE DOMAIN money AS DECIMAL (7,2);
CREATE DOMAIN account_type AS CHAR (1)
  DEFAULT 'C'
  CONSTRAINT account_type_check CHECK ( value IN ('C', 'S', 'M'));
CREATE TABLE accounts
  (account_id INTEGER,
   balance money,
   type account_type);
```

**SQL-invoked Routines**

- Named persistent code to be invoked from SQL
  - SQL-invoked procedures
  - SQL-invoked functions
    - SQL-invoked methods
- Created directly in a schema or in a SQL-server module
  - schema-level routines
  - module-level routines
- Have schema-qualified 3-part names
- Supported DDL
  - CREATE and DROP statements
  - ALTER statement -- still limited in functionality
  - EXECUTE privilege controlled through GRANT and REVOKE statements
- Described by corresponding information schema views
SQL-invoked Routines (cont.)

- Have a header and a body
  - Header consists of a name and a (possibly empty) list of parameters.
- Parameters of procedures may specify parameter mode
  - IN
  - OUT
  - INOUT
- Parameters of functions are always IN
- Functions return a single value
  - Header must specify data type of return value via RETURNS clause
- SQL routines
  - Both header and body specified in SQL
- External routines
  - Header specified in SQL
  - Bodies written in a host programming language
    - May contain SQL by embedding SQL statements in host language programs or using CLI

Advantages of external routines:
- Can utilize more "complete" languages
- Can take advantage of existing code libraries
- Can use widely-accepted languages

Disadvantages of external routines:
- Need to learn two different languages (no integrated programming environment)
- Need to convert between SQL data types and host language data types (loss of type behavior and type checking)

Advantages of SQL routines:
- Integrated programming language and programming environment (easier to use)
- No need for mapping SQL data types to host language types (type behavior and type checking are not lost across the boundary)

Disadvantages of SQL routines:
- Not as complete as host languages
- No wide acceptance
SQL Routines

- Parameters
  - Must have a name
  - Can be of any SQL data type

- Routine body
  - Consists of a single SQL statement
    - Can be a compound statement: BEGIN ... END
  - Not allowed to contain
    - DDL statement
    - CONNECT or DISCONNECT statement
    - Dynamic SQL
    - COMMIT or ROLLBACK statement

CREATE PROCEDURE get_balance(IN acct_id INT, OUT bal DECIMAL(15,2))
BEGIN
  SELECT balance INTO bal
  FROM accounts WHERE account_id = acct_id;
  IF bal < 100 THEN SIGNAL low_balance
  END IF;
END

SQL Routines (cont.)

- Routine body
  - RETURN statement allowed only inside the body of a function
    - Exception raised if function terminates not by a RETURN

CREATE FUNCTION get_balance(acct_id INT) RETURNS DECIMAL(15,2))
BEGIN
  DECLARE bal DECIMAL(15,2);
  SELECT balance INTO bal
  FROM accounts
  WHERE account_id = acct_id;
  IF bal < 100 THEN SIGNAL low_balance
  END IF;
  RETURN bal;
END
External Routines

- Parameters
  - Names are optional
  - Cannot be of any SQL data type
  - Permissible data types depend on the host language of the body

- LANGUAGE clause
  - Identifies the host language in which the body is written

- NAME clause
  - Identifies the host language code, e.g., file path in Unix
  - If unspecified, it corresponds to the routine name

```
CREATE PROCEDURE get_balance (IN acct_id INT, OUT bal DECIMAL(15,2))
  LANGUAGE C
  EXTERNAL NAME 'bank\balance_proc'

CREATE FUNCTION get_balance( IN INTEGER) RETURNS DECIMAL(15,2))
  LANGUAGE C
  EXTERNAL NAME 'usr/McKnight/banking/balance'
```

External Routines (cont.)

- RETURNS clause may specify CAST FROM clause

```
CREATE FUNCTION get_balance( IN INT)
  RETURNS DECIMAL(15,2)) CAST FROM REAL
  LANGUAGE C
```
  - C program returns a REAL value, which is then cast to DECIMAL(15,2) before returning to the caller.

- Special provisions to handle null indicators and the status of execution (SQLSTATE)
  - PARAMETER STYLE SQL (is the default)
  - PARAMETER STYLE GENERAL
External Routines (cont.)

- PARAMETER STYLE SQL
  - Additional parameters necessary for null indicators, returning function results, and returning SQLSTATE value
  - External language program (i.e., the body) has $2n+4$ parameters for procedures and $2n+6$ parameters for functions where $n$ is the number of parameters of the external routine

```sql
CREATE FUNCTION get_balance( IN INTEGER)
RETURNS DECIMAL(15,2)) CAST FROM REAL
LANGUAGE C
EXTERNAL NAME 'bank\balance'
PARAMETER STYLE SQL

void balance (int* acct_id,
      float* rtn_val,
      int* acct_id_ind,
      int* rtn_ind,
      char* sqlstate[6],
      char* rtn_name [512],
      char* spc_name [512],
      char* msg_text[512])
{
    ...
}
```

External Routines (cont.)

- PARAMETER STYLE GENERAL
  - No additional parameters
  - External language program (i.e., the body) must have exactly the same number of parameters
  - Cannot deal with null values
  - Exception is raised if any of the arguments evaluate to null
  - Value returned in an implementation-dependent manner

```sql
CREATE FUNCTION get_balance( IN INTEGER)
RETURNS DECIMAL(15,2)) CAST FROM REAL
LANGUAGE C
EXTERNAL NAME 'bank\balance'
PARAMETER STYLE GENERAL

float* balance (int* acct_id)
{
    ...
}
```
Routine Characteristics

- DETERMINISTIC or NOT DETERMINISTIC
  - DETERMINISTIC (default)
    - Routine is expected to return the same result/output values for a given list of input values. (However, no checks are done at run time.)
  - NOT DETERMINISTIC routines not allowed in
    - Constraint definitions
    - Assertions
    - In the condition part of CASE expressions
    - CASE statements

- RETURNS NULL ON NULL INPUT or CALLED ON NULL INPUT (default)
  - RETURNS NULL ON NULL INPUT
    - An invocation returns null result/output value if any of the input values is null without executing the routine body

- DYNAMIC RESULT SETS <unsigned integer>
  - Valid on procedures only (SQL or external)
  - Defined number of result sets that the procedure is allowed to return
  - If unspecified, DYNAMIC RESULT SETS 0 is implicit

Routine Characteristics (cont.)

- CONTAINS SQL, READS SQL DATA, or MODIFIES SQL DATA
  - External routines may in addition specify NO SQL
  - CONTAINS SQL (default)
  - For SQL routines -- check may be done at routine creation time
  - For both SQL and external routines -- exception raised if a routine attempts to perform actions that violate the specified characteristic
  - Routines with MODIFIES SQL DATA not allowed in
    - Constraint definitions
    - Assertions
    - Query expressions other than table value constructors
    - Triggered actions of BEFORE triggers
    - Condition part of CASE expressions
    - CASE statements
    - searched delete statements
    - search condition of searched update statements (are allowed in SET clause)
Privilege Requirements

- **SQL routine**
  - Creator must have all the privileges required for execution of the routine body
  - Creator gets the EXECUTE privilege on the routine automatically
    - GRANT OPTION on EXECUTE privilege given if creator has GRANT OPTION on all the privileges required for execution of the routine body
    - Creator loses the GRANT OPTION if at any time he/she loses any of the privileges required for successful execution of the routine body
  - Routine is dropped if at any time the creator loses any of the privileges required for execution of the routine body (in CASCADE mode)

- **External routine**
  - Creator gets the EXECUTE privilege with GRANT OPTION on the routine automatically

Routine Overloading

- **Overloading -- multiple routines with the same unqualified name**

  S1.F (p1 INT, p2 REAL)
  S1.F (p1 REAL, p2 INT)
  S2.F (p1 INT, p2 REAL)

- **Within the same schema**
  - Every overloaded routine must have a unique signature, i.e., different number of parameters or different types for the same parameters

  S1.F (p1 INT, p2 REAL)
  S1.F (p1 REAL, p2 INT)

- **Across schemas**
  - Overloaded routines may have the same signature

  S1.F (p1 INT, p2 REAL)
  S2.F (p1 INT, p2 REAL)

- **Only functions can be overloaded. Procedures cannot be overloaded.**
**Specific Names**

- Uniquely identifies each routine in the database
  - If unspecified, an implementation-dependent name is generated.

```sql
CREATE FUNCTION get_balance(acct_id INTEGER)
RETURNS DECIMAL(15,2)
SPECIFIC func1
BEGIN
...
RETURN ...;
END
```

- Can only be used to identify the routine in ALTER, DROP, GRANT, and REVOKE statements
  ```sql
  DROP SPECIFIC FUNCTION func1 RESTRICT;
  ```

- DDL statements can also identify a routine by providing the name and the list of parameter types
  ```sql
  DROP FUNCTION get_balance(INTEGER) CASCADE;
  ```

- Cannot be used to invoke a routine

**Routine Invocation**

- Procedure -- invoked by a CALL statement:
  ```sql
  CALL get_balance (100, bal);
  ```

- Function -- invoked as part of an expression:
  ```sql
  SELECT account_id, get_balance (account_id)
  FROM accounts
  ```

- Requires the invoker to have EXECUTE privilege on the routine -- otherwise no routine will be found for the invocation
  - It is not an authorization violation!!!
Subject Routine Determination

- Decides the function to invoke for a given invocation based on the
  - Compile-time data types of all arguments
  - Type precedence list of the data types of the arguments
  - SQL path
- Always succeeds in finding a unique subject function, if one exists.

- Type precedence list is a list of data type names
  - Predefined types -- defined by the standard based on increasing precision/length
    - SMALLINT: SMALLINT, INTEGER, DECIMAL, NUMERIC, REAL, FLOAT, DOUBLE
    - CHAR: CHAR, VARCHAR, CLOB
  - User-defined types is determined by the subtype-supertype relationship
    - If B is a subtype of A and C is a subtype of B, then the type precedence list for C is (C, B, A).

Path is a list of schema names.

- Can be specified during the creation of a schema, SQL-client module, or a SQL-server module

CREATE SCHEMA schema5
PATH schema1,schema3
...;

- Every session has a default path, which can be changed using the SET statement.

SET PATH 'schema1, schema2'
1. Determine the set of candidate functions for a given function invocation, F(a1, a2, ..., an):
   - Every function contained in S1 that has name F and has n parameters if the function name is fully qualified, i.e., the function invocation is of the form S1.F(a1, a2, ..., an), where S1 is a schema name.
   - Every function in every schema of the applicable path that has name F and has n parameters if the function name is not fully qualified.

2. Eliminate unsuitable candidate functions
   a. The invoker has no EXECUTE privilege
   b. The data type of i-th parameter of the function is not in the type precedence list of the static type of the i-th argument (for parameter)

3. Select the best match from the remaining functions
   a. Examine the type of the 1st parameter of each function and keep only those functions such that the type of their 1st parameter matches best the static type of the 1st argument (i.e., occurs earliest in the type precedence list of the static type of the argument), and eliminate the rest.
   b. Repeat Step b for the 2nd and subsequent parameters. Stop whenever there is only one function remaining or all parameters are considered.

4. Select the "subject function"
   a. From the remaining functions take the one whose schema appears first in the applicable path (if there is only one function, then it is the "subject function")

Assume Y is a subtype of X. Assume the following three functions (with specific names F1, F2, and F3):

- F1: F(p1 X, p2 Y)
- F2: F(p1 Y, p2 Y)
- F3: F(p1 X, p2 REAL)

The subject function for F(y,y) where the static type of y is Y is F2.

Now, assume the following three functions (with specific names F4, F5, and F6):

- F4: F(p1 X, p2 Y)
- F5: F(p1 X, p2 X)
- F6: F(p1 X, p2 REAL)

The subject function for F(y,y) where the static type of y is Y is F4.
Dropping Routines

- Routines can be dropped using DROP statement.

```
DROP FUNCTION get_balance(INTEGER) CASCADE;
DROP FUNCTION get_balance(INTEGER) RESTRICT;
```

- Normal RESTRICT/CASCADE semantics applies with respect to dependent objects:
  - Routines
  - Views
  - Constraints
  - Triggers

Altering Routines

- Routines can be altered with ALTER statement.
- Allowed only for external routines and for the following routine characteristics:
  - Language
  - Parameter style
  - SQL data access indication
  - Null behavior
  - Dynamic result set specification
  - NAME clause

```
ALTER FUNCTION get_balance (INTEGER)
READS SQL DATA
RESTRICT
```

- RESTRICT is the only allowed option, i.e., a routine cannot be altered if there are any dependent objects.
Object-Relational Support

- The queries have changed
  - How many programmers with skills in SQL and Objects are working on the most profitable product?
  - How many accidents happened within 0.2 miles from highway exits which damaged the front bumpers of red cars?
  - Tell me the sales regions in which my top 5 products had a sales drop of more than 10%.
  - Give me the marketing campaigns that used images of sunny beaches with white sands.

Object-Relational Support: Motivation

- Database systems provide
  - A set of types used to represent the data in the application domain
  - A set of operations (functions) to manipulate these types

<table>
<thead>
<tr>
<th>TYPE</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER</td>
<td>+, -, /, *, ...</td>
</tr>
<tr>
<td>CHAR</td>
<td>SUBSTRING, CONCAT, ...</td>
</tr>
<tr>
<td>DATE</td>
<td>DAY, MONTH, YEAR, ...</td>
</tr>
</tbody>
</table>

- Increasing need for extension
  - New types required to better represent the application domain
  - New operations (functions) required to better reflect the behavior of the types

<table>
<thead>
<tr>
<th>TYPE</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>MONEY</td>
<td>+, -, INTEREST, ...</td>
</tr>
</tbody>
</table>
| CHAR   | CONTAINS, SPELLCHECKING, ...
| IMAGE  | WIDTH, HEIGHT, THUMBNAIOL, ... |
Major Extensions in SQL99

- Mechanism for "users" to extend the database with application "objects" (specific types and their behavior - functions/methods)
  - User Defined Types (UDTs): Text, Image, CAD/CAM Drawing, Video ...
  - User Defined Functions (UDFs): Contains, Display, Rotate, Play, ...
- Support for storage/manipulation of large data types
  - Large Object Support (LOBs): Binary, Character
- Mechanism to improve the DB integrity and to allow checking of business rules inside the DBMS
  - Triggers: Auditing, Cross-Referencing, Alerts ...
- Means to express complex data relationships such as hierarchies, bills-of-material, travel planning ...
  - Recursion
  - Update through UNION and JOIN
  - Common Table Expressions

Upward compatible extension of SQL to guarantee application portability and database independence!

Object-Relational Support

- Large Objects (LOBs)
  - Binary
  - Character
- User-Defined Data Types
  - Distinct types
  - Structured types
- Type Constructors
  - Row types
  - Reference types
- Collection Types
  - Arrays
- User-Defined Methods, Functions, and Procedures
- Typed tables and views
  - Table hierarchies
  - View hierarchies (object views)
**What are Large Objects (LOBs)?**

- **LOBs** are a new set of data types
  - LOBs store strings of up to gigabytes
- There are 2 new data types
  - BLOB - Binary Large Object
    - Useful for Audio, Image data
  - CLOB - Character Large Object
    - For character data

<table>
<thead>
<tr>
<th>Object</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check Image</td>
<td>45K</td>
</tr>
<tr>
<td>Text</td>
<td>30-40 K/page</td>
</tr>
<tr>
<td>Small Image</td>
<td>30-40K</td>
</tr>
<tr>
<td>Large Image</td>
<td>200K-3M</td>
</tr>
<tr>
<td>Color Image</td>
<td>20-40M</td>
</tr>
<tr>
<td>Radiology Image</td>
<td>40-60M</td>
</tr>
<tr>
<td>Video</td>
<td>1G/Hour</td>
</tr>
<tr>
<td>High Res Video</td>
<td>3G/Hour</td>
</tr>
<tr>
<td>High Definition TV</td>
<td>200M/sec</td>
</tr>
</tbody>
</table>

**Large Object Data Types**

- Maintained directly in the database
- Not in "external files"
- LOB size can be specified at column definition time (in terms of KB, MB, or GB)

```
CREATE TABLE Booktable
(title VARCHAR(200),
book_id INTEGER,
summary CLOB(32K),
book_text CLOB(20M),
movie BLOB(2G))
```
How do you Use LOBS?

- LOBs may be retrieved, inserted, updated like any other type
  - You must acquire buffers large enough to store the LOBs
  - This may be difficult for very large LOBs

EXEC SQL
SELECT summary, book_text, movie
INTO :bigbuf, :biggerbuf, :massivebuf
FROM BOOKTABLE
WHERE title = 'Moby Dick';
Large Object Data Types

- LOBs are excluded from some operations:
  - Greater Than and Less Than operations
  - Primary, unique, and foreign keys
  - GROUP BY and ORDER BY
  - UNION operator, INTERSECT, EXCEPT
  - Joins (as join columns)

- Some operations are supported for LOBs
  - Retrieve value (or partial value)
  - Replace value
  - LIKE predicate
  - Concatenation
  - SUBSTRING, POSITION, and LENGTH function
  - TRIM
  - OVERLAY

LOB Functions

- Functions that support LOBs
  - CONCATENATION `string1 || string2`
  - SUBSTRING(`string FROM start FOR length`)
  - LENGTH(`expression`)
  - POSITION(`search-string IN source-string`)
  - NULLIF/COALESCE
  - TRIM
  - OVERLAY
  - Cast
  - User-defined functions
  - LIKE predicate

EXEC SQL
SELECT position('Chapter 1' IN book_text) INTO :int_variable
FROM BOOKTABLE
WHERE title = 'Moby Dick';
LOBs may be unmanageable in application programs
  - Huge amounts of storage may be needed to buffer their values
    - It may not be possible to acquire contiguous buffers of sufficient size
  - Applications may want to deal with LOBS a piece at a time
    - In the above example, multiple SELECTs would be required

SQL99 provides locators to make LOB access manageable

```sql
EXEC SQL
SELECT summary, book_text, movie
  INTO :bigbuf, :biggerbuf, :massivebuf
FROM BOOKTABLE
WHERE title = 'Moby Dick';
```

**BOOKTABLE:**
- title VARCHAR(200)
- book_id ROWID
- summary CLOB(32K)
- book_text CLOB(20M)
- movie BLOB(2G)

**Locators**
- locator: 4-byte value stored in a host variable that a program can use to refer to a LOB value
  - Application declares locator variable, and then may set it to refer to the current value of a particular LOB
  - A locator may be used anywhere a LOB value can be used

```sql
EXEC SQL BEGIN DECLARE SECTION;
SQL TYPE IS BLOB_LOCATOR
  movie_loc;
EXEC SQL END DECLARE SECTION;
EXEC SQL
SELECT movie
  INTO :movie_loc
FROM BOOKTABLE
WHERE title = 'Moby Dick';
```
Locators may also represent LOB expressions

- A LOB expression is any expression that refers to a LOB column or results in a LOB data type
  - LOB functions may be part of LOB expressions
  - LOB expressions may even reference other locators
  - LOB expressions may be VERY complicated
- The above example associates only the first chapter with a locator

```
SELECT SUBSTRING(book_text, 
                      POSITION('Chapter 1' IN book_text), 
                      POSITION('Chapter 2' IN book_text) - 
                      POSITION('Chapter 1' IN book_text) 
                  )
FROM Booktable
INTO :Chapt1Loc
WHERE title = 'Moby Dick';
```

Locators can be holdable

- HOLD locator
  - Maintains the LOB value and locator after the commit of a transaction
- FREE locator
  - Frees a locator and its LOB value

```
SELECT book_text
INTO :LOB_locator
FROM Booktable WHERE title = 'Moby Dick';
HOLD LOCATOR :LOB_locator;
COMMIT;
INSERT INTO my_favor_books
VALUES (...,:LOB_locator,...)
```
HOLD LOCATOR Statement

- Locators, when created, are marked valid. A valid locator normally becomes invalid at the end of transaction (when COMMIT or ROLLBACK happens).
- HOLD LOCATOR statement marks a host variable or host parameter locator as holdable:

```
HOLD LOCATOR :emp;
```

- Holdable locators remain valid across transaction boundaries that end successfully.
- Not allowed for parameters or result of external routines.

FREE LOCATOR Statement

- FREE LOCATOR statement marks a valid host variable or host parameter locator as invalid:

```
FREE LOCATOR :emp;
```

- All valid locators are marked invalid if the transaction ends with ROLLBACK statement.
User-defined types

- User-defined data types
  - User-defined, named type representing entities
    - employee, project, money, polygon, image, text, language, format, ...

- User-defined methods and functions (operators)
  - User-defined operation representing the behavior of entities in the application domain
    - hire, appraisal, convert, area, length, contains, ranking, ...

Definition:
- User-defined data type
  - Name
  - Representation
  - Relationship to other types

- User-defined method (and function)
  - Name
  - Signature (i.e., parameter list)
  - Result
  - Implementation

User-defined Types: Key Features

- New functionality
  - Users can indefinitely increase the set of provided types
  - Users can indefinitely increase the set of operations on types and extend SQL to automate complex operations/calculations

- Flexibility
  - Users can specify any semantics and behavior for a new type

- Consistency
  - Strong typing insures that functions are applied on correct types

- Encapsulation
  - Applications do not depend on the internal representation of the type

- Performance
  - Potential to integrate types and functions into the DBMS as “first class citizens”
User-defined Types: Benefits

- Simplified application development
  - Code Re-use - allows reuse of common code
  - Overloading and overriding - makes application development easier -- single function name for a set of operations on different types, e.g., area of circles, triangles, and rectangles

- Consistency
  - Enables definition of standard, reusable code shared by all applications (guarantee consistency across all applications using type/function)

- Easier application maintenance
  - Changes are isolated: if application model changes, only the corresponding types/functions need to change instead of code in each application program

User-defined Distinct Types

```
CREATE TABLE RoomTable (RoomID CHAR(10), RoomLength INTEGER, RoomWidth INTEGER, RoomArea INTEGER, RoomPerimeter INTEGER);
```

