Procedural Extension to SQL using Triggers

SS Chung
Content

1 Limitations of Relational Data Model for performing Information Processing

2 Database Triggers in SQL

3 Using Database Triggers for Information Processing within DBMS

4 Restrictions for Database Triggers
Limitations of Relational Data Model

- **Database vs. Information System (IS)**
  - DBMS manages data *regardless of* its usage
  - IS processes information *with respect to* its usage
- **Data model vs. system architecture**
  - Data model does not give interpretation in terms of the application domain
    - e.g. relational model, hierarchical model
  - IS architecture is developed so that the data can be interpreted as information about a particular applied domain
    - e.g. HR information, financial information, sales information
Table Constraints with Complex CHECK

*Number of boats plus number of sailors is* < 100

CREATE TABLE Sailors
  ( sid INTEGER,
  sname CHAR(10),
  rating INTEGER,
  age REAL,
  PRIMARY KEY (sid),
  CHECK
  ( (SELECT COUNT (S.sid) FROM Sailors S)
   + (SELECT COUNT (B.bid) FROM Boats B)
   < 100 )

- Symmetric constraint, yet associated with Sailors.
- If Sailors is empty, the number of Boats tuples can be anything!
Assertions:
Constraints over Multiple Relations

CREATE TABLE Sailors
    ( sid INTEGER,
      sname CHAR(10),
      rating INTEGER,
      age REAL,
      PRIMARY KEY (sid),
      CHECK
      ( (SELECT COUNT (S.sid) FROM Sailors S)
      + (SELECT COUNT (B.bid) FROM Boats B) < 100 )

ASSERTION
Not associated with either table.

CREATE ASSERTION smallClub
CHECK
( (SELECT COUNT (S.sid) FROM Sailors S)
+ (SELECT COUNT (B.bid) FROM Boats B) < 100 )
Triggers (Active database)

- **Trigger**: A procedure that starts automatically if specified changes occur to the DBMS
- Analog to a "daemon" that monitors a database for certain events to occur
- Three parts:
  - **Event** (activates the trigger)
  - **Condition** (tests whether the triggers should run) [Optional]
  - **Action** (what happens if the trigger runs)
- Semantics:
  - When event occurs, and condition is satisfied, the action is performed.
Event-Condition-Action (ECA)

- **Event** occurs in databases
  - e.g. addition of a new row, deletion of a row
- **Conditions** are checked
  - e.g. Is batch complete? Has student passed?
- **Actions** are executed if conditions are satisfied
  - e.g. send batch to supplier, congratulate student
Extending Information Processing Capabilities of DBMS using Triggers

- Processing of database content, performed by the DBMS engine itself, not by the application client
  - execution of the trigger (Event)
- Initiated by certain specified condition, mostly when DML, depending on the type of the trigger
  - firing of the trigger (Condition)
- All data actions performed by the trigger execute within the same transaction in which the trigger fires, but in a separate session (Action)
  - Triggers are checked for different privileges as necessary for the processed data
  - Cannot contain transaction control statements (COMMIT, SAVEPOINT, ROLLBACK not allowed)
Database Triggers in SQL

- Not specified in SQL-92, but standardized in SQL3 (SQL1999)
- Available in most enterprise DBMSs (Oracle, IBM DB2, Teradata, MS SQL server) and some public domain DBMSs (Postgres)
  - but not present in smaller desktop (Oracle Lite) and public domain DBMS (MySQL)
- Some vendor DBMS permit native extensions to SQL for specifying the triggers
  - e.g. PL/SQL in Oracle, Transact SQL in MS SQL Server
- Some DBMS also allow use of general purpose programming language instead of SQL
  - e.g. C/C++ in Poet, Java in Oracle, C#/VB in SQL Server
- Some DBMS extend the triggers beyond tables
  - for example also to views as in Oracle, Teradata
Variations in DBMS

Example: Triggers in SQL Server

- An SQL Server **trigger** is a T/SQL procedure that is invoked when a specified database activity occurs.
- Triggers can be used to:
  - Enforce business rules
  - Set complex default values
  - Update views
  - Implement referential integrity actions
- SQL Server only supports **INSTEAD OF** and **AFTER** triggers:
  - A **table** may have one or more AFTER triggers
  - AFTER triggers may **not** be assigned to views
  - A **view** or **table** may have only one INSTEAD OF trigger for each triggering action
- Triggers can roll back the transactions that caused them to be fired
Triggers – Event, Condition, Action

