ISO/IEC JTC 1/SC 32 N 0283

Date: 1999-05-21

REPLACES: --

ISO/IEC JTC 1/SC 32

Data Management and Interchange

Secretariat: United States of America (ANSI)

Administered by Pacific Northwest National Laboratory on behalf of ANSI

DOCUMENT TYPE	Other document (Open)
TITLE	Working Group 3 Tutorial Presentation
SOURCE	Krishna Kulkarni SC 32/WG 3
PROJECT NUMBER	
STATUS	Presentation for WG 3 at the Group Tutorial ISO/IEC JTC 1/SC 32 on 1999-05-21
REFERENCES	
ACTION ID.	FYI
REQUESTED ACTION	
DUE DATE	
Number of Pages	81
LANGUAGE USED	English
DISTRIBUTION	P & L Members SC Chair WG Conveners and Secretaries

Douglas Mann, Secretariat, ISO/IEC JTC 1/SC 32

Pacific Northwest National Laboratory *, 901 D Street, SW., Suite 900, Washington, DC, 20024-2115, United States of America

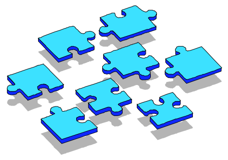
Telephone: +1 703 575 2114; Facsimile; +1 703 681 9180; E-mail: MannD@battelle.org

*Pacific Northwest National Laboratory (PNL) administers the ISO/IEC JTC 1/SC 32 Secretariat on behalf of ANSI

Working Group 3 Tutorial

Krishna Kulkarni, USA

Presented to SC32 Working Group Tutorial Meeting on 1999-05-21



Working Group 3

- Title: Database Languages
- Area of Work (Ref: SC32 N249)
  - Develop and maintain languages for the dynamic specification, maintenance and description of database structures and contents in multi-user amd multi-server environments. May include
    - Data types, behaviors and integrity constraints on the contents of the defined structures.
    - Mechanisms for the creation and generation of new data types and behaviors
  - Develop and maintain languages that provide for the storage, access and manipulation of data in database structures by multiple concurrent users.
  - Provide interfaces for the languages developed to other standard programming languages
  - Provide interfaces or access to other standards describing data types, behaviors or database content to users of the languages developed.
- Convenor: Stephen Cannan
- Editor: Jim Melton

WG3 Published Standards

- 3.3 Database Language SQL
  - First Edition published in 1987 as ISO/IEC 9075:1986
  - Second Edition published in 1989 as ISO/IEC 9075:1989
  - Third Edition publsihed in 1992 as ISO/IEC 9075:1992
- 3.3.1 SQL Call-Level Interface (SQL/CLI)
  - Published in 1995 as ISO/IEC 9075-3:1995
- 3.3.2 SQL Persistent Stored Modules (SQL/PSM)
  - Published in 1996 as ISO/IEC 9075-4:1996
- 3.4.1.1 SQL Corrigendum 3, 3.4.1.2 CLI Corrigendum 1, 3.4.1.3 PSM Corrigendum 1
  - Published in 1999 as ISO/IEC 9075:1992 C3

WG3 Project List

- Currently under FDIS ballot
  - 3.4.1 SQL/Framework
  - 3.4.2 SQL/Foundation
  - 3.4.3 SQL/CLI
  - 3.4.4 SQL/PSM
  - 3.4.5 SQL/Bindings
- FCD ballot just closed
  - 3.4.10 SQL/OLB (Object Language Binding)
- CD ballot just closed
  - 3.4.9 SQL/MED (Management of External Data)
- Working Drafts
  - 3.4.6 SQL/Transaction
  - 3.4.7 SQL/Temporal

SQL/Framework Overview

- Provides an overview of the complete standard
  - Includes conceptual material common to all parts
- Describes the conformance requirements
  - Conformance model based on Core SQL and packages.
  - The content of SQL divided into a number of "features", each identifed and precisely specified. 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.

SQL/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

SQL/Foundation Overview ...

- Triggers
  - Provide automatic execution of SQL logic when a specific event occurs
    - Transforms a passive DBMS into an active DBMS
  - Guaranteed enforcement of business rules
    - 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

SQL/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

SQL/Foundation Overview ...

- 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/Foundation Overview ...

