Extension to SQL: View, Triggers, Cursor

SS Chung
Views (Virtual Tables) in SQL

- Concept of a view in SQL
  - Single table derived from other tables
  - Considered to be a virtual table
Specification of Views in SQL

- **CREATE VIEW** command
  - Give Table (View) name, list of attribute names for the View (table), and a Query to specify the contents of the View

V1: `CREATE VIEW WORKS_ON1
    AS SELECT Fname, Lname, Pname, Hours
    FROM EMPLOYEE, PROJECT, WORKS_ON
    WHERE Ssn=Essn AND Pno=Pnumber;`

V2: `CREATE VIEW DEPT_INFO(Dept_name, No_of_emps, Total_sal)
    AS SELECT Dname, COUNT (*), SUM (Salary)
    FROM DEPARTMENT, EMPLOYEE
    WHERE Dnumber=Dno
    GROUP BY Dname;`
Alterning Existing View

Replace View *Existing_View_Name* As
Select... From... Where...;

Replace View *Works_On1* As
Select  Fname, Lname, Pno, Pname, Hours
From    Employee, Works_On, Project
Where   Ssn = Essn and Pno = Pnumber
Specification of Views in SQL (cont’d.)

- Specify SQL queries on a view
- View always up-to-date
  - Responsibility of the DBMS and not the user
- **DROP VIEW** command
  - Dispose of a view
Views

Students (sid, name, address, gpa)
Completed ( sid, course, grade)

- A view is a query stored in the database
  - Think of it as a table definition for future use
- Example view definition:
  
  ```sql
  CREATE VIEW GStudents AS
  SELECT *
  FROM Students
  WHERE gpa >= 2.5
  ```

- Views can be used like base tables, in any query or in any other view. Like a Macro. Different from Insert INTO.
Example view use: simpler queries

- Suppose you want to retrieve good students who have completed CIS430.

- \textit{GStudents} \text{(sid, name, address, gpa)}

\begin{verbatim}
SELECT S.name, S.phone 
FROM Gstudents S INNER JOIN Completed C 
  ON S.sid = C.sid 
WHERE C.course = 'CIS430';
\end{verbatim}

- It’s easier to write the query using the view.
Views for Security

This is the student table without the gpa field to hide gpa to any user in DB.

CREATE VIEW Sstudents AS
SELECT sid, name, address
FROM students
Views for Extensibility

- An old company’s database includes a table:
  
  **Part** \((\text{PartID, Name, Weight})\)

- Weight is stored in pounds
- The company is purchased by a new firm that uses metric weights
- The two databases, old and new, must be integrated and use Kg.
- But there’s lots of old software using pounds.
- Solution: views!
Views for extensibility (ctd)

Solution:

1. Base table with kilograms: \texttt{NewPart} for \texttt{new} integrated company

2. \texttt{CREATE VIEW Part AS}
   
   \texttt{SELECT PartID, Name,}
   
   \texttt{2.2046*Weight}
   
   \texttt{FROM NewPart;}

3. Old programs still call the table “Part”
View Update and Inline Views

Clause WITH CHECK OPTION
- Must be added at the end of the view definition if a view is to be updated

In-line view (dynamic Table)
- Defined in the **FROM** clause of an SQL query
View Implementation, View Update, and Inline Views

- Complex problem of efficiently implementing a view for querying

- **Query modification** approach
  - Modify view query into a query on underlying base tables
  - Disadvantage: inefficient for views defined via complex queries that are time-consuming to execute
**View Implementation**

- **View materialization approach**
  - Physically create a temporary view table when the view is first queried
  - Keep that table on the assumption that other queries on the view will follow
  - Requires efficient strategy for automatically updating the view table when the base tables are updated
View Implementation (cont’d.)

