Exam Review Document

Stored procedures

This chapter explains how you can take advantage of stored procedures in developing a distributed application. Stored procedures provide a standard way to call an external procedure from within an application by using an SQL statement.

_ The concepts and benefits of stored procedures
_ The types of stored procedures
_ The CREATE PROCEDURE command

Stored procedure types

There are two categories into which stored procedures can be divided:

1. SQL stored procedures
2. External stored procedures

SQL stored procedures

SQL stored procedures are written in the SQL language. This makes it easier to port stored procedures from other database management systems (DBMS) to the iSeries server and from the iSeries server to other DBMS. Implementation of the SQL stored procedures is based on procedural SQL standardized in SQL99.

External stored procedure

An external stored procedure is written by the user in one of the programming languages on the iSeries server. You can compile the host language programs to create *PGM objects or Service Program. To create an external stored procedure, the source code for the host language must be compiled so that a program object is created. Then the CREATE PROCEDURE statement is used to tell the system where to find the program object that implements this stored procedure. The stored procedure registered in the following example returns the name of the supplier with the highest sales in a given month and year. The procedure is implemented in ILE RPG with embedded SQL:

```
c/EXEC SQL
c+ CREATE PROCEDURE HSALE
c+ (IN YEAR INTEGER ,
c+ IN MONTH INTEGER ,
c+ OUT SUPPLIER_NAME CHAR(20) ,
c+ OUT HSALE DECIMAL(11,2))
c+ EXTERNAL NAME SPROCLIB.HSALES
c+ LANGUAGE RPGLE
c+ PARAMETER STYLE GENERAL
c/END_EXEC
```

The following SQL CALL statement calls the external stored procedure, which returns a supplier name with the highest sales:

```
c/EXEC SQL
```

```
c+ CALL HSALE(:PARM1, :PARM2, :PARM3, :PARM4)
c/END-EXEC
```

CREATE PROCEDURE

The CREATE PROCEDURE statement can be used to create any of the two types of stored procedures. This statement can be issued interactively, or it can be embedded in an application program. After a procedure is registered, it can be called from any interface supporting the SQL CALL statement. During stored procedure creation, you can control characteristics that affect the way the stored procedure is identified in RELATIONAL Universal Database. This section explains some of them.

SPECIFIC specific-name

RELATIONAL Universal Database identifies each stored procedure with a specific name that, combined with the specific schema, must be unique in the system. This gains importance because multiple stored procedures with the same name but different signatures must have different specific names. If you do not provide a specific name, RELATIONAL Universal generates one automatically. If the SQL procedure name is longer than 10 characters, this name can be used to specify the C program name instead of having RELATIONAL generate one automatically, as shown in the following example:

```
CREATE PROCEDURE SAMPLE.ALLOCATECOSTS(...)
...
SPECIFIC ALLOCATECOSTS_3PARMS
```

CONTAINS SQL, READS SQL DATA, MODIFIES SQL DATA, or NO SQL

These options allow you to set some limits in regard to what the stored procedure can use.

CONTAINS SQL The stored procedure contains SQL. It can only contain:

- Non-executable statements (such as DECLARE statements)
- CALL statements to procedures with NO SQL or CONTAINS SQL attribute
- FREE LOCATOR
- SET RESULT SET
- SET assignment and VALUES INTO as long as only variables or constants are referenced
- COMMIT, ROLLBACK, or SET TRANSACTION
- CONNECT, DISCONNECT, RELEASE, or SET CONNECTION

NO SQL The stored procedure does not contain SQL statements.

