

SQL STORED ROUTINES

CS121: Relational Databases
Fall 2017 – Lecture 9

SQL Functions

2

- 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)

3

- 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

Example SQL Function

4

- 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
```

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

Example SQL Function (2)

5

- 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

6

- ❑ 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

7

- 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

8

- 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

9

- 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

10

- 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

11

- 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

12

- 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

13

- 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

14

- 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

Example Procedure

15

- Write a procedure that returns both the number of accounts a customer has, and their total balance

- Results are passed back using out-parameters

```
CREATE PROCEDURE account_summary(  
    IN customer_name VARCHAR(20),  
    OUT a_count INTEGER,  
    OUT total_balance NUMERIC(12,2)  
)  
BEGIN  
    SELECT COUNT(*), SUM(balance)  
    INTO a_count, total_balance  
    FROM depositor AS d NATURAL JOIN account  
    WHERE d.customer_name = customer_name;  
END
```

- Default parameter type is **IN**

Calling a Procedure

16

- Use the **CALL** statement to invoke a procedure

```
CALL account_summary(...);
```

- To use this procedure, must also have variables to receive the values

- MySQL SQL syntax:

```
CALL account_summary('Johnson',  
                    @j_count, @j_total);  
SELECT @j_count, @j_total;
```

- ▣ **@var** declares a temporary session variable

```
+-----+-----+  
| @j_cnt | @j_tot |  
+-----+-----+  
| 2      | 1400.00 |  
+-----+-----+
```

Conditional Operations

17

- SQL provides an if-then-else construct

```
IF cond1 THEN command1
```

```
ELSEIF cond2 THEN command2
```

```
ELSE command3
```

```
END IF
```

- ▣ Branches can also specify compound statements instead of single statements
 - Enclose compound statements with **BEGIN** and **END**
- ▣ Can leave out **ELSEIF** and/or **ELSE** clauses, as usual

Looping Constructs

18

- SQL also provides looping constructs

- **WHILE** loop:

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

- **REPEAT** loop:

```
REPEAT  
    SET n = n - 1;  
UNTIL n = 0  
END REPEAT;
```

Iteration Over Query Results

19

- Sometimes need to issue a query, then iterate over each row in result
 - ▣ Perform more sophisticated operations than a simple SQL query can perform
- Examples:
 - ▣ Many kinds of values that standard OLTP databases can't compute quickly!
 - ▣ Assign a rank to a collection of rows:
 - Can compare each row to all other rows, typically with a cross-join
 - Or, sort rows then iterate over results, assigning rank values
 - ▣ Given web logs containing individual HTTP request records:
 - Compute each client's "visit length," from requests that are within 20 minutes of some other request from the same client

Cursors

20

- Need to issue a query to fetch specific results
- Then, need to iterate through each row in the result
 - ▣ Operate on each row's values individually
- A cursor is an iterator over rows in a result set
 - ▣ Cursor refers to one row in query results
 - ▣ Can access row's values through the cursor
 - ▣ Can move cursor forward through results
- Cursors can provide different features
 - ▣ Read-only vs. read-write
 - ▣ Forward-only vs. bidirectional
 - ▣ Static vs. dynamic (when concurrent changes occur)

Cursor Notes

21

- Cursors can be expensive
- Can the operation use a normal SQL query instead?
 - ▣ (Usually, the answer is yes...)
 - ▣ Cursors let you do what databases do, but slower
- Cursors might also hold system resources until they are finished
 - ▣ e.g. DB might store query results in a temporary table, to provide a read-only, static view of query result
- Syntax varies widely across DBMSes
- Most external DB connectivity APIs provide cursor capabilities

Stored Routines and Cursors

22

- Can use cursors inside stored procedures and UDFs
- Syntax from the book:

