Database Management Systems

Triggers

1. Active Database Systems
2. Oracle Triggers
3. Differences between Oracle and DB2 Triggers
4. Guidelines in writing triggers in Oracle
5. Trigger Design

Active database systems

- Traditional DBMS operation is passive
  - Queries and updates are explicitly requested by users
  - The knowledge of processes operating on data is typically embedded into applications
- Active database systems
  - Reactivity is a service provided by a normal DBMS
  - Reactivity monitors specific database events and triggers actions in response

Active rules

- Event
  - Database modification operation
- Condition
  - Predicate on the database state
  - If the condition is true, the action is executed
- Action
  - Sequence of SQL instructions or application procedure

Eleni Baralis
Politecnico di Torino
Rule engine

- Component of the DBMS, in charge of
  - Tracking events
  - Executing rules when appropriate
  - Based on the execution strategy of the DBMS
- Rule execution is interleaved with traditional transaction execution

Example

- The active rule manages reorder in an inventory stock
  - When the stocked quantity of a product goes below a given threshold
  - A new order for the product should be issued
- Event
  - Update of the quantity on hand for product x
  - Insert of a new product x

Example

- The active rule manages reorder in an inventory stock
  - When the stocked quantity of a product goes below a given threshold
  - A new order for the product should be issued
- Condition
  - The quantity on hand is below a given threshold and there are no pending orders for product x
- Action
  - Issue an order with given reorder quantity for product x

Applications of active rules

- Internal applications
  - Maintenance of complex integrity constraints
  - Replication management
  - Materialized view maintenance
- Business Rules
  - Incorporate into the DBMS application knowledge
    - E.g., reorder rule
- Alerters
  - Widely used for notification

Triggers

- Commercial products implement active rules by means of triggers
- SQL provides instructions for defining triggers
  - Triggers are defined by means of the DDL instruction CREATE TRIGGER
- Trigger syntax and semantics are covered in the SQL3 standard
  - Some commercial products implement different features with respect to the standard

Trigger structure

- Event
  - Insert, delete, update of a table
  - Each trigger can only monitor events on a single table
- Condition
  - SQL predicate (it is optional)
- Action
  - Sequence of SQL instructions
  - Proprietary programming language blocks
    - E.g., Oracle PL/SQL
  - Java block
Execution process

When the events take place [triggering]
If the condition is true [evaluation]
Then the action is executed [execution]

Seems very simple but...
- Execution modes
- Execution granularity

Execution mode

Immediate
- The trigger is executed immediately before or after the triggering statement
Deferred
- The trigger is executed immediately before commit
- Only the immediate option is available in commercial systems

Execution granularity

Tuple (or row level)
- One separate execution of the trigger for each tuple affected by the triggering statement
Statement
- One single trigger execution for all tuples affected by the triggering statement

Granularity example

Table T

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>8</td>
<td>20</td>
</tr>
</tbody>
</table>

Transaction statement
UPDATE T
SET A=A+1
WHERE B<10;

Trigger execution
- A row level trigger executes twice
- A statement level trigger executes once

Trigger syntax

CREATE TRIGGER TriggerName
Mode Event (OR Event)
ON TargetTable
[[ REFERENCING ReferenceName]]
FOR EACH ROW
[WHEN Predicate]]
PL/SQL Block
Triggers

Trigger syntax

CREATE TRIGGER TriggerName
  Mode Event (OR Event)
  ON TargetTable
  [[ REFERENCING ReferenceName]]
  FOR EACH ROW
  [WHEN Predicate]]
  PL/SQL Block

- Mode is BEFORE or AFTER
- Also INSTEAD OF but it should be avoided

Event ON TargetTable is
- INSERT
- DELETE
- UPDATE [OF ColumnName]

FOR EACH ROW specifies row level execution semantics
- If omitted, the execution semantics is statement level

The old and new states of the row triggering a row level trigger may be accessed by means of the
- OLD.ColumnName variable
- NEW.ColumnName variable