- User-defined types
  - Named types representing entitites
    - employee, project, money, shape, image, text ...
  - Two kinds of user-defined types
    - Distinct types
      - based on a predefined type
      - no inheritance
    - Structured types
      - one or more attributes
      - type hierarchy supported
  - Type-specific behavior specified as methods/functions
    - hire, budget, convert, area, contains, ...
  - Strong typing
  - User-defined ordering
  - User-defined casts
  - User-defined transforms

SQL/Foundation Overview ...

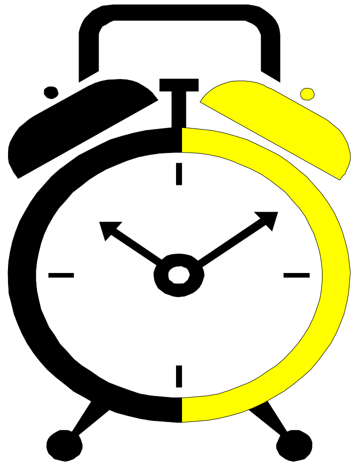
- Collection types
  - Arrays
- Row types
  - Like record structures in programming languages
  - Type of rows in tables
  - Nesting (rows with row-valued fields)
- Reference types
  - Support "object identity"
  - Navigational access (path expressions)
- Typed tables and views
  - Table hiearchies
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

SQL/Foundation Overview ...

- Subtables (table hierarchies)
  - Increase modeling power and expressive power of queries
  - Means to model collection hierarchies
    - 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

BEFORE Triggers

```
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_#;
```

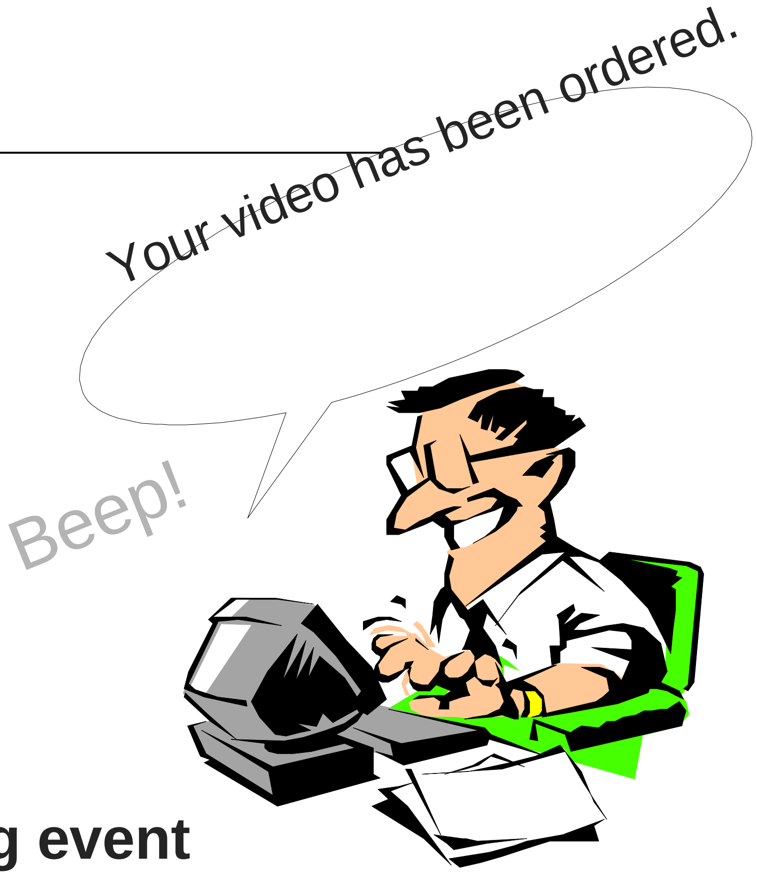


BEFORE

- **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**

AFTER Triggers

```
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 ...;
```