```
DECLARE n INTEGER DEFAULT 0;
FOR r AS SELECT balance FROM account
        WHERE branch_name='Perryridge'
DO
        SET n = n + r.balance;
END FOR
```

- ▣ Iterates over account balances from Perryridge branch, summing balances
- ▣ **r** is implicitly a cursor
 - **FOR** construct automatically moves the cursor forward
- ▣ (Could compute this with a simple SQL query, too...)

MySQL Cursor Syntax

23

- *Must explicitly declare cursor variable*
`DECLARE cur CURSOR FOR
SELECT ... ;`
- *Open cursor to use query results:*
`OPEN cur;`
- *Fetch values from cursor into variables*
`FETCH cur INTO var1, var2, ... ;`
 - ▣ Next row is fetched, and values are stored into specified variables
 - ▣ Must specify the same number of variables as columns in the result
 - ▣ A specific error condition is flagged to indicate end of results
- *Close cursor at end of operation*
`CLOSE cur;`
 - ▣ Also happens automatically at end of enclosing block

Handling Errors

24

- Many situations where errors can occur in stored procedures
 - ▣ Called conditions
 - ▣ Includes errors, warnings, other signals
 - ▣ Can also include user-defined conditions
- Handlers can be defined for conditions
- When a condition is signaled, its handler is invoked
 - ▣ Handler can specify whether to continue running the procedure, or whether to exit procedure instead

Conditions

25

- Predefined conditions:
 - **NOT FOUND**
 - Query fetched no results, or command processed no results
 - **SQLWARNING**
 - Non-fatal SQL problem occurred
 - **SQLException**
 - Serious SQL error occurred

Conditions (2)

26

- Can also define application-specific conditions
 - Examples:
 - “Account overdraft!”
 - “Inventory of item hit zero.”
- Syntax for declaring conditions:

```
DECLARE acct_overdraft CONDITION
DECLARE zero_inventory CONDITION
```
- Not every DBMS supports generic conditions
 - e.g. MySQL supports assigning names to existing SQL error codes, but not creating new conditions

Handlers

27

- Can declare handlers for specific conditions
- Handler specifies statements to execute
- Handler also specifies what should happen next:
 - ▣ Continue running the procedure where it left off
 - ▣ Exit the stored procedure completely
- Syntax:
 - ▣ A continue-handler:
`DECLARE CONTINUE HANDLER FOR condition statement`
 - ▣ An exit-handler:
`DECLARE EXIT HANDLER FOR condition statement`
 - ▣ Can also specify a statement-block instead of an individual statement

Handlers (2)

28

- Handlers can do very simple things
 - ▣ e.g. set a flag to indicate some situation
- Can also do very complicated things
 - ▣ e.g. insert rows into other tables to log failure situations
 - ▣ e.g. properly handle an overdrawn account

Total Account Balance – MySQL

29

- Declared as a function – returns a value

```
CREATE FUNCTION acct_total(cust_name VARCHAR(20))
RETURNS NUMERIC(12,2)
BEGIN
    -- Variables to accumulate into
    DECLARE bal NUMERIC(12,2);
    DECLARE total NUMERIC(12,2) DEFAULT 0;

    -- Cursor, and flag for when fetching is done
    DECLARE done INT DEFAULT 0;
    DECLARE cur CURSOR FOR
        SELECT balance
        FROM account NATURAL JOIN depositor AS d
        WHERE d.customer_name = cust_name;
```

Total Account Balance (2)

30

```
-- When fetch is complete, handler sets flag
-- 02000 is MySQL error for "zero rows fetched"
DECLARE CONTINUE HANDLER FOR SQLSTATE '02000'
    SET done = 1;

OPEN cur;
REPEAT
    FETCH cur INTO bal;
    IF NOT done THEN
        SET total = total + bal;
    END IF;
UNTIL done END REPEAT;
CLOSE cur;
RETURN total;

END
```

Using Our User-Defined Function

31

- Can compute total balances now:

```
SELECT customer_name,  
       acct_total(customer_name) AS total  
FROM customer;
```

- Result:

| customer_name | total |
|---------------|---------|
| Adams | 0.00 |
| Brooks | 0.00 |
| Curry | 0.00 |
| Glenn | 0.00 |
| Green | 0.00 |
| Hayes | 900.00 |
| Jackson | 0.00 |
| Johnson | 1400.00 |
| Jones | 750.00 |
| Lindsay | 700.00 |
| Majeris | 850.00 |
| McBride | 0.00 |
| Smith | 1325.00 |
| Turner | 350.00 |
| Williams | 0.00 |

Stored Routine Benefits

32

- Very effective for manipulating large datasets in unusual ways, within the database
 - ▣ Don't incur communications overhead of sending commands and exchanging data
 - ▣ Database can frequently perform such tasks more efficiently than the applications can
- Often used to provide a secure interface to data
 - ▣ e.g. banks will lock down data tables, and only expose certain operations through stored procedures
- Can encapsulate business logic in procedures
 - ▣ Forbid invalid states by requiring all operations go through stored procedures

Stored Routine Drawbacks

33

- Increases load on database system
 - ▣ Can reduce performance for *all* operations being performed by DBMS
 - ▣ Need to make sure the operation *really* requires a stored procedure...
 - Most projects do not need stored procedures!
- Very hard to migrate to a different DBMS
 - ▣ Different vendors' procedural languages have *many* distinct features and limitations