```
UPDATE RoomTable
SET RoomArea = RoomLength;
```

- No Error Results

Before SQL99, columns could only be defined with the existing built-in data types
  - There was no strong typing
  - Logically incompatible variables could be assigned to each other
User-defined Distinct Types

Each UDT is logically incompatible with all other type

Based on name equivalence (strongly typed)
- Renamed type, with different behavior than its source type.
  - Shares internal representation with its source type
  - Source and distinct type are not directly comparable

CREATE TYPE US_DOLLAR AS DECIMAL (9,2) FINAL

Operations defined on distinct types (behavior)
- Comparison operators
  - Can be defined based on the comparison of their source type
- Casting
  - Used to explicitly cast instances of the distinct type and instances of source type to and from one another
  - Used to obtain "literals"
- Methods and functions
- No inheritance or subtyping
User-defined Structured Types

- User-defined, complex data types
  - Can be used as column types and/or table types
- Column Types
  - E.g., text, image, audio, video, time series, point, line,...
  - For modeling new kinds of facts about enterprise entities
  - Enhanced infrastructure for SQL/MM
- Row Types
  - Types and functions for rows of tables
    - E.g., employees, departments, universities, students, ...
    - For modeling entities with relationships & behavior
  - Enhanced infrastructure for business objects

CREATE TYPE employee
AS
(id INTEGER,
 name VARCHAR (20))

CREATE TYPE address AS
(street CHAR (30),
city CHAR (20),
state CHAR (2),
zip INTEGER) NOT FINAL

CREATE TYPE real_estate AS
(owner REF (person),
price money,
rooms INTEGER,
size DECIMAL(8,2),
location address,
text_description text,
front_view_image bitmap,
document doc) NOT FINAL

Structured Types: Example
Use of Structured Types

- Wherever other (predefined data) types can be used in SQL
  - Type of attributes of other structured types
  - Type of parameters of functions, methods, and procedures
  - Type of SQL variables
  - Type of domains or columns in tables

CREATE TYPE address AS (street CHAR (30), ...) NOT FINAL
CREATE TYPE real_estate AS (... location address, ...) NOT FINAL

- To define tables and views

CREATE TABLE properties OF real_estate ...

Methods

- What are methods?
  - SQL-invoked functions "attached" to user-defined types

- How are they different from functions?
  - Implicit SELF parameter (called subject parameter)
  - Two-step creation process: signature and body specified separately.
  - Must be created in the type’s schema
  - Different style of invocation (UDT value.method(...))

CREATE TYPE employee AS
(name CHAR(40),
base_salary DECIMAL(9,2),
bonus DECIMAL(9,2))
INSTANTIABLE NOT FINAL
METHOD salary() RETURNS DECIMAL(9,2);

CREATE METHOD salary() FOR employee
BEGIN
....
END;
Methods (cont.)

- Two kinds of methods:
  - **Original** methods: methods attached to super type
  - **Overriding** methods: methods attached to subtypes

```sql
CREATE TYPE employee AS
    (name CHAR(40),
    base_salary DECIMAL(9,2),
    bonus DECIMAL(9,2))
INSTANTIABLE NOT FINAL
METHOD salary() RETURNS DECIMAL(9,2);
```

```sql
CREATE TYPE manager UNDER employee AS
    (stock_option INTEGER)
INSTANTIABLE NOT FINAL
OVERRIDING METHOD salary() RETURNS DECIMAL(9,2), -- overriding
METHOD vested() RETURNS INTEGER -- original;
```

- Signature of an overriding method must match with the signature of an original method, except for the subject parameter.

Methods (cont.)

- Invoked using dot syntax (assume dept table has mgr column):

```sql
SELECT mgr.salary() FROM dept;
```

- Subject routine determination picks the "best" method to invoke.
  - Same algorithm as used for regular functions
  - SQL path is temporarily set to a list with the schemas of the supertypes of the static type of the self argument.

- Dynamic dispatch executed at runtime
  - Overriding methods considered at execution time
  - Overriding method with the best match for the dynamic type of the self argument is selected.
  - Schema evolution affects the actual method that gets invoked. If there is a new overriding method defined it may be picked for execution.
Creating Structured Types

- System-supplied constructor function
  - address () -> address or real_estate () -> real_estate
    - Returns new instance with attributes initialized to their default
- NEW operator
  - NEW <method name> <list of parameters>
    - Invokes constructor function before invoking method
- INSERT statement against a typed table

CREATE TABLE properties OF real_estate ...  

INSERT INTO properties VALUES (:owner, money (350000), 15, 4500, NEW address ('1543 3rd Ave. North, Sacramento, CA 93523') ...)

SELECT owner, price FROM properties  
WHERE address = gen_address (address(), '1543 3rd Ave. North, Sacramento, CA 93523')

Uninstantiable Types

- Structured types can be uninstantiable
  - Like abstract classes in OO languages
  - No system-supplied constructor function is generated
  - Type does not have instances of its own
  - Instances can be defined on subtypes
- By default, structured types are instantiable
- Distinct types are always instantiateable

CREATE TYPE person AS
(name VARCHAR (30),
address address,
sex CHAR (1)) NOT INSTANTIABLE NOT FINAL
Manipulating Attributes

- Observer and mutator methods are used to access and modify attributes
  - Automatically generated when type is defined

```sql
CREATE TYPE address AS (street CHAR (30), city CHAR (20), state CHAR (2), zip INTEGER) NOT FINAL

  address_expression.street () -> CHAR (30)
  address_expression.city () -> CHAR (20)
  address_expression.state () -> CHAR (2)
  address_expression.zip () -> INTEGER
  address_expression.street (CHAR (30)) -> address
  address_expression.city (CHAR (20)) -> address
  address_expression.state (CHAR (2)) -> address
  address_expression.zip (INTEGER) -> address

SELECT location.street, location.city (), location.state, location.zip ()
FROM properties
WHERE price < 100000
```

Manipulating Attributes

- Queries over type tables access attributes (columns)
- Update statements on typed tables modify attributes

```sql
CREATE TABLE properties OF real_estate ...

SELECT owner, price
FROM properties
WHERE address = NEW address '1543 3rd Ave. North, Sacramento, CA 93523'

UPDATE properties
SET price = 350000
WHERE address = new address '1543 3rd Ave. North, Sacramento, CA 93523'
```
Dot Notation

- "Dot" notation must be used to invoke methods (e.g., to access attributes)
- Methods without parameters do not require use of "()

DECLARE r real_estate;
...
SET r.size = 2540.50; -- same as r.size(2540.50)
...
SET ... = r.location.state; -- same as r.location().state()
SET r.location.city = 'LA'; -- same as r.location(r.location.city('LA'))

- Support for several 'levels' of dot notation (a.b.c.d.e)
- Allow "navigational" access to structured types
- Dot notation does not 'reveal' physical representation (keeps encapsulation)

Initializing Instances: Constructor

- Instances are generated by the system-provided constructor function
  - Attributes are initialized with their default values
- Attributes are modified (further initialized) by invoking the mutator functions

BEGIN
  DECLARE re real_estate;
  SET re = real_estate(); -- generation of a new instance
  SET re.rooms = 12; -- initialization of attribute rooms
  SET re.size = 2500; -- initialization of attribute size
END

BEGIN
  DECLARE re real_estate;
  SET re = real_estate().rooms(12).size(2500); -- same as above
END
Initializing Instances: NEW Operator

- Users can define any number of "initializer" methods and invoke them with NEW operator

```sql
CREATE TYPE real_estate AS ( ....)
METHOD real_estate (r INTEGER, s DECIMAL(8,2)) RETURNS real_estate

CREATE METHOD real_estate (r INTEGER, s DECIMAL(8,2)) RETURNS real_estate
BEGIN
    SET self.rooms = r;
    SET self.size = s;
    RETURN re;
END

BEGIN
    DECLARE re real_estate;
    SET re = NEW real_estate(12, (2500);  -- same as previously
END
```

Manipulating Structured Types

- Structured types are manipulated by invoking methods and functions defined on them
  - May be invoked anywhere scalar values are allowed in SQL

```sql
INSERT INTO properties
VALUES (:owner, us_dollar (300000), 15, 4650, NEW address
('2225 Coral Drive', 'San Jose', 'CA', 95125), ...);

UPDATE properties
SET price = US_dollar (0.9 x amount (price))
WHERE location.state () = 'CA';

SELECT D_mark (price), owner, location.city ()
FROM properties
WHERE location.zip () = 95453
AND contains (text_description.schools(), 'excellent school
district')
AND contains (front_view_image, 'tree');
```
Subtyping and Inheritance

- Structured types can be a subtype of another UDT
- UDTs inherit structure (attributes) and behavior (methods) from their supertypes
  - Single inheritance (multiple inheritance moved to SQL4)
- FINAL and NOT FINAL
  - FINAL types may not have subtypes
    - In SQL99, structured types must be NOT FINAL and distinct types must be FINAL
    - In SQL4, both options will be allowed

CREATE TYPE real_estate ... NOT FINAL
CREATE TYPE condo UNDER real_estate ... NOT FINAL
CREATE TYPE house UNDER real_estate ... NOT FINAL

CREATE TYPE address AS
(street CHAR (30), city CHAR(20), state CHAR (2), zip INTEGER) NOT FINAL

CREATE TYPE german_addr UNDER address
(family_name VARCHAR (30) ) NOT FINAL

CREATE TYPE brazilian_addr UNDER address
(neighborhood VARCHAR (30) ) NOT FINAL

CREATE TYPE us_addr UNDER address
(area_code INTEGER, phone INTEGER) NOT FINAL

CREATE TYPE us_bus_addr UNDER address
(bus_area_code INTEGER, bus_phone INTEGER) NOT FINAL
Value Substitutability

■ Each row can have a value a different subtype

```sql
INSERT INTO properties (price, owner, location)
VALUES (US_dollar (100000), REF('Mr.S.White'), NEW us_addr ('1654 Heath Road', 'Heath', 'OH', 45394, ... ) )

INSERT INTO properties (price, owner, location)
VALUES (real (400000), REF('Mr.W.Green'), NEW brazilian_addr ('245 Cons. Xavier da Costa', 'Rio de Janeiro', 'Copacabana') )

INSERT INTO properties (price, owner, location)
VALUES (german_mark (150000), REF('Mrs.D.Black'), NEW german_addr ('305 Kurt-Schumacher Strasse', 'Kaiserslautern', 'Prof. Dr. Heuser') )
```

<table>
<thead>
<tr>
<th>price</th>
<th>owner</th>
<th>location</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;us_dollar&gt; amount 100,000</td>
<td>'Mr. S. White'</td>
<td>&lt;us_addr&gt; '1654 Heath...'</td>
</tr>
<tr>
<td>&lt;real&gt; amount 400,000</td>
<td>'Mr. W. Green'</td>
<td>&lt;brazilian_addr&gt; '245 Cons. Xavier...'</td>
</tr>
<tr>
<td>&lt;german_mark&gt; amount 150,000</td>
<td>'Mrs. D. Black'</td>
<td>&lt;german_addr&gt; '305 Kurt-Schumacher...'</td>
</tr>
</tbody>
</table>

An instance of a subtype can be found at runtime (requires dynamic dispatch - late binding)

```sql
SELECT owner, price.dollar_amount ( )
FROM properties
WHERE price.dollar_amount ( ) < US_dollar (500000)
```

- Will cause the invocation of a different method, depending on the type of money stored in the column PRICE (i.e., US_dollar, CDN_dollar, D_mark, S_frank, real, ...)
- Only methods are dynamically dispatched
  - Functions are statically selected
Structured Types as Column Types

1. CREATE TYPE envelope (xmin INTEGER, ymin INTEGER, xmax INTEGER, ymax INTEGER);
2. CREATE TYPE geometry (gtype INTEGER, refsystem INTEGER, tolerance FLOAT, area FLOAT, length FLOAT, mbr envelope, numparts INTEGER, numpoints INTEGER, points points, zvalue BLOB(500k), measure BLOB(500k));
3. CREATE TYPE point UNDER geometry;
4. CREATE TYPE line UNDER geometry;
5. CREATE TYPE polygon UNDER geometry;

CREATE FUNCTION distance (s1 geometry, s2 geometry) RETURNS BOOLEAN EXTERNAL NAME '/usr/lpp/db2se/gis!shapedist' ...

CREATE FUNCTION within (s1 geometry, s2 geometry) RETURNS BOOLEAN EXTERNAL NAME '/usr/lpp/db2se/gis!shapewithin' ...

5. CREATE TABLE customers (cid INTEGER, name VARCHAR(20), income INTEGER, addr CHAR(20) loc point);

CREATE TABLE stores (sid INTEGER, name VARCHAR(20), addr CHAR(20), loc point, zone polygon);

CREATE TABLE sales (sid INTEGER, cid INTEGER, amount INTEGER);
Structured Types as Column Types

SELECT * FROM stores s, customers c
WHERE within(c.loc, s.zone)=1
or distance(c.loc, s.loc)<100
ORDER BY s.name, c.name;

"Tell me all the information I have about each customer who either lives within a stores' zone or within 100 miles of the store."

Structured Types as Row Types: Typed Tables

Structured types can be used to define typed tables
- Attributes of type become columns of table
- Plus one column to define REF value for the row (object id)

CREATE TYPE real_estate AS
(owner REF (person),
price money,
rooms INTEGER,
size DECIMAL(8,2),
location address,
text_description text,
front_view_image bitmap,
document doc) NOT FINAL

CREATE TABLE properties OF real_estate
(REF IS oid USER GENERATED)
Reference Types

- Structured types have a corresponding reference type
  - Can be used wherever other types can be used

- Representation
  - User generated (REF USING <predefined type>)
  - System generated (REF IS SYSTEM GENERATED)
  - Derived from a list of attributes (REF (<list of attributes>))
    - Default is system generated

CREATE TYPE real_estate AS (owner REF (person), ...)
NOT FINAL REF USING INTEGER

CREATE TYPE person AS (ssn INTEGER, name CHAR(30),...)
NOT FINAL REF (ssn)

Reference Types

- Reference values can be scoped
  - Only scoped ones can be dereferenced

CREATE TYPE person (ssn INTEGER, name ...) NOT FINAL

CREATE TYPE real_estate (owner REF (person), ...) NOT FINAL

CREATE TABLE people OF person ( ...)
CREATE TABLE properties OF real_estate (owner WITH OPTIONS SCOPE people)
**Reference Types**

- References do not have the same semantics as referential constraints

  ```sql
  CREATE TABLE T1
  (C1 REAL PRIMARY KEY, ...)
  CREATE TABLE T2
  (C2 DECIMAL (7,2) PRIMARY KEY, ...)
  CREATE TABLE T
  (C INTEGER, ...
  FOREIGN KEY (C) REFERENCES T1 (C1) NO ACTION,
  FOREIGN KEY (C) REFERENCES T2 (C2) NO ACTION)
  ```

- Referential constraints specify inclusion dependencies
  - It is unclear which table to access during dereferencing
  - There is no notion of strong typing

**Subtables: Table Hierarchies**

- Typed tables can have subtables
  - Inherit columns, constraints, triggers, ... the supertable

  ```sql
  CREATE TYPE person ... NOT FINAL
  CREATE TYPE real_estate ... NOT FINAL
  CREATE TYPE condo UNDER real_estate ... NOT FINAL
  CREATE TYPE house UNDER real_estate ... NOT FINAL
  ```

  ```sql
  CREATE TABLE people OF person ( ...)
  CREATE TABLE properties OF real_estate
  CREATE TABLE condos OF condo UNDER properties
  CREATE TABLE houses OF house UNDER properties
  ```
Substitutability

- Queries on table hierarchies range over the rows of every subtable

SELECT price, location.city, location.state FROM properties
WHERE contains (text_description, 'excellent school district')
  ▷ Returns properties, condos, and houses

- Queries on a subtable require SELECT privilege on that subtable
  SELECT * FROM condos...

- Additional authorization required for queries that involve ONLY, or DEREF on self-referencing column....

Substitutability: Type Predicate and ONLY on Typed Tables

- Type predicate can be used to restrict selected rows

SELECT price, location.city, location.state FROM properties
WHERE contains (text_description, 'excellent school district')
AND DEREF (oid) IS OF (house)

- ONLY restricts selected rows to rows whose most specific type is the type of the typed table

SELECT price, location.city, location.state FROM ONLY (properties)
WHERE contains (text_description, 'excellent school district')
  ▷ Queries on the target typed table that involve the ONLY modifier (or the DEREF operation on its self-referencing column) require WITH HIERARCHY OPTION on that target table.

GRANT SELECT WITH HIERARCHY OPTION ON TABLE properties TO PUBLIC
Path Expressions - <dereference operator>

- Scoped references can be used in path expressions

```sql
SELECT prop.price, prop.owner->name FROM properties.prop
WHERE prop.owner->address.city = "Hollywood"
```

- Authorization checking follows SQL authorization model
  - user must have SELECT privilege on name and address

```sql
SELECT prop.price, (SELECT name FROM people p WHERE p.oid = prop.owner)
FROM properties.prop
WHERE (SELECT p.address.city FROM people p WHERE p.oid = owner) = "Hollywood"
```

```sql
SELECT prop.price, p.name
FROM properties prop LEFT JOIN people p ON (prop.owner = p.oid)
WHERE p.address.city = "Hollywood"
```

Method Reference

- References can be used to invoked methods on the corresponding structured type

```sql
SELECT prop.price, prop.owner->income (1998)
FROM properties.prop
```

- Invocation of methods given a reference value require select privilege on the method for the target typed table

```sql
GRANT SELECT (METHOD income FOR person) ON TABLE people TO PUBLIC
```

- Allows the table owner control who is authorized to invoked methods on the rows of his/her table
Reference Resolution: Nesting

- References can be used to obtain the structured type value that is being referenced
  - Enables nesting of structured types

SELECT prop.price, DEREF(prop.owner)
FROM properties.prop

- Reference resolution requires SELECT privilege WITH HIERARCHY OPTION on the target typed table

GRANT SELECT WITH HIERARCHY OPTION ON TABLE people TO PUBLIC

- DEREF nests rows from subtables, respecting value substitutability

Object Views

- Views have been extended to support
  - Typed views
  - View hierarchies
  - References on base tables can be mapped to references on views
**Object Views: Example**

CREATE TYPE propViewType AS (owner REF (person), location address) NOT FINAL

CREATE TYPE condViewType UNDER propViewType ...
CREATE TYPE housViewType UNDER propViewType ...

CREATE VIEW propView OF propViewType REF IS propID USER GENERATED (owner WITH OPTIONS SCOPE peopleView) AS (SELECT owner, location FROM ONLY (properties) )

CREATE VIEW housView OF housViewType UNDER propView AS (SELECT owner, location FROM ONLY (houses) )

CREATE VIEW condView OF condViewType UNDER propView AS (SELECT owner, location FROM ONLY (condos) )

**Comparison of UDT Values**

- CREATE ORDERING statement specifies
  - Which comparison operations are allowed for a user-defined type
  - How such comparisons are to be performed.

CREATE ORDERING FOR employee EQUALS ONLY BY STATE;

CREATE ORDERING FOR complex ORDER FULL BY RELATIVE WITH FUNCTION complex_order (complex,complex);

- ORDERING form:
  - EQUALS ONLY
    - Only comparison operations allowed are =, <>
  - ORDER FULL
    - All comparison operations are allowed
Comparison of UDT Values (cont.)

- Ordering category
  - STATE
    - An ordering function is implicitly created with two UDT parameters and returning Boolean
    - Compares pairwise the UDT attributes
  - RELATIVE
    - User must specify an ordering function with two UDT parameters and returning INTEGER
    - 0 for equal, positive for >, negative for <
  - MAP
    - User must specify an ordering function with one UDT parameter and returning a value of a predefined type
    - Comparisons are made based on the value of the predefined type

Comparison of UDT Values (cont.)

- Ordering category - Rules:
  - STATE cannot be specified for distinct types.
  - STATE and RELATIVE must be specified for the maximal supertype in a type hierarchy.
  - MAP can be specified for more than one type in a type hierarchy, but all such types must specify MAP and all such types must have the same ordering form.
  - STATE is allowed only for EQUALS ONLY.
  - If ORDER FULL is specified, then RELATIVE or MAP must be specified.
Comparison of UDT Values (cont.)

- Comparison type of a given type:
  - The nearest supertype for which a comparison was defined.
  - Comparison form, comparison category, and comparison function of a type are the ordering form, ordering category, ordering function of its comparison type.

- A value of type T1 is comparable to a value of type T2 if
  - T1 and T2 are in the same subtype family.
  - Comparison types of T1 and T2 both specify the same comparison category (i.e., STATE, RELATIVE, or MAP)

Example
- Person has subtypes: emp and mgr
- Person has an ordering form, ordering category, and an ordering function
- emp and mgr types have none
- Person is the comparison type of emp and mgr
- Two emp values, two mgr values, or a value of emp and a value of mgr can be compared.

Comparison of UDT values (cont.)

- No comparison operations are allowed on values of structured types by default.
- All comparison operations are allowed on values of distinct types by default.
  - Based on the comparison of values of source type.
  - Whenever a distinct type is created, a CREATE ORDERING statement is implicitly executed (SDT is the source type).
  - The ordering function is the system-generated cast function

CREATE ORDERING FOR DT
ORDER FULL BY MAP WITH FUNCTION SDT(DT);
Comparison of UDT values (cont.)

- A predicate of the form "V1 = V2" is transformed into the following expression depending on the comparison category:
  - STATE
    - "SF(V1,V2) = TRUE"
    - SF is the comparison function
  - MAP
    - "MF1(V1) = MF2(V2)"
    - MF1 and MF2 are comparison functions
  - RELATIVE
    - "RF(V1,V2) = 0"
    - RF is the comparison function

Comparison of UDT Values (cont.)

- DROP ORDERING
  - Removes the ordering specification for an UDT
  
  DROP ORDERING FOR employee RESTRICT;

- RESTRICT implies
  - There cannot be any
    - SQL-invoked routine
    - View
    - Constraint
    - Assertion
    - Trigger
  that has a predicate involving employee values or values of subtypes thereof.
User-defined Casts

- Allow a value of one type to be cast into a value of another type
  - At least one of the types in an user-defined cast must be a user-defined type or a reference type.

CREATE CAST(t1 AS t2) WITH FUNCTION foo (t1);

SELECT CAST(c1 AS t2) FROM TAB1;

- May optionally be tagged AS ASSIGNMENT

CREATE CAST(t1 AS t2) WITH FUNCTION foo (t1) AS ASSIGNMENT;

  - Such casts get invoked implicitly during assignment operations.
  - Above user-defined cast makes the following assignment legal:

DECLARE v1 t1, v2 t2;
SET V2 = V1;

User-defined Casts (cont.)

- DROP CAST
  - Removes the user-defined cast
  - Does not delete the corresponding function (only its cast flag is removed)

DROP CAST (T1 AS T2) RESTRICT;

- RESTRICT implies:
  - There cannot be any
    - Routine
    - View
    - Constraint
    - Assertion
    - Trigger
  - An expression of the form "CAST(V1 AS T2)" where V1 is of type T1 or any subtype of T1;
  - A DML statement that implicitly invokes the user-defined cast function.
Cast Functions for Distinct Types

- Automatically defines cast functions to and from the source type for a user-defined distinct type
  - Casts will also be allowed from any type that is promotable to the source type of the user-defined type (i.e., that has the source type in its type precedence list)
  - Casting from a SMALLINT to a UDT sourced on an integer is OK