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**

Trigger Granularity

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

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

- **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**

Recursive SQL

- 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

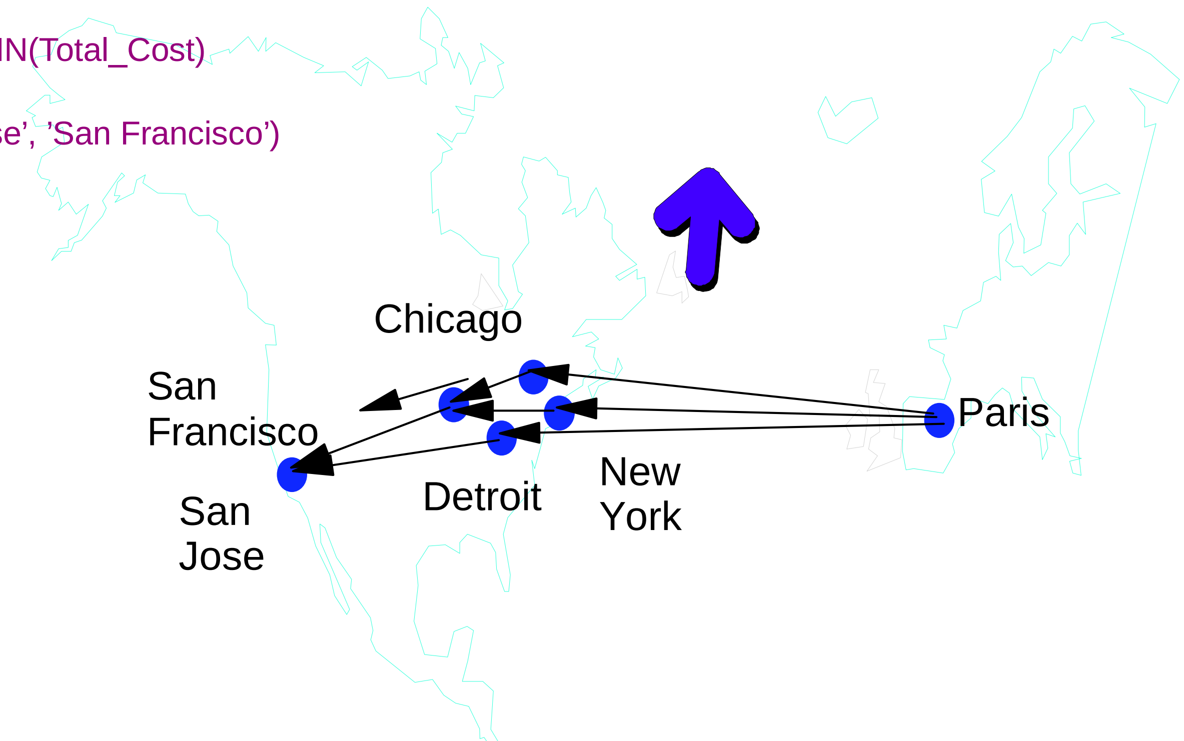
Recursive SQL ...

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
```

Reachable_From

SOURCE	DESTIN	MIN (Total_ Cost)
Paris	San Fran	14
Paris	San Jose	10



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 ...

- 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’
```

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.

Distinct Types

```
CREATE TYPE plan.roomtype
AS CHAR(10) FINAL;

CREATE  TYPE plan.meters
AS INTEGER FINAL;

CREATE  TYPE plan.squaremeters
AS INTEGER FINAL;

CREATE TABLE RoomTable (
RoomID              plan.roomtype,
RoomLength          plan.meters,
RoomWidth           plan.meters,
RoomPerimeter       plan.meters,
RoomArea            plan.squaremeters);
```

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

ERROR

```
UPDATE RoomTable
SET RoomLength =
RoomWidth;
```

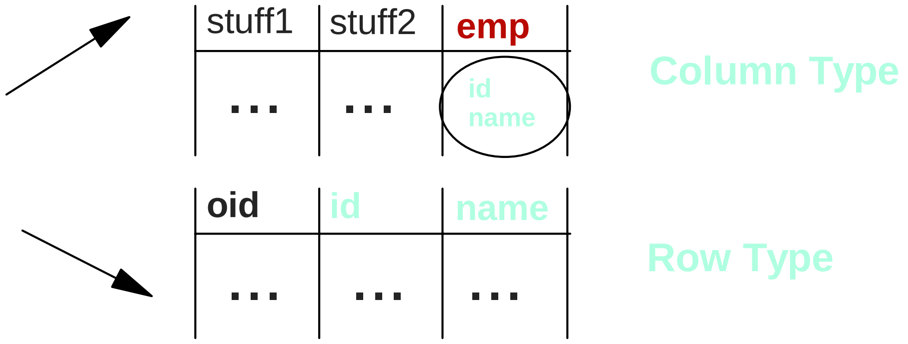
NO ERROR RESULTS

Each UDT is logically incompatible with all other type

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))
```



Structured Types: Example

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

CREATE TYPE bitmap AS BLOB 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
```

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 ...