To rename the state variables
- REFERENCING OLD AS OldVariableName
- similarly for NEW
Triggers

Trigger syntax

CREATE TRIGGER TriggerName
Mode Event (OR Event)
ON TargetTable
[[ REFERENCING ReferenceName]]
FOR EACH ROW
[WHEN Predicate]]
PL/SQL block

- The action is
  - a sequence of SQL instructions
  - a PL/SQL block
- No transactional and DDL instructions

Execution modes
- immediate before
- immediate after

Granularity is
- row (tuple)
- statement

Execution is triggered by insert, delete, or update statements in a transaction

Trigger semantics

Execution algorithm
1. Before statement triggers are executed
2. For each tuple in TargetTable affected by the triggering statement
   a) Before row triggers are executed
   b) The triggering statement is executed
      + integrity constraints are checked on tuples
   c) After row triggers are executed
3. Integrity constraints on tables are checked
4. After statement triggers are executed

Trigger semantics

- The execution order for triggers with the same event, mode and granularity is not specified
  - it is a source of non determinism
- When an error occurs
  - rollback of all operations performed by the triggers
  - rollback of the triggering statement in the triggering transaction

Non termination

- Trigger execution may activate other triggers
  - Cascaded trigger activation may lead to non termination of trigger execution

- A maximum length for the cascading trigger execution may be set
  - default = 32 triggers
- If the maximum is exceeded
  - an execution error is returned

Mutating tables

- A mutating table is the table modified by the statement (i.e., event) triggering the trigger
- The mutating table
  - cannot be accessed in row level triggers
  - may only be accessed in statement triggers

- Limited access on mutating tables only characterizes Oracle applications
  - accessing mutating tables is always allowed in SQL3
Example

- Trigger to manage reorder in an inventory stock
  - when the stocked quantity of a product goes below a given threshold
  - a new order for the product should be issued
- The following database schema is given
  - Inventory (Part#, QtyOnHand, ThresholdQty, ReorderQty)
  - PendingOrders (Part#, OrderDate, OrderedQty)

Example

- Trigger to manage reorder in an inventory stock
  - when the stocked quantity of a product goes below a given threshold
  - a new order for the product should be issued
- Event
  - Update of the quantity on hand for product x
  - Insert of a new product x
- Execution semantics
  - After the modification event
  - Separate execution for each row of the Inventory table

Example

- Trigger to manage reorder in an inventory stock
  - when the stocked quantity of a product goes below a given threshold
  - a new order for the product should be issued
- Condition
  - The quantity on hand is below a given threshold

Example

- Trigger to manage reorder in an inventory stock
  - when the stocked quantity of a product goes below a given threshold
  - a new order for the product should be issued
- Condition
  - The quantity on hand is below a given threshold
- Action
  - Issue an order with given reorder quantity for product x

Example

- Trigger to manage reorder in an inventory stock
  - when the stocked quantity of a product goes below a given threshold
  - a new order for the product should be issued
- Condition
  - The quantity on hand is below a given threshold and there are no pending orders for product x
  - This part cannot be introduced into the WHEN clause
- Action
  - Issue an order with given reorder quantity for product x
**Example: Trigger body**

```sql
DECLARE
N number;
BEGIN
select count(*) into N
from PendingOrders
where Part# = :NEW.Part#;
If (N=0) then
    insert into PendingOrders(Part#,OrderedQty,OrderDate)
    values (:NEW.Part#, :NEW.ReorderQty, SYSDATE);
end if;
END;
```

**Complete trigger example**

```sql
CREATE TRIGGER Reorder
AFTER UPDATE OF QtyOnHand OR INSERT ON Inventory
FOR EACH ROW
WHEN (NEW.QtyOnHand < NEW.ThresholdQty)
DECLARE
N number;
BEGIN
select count(*) into N
from PendingOrders
where Part# = :NEW.Part#;
If (N==0) then
    insert into PendingOrders(Part#,OrderedQty,OrderDate)
    values (:NEW.Part#, :NEW.ReorderQty, SYSDATE);
end if;
END;
```

