Introduction of SQL Programming

SS Chung
Embedded SQL

- SQL is not enough! Needs to be embedded in a general purpose language to get
  - GUI
  - Flow of control
  - Generate SQL dynamically based on user input

- The SQL standard defines embeddings of SQL in a variety of programming languages

- SQL commands can be called from within a host language (e.g., C/C++, Basic, .NET or Java) program or scripting language (e.g., PHP, Ruby)

- A query answer is a bag of records - with arbitrarily many rows! No such data structure in most languages. Called *impedance mismatch*

- The SQL standard supports a *cursor* to handle this
Embedded SQL

- A language to which SQL queries are embedded is referred to as a host language, and the SQL structures permitted in the host language comprise *embedded* SQL.

- The basic form of these languages follows that of the System R embedding of SQL into PL/I.

- EXEC SQL statement is used to identify embedded SQL request to the preprocessor

`EXEC SQL <embedded SQL statement >
END_EXEC`

Note: this varies by language (for example, the Java embedding uses `# SQL { .... };` )
You can declare a cursor on a table or query statement (which generates a table as a result).

You can open a cursor, and repeatedly fetch a tuple then move the cursor, until all tuples have been retrieved.

You can modify/delete a tuple pointed to by a cursor.

SQL must be able to report data-generated errors.
Cursors

- Cursor points to the current row.

- DECLARE `cName` CURSOR
  FOR select statement
  – declares the select statement

- OPEN `cName`
  – Executes the select statement
Embedding SQL in C: An Example

```c
Void ListAges( int minzip)
{
    char SQLSTATE[6];

    EXEC SQL BEGIN DECLARE SECTION
    char c_name[20];
    char c_party[3];
    integer minzip;
    EXEC SQL END DECLARE SECTION

    • SQLSTATE holds SQL error codes
    • EXEC SQL denotes embedded SQL section
    • DECLARE SECTION binds variables into SQL
```
Cursor that gets names of candidates who have a principal committee, in alphabetical order

```
EXEC SQL DECLARE cinfo CURSOR FOR
    SELECT N.Candname
    FROM Cand N JOIN Comm M ON (N.Princomm = M.Commid)
    ORDER BY N.Candname;

OPEN cinfo;

FETCH cinfo INTO :c-name;
    (probably in a loop in your program)

CLOSE cinfo;
```

Notice the colon in :c-name – it refers to a variable that has been declared in the surrounding program
Embedding in C: An Example

EXEC SQL DECLARE cinfo CURSOR FOR
SELECT N. candname, N.par
FROM cand N
WHERE zip > :minzip
ORDER BY N.candname;

- DECLARE cinfo CURSOR: defines a name for Cursor
- SELECT ...: SQL whose results the cursor will point to
- :minzip: Note the colon referring to a C variable declared previously
**Embedding in C: An Example**

```c
EXEC SQL OPEN cinfo;
EXEC SQL FETCH cinfo INTO :c_name;
While(SQLSTATE != "02000")
{
    printf("Candidate name is %s\n", candname);
    EXEC SQL FETCH cinfo INTO :c_name;
}
EXEC SQL CLOSE cinfo;
```

- **OPEN cinfo**: Executes the query and positions the cursor before the first row of the output.
- **FETCH cinfo INTO**: assigns the data of the first row (if it exists) into C program variables.
- **CLOSE cinfo**: Free the cursor’s resources.
Example Query

From within a host language, find the names and cities of customers with more than the variable amount dollars in some account.

Specify the query in SQL and declare a cursor for it

EXEC SQL

Declare c Cursor for

Select Depositor.Customer_name, Customer_city
From Depositor, Customer, Account
Where Depositor.Customer_name = Customer.Customer_name
    and Depositor.Account_number = Account.Account_number
    and Account.Balance > :amount

END_EXEC
Embedded SQL (Cont.)

- The **open** statement causes the query to be evaluated
  
  ```sql
  EXEC SQL open c END_EXEC
  ```

- The **fetch** statement causes the values of one tuple in the query result to be placed on host language variables.
  
  ```sql
  EXEC SQL fetch c into :cn, :cc END_EXEC
  ```

Repeated calls to **fetch** get successive tuples in the query result.

- A variable called **SQLSTATE** in the SQL communication area (SQLCA) gets set to ‘02000’ to indicate no more data is available.

- The **close** statement causes the database system to delete the temporary relation that holds the result of the query.
  
