Stored Routines

A *stored routine* is a set of SQL statements that are stored on the database server and given a name.

- includes *stored procedures* and *functions*

- why use stored routines?
  - convenience – to encapsulate and reuse complex operations
  - data integrity – to enforce database policies
  - security – can grant permissions for a stored routine without granting access to the underlying tables and operations

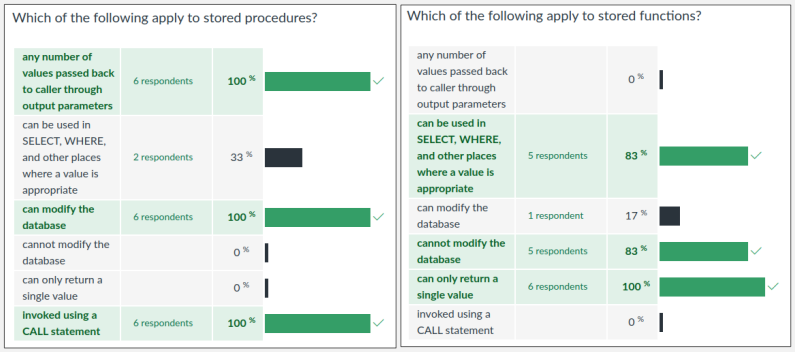
Stored Routines – Advantages

- modeling power
  - can provide more complex kinds of derived data than views
  - can enforce more complex constraints than triggers
  - can encapsulate complex operations
- reuse
  - multiple client programs written in different languages can perform the same database operations without re-implementing in each program/language
- security
  - can limit users to only executing stored routines, without having to grant access to underlying tables
- may improve performance
  - less data sent between client and server (but increases workload on server)

Stored Routines – Disadvantages

- reduced modularity
  - can lead to bottlenecks and reduced scalability
  - application logic no longer limited to the application tier
    - (though there are advantages to DB specialists writing DB-related code)
- increased complexity
  - add specific operations to the database schema
  - dependencies are hidden
- increased development and maintenance challenges
  - more difficult to test and debug
  - less portable
  - database elements often outside software development processes such as source control

Quiz



(typically) stored procedures do something – modify the database and/or produce a result set

can compute a fixed number of values

cannot be used in a context that expects a table, even if a result set is produced

stored functions compute something and return a single computed value

they can't have side effects (cannot modify the database)

Stored Routines

Stored procedures:

- invoked via CALL
- any number of values passed back to caller through output parameters
- can modify the database
- can produce a result set (table) but can't be used in a subquery

Stored functions:

- called like a function
- can only return a single value
- cannot modify the database – no side effects!

Stored routines are associated with a particular database.

Stored Routines vs Views

- functions — for reusable calculations
  - must return exactly one value
  - called inline (in an SQL statement)
  - can only calculate – cannot modify data or interact with systems outside the DB
- procedures — for complex, multi-step operations
  - can return zero or more results (through output parameters)
  - can produce a result set but can't be the target of SELECT
  - all about the side effects – can modify data and interact with systems outside the DB (e.g. export data)
    - can contain INSERT/UPDATE/DELETE statements but cannot be the target of them
- views — to simplify data access and security through abstraction
  - can only show data, not modify it
  - produces a result set which can be used in SELECT
  - in some cases, views can be the target of INSERT/UPDATE/DELETE but cannot contain them

Questions

Are stored routines created when a database is created and stored for later use, or must the user keep redefining them?

- stored routines are part of the implemented database, along with the tables, constraints, views, triggers
- defined by the database admin or another database user with permission for CREATE ROUTINE
- not intended as something users accessing data would create

Questions

When would you want to use loops?

- batch processing with complex conditions
  - e.g. update inventory records based on a list of items
- row-by-row processing
  - when the operations performed depend on the specific data in the row
  - e.g. customized email notifications to customers with overdue balances
  - e.g. apply interest to bank accounts when the amount of interest depends on the account type and balance
- cumulative operations
  - e.g. computing running total of daily sales amounts over a month, where the date and running total is needed for each day
  - e.g. computing compounded interest over multiple periods

Questions

When would you want to use loops? (continued)

- retry logic
  - e.g. attempt a connection multiple times before failing
- dynamic SQL execution
  - loops can help with building and running multiple statements

Stored Procedures

- define

```
CREATE PROCEDURE name ( parameters )
BEGIN
  statements
END
```

- call

```
CALL name(values)
```


Delimiters

Every SQL statement ends with ;, though this can be omitted in MySQL Workbench when there's only a single statement.

- but how to distinguish the ; ending statements in a routine's body and the ; that ends the routine definition?
- solution is to temporarily change delimiters
  - choose something not used elsewhere

```
DELIMITER $$
CREATE PROCEDURE name ( parameters )
BEGIN
  statements
END$$
DELIMITER ;
```


Example

Retrieve all reservations (and related information) from the sailors database –

```
CREATE PROCEDURE AllReservations ()
BEGIN
  SELECT *
  FROM SAILOR NATURAL JOIN RESERVATION
    NATURAL JOIN BOAT;
END
```

- call

```
CALL AllReservations()
```

(better done with a view)

Parameters

Parameter declaration:

MODE name type

Mode:

- IN – value comes in; any changes in the body of the procedure has no effect on the parameter's value
- OUT – procedure changes value and passes back to caller
- INOUT – value is passed in and changed value is passed back to caller

Notes:

- name cannot be the same as a table; must qualify column names if a parameter has the same name
- type should be compatible with actual column type, but doesn't have to be exactly the same

Example – Parameters

Retrieve all reservations (and related information) for a particular sailor (by name) from the sailors database –

```
CREATE PROCEDURE ReservationsFor
  ( IN sailor VARCHAR(30) )
BEGIN
  SELECT *
  FROM SAILOR NATURAL JOIN RESERVATION
    NATURAL JOIN BOAT
  WHERE Sname=sailor;
END
```

- call

```
CALL ReservationsFor('Dustin')
```


Example – Parameters

Delete a sailor and that sailor's reservations –

```
CREATE PROCEDURE DeleteSailor ( IN delsid INT )
BEGIN
  DELETE FROM RESERVATION WHERE Sid=delsid;
  DELETE FROM SAILOR WHERE Sid=delsid;
END
```


Example – OUT Parameters

Retrieve the number of reservations for a particular sailor from the sailors database –

```
CREATE PROCEDURE NumReservations( IN sailor VARCHAR(30),
                                  OUT numres INT )
BEGIN
  SELECT COUNT(*)
  FROM SAILOR NATURAL JOIN RESERVATION NATURAL JOIN BOAT
  WHERE Sname=sailor
  INTO numres;
END
```

(better done with a stored function)

Accessing OUT Parameters

Call:

`CALL` *`name`* `(` *`vars`* `)`

- `@varname` to reference a variable

Access result:

```
SELECT @varname
```


Example – OUT Parameters

Call the stored procedure and access the result:

```
CALL NumReservations('Dustin',@num);
SELECT @num;
```

- the number of reservations ends up in `@num`

@num can be used in places a value is expected:

```
CALL NumReservations('Dustin',@num);
SELECT Sname,COUNT(*) AS 'Number of Reservations'
FROM RESERVATION NATURAL JOIN SAILOR
GROUP BY Sid
HAVING COUNT(*) = @num;
```

- retrieve names and number of reservations for each sailor having the same number of reservations as Dustin