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

```
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 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 ...

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

```
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.

Manipulating Attributes

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

```
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

```
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 "()"

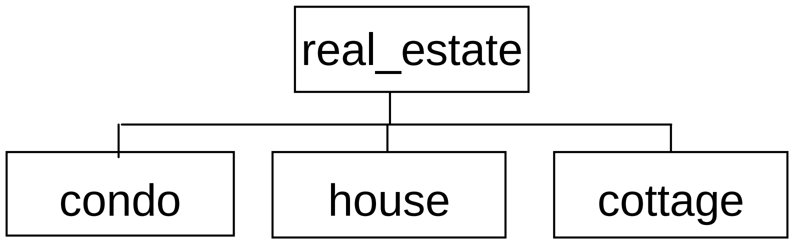
```
DECLARE  r     real_estate;
...
SET r.size = 2540.50;          -- same as r.size (2540.50)
...
SET ... = r.location.state;    -- same as r.location().state()
SET r.location.city = ‘LA’;    -- same as r.location(r.location.city(‘LA’))
```

- 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)

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
```



Subtyping and Inheritance

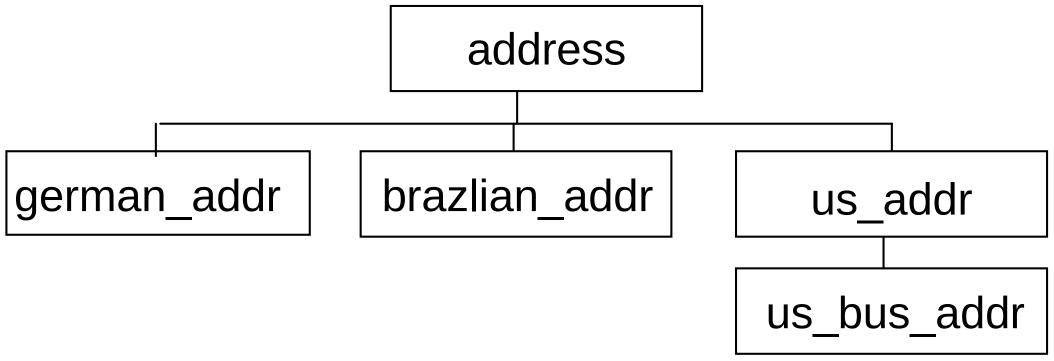
```
CREATE TYPE address AS
(street     CHAR (30), city     CHAR(20), state CHAR (2), zip INTEGER) NOT FINAL

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

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’))

price	owner	location
<us_dollar> amount 100,000	’Mr. S. White’	*<us_addr>* ’1654 Heath ...’
<real> amount 400,000	’Mr. W. Green’	<brazilian_addr> ’245 Cons. Xavier ...’
<german_mark> amount 150,000	’Mrs. D. Black’	<german_addr> ’305 Kurt-Schumacher ...’

type tag

Value Substitutability

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

```
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

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

```
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)
```

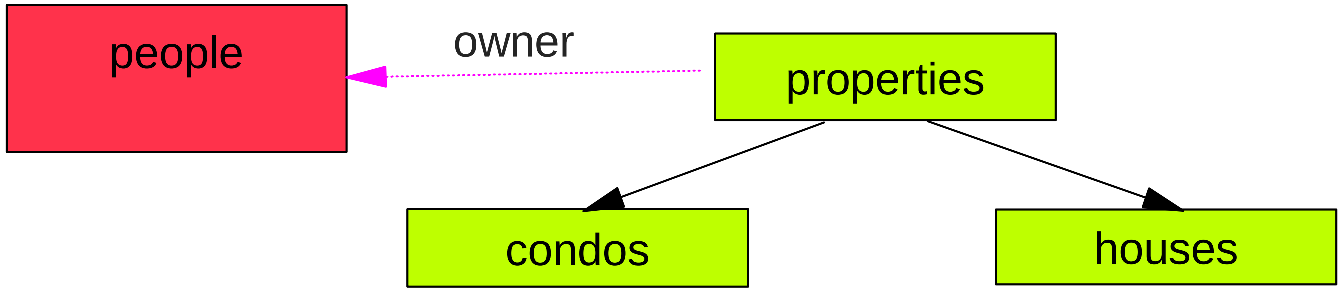
- Referential constraints specify inclusion dependencies
  - It is unclear which table to access during dereferencing
- There is no notion of strong typing

Subtables: Table Hierarchies

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