READS SQL DATA The stored procedure possibly reads data using SQL. It can contain SQL statements other than:

- COMMIT, ROLLBACK, or SET TRANSACTION
- CONNECT, DISCONNECT, RELEASE, or SET CONNECTION
- DELETE, INSERT, or UPDATE
- ALTER TABLE, COMMENT ON, any CREATE, DROP, GRANT, LABEL
- ON, RENAME, or REVOKE statement

MODIFIES SQLDATA The stored procedure possibly modifies data using SQL. It can contain SQL statements other than:

- COMMIT, ROLLBACK, or SET TRANSACTION
- CONNECT, DISCONNECT, RELEASE, or SET CONNECTION

The following example shows using some of the options in a stored procedure creation:

Example of a single SQL statement stored procedure

This SQL procedure receives a customer number and customer name, and updates the
customer file with the new name:

```
CREATE PROCEDURE UPDCUST
(IN i_cusnbr CHARACTER(5),
IN i_cusnam CHARACTER(20))
      LANGUAGE SQL
      UPDATE ordapplib.customer SET CUSNAM = i_cusnam
      WHERE cusnbr = i_cusnbr;
```

Example of a compound SQL statement

Here is a more complicated example in which the procedure receives an input parameter, which is the percentage of increase for the customer credit limit. This increase is applied to all customers. The procedure returns the number of records that were updated to the calling program:

```
CREATE PROCEDURE CREDITP
(IN i_perinc DECIMAL(3,2),
OUT o_numrec DECIMAL(5,0))
LANGUAGE SQL
BEGIN
DECLARE proc_cusnbr CHAR(5);
DECLARE proc_cuscrd DECIMAL(11,2);
DECLARE numrec DECIMAL(5,0);
DECLARE at_end INT DEFAULT 0;
      DECLARE not_found
      CONDITION FOR '02000';

      DECLARE c1 CURSOR FOR
      SELECT cusnbr, cuscrd
      FROM ordapplib.customer;

DECLARE CONTINUE HANDLER FOR not_found

SET at_end = 1;
SET numrec = 0;

OPEN c1;
```

```
FETCH c1 INTO proc_cusnbr, proc_cuscrd;

WHILE at_end = 0 DO

      SET proc_cuscrd = proc_cuscrd +(proc_cuscrd * i_perinc);
      UPDATE ordapplib.customer
      SET cuscrd = proc_cuscrd
      WHERE CURRENT OF c1;
      SET numrec = numrec + 1;
      FETCH c1 INTO proc_cusnbr, proc_cuscrd;

END WHILE;

SET o_numrec = numrec;
CLOSE c1;
END
```

Returning a result set into from a Procedure

Let us look at an example that demonstrates the utility of the open cursor. In our example, a stored procedure named Get_Free_Employees returns a result set that contains those employees who are currently free from any project engagements. An input parameter is used to limit the search for free employees to a specific department.

The procedure header includes the count of result sets returned by the procedure. Within the procedure itself, result sets are returned by including the WITH RETURN clause on the cursor definition and then leaving that cursor open before exiting the procedure.

.

```
CREATE PROCEDURE Get_Free_Employees (IN dept CHAR(3))
BEGIN
      IF dept NOT IN ('D11','D21','ALL') THEN
            SIGNAL SQLSTATE VALUE '75001' SET MESSAGE_TEXT
      ='Invalid department input';
      END IF;
SELECT distinct firstname AS fname, lastname AS lname, workdept
AS deptnum
FROM employee e
WHERE workdept=dept AND
((CURRENT DATE > ALL (SELECT emp_enddate FROM emp_activeprojects
p
WHERE e.empno=p.empno AND emp_enddate IS NOT NULL)
AND empno NOT IN (SELECT empno FROM emp_activeprojects WHERE
emp_enddate IS NULL));

END;
```

Trigger concepts

Triggers are *application independent.* They are user-written programs that are activated by the database manager when a data change is performed in the database. Triggers are mainly intended for monitoring database changes and taking appropriate actions. The main advantage of using triggers, instead of calling the program from within an application, is that triggers are activated automatically, regardless of the interface that generated the data change.

In addition, once a trigger is in place, application programmers and end users cannot circumvent it. When a trigger is activated, the control shifts from the application program to the database manager. The operating system executes your coded trigger program to perform the actions you designed. The application waits until the trigger ends and then gains control again.

Types of triggers

There are two types of triggers available in RELATIONAL for database tables. Up to 300 triggers can be defined for a single table. The two types of triggers are:

- SQL triggers
- External triggers

SQL triggers

For a SQL trigger, the program performing the tests and actions is written using SQL statements. The SQL CREATE TRIGGER statement provides a way for the database management system to actively control, monitor, and manage a group of tables whenever an insert, update, or delete operation is performed. The statements specified in the SQL trigger are executed each time an insert, update, or delete operation is performed. An SQL trigger may call stored procedures or user-defined functions to perform additional processing when the trigger is executed.