```sql
CREATE TYPE plan.meters AS INTEGER FINAL
CAST (SOURCE AS DISTINCT) WITH meters
CAST (DISTINCT AS SOURCE) WITH integer
```

Implicit Cast Functions created:
- `plan.meters(integer)` returns meters;
- `plan.integer(meters)` returns integer;

Example Casting Expressions:
- `... SET RoomWidth = CAST (integerCol AS meters)`
- `or meters(integerCol)`
- `or meters(smallintCol)`

---

Cast Functions: Comparison Rules

- Casts must be used to compare distinct type values with source-type values.
  - Constants are always considered to be source type values
  - You may cast from source type to UDT, or vice-versa

```sql
SELECT * FROM RoomTable
WHERE RoomID = 'Bedroom';
ERROR
SELECT * FROM RoomTable
WHERE RoomID = roomtype('Bedroom');
or
SELECT * FROM RoomTable
WHERE char(RoomID) = 'Bedroom';
No Error Results
```
Cast Functions: Assignment Rules

- In general source-type values may not be assigned to user-defined type targets.
- The strong typing associated with UDTs is relaxed for assignment operations, IF AND ONLY IF
  - A cast function between source and target type has been defined with the AS ASSIGNMENT clause.

Transforms

- Transforms are user-defined functions or methods that get invoked automatically whenever UDT values are exchanged between SQL and external programs.
- Each UDT is associated with a collection of transform groups; each transform group is associated with:
  - A from_sql function that maps a UDT value into a value of predefined type.
  - A to_sql function that maps a value of a predefined type into a UDT value.
**Transforms (cont.)**

- CREATE TRANSFORM statement specifies a transform for a given UDT

  ```sql
  CREATE TRANSFORM FOR point
    group1(FROM SQL WITH FUNCTION from_point1(point),
           TO SQL WITH FUNCTION to_point1(char(27))
    group2(FROM SQL WITH FUNCTION from_point2(point),
           TO SQL WITH FUNCTION to_point2(char(50)));
  ```

- A transform group with a given name can be specified for only one type within a type hierarchy.
- An implicit transform is created for every distinct type on its creation, based on its cast functions.

**Methods as Transform Functions**

- Both `from_sql` and `to_sql` functions can be specified as methods:

  ```sql
  CREATE TRANSFORM FOR point
    group1(FROM SQL WITH METHOD from_point1() FOR point,
           TO SQL WITH METHOD to_point1(char(27) FOR point)
    group2(FROM SQL WITH METHOD from_point2() FOR point,
           TO SQL WITH METHOD to_point2(char(50) FOR point);
  ```

- Both `from_sql` and `to_sql` methods can be overridden to define subtype-specific transform methods.
Transforms in Embedded Programs

- An embedded program can specify transform groups for use during the execution of program:
  
  TRANSFORM GROUP group1
  TRANSFORM GROUP group2 FOR TYPE point

- A host variable whose data type is a UDT must specify a predefined type; must be same as the return type of from_sql function of the transform group specified for the UDT:
  
  SQL TYPE IS point AS CHAR(50) pointvar

Transforms in Embedded Programs (cont.)

- from_sql function or method is automatically invoked on the UDT value and the result is passed to the host variable:
  
  EXEC SQL SELECT center INTO :pointvar FROM circles WHERE ...

- to_sql function or method is automatically invoked on the host variable value and the result is passed to SQL:
  
  EXEC SQL UPDATE circles
  SET center = :pointvar
  WHERE ...
An external routine can specify transform groups for use during the execution of routine:

```sql
CREATE FUNCTION foo(p1 point)
RETURNS INTEGER
EXTERNAL
TRANSFORM GROUP group1;
```

The parameter in the external program corresponding to 'p1' must specify a host language type that corresponds to `CHAR(27)`.

Transform functions for UDT parameters are picked during the creation of external routines; once selected, the transform functions are frozen.

If there is no transform available for a UDT with a given group name, then a transform defined for one of its supertypes is picked.

If transform functions are methods, the dynamic binding rules apply, i.e., if there is an overriding method available, that method is picked for execution.
**Transforms in Dynamic SQL**

- SET TRANSFORM GROUP statement sets the transform group for one or more UDTs for use during execution of dynamic SQL statements:
  
  ```sql
  SET DEFAULT TRANSFORM GROUP group1;
  SET TRANSFORM GROUP FOR TYPE point group2;
  ```

- Two special registers are provided to inquire about the session defaults:
  
  ```sql
  CURRENT_DEFAULT_TRANSFORM_GROUP;
  CURRENT_TRANSFORM_GROUP_FOR_TYPE point;
  ```

**Dropping Transforms**

- DROP TRANSFORM statement can be used to drop either a transform group or all transform groups attached to a UDT:
  
  ```sql
  DROP TRANSFORM group1 FOR point RESTRICT;
  DROP TRANSFORM ALL FOR point CASCADE;
  ```

- Dependencies between a transform group and the external routines that depend on that transform group are taken into account during dropping of transforms.
Arrays

- The only collection type of SQL99
- Why arrays?
  - Tables with collection-valued columns
    - "repeating groups"
    - n1NF tables
  - Important building block for imperative code
  - Heavily used in Standard Type Libraries
    - SQL/MM Full-Text
    - SQL/MM Spatial

Arrays (cont.)

- Array characteristics
  - Maximal length vs actual length (like CHARACTER VARYING)
  - Any element type admissible (except array types)
  - Substitutability applies at element level
  - "Arrays anywhere"
- Array operations
  - Element access by ordinal number
  - Cardinality
  - Comparison
  - Constructors
  - Assignment
  - Concatenation
  - CAST
  - Declarative selection facilities over arrays
Arrays (cont.)

- Tables with array-valued columns
  ```
  CREATE TABLE reports
  (id INTEGER,
   authors VARCHAR(15) ARRAY[20],
   title VARCHAR(100),
   abstract FullText)
  ```

- Appropriate DML operations
  ```
  INSERT INTO reports(id, authors, title)
  VALUES (10, ARRAY ['Date', 'Darwen'], 'A Guide to the SQL Standard')
  ```

Arrays (cont.)

- Access to array elements
  - By ordinal position
  - Declarative (i.e. query) facility
    - Implicitly transforms array into table
    - Selection by element content and/or position
    - Unnesting
  - Examples:
    ```
    SELECT id, authors[1] AS name FROM reports
    ```
    ```
    SELECT r.id, a.name
    FROM reports AS r, UNNEST (r.authors) AS a (name)
    ```
UDT and Array Locators

- Similar to large object locators.
- A host variable can be specified as a locator variable for a UDT or an array type:
  SQL TYPE IS point AS LOCATOR pointvar;
  SQL TYPE IS INTEGER ARRAY[10] AS LOCATOR avar;

- An unique implementation-dependent 4-octet integer locator value is generated and passed to the host variable:
  EXEC SQL SELECT center INTO :pointvar
  FROM circles WHERE ...

When locators are used in assignment statements, the UDT or the array value corresponding to the given locator value is first found, and the result is then used in the assignment:

EXEC SQL UPDATE circles
    SET center = :pointvar
    WHERE ...

A parameter of an external routine can be specified as locator parameter if its data type is either a UDT or an array type, or the returns type of an external function can specify AS LOCATOR if it is either a UDT or an array type:

CREATE FUNCTION foo(p1 emp AS LOCATOR)
    RETURNS emp AS LOCATOR
    EXTERNAL ...
When the routine is invoked, an unique implementation-dependent 4-octet integer locator value is generated for each input locator parameter and passed as the argument value.

After the routine finishes execution, for each output locator parameter or function result, the UDT or the array value corresponding to the locator value is first found, and the result is then returned to the caller.

Some examples used throughout this presentation are based on the following schema:

```
CREATE TABLE people(
    lname CHAR(20),
    fname CHAR(20),
    nick CHAR(20)
    UNIQUE (lname, fname)
);
```

```
CREATE TABLE hobbies
    (last CHAR(20),
    first CHAR(20),
    hobby CHAR(15)
);
```

```
SQL> SELECT * FROM hobbies;
```

<table>
<thead>
<tr>
<th>LAST</th>
<th>FIRST</th>
<th>HOBBY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Holland</td>
<td>Smith</td>
<td>travel</td>
</tr>
<tr>
<td>Smith</td>
<td>Roberta</td>
<td>sailing</td>
</tr>
</tbody>
</table>
Constraints

- Constraints have names
- Checking of constraints can be
  - Performed at the end of SQL statement (SQL/89)
  - Deferred until the end of transaction
- Constraints and assertions can be defined as
  - Deferrable or not deferrable
  - Initially deferred or initially immediate

```sql
ALTER TABLE people
  ADD CONSTRAINT pk UNIQUE (lname, fname) DEFERRABLE INITIALLY IMMEDIATE;
SET CONSTRAINTS pk DEFERRED;
... COMMIT;
```

- At commit time if any constraint is not satisfied, then
  - An exception is raised
  - The transaction is rolled back

Constraints (cont.)

- SQL92 has defined several kinds of constraints:
  - Primary key specification
  - NOT NULL
  - UNIQUE constraints
  - Check constraints - defined at table level
  - Assertions - defined at schema level
  - Referential Constraints

```sql
CREATE TABLE EMPLOYEES
  (ID INTEGER PRIMARY KEY,
   NAME VARCHAR (30) NOT NULL,
   DEPT SMALLINT REFERENCES DEPTMENTS,
   JOB CHAR (5) CHECK (JOB IN ('SALES', 'MGR', 'CLERK')) )
```
UNIQUE Constraint

- Columns of a UNIQUE constraint can be nullable (do not require NOT NULL)
- A row with a null value in a column of a UNIQUE constraint is not a duplicate of any other row

```sql
ALTER TABLE hobbies
ADD CONSTRAINT
UNIQUE (last, first, hobby);
```

### Hobbies

<table>
<thead>
<tr>
<th>LAST</th>
<th>FIRST</th>
<th>HOBBY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Holland</td>
<td>William</td>
<td>fishing</td>
</tr>
<tr>
<td>Holland</td>
<td>William</td>
<td>NULL</td>
</tr>
<tr>
<td>Holland</td>
<td>William</td>
<td>NULL</td>
</tr>
</tbody>
</table>

Check Constraints

- What they are:
  - Constraints over the values of columns in tables
  - Associated with base table or individual columns
  - Defined and/or altered at any time
  - Subqueries are allowed in check constraints

- Uses:
  - Define a range of allowable values:
    - The values of department number must lie in the range 10 to 100.
  - Define a list of possible values:
    - The job of an employee can only be sales, manager, or clerk.
  - Keep interdependencies:
    - Every employee that has been with the company more than 8 years must make more than $40,500.

```sql
CREATE TABLE EMPLOYEES
(id INTEGER NOT NULL,
 name VARCHAR (30),
 hiredate DATE,
 dept SMALLINT CHECK (dept BETWEEN 10 AND 100),
 job CHAR (5) CHECK (job IN ('SALES', 'MGR', 'CLERK')),
 salary US_DOLLAR CHECK (salary > US_DOLLAR (1000)))
...
CONSTRAINT YEARSAL CHECK (YEAR (hiredate) > 1986 OR SALARY > US_DOLLAR (40500)))
```
Assertions

- Constraints that can apply to an entire table or to multiple tables

CREATE ASSERTION max_employee
  CHECK (( SELECT COUNT (*)
          FROM employee ) < 250000 );

CREATE ASSERTION max_sal_expense
  CHECK (( SELECT SUM ( salary )
          FROM employee )
        <
        .8*( SELECT SUM ( budget )
          FROM dept) );

Referential Constraints

- Three flavors of referential constraints for handling nulls
  - "vanilla" (nulls don’t match, since equality requirement not satisfied)
  - PARTIAL MATCH
  - FULL MATCH
### Referential Constraints, Vanilla

**CREATE TABLE** hobbies

```sql
(...
FOREIGN KEY (last, first)
REFERENCES people (lname, fname));
```

<table>
<thead>
<tr>
<th>People</th>
<th>Hobbies</th>
</tr>
</thead>
<tbody>
<tr>
<td>LNAME</td>
<td>FNAME</td>
</tr>
<tr>
<td>Holland</td>
<td>William</td>
</tr>
<tr>
<td>Zysko</td>
<td>William</td>
</tr>
<tr>
<td></td>
<td>LAST</td>
</tr>
<tr>
<td>Holland</td>
<td>William</td>
</tr>
<tr>
<td>Zysko</td>
<td>William</td>
</tr>
<tr>
<td>Smith</td>
<td>(null)</td>
</tr>
<tr>
<td>(null)</td>
<td>William</td>
</tr>
<tr>
<td>(null)</td>
<td>(null)</td>
</tr>
</tbody>
</table>

- Columns of referenced table must be defined in a unique constraint
- For each row of the referenced table, the matching rows in the referencing table are those for which corresponding columns are equal
- Null value in any column of foreign key means that row will not be checked (note: such a row is not a matching row)

### Ref Constraints: MATCH PARTIAL

**ALTER TABLE** hobbies

```sql
ADD FOREIGN KEY (last, first)
REFERENCES people (lname, fname)
MATCH PARTIAL);
```

<table>
<thead>
<tr>
<th>People</th>
<th>Hobbies</th>
</tr>
</thead>
<tbody>
<tr>
<td>LNAME</td>
<td>FNAME</td>
</tr>
<tr>
<td>Holland</td>
<td>William</td>
</tr>
<tr>
<td>Zysko</td>
<td>William</td>
</tr>
<tr>
<td></td>
<td>LAST</td>
</tr>
<tr>
<td>Holland</td>
<td>William</td>
</tr>
<tr>
<td>Zysko</td>
<td>William</td>
</tr>
<tr>
<td>Smith</td>
<td>(null)</td>
</tr>
<tr>
<td>(null)</td>
<td>William</td>
</tr>
<tr>
<td>(null)</td>
<td>(null)</td>
</tr>
</tbody>
</table>

- All of the values of the FK that are not null must match corresponding columns of a row in the referenced table
- Matching rows for a given row in the referenced table have at least one non-null key column and are equal to the corresponding column in the referenced table
- Unique matching rows for a given row in the referenced table are those that are matching rows only to that row of the referenced table
Ref Constraints: MATCH FULL

ALTER TABLE hobbies
ADD FOREIGN KEY (last, first)
REFERENCES people (lname, fname)
MATCH FULL);

<table>
<thead>
<tr>
<th>LNAME</th>
<th>FNAME</th>
<th>NICK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Holland</td>
<td>William</td>
<td>Bill</td>
</tr>
<tr>
<td>Zysko</td>
<td>William</td>
<td>Willy</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LAST</th>
<th>FIRST</th>
<th>HOBBY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Holland</td>
<td>William</td>
<td>Fishing</td>
</tr>
<tr>
<td>Zysko</td>
<td>William</td>
<td>Dancing</td>
</tr>
<tr>
<td>Smith</td>
<td>(null)</td>
<td>Painting</td>
</tr>
<tr>
<td>(null)</td>
<td>William</td>
<td>Sailing</td>
</tr>
<tr>
<td>(null)</td>
<td>(null)</td>
<td>Garden</td>
</tr>
</tbody>
</table>

- Rows of the referencing table must contain either
  - All null values or all non-null values for the columns of the foreign key
- Matching rows are the same as when the match type is not specified
- MATCH FULL or (the default) MATCH SIMPLE differ with regard to admissible non-matching

Referential Constraints: Actions

- Define actions on update or delete of rows in the referenced table
  - NO ACTION
  - RESTRICT
  - CASCADE
  - SET NULL
  - SET DEFAULT

- Rules must be deterministic
  - Where the result would be order-dependent an exception will be raised
- Checked after the execution of an SQL statement
- Matching rows are determined before referential actions take place
- A matching row relationship may be dropped during execution of referential actions (except for restrict)
- New relationships will not be recognized until after the action has completed (PARTIAL MATCH only)
### Ref Constraints: NO ACTION

```sql
ALTER TABLE hobbies
ADD FOREIGN KEY (last, first)
REFERENCES people (lname, fname)
ON DELETE NO ACTION);
```

#### People

<table>
<thead>
<tr>
<th>LNAME</th>
<th>FNAME</th>
<th>NICK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Holland</td>
<td>William</td>
<td>Bill</td>
</tr>
<tr>
<td>Zysko</td>
<td>William</td>
<td>Willy</td>
</tr>
</tbody>
</table>

#### Hobbies

<table>
<thead>
<tr>
<th>LAST</th>
<th>FIRST</th>
<th>HOBBY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Holland</td>
<td>William</td>
<td>Fishing</td>
</tr>
<tr>
<td>Zysko</td>
<td>William</td>
<td>Dancing</td>
</tr>
<tr>
<td>(null)</td>
<td>William</td>
<td>Sailing</td>
</tr>
</tbody>
</table>

DELETE FROM people
WHERE lname='Zysko';

Exception: integrity constraint violation

### Ref Constraints: ON UPDATE NO ACTION

```sql
ALTER TABLE hobbies
ADD FOREIGN KEY (last, first)
REFERENCES people (lname, fname)
ON UPDATE NO ACTION);
```

#### People

<table>
<thead>
<tr>
<th>LNAME</th>
<th>FNAME</th>
<th>NICK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Holland</td>
<td>William</td>
<td>Bill</td>
</tr>
<tr>
<td>Zysko</td>
<td>William</td>
<td>Willy</td>
</tr>
</tbody>
</table>

#### Hobbies

<table>
<thead>
<tr>
<th>LAST</th>
<th>FIRST</th>
<th>HOBBY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Holland</td>
<td>William</td>
<td>Fishing</td>
</tr>
<tr>
<td>Zysko</td>
<td>William</td>
<td>Dancing</td>
</tr>
<tr>
<td>(null)</td>
<td>William</td>
<td>Sailing</td>
</tr>
</tbody>
</table>

UPDATE people SET lname=CASE lname
WHEN 'Holland' THEN 'Zysko'
WHEN 'Zysko' THEN 'Holland'
END;

DELETE FROM people
WHERE lname='Zysko';

Exception: integrity constraint violation

### People

<table>
<thead>
<tr>
<th>LNAME</th>
<th>FNAME</th>
<th>NICK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zysko</td>
<td>William</td>
<td>Bill</td>
</tr>
<tr>
<td>Holland</td>
<td>William</td>
<td>Willy</td>
</tr>
</tbody>
</table>

### Hobbies

<table>
<thead>
<tr>
<th>LAST</th>
<th>FIRST</th>
<th>HOBBY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Holland</td>
<td>William</td>
<td>Fishing</td>
</tr>
<tr>
<td>Zysko</td>
<td>William</td>
<td>Dancing</td>
</tr>
<tr>
<td>(null)</td>
<td>William</td>
<td>Sailing</td>
</tr>
</tbody>
</table>

187 188
**Ref Constraints: ON UPDATE**

ALTER TABLE hobbies
ADD FOREIGN KEY (last, first)
REFERENCES people (lname, fname)
ON UPDATE RESTRICT;

**Ref Constraints: ON DELETE**

ALTER TABLE hobbies
ADD FOREIGN KEY (last, first)
REFERENCES people (lname, fname)
ON DELETE CASCADE;
Ref Constraints: ON UPDATE SET NULL

ALTER TABLE hobbies
ADD FOREIGN KEY (last, first)
REFERENCES people (lname, fname)
ON UPDATE SET NULL);

<table>
<thead>
<tr>
<th>LNAME</th>
<th>FNAME</th>
<th>NICK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Holland</td>
<td>William</td>
<td>Bill</td>
</tr>
<tr>
<td>Zysko</td>
<td>William</td>
<td>Willy</td>
</tr>
</tbody>
</table>

People referenced, matching rows

Hobbies

<table>
<thead>
<tr>
<th>LAST</th>
<th>FIRST</th>
<th>HOBBY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Holland</td>
<td>William</td>
<td>Fishing</td>
</tr>
<tr>
<td>Zysko</td>
<td>William</td>
<td>Dancing</td>
</tr>
<tr>
<td>(null)</td>
<td>William</td>
<td>Sailing</td>
</tr>
</tbody>
</table>

People referencing, matching rows

UPDATE people
SET fname='John'
WHERE lname='Holland';

Ref Constraints: MATCH PARTIAL ON UPDATE CASCADE

ALTER TABLE hobbies
ADD FOREIGN KEY (last, first)
REFERENCES people (lname, fname)
MATCH PARTIAL ON UPDATE CASCADE);

<table>
<thead>
<tr>
<th>LNAME</th>
<th>FNAME</th>
<th>NICK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Holland</td>
<td>John</td>
<td>Bill</td>
</tr>
<tr>
<td>Zysko</td>
<td>William</td>
<td>Willy</td>
</tr>
</tbody>
</table>

People referenced, matching rows

Hobbies

<table>
<thead>
<tr>
<th>LAST</th>
<th>FIRST</th>
<th>HOBBY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Holland</td>
<td>John</td>
<td>Fishing</td>
</tr>
<tr>
<td>Zysko</td>
<td>William</td>
<td>Dancing</td>
</tr>
<tr>
<td>(null)</td>
<td>William</td>
<td>Sailing</td>
</tr>
</tbody>
</table>

People referencing, matching rows

UPDATE people
SET fname='John'
WHERE lname='Holland';
Ref Constraints: MATCH PARTIAL ON UPDATE CASCADE

ALTER TABLE hobbies
ADD FOREIGN KEY (last, first)
REFERENCES people (lname, fname)
MATCH PARTIAL ON UPDATE CASCADE;

People

<table>
<thead>
<tr>
<th>LNAME</th>
<th>FNAME</th>
<th>NICK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Holland</td>
<td>William</td>
<td>Bill</td>
</tr>
<tr>
<td>Zysko</td>
<td>William</td>
<td>Willy</td>
</tr>
</tbody>
</table>

Hobbies

<table>
<thead>
<tr>
<th>LAST</th>
<th>FIRST</th>
<th>HOBBY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Holland</td>
<td>William</td>
<td>Fishing</td>
</tr>
<tr>
<td>Zysko</td>
<td>William</td>
<td>Dancing</td>
</tr>
<tr>
<td>(null)</td>
<td>William</td>
<td>Sailing</td>
</tr>
</tbody>
</table>

Exception: triggered data change operation

UPDATE people
SET fname=nick'
WHERE fname='William';

People

<table>
<thead>
<tr>
<th>LNAME</th>
<th>FNAME</th>
<th>NICK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Holland</td>
<td>Bill</td>
<td>Bill</td>
</tr>
<tr>
<td>Zysko</td>
<td>Willy</td>
<td>Willy</td>
</tr>
</tbody>
</table>

Hobbies

<table>
<thead>
<tr>
<th>LAST</th>
<th>FIRST</th>
<th>HOBBY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Holland</td>
<td>Bill</td>
<td>Fishing</td>
</tr>
<tr>
<td>Zysko</td>
<td>Willy</td>
<td>Dancing</td>
</tr>
<tr>
<td>(null)</td>
<td>???</td>
<td>Sailing</td>
</tr>
</tbody>
</table>

Referential Constraints

- More than one action can be applied to a single row
- Both actions are performed to avoid order-dependent results
Ref Integrity between Comparable Types

- Foreign key does not have to be exact type as primary key, just comparable. For example:
  - VARCHAR to CHAR
  - INTEGER to DECIMAL to REAL
  - Supertype to subtype (e.g., person to manager)

CREATE TABLE people
  (lname VARCHAR(40),
   fname VARCHAR(30),
   nick CHAR (15),
   PRIMARY KEY(lname,fname)
  ...

CREATE TABLE hobbies
  (last CHAR(30),
   first CHAR(15),
   hobby VARCHAR (50)
  FOREIGN KEY (last, first)
   REFERENCES people (lname, fname));

Triggers Overview

- Triggers provide automatic execution of a set of SQL statements when a specific data change operation (UPDATE, INSERT, DELETE) occurs
- Bring application logic into the database
- Transform a passive to active DBMS
- Benefits of triggers include
  - Code reuse
  - Faster application development
  - Easier Maintenance
  - Guaranteed enforcements of business rules
Common Uses for Triggers

- Enforce "transitional" business rules
- Validate input data
- Generate new values for inserted / updated rows
- Cross-reference other tables
- Maintain audit, summary or mirror data in other tables
- Support "alerts"
  - E-mail notification
  - Initiate external actions

Trigger Flow

Insert 'Z-Files' row into Video_Table

<table>
<thead>
<tr>
<th>Video_Table</th>
<th>Title</th>
<th>Code</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Toy Glory</td>
<td>C28</td>
<td>14.95</td>
</tr>
<tr>
<td></td>
<td>Star Trak</td>
<td>S31</td>
<td>14.95</td>
</tr>
<tr>
<td></td>
<td>Indiana Bones</td>
<td>A07</td>
<td>15.95</td>
</tr>
<tr>
<td></td>
<td>Titanic</td>
<td>R67</td>
<td>19.95</td>
</tr>
<tr>
<td></td>
<td>Z-Files</td>
<td>S31</td>
<td>25.95</td>
</tr>
</tbody>
</table>

Update Category_Table
Set Cat_Total = Cat_Total + 1
Where CatNo = Code
Trigger: complete flow

Inserts
Updates
Deletes

integrity
constraint
checking

Before
Case
When Salary <= 10000 Then Tax = A03
When Salary <= 14000 Then Tax = A04
When Salary <= 19000 Then Tax = B11
When Salary <= 20000 Then Tax = A00
Else Tax = A05
End

After
Update Tax_Table
Set Tax_Count = Tax_Count + 1
Where Tax_Level = Tax

Trigger Characteristics

CREATE TRIGGER Payroll
AFTER UPDATE OF salary ON Paytable
FOR EACH STATEMENT
INSERT INTO PAYROLL_LOG ...;

Triggering Table: Table on which the trigger is defined
Triggering Event:
An SQL Data Change Operation (INSERT,DELETE,UPDATE)
UPDATE can be qualified by column
ON the triggering table
Trigger Activation Time: BEFORE or AFTER
Trigger Granularity: FOR EACH ROW or FOR EACH STATEMENT
Trigger Activation Time

### BEFORE

Evaluated entirely before triggering event
Can be considered an extension of the constraint system
Prevent invalid update operations
Useful for conditioning of input data
Validate or directly modify input values
SET allows you to modify values of affected rows
Only allowed in BEFORE triggers