- Events could be:
  
  BEFORE | AFTER | INSERT | UPDATE | DELETE ON <tableName>

  e.g.: BEFORE INSERT ON Manager

- Condition is SQL expression or even an SQL query
  (query with non-empty result means TRUE)

- Action can be many different choices:
  - SQL statements, DDL and transaction-oriented statements like “commit”.
**Syntax for creating triggers in SQL**

- **Trigger name** - unique within one database schema
- **Timing** - depends on the order of controlled events (before or after or instead of)
- **Triggering event** - event which fires the trigger *(E)*
- **Filtering condition** - checked when the triggering event occurs *(C)*
- **Target** - table (or view) against which the trigger is fired; they should be both created within the same schema
- **Trigger Parameters** - parameters used to denote the record columns; preceded by colon
  - :new, :old for new and old versions of the values respectively
- **Trigger action** - SQL statements, executed when the trigger fires; surrounded by **Begin ... End** *(A)*
Types of SQL Triggers

How many times should the trigger body execute when the triggering event takes place?

- **Per statement**: the trigger body executes once for the triggering event. This is the default.
- **For each row**: the trigger body executes once for each row affected by the triggering event.

When the trigger can be fired

- Relative to the execution of an SQL DML statement (before or after or instead of it)
- Exactly in a situation depending on specific system resources (e.g. signal from the system clock, expiring timer, exhausting memory)
Syntax for Creating Statement Triggers

CREATE [OR REPLACE] TRIGGER trigger_name
timing event1 [OR event2 OR event3]
ON table_name
BEGIN
    SQL statements;
END;

The trigger body consisting of *SQL statements* will be executed only *once* according to the prescribed *timing*, when the *event1* (*event2*, *event3*) occurs against the monitored table in question *table_name*. 
Example: Registering Operations

```sql
SQL> CREATE TRIGGER increase_salary_trg
2  BEFORE UPDATE OF sal
3  ON emp
4  BEGIN
5    INSERT INTO sal_hist(increased, changedOn)
6      VALUES (‘YES’, SYSDATE);
7  END;
8  /
```

**Trigger name:** increase_salary_trg  
**Timing:** BEFORE executing the statement  
**Triggering event:** UPDATE of sal column  
**Target:** emp table  
**Trigger action:** INSERT values INTO sal_hist table
Syntax for Creating Row Triggers

CREATE [OR REPLACE] TRIGGER trigger_name
    timing event1 [OR event2 OR event3]
ON table_name
    [REFERENCING OLD AS old / NEW AS new]
FOR EACH ROW
    [WHEN condition]
BEGIN
    SQL statements;
END

The trigger body consisting of SQL statements will be executed once for each row affected by event1 (event2, event3) in the table named table_name subject to the additional condition.
Example Trigger

Assume our DB has a relation schema:

Manager (Num, Name, salary)

We want to write a trigger that:

Ensures that any new manager inserted has:

salary $\geq$ 60000
Example Trigger

CREATE TRIGGER minSalary BEFORE INSERT ON Professor

for what context?

BEGIN

check for violation here?

END;
Example Trigger

CREATE TRIGGER minSalary BEFORE INSERT ON Manager

FOR EACH ROW

BEGIN

Violation of Minimum Manager Salary?

END;
Example Trigger

```
CREATE TRIGGER minSalary BEFORE INSERT ON Manager

FOR EACH ROW

BEGIN

   IF (:new.salary < 60000) THEN
      RAISE_APPLICATION_ERROR (-20004, 'Violation of Minimum Manager Salary');
   END IF;

END IF;

END;
```
Example trigger

```
CREATE TRIGGER minSalary BEFORE INSERT ON Manager
FOR EACH ROW
DECLARE temp int; -- dummy variable not needed
BEGIN
    IF (:new.salary < 60000)
        THEN RAISE_APPLICATION_ERROR (-20004,
            'Violation of Minimum Manager Salary');
    END IF;
    temp := 10; -- to illustrate declared variables
END;
.
run;
```
Details of Trigger Example

