SQL STORED ROUTINES

SQL Functions

- SQL queries can use sophisticated math operations and functions
  - Can compute simple functions, aggregates
  - Can compute and filter results
- Sometimes, apps require specialized computations
  - Would like to use these in SQL queries, too
- SQL provides a mechanism for defining functions
  - Called User-Defined Functions (UDFs)

SQL Functions (2)

- Can be defined in a procedural SQL language, or in an external language
  - SQL:1999, SQL:2003 both specify a language for declaring functions and procedures
- Different vendors provide their own languages
  - Oracle: PL/SQL
  - Microsoft: Transact-SQL (T-SQL)
  - PostgreSQL: PL/pgSQL
  - MySQL: stored procedure support strives to follow specifications (and mostly does)
  - Some also support external languages: Java, C, C#, etc.
- As usual, lots of variation in features and syntax

What is a User-defined Function?

- Can have parameters
- Returns a value, either
  - A single scalar value
    - Unlike a stored procedure
    - Most data types are legal
  - A table
- Can be called from a SELECT statement
  - Unlike a stored procedure

Syntax

```
CREATE FUNCTION someName(
    parameters
) RETURNS someDataType
BEGIN
  code
  RETURN variable | SELECT statement;
END
```

- Function can take arguments and return values
- Can use SQL statements and other operations in body

Example SQL Function

A SQL function to count how many bank accounts a particular customer has:

```
CREATE FUNCTION account_count(
    customer_name VARCHAR(20)
) RETURNS INTEGER
BEGIN
  DECLARE a_count INTEGER;
  SELECT COUNT(*) INTO a_count FROM depositor AS d
  WHERE d.customer_name = customer_name;
  RETURN a_count;
END
```

Example SQL Function (2)

- Can use our function for individual accounts:

```
SELECT account_count('Johnson');
```

- Can include in computed results:

```
SELECT customer_name,
       account_count(customer_name) AS accts
FROM customer;
```

- Can include in `WHERE` clause:

```
SELECT customer_name FROM customer
WHERE account_count(customer_name) > 1;
```

Arguments and Return-Values

- Functions can take any number of arguments (even 0)
- Functions *must* return a value
  - Specify type of value in `RETURNS` clause
- From our example:

```
CREATE FUNCTION account_count(
        customer_name VARCHAR(20)
) RETURNS INTEGER
```

  - One argument named `customer_name`, type is `VARCHAR(20)`
  - Returns some `INTEGER` value

Table Functions

- SQL:2003 spec. includes table functions
  - Return a whole table as their result
  - Can be used in FROM clause
- A generalization of views
  - Can be considered to be parameterized views
  - Call function with specific arguments
  - Result is a relation based on those arguments
- Although SQL:2003 not broadly supported yet, most DBMSes provide a feature like this
  - …in various ways, of course…

Function Bodies and Variables

- Blocks of procedural SQL commands are enclosed with `BEGIN` and `END`
  - Defines a compound statement
  - Can have nested `BEGIN` … `END` blocks
- Variables are specified with `DECLARE` statement
  - Must appear at start of a block
  - Initial value is `NULL`
  - Can initialize to some other value with `DEFAULT` syntax
  - Scope of a variable is within its block
  - Variables in inner blocks can shadow variables in outer blocks

Example Blocks and Variables

- Our account_count function's body:

```
BEGIN
    DECLARE a_count INTEGER;
    SELECT COUNT(*) INTO a_count FROM depositor AS d
    WHERE d.customer_name = customer_name;
    RETURN a_count;
END
```

- A simple integer variable with initial value:

```
BEGIN
    DECLARE result INTEGER DEFAULT 0;
    ...
END
```

Assigning to Variables

- Can use SELECT … INTO syntax
  - For assigning the result of a query into a variable

```
SELECT COUNT(*) INTO a_count
FROM depositor AS d
WHERE d.customer_name = customer_name;
```

  - Query must produce a single row

    Note: `SELECT INTO` sometimes has multiple meanings!

    This form is specific to the body of stored routines.
    - e.g. frequently used to create a temp table from a SELECT
- Can also use `SET` syntax
  - For assigning result of a math expression to a variable

```
SET result = n * (n + 1) / 2;
```

Assigning Multiple Variables

- Can assign to multiple variables using `SELECT INTO` syntax
- Example: Want both the number of accounts and the total balance

```
DECLARE a_count INTEGER;
DECLARE total_balance NUMERIC(12,2);

SELECT COUNT(*), SUM(balance)
INTO a_count, total_balance
FROM depositor AS d NATURAL JOIN account
WHERE d.customer_name = customer_name;
```

Another Example

- Simple function to compute sum of 1..N

```
CREATE FUNCTION sum_n(n INTEGER) RETURNS INTEGER
BEGIN
    DECLARE result INTEGER DEFAULT 0;
    SET result = n * (n + 1) / 2;
    RETURN result;
END
```

- Lots of extra work in that! To simplify:

```
CREATE FUNCTION sum_n(n INTEGER) RETURNS INTEGER
BEGIN
    RETURN n * (n + 1) / 2;
END
```