```sql
CREATE TRIGGER update_balance
BEFORE INSERT ON account_history /* event */
REFERENCING NEW AS ta
FOR EACH ROW
WHEN (ta.TA_type = 'W') /* condition */
UPDATE accounts /* action */
SET balance = balance - ta.amount
WHERE account_# = ta.account_#;
```

### AFTER

Evaluated entirely after the triggering event
Can be considered an encapsulation of application logic that normally would be performed by the updating application
Perform audit trail logging or maintain summary data
Perform actions outside the database such as writing to an external dataset or sending an e-mail message

```sql
CREATE TRIGGER take_action
AFTER UPDATE OF balance ON accounts
REFERENCING OLD AS old_value
NEW AS new_value
FOR EACH ROW
WHEN (new_value.balance < 0)
IF account_type = 'VIP' THEN INSERT INTO send_letters ... ELSE INSERT INTO blocked_accounts ...;
```
**Trigger Granularity**

Granularity controls how many times the trigger is executed

**FOR EACH ROW:** Executed once for each row modified by the triggering event
  - Referred to as a row trigger or a row-level trigger

**FOR EACH STATEMENT:** Executed once each time the triggering SQL statement is issued
  - Referred to as a statement trigger or a statement-level trigger

```sql
CREATE TRIGGER AddOrder
BEFORE INSERT ON Order
REFERENCING NEW AS NewRow
FOR EACH ROW
SET NewRow.Date = CURRENT_DATE;
```

```sql
CREATE TRIGGER Purchase
AFTER INSERT ON Order
FOR EACH STATEMENT
CALL E-MAIL_CONFIRMATION;
```

**Triggered Action Condition**

- Triggered action condition is optional
- Condition can be any SQL condition (involving complex queries)
  - In the form of a WHEN clause (similar syntax to a WHERE clause)
  - Trigger will not fire if WHEN clause not satisfied

```sql
CREATE TRIGGER ReOrder
AFTER UPDATE OF InStock ON Video_Table
REFERENCING NEW AS N
FOR EACH ROW
WHEN (N.InStock < 0.10 * N.MaxStock)
   CALL ORDER_VIDEO(N.MaxStock - N.InStock, N.Video_Num);
```
Triggered SQL Statements

CREATE TRIGGER AddVideo
AFTER INSERT ON Video_Table
REFERENCING NEW AS Newrow
FOR EACH ROW
BEGIN ATOMIC
  UPDATE Item_Table SET Item_cnt = Item_cnt + 1
  WHERE ItemNo = Newrow.ItemNo;
  CALL E-MAIL_CUSTOMERS;
END;

Transition Variables

CREATE TRIGGER Increase
BEFORE UPDATE OF Salary_Table ON Employee
REFERENCING OLD AS Oldrow
NEW AS Newrow
FOR EACH ROW
WHEN (Newrow.Salary > Oldrow.Salary * 1.20)
SET Newrow.Salary = Oldrow.Salary * 1.20;

Transition Variables:

- Contain column values of row affected by triggering operation
- REFERENCING clause enables a correlation name to be assigned to the before and after states of the row
  - OLD AS Oldrow: Value of row before triggering SQL operation
  - NEW AS Newrow: Value of row after triggering SQL operation
Transition Tables

- Transition Tables
  - Contains entire set of rows affected by triggering operation
  - Apply aggregations over the set of affected rows (MAX, MIN, AVG)
  - REFERENCING clause specifies a table identifier
    - OLD_TABLE AS identifier: Table of BEFORE values
    - NEW_TABLE AS identifier: Table of AFTER values
  - Only valid for AFTER triggers

Valid Trigger Characteristic Combinations

<table>
<thead>
<tr>
<th>Granularity</th>
<th>Activation Time</th>
<th>Triggering Operation</th>
<th>Transition Variables Allowed</th>
<th>Transition Tables Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROW</td>
<td>BEFORE</td>
<td>INSERT</td>
<td>NEW</td>
<td>NONE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>UPDATE</td>
<td>OLD, NEW</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>DELETE</td>
<td>OLD</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AFTER</td>
<td>INSERT</td>
<td>NEW</td>
<td>NEW_TABLE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>UPDATE</td>
<td>OLD, NEW</td>
<td>OLD_TABLE, NEW_TABLE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DELETE</td>
<td>OLD</td>
<td>OLD_TABLE</td>
</tr>
<tr>
<td>STATEMENT</td>
<td>BEFORE</td>
<td>INSERT</td>
<td>NONE</td>
<td>NONE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>UPDATE</td>
<td>NONE</td>
<td>NONE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DELETE</td>
<td>NONE</td>
<td>NONE</td>
</tr>
<tr>
<td></td>
<td>AFTER</td>
<td>INSERT</td>
<td>NEW</td>
<td>NEW_TABLE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>UPDATE</td>
<td>OLD_TABLE, NEW_TABLE</td>
<td>OLD_TABLE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DELETE</td>
<td>OLD</td>
<td>OLD_TABLE</td>
</tr>
</tbody>
</table>
Statements Allowed as Triggered SQL

- Allowed in BEFORE triggers:
  - All DML statements except INSERT, UPDATE, and DELETE statements, invocations of routines that possibly modify SQL-data, connection and transaction statements
  - SET new transition variable

- Allowed in AFTER triggers:
  - All DML statements, except connection and transaction statements

Invoking UDFs and Stored Procedures

- Triggers can only perform SQL operations
- Ability to invoke stored procedures and user-defined functions expands types of possible triggered actions to include:
  - Conditional logic and looping
  - Initiation of external actions
  - Access to non-database resources
Raising Error Conditions

Triggers can be used for stopping invalid updates and for detecting other invalid conditions.

- **SIGNAL SQLSTATE** - New SQL statement that halts processing and returns the requested SQLSTATE and message to the application.

  Format:
  
  ```
  SIGNAL SQLSTATE sqlstate-string-constant (diagnostic-string-constant)
  ```

  - Only valid in triggered actions

```
CREATE TRIGGER Creditck
AFTER UPDATE OF Balance ON Customer
REFERENCING NEW AS Newrow
FOR EACH ROW
WHEN (Newrow.Balance > Newrow.CreditLimit)
SIGNAL SQLSTATE '75001' ('Credit Limit Exceeded - Shred Card');
```

---

Triggers - Misc.

- **Trigger cascading**
  - Triggers can cause other triggers to fire

- **Interaction with RI and other constraints**
  - Trigger actions can cause checking constraints
  - Trigger actions can cause RI checks to be performed
  - RI actions can activate triggers

- **Several triggers per event/activation time**
  - Ordering: ascending order of creation
SQL3’s Trigger Execution Model

Predicates

- From SQL/89
  - Comparison ( =, <, >, <=, >=, <>)
  - value1 BETWEEN value2 AND value3
  - value1 IN subquery or (value-list)
  - column LIKE pattern ESCAPE char
  - column IS NULL
  - value comp-op ALL or ANY subquery
  - EXISTS subquery
- Added in SQL/92
  - row-value IS NULL
  - row-value comp-op ALL or ANY or SOME subquery
  - UNIQUE subquery
  - row-value MATCH options subquery
  - row-value1 OVERLAPS row-value2
  - pred IS TRUE or FALSE or UNKNOWN
New and Extended Predicates

- Extensions
  - BETWEEN predicate (syntactic sugar)
  - LIKE predicate (BLOB support)
  - Matching rows: SIMPLE match (syntactic salt)

- New predicates
  - DISTINCT predicate (no SIMPLE match)
  - SIMILAR predicate (GREP facilities)
  - Type predicate (tests dynamic types)

Predicate Extensions: BETWEEN

- **A BETWEEN B AND C**
  - B    A    C
- SQL92: Implicit assumption: B<=C
- SQL99: new syntax, same behavior
  - A BETWEEN B AND C
  - same as
  - A BETWEEN ASYMMETRIC B AND C
- New in SQL99: Limits may be specified in any order:
  - A BETWEEN SYMMETRIC C AND B
Predicate Extensions: MATCH SIMPLE

- No new functionality (over SQL92)
- Better syntactic visibility for testing whether a row has a matching row in the result of a subquery
- Example:

  C.ForeignKey MATCH SIMPLE (VALUES (P.PrimaryKey))
  - Match, if operands NOT DISTINCT

- Corresponding MATCH option in foreign key definition

New in SQL99: DISTINCT Predicate

- Tests distinctness of two rows
- 2 rows DISTINCT, if at least two corresponding fields distinct
  - Scalar fields are not distinct if:
    - One of them NULL or
    - Both equal
  - Row-valued fields: recurse
  - Array-valued fields: Analogously to rows
- Note: NOT DISTINCT does not necessarily imply equality
**New in SQL99: SIMILAR Predicate**

- Specify character string similarity by regular expressions
- Well-known UNIX feature (e.g. GREP)
- Search patterns allow for
  - Masking symbols (%, _ (like LIKE))
  - Repetition (*, +)
  - Enumeration (e.g. [..; !?])
  - Negation (e.g. [^02468])
- Example:  
  WHERE T.Name SIMILAR TO 'St[.]*[aA]nford'

**New in SQL99: Type predicate**

- Allows determination of dynamic type
- Purpose
  - Allows row selection by specific subtypes (e.g. only with EURO in MONEY column)
  - Allows to prune off certain subtypes (e.g. French Francs)
- Example: Find items from table `real_estate_info` table that are priced in EURO (but not in any of its substitutes, e.g. Dutch guilders):  
  SELECT * FROM real_estate_info  
  WHERE Price IS OF ONLY (EURO)
**SET Operators**

- From SQL/89
  - UNION, but only in cursor declarations
- Added in SQL/92
  - UNION usable “everywhere”
  - INTERSECT
  - EXCEPT ("difference")
  - CORRESPONDING options

Find all people that do not have hobbies:

\[
(\text{SELECT } \text{lname}, \text{fname} \\
\text{FROM } \text{people}) \quad \text{EXCEPT} \\
(\text{SELECT } \text{last}, \text{first} \\
\text{FROM } \text{hobbies})
\]

**DML Orthogonality**

- Constructors exist for rows and tables

\[
\text{VALUES ('Holland', 'John')} \\
\text{VALUES ( ('Bartz', 'Mary'), ('Lindsay', 'Bob') )}
\]

- Special 1-row SELECTs:

\[
\text{VALUES (CURRENT TIME) INTO :hv}
\]

- Multirow inserts:

\[
\text{INSERT INTO PROVINCE} \\
\text{VALUES (('BC', 'British Columbia'),} \\
\text{ ('AB', 'Alberta'), ...} \\
\text{ ('NF', 'Newfoundland'))}
\]

- In-memory (transient) “tables”:

\[
\text{SELECT NATION, POPULATION} \\
\text{FROM NATIONS} \\
\text{UNION ALL} \\
\text{VALUES ('Quebec',6000000),} \\
\text{ ('California',24000000)}
\]
DML Orthogonality (cont.)

- Predicates operate on rows, rather than scalars

  (SELECT lname, fname FROM people WHERE ...) -- row subquery
  =
  (SELECT last, first FROM hobbies WHERE ...) -- row subquery

- Subquery can be used wherever expressions are allowed
  - Implicit conversion from row with single column to scalar value (problem)

  - On right hand side of expressions:

    SELECT *
    FROM people
    WHERE lname = (SELECT last
                   FROM hobbies
                   WHERE hobby = 'travel')

    UPDATE based on another table:
    UPDATE employee
    SET salary = (SELECT MAX(salary)
                  FROM mgr_employee)
    WHERE employee.name = 'Doe'

  - In the SELECT list:

    SELECT AVG(salary),
           (SELECT AVG(salary) FROM mgr_employee),
           (SELECT AVG(salary) FROM temp_employee)
    FROM employee
    SELECT last, first, ( SELECT description
                            FROM hobby_description h
                            WHERE h.name = hobbies.hobby)
    FROM hobbies
    WHERE ...
CAST Specification

CAST (<expression> AS <data type>)

- Converts a value of one data type into a value of another data type

CAST (salary AS CHAR (10))
CAST (:string AS INTEGER)
CAST (mtg_date AS CHAR (14))

- Not all combinations of source and target data type are valid
- When particular values cannot be represented in the target data type an exception is raised

CAST (’1992-FEBRUARY-29’ AS DATE)

- Uses
  - Force proper data types for UNION and similar operators
  - Assignment to host variables (e.g., DATE)
  - Produce printable results

CASE Expression

- If/then/else logic in SQL
  Powerful way of embedding logic in SELECT and WHERE clauses

  - Translation of encoded values
    SELECT abbreviation, CASE
    WHEN abbreviation = ‘CA’ THEN ‘California’
    WHEN abbreviation = ‘SD’ THEN ‘South Dakota’
    WHEN ...
    ELSE ‘Unknown’
    END
    FROM states
    WHERE ...

  - Protection against exceptions
    SELECT emp_name, deptno
    FROM employee
    WHERE { CASE bonus + commission
    WHEN 0 THEN NULL
    ELSE salary / (bonus+commission) 
    } > 10

- Used to “implement” COALESCE
  SELECT COALESCE (nick, first, last, ‘Unknown’)
  FROM people
### Joined Tables

- SQL-89 provides only
  - Cross products
  - Inner joins

  ```sql
  SELECT lname, nick, hobby
  FROM people, hobbies
  WHERE people.fname = hobbies.first
  AND people.lname = hobbies.last
  ```

- SQL-92 adds 3 types of joined table
  - Cross join (new syntax)
  - Inner join (new syntax)
  - Union join
  - Outer join - left, right, and full

- Variations
  - Named columns join
  - Natural join
  - Cross join
    - "Old style" join without WHERE clause

  ```sql
  SELECT * FROM (people FROM people, hobbies
  CROSS JOIN hobbies) AS result
  ```

### Outer Join

- Left, Right, and Full
- May be nested
- Joins two tables in such a way that includes rows from one table that have no match in the other

<table>
<thead>
<tr>
<th>People</th>
<th>Hobbies</th>
</tr>
</thead>
<tbody>
<tr>
<td>LNAME</td>
<td>FIRST</td>
</tr>
<tr>
<td>Adams</td>
<td>John</td>
</tr>
<tr>
<td>Holland</td>
<td>William</td>
</tr>
<tr>
<td>Zysko</td>
<td>William</td>
</tr>
</tbody>
</table>

Show people who have no hobbies

```sql
SELECT lname, fname, hobby
FROM people
LEFT OUTER JOIN hobbies
ON lname=last;
```
### Table Expressions

- SQL92 supports the notion of table expressions
  - View definition bodies placed inside the SQL statement instead of a view reference! Allows complex queries to be expressed in a single table-expression (SQL thereby becomes a relationally complete language).
  - Correlation name must be specified
  - Columns may be renamed

```
SELECT AVG(n_hobbies)
FROM (SELECT last, first, COUNT(*)
      FROM hobbies
      GROUP BY last, first)
      AS grouped_hobbies (last, first, n_hobbies)
WHERE grouped_hobbies.last LIKE 'N%'
```

- Why use them?
  - To avoid temporary view creation
  - To enable grouping on expressions
  - To allow host variables in the "view"

```
SELECT Source, Destination,
      MIN(New_cost)
FROM (SELECT Source, Destination, Carrier,
        Cost * :discount_rate AS New_cost
        FROM Flights)
      AS New_price
GROUP BY Source, Destination
```

### Common Table Expressions

- SQL3 expands SQL92's notion of table expressions, allowing them to be defined once and used multiple times
  - "Reusable" table expressions

- Why use them?
  - To avoid overhead of re-evaluation for each reference
  - To avoid errors associated with each reference possibly returning different results.
  - Enable recursive queries (see later charts)

For each carrier, give its lowest fare between two cities, and all carriers who offer a lower or equal lowest fare between the same two cities.