  ```sql
  EXEC SQL close c END_EXEC
  ```

Note: above details vary with language. For example, the Java embedding defines Java iterators to step through result tuples.
Updates Through Cursors

- Can update tuples fetched by cursor by declaring that the cursor is for update

  ```sql
  declare c cursor for
  select *
  from account
  where branch_name = 'Perryridge'
  for update
  ```

- To update tuple at the current location of cursor `c`

  ```sql
  update account
  set balance = balance + 100
  where current of c
  ```
Dynamic SQL

- Allows programs to construct and submit SQL queries at run time.
- Example of the use of dynamic SQL from within a C program.

```c
char * sqlprog = "update account
    set balance = balance * 1.05
    where account_number = ?"
EXEC SQL prepare dynprog from :sqlprog;
EXEC SQL execute dynprog using :account;
```

- The dynamic SQL program contains a ?, which is a place holder for a value that is provided when the SQL program is executed.
Procedural Extensions:

**Stored Procedures**

- Stored Procedure is a function in a shared library accessible to the database server.
- Can also write Stored Procedures using languages such as C or Java.
- Advantages of Stored Procedure:
  - Flexibility to SQL,
  - Reduced network traffic for speed.
- The more SQL statements that are grouped together for execution, the larger the savings in network traffic.
Procedural Extensions: Stored Procedures

In SQL, the analog of a subroutine is called a **Stored Procedure (SP)**.

Similarities between SPs and subroutines
- SPs can be written in many languages
- An **SP returns a structure**
  - In SQL-land this means a **Table**
  - A call to an **SP** can appear in the **FROM** clause of a query

Difference between SPs and subroutines
- An **SP executes in the DBMS's address space**, not the clients'
- This has **huge** performance implications.
Applications using stored procedures
Bad news, Good news

- Stored Procedures are not in any SQL standard
  - Each Server vendors' implementation is different
    - But each vendor gives porting advice, e.g.,

- Stored Proc can be very efficient compared to SQL
  - Suppose a program's logic is $S_1;S_2;...;S_n$, each $S_i$ is a SQL statement

---

Client | Server
---|---
| $S_1$|  
| $S_2$|  
| ...|  
| $S_n$|  

Implemented as SQL

---

Client | Server
---|---
|  | $S_1$
|  | $S_2$
|  | ...  
|  | $S_n$

Implemented as SP
Efficiency of SPs

If you write the program in SQL, the cost is
- 2n context switches +
- n network traversals to send queries to the server +
- n network traversals to send answers to the client

If you write the program as a Stored Procedure, the cost is
- 2 context switches
- 1 network traversals to send queries to the server
- 1 network traversals to send answers to the client

Depending on the size of n, the size of each query and the size of the intermediate and final answers, this can be a huge difference.
Procedural Extensions: Functions / Stored Procedures

- Stored Procedures
  - Named blocks of PL/SQL complied program stored within the database
  - Execute them using the Call statement
  - Permit external applications to operate on the database without knowing about internal details

- Functions are particularly useful with specialized data types such as images and geometric objects
  - Example: functions to check if polygons overlap, or to compare images for similarity

- Table-valued functions, which can return a relation as a result
Procedural Extensions:
Functions / Stored Procedures

SQL provides a rich set of imperative constructs including:
- For and While Loops
- Assignment
- if-then-else statements.

SP returns a structure **Table**
- A call to an SP can appear in the FROM clause of a query

**Table-valued functions**, which can return a relation as a result
Procedural Extensions: Functions / Stored Procedures

- Stored Procedures can:
  - Have parameters
  - Invoke other procedures and functions
  - Return values
  - Raise exceptions
  - Syntax checked and compiled into p-code
  - The p-code is stored in the database

- Creating stored procedures
  - Write a stored procedure in a text file and process the commands using Client Interface tool (for example, the Query Analyzer or SQL Server Enterprise Manager in MS SQL Server)
3 types of Parameters

Procedures can have 3 types of parameters

- **IN**
  - used for input

- **OUT**
  - used for output
  - side effects ⇒ hard to read / debug the code

- **INOUT**
  - used for input + output

Functions

- standard: only IN parameters
- Oracle: all kinds of parameters
  - Don’t use OUT and INOUT with functions!
Compiling and recompiling stored procedures

- Stored Procedures / Functions are automatically compiled when recreated.
- If one of the tables used in a procedures is altered, the procedure / function must be recompiled
  - `Alter procedure procedureName compile;`
Some PL/SQL Construct in Stored Procedures and Functions

- Call pName(parameters)
  - call another procedure
- Return value
  - return from a function
- Variable := value
  - assignment

- begin ... end
  - statement group
- if condition then
  - statements
- else
  - statements
- end if
- For loop
- While loop
- General loop
  - Inner exit statement
SQL statements

- Stored Procedures / Functions can contain SQL statements
  - select, insert, update, delete

Select syntax [result: one value]
  - select attr into variable from ...
SQL Functions

- Define a function that, given the name of a customer, returns the count of the number of accounts owned by the customer.