---

**Concise comparison between Oracle and DB2 Triggers**

<table>
<thead>
<tr>
<th></th>
<th>Oracle</th>
<th>DB2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference to Old_Table and New_Table in statement triggers</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>When clause in statement triggers</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Execution order between row and statement triggers with same mode</td>
<td>Specified</td>
<td>Arbitrary</td>
</tr>
<tr>
<td>Execution order between triggers with same event, mode and granularity</td>
<td>Unspecified</td>
<td>Creation Order</td>
</tr>
<tr>
<td>More than one triggering event allowed</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Forbidden access to the mutating table</td>
<td>Yes for row triggers</td>
<td>No</td>
</tr>
<tr>
<td>Availability of the instead semantics</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Database modifications allowed in before triggers</td>
<td>Yes</td>
<td>Only NEW variables</td>
</tr>
</tbody>
</table>

---

**Guidelines in writing triggers in Oracle**

- Execution Mode INSTEAD OF is allowed in Oracle but it should be avoided.
- Usage of before triggers in Oracle to be compliant with the standard:
  - Modifications of the NEW variable in tuples affected by the triggering statement are allowed in before triggers.
  - Other databases modifications apart those reported in the previous point are not allowed on before triggers.
  - Before triggers cannot trigger other triggers.
The design of a single trigger is usually simple.

- Identify:
  - execution semantics
  - event
  - condition (optional)
  - action

Understanding mutual interactions among triggers is more complex.

- The action of one trigger may be the event of a different trigger.
- Cascaded execution
- If mutual triggering occurs
- Infinite execution is possible

Trigger execution properties

- Termination:
  - For an arbitrary database state and user transaction, trigger execution terminates in a final state (also after an abort).

- Confluence:
  - For an arbitrary database state and user transaction, trigger execution terminates in a unique final state, independently of the execution order of triggers.

- Termination is the most important property.
- Confluence is enforced by deterministic trigger execution.

Guaranteeing termination

- Termination is guaranteed at run time by aborting trigger execution after a given cascading length.
- Termination may be verified at design time by means of the triggering graph:
  - a node for each trigger
  - a directed edge \( T_i \rightarrow T_j \) if trigger \( T_i \) is performing an action triggering trigger \( T_j \)
- A cycle in the graph shows potential non-terminating executions.

Example

- Trigger managing salary amounts:
  - When a given average salary value is exceeded, a salary reduction is automatically enforced.

- The following table is given:
  - Employee (Emp#, Enamer, ..., Salary)

- Event:
  - Update of the Salary attribute in Employee
  - Insert into Employee
    - Will write only trigger for update.
Example

- Trigger managing salary amounts
  - When a given average salary value is exceeded, a salary reduction is automatically enforced
- The following table is given
  Employee (Emp#, Ename, ..., Salary)
- Execution semantics
  - After the modification events
  - Separate execution for each update instruction
- No condition for execution

Example

CREATE TRIGGER SalaryMonitor
AFTER UPDATE OF Salary ON Employee
FOR EACH STATEMENT
BEGIN
  update Employee
  set Salary = Salary * K
  where 2500 < (select AVG(Salary) from Employee);
END;

The value of K may be
K = 0.9 → execution terminates
K = 1.1 → infinite execution

Example

Trigger managing salary amounts
- When a given average salary value is exceeded, a salary reduction is automatically enforced
- The following table is given
  Employee (Emp#, Ename, ..., Salary)
- Execution semantics
  - After the modification events
  - Separate execution for each update instruction
- No condition for execution

Example

CREATE TRIGGER SalaryMonitor
AFTER UPDATE OF Salary ON Employee
FOR EACH STATEMENT
BEGIN
  update Employee
  set Salary = Salary * K
  where 2500 < (select AVG(Salary) from Employee);
END;

The value of K may be
K = 0.9 → execution terminates
K = 1.1 → infinite execution

Example

CREATE TRIGGER SalaryMonitor
AFTER UPDATE OF Salary ON Employee
FOR EACH STATEMENT
BEGIN
  update Employee
  set Salary = Salary * K
  where 2500 < (select AVG(Salary) from Employee);
END;