```
WITH New_Price AS (
    SELECT Source, Destination, Carrier,
           Cost * :discount_rate AS New_Cost
    FROM Flights)
SELECT a.Source, a.Destination, a.Carrier, a.New_Cost,
     b.Carrier, b.New_Cost
FROM New_Price a, New_Price b
```
**Recursive SQL: A First Example**

*Find the cheapest flight from Paris to San Jose or San Francisco.*

WITH RECURSIVE Reachable_From (Source, Destin, Total_Cost) AS
  ( SELECT Source, Destination, Cost
    FROM Flights
    WHERE Source = 'Paris'
    UNION
    SELECT in.Source, out.Destination, in.Total_Cost + out.Cost
    FROM Reachable_From in, Flights out
    WHERE in.Destin = out.Source
  )
SELECT Source, Destin, MIN(Total_Cost)
FROM Reachable_From
WHERE Destin in ('San Jose', 'San Francisco')
GROUP BY Source, Destin

---

**Recursive SQL: Rationale and Challenges**

- **What is recursive SQL?**
  - self-referencing table expressions
  - self-referencing views
- **Why use recursion?**
  - Bill of material processing
  - Network traversals (e.g. airline routing)
- **Functionality and performance benefits**
- **Challenge: integration into SQL**
  - Syntax in analogy to Datalog
  - Advanced recursion (e.g. mutual recursion)
  - Integration with different forms of joins
  - Allowing for duplicates
  - Graph traversal in "depth first" or "breadth first"
  - Cycle control
**Recursive SQL: Advanced Use of Join Facilities**

- Find all connections between two cities and the number of risky hops
  - Table risky_cities contains all the cities with insecure airports

```sql
with RECURSIVE reachable_from (source, destin, risk_count) AS
    (select source, destin, 0
     from flights
     union all
     select in.source, out.destin,
           -- Add one to the risk count for this flight.
           risk_count + case when risky_cities.name is null then 0 else 1 end
           as risk_count
     from reachable_from in
           inner join (flights out left outer join risky_cities
                        on (out.source = risky_cities.name))
           on (in.destin = out.source)
    )
select * from reachable_from

Because not every flight goes through a risky city, we use an outer join instead of a regular join
```

**Bill of Material Queries**

### PART_PART

<table>
<thead>
<tr>
<th>Major</th>
<th>Minor</th>
<th>Qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>P2</td>
<td>2</td>
</tr>
<tr>
<td>P1</td>
<td>P3</td>
<td>3</td>
</tr>
<tr>
<td>P1</td>
<td>P4</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>P3</td>
<td>3</td>
</tr>
<tr>
<td>P2</td>
<td>P5</td>
<td>2</td>
</tr>
<tr>
<td>P3</td>
<td>P5</td>
<td>4</td>
</tr>
<tr>
<td>P3</td>
<td>P6</td>
<td>2</td>
</tr>
<tr>
<td>P4</td>
<td>P3</td>
<td>5</td>
</tr>
<tr>
<td>P4</td>
<td>P6</td>
<td>2</td>
</tr>
<tr>
<td>P7</td>
<td>P8</td>
<td>3</td>
</tr>
<tr>
<td>P7</td>
<td>P9</td>
<td>2</td>
</tr>
</tbody>
</table>

### PARTMASTER

<table>
<thead>
<tr>
<th>Pno</th>
<th>Pname</th>
<th>Levelcode</th>
<th>IsPhantom</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>Main</td>
<td>root</td>
<td>no</td>
</tr>
<tr>
<td>P2</td>
<td>Rack</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>Fitting</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>Cover</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>P5</td>
<td>Brace</td>
<td>leaf</td>
<td>no</td>
</tr>
<tr>
<td>P6</td>
<td>Angle</td>
<td>leaf</td>
<td>no</td>
</tr>
</tbody>
</table>
Bill of Material Queries: Quantity Calculation

- Preliminary quantities calculation

WITH RECURSIVE px (Major, Minor, Qty) AS
  ( SELECT Major, Minor, Qty
    FROM Part_Part pp
    WHERE pp.Major = 'P1'
    UNION ALL
    FROM px, Part_Part pp
    WHERE pp.Major = px.Minor )

SELECT Major, Minor, Qty FROM px;
(counts along "paths": e.g. P1 contains 4 P5 items along path P1->P2->P5)

- Summarized quantities calculation

WITH RECURSIVE px (Major, Minor, Qty) AS
  ( SELECT Major, Minor, Qty
    FROM Part_Part pp
    WHERE pp.Major = 'P1'
    UNION ALL
    FROM px, Part_Part pp
    WHERE pp.Major = px.Minor )

SELECT Minor, SUM(Qty)
FROM px
GROUP BY Minor;
(counts total quantity for each part)

Bill of Material Queries: Graph Restructuring

- Eliminating intermediate ‘phantom’ parts

WITH RECURSIVE PPX (Major, IsNewArc, Minor, Qty) AS
  ( SELECT Major, pm.isPhantom, Minor, Qty
    FROM Part_Part pp, Partmaster pm
    WHERE pp.Major = 'P1' and pp.Major = pm.Pno
    AND pm.isPhantom = 'NO'
    UNION ALL
    SELECT case pm.isPhantom
      WHEN 'YES' THEN pp.Major ELSE pp.Major END,     --- Major
      pm.isPhantom,                                    --- IsNewArc
      pp.Minor,                                        --- Minor
      CASE pm.isPhantom
        WHEN 'YES' THEN pp.Qty * pp.Qty ELSE pp.Qty END,--- Qty
        FROM     pp, Part_Part pp, Partmaster pm

FROM pp, Partmaster pm
WHERE pp.Minor = pm.Pno AND
  pm.isPhantom = 'NO'
Bill of Material Queries: Generation of Search Order Columns

- Parts Explosion with generated column for "depth-first" search order (in essence: concatenated key info)

```
WITH RECURSIVE px (Major, Minor) AS
    ( SELECT Major, Minor FROM Part_Part pp WHERE pp.Major = 'P1'
        UNION ALL
    )
SELECT DepthFirstSearch ORDER BY Major, Minor
FROM px
ORDER BY CatenatedKey
```

- Parts Explosion with generated column for "breadth-first" search order (level and key info)

```
WITH RECURSIVE px (Major, Minor) AS
    ( SELECT Major, Minor FROM Part_Part pp WHERE pp.major = 'P1'
        UNION ALL
    )
SEARCH BREADTH FIRST BY Major, Minor SET OrderColumn
SELECT Major, Minor
FROM px
ORDER BY OrderColumn;
```

- BREADTH or DEPTH feature only applicable for "simple" recursive queries
- Results in query rewrite

Bill of Material Queries: Protection against Looping

- Duplicate suppression avoids cycles
- Use of CYCLE clause for simple recursive queries
  - Effects rewrite of query s.t. traversed paths are maintained (column path) using "navigator columns" (here column Minor), and rows with duplicate paths are marked (see column CycleMark)
  - Query techniques used in rewrite process also usable "manually"

```
WITH RECURSIVE px (Major, Minor) AS
    ( SELECT Major, Minor
        FROM Part_Part pp
        WHERE pp.Major = 'P1'
        UNION ALL
        SELECT pp.Major, pp.Minor,
        FROM px, Part_Part pp
        WHERE pp.Major = px.Minor
    )
CYCLE Minor SET CycleMark TO '1' DEFAULT '0' USING Path
SELECT Major, Minor
FROM px
ORDER BY Major, Minor DESC;
```
Bill of Material queries: Graph Pruning Examples

- "where-used"

```sql
WITH RECURSIVE px (Major, Minor) AS 
  ( SELECT Major, Minor 
  FROM Part_Part pp 
  WHERE pp.Minor = 'P5' 
  UNION 
  SELECT pp.Major, pp.Minor 
  FROM px, Part_Part pp 
  WHERE pp.Minor = px.Major) 
SELECT Major, Minor FROM px;
```

- Parts explosion excluding base parts

```sql
WITH RECURSIVE px (Major, Minor, Pname, Levelcode) AS 
  ( SELECT Major, Minor, pm.pname, pm.Levelcode 
  FROM Part_Part pp, Partmaster pm 
  WHERE pp.Major = 'P1' 
  AND pp.Minor = pm.Pno 
  UNION ALL 
  SELECT pp.Major, pp.Minor, pm.Pname, pm.Levelcode 
  FROM px, Part_Part pp, partmaster 
  WHERE pp.Major = px.Minor 
  AND pp.Minor = pm.Pno 
  AND px.Levelcode <> 'LEAF') 
SELECT Major, Minor, Pname, Levelcode FROM px;
```

SQL99 OLAP SQL Extensions

- Extension to GROUP BY clause
- Produces "super aggregate" rows
- ROLLUP equivalent to "control breaks"
- CUBE equivalent to "cross tabulation"
- GROUPING SETS equivalent to multiple GROUP BYs
- Provides "data cube" collection capability
  ▶ Often used with data visualization tool
OLAP Schema

- Typically uses a "STAR" structure
  - Dimension tables tend to be small
  - Fact table tends to be huge

```
CREATE VIEW Sales AS
(SELECT ds.*, YEAR (sales_date) AS year, MONTH (sales_date) AS month,
DAY (sales_date) AS day
FROM (Detailed_Sales NATURAL JOIN Store NATURAL JOIN Product
NATURAL JOIN Period) ds
```

ROLLUP

- Extends grouping semantics to produce "subtotal" rows
  - Produces "regular" grouped rows
  - Produces same groupings reapplied down to grand total

```
SELECT month, city, producer, SUM(units) AS sum_units
FROM Sales
WHERE year = 1998
GROUP BY ROLLUP (month, city, producer)
```
ROLLUP

Find the total sales per region and sales manager during each month of 1996, with subtotals for each month, and concluding with the grand total:

**SQL Query:**
```
SELECT month, region, sales_mgr, SUM(price)
FROM Sales
WHERE year = 1996
GROUP BY ROLLUP (month, region, sales_mgr)
```

<table>
<thead>
<tr>
<th>MONTH</th>
<th>REGION</th>
<th>SALES_MGR</th>
<th>SUM(price)</th>
</tr>
</thead>
<tbody>
<tr>
<td>April</td>
<td>Central</td>
<td>Chow</td>
<td>25 000</td>
</tr>
<tr>
<td>April</td>
<td>Central</td>
<td>Smith</td>
<td>15 000</td>
</tr>
<tr>
<td>April</td>
<td>Central</td>
<td></td>
<td>40 000</td>
</tr>
<tr>
<td>April</td>
<td>NorthWest</td>
<td>Smith</td>
<td>15 000</td>
</tr>
<tr>
<td>April</td>
<td>NorthWest</td>
<td></td>
<td>15 000</td>
</tr>
<tr>
<td>April</td>
<td></td>
<td></td>
<td>55 000</td>
</tr>
<tr>
<td>May</td>
<td>Central</td>
<td>Chow</td>
<td>25 000</td>
</tr>
<tr>
<td>May</td>
<td>Central</td>
<td></td>
<td>25 000</td>
</tr>
<tr>
<td>May</td>
<td>NorthWest</td>
<td>Smith</td>
<td>15 000</td>
</tr>
<tr>
<td>May</td>
<td>NorthWest</td>
<td></td>
<td>15 000</td>
</tr>
<tr>
<td>May</td>
<td></td>
<td></td>
<td>40 000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>95 000</td>
</tr>
</tbody>
</table>

CUBE

- Further extends grouping semantics to produce multidimensional grouping and "subtotal" rows
  - Superset of ROLLUP
  - Produces "regular" grouped rows
  - Produces same groupings reapplied down to grand total
  - Produces additional groupings on all variants of the CUBE clause

**SQL Query:**
```
SELECT month, city, product_id, SUM(units)
FROM Sales
WHERE year = 1998
GROUP BY CUBE (month, city, product_id)
```
### SELECT ... GROUP BY CUBE

SELECT month, region, sales_mgr, SUM(price)
FROM Sales
WHERE year = 1996
GROUP BY CUBE (month, region, sales_mgr)

<table>
<thead>
<tr>
<th>MONTH</th>
<th>REGION</th>
<th>SALES_MGR</th>
<th>SUM(price)</th>
</tr>
</thead>
<tbody>
<tr>
<td>April</td>
<td>Central</td>
<td>Chow</td>
<td>25 000</td>
</tr>
<tr>
<td>April</td>
<td>Central</td>
<td>Smith</td>
<td>15 000</td>
</tr>
<tr>
<td>April</td>
<td>Central</td>
<td></td>
<td>40 000</td>
</tr>
<tr>
<td>April</td>
<td>NorthWest</td>
<td>Smith</td>
<td>15 000</td>
</tr>
<tr>
<td>April</td>
<td>NorthWest</td>
<td></td>
<td>55 000</td>
</tr>
<tr>
<td>April</td>
<td>-</td>
<td></td>
<td>25 000</td>
</tr>
<tr>
<td>May</td>
<td>Central</td>
<td>Chow</td>
<td>25 000</td>
</tr>
<tr>
<td>May</td>
<td>Central</td>
<td></td>
<td>25 000</td>
</tr>
<tr>
<td>May</td>
<td>NorthWest</td>
<td>Smith</td>
<td>15 000</td>
</tr>
<tr>
<td>May</td>
<td>NorthWest</td>
<td></td>
<td>15 000</td>
</tr>
<tr>
<td>May</td>
<td>-</td>
<td></td>
<td>40 000</td>
</tr>
<tr>
<td>-</td>
<td>Central</td>
<td>Chow</td>
<td>50 000</td>
</tr>
<tr>
<td>-</td>
<td>Central</td>
<td>Smith</td>
<td>15 000</td>
</tr>
<tr>
<td>-</td>
<td>NorthWest</td>
<td>Smith</td>
<td>65 000</td>
</tr>
<tr>
<td>-</td>
<td>NorthWest</td>
<td></td>
<td>30 000</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td></td>
<td>50 000</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>Smith</td>
<td>45 000</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td></td>
<td>95 000</td>
</tr>
</tbody>
</table>

### GROUPING SETS

- Multiple "groupings" in a single pass
  - Used in conjunction with usual aggregation (MAX, MIN, SUM, AVG, COUNT, ...)
  - Allows multiple groups e.g. (month, region) and (month, sales_mgr)
  - Result can be further restricted via HAVING clause

Find the total sales during each month of 1996, per region and per sales manager:

SELECT month, region, sales_mgr, SUM(price)
FROM Sales
WHERE year = 1996
GROUP BY GROUPING SETS ((month, region), (month, sales_mgr))

<table>
<thead>
<tr>
<th>MONTH</th>
<th>REGION</th>
<th>SALES_MGR</th>
<th>SUM(SALES)</th>
</tr>
</thead>
<tbody>
<tr>
<td>April</td>
<td>Central</td>
<td></td>
<td>40 000</td>
</tr>
<tr>
<td>April</td>
<td>NorthWest</td>
<td></td>
<td>15 000</td>
</tr>
<tr>
<td>April</td>
<td></td>
<td>Chow</td>
<td>25 000</td>
</tr>
<tr>
<td>April</td>
<td></td>
<td>Smith</td>
<td>30 000</td>
</tr>
<tr>
<td>May</td>
<td>Central</td>
<td></td>
<td>25 000</td>
</tr>
<tr>
<td>May</td>
<td>NorthWest</td>
<td></td>
<td>15 000</td>
</tr>
<tr>
<td>May</td>
<td></td>
<td>Chow</td>
<td>25 000</td>
</tr>
<tr>
<td>May</td>
<td></td>
<td>Smith</td>
<td>15 000</td>
</tr>
</tbody>
</table>
Generating Grand Total Rows

- Special syntax available to include a "grand total" row in the result
  - Grand totals are generated implicitly with ROLLUP and CUBE operations
  - Syntax allows grand totals to be generated without additional aggregates

Get total sales by month, region, and sales manager and also the overall total sales:

```sql
SELECT month, region, sales_mgr, SUM(price)
FROM Sales
WHERE year = 1996
GROUP BY GROUPING SETS ( (month, region, sales_mgr), () )
```

<table>
<thead>
<tr>
<th>MONTH</th>
<th>REGION</th>
<th>SALES_MGR</th>
<th>SUM(SALES)</th>
</tr>
</thead>
<tbody>
<tr>
<td>April</td>
<td>Central</td>
<td>Chow</td>
<td>25 000</td>
</tr>
<tr>
<td>April</td>
<td>Central</td>
<td>Smith</td>
<td>15 000</td>
</tr>
<tr>
<td>April</td>
<td>NorthWest</td>
<td>Smith</td>
<td>15 000</td>
</tr>
<tr>
<td>May</td>
<td>Central</td>
<td>Chow</td>
<td>25 000</td>
</tr>
<tr>
<td>May</td>
<td>NorthWest</td>
<td>Smith</td>
<td>15 000</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>95 000</td>
</tr>
</tbody>
</table>

The GROUPING Function

- New column function
  - Allows detection of rows that were generated during the execution of CUBE and ROLLUP i.e. generated nulls to be distinguished from naturally occurring ones

Example:

Run a rollup, and flag the generated rows...

```sql
SELECT month, region, sales_mgr, SUM(price), GROUPING(sales_mgr)
FROM Sales
WHERE year = 1996
GROUP BY ROLLUP (month, region, sales_mgr)
```
SELECT month, region, sales_mgr, SUM(price), GROUPING(sales_mgr) AS GROUPED
FROM Sales
WHERE year = 1996
GROUP BY ROLLUP (month, region, sales_mgr)

<table>
<thead>
<tr>
<th>MONTH</th>
<th>REGION</th>
<th>SALES_MGR</th>
<th>SUM(SALES)</th>
<th>GROUPED</th>
</tr>
</thead>
<tbody>
<tr>
<td>April</td>
<td>Central</td>
<td>Chow</td>
<td>25 000</td>
<td>0</td>
</tr>
<tr>
<td>April</td>
<td>Central</td>
<td>Smith</td>
<td>15 000</td>
<td>0</td>
</tr>
<tr>
<td>April</td>
<td>Central</td>
<td>-</td>
<td>40 000</td>
<td>1</td>
</tr>
<tr>
<td>April</td>
<td>NorthWest</td>
<td>Smith</td>
<td>15 000</td>
<td>0</td>
</tr>
<tr>
<td>April</td>
<td>NorthWest</td>
<td>-</td>
<td>15 000</td>
<td>1</td>
</tr>
<tr>
<td>April</td>
<td>-</td>
<td>-</td>
<td>55 000</td>
<td>1</td>
</tr>
<tr>
<td>May</td>
<td>Central</td>
<td>Chow</td>
<td>25 000</td>
<td>0</td>
</tr>
<tr>
<td>May</td>
<td>Central</td>
<td>-</td>
<td>25 000</td>
<td>1</td>
</tr>
<tr>
<td>May</td>
<td>NorthWest</td>
<td>Smith</td>
<td>15 000</td>
<td>0</td>
</tr>
<tr>
<td>May</td>
<td>NorthWest</td>
<td>-</td>
<td>15 000</td>
<td>1</td>
</tr>
<tr>
<td>May</td>
<td>-</td>
<td>-</td>
<td>40 000</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>95 000</td>
<td>1</td>
</tr>
</tbody>
</table>

Selecting Nongrouped Columns

Nongrouped columns can sometimes be selected based on functional dependencies:

SELECT e.deptno, d.location, AVG(e.salary) AS average
FROM Emp e, Dept d
WHERE e.deptno = d.deptno
GROUP BY e.deptno

SELECT e.deptno, e.name, AVG(e.salary) AS Average
FROM Emp e, Dept d
WHERE e.deptno = d.deptno
GROUP BY e.deptno

e.deptno determines d.deptno (equals in WHERE clause),
and d.deptno determines d.location (deptno is PK of Dept);
therefore, d.deptno and d.location are consistent within any group. This is functional dependency analysis in action.
UPDATE through UNION

CREATE VIEW people_or_hobbies
AS SELECT fname, lname
FROM  people
UNION ALL
SELECT first, last
FROM hobbies;

UPDATE  people_or_hobbies
SET fname='Wilhemina' ,
lname='Jing'
WHERE fname='Zysko' and
lname='William' ;

UPDATE through JOIN

DECLARE c1 CURSOR FOR
WITH people_and_hobbies AS
(peoples INNER JOIN hobbies
ON (fname=first AND lname=last))
SELECT *
FROM people_and_hobbies;

UPDATE people_and_hobbies
SET hobby='birdwatching'
WHERE CURRENT OF c1;
**INSERT through JOIN**

```
CREATE VIEW Dancers AS
  people JOIN hobbies
  ON (fname=first AND last=lname)
  WHERE hobby='Dancing'
  WITH CHECK OPTION;
```

```
INSERT INTO dancers
VALUES ('John', 'Harshman', 'John'
  'John', 'Harshman', 'Dancing);
```

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LNAME</td>
<td>FNAME</td>
<td>NICK</td>
<td>LAST</td>
<td>FIRST</td>
<td>HOBBY</td>
</tr>
<tr>
<td>Holland</td>
<td>William</td>
<td>Bill</td>
<td>Holland</td>
<td>William</td>
<td>Fishing</td>
</tr>
<tr>
<td>Zysko</td>
<td>William</td>
<td>Willy</td>
<td>Zysko</td>
<td>William</td>
<td>Dancing</td>
</tr>
<tr>
<td>Smith</td>
<td>(null)</td>
<td>Painting</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Dancers**

```
INSERT INTO dancers
VALUES ('John', 'Harshman', 'John'
  'John', 'Harshman', 'Dancing');
```

```
<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LNAME</td>
<td>FNAME</td>
<td>NICK</td>
<td>LAST</td>
<td>FIRST</td>
<td>HOBBY</td>
</tr>
<tr>
<td>Zysko</td>
<td>William</td>
<td>Willy</td>
<td>Zysko</td>
<td>William</td>
<td>Dancing</td>
</tr>
<tr>
<td>Harshman</td>
<td>John</td>
<td>John</td>
<td>Harshman</td>
<td>John</td>
<td>Dancing</td>
</tr>
</tbody>
</table>
```

**Named Expressions**

- Provide name for column in result table
- Can be used in ORDER BY

```
DECLARE payroll CURSOR FOR
  SELECT name, base_salary + commission AS pay
  FROM employees
ORDER BY pay
```
**Scrollable Cursors**

- In SQL89, `FETCH` always retrieves “next” row
- In SQL92, cursors are scrollable:
  - Allows both forward and backward movement of the cursor
  - Allows skipping of rows

```sql
EXEC SQL DECLARE c SCROLL CURSOR FOR SELECT ...;
EXEC SQL OPEN c ;
EXEC SQL FETCH ABSOLUTE 10 FROM c INTO ...;
EXEC SQL FETCH RELATIVE 32 FROM C INTO ...;
EXEC SQL FETCH PRIOR FROM C INTO ...;
```

- `FETCH` options are:
  - `FIRST`
  - `LAST`
  - `NEXT`
  - `PRIOR`
  - `ABSOLUTE n`
  - `RELATIVE n`

---

**Scrollable Cursor Example**

<table>
<thead>
<tr>
<th>FIRST</th>
<th>RELATIVE -3</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRIOR</td>
<td>current position</td>
</tr>
<tr>
<td>NEXT</td>
<td></td>
</tr>
<tr>
<td>ABSOLUTE 10</td>
<td></td>
</tr>
<tr>
<td>RELATIVE 10</td>
<td></td>
</tr>
<tr>
<td>LAST</td>
<td></td>
</tr>
</tbody>
</table>
READ ONLY and FOR UPDATE

- Allows an explicit specification of the columns of a cursor that may be updated

```sql
EXEC SQL DECLARE c1 CURSOR FOR
SELECT lname, fname FROM people
READ ONLY ;
```

```sql
EXEC SQL DECLARE c2 CURSOR FOR
SELECT lname, fname, nick FROM people
FOR UPDATE OF nick;
```

Cursor Sensitivity

- SQL92 feature
- New in SQL99: ASENSITIVE
  - Same as SQL92, when neither SENSITIVE nor INSENSITIVE specified
  - Effect implementation-defined
- Purpose of cursor sensitivity: controls whether cursor Cx can see changes which have been
  - Affected in same TX (say TX1)
  - Not caused by CX itself (e.g. by cursor CY; by INSERT statement)
Cursor Sensitivity (cont.)

- SENSITIVE: changes are visible
- INSENSITIVE: changes are invisible
  - Cursor is READ ONLY
- Example:
  ```sql
  EXEC SQL;
  DECLARE C CURSOR SENSITIVE FOR
  SELECT * FROM People;
  ```
- Note: Visibility of changes from foreign TXs also controlled by isolation level
- If cursor is holdable and kept open:
  - SENSITIVE: changes of TX1 and subsequent TX2 remain/become visible
  - INSENSITIVE: these changes are not visible

Holdable cursors

- New in SQL99
- Let TXC be the TX in which a cursor C is created
- Classical (non-holdable) cursors are closed when TXC is terminated (i.e. they do not survive TXC)
- Open holdable cursors
  - Remain open when TX is committed
  - Are closed and destroyed when
    - TXC is rolled back
    - Session is terminated
**ORDER BY Extensions**

- ORDER BY on columns not in the select list