  Create function `account_count` (customer_name varchar(20))
  returns integer
  begin
    declare a_count integer;
    select count (*) into a_count
    from depositor
    where depositor.customer_name = customer_name
    return a_count;
  end

- Find the name and address of each customer that has more than one account.

  select customer_name, customer_street, customer_city
  from customer
  where account_count (customer_name ) > 1
Create Or Replace Procedure <name> (<arglist>) AS
<declarations>
BEGIN
    <procedure statements>
END

- Tasks performed by the client application
- Tasks performed by the stored procedure, when invoked
- The CALL statement
- Explicit parameter to be defined:
  - **IN**: Passes a value to the stored procedure from the client application
  - **OUT**: Stores a value that is passed to the client application when the stored procedure terminates.
  - **INOUT**: Passes a value to the stored procedure from the client application, and returns a value to the Client application when the stored procedure terminates.
CREATE PROCEDURE UPDATE_SALARY_1
  (IN   EMPLOYEE_NUMBER   CHAR(6),
   IN   RATE   INTEGER)

LANGUAGE SQL
BEGIN
  UPDATE EMPLOYEE
  SET SALARY = SALARY * (1.0 * RATE / 100.0 )
  WHERE SSN = EMPLOYEE_NUMBER;
END

LANGUAGE value of SQL and the BEGIN...END block, which forms the procedure body, are particular to an SQL procedure

1) The stored procedure name is UPDATE_SALARY_1.
2) The two parameters have data types of CHAR(6) and INTEGER. Both are input parameters.
3) LANGUAGE SQL indicates that this is an SQL procedure, so a procedure body follows the other parameters.
4) The procedure body consists of a single SQL UPDATE statement, which updates rows in the employee table.
The `account_count` function could instead be written as procedure:

Create procedure `account_count_proc` (in `customer_name` varchar(20),
out `a_count` integer)

begin
    select count(*) into `a_count`
    from depositor
    where depositor.customer_name = account_count_proc.customer_name
end

Procedures can be invoked either from an SQL procedure or from embedded SQL, using the `call` statement.

- `declare a_count integer;`
- `call account_count_proc('Smith', a_count);`

Procedures and functions can be invoked also from dynamic SQL:

SQL:1999 allows more than one function/procedure of the same name (called name overloading), as long as the number of arguments differ, or at least the types of the arguments differ.
Procedural Constructs

- CASE statement
- FOR statement
- GOTO statement
- IF statement
- ITERATE statement
- RETURN statement
- WHILE statement
Procedural Constructs

- Compound statement: `begin ... end`,
  - May contain multiple SQL statements between `begin` and `end`.
  - Local variables can be declared within a compound statements.

- **While** and **repeat** statements:
  ```sql
  declare n integer default 0;
  while n < 10 do
    set n = n + 1
  end while

  repeat
    set n = n - 1
  until n = 0
  end repeat
  ```
**For loop**

- Permits iteration over all results of a query
- Example: find total of all balances at the Perryridge branch

```sql
declare n integer default 0;
for r as
    select balance
    from account
    where branch_name = 'Perryridge'
do
    set n = n + r.balance
end for
```
Procedural Constructs (cont.)

- **Conditional statements (if-then-else)**
  E.g. To find sum of balances for each of three categories of accounts (with balance <1000, >=1000 and <5000, >= 5000)
  
  ```
  if r.balance < 1000
    then set l = l + r.balance
  else if r.balance < 5000
    then set m = m + r.balance
  else
    set h = h + r.balance
  end if
  ```