- **BEFORE INSERT ON Manager**
  - This trigger is checked before the tuple is inserted.
- **FOR EACH ROW**
  - specifies that trigger is performed for each row inserted.
- **:new**
  - refers to the new tuple inserted.
- **If (:new.salary < 60000)**
  - then an application error is raised and hence the row is not inserted; otherwise the row is inserted.
- Use error code: -20004;
  - this is in the valid range.
Example Trigger Using Condition

CREATE TRIGGER minSalary BEFORE INSERT ON Manager FOR EACH ROW WHEN (new.salary < 60000) BEGIN RAISE_APPLICATION_ERROR (-20004, 'Violation of Minimum Manager Salary'); END;

- Conditions can refer to old/new values of tuples modified by the statement activating the trigger.
Triggers: REFERENCING

CREATE TRIGGER minSalary BEFORE INSERT ON Manager

REFERENCING NEW as newTuple

FOR EACH ROW

WHEN (newTuple.salary < 60000)

BEGIN
    RAISE_APPLICATION_ERROR (-20004, 'Violation of Minimum Manager Salary');
END;

run;
Example Trigger

CREATE TRIGGER minSalary
BEFORE UPDATE ON Manager
REFERENCING OLD AS oldTuple NEW as newTuple
FOR EACH ROW
WHEN (newTuple.salary < oldTuple.salary)
BEGIN
    RAISE_APPLICATION_ERROR (-20004, 'Salary Decreasing !!');
END;

Ensure that salary does not decrease
CREATE TRIGGER youngSailorUpdate
AFTER INSERT ON SAILORS
REFERENCING NEW TABLE AS NewSailors
FOR EACH STATEMENT
INSERT INTO YoungSailors(sid, name, age, rating)
SELECT sid, name, age, rating
FROM NewSailors N
WHERE N.age <= 18
Row vs Statement Level Trigger

- **Row** level: activated once per modified tuple
- **Statement** level: activate once per SQL statement
- **Row** level triggers can access new data, statement level triggers cannot always do that (depends on DBMS).
- **Statement** level triggers will be more efficient if we do not need to make row-specific decisions
Row vs Statement Level Trigger

- Example: Consider a relation schema

  Account (num, amount)

where we will allow creation of new accounts only during normal business hours.
Example: Statement level trigger

CREATE TRIGGER MYTRIG1
BEFORE INSERT ON Account
FOR EACH STATEMENT
--- is default
BEGIN
    IF (TO_CHAR(SYSDATE,'dy') IN ('sat','sun')) OR
        (TO_CHAR(SYSDATE,'hh24:mi') NOT BETWEEN '08:00' AND '17:00')
    THEN
        RAISE_APPLICATION_ERROR(-20500,'Cannot create new account now !!');
    END IF;
END;
Statement and Row Triggers

Example 1: Monitoring Statement Events

```sql
SQL> INSERT INTO dept (deptno, dname, loc) VALUES (50, 'EDUCATION', 'NEW YORK');
```

Execute only once even if multiple rows affected

Example 2: Monitoring Row Events

```sql
SQL> UPDATE emp
2  SET sal = sal * 1.1
3  WHERE deptno = 30;
```

Execute for each row of the table affected by the event
### DEPT table

<table>
<thead>
<tr>
<th>DEPTNO</th>
<th>DNAME</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>ACCOUNTING</td>
<td>NEW YORK</td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>DALLAS</td>
</tr>
<tr>
<td>30</td>
<td>SALES</td>
<td>CHICAGO</td>
</tr>
<tr>
<td>40</td>
<td>OPERATIONS</td>
<td>BOSTON</td>
</tr>
</tbody>
</table>

- **BEFORE statement trigger**
- **BEFORE row trigger**
- **AFTER row trigger**
- **AFTER statement trigger**

**Firing Sequence of Database Triggers on a Single Row**
Firing Sequence of Database Triggers on Multiple Rows

**EMP table**

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>ENAME</th>
<th>DEPTNO</th>
</tr>
</thead>
<tbody>
<tr>
<td>7839</td>
<td>KING</td>
<td>30</td>
</tr>
<tr>
<td>7698</td>
<td>BLAKE</td>
<td>30</td>
</tr>
<tr>
<td>7788</td>
<td>SMITH</td>
<td>30</td>
</tr>
</tbody>
</table>