```sql
DECLARE CURSOR FOR
    SELECT empno, name
    FROM employee
    ORDER BY salary DESC
```

- ORDER BY expressions

```sql
DECLARE C2 CURSOR FOR
    SELECT empno, name
    FROM employee
    ORDER BY salary + bonus ASC
```

**Temporary Tables**

- Used to increase concurrency and decrease processing costs
  - May be updated even if the transaction has an access mode of read-only
- Created Global Temporary Tables
  - Table definition is persistent (i.e., exists within a schema)
  - A new instance of the table is created for each SQL-session
  - Table may be shared by procedures in multiple modules in the same session

```sql
CREATE GLOBAL TEMPORARY TABLE VIP_accounts
(account_id INTEGER,
balance money,
type account_type)
ON COMMIT PRESERVE ROWS; --- DELETE rows also supported
```

- Created Local Temporary Tables
  - Distinct instances are created for each SQL-session/module combination

```sql
CREATE LOCAL TEMPORARY TABLE VIP_accounts
(account_id INTEGER,
balance money,
type account_type)
ON COMMIT DELETE ROWS;
```
Temporary Tables

- **Declared Local Temporary Tables**
  - Table definition is not persistent (i.e., exists only in the module it is defined)
  - An instance is created for each SQL-session/module combination

```sql
MODULE ...
LANGUAGE ...
SCHEMA ...
AUTHORIZATION ...
DECLARE LOCAL TEMPORARY TABLE MODULE.t1
(account_id INTEGER,
balance money,
type account_type)
ON COMMIT DELETE ROWS;
```

- **Constraints between temporary tables and persistent tables**

<table>
<thead>
<tr>
<th>Source Table</th>
<th>Target Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>base</td>
<td>any temporary</td>
</tr>
<tr>
<td>any temporary</td>
<td>base</td>
</tr>
<tr>
<td>created global</td>
<td>created global</td>
</tr>
<tr>
<td>created local</td>
<td>created global</td>
</tr>
<tr>
<td>created local</td>
<td>created local</td>
</tr>
<tr>
<td>declared local</td>
<td>created global</td>
</tr>
<tr>
<td>declared local</td>
<td>created local</td>
</tr>
<tr>
<td>declared local</td>
<td>declared local</td>
</tr>
</tbody>
</table>

Roles

- **New in SQL99; benefits:**
  - Simplifies definition of complex sets of privileges

- **Roles are created**
  - Note: definition of users implementation-defined
  ```sql
  CREATE ROLE Auditor
  CREATE ROLE AuditorGeneral
  ```

- **Roles may be assigned to users & roles**
  ```sql
  GRANT Auditor TO AuditorGeneral
  WITH ADMIN OPTION
  GRANTED BY CURRENT ROLE

  GRANT Auditor TO Smith
  ```

- **Controllable whether to grant as user or role**
Roles (cont.)

- Roles (like users) may own objects
- As to users, privileges may be granted to roles
  - Grant INSERT ON TABLE Budget TO Auditor
    - This privilege also among privileges of AuditorGeneral
- A role R identifies a set of privileges:
  - Those directly granted to R
  - Those of the roles granted to R

Roles (cont.)

- At any time there is at least either a valid current user or a valid current role
- Current user can be set
  - Invalidates current role
    - SET SESSION AUTHORIZATION 'JDOE'
- Current role can be set or invalidated
  - SET ROLE Auditor
- Operations (e.g. INSERT) determine the kind of required privileges
  - Often: union of user’s and role’s privileges
- Session context maintains stack of user and role identifier pairs
  - New pair is pushed when externally invoked procedure is executed
  - Temporarily makes client module identifier the current user
  - Enables invoker’s rights in a limited fashion
Error Handling: SQLSTATE

- In SQL/89
  - SQLCODE integer value: 0, +100, negative values deprecated feature
  - 5-character SQLSTATE may be used in place of SQLCODE
    - 2 characters represent class
    - 3 characters represent subclass
  - Classes and subclasses reserved for vendor extensions

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>SQLCODE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'00000'</td>
<td>0</td>
<td>Successful completion</td>
</tr>
<tr>
<td>'02000'</td>
<td>100</td>
<td>No data</td>
</tr>
<tr>
<td>'22001'</td>
<td>-n</td>
<td>Data exception - string data, right truncation</td>
</tr>
<tr>
<td>'22012'</td>
<td>-n</td>
<td>Data exception - division by zero</td>
</tr>
</tbody>
</table>

Error Handling: Diagnostics Area

- Records information about exceptions, warnings, no data, and successful completion
- Can retain information about several exceptions
  - Execution of a statement may raise multiple exceptions
  - SQLSTATE, relevant catalog, schema, or table, relevant constraint, relevant cursor, message text

```
EXEC SQL BEGIN DECLARE SECTION;
int   n, i;
char   sqlstate [5], s [5];
EXEC SQL END DECLARE SECTION;

report_error ()
{
    EXEC SQL GET DIAGNOSTICS :n = NUMBER;
    for (i = 1, i <=n, i++)
    {
        EXEC SQL GET DIAGNOSTICS EXCEPTION :i
        :s = RETURNED_SQLSTATE, ... ;
        printf (...) ;
    }
```

Transactions in SQL92

- Transactions start implicitly and end with COMMIT WORK or ROLLBACK WORK
  - Attributes
    - Access mode: READ ONLY and READ WRITE
    - Isolation level: READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, and SERIALIZABLE
    - Diagnostics area size

<table>
<thead>
<tr>
<th>Isolation level</th>
<th>Dirty read</th>
<th>Non-repeatable read</th>
<th>Phantom</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ UNCOMMITTED</td>
<td>possible</td>
<td>possible</td>
<td>possible</td>
</tr>
<tr>
<td>READ COMMITTED</td>
<td>not possible</td>
<td>possible</td>
<td>possible</td>
</tr>
<tr>
<td>REPEATABLE READ</td>
<td>not possible</td>
<td>not possible</td>
<td>possible</td>
</tr>
<tr>
<td>SERIALIZABLE</td>
<td>not possible</td>
<td>not possible</td>
<td>not possible</td>
</tr>
</tbody>
</table>

Transactions in SQL92 (cont.)

- SET TRANSACTION statement cannot be executed while a transaction is active
  SET TRANSACTION READ WRITE, READ COMMITTED DIAGNOSTICS SIZE 20;

- An implementation does not have to support DML and DDL statements mixed within a single transaction

- An SQL-transaction may be part of an encompassing transaction
  - If an SQL-transaction has been started by a non-SQL agent, then a COMMIT statement will raise an exception

- Keyword WORK is optional
Transaction Management: New in SQL99

- New statements for
  - Explicitly starting TXs
    - Also sets TX characteristics
  - Establishing savepoints
  - Destroying savepoints

- Extensions of SQL 92 features
  - Savepoint processing in commit, rollback
  - CHAIN option for commit and rollback
    - Initiates new TX
  - LOCAL option of set TX statement
    - If TX served by n>1 servers ("TX branches")
    - Allows different "branch" characteristics

Savepoint: Example

- Insert row into table People and establish savepoint
  
  INSERT INTO People
  VALUES('Doe', 'John', 'Hans');
  SAVEPOINT SP1

- Change that row
  
  UPDATE People SET Nick = 'Jean'
  WHERE LName = 'Doe'

- Undo the last change
  
  ROLLBACK TO SAVEPOINT SP1

- Note: TX remains open; Nickname reset to 'Hans'
Connections

- Association between an SQL-client and an SQL-server.
- There is an SQL-session associated with each connection.

```sql
env = "IBMSYS";
connect = "STLconnection";
user = "Todd";
EXEC SQL CONNECT TO :env AS :connect USER :user
...
EXEC SQL COMMIT;
env = "IBMSYS2";
EXEC SQL SET CONNECTION :env;
```

- Transactions that affect more than one SQL-environment do not have to be supported.

---

SQL Flagger

- Used to flag:
  - Extensions to the level of SQL/99 chosen
  - Conforming language that is being processed in a non-conforming way
- Must be provided with an SQL implementation

- Provides one or more of the following "levels of flagging":
  - Core SQL flagging
  - Part SQL flagging
  - Package SQL flagging

- Provides one or more of the following "extent" options:
  - Syntax only
  - Catalog lookup
Module Language

- Module definition
  module read
  Language C
  Authorization reader
  DECLARE people CURSOR FOR
      SELECT last, first
      FROM    hobbies
      WHERE    hobbies = :h
  PROCEDURE open_people (SQLSTATE, :h CHAR (5));
      OPEN people;
  PROCEDURE fetch_people (SQLSTATE, :last CHAR(20), :first CHAR(20));
      FETCH people INTO :last, :first;
  PROCEDURE close_people SQLSTATE;
      CLOSE people;
- Application program
  main()
  }
  char SQLSTATE[6];
  char last [21], first [21];
  OPEN_people (SQLSTATE, "travel");
  while...
    FETCH_people (SQLSTATE, last, first);
  }

SQL99 PSM Overview

- Procedural Extensions
  - 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
SQL/PSM

- Includes two major aspects:
  - Procedural extensions (aka control statements) - features from block-structured languages, including exception handling.
  - SQL-server modules - groups of SQL-invoked routines managed as named, persistent objects.
- Consider a C program with embedded SQL statements:

```c
void main
EXEC SQL INSERT INTO employee
VALUES (...);
EXEC SQL INSERT INTO department
VALUES (...);
}
```

- Using PSM-96 procedural extensions, the same program can be written as:

```c
void main
{
EXEC SQL
BEGIN
INSERT INTO employee VALUES (...);
INSERT INTO department VALUES (...);
END;
}
```

SQL/PSM (cont.)

- If we create a SQL procedure first:

```sql
CREATE PROCEDURE proc1 ()
{
BEGIN
INSERT INTO employee VALUES (...);
INSERT INTO department VALUES (...);
END;
}
```

- Then the embedded program can be written as:

```c
void main
{
EXEC SQL CALL proc1();
}
```
SQL Procedural Language Extensions

- Compound statement
- SQL variable declaration
- If statement
- Case statement
- Loop statement
- While statement
- Repeat statement
- For statement
- Leave statement
- Return statement
- Call statement
- Assignment statement
- Signal/resignal statement

BEGIN ... END;
DECLARE var CHAR (6);
IF subject (var <> 'urgent') THEN ...
CASE subject (var)
    WHEN 'SQL' THEN ...;
LOOP < SQL statement list> END LOOP;
WHILE i<100 DO ... END WHILE;
REPEAT ... UNTIL i<100 END REPEAT;
FOR result AS ... DO ... END FOR;
LEAVE ...;
RETURN 'urgent';
CALL procedure_x (1,3,5);
SET x = 'abc';
SIGNAL division_by_zero

Compound Statement

A compound statement is a group of SQL statements to be executed sequentially.

BEGIN [ [ NOT ] ATOMIC ]
[ <local declaration list> ]
[ <local cursor declaration list> ]
[ <local handler declaration list> ]
[ <SQL statement list> ]
END <ending label>

Example:
BEGIN
UPDATE accounts SET balance = balance-100
WHERE ...;
INSERT INTO account_history
(SELECT account#, CURRENT_DATE, 'debit', 100 FROM accounts WHERE ...);
END
A compound statement may specify ATOMIC or NOT ATOMIC. (If unspecified, NOT ATOMIC is implicit.)

```
BEGIN ATOMIC
  UPDATE accounts SET balance = balance-100
  WHERE ...;
  INSERT INTO account_history
    (SELECT account#, CURRENT_DATE, 'debit', 100 FROM accounts
     WHERE ...);
END
```

Assume UPDATE statement succeeds, but INSERT statement fails.

- If NOT ATOMIC is specified or implied, effects of UPDATE statement persist, but effects of INSERT statement are undone.
- If ATOMIC is specified, effects of both UPDATE and INSERT statement are undone if either of them fail. Does not imply transaction rollback.

Variables, cursors, conditions, and handlers can be declared inside a compound statement.

```
BEGIN ATOMIC
  DECLARE account, branch INTEGER DEFAULT 0;
  DECLARE curs1 CURSOR FOR ...
  DECLARE too_many_accounts CONDITION FOR SQLSTATE VALUE 'SS000';
  DECLARE UNDO HANDLER FOR too_many_accounts BEGIN ... END;
  ...
END
```

Variables, cursors, conditions, and condition handlers declared inside a compound statement have the scope and lifetime of the containing compound statement.

- A variable declaration must declare the name and data type. It can optionally specify a default value.
- If a condition declaration explicitly specifies a SQLSTATE value, it associates a condition name with that SQLSTATE value; otherwise, it associates a condition name with a predefined SQLSTATE value (45000).
Compound Statement (cont.)

- Compound statements can be nested.
- Normal scope rules apply, i.e., declarations in an inner compound statement occludes the declarations with the same name in an outer compound statement.
- A variable can be of any SQL data type. NULL is a valid value for a variable.
- A compound statement is associated with a label; if unspecified, an implementation-dependent label is implicit.
- The body of a compound statement may contain:
  - DDL statements
  - DML statements
  - Control statements
  - COMMIT and ROLLBACK statements (not allowed inside an ATOMIC compound statement)
  - Session and connection management statements
  - GET DIAGNOSTICS statements
  - Dynamic SQL statements

Assignment Statement

- A variable can be assigned a value either by a SELECT ... INTO or an assignment statement.
- An assignment statement uses the SET keyword.
  
  DECLARE bal DECIMAL(15,2);
  SELECT balance INTO bal FROM accounts WHERE ...;
  SET bal = 0.00;
  SET bal = (SELECT balance FROM accounts WHERE ...);
**LEAVE Statement**

- Terminates the execution of labelled statements.

```sql
outer: BEGIN
  DECLARE bal DECIMAL(15,2);
  SELECT balance INTO bal FROM accounts WHERE ...;
  IF (bal -10.0) < 0 THEN
    BEGIN
      CALL print_message(...);
      LEAVE outer;
      END;
    END;
  END;
```

**IF Statement**

- Conditional execution of statements.

```sql
BEGIN
  DECLARE bal DECIMAL(15,2);
  SELECT balance INTO bal FROM accounts WHERE ...;
  IF bal BETWEEN 0 AND 1000 THEN ...
  ELSEIF bal BETWEEN 1001 AND 2500 THEN ...
  ELSE ...
  END IF;
END;
```
**CASE Statement**

- Simple CASE statement: selects an execution path based on the result of an expression.
  
  ```sql
  CASE (SELECT status FROM accounts WHERE ...)
  WHEN 'VIP' THEN ...
  WHEN 'BUSINESS' THEN ...
  ELSE ...
  END CASE;
  ```

- Searched CASE statement: selects an execution path based on the truth value of a predicate.
  
  ```sql
  SELECT balance INTO bal FROM accounts WHERE ...;
  CASE
  WHEN bal BETWEEN 0 AND 1000 THEN ...
  WHEN bal BETWEEN 1001 AND 2500 THEN ...
  ELSE ...
  END CASE;
  ```

  In both forms, specification of ELSE case is optional. If ELSE case is unspecified, and none of the branches evaluates to TRUE, then an exception is raised.

**LOOP Statement**

- Executes a group of statements repeatedly.
- LOOP statement is a labelled statement.
- Does not allow termination test; LEAVE statement is used to exit the LOOP statement.

```sql
DECLARE X INTEGER DEFAULT 0;
L1: LOOP
  IF X > 10 THEN LEAVE L1;
  SET X = X + 1;
END LOOP;
```
WHILE Statement

- Executes a group of statements repeatedly.
- WHILE statement is a labelled statement.
- Allows termination test; terminates when the termination test evaluates to FALSE or UNKNOWN.

```plaintext
DECLARE X INTEGER DEFAULT 0;
WHILE X <= 10 DO
  SET X = X + 1;
END WHILE;
```

REPEAT Statement

- Executes a group of statements repeatedly.
- REPEAT statement is a labelled statement.
- Allows termination test; terminates when the termination test evaluates to TRUE.

```plaintext
DECLARE X INTEGER DEFAULT 0;
REPEAT
  SET X = X + 1;
UNTIL X = 10
END REPEAT;
```
**FOR Statement**

- Executes a group of statements repeatedly.
- FOR statement is a labelled statement.
- Must be associated with a query expression; terminates after the group of statements is executed for every row in the result of query expression.

```
DECLARE X INTEGER DEFAULT 0;
FOR L1 AS SELECT balance FROM accounts DO
  SET X = X + balance;
END FOR;
```

- Body of a FOR statement is not allowed to contain a LEAVE statement that refers to L1.
- A cursor is implicitly opened at the beginning of execution; closed automatically at the end of execution.
- It is possible to specify a name for the implicit cursor:

```
FOR L1 AS curs1 CURSOR FOR
  SELECT * FROM accounts WHERE balance = 0 DO
    DELETE FROM accounts WHERE CURRENT OF curs1;
  END FOR;
```

- The body of FOR statement is not allowed to contain a OPEN, FETCH, or CLOSE statement that refers to curs1.

**ITERATE Statement**

- Allowed inside a LOOP, WHILE, REPEAT and FOR statement only.
- Terminates the current iteration through the loop.
Condition Handling

- A compound statement may contain any number of condition handlers.
  - A condition handler must specify
  - A set of conditions it is prepared to handle
  - Action to perform to handle the condition
  - Where to resume the execution after handling the condition
- Action specified in a condition handler can be any SQL statement, including a compound statement.

```sql
BEGIN
  DECLARE low_balance CONDITION;
  DECLARE CONTINUE HANDLER FOR low_balance
  BEGIN
    ...
  END;
...
END
```

- A condition handler gets executed automatically when a condition it is prepared to handle is detected anytime during the execution of the containing compound statement.

Condition Handling (cont.)

- Conditions specified in a condition handler can be:
  - SQLSTATE value
  - Condition name
  - SQLEXCEPTION (all SQLSTATE values with class other than 00, 01, or 02)
  - SQLWARNING (all SQLSTATE values with class 01)
  - NOT FOUND (all SQLSTATE values with class 02)
- A condition handler may specify:
  - CONTINUE: resume the execution with the statement following the one that raised the condition.
  - EXIT: resume the execution with the statement following the compound statement.
  - UNDO: (allowed inside ATOMIC compound statements only) undo the effects of the compound statement and resume the execution with the statement following the compound statement.
- Conditions can be raised implicitly by the system or explicitly by means of SIGNAL or RESIGNAL statements.
Condition Handling (cont.)

BEGIN
DECLARE CONTINUE HANDLER FOR low_balance
BEGIN
... IF ... THEN RESIGNAL db_inconsistency END IF;
END;
...
END

- If the RESIGNAL statement specifies a condition name, then a new condition is pushed onto the diagnostics area, which becomes the active condition; otherwise, the condition that caused the handler to execute continues to be the active condition.
- The condition handler and the compound statement terminate their execution and execution resumes with the condition handler associated with the outer compound statement.
- An implicit RESIGNAL statement gets executed if a compound statement or a handler action completes with a condition other than successful completion.

Embedding of Control Statements

- All control statements can be embedded in a host language program.
- Only CALL statement is allowed to be dynamically prepared and executed.
SQL99 Bindings Overview

- Embedded SQL
  - ADA
  - C
  - Cobol
  - Fortran
  - Mumps
  - Pascal
  - PL/I
- Dynamic SQL
- Direct SQL

Embedded SQL

- An embedded host language program is transformed into a pure host language program and an "abstract" SQL module

- SQL modules are the way used for the standards to describe the semantics of embedded SQL (don't need to be implemented this way)
Dynamic SQL

- Needed when the tables, columns, or predicates are not known when the application is written

- Execute statement immediately (once)
  
  ```
  s = "INSERT INTO people VALUES ('Harris', ...)"; 
  EXEC SQL EXECUTE IMMEDIATE :s; 
  ```

- Execute statement more than once
  
  ```
  EXEC SQL PREPARE stmt FROM :s; 
  EXEC SQL EXECUTE stmt; 
  EXEC SQL EXECUTE stmt; 
  ```

- Dynamic parameter makers
  
  ```
  s = "INSERT INTO people VALUES (?, ?, ...)"; 
  EXEC SQL PREPARE stmt FROM :s; 
  lname = "Harris"; 
  fname = "Todd"; 
  EXEC SQL EXECUTE stmt USING :lname, :fname, ... : 
  ```

- Descriptor area can be used if the number of dynamic parameters is not known.

- Descriptor area is an encapsulated structure managed by the DBMS

- Different from commercial practice (where application allocates a SQLDA)
  
  ```
  s = "INSERT INTO people VALUES (?, ?, ...)"; 
  EXEC SQL PREPARE stmt FROM :s; 
  EXEC SQL ALLOCATE DESCRIPTOR 'input_params'; 
  EXEC SQL DESCRIBE INPUT stmt INTO SQL DESCRIPTOR 'input_params'; 
  EXEC SQL GET DESCRIPTOR 'input_params' :n = COUNT; 
  for (i = 1; i < n; i++) 
  { 
    EXEC SQL GET DESCRIPTOR 'input_params' VALUE :i, 
      :t = TYPE, ... ; 
    EXEC SQL SET DESCRIPTOR 'input_params' VALUE :i 
      DATA = :d, INDICATOR = :ind; 
  } 
  EXEC SQL EXECUTE stmt USING SQL DESCRIPTOR 'input_params'; 
  ```

- Implicit conversions from database to descriptor area (and vice versa) and from descriptor area to application program (and vice versa)

- Descriptor area can be used as intermediate location for data
**Dynamic SQL**

- Extended dynamic statement names, cursor names, and descriptor area names may be used when the number of dynamic statements is not known at the time the application is written

```sql
s = "...";
s_stmt = "my_stmt";
s_desc = "my_descriptor";
s_cursor = "my_cursor";