- SQL:1999 also supports a **case statement** similar to C case statement

- Signaling of exception conditions, and declaring handlers for exceptions
  ```
  declare out_of_stock condition
  declare exit handler for out_of_stock
  begin
    ...
    .. signal out-of-stock
  end
  ```
  - The handler here is `exit` -- causes enclosing `begin..end` to be exited
  - Other actions possible on exception
CREATE PROCEDURE UPDATE_SALARY_IF
(IN employee_number CHAR(6),
   IN rating SMALLINT)
BEGIN
    SET counter = 10;
    WHILE (counter > 0) DO
        IF (rating = 1) THEN
            UPDATE employee
            SET salary = salary * 1.10, bonus = 1000
            WHERE empno = employee_number;
        END IF;
        ELSE IF (rating = 2) THEN
            UPDATE employee
            SET salary = salary * 1.05, bonus = 500
            WHERE empno = employee_number;
        END IF;
        ELSE
            UPDATE employee
            SET salary = salary * 1.03, bonus = 0
            WHERE empno = employee_number;
        END IF;
        SET counter = counter - 1;
    END WHILE;
END
Calling Stored Procedure

Syntax

ProcedureName(ProcedureName (Parameter1, Parameter2, ...)

Two ways to link formal and actual parameters

- Position
  - Like Java: 1st parameter formal parameter linked to 1st actual parameter, etc.

- Named
  - FormalParameterName => value
Invoking Procedures
Can invoke Stored procedure stored at the location of the database by using the SQL CALL statement

Nested SQL Procedures:
To call a SQL procedure from within a caller SQL procedure, simply include a CALL statement with the appropriate number and types of parameters in your caller procedure.

```
CREATE PROCEDURE NEST_SALES(OUT budget DECIMAL(11,2))
LANGUAGE SQL
BEGIN
  DECLARE total INTEGER DEFAULT 0;
  SET total = 6;
  CALL SALES_TARGET(total);
  SET budget = total * 10000;
END
```
Parameters

Variables are declared after the keyword AS

CREATE PROCEDURE Customer_Insert
(@NewName char(50)),
(@NewAreaCode char(5)),
(@NewPhone char(10)),
(@Nationality char(25))
AS
DECLARE @Count as smallint,
DECLARE @Aid as int,
DECLARE @Cid as int
/* Check to see if customer already exists */
SELECT @Count = Count(*)
FROM dbo CUSTOMER
WHERE (Name)=@NewName AND AreaCode=@NewAreaCode AND PhoneNumber=@NewPhone
IF @Count > 0
BEGIN
PRINT 'Customer Already Exists -- No Action Taken'
RETURN
END
/* Add new Customer data */
INSERT INTO dbo CUSTOMER
([Name],AreaCode,PhoneNumber)
VALUES
[@NewName, @NewAreaCode, @NewPhone]
/* Get new surrogate key value */
SELECT @Cid = CustomerID
FROM dbo CUSTOMER
WHERE (Name)=@NewName AND AreaCode=@NewAreaCode
AND PhoneNumber=@NewPhone
/* Now create intersection record for each appropriate artist */
DECLARE Artist_Cursor CURSOR
FOR
SELECT ArtID FROM dbo ARTIST
WHERE Nationality = @Nationality
/* process each Artist of specified nationality */
OPEN Artist_Cursor
FETCH NEXT FROM Artist_Cursor INTO @Aid
WHILE @@FETCH_STATUS = 0
BEGIN
    @Aid = ArtID
    @Cid = CustomerID
    INSERT INTO dbo [CUSTOMER_ARTIST_INT]
    (ArtID,CustomerID)
    VALUES (@Aid, @Cid)
    FETCH NEXT FROM Artist_Cursor INTO @Aid
END
CLOSE Artist_Cursor
DEALLOCATE Artist_Cursor
SQL:1999 permits the use of functions and procedures written in other languages such as C or C++

Declaring external language procedures and functions

```sql
CREATE PROCEDURE account_count_proc(
    in customer_name VARCHAR(20),
    out count INTEGER
) LANGUAGE C
EXTERNAL NAME '/usr/avi/bin/account_count_proc'
```

```sql
CREATE FUNCTION account_count(
    customer_name VARCHAR(20)
) RETURNS INTEGER
LANGUAGE C
EXTERNAL NAME '/usr/avi/bin/account_count'
```
Benefits of external language functions/procedures:

- more efficient for many operations, and more expressive power

Drawbacks

- Code to implement function may need to be loaded into the database system and executed in the database system’s address space
  - risk of accidental corruption of database structures
  - security risk, allowing users access to unauthorized data
- There are alternatives, which give good security at the cost of potentially worse performance
- Direct execution in the database system’s space is used when efficiency is more important than security
Security with External Language Routines

- To deal with security problems
  - Use sandbox techniques
    - that is use a safe language like Java, which cannot be used to access/damage other parts of the database code
  - Or, run external language functions/procedures in a separate process, with no access to the database process’ memory
    - Parameters and results communicated via inter-process communication
- Both have performance overheads
- Many database systems support both above approaches as well as direct executing in database system address space
Table Functions