**Trigger Firing Sequence**

1. BEFORE statement trigger
2. BEFORE row trigger
3. AFTER row trigger
4. BEFORE row trigger
5. AFTER row trigger
6. BEFORE row trigger
7. AFTER row trigger
8. AFTER row trigger
9. AFTER statement trigger
Example: Calculating Derived Columns

```
SQL> CREATE OR REPLACE TRIGGER derive_commission_trg
2   BEFORE UPDATE OF sal ON emp
3   FOR EACH ROW
4   WHEN (new.job = 'SALESMAN')
5   BEGIN
6     :new.comm := :old.comm * (:new.sal/:old.sal);
7   END;
8 /
```

**Trigger name:** derive_commission_trg

**Timing:** BEFORE executing the statement

**Triggering event:** UPDATE of sal column

**Filtering condition:** job = 'SALESMAN'

**Target:** emp table

**Trigger parameters:** old, new

**Trigger action:** calculate the new commission to be updated

Note: no (colon :) before new in WHEN
Trigger Execution order

1. Execute all BEFORE STATEMENT triggers
2. Disable temporarily all integrity constraints recorded against the table
3. Loop for each row in the table
   - Execute all BEFORE ROW triggers
   - Execute the SQL statement against the row and perform integrity constraint checking of the data
   - Execute all AFTER ROW triggers
4. Complete deferred integrity constraint checking against the table
5. Execute all AFTER STATEMENT triggers
Controlling Triggers using SQL

- **Disable or Re-enable a database trigger**
  
  \[\text{ALTER TRIGGER } trigger\_name \text{ DISABLE | ENABLE}\]

- **Disable or Re-enable all triggers for a table**
  
  \[\text{ALTER TABLE } table\_name \text{ DISABLE | ENABLE \ ALL TRIGGERS}\]

- **Drop a trigger**
  
  \[\text{DROP TRIGGER } trigger\_name\]
Using Database Triggers for Information Processing

- **Auditing Table Operations**
  - each time a table is accessed, auditing information is recorded against it

- **Tracking Record Value Changes**
  - each time a record value is changed, the previous value is recorded

- **Protecting Database Referential Integrity**: if foreign key points to changing records
  - referential integrity must be maintained

- **Maintenance of Semantic Integrity**
  - e.g. when the factory is closed, all employees should become unemployed

- **Storing Derived Data**
  - e.g. the number of items in the trolley should correspond to the current session selection

- **Security Access Control**
  - e.g. checking user privileges when accessing sensitive information
Auditing Table Operations

<table>
<thead>
<tr>
<th>USER_NAME</th>
<th>TABLE_NAME</th>
<th>COLUMN_NAME</th>
<th>INS</th>
<th>UPD</th>
<th>DEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCOTT</td>
<td>EMP</td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SCOTT</td>
<td>EMP</td>
<td>SAL</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>JONES</td>
<td>EMP</td>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MAX_INS</th>
<th>MAX_UPD</th>
<th>MAX_DEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Example: Counting Statement Execution

Whenever an employee record is deleted from the database, the counter in an audit table registering the number of deleted rows for the current user in system variable USER is incremented.

```
SQL> CREATE OR REPLACE TRIGGER audit_emp
2   AFTER DELETE ON emp
3   FOR EACH ROW
4   BEGIN
5       UPDATE audit_table SET del = del + 1
6       WHERE user_name = USER
7       AND table_name = 'EMP';
7   END;
8 /```

Example: Tracing Record Value Changes

<table>
<thead>
<tr>
<th>USER_NAME</th>
<th>TIMESTAMP</th>
<th>ID</th>
<th>OLD_LAST_NAME</th>
<th>NEW_LAST_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>EGRAVINA</td>
<td>12-SEP-04</td>
<td>7950</td>
<td>NULL</td>
<td>HUTTON</td>
</tr>
<tr>
<td>NGREENBE</td>
<td>10-AUG-04</td>
<td>7844</td>
<td>MAGEE</td>
<td>TURNER</td>
</tr>
</tbody>
</table>

... continuation

<table>
<thead>
<tr>
<th>OLD_TITLE</th>
<th>NEW_TITLE</th>
<th>OLD_SALARY</th>
<th>NEW_SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>ANALYST</td>
<td>NULL</td>
<td>3500</td>
</tr>
<tr>
<td>CLERK</td>
<td>SALESMAN</td>
<td>1100</td>
<td>1100</td>
</tr>
</tbody>
</table>
Example: Recording Changes

Whenever some details for an employee are deleted or updated, both the previous and new details are recorded in an audit table to allow tracing the history of changes. An insert operation cannot be recorded with this trigger as old.empno has no value.
Example: Protecting Referential Integrity