```
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

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

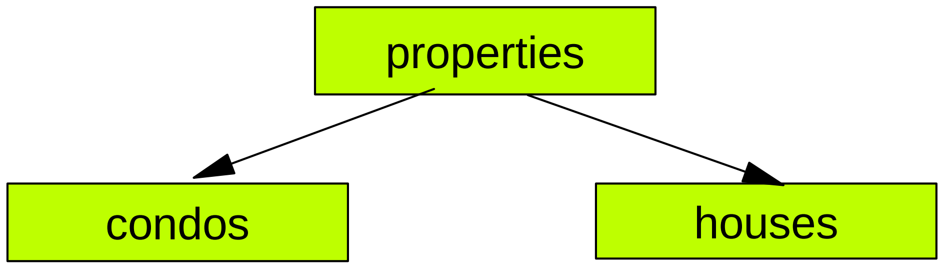
```
SELECT price, location.city, location.state  FROM properties
WHERE contains (text_description, ’excellent school district’)
```

  - 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

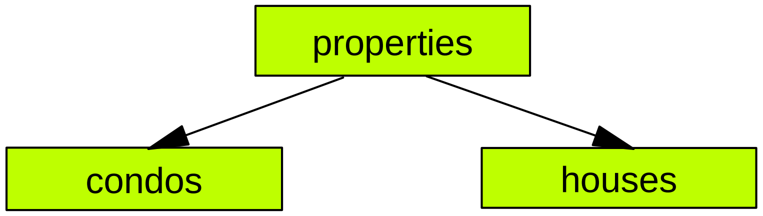
```
SELECT price, location.city, location.state
FROM properties
WHERE contains (text_description, ’excellent school district’)
AND DEREF (oid) IS OF (house)
```

- **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

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

```
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"
```

```
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

```
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

```
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

Arrays

- The only collection type of SQL99
- Tables with array-valued columns

```
CREATE TABLE reports
(id          INTEGER,
 authors     VARCHAR(15) ARRAY[20],
 title       VARCHAR(100),
 abstract    FullText)
```

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

Arrays (cont.)

- Array operations
  - Element access by ordinal number
  - Cardinality
  - Comparison
  - Constructors
  - Assignment
  - Concatenation
  - CAST
  - Declarative selection facilities over arrays

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)
```

Overview of SQL 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
  - 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

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.

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 concatentation
  - 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 opredicate
  - IN predicate with list of values
  - LIKE predicate
  - LIKE predicate with ESCAPE clause
  - NULL predicate
  - Quantified comparsion predicate
  - EXISTS predicate
  - Subqueries in comparision predicate
  - Subqueries in IN predicate
  - Subqueries in quantified comparison predicate
  - Correlated subqueries
  - Search condition
- Basic query expressions
  - UNION ALL
  - EXCEPT DISTINCT
  - Columns combined via UNION and EXCEPT do not have to be exact same data types
  - Table subquery can specify UNION and

Core SQL (cont.)

- 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 list
  - 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
- Basic SET TRANSACTION statement
  - with ISOLATION LEVEL SERIALIZABLE clause
  - with READ ONLY and READ WRITE clauses
  - with DIAGNOSTIC SIZE clause
- Updateable queries with subqueries
- SQL comments using leading double minus
- SQLSTATE support
- 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
  - Comparisoin 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 defintion statement
  - CREATE SCHEMA
  - CREATE TABLE for persistent base tables
  - CREATE VIEW
  - CREATE VIEW: WITH CHECK OPTION
  - GRANT statemtn
- Scalar subquery values
- Expanded NULL predicate (the <row value expression> can be something other than a <column reference>

Core SQL (cont.)

- 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

SQL/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:

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

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

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

SQL/PSM (cont.)