- Functions that return a relation as a result
- Example: Return all accounts owned by a given customer

Create function `accounts_of (customer_name char(20))` returns `table (account_number char(10), branch_name char(15), balance numeric(12,2))` AS return (Select `account_number, branch_name, balance` From `account A` Where Exists (Select * From `depositor D` Where `D.customer_name = accounts_of.customer_name` and `D.account_number = A.account_number` ));
Table Functions (cont’d)

Use the created table in From clause in SQL like base table or view:

Select *
From table (accounts_of (‘Smith’));
CREATE FUNCTION TrackingItemsModified(@minId int)
RETURNS @trackingItems TABLE
    ( Id int NOT NULL,
    Issued date NOT NULL,
    Category int NOT NULL,
    Modified datetime NULL ) AS
BEGIN
    INSERT INTO @trackingItems (Id, Issued, Category)
    SELECT ti.Id, ti.Issued, ti.Category
    FROM TrackingItem ti
    WHERE ti.Id >= @minId;
    UPDATE @trackingItems
    SET Category = Category + 1,
        Modified = GETDATE()
    WHERE Category%2 = 0;
    RETURN;
END;
User-Defined Types

- **Create type** construct in SQL creates user-defined type

  Create type *Dollars* as numeric (12,2) final

- **Create domain** construct creates user-defined domain types

  Create domain *person_name* char(20) not null

- Types and domains are similar. Domains can have constraints, such as not null, specified on them.
Scalar User-Defined Types

A Scalar user-defined function returns one of the scalar data types

CREATE FUNCTION WhichContinent (@Country varchar(15))
    RETURNS varchar(30)
AS
BEGIN
    declare @return varchar(30)
    select @return = Case @Country
        when 'Argentina' then 'South America'
        when 'Belgium' then 'Europe'
        when 'Brazil' then 'South America'
        when 'Canada' then 'North America'
        when 'Denmark' then 'Europe'
        when 'Finland' then 'Europe'
        when 'France' then 'Europe'
        else 'Unknown'
    End
    return @return
End
Scalar UDT

- Can be used anywhere a varchar(30) expression is allowed such as a computed column in a table, view, SQL select list
- Use of UDT

Select `dbo.WhichContinent(Customers.Country), Customers.name` From Customers;

Create table test (Country varchar(15),
Continent as (`dbo.WhichContinent(Country)`))

Insert into test values (‘USA’);
CREATE FUNCTION dbo.customersbycountry (@Country varchar(15))
RETURNS @CustomersByCountryTab table (  
    [CustomerID] [nchar] (5), [CompanyName] [nvarchar] (40),  
    [ContactName] [nvarchar] (30), [ContactTitle] [nvarchar] (30),  
    [Address] [nvarchar] (60),  
    [City] [nvarchar] (15),  
    [PostalCode] [nvarchar] (10),  
    [Country] [nvarchar] (15),  
    [Phone] [nvarchar] (24), [Fax] [nvarchar] (24) )  
AS  
BEGIN
    INSERT INTO @CustomersByCountryTab
    SELECT [CustomerID],  
            [CompanyName], [ContactName], [ContactTitle], [Address], [City], [PostalCode],  
            [Country], [Phone], [Fax]
    FROM [Northwind].[dbo].[Customers]
    WHERE country = @Country

    DECLARE @cnt INT
    SELECT @cnt = COUNT(*)
    FROM @customersbyCountryTab

    IF @cnt = 0
        INSERT INTO @CustomersByCountryTab ( [CustomerID], [CompanyName], [ContactName],  
                                              [ContactTitle], [Address], [City], [PostalCode], [Country], [Phone], [Fax] )  
        VALUES ("","No Companies Found","","","","","","","")

    RETURN
END

SELECT * FROM dbo.customersbycountry('USA')
SELECT * FROM dbo.customersbycountry('CANADA')
SELECT * FROM dbo.customersbycountry('ADF')
CREATE FUNCTION CustomersByContinent (@Continent varchar(30))
RETURNS TABLE
AS
RETURN
SELECT dbo.WhichContinent(Customers.Country) as continent, Customers.*
FROM Customers
WHERE dbo.WhichContinent(Customers.Country) = @Continent

SELECT * from CustomersbyContinent('North America')
SELECT * from CustomersByContinent('South America')
SELECT * from CustomersbyContinent('Unknown')