EXEC SQL PREPARE :s_stmt FROM :s;
EXEC SQL ALLOCATE DESCRIPTOR :s_desc;
EXEC SQL ALLOCATE :s_cursor CURSOR FOR :s_stmt;
EXEC SQL OPEN :s_cursor;
```

- Extended names
  - may be LOCAL or GLOBAL (referring to the scope of a module or session)

```sql
PROCEDURE (... :s_cursor, ...);
ALLOCATE GLOBAL :s_cursor CURSOR FOR ...;
```

**Direct SQL**

- Implementation-defined mechanism for executing direct SQL statements
  - In effect, prepared immediately before execution
  - Cannot issue dynamic SQL using direct SQL

- Invocation, method of raising error conditions, method of accessing diagnostics information, and the method of returning results are all implementation-defined
Call Level Interface Overview

- An alternative mechanism for invoking SQL from application programs
  - Similar to dynamic SQL
- Provided for vendors of truly portable "shrink wrapped" software
  - CLI does not require pre-compilation of the application program
  - Application program can be delivered in "shrink wrapped", object-code form
- It is not:
  - Some new way of achieving interoperability
  - An alternative to distributed database protocols such as ISO's RDA
- Based on
  - CLI from SQL Access Group (SAG) and X/Open
  - OBDC (Open DataBase connection)

Call Level Interface (cont.)

- Functional interface to database
- Consists of over 40 routine specifications
  - Control connections to SQL-servers
  - Allocate and deallocate resources
  - Execute SQL statements
  - Control transaction termination
  - Obtain information about the implementation

```
application

CLI call handler
(sometimes called CLI driver)

Database Manager - Run Time Support
```

303
Call Level Interface

- Uses handles to "manage" resources
  - Environment is the root of all capabilities
  - Other handles exist in the context of an environment
  - Connection handles manage connections to "servers"
  - Statement handles manage SQL statements and cursors
- SQL/CLI behaves much like dynamic SQL
- Uses "CLI Descriptor Area"
  - Analogous to dynamic SQL's system descriptor area, but ...
    - CLI has four descriptors
      - Application parameter descriptor (APD)
      - Application row descriptor (ARD)
      - Implementation parameter descriptor (IPD)
      - Implementation row descriptor (IRD)

C Example (page 1)

```c
#include "sqlcl.h"
#include <string.h>

#ifndef NULL
#define NULL 0
#endif

int print_err(HDBC hdbc, HSTMT hstmt);
int example1 (server, uid, pwd)
  SQLCHAR *server;
  SQLCHAR * uid;
  SQLCHAR * pwd;
}

HENV henv;
HDBC hdbc;
HSTMT hstmt;
SQLINTEGER id;
SQLCHAR name[51];
SQLINTEGER namelen;
SQLSMALLINT scale;
```
C Example (page 2)

/* connect to database */
/* EXEC SQL CONNECT TO :server USER :uid using :auth_string; */

SQLAllocENV(&henv); /* allocate an environment handle */
SQLAllocConnect(henv, &hdbc); /* allocate a connection handle */
if (SQLConnect(hdbc, server, SQL_NTS, uid, SQL_NTS, pwd, SQL_NTS) != SQL_SUCCESS)
    return( print_err(hdbc, SQL_NULL_HSTMT) );

/* create a table */
/* EXEC SQL CREATE TABLE NAMEID (ID integer, NAME Varchar(50)); */
SQLAllocStmt(hdbc, &hstmt); /* allocate a statement handle */

SQLCHAR create[] = "CREATE TABLE NAMEID (ID integer, NAME varchar(50))":
if SQLExecDirect(hstmt, create, SQL_NTS) != SQL_SUCCESS)
    return(print_err(hdbc, hstmt));

/* commit the created table */
/* EXEC SQL COMMIT WORK; */
SQLEndTran(henv, hdbc, SQL_COMMIT):

C Example (page 3)

/* insert a row into the table */
/*EXEC SQL INSERT INTO NAMEID VALUES (:id, :name );*/
/*EXEC SQL COMMIT WORK ;*/

SQLCHAR insert[] = "INSERT INTO NAMEID VALUES (?, ?)":

/* prepare the insert */
if (SQLPrepare(hstmt, insert, SQL_NTS) != SQL_SUCCESS)
    return(print_err(hdbc, hstmt));

SQLBindParam(hstmt, 1, SQLBUF_LONG, SQL_INTEGER, (SQLINTEGER)sizeof(SQLINTEGER), scale, (SQLPOINTER)&id, (SQLINTEGER*)NULL);
SQLBindParam(hstmt, 2, SQLBUF_CHAR, SQL_VARCHAR, (SQLINTEGER)sizeof(name), scale, (SQLPOINTER)name, (SQLINTEGER*)NULL);

/* now assign parameter values and execute the insert */

id=500
(void)strcpy(name, "Babbage");
if (SQLExecute(hstmt) != SQL_SUCCESS)
    return(print_err(hdbc, hstmt));

} SQLEndTran(henv, hdbc, SQL_COMMIT); /* commit inserts */
/* fetch a row from the table */
/*EXEC SQL DECLARE c1 CURSOR FOR SELECT ID, NAME FROM NAMEID;*/
/*EXEC SQL OPEN c1; */
{
    SQLCHAR Select[] ="select ID, NAME from NAMEID",
    if(SQLExecDirect(hstmt, select, SQL_NTS) !=SQL_SUCESS)
        return(print_err(hdbc, hstmt));
}
/*EXEC SQL FETCH c1 INTO :id, :name,*/
/*use column binding */
SQLBindCol(hstmt, 1, SQLBUF_LONG, (SQLPOINTER)&id,
    (SQLINTEGER)sizeof(SQLINTEGER), (SQLINTEGER *)NULL);
SQLBindCol(hstmt,2,SQLBUF_CHAR, (SQLPOINTER)name,
    (SQLINTEGER)sizeof(name), &namelen);
/*now execute the fetch*/
SQLFetch(hstmt);
/* commit, discard hstmt, disconnect*/
/*EXEC SQL COMMIT WORK;*/
/*EXEC SQL CLOSE c1; */
/*EXEC SQL DISCONNECT; */
SQLEndTran(henv, hdbc, SQL_COMMIT); /*commit the transaction */
SQLFreeStmt(hstmt, SQL_DROP); /* free the statement handle*/
SQLDisconnect(hdbc) /*disconnect from the database*/
SQLFreeConnect(hdbc) /* free the connection handle*/
SQLFreeEnv(henv) /*free the environment handle*/
return(0); }
New CLI99 Features

- SQL99 data type support
  - BOOLEAN
  - LOBs with optional locators and helper routines (GetLength,GetPosition,GetSubstr)
  - UDTs with locators and transformation functions
  - Arrays with locators only
  - Reference types with table scope
  - Can retrieve/store unnamed ROW types

New CLI99 Features

- CLI descriptor model aligned with ODBC 3.x (defaults, Get/Set restrictions, etc.)
- JDBC 2.0 support for user-defined types
- Multi-row fetch a la ODBC
- Catalog routines aligned with SQL99 and ODBC
- Parallel result set processing after CALL statement
- SQL99 savepoints
- General SQL99 alignment (roles, user-defined casts, SQLSTATEs, etc.)
Conformance

- SQL-92 used incremental levels of conformance (Entry, Intermediate, Full)
- SQL99 consists of a large number of small "features", each identified and precisely specified as to its content
- Each feature is specified either to be a constituent of "Core SQL", or not a constituent of Core SQL
- A non-core feature might be specified as a constituent of one of the named and defined "Packages", each of which require conformance to Core.

Overview of SQL99

Core Features

- All of SQL-92 Entry level
- Some Transitional SQL-92 features
- Some Intermediate SQL-92 features
- Some Full SQL-92 features
- The following new features of SQL3
  - Distinct data types, including USER_DEFINED_TYPES view
  - WITH HOLD cursors
  - SQL-invoked routines, but not the ability to explicitly specify a PATH:
    - CALL statement (with the extension to dynamic SQL to support CALL)
    - RETURN statement
    - ROUTINES and PARAMETERS view
  - SQL-invoked routines written in both SQL and an external language (one can conform by supporting only one)
  - Value expression in order by clause
Core SQL99 Features

- Numeric data types
  - All spellings of INTEGER and SMALLINT
  - REAL, DOUBLE PRECISION, FLOAT
  - DECIMAL and NUMERIC
  - Arithmetic operators
  - Numeric comparison
  - Implicit casting among numeric data types

- Character data types
  - CHARACTER (all spellings)
  - CHARACTER VARYING (all spellings)
  - Character literals
  - Functions
    - CHARACTER_LENGTH
    - OCTET_LENGTH
    - SUBSTRING
    - UPPER
    - LOWER
    - TRIM
    - POSITION
  - Character concatenation
  - Implicit casting among character data types
  - Character comparison

- Identifiers
  - Delimited identifiers
  - Lower case identifiers
  - Trailing underscore

- Basic query specification
  - SELECT distinct
  - GROUP BY clause
  - GROUP BY with columns not in column list
  - AS clause
  - HAVING clause
  - Qualified * in select list
  - Correlation names in FROM
  - AS in FROM clause (rename columns)

- Basic predicates and search conditions
  - Comparison predicate
  - BETWEEN predicate
  - IN predicate with list of values
  - LIKE predicate
  - LIKE predicate with ESCAPE clause
  - NULL predicate
  - Quantified comparison predicate
  - EXISTS predicate
  - Subqueries in comparison predicate
  - Subqueries in IN predicate
  - Subqueries in quantified comparison predicate
  - Correlated subqueries
  - Search condition

- Basic query expressions
  - UNION ALL
  - EXCEPT DISTINCT

- Basic privileges
  - SELECT, DELETE
  - INSERT at table level
  - UPDATE at table and column levels
  - REFERENCES at table and column levels
  - WITH GRANT OPTION

- SET functions
  - AVG, COUNT, MAX, MIN, SUM
  - ALL and DISTINCT quantifiers

- Basic data manipulation
  - INSERT statement
  - Searched UPDATE, DELETE
  - Single-row SELECT statements

- Basic cursor support
  - DECLARE CURSOR
  - ORDER BY columns need not be in SELECT
    - Value expressions in ORDER BY clause
    - OPEN, CLOSE, FETCH (implicit NEXT)
    - Positioned UPDATE and DELETE
    - WITH HOLD cursors

- Null value support

- Basic integrity constraints
  - NOT NULL constraints
  - UNIQUE constraints of NOT NULL columns
  - PRIMARY KEY constraints
  - Basic FOREIGN KEY constraints with the NO ACTION default for both referential delete and referential update action
  - CHECK constraints
  - Column defaults
  - NOT NULL inferred on PRIMARY KEY
  - Names in a foreign key can be specified in any order

- Transaction support
  - COMMIT and ROLLBACK
  - Module language (at least one binding to a standard host language using either module language, embedded SQL, or both)

- Basic information schema views
  - COLUMNS, TABLES, VIEWS, TABLE_CONSTRAINTS, REFERENTIAL_CONSTRAINTS, CHECK_CONSTRAINTS
Core SQL (cont.)

Basic schema manipulation
- CREATE TABLE for persistent base tables
- CREATE VIEW
- GRANT
- ALTER TABLE ADD COLUMN
- DROP TABLE, DROP VIEW, and REVOKE, all with RESTRICT clause

Basic joined table
- INNER join (but not necessarily INNER keyword)
- LEFT and RIGHT OUTER JOIN
- Nested outer joins
- The inner table in a left or right outer join can also be used in an inner join
- All comparison operators are supported

Basic date and time
- DATE data type and DATE literal
- TIME data type with fractional seconds precision of at least 0 (also literal)
- TIMESTAMP data type (and literal) with fractional seconds precision of at least 0 and 6
- Comparison predicate on DATE, TIME, and TIMESTAMP data types
- Explicit CAST between datetime types and character types
- CURRENT_DATE, LOCALTIME, and LOCALTIMESTAMP functions
- UNION and EXCEPT in views

Grouped operations
- Multiple tables supported in queries with grouped views
- Set functions supported in queries with grouped views
- Subqueries with GROUP BY and HAVING clauses and grouped views
- Single row SELECT with GROUP BY and HAVING clauses and grouped views
- The ability to associate multiple host compilation units with a single SQL-session at one time
- CAST function where relevant for all supported data types
- Explicit defaults including its use in UPDATE and INSERT statements
- CASE expressions
  - Simple and searched
  - NULLIF
  - COALESCE
- Schema definition statement
  - CREATE SCHEMA
  - CREATE TABLE for persistent base tables
  - CREATE VIEW
  - CREATE VIEW: WITH CHECK OPTION
  - GRANT statement
- Scalar subquery values
- Expanded NULL predicate (the <row value expression> can be something other than a <column reference>)

Features and conformance views
- SQL_FEATURES, SQL_SIZING, and SQL_LANGUAGE views

Basic flagging
- Core SQL level
- Syntax Only extent

Distinct data types
- USER_DEFINED_TYPES view

Basic SQL-invoked routines
- "Routine" is the collective term for functions, methods, and procedures
- Overloading for functions and procedures is not part of Core
- Function invocation
- CALL and RETURN statements
- ROUTINES and PARAMETERS views
 Packages

PKG001  Enhanced Datetime Facilities
PKG002  Enhanced Integrity Management
PKG003  OLAP Features
PKG004  PSM (i.e., Part 4)
PKG005  CLI (i.e., Part 3)
PKG006  Basic Object Support
PKG007  Enhanced Object Support
PKG008  Active Database (Triggers - row-level only)
PKG009  SQL/MM Support

Others might be defined, not necessarily in the SQL standard itself.

 PKG001 - Enhanced DATETIME Facilities

- Intervals and datetime arithmetic
- Time zones
- Enhanced seconds precision
  (sub-microsecond)
PKG002 - Enhanced Integrity Management

ON DELETE ....
ON UPDATE ...
CREATE ASSERTION
- Constraint Management
  via constraint names
- Subqueries in CHECK constraints
- Triggers
  row-level and statement-level

PKG003 - OLAP Facilities

CUBE, ROLLUP
INTERSECT, EXCEPT ALL
FULL JOIN
- Derived tables in the FROM clause
- More than one row in VALUES
PKG006 - Basic Object Support

- Structured types with restrictions on use
- Reference types, with restrictions
- Typed base tables
- Predicate to test most specific type of a value
- Basic LOB support including locators

PKG007 - Enhanced Object Support

- ALTER TYPE
- Static methods
- Structured types without restrictions
- Reference types without restrictions
- Schema paths
- Subtables
- TREAT
- User-defined casts and transforms
- Locators for structured type values
PKG009 - SQL/MM Support

- Structured types without restriction
- Arrays/incl. arrays of structured types
- User-defined cast
- Overloading (routines with same name in same schema)

SQL/MM Motivation

- Enabling functionality

  - Definition of user-defined, application specific data types
  - Implementation of user-defined functions to support application specific operations on the data types
  - Storage of large objects (BLOBs and CLOBs)
  - Powerful trigger and constraint mechanisms to maintain the integrity and semantics of the new data types
  - Storage and execution of user-defined stored procedures in the server

- This enables ...

  - Development of application specific collections of user-defined types, user-defined functions, triggers, constraints, and stored procedures (i.e. libraries) "tight" to the DBMS engine
SQLMM Overview

Multipart standard:

- SQL/MM Framework (Part 1)
  - Overview and conformance

- SQL/MM Full-text (Part 2)
  - Information about construction of text and search patterns, and for the searching of text

- SQL/MM Spatial (Part 3)
  - Information about storing, managing, and retrieving information related to spatial data such as geometry and topography

- SQL/MM Still-image (Part 5)
  - Information about searching large collections of still images

SQL/MM Full-Text

- Why Full-Text standard library?
  - Built-in search facilities (LIKE, SIMILAR) not powerful enough (text viewed as string of characters).
  - Need higher level notion of text

- Structural units in Full-Text:
  - Words
  - Sentences
  - Paragraphs

- Operations in Full-Text:
  - Boolean Search
  - Ranking
  - Conceptual Search
SQL/MM Full-Text: Boolean Search

- Full-Text sample:
  Every text value is associated with a specific language.
- Full-Text items have language attribute
- Boolean query facilities
  - Single word search
  - Phrase search
  - Context based search
  - Linguistic search
  - Stopword processing
  - Masking facilities
  - Search pattern expansion, e.g.:
    - Sound expansion
    - Broader/narrower term expansion
    - Synonym expansion

SQL/MM Full-Text: Boolean search examples

- Single word search:
  ```sql
  SELECT * FROM myDocs
  WHERE 1 = CONTAINS(TextBody, 'specific')
  every text value is associated with a specific language.
  ```
- Phrase search:
  ```sql
  SELECT * FROM myDocs WHERE 1 =
  WHERE 1 = CONTAINS(TextBody, 'specific language')
  every text value is associated with a specific language.
  ```
- Context search:
  ```sql
  SELECT * FROM myDocs WHERE 1 = CONTAINS(TextBody, '
  "text IN SAME SENTENCE AS language"
  every text value is associated with a specific language.
  ```
- Stopwords:
  ```sql
  SELECT * FROM myDocs WHERE CONTAINS(TextBody, '
  "value was associated"
  every text value is associated with a specific language.
  ```
- Linguistic search:
  ```sql
  SELECT * FROM myDocs WHERE CONTAINS(TextBody, '
  "STEMMED FORM OF "values are associated"
  every text value is associated with a specific language.
  ```
**SQL/MM Full-Text**

- **Ranking**
  
  ```sql
  SELECT * FROM myDocs
  WHERE 1.2 < RANK(TextBody, 'specific')
  ```
  
  - Ranks according to implementation-defined criteria (e.g., frequency of "specific")

- **Conceptual search**
  
  ```sql
  SELECT * FROM myDocs
  WHERE 1 = CONTAINS(TextBody, 'IS ABOUT "every text value is associated with a specific language"')
  ```
  
  - Identifies Full-Text items which are pertinent to rhs of "IS ABOUT" operator

**SQL/MM Spatial: Goals, Motivation**

- **Goals**
  
  - Support for "flat world" (2-d) geometric objects and operations
  - Coverage of important application areas
  - Simple features

- **Motivation**
  
  - Breaking ground for standard type library
  - Promote efficient access methods on relational platforms
**SQL/MM Spatial: Players**

- JTC1 SC32 WG4: SQL/MM Spatial
- ISO TC211: Geomatics
- Open GIS Consortium:
  - OpenGIS Simple Feature Specification
    - SQL2 Bindings
    - CORBA Binding
    - OLE Binding
  - SQL3 Bindings: SQL/MM Spatial
    - Guarantees implementations
    - Established verification procedures

---

**Spatial Objects**

- o-dim. objects: points
- 1-dim. objects: (planar) curves; sub-types differ by interpolation between points
  - ST_LineString: linear interpolation
  - ST_CircularString (opt): circular arcs
  - ST_CompoundString (opt): mixed
- 2-dim. objects: (planar) surfaces
  - ST_Polygon: ST_LineString boundaries
  - ST_CurvePolyon (opt): ST_CompoundString boundaries
Spatial Objects (cont.)

- Collection valued objects:
  - ST_Geometry
    - Reference system: same for all elements
    - Any geometry type admissible
    - Subtypes of ST_Geometry with restrictions on element types
      - ST_MultiPoint
      - ST_MultiCurve
      - ST_MultiPolygon

SQL/MM Spatial: Operations

- Usual observers and mutators
- Transform routines
  - Transform objects into binary or textual representations (and vice versa)
  - Enables implementation by 3GL functions using minimal SQL3 machinery
- Important topical operations, e.g.
  - Constructors (controlling wellformedness)
  - Distance
  - Tests (contains, overlaps, touches, crosses, ...)
  - Intersection, difference, union
  - Find referencing system
  - Length, area, perimeter
SQL/MM Spatial: Example

```
SELECT * FROM stores s, customers c
WHERE within(c.loc, s.zone)
  or distance(c.loc, s.loc)<100
ORDER BY s.name, c.name;
```

"Tell me all the information I have about each customer who either lives within a store's zone or within 100 miles of the store."

---

SQL/MM Spatial: Type Hierarchy

```
ST_Geometry
  └── ST_GeomCollection
    └── ST_MultiSurface
        │   └── ST_MultiPolygon
        │       └── ST_Polygon
        │           └── ST_MultiPoint
        └── ST_MultiCurve
          └── ST_MultiLineString
```

---
Spatial Reference System

- Controls aspects like units, prime meridian, coordinate system etc.
- Relies on reference systems defined by other authorities.
- Defined representation of reference system values
- One common spatial reference system value:
  - For elements of ST_Geometry values
  - Within column of type ST_Geometry

SQL/MM Still-Image: Goals

- Enable screening of large imagebases
- Support for proven set of image features
- Type structure adaptable to evolving image processing technology
- Example: Find all possibly infringed logos by scoring them against a new logo.

```sql
SELECT * FROM RegLogos
WHERE 1.2 < SI_findTexture(newLogo).SI_Score(Logo)
```
**SQL/MM Still-Image Objects**

- SI_StillImage: raster images
- Abstract SI Feature with subtypes
  - SI_AverageColor
  - SI_ColorHistogram
  - SI_PositionalColor: average colors of n*m image segments
  - SI_Texture: coarseness, contrast, directionality
- SI_FeatureList: weighted list of SI_Feature items

**SI_StillImage: Operations**

- Constructor function
- Observer Methods for
  - Raw picture data
  - Image format (e.g. JPEG)
  - Pixel properties (bits per color, per pixel)
  - Size ..
  - Generation time, last update time
- Mutator for (raw) image content
SI_Feature, SI_FeatureList: Operations

- All: scoring method (SI_Score)
  - Scores image w.r.t. a given feature
- All subtypes of SI_Feature
  - function extracting feature from images
- SI_AverageColor, SI_ColorHistogram
  - function for "manual" feature construction
- SI_FeatureList: feature/weight pairs lists
  - Constructor function for list header
  - Append method to extend feature list by another feature/weight pair

SQL/MM Still-Image: Final Example

Screen all logos in table RegLogos against a given logo (newLogo); use the texture and average colors of a standard grid of image segments ("positional color") for scoring; give these features of newLogo the weights 80% and 20%, resp.