- If we create a SQL procedure first:

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

- Then the embedded program can be written as

```
void main
{
EXEC SQL CALL proc1();
}
```

SQL/PSM Procedural Language Extensions

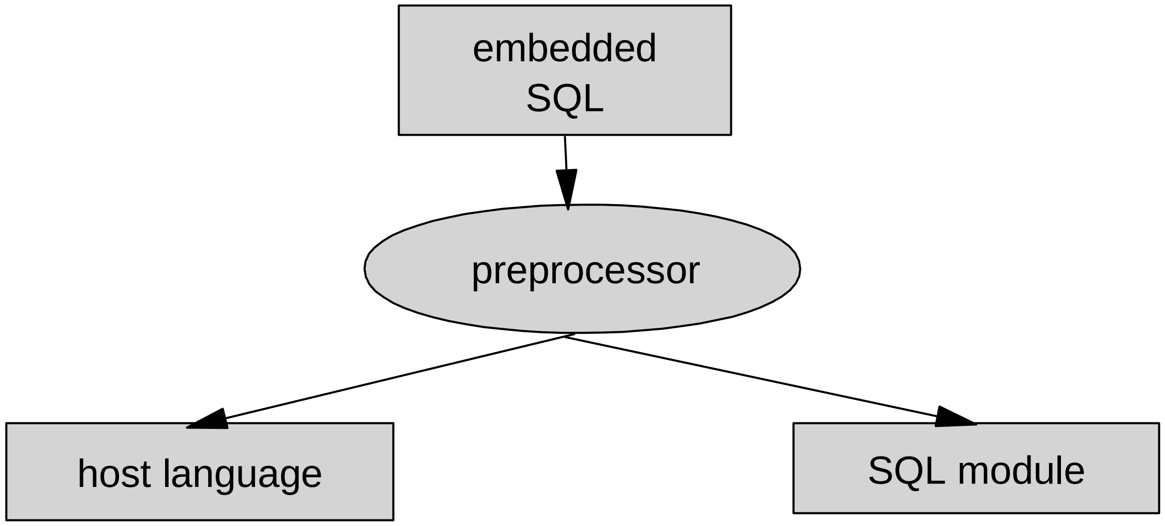
Compound statement	BEGIN ... END;
SQL variable declaration	DECLARE var CHAR (6);
If statement	IF subject (var <> ’urgent’) THEN ... ELSE ...;
Case statement	CASE subject (var) WHEN ’SQL’ THEN ... WHEN ...;
Loop statement	LOOP < SQL statement list> END LOOP;
While statement	WHILE i<100 DO .... END WHILE;
Repeat statement	REPEAT ... UNTIL i<100 END REPEAT;
For statement	FOR result AS ... DO ... END FOR;
Leave statement	LEAVE ...;
Return statement	RETURN ’urgent’;
Call statement	CALL procedure_x (1,3,5);
Assignment statement	SET x = ’abc’;
Signal/resignal statement	SIGNAL division_by_zero