```sql
SQL> CREATE OR REPLACE TRIGGER cascade_updates
    2  AFTER UPDATE OF deptno ON dept
    3  FOR EACH ROW
    4  BEGIN
    5    UPDATE emp
    6    SET emp.deptno = :new.deptno
    7    WHERE emp.deptno = :old.deptno;
    8  END
    9  /
```

Whenever the department number changes, all employee records for this department will automatically be changed as well, so that the employees will continue to work for the same department.
Restrictions for Database Triggers

**Problem:** impossible to determine certain values during execution of a sequence of operations belonging to one and the same transaction

- **Mutating tables:** contain rows which change their values after certain operation and which are used again before the current transaction commits

- **Preventing table mutation:**
  - Should not contain rows which are constrained by rows from other changing tables (Trigger body should not update rows from another table that is changing by other event or trigger.)
  - Should not contain rows which are updated and read in one and the same operation (Row Trigger body should not contain (refer) its target table (triggering event table))
  - Should not contain rows which are updated and read via other operations during the same transaction (Trigger body should not update (or read) rows which are changing by other event or trigger during the same transaction.)
Example: Mutating Table

```
SQL> CREATE OR REPLACE TRIGGER emp_count
   2  AFTER DELETE ON emp
   3  FOR EACH ROW
   4  DECLARE
   5      num INTEGER;
   6  BEGIN
   7      SELECT COUNT(*) INTO num FROM emp;
   8      DBMS_OUTPUT.PUT_LINE(' There are now ' ||
                                    num || ' employees.');
   9  END;
 10  /
```

```
SQL> DELETE FROM emp
   2  WHERE  deptno = 30;
ERROR at line 1:
ORA-04091: table CGMA2.EMP is mutating, trigger/function may not see it
```

Under the bar is code entered in SQL-PLUS which triggers cascade_updates in this case. Triggers are not executed directly.
Example: Mutating Table (fixed)

```
SQL> CREATE OR REPLACE TRIGGER emp_count
2  AFTER DELETE ON emp
3  -- FOR EACH ROW
4  DECLARE
5      num INTEGER;
6  BEGIN
7      SELECT COUNT(*) INTO num FROM emp;
8      DBMS_OUTPUT.PUT_LINE(' There are now ' ||
                   num || ' employees.');
9  END;
10  /
```

```
SQL> DELETE FROM emp WHERE deptno = 30;

There are now 8 employees.