The value of K may be
K = 0.9 → execution terminates
K = 1.1 → infinite execution

Example

CREATE TRIGGER SalaryMonitor
AFTER UPDATE OF Salary ON Employee
FOR EACH STATEMENT
BEGIN
  update Employee
  set Salary = Salary * K
  where 2500 < (select AVG(Salary) from Employee);
END;

The value of K may be
K = 0.9 → execution terminates
K = 1.1 → infinite execution

Example

CREATE TRIGGER SalaryMonitor
AFTER UPDATE OF Salary ON Employee
FOR EACH STATEMENT
BEGIN
  update Employee
  set Salary = Salary * K
  where 2500 < (select AVG(Salary) from Employee);
END;

The value of K may be
K = 0.9 → execution terminates
K = 1.1 → infinite execution

Trigger applications

- Internal applications
  - maintenance of complex integrity constraints
  - replication management
  - materialized view maintenance
- Business Rules
  - Incorporate into the DBMS application knowledge
    - E.g., reorder rule
- Alerters
  - widely used for notification

Triggers for constraint management

- Triggers are exploited to enforce complex integrity constraints
- Design procedure
  1. Write the constraint as a SQL predicate
     - It provides a condition for the trigger execution
  2. Identify the events which may violate the constraint
     - I.e. the condition
  3. Define the constraint management technique in the action

Design example (1)

The following tables are given
- Supplier S (S#, SName, ...)
- Part P (P#, PName, ...)
- Supply SP (S#, P#, Qty)
- Constraint to be enforced
  - A part may be supplied by at most 10 different suppliers

Design example (1)

Constraint predicate
  select P#
  from SP
  group by P#
  having count(*) > 10
  - set of parts violating the constraint

Events
  - insert on SP
  - update of P# on SP

Action
  - reject the violating transaction

Design example (1)

Constraint predicate
  select P#
  from SP
  group by P#
  having count(*) > 10
  - set of parts violating the constraint

Events
  - insert on SP
  - update of P# on SP

Action
  - reject the violating transaction
Design example (1)

- **Execution semantics**
  - after the modification
  - state level
    - to capture the effect of the entire modification
    - (Oracle) to allow access to the mutating table
- (Oracle) No condition
  - The condition cannot be specified in the WHEN clause
  - It is checked in the trigger body
- Design for Oracle trigger semantics

```
CREATE TRIGGER TooManySuppliers
AFTER UPDATE OF P# OR INSERT ON SP
DECLARE
  N number;
BEGIN
  select count(*) into N
  from SP
  where P# IN (select P# from SP
                group by P#
                having count(*) > 10);
  if (N <> 0) then
    raise_application_error (xxx, 'constraint violated');
  end if;
END;
```

Design example (2)

- The following tables are given
  - Supplier S (S#, SName, ...)
  - Part P (P#, PName, ...)
  - Supply SP (S#, P#, Qty)
- Constraint to be enforced
  - The quantity of a product supply cannot be larger than 1000. If it is larger, trim it to 1000.
- Check constraints do not allow compensating actions
  - Implement with a trigger

```
CREATE TRIGGER ExcessiveQty
BEFORE UPDATE OF Qty OR INSERT ON SP
FOR EACH ROW
WHEN (NEW.Qty > 1000)
BEGIN
  :NEW.Qty := 1000;
END;
```
Triggers for materialized view maintenance

- Materialized views are queries persistently stored in the database
  - provide increased performance
  - contain redundant information
    - e.g., aggregate computations
- Triggers are exploited to maintain redundant data
  - Propagate data modifications on tables to materialized view

Tables
- Student S (SId, SName, DCId)
- Degree course DC (DCId, DCName)

Materialized view
- Enrolled students ES (DCId, TotalStudents)
  - For each degree course, TotalStudents counts the total number of enrolled students
  - Defined by query
    $$\text{SELECT DCId, COUNT(*) FROM S GROUP BY DCId;}$$