- **Incremental update strategies**
  - DBMS determines what new tuples must be inserted, deleted, or modified in a materialized view table
Problem with views: update

- Views cannot always be updated unambiguously. Consider

```
Emp (empid, ename, address, deptid)
Dept (deptid, dname)
```

CREATE VIEW EMPDEPT AS
SELECT ename, dname
FROM Emp InnerJOIN Dept ON Emp.deptid=Dept.deptid;

<table>
<thead>
<tr>
<th>EMPDEPT</th>
<th>ename</th>
<th>dname</th>
</tr>
</thead>
<tbody>
<tr>
<td>jim</td>
<td>shoe</td>
<td></td>
</tr>
<tr>
<td>joe</td>
<td>suit</td>
<td></td>
</tr>
</tbody>
</table>

- I want to delete (jim, shoe) from EMPDEPT.
- Can I do that?
View Update

View can be updated if

- It is defined on a single base table
- Using only Selection and Projection
- No Aggregates
- No DISTINCT
View Update and Inline Views

- **Update on a view defined on a single table without any aggregate functions**
  - Can be mapped to an update on underlying base table

- **View involving joins can NOT be updated**
  - Often not possible for DBMS to determine which of the updates is intended
Levels of Abstraction

- **Physical Schema**: Physical storage; DBA
- **Conceptual Schema**: Logical storage; data designer
- **External view**: User and data designer
Physical Schema

The **physical schema** is a description of how the data is physically stored in the database. It includes:

- Where the data is located
- File structures
- Access methods
- Indexes

The physical schema is managed by the DBA.
The conceptual schema:
- a logical description of how the data is stored.
- It consists of the schemas we have described with `CREATE TABLE` statements.

It is managed by the data designer.
External Schemas

Each external schema is a combination of base tables and views, tailored to the needs of a single user. It is managed by the data designer and the user.
Data Independence

A database model possesses data independence if application programs are immune to changes in the conceptual and physical schemas.

Why is this important? Everything changes.

How does the relational model achieve logical (conceptual) data independence?

- Through views

- If the conceptual schema changes, a view can be defined to preserve existing applications
Data Independence (ctd.)

- How does the relational model achieve physical data independence?
  1. Conceptual level contains no physical info
  2. SQL can program against the conceptual level
     - Earlier DBMSs (network, hierarchical) did not have these properties.
     - Their languages had physical properties embedded in them.

- That is the primary reason for the success of the relational model
Views: Summary

- A view is a stored query definition
- Views can be very useful
  - Easier query writing, security, extensibility
- But views cannot be unambiguously updated
- Three levels of abstraction in a relational DBMS
  - Yields data independence: logical and physical
Specifying Constraints as Assertions and Actions as Triggers

- **CREATE ASSERTION**
  - Specify additional types of constraints outside scope of built-in relational model constraints

- **CREATE TRIGGER**
  - Specify automatic actions that database system will perform when certain events and conditions occur
Specifying General Constraints as Assertions in SQL

- **CREATE ASSERTION**
  - Specify a query that selects any tuples that violate the desired condition
  - Use only in cases where it is not possible to use `CHECK` on attributes and domains

```sql
CREATE ASSERTION SALARY_CONSTRAINT
CHECK ( NOT EXISTS ( SELECT * FROM EMPLOYEE E, EMPLOYEE M,
                      DEPARTMENT D
                      WHERE E.Salary>M.Salary
                      AND E.Dno=D.Dnumber
                      AND D.Mgr_ssn=M.Ssn ) );
```
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 )
Assertions: Check over several tables

Employee( id, name, address, mgr, salary )

Any logical expression involving an SQL statement can be used to constrain tables in the database.

CREATE ASSERTION Salary_Mgr
CHECK (NOT EXISTS
  (SELECT *
    FROM Employee E,
    Employee M
  WHERE E.salary >
    M.salary AND
    E.mgr = M.id ))
Introduction to Triggers in SQL

- **CREATE TRIGGER** statement
  - Used to monitor the database
- Typical trigger has three components:
  - Event(s)
  - Condition
  - Action
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.
Triggers (Not a constraint)

- Three parts:
  - **Event** (activates the trigger)
    - This will be an **insert, delete and/or update** to a table
  - **Condition** (tests whether the triggers should run)
    - A **logical statement or a query**
  - **Action** (what happens if the trigger runs)
    - Can execute queries, execute data-definition commands, transaction-oriented commands, and host-language procedures

- **When** does the Action execute?
  - Specified with **Event (BEFORE, AFTER)**
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
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”.
Trigger Syntax

CREATE TRIGGER <triggerName>
BEFORE|AFTER  INSERT|DELETE|UPDATE
[OF <columnList>] ON <tableName>|<viewName>
[REFERENCING [OLD AS <oldName>]
  [NEW AS <newName>]]

[FOR EACH ROW]
( OR [FOR EACH STATEMENT] by default)

[WHEN (<condition>)]

Begin

<Trigger Action body>