6 rows deleted.
```

Now the trigger becomes a statement trigger and the EMP table is no longer mutating.
Rules for Good Practice

**Rule 1:** Do not *change data* in the primary key, foreign key, or unique key columns of any table

**Rule 2:** Do not *update records* in the same table you read during the same transaction

**Rule 3:** Do not *aggregate* over the same table you are updating

**Rule 4:** Do not *read data* from a table which is updated during the same transaction

**Rule 5:** Do not use SQL DCL (Data Control Language: GRANT, REVOKE and DENY Database Permissions) statements in triggers
When to use BEFORE/AFTER

- Based on efficiency considerations or semantics.
- Suppose we perform statement-level after insert, then all the rows are inserted first, then if the condition fails, and all the inserted rows must be “rolled back”
- Not very efficient!!
Combining multiple events into one trigger

CREATE TRIGGER salaryRestrictions
AFTER INSERT OR UPDATE ON Manager
FOR EACH ROW
BEGIN
IF (INSERTING AND :new.salary < 60000) THEN
    RAISE_APPLICATION_ERROR (-20004, 'below min salary');
END IF;
IF (UPDATING AND :new.salary < :old.salary) THEN
    RAISE_APPLICATION_ERROR (-20004, 'Salary Decreasing !!');
END IF;
END;
END;
Some Points about Triggers

- Check the system tables:
  - user_triggers
  - user_trigger_cols
  - user_errors

- **ORA-04091**: mutating relation problem
  - In a **row level trigger**, you **cannot** have the body refer to the table specified in the event
Some Points about Triggers

- **Circular Trigger Problem:**
  - Target table of Event is changed in Trigger Body, which triggers itself.
  - Two triggers keep firing each other

- **Cascade Trigger**
  The action of a Trigger body triggers another trigger
  - All the cascade triggers should move (fire) forward in a direction, not in a circle.
Solutions to avoid Table Mutating Errors in Trigger

- **Compound Trigger**
  - Use Combination of Row Trigger and Statement Trigger to do the task
- Use Combination of **Compound Trigger** and **Package** to pass old and new with Local Variables for row trigger values to do the task
Syntax of Compound triggers in Oracle Server

CREATE OR REPLACE TRIGGER compound_trigger
FOR UPDATE OF salary ON employees
COMPOUND TRIGGER
-- Declarative part (optional)
-- Variables declared here have firing-statement duration.

threshold CONSTANT SIMPLE_INTEGER := 200;

BEFORE STATEMENT IS
BEGIN
    NULL;
END BEFORE STATEMENT;

BEFORE EACH ROW IS
BEGIN
    NULL;
END BEFORE EACH ROW;

AFTER EACH ROW IS
BEGIN
    NULL;
END AFTER EACH ROW;

AFTER STATEMENT IS
BEGIN
    NULL;
END AFTER STATEMENT;
END compound_trigger;
Compound Trigger Example

--FK MGRSSN of DEPARTMENT on DELETE or UPDATE of PK SSN

CREATE OR REPLACE TRIGGER EMPSUPERFK
FOR UPDATE OR DELETE OF SSN ON EMPLOYEE
COMPOUND TRIGGER
   -- declaring section
   old_ssn_value EMPLOYEE.SSN%TYPE;
   ...
   -- Row Trigger Section
   AFTER EACH ROW IS
   BEGIN
      IF UPDATING THEN
         old_ssn_value := :old.ssn;
         ...
      END IF;
      IF DELETING THEN
         ...
      END IF;
   END AFTER EACH ROW;
   -- statement Trigger section
   AFTER STATEMENT IS
   BEGIN
      UPDATE DEPARTMENT SET MGRSSN = new_ssn_value
      WHERE MGRSSN = old_ssn_value;
   END AFTER STATEMENT;
END;
/

create or replace trigger "EMPFK_UPDATE"
AFTER
update of "SSN" on "EMPLOYEE"
for each row
  DECLARE
    PRAGMA AUTONOMOUS_TRANSACTION;
begin
  update employee
  set FKssn = :new.ssn
  where FKssn = :old.ssn;
  commit;
end;
Invoking a (Java) Procedure from a Trigger

CREATE OR REPLACE PROCEDURE Before_delete (Id IN NUMBER, Ename VARCHAR2) IS language Java name 'thjvTriggers.beforeDelete (oracle.sql.NUMBER, oracle.sql.CHAR)';

CREATE OR REPLACE TRIGGER Pre_del_trigger BEFORE DELETE ON Tab FOR EACH ROW CALL Before_delete (:OLD.Id, :OLD.Ename)
The corresponding Java file is thjvTriggers.java

```java
import java.sql.*
import java.io.*
import oracle.sql.*
import oracle.oracore.*
public class thjvTriggers {
    public state void beforeDelete (NUMBER old_id, CHAR old_name)
        throws SQLException, CoreException
    {
        Connection conn = JDBCConnection.defaultConnection();
        Statement stmt = conn.createStatement();
        String sql = "insert into logtab values (" + old_id.intValue() + ", " + old_ename.toString() + ", BEFORE DELETE");
        stmt.executeUpdate(sql);
        stmt.close();
        return;
    }
}
```