SQL/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, ... :
```

Direct SQL

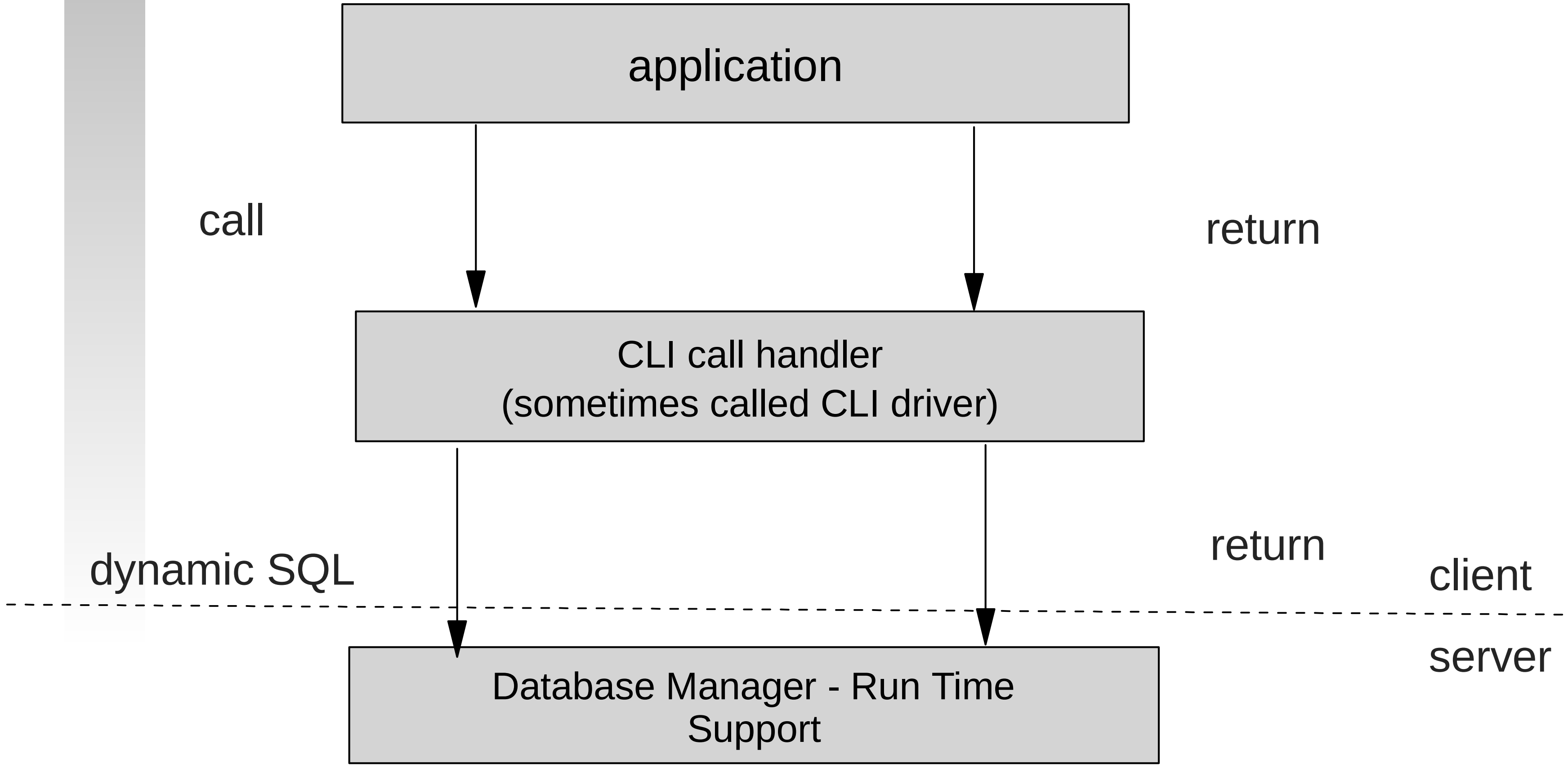
- Implementation-defined mechanism for executing direct SQL statements
  - In effect, prepared immediately before execution
  - Cannot issue dynamic SQL using direct SQL
- Invocation, method of raising error conditions, method of accessing diagnostics information, and the method of returning results are all implementation-defined

SQL/CLI 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
  - ODBC (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



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)

What is new in SQL/CLI?

- 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

What is new in SQL/CLI?

- 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 SQL and ODBC
- Parallel result set processing after CALL statement
- Support for savepoints
- Misc. SQL alignment (roles, user-defined casts, SQLSTATEs, etc.)

SQL/OLB Overview

- Embedded SQL in Java
  - A version based on SQL-92 and JDBC accepted as ANSI standard "Database Language - SQL, Part 10 Object Language Bindings (SQL/OLB)", ANSI X3.135.10:1998
- SQL/OLB programs are smaller than JDBC applications
- Binary portability

SQL/OLB Syntax

- SQL/OLB clauses are statements or declarations
  - Clause begins with "**#sql**" token
- An SQL statement appears as an SQL/OLB statement clause
  - May contain host-variable references (e.g., :x) or host expressions (e.g., :(x + y))
  - May specify explicit connection or use default connection

#sql [**[**<context>**]**] **{** <statement spec clause> **}**

SQL/OLB vs. JDBC: Retrieve Single Row

- SQL/OLB

```
#sql [con] { SELECT ADDRESS INTO :addr FROM EMP
          WHERE NAME=:name };
```

- JDBC

```
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();
```

Result Set Iterators

- Mechanism for accessing the rows returned by a query
  - Comparable to an SQL cursor
- SQL/OLB Iterator declaration clause results in generated iterator class
  - Iterator is a Java object
  - Iterators are strongly typed
  - Generic methods for advancing to next row
- SQL/OLB assignment clause assigns query result to iterator
- Two types of iterators
  - Named iterator
  - Psitioned iterator

Named Iterator

- Generated iterator class has accessor methods for each result column