Unlike stored procedures, an SQL trigger cannot be directly called from an application. Instead, an SQL trigger is invoked by the database management system upon the execution of a triggering insert, update, or delete operation. The definition of the SQL trigger is stored in the database management system and is invoked by the database management system, when the SQL table, that the trigger is defined on, is modified.

External triggers

For an external trigger, the program containing the set of trigger actions can be defined in any supported high-level language that creates a *PGM object. The trigger program can have SQL embedded in it. To define an external trigger, you must create a trigger program and

add it to a table using the Add Physical File Trigger (ADDPFTRG) CL command. To add a trigger to a table, you must:

1. Identify the table.
2. Identify the kind of operation.
3. Identify the program that performs the desired actions.
4. Provide a unique name for the trigger or let the system generate a unique name.

Enabling and disabling a trigger

Triggers need to be enabled in order to run. However, disabling a trigger allows you to work with the table without causing the trigger to run. This can be useful in cases where a long batch processing has to be done or a large data load is going to takeplace. In such cases, it can be beneficial to disable the triggers associated with a table.

Structure of an SQL trigger

A SQL trigger can be created by either specifying the CREATE TRIGGER SQL statement or by using iSeries Navigator. The SQL-routine-body is the executable part of the trigger that is transformed by the database manager into a program. When an SQL routine or trigger is created, SQL creates a temporary source file (QTEMP/QSQLSRC) that will contain C source code with embedded SQL statements. An SQL trigger is created as a program (*PGM) object using the Create SQL CI (CRTSQLCI) and Create Program (CRTPGM) commands. The program is created in the library that is the implicit or explicit qualifier of the trigger name.

The SQL routine body is a single SQL statement, including an SQL control statement. When the program is created, the SQL statements other than control statements become embedded SQL statements in the program.

```
CREATE TRIGGER name-of-trigger
ON TABLE_NAME
      FOR EACH ROW / FOR EACH STATEMENT
            BEGIN
            IF (Condition)

                  Routine body of the trigger
            END
```

Components of the SQL trigger definition

Several criteria are defined when creating an SQL trigger:

- **Subject table**: Defines the table for which the trigger is defined.
- **Trigger event**: Defines a specific SQL operation that modifies the subject table. The Operations can be to DELETE, INSERT, or UPDATE a row.

Activation time: Defines whether the trigger should be activated *before* or *after* the trigger event is performed on the subject table.

Therefore, a table can be associated with six types of SQL triggers:

- Before Delete trigger
- Before Insert trigger
- Before Update trigger
- After Delete trigger
- After Insert trigger
- After Update trigger

The Update SQL trigger allows you to define a trigger at a column level for the subject table.

Trigger granularity: Defines whether the actions of the trigger will be performed once FOR EACH STATEMENT or once FOR EACH ROW in the set of affected rows.

Correlation variables: The triggered action may refer to the values in the set of affected rows; this is similar to the concept of the trigger buffer used in the system (external) triggers. In SQL triggers, this is supported through a transition variable or correlation variables. Correlation variables use the names of the columns in the subject table qualified by a specified name (for example, OLD. or NEW.) that identifies whether the reference is to the old value (BEFORE the UPDATE) or the new value (AFTER the UPDATE). The new value can also be changed using the SET correlation variable SQL statement in BEFORE UPDATE or INSERT triggers.

Trigger mode: There are two trigger modes:

1. **MODE RELATIONALROW**
   - Triggers are activated on each row operation.
   - Valid for both BEFORE and AFTER activation time.
   - Applies to row-level triggers only.

2. **MODE RELATIONALSQL**
   - Triggers are activated after all row operations.
     - Row-level: Trigger called n times after nth row processed
     - Statement-level: Trigger called once after the *n*th row is processed
   - Only allowed on AFTER triggers (mode is automatically changed to RELATIONALROW on BEFORE triggers).
   - Not as efficient as RELATIONALROW since each row is processed twice.