```sql
SELECT * FROM RegLogos
WHERE 1.2 <
  SI_InitFeatureList
  (SI_findTexture(newLogo), 0.8).SI_Append
  (SI_findPositionalColor(newLogo), 0.2)
  .SI_Score(Logo)
```
SQLJ

- SQLJ Part 0
  - Embedded SQL in Java
  - Currently based on SQL-92, JDBC 1.2
  - Accepted ANSI standard "Database Language - SQL, Part 10 Object Language Bindings (SQL/OLB)", ANSI X3.135.10:1998
  - Currently being processed by ISO, (SQL-99 alignment)

- SQLJ Part 1
  - Java static methods as SQL UDFs and stored procedures SQL routines (stored procedures, user-defined functions)
  - Currently a working draft, being prepared for submission to ANSI

- SQLJ Part 2
  - Use of Java classes to define SQL types
  - Potential for wide DBMS vendor acceptance
  - Cloudbase, Compaq (Tandem), IBM, Informix, MicroFocus, Oracle, Sun, Sybase
  - Tremendous possibilities
    - Baan, PeopleSoft, SAP exploitation?
    - the "next ODBC"?

SQLJ Part 0: Overview

- Static SQL syntax for Java
  - INSERT, UPDATE, DELETE, CREATE, GRANT, etc.
  - Singleton SELECT and cursor-based SELECT
  - Calls to stored procedures (including result sets)
  - COMMIT, ROLLBACK
  - Methods for CONNECT, DISCONNECT

- Similar tradeoffs
  - Static vs. dynamic
  - SQLJ vs. JDBC
  - Less flexible at run-time
  - Allows error checking at development time
  - Static SQL is faster!!!
SQLJ Part 0: Overview (cont.)

- Static SQL authorization
  - Static SQL is associated with "program"
    - Plans/packages identify "programs" to DB
    - Program author's table privileges are used
    - Users are granted EXECUTE on program
  - Dynamic SQL is associated with "user"
    - No notion of "program"
    - End users must have table privileges
    - BIG PROBLEM FOR A LARGE ENTERPRISE !!!
- SQLJ programs are smaller than JDBC applications
- Can be used in client code and stored procedures
  - Easier than JDBC, better performance too!
- Binary portability

SQLJ Syntax

- SQLJ clauses are statements or declarations
  - Clause begins with "#sql" token
- An SQL statement appears as an SQLJ statement clause
  - May contain host-variable references (e.g., :x) or host expressions (e.g., :(x + y))
  - Can span multiple lines
  - May specify explicit connection or use default connection

```sql
#sql [ [<context>] ] { <statement spec clause> }
```
SQLJ vs. JDBC: Retrieve Single Row

**SQLJ**

```sql
#sql [con] { SELECT ADDRESS INTO :addr FROM EMP WHERE NAME=:name ; }
```

**JDBC**

```java
java.sql.PreparedStatement ps = con.prepareStatement("SELECT ADDRESS FROM EMP WHERE NAME=?");
ps.setString(1, name);
java.sql.ResultSet names = ps.executeQuery();
names.next();
name = names.getString(1);
names.close();
```

---

SQLJ vs. JDBC: Insert One Row

**SQLJ**

```sql
#sql [con] {INSERT INTO T1 VALUES( :hv1, :hv2, :hv3) ; }
```

**JDBC**

```java
CallableStatement mystmt =
    con.prepareCall("INSERT INTO T1 VALUES(?,?,?)");
mystmt.setString(1,hv1);
mystmt.setString(2,hv2);
mystmt.setInt(3,hv3);
mystmt.executeUpdate();
```
**SQLJ vs. JDBC: Call Stored Procedure**

- **SQLJ**
  ```sql
  #sql [con] {CALL PROC1(:IN hv1, :OUT hv2) };
  ```

- **JDBC**
  ```java
  CallableStatement mystmt =
      con.prepareCall("CALL PROC1(? ,?)");
  mystmt.setString(1, hv1);
  mystmt.registerOutParameter(2, java.sql.Types.VARCHAR);
  mystmt.executeUpdate();
  hv2 = mystmt.getString(2);
  ```

**Result Set Iterators**

- Mechanism for accessing the rows returned by a query
  - Comparable to an SQL cursor
- SQLJ Iterator declaration clause results in generated iterator class
  - Iterator is a Java object
  - Iterators are strongly typed
  - Generic methods for advancing to next row
- SQLJ assignment clause assigns query result to iterator
- Two types of iterators
  - Named iterator
  - Positioned iterator
**Named Iterator**

- Generated iterator class has accessor methods for each result column

```java
#sql iterator Honors ( String name, float grade );
Honors honor;
#sql [recs] honor =
    { SELECT SCORE AS "grade", STUDENT AS "name"
      FROM GRADE_REPORTS
      WHERE SCORE >= :limit AND ATTENDED >= :days AND
          DEMERITS <= :offences
      ORDER BY SCORE DESCENDING };
while (honor.next()) {
    System.out.println( honor.name() + " has grade "
        + honor.grade() );
}
```

**Positioned Iterator**

- Use FETCH statement to retrieve result columns into host variables based on position

```java
#sql iterator Honors ( String, float );
Honors honor;
String name;
float grade;
#sql [recs] honor =
    { SELECT STUDENT, SCORE FROM GRADE_REPORTS
      WHERE SCORE >= :limit AND ATTENDED >= :days AND
          DEMERITS <= :offences
      ORDER BY SCORE DESCENDING };
while (true) {
    #sql (FETCH :honor INTO :name, :grade );
    if (honor.endFetch()) break;
    System.out.println( name + " has grade " + grade );
}
```
Connection Contexts

- Used to associate execution of SQL statements with a database connection
- Explicit connection context
  - Declare connection context class
    \texttt{#sql context DB1con;}
  - Create connection context object
    \texttt{String url = "jdbc:....."; \ DB1con con = new DB1con(url, "user", "password", false);}
  - Use connection context in SQLJ statement clause
    \texttt{#sql [con] \{SELECT c INTO \:x FROM mytable\};}
- Default connection context
  - Used if no explicit connection context is specified in SQLJ clause
  - Important usage: stored procedures written in SQLJ
    - Default context provided by database environment
- Application can use multiple connections to the same or different databases at the same time
- Connections can be shared across threads in a multi-threaded application

Execution Contexts

- Describes execution semantics of SQL operations
  - Control execution environment
    - \texttt{MaxRows, MaxFieldSize, QueryTimeout}
  - Get description of results of SQL statement execution
    - \texttt{UpdateCount, SQLWarnings}
- Example
  - Create new execution context
    \texttt{ExecutionContext exec = new ExecutionContext();}
  - Set execution context attribute
    \texttt{exec.setQueryTimeout(3); // wait only 3 seconds}
  - Use execution context in SQLJ statement clause
    \texttt{#sql [con, exec] \{ DELETE FROM mytable WHERE ...\};}
  - Get execution information
    \texttt{System.out.println}  
    \texttt{("deleted " + exec.getUpdateCount() + " rows");}
Advanced Features

- Multiple result sets from stored procedures
  - Side-channel result sets
  - Use method "getNextResultSet" on execution context to navigate through results
- SQLJ and JDBC interoperability
  - Mixing SQLJ (static SQL) and JDBC (dynamic SQL) in the same application
  - SQLJ and JDBC can share the same connections
  - JDBC result sets can be turned into SQLJ iterators, and vice versa

Compiling an SQLJ Application

Most vendors use default JDBC "stub"
Binary Portability

- Static SQL portability problems
  - 3GL language not 100% portable
  - Each DBMS has unique precompiler output
    - No binary portability across DBMSs
- SQLJ advantages
  - Java is platform-independent
    - Compiled SQLJ applications are pure Java
  - Generic SQLJ translator (works for all DBMSs)
  - SQLJ application binaries (Java bytecodes) are portable across DBMSs
  - Vendor-specific customizations can be performed after compilation
    - Performance optimizations, ...

SQLJ Part 1: Overview

- Java static methods as SQL UDFs and stored procedures
  - Can contain JDBC or SQLJ calls
- Many advantages
  - Processing power on database server
  - Reduce volume of data transfer by sending final answer sets
  - Centralize the administration of the business logic
  - Access operations not available on client/gateway tier
  - Direct use of pre-written Java libraries
  - Portable across DBMSs and platforms
  - Deployable across different tiers
- Invocation
  - Can be called by any Java or non-Java client code
  - DBMS invokes the JVM to run the Java application
  - DBMS handles type conversions between Java and SQL
Installing Java Classes in the DB

- Installation
  - New install_jar procedure
    `sqlj.install_jar ('file:~/classes/routines.jar', 'routines_jar')`
  - Parameters: URL of JAR file with Java class and string to identify the JAR in SQL
  - Install all classes in the JAR file
  - Uses Java reflection to determine names, methods, signatures
  - Optionally uses deployment descriptor file found in JAR to create SQL routines

- Removal
  - `sqlj.remove_jar ('routines_jar')`

- Replacement
  - `sqlj.replace_jar ('file:~/classes/routines.jar', 'routines_jar')`

Creating Procedures and UDFs

```
sqlj.install_jar ('file:~/classes/routines.jar', 'routines_jar')
```

Java return type 'void' -> stored procedure
otherwise -> user-defined function

```
CREATE PROCEDURE modify_address (ssn INTEGER, addr CHAR (40))
MODIFIES SQL DATA
EXTERNAL NAME 'routines_jar:addr.modifyaddr'
LANGUAGE JAVA
PARAMETER STYLE JAVA
```

```
CREATE FUNCTION zip (addr CHAR (40)) RETURNS INTEGER
NO SQL
DETERMINISTIC
EXTERNAL NAME 'routines_jar:addr.zip'
LANGUAGE JAVA
PARAMETER STYLE JAVA
```

Invoking SQLJ Routines

- Privileges
  - Usage privilege on installed JAR file is grantable
    GRANT USAGE ON JAR routines_jar TO bryan
  - Execute privilege on routines is grantable
    GRANT EXECUTE ON modify_address TO bryan
    GRANT EXECUTE ON zip TO bryan

- Invocation
  - User-defined function
    SELECT zip (home_addr) FROM employees
  - Stored procedure
    CALL modify_address (64148342, '1664 Tunis Rd, San Bruno, CA')

SQLJ Stored Procedures

- OUT and INOUT parameters
  - CREATE PROCEDURE
    avgSal (IN dept VARCHAR(30), OUT avg DECIMAL(10, 2))
    ...
  - Java method declares them as arrays
  - Array acts as container that can be filled/replaced by the method implementation to return a value
  - public static void averageSalary (String dept, BigDecimal[] avg) ...

- Returning result set(s)
  - CREATE PROCEDURE ranked_emps (region INTEGER)
    DYNAMIC RESULT SETS 1 ...
  - Java method declares explicit parameters for returned result sets of type
    • array of (JDBC) ResultSet
    • array of (SQLJ) iterator class, prev. declared in "#sql iterator ...
  - public static void ranked_emps (int region, ResultSet[] rs) ...
  - Java method body assigns (open) result sets as array elements of result set parameters
  - Multiple result sets can be returned
Error Handling

- Java method throws an SQLException to indicate error to the SQL engine
  - ... throws new SQLException ("Invalid input parameter", "38001");
  - SQLSTATE value provided has to be in the "38xxx" range
- Any other uncaught Java exception is turned into a SQLException "Uncaught Java exception" with SQLSTATE "38000" by the SQL engine
- Java exceptions that are caught within an SQLJ routine are internal and do not affect SQL processing

Additional Features

- Java "main" methods
  - Java signature has to have single parameter of type String[]
  - Corresponding SQL routine has
    - Either 0 or more CHAR/VARCHAR parameters,
    - or a single parameter of type array of CHAR/VARCHAR
- NULL value treatment
  - Use Java object types as parameters (see JDBC)
    - SQL NULL turned into Java null
  - Specify SQL routine to return NULL if an input parameter is NULL
    - CREATE FUNCTION foo(integer p) RETURNS INTEGER
      RETURNS NULL ON NULL INPUT
  - Otherwise run-time exception will be thrown
- Static Java variables
  - Can be read inside SQL routine
  - Should not be modified (result is implementation-defined)
- Overloading
  - SQL rules may be more restrictive
  - Map Java methods with same name to different SQL routine names
Conformance

- For SQLJ Part 1 Core compliance, a lot of features are optional, such as
  - DDL vs. deploy/undeploy descriptors in SQLJ JAR files
    - Either DDL or descriptors for CREATE PROCEDURE/FUNCTION has to be supported
  - SQLJ Paths
  - INOUT, OUT parameters for stored procedures
  - support for returning result sets in stored procedures
  - SQL array datatype for support Java 'main' arguments
  - GRANT statement, WITH GRANT OPTION
  - REVOKE statement

JDBC 2.0 Extensions for SQL99 Types

- Added support for object-relational data types
  - User-defined types (UDTs)
    - Structured types
    - Distinct types
  - References
    - Arrays (collection types)
- Supports mappings of DB structured types to/from Java classes
  - Focus on object state, not on interface (behavior)
  - Provides sufficient basis for mapping tools
  - Allows application to provide mapping information to JDBC driver
- Capabilities for handling LOBs
  - Work with character LOBs and binary LOBs
  - LOB locator support
"Native" Java Object Support

- "Native" Java Objects in the database as "BLOBs"
  - Based on Java Serialization
- Support built into the JVM
- Too restrictive
  - Persistence for Java objects
  - DB sees Java object as a black-box
  - Based on Java type system, not SQL
  - Object state cannot be introspected at the DB server side
    - Private attributes
    - Not usable for row objects/typed tables
    - Client applications written in other programming languages not supported
    - Performance implications
      - DB attribute accessors and functions/methods need to de-serialize the Java object state for execution

Mapping Java Objects to Structured Types

- Support built into the DBMS
- Very flexible
  - DB understands internal structure of type
  - Based on SQL type system
  - Client applications written in other programming languages are supported
  - Can be used to define row types/typed tables
  - DB functions/methods can be implemented in other programming language
- Potential for better performance
- Requires conversion (Java <-> SQL)
Materializing SQL99 types as Java objects
- SQL99 types manipulated using existing result set or prepared statement interfaces
- `get/setObject(<column>)` simply "works" for structured types
- Example:
  ```java
  ResultSet rs = stmt.executeQuery("SELECT e.addr FROM Employee e");
  rs.next();
  Residence addr = (Residence)rs.getObject(1);
  ```

Java
```
public class Residence {
    public int door;
    public String street;
    public String city;
}
```

SQL
```
CREATE TYPE residence (
    door INTEGER,
    street VARCHAR(100),
    city VARCHAR(50))
```
Mapping (Example)

- **Java class**
  public class Residence implements SQLData {
    public int door;
    public String street;
    public String city;
    public void readSQL(SQLInput stream, ...) throws SQLException {
      door = stream.readInt();
      street = stream.readString();
      city = stream.readString();
    }
    public void writeSQL(SQLOutput stream, ...) throws SQLException {
      stream.writeInt(door);
      stream.writeString(street);
      stream.writeString(city);
    }
  }

- **SQL99 type**
  CREATE TYPE residence (door INTEGER, street VARCHAR(100), city VARCHAR(50))

- **JDBC driver (SQLJ) automatically generates client object**
  invokes method to 'internalize' state

- **Mapping table**
  records correspondence DB2 type/Java class

- **Server-side SQL99 transformation**
  defines how UDT is passed to/from the client

Structured Types: Default Mapping

- Uses new JDBC interface 'Struct'
  - Struct st = (Struct)resultset.getObject(1)
  - public interface Struct extends SQLData {
      SQLType getSQLType();
      Object[] getAttributes();
    }
  - ResultSet.getObject() will now return an object implementing the Struct interface

- **JDBC driver includes a new Java class implementing the Struct interface**

- Generic way of handling a structured object as an array of Java objects that represent the individual attribute values
  - Useful for generic applications/tools
Object References

- New methods on ResultSet, PreparedStatement
  - `Ref ref = rs.getRef(1);`
- Ref interface
  - Has method for determining the (static) type of the referenced object
    - Hides the underlying data type of the reference
- A Ref object can be used as a parameter in other SQL statements
  - Dereference
  - Path expressions
  - Updates
  - ...

Manipulating Large Objects

- Existing approach: treat them as LONG VARCHAR, LONG VARBINARY types
- Adding support that allows for LOB locators
  - Introducing type code for BLOB, CLOB
  - Additional methods on ResultSet, PreparedStatement
    - `Blob blob = rs.getBlob(1);`
    - `Clob clob = rs.getClob(2);`
  - Additional interfaces Blob, Clob for manipulating LOB data
    - Operations for piecemeal access to LOB data, finding substrings, etc.
    - Targeted to locator-based implementation as a default
- JDBC permits mechanisms to tell the driver that LOB should be retrieved as locator or as LOB value
  - Vendor-specific extension of JDBC
Arrays

- Retrieving/storing arrays
  - `get/setArray()` methods on ResultSet, PreparedStatement
  - Array interface supports methods to:
    - Determine the element type
    - Retrieve an array as a Java array, list of Java objects
    - Open a result set on an array (i.e., turn array into a table)
  - Implementation based on array locators

SQLJ Part 2:
Java Classes as SQL Types

- Use of Java classes to define SQL types
  - Can be mapped to structured types or "native" Java types (blobs)
  - Can be used to define columns in tables
  - Can be used to define SQL99 tables (structured types)
- Mapping of object state and behavior
  - Java methods become SQL99 methods on SQL type
  - Java methods can be invoked in SQL statements
- Automatic mapping to Java object on fetch and method invocation
  - Java Serialization
  - JDBC 2.0 SQLData interface
- Includes handling of USAGE privilege on SQL type
- Use the procedures introduced in SQLJ Part 1 to install, remove, and replace SQLJ JAR files
Mapping Java Classes to SQL

- Described using extended CREATE TYPE syntax
  - DDL statement, or
  - Mapping description in the deployment descriptor
- Supported Mapping

<table>
<thead>
<tr>
<th>Java</th>
<th>SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>class</td>
<td>user-defined (structured) type</td>
</tr>
<tr>
<td>member variable</td>
<td>attribute</td>
</tr>
<tr>
<td>method</td>
<td>method</td>
</tr>
<tr>
<td>constructor</td>
<td>initializer method</td>
</tr>
<tr>
<td>static method</td>
<td>static method</td>
</tr>
<tr>
<td>static variable</td>
<td>static observer method</td>
</tr>
</tbody>
</table>

- SQL initializer methods
  - Have the same name as the type for which they are defined
  - Are invoked using the NEW operator (just like in Java)
- SQL does not know static member variables
  - Mapped to a static SQL method that returns the value of the static variable
  - No support for modifying the static variable

Mapping Example

- Java class

```java
public class Residence implements SQLData {
    public int door;
    public String street;
    public String city;
    public static String country = "USA";
    public String printAddress() { ... };
    public void changeResidence(String adr) { ... // parse and update fields ... };
}
```

- SQL DDL(descriptor statement)

```sql
CREATE TYPE Address EXTERNAL NAME 'Residence' language java (
    number INTEGER EXTERNAL NAME 'door',
    street VARCHAR(100),
    city VARCHAR(50),
    STATIC METHOD country( ) RETURNS CHAR(3) EXTERNAL VARIABLE NAME 'country',
    METHOD print() RETURNS VARCHAR(200) EXTERNAL NAME 'printAddress',
    METHOD changeAddress (varchar(200)) RETURNS Address SELF AS RESULT EXTERNAL NAME 'changeResidence'
)
```
Instance Update Methods

- Java and SQL have different object update models
  - Java model is object-based
    - Object method updates object member variables, usually returns void
  - SQL model is value-based
    - Object method returns a modified copy of the object
    - UPDATE statement is required to make object modification permanent
- SQLJ permits mapping without requiring modification of Java methods
  - SELF AS RESULT in deployment descriptor identifies an instance update method
  - Java class
    public class Residence implements SQLData {
      ...
      public void changeResidence(String adr) { ... // parse and update fields ...}
    }
  - SQL type
    CREATE TYPE Address EXTERNAL NAME 'Residence' LANGUAGE JAVA (
      ...
      METHOD changeAddress(varchar(200)) RETURNS Address SELF AS RESULT
      EXTERNAL NAME 'changeResidence'
    )
  - At runtime, the SQL system
    - Invokes the original Java method (returning void) on (a copy of) the object
    - Is responsible for returning the modified object

Usage Examples

- Use type as column type
  CREATE TABLE employees (name VARCHAR(40), addr Address)

- Insert object
  INSERT INTO employees VALUES('John Doe', NEW Address( ))

- Update object
  UPDATE employees
  SET addr =
    addr.changeAddress('1234 Parkway Dr., San Leandro')
  WHERE name = 'John Doe'

- Select object information
  SELECT addr.print()
  FROM employees
  WHERE addr.city = 'San Leandro'
SQL/MED (Management of External Data)

- Still undergoing revisions
- Purpose is to tie SQL with the management of data outside of the database (files)
  - Adds new data type: datalink
    - Link type
    - scheme (http or file)
    - file server
    - file path
    - comment
  - Abstract LOB type: used to define routines that are allowed on a LOB
  - Abstract tables: Allows for definition of access routines (user-defined routines) such as iterate, update, delete, etc.

Datalinks

- Helps maintain integrity of links from "database" attributes to data in files.
- The standardized part is datalink data type itself, not the file manager piece.

Applications

File Manager

Employee Table

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Ted</td>
<td>1956</td>
<td></td>
</tr>
<tr>
<td>Sharon</td>
<td>1961</td>
<td></td>
</tr>
<tr>
<td>Abby</td>
<td>1950</td>
<td></td>
</tr>
<tr>
<td>John</td>
<td>1948</td>
<td></td>
</tr>
</tbody>
</table>

SQL API requests

File API requests
**SQL/MED**

- Abstract tables
  - Lets users write SQL queries on data that is stored in another file system
  - Routines manipulate the data
  ```
  CREATE ABSTRACT TABLE XRAY
  STATE <routine-name>
  ITERATE <routine-name>
  COMMIT <commit-routine-name>
  ```

- Abstract LOBs
  - Like Abstract tables, but for LOBs
  - Routines for locators, concatenation, overlay, substring, etc

**What of the Future? (SQL4)**

- OLAP stuff
  - Additional functions, like RANK moving sum, average, ratio, additional aggregation functions
  - Summary tables

- More collection data types
  - set (unordered, no duplicates)
  - list (ordered, may contain duplicates)
  - multiset (unordered, may contain duplicates)

- Type migration
  - How do you make an employee a manager in a table hierarchy?
  - ...

In some cases, function is already delivered. Not the ideal model.