```
#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

```
#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 );
 }
```

Binary Portability

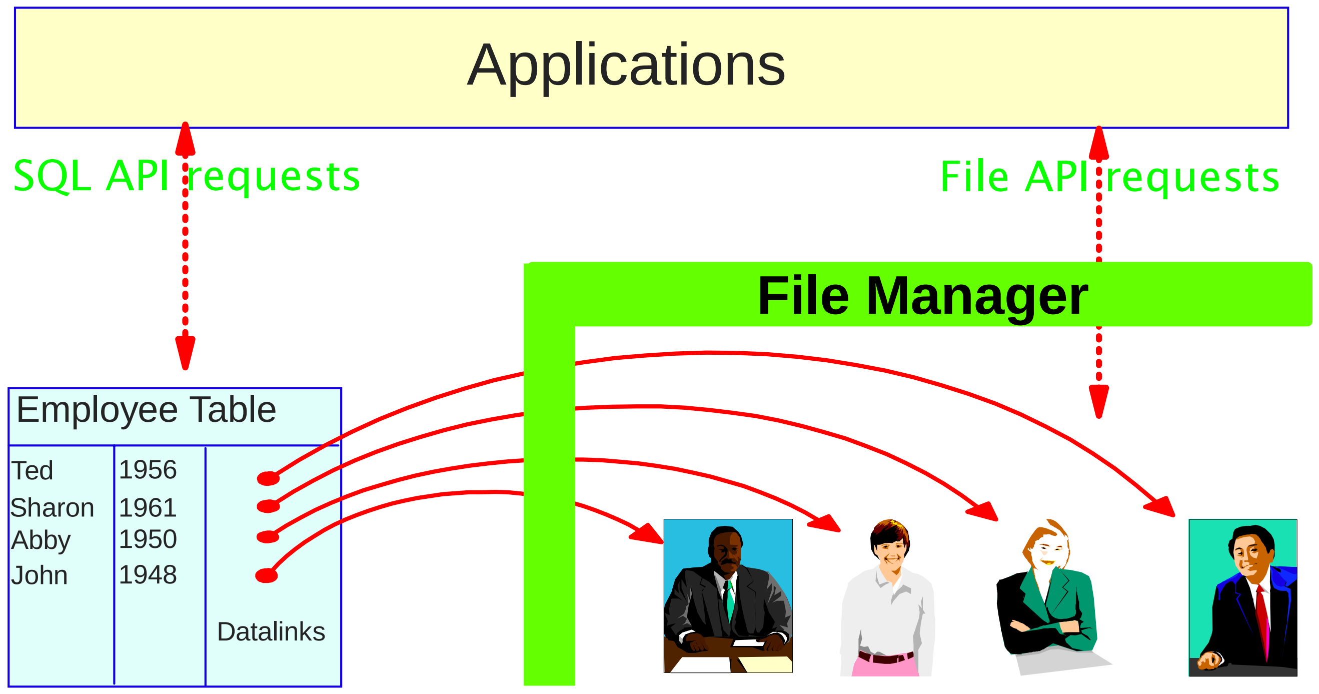
- Static SQL portability problems
  - 3GL language not 100% portable
  - Each DBMS has unique precompiler output
    - No binary portability across DBMSs
- SQL/OLB advantages
  - Java is platform-independent
    - Compiled SQLJ applications are pure Java
  - Generic SQL/OLB translator (works for all DBMSs)
  - SQL/OLB application binaries (Java bytecodes) are portable across DBMSs
  - Vendor-specific customizations can be performed after compilation
    - Performance optimizations, ...

SQL/MED Overview

- Extensions to manipulate heterogeneous, external data sources using SQL statements
  - 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.



Abstract Tables, Abstract LOBs

- 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

SQL/Transaction Overview

- Interface specification aimed at enabling an SQL-server to participate in global transactions
- Predates the ISO XA standard

SQL/Temporal Overview

- Language extensions for manipulating tables containing temporal data
- Currently contains:
  - PERIOD data type constructor
  - Expressions and predicates involving PERIOD values

Future plans

- Package Definitions
- New Parts
- New Features
  - OLAP extensions
    - RANK, moving sum, moving average, etc.
    - 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?
  - ...
- Profiles