Trigger body: In the body of the trigger that starts with the BEGIN clause and ends with the END clause, you can code the following SQL statements:

- SQL procedure statement
- SQL control statement
- Assignment statement
- CALL statement
- CASE statement
- Compound statement
- FOR statement
- GET diagnostics statement

- GOTO statement
- IF statement
- ITERATE statement
- LEAVE statement
- LOOP statement
- REPEAT statement
- RESIGNAL statement
- RETURN statement
  - SIGNAL statement

Simple SQL trigger example

The trigger is named NEW_HIRE, so that it increments the number of employees each time a new person is hired; that is, each time a new row is inserted into the EMPLOYEE table, it increases the value of column NBEMP in table COMPANY_STATS by one.

```
CREATE TRIGGER NEW_HIRE
AFTER INSERT ON EMPLOYEE
      FOR EACH ROW MODE RELATIONALSQL
      UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1;
```

Example of a trigger program using WHEN condition

The WHEN condition is used. It also illustrates the use of correlation variables, which are discussed “Correlation variables”. The trigger will insert a record into the SALARY_CTL table whenever the salary of an employee is decreased.

```
CREATE TRIGGER SALARY_TRACK
AFTER UPDATE OF SALARY ON EMPLOYEE
       FOR EACH ROW MODE RELATIONALSQL
       WHEN (NEW.SALARY < OLD.SALARY)
       BEGIN
       INSERT INTO SALARY_CTL (EMPNO, NEW_SALARY,
       OLD_SALARY, UPDATE_TIMESTAMP
       )
       VALUES (NEW.EMPNO, NEW.SALARY, OLD.SALARY,
       CURRENT TIMESTAMP
       );
END
```

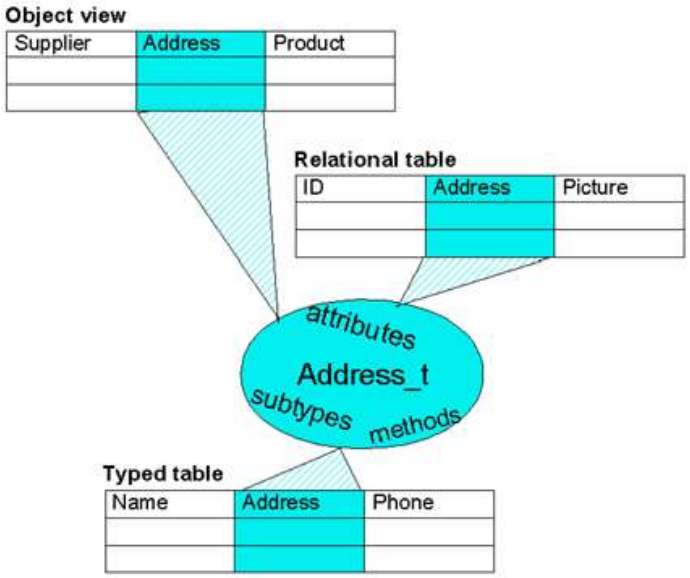
RELATIONAL's object-relational

The addition of large object types let applications handle large, complex types in familiar ways; that is, by using the relational database to store and retrieve data values. In addition, being able to name those types and to define behaviors for those types relieves the application from supporting semantic checking and business logic.

Nevertheless, all data that is stored in a large object type is essentially flat -a string of characters or bits- that may very well bear no relation to the business object that is stored there. This can put the burden on the application to read the large object into the application's work area, to "parse" the object to the place that needs to be changed, and then to store the object back in the database.

Structured types provide a way to create a schema that is more natural for those who are comfortable with OO programming languages-they include the properties of inheritance and the establishment of "is-a" relationships (value substitutability). Structured types are a more natural fit for populating Java objects in the application program.

Structured types reused in various places.