Dropping Functions

- Can't simply overwrite functions in the database
  - Same as tables, views, etc.
- First, drop old version of function:

```
DROP FUNCTION sum_n;
```

- Then create new version of function:

```
CREATE FUNCTION sum_n(n INTEGER)
RETURNS INTEGER
BEGIN
    RETURN n * (n + 1) / 2;
END
```

SQL Procedures

- Functions have specific limitations
  - Must return a value
  - All arguments are input-only
  - Typically cannot affect current transaction status (i.e. function cannot commit, rollback, etc.)
  - Usually not allowed to modify tables, except in particular circumstances
- Stored procedures are more general constructs without these limitations
  - Generally can't be used in same places as functions
    - e.g. can't use in `SELECT` clause
  - Procedures don't return a value like functions do

buat sebuah SQL function yang menghitung jumlah peminjaman berdasarkan MemberID

Exercise

Books(BookID, Title, AuthorID, PublicationYear, Genre)

Authors(AuthorID, Name, BirthYear, Nationality)

Members(MemberID, Name, MembershipStartDate, Email)

Borrowings(BorrowingID, BookID, MemberID, BorrowDate, ReturnDate)

Create a SQL function to …

1. return the number of books borrowed by member with the given MemberID
2. find books of a specific genre
3. calculate the length of time to borrow a book in days

Thank You!

Defining Views

Views are relations, except that they are not physically stored.

For presenting different information to different users

Employee(ssn, name, department, project, salary)

```
CREATE VIEW  Developers AS
  SELECT name, project
  FROM  Employee
  WHERE department = “Development”
```

Payroll has access to Employee, others only to Developers

A Different View

Person(name, city)
Purchase(buyer, seller, product, store)
Product(name, maker, category)

```
CREATE VIEW  Seattle-view  AS

        SELECT  buyer, seller, product, store
        FROM    Person, Purchase
        WHERE   Person.city = "Seattle"    AND
                Person.name = Purchase.buyer
```

We have a new virtual table:
Seattle-view(buyer, seller, product, store)

A Different View

We can later use the view:

```
SELECT   name, store
FROM     Seattle-view, Product
WHERE    Seattle-view.product = Product.name  AND
         Product.category = “shoes”
```

What Happens When We Query a View ?

```
SELECT   name, Seattle-view.store
FROM     Seattle-view, Product
WHERE    Seattle-view.product = Product.name  AND
         Product.category = "shoes"
```

```
SELECT  name, Purchase.store
FROM    Person, Purchase, Product
WHERE   Person.city = "Seattle"    AND
        Person.name = Purchase.buyer   AND
        Purchase.poduct = Product.name  AND
        Product.category = "shoes"
```

Types of Views

- Virtual views:
  - Used in databases
  - Computed only on-demand – slow*er* at runtime
  - Always up to date
- Materialized views
  - Used in data warehouses
  - Precomputed offline – fast*er* at runtime
  - May have stale data

Updating Views

How can I insert a tuple into a table that doesn't exist?

Employee(ssn, name, department, project, salary)

```
CREATE VIEW  Developers AS
  SELECT name, project
  FROM  Employee
  WHERE department = "Development"
```

If we make the following insertion:

```
INSERT INTO  Developers
VALUES("Joe", "Optimizer")
```

It becomes:

```
INSERT INTO  Employee
VALUES(NULL, "Joe", NULL, "Optimizer", NULL)
```

Non-Updatable Views

```
CREATE VIEW  Seattle-view  AS

        SELECT  seller, product, store
        FROM    Person, Purchase
        WHERE   Person.city = "Seattle"    AND
                Person.name = Purchase.buyer
```

How can we add the following tuple to the view?

("Joe", "Shoe Model 12345", "Nine West")

We need to add "Joe" to Person first, but we don't have all its attributes

Answering Queries Using Views

- What if we want to *use* a set of views to answer a query.
- Why?
  - The obvious reason…
  - Answering queries over web data sources.
- *Very* cool stuff! (i.e., I did a lot of research on this).

Reusing a Materialized View

- Suppose I have **only** the result of SeattleView:

```
SELECT  buyer, seller, product, store
FROM    Person, Purchase
WHERE   Person.city = 'Seattle'    AND
                Person.per-name = Purchase.buyer
```

- and I want to answer the query

```
SELECT  buyer, seller
FROM    Person, Purchase
WHERE   Person.city = 'Seattle'    AND
                Person.per-name = Purchase.buyer AND
                Purchase.product='gizmo'.
```

Then, I can rewrite the query using the view.

Query Rewriting Using Views

Rewritten query:

```
SELECT  buyer, seller
FROM    SeattleView
WHERE   product= ‘gizmo’
```

Original query:

```
SELECT  buyer, seller
FROM    Person, Purchase
WHERE   Person.city = ‘Seattle’    AND
        Person.per-name = Purchase.buyer AND
        Purchase.product=‘gizmo’.
```