Design example (3)

Tables
- Student S (SId, SName, DCId)
- Degree course DC (DCId, DCName)
- Enrolled students ES (DCId, TotalStudents)

Materialized view
- For each degree course, TotalStudents counts the total number of enrolled students
- A new degree course is inserted in materialized view ES when the first student is enrolled in it
- A degree course is deleted from ES when the last student quits it

Database schema
- S (SId, SName, DCId)
- DC (DCId, DCName)
- ES (DCId, TotalStudents)

Propagate modifications on table S to materialized view (table) ES
- Inserting new tuples into S
- Deleting tuples from S
- Updating the DCId attribute in one or more tuples of S

Design example (3)

Design three triggers to manage separately each data modification
- Insert trigger, delete trigger, update trigger
- All triggers share the same execution semantics

Execution semantics
- after the modification takes place
  - Table ES is updated after table S has been modified
- row level
  - Separate execution for each tuple of table S
    - significantly simpler to implement

Event
- insert on S
No condition
- It is always executed
Action
- if table ES contains the DCId in which the student is enrolled
  - increment TotalStudents
- otherwise
  - add a new tuple in table ES for the degree course, with TotalStudents set to 1
Insert trigger (3)

CREATE TRIGGER InsertNewStudent
AFTER INSERT ON S
FOR EACH ROW
DECLARE
N number;
BEGIN
  --- check if table ES contains the tuple for the degree course NEW.DCId in which the student enrolls
  select count(*) into N
  from ES
  where DCId = :NEW.DCId;

  if (N <> 0) then
    --- the tuple for the NEW.DCId degree course is available in ES
    update ES
    set TotalStudents = TotalStudents + 1
    where DCId = :NEW.DCId;
  else
    --- no tuple for the NEW.DCId degree course is available in ES
    insert into ES (DCId, TotalStudents) values (:NEW.DCId, 1);
  end if;
END;

Delete trigger (3)

CREATE TRIGGER DeleteStudent
AFTER DELETE ON S
FOR EACH ROW
DECLARE
N number;
BEGIN
  --- read the number of students enrolled on the degree course OLD.DCId
  select TotalStudents into N
  from ES
  where DCId = :OLD.DCId;

  if (N > 1) then
    --- there are many enrolled students
    update ES
    set TotalStudents = TotalStudents - 1
    where DCId = :OLD.DCId;
  else
    --- there is a single enrolled student
    delete from ES
    where DCId = :OLD.DCId;
  end if;
END;

Update trigger (3)

CREATE TRIGGER UpdateDCId
AFTER UPDATE OF DCId ON S
FOR EACH ROW
DECLARE
BEGIN
  --- update table ES for the degree course where the student was enrolled
  update table ES for the degree course where the student is currently enrolled
  decrement TotalStudents, or delete tuple if last student
  increment TotalStudents, or insert new tuple if first student
END;
CREATE TRIGGER UpdateDegreeCourse
AFTER UPDATE OF DCId ON S
FOR EACH ROW
DECLARE
  N number;
BEGIN
  --- read the number of students enrolled in
  --- degree course OLD.DCId
  select TotalStudents into N
  from ES
  where DCId = :OLD.DCId;
  if (N > 1) then
    --- there are many enrolled students
    update ES
    set TotalStudents = TotalStudents - 1
    where DCId = :OLD.DCId;
  else
    --- there is a single enrolled student
    delete from ES
    where DCId = :OLD.DCId;
  end if;
END;

--- check if table ES contains the tuple for the degree
--- course NEW.DCId in which the student is enrolled
select count(*) into N
from ES
where DCId = :NEW.DCId;
if (N <> 0) then
  --- the tuple for the NEW.DCId degree course is available in ES
  update ES
  set TotalStudents = TotalStudents + 1
  where DCId = :NEW.DCId;
else
  --- no tuple for the NEW.DCId degree course is available in ES
  insert into ES (DCId, TotalStudents)
  values (:NEW.DCId, 1);
end if;
END;