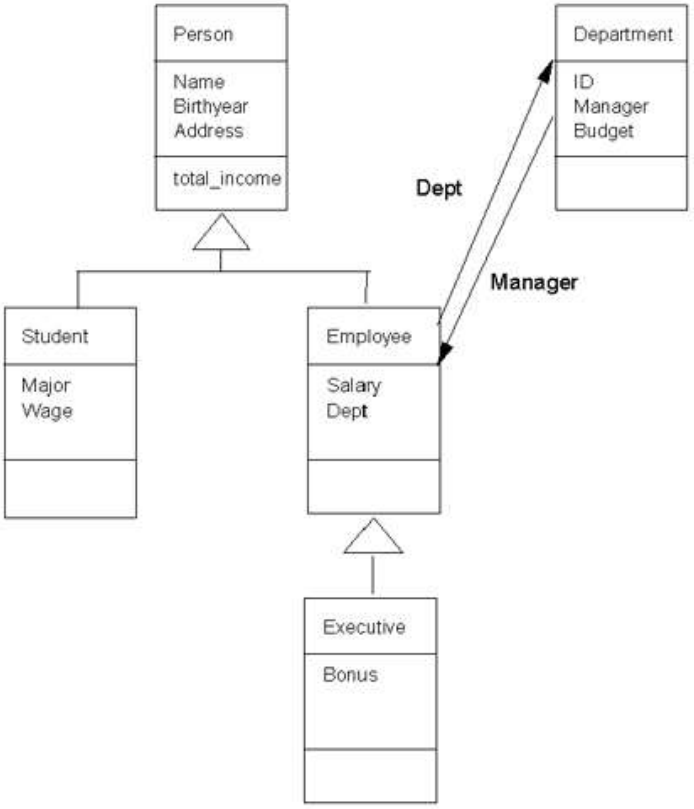


Structured type tables ("typed tables")

In this section, we focus on the basics of using typed tables. By creating an object model, it is much easier to think about mapping the model to the available constructs in RELATIONAL.

Figure 3: UML-like model of relations among people and

departments.



Think about the following:

- Will it benefit you to take advantage of type hierarchies and inheritance? With RELATIONAL's structured types, you can define type hierarchies. Just as with class hierarchies in object-oriented programming languages, objects that are modelled using RELATIONAL structured types inherit attributes from their supertypes (types that are above it in the hierarchy). In the model above, we are using inheritance for the various types of people objects that we represent.

Creating "classes": type hierarchies in RELATIONAL

When you create a user-defined structured type, the syntax is very similar to that of creating a table; you specify the name of the type, its attribute names and the data types of the attributes. If you create subtypes under a structured type, those subtypes will automatically inherit

the attributes of the type above it in the hierarchy (the *supertype*).

Figure 4: Mapping model to type hierarchies.

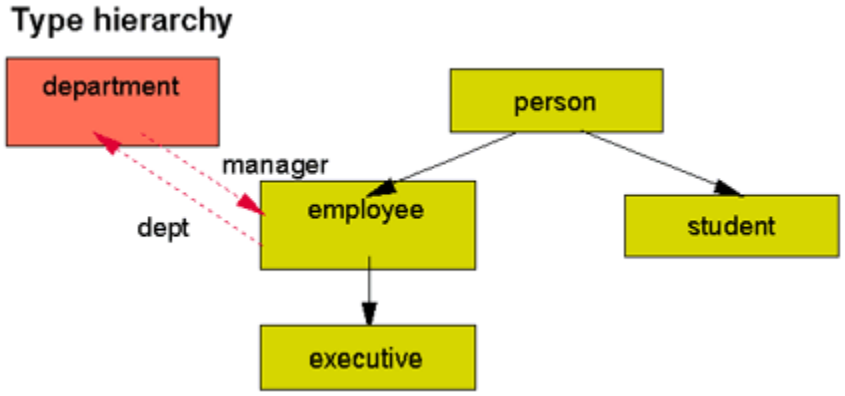


Table 1. SQL definitional statements

SQL definition statement	Action
CREATE TYPE person AS (name varchar(20), birthyear INTEGER, address varchar(40)) MODE RELATIONALSQL;	Create the root type of the hierarchy first.
CREATE TYPE employee UNDER person AS (salary INTEGER) MODE RELATIONALSQL;	Create the subtype employee UNDER the Person type. All "person" attributes are inherited by employees. All you need to specify is additional attributes that are unique to employees (or to types created UNDER employees).
CREATE TYPE executive UNDER employee AS (bonus integer) MODE RELATIONALSQL;	Executives get bonuses.
CREATE TYPE department AS (ID INTEGER, manager REF(employee), Budget INTEGER) MODE RELATIONALSQL METHOD BudgetperPerson() RETURNS INTEGER ...	The department type has no subtypes or supertypes. REF(employee) indicates that the manager is an employee, and information about employees is in the employee type. Method specifications are included with the type definition (or can be added later with ALTER TYPE).

CREATE METHOD BudgetperPerson() RETURNS INTEGER FOR department ...	As specified in the SQL99 standard, the method body is specified in a separate CREATE METHOD statement.
ALTER TYPE employee ADD ATTRIBUTE dept ref(department)	Now that department is defined, create a cyclic reference (departments referencing employees (i.e. as managers) and employees working in departments)).
CREATE TYPE student UNDER person AS (major VARCHAR(10), wage DECIMAL) MODE RELATIONALSQL;	Students are the other leg of our person hierarchy. They share attributes of person, but also have an additional attribute to indicate the major field of study.

Storing objects: Creating the table hierarchies

Table hierarchies are used to store objects and to allow SQL to change those objects. Conceptually, table hierarchies mirror the associated type hierarchies. However, types are like class libraries that can be used in different contexts, so it may be that not all of your types in a type hierarchy will be in the same table hierarchy. The only requirement is that the table hierarchy is a subset of the type hierarchy.

When you create a table, you specify an additional column for the table that stores the object identifer values for each object (each row) in the table. This object identifier (OID) is created as the first column in the table, followed by the attribute columns. The OID is used in dereference operations; that is, it is used to "grab" onto a row and turn it into a structured type instance. This is described in more detail in Part 2, where we describe how to invoke methods on objects stored in rows.

SQL definitional statements

Table 2. Table hierarchies

SQL definition statement	Action
CREATE TABLE person_table OF person (REF IS oid USER GENERATED, name WITH OPTIONS NOT NULL);	The CREATE TABLE statement is pretty simple. For the root table in the hierarchy, we have to give the name of the

	column that stores the OID values. In this case, we have given it the creative name of 'oid'.
CREATE TABLE employee_table OF employee UNDER person_table INHERIT SELECT PRIVILEGES;	For any subtables, the clause INHERIT SELECT PRIVILEGES indicates that all users who hold select privileges on the root table when that table is created are also granted those privileges on the subtables.
CREATE TABLE exec_table of executive UNDER employee_table INHERIT SELECT PRIVILEGES;	
CREATE TABLE dept_table OF DEPARTMENT (REF IS oid USER GENERATED, manager WITH OPTIONS SCOPE employee_table)	Here we create the department table. The SCOPE option on the column options for manager indicates that employee objects referred to from the manager column of the department table will reside in the employee table or any subtable of the employee table.
ALTER TABLE employee_table ALTER COLUMN dept ADD SCOPE dept_table	Here we create the scope of the reference going in the other direction. All departments referred to in the employee table will reside in the department table (or in any subtables subsequently created).

Inserting data

Inserting into a typed table applies to the specified target table (unlike SELECT, DELETE, and UPDATE, which apply to all tables below the target table in the hierarchy). The INSERT statement is unchanged for typed tables, other than you must remember to include the object identifier and to cast it to the appropriate REF type.

Table 3. Inserting into a typed table

Insert statement	Action
INSERT INTO dept_table (oid, ID, Budget) VALUES (department('1'), 100, 50000);	This department has an object identifier of '1'. The character constant is cast to the REF(Department) type.

```
INSERT INTO employee_table
(oid, Name, Birthyear,
Address, Salary, Dept)
VALUES (employee('6'),
'Meg Gomez', 1980,
'25 Hereandnow', 28000,
(SELECT oid FROM
Dept_table WHERE ID=100));
```

The dept attribute could be added as a constant (cast to REF(department)). In this case, a subquery is used to populate the column value.

After the employee table is populated, we can assign Harry Potts as the manager of department 100 as follows:

```
UPDATE dept_table
SET Manager=(SELECT oid from employee_table where
name='Harry Potts')
WHERE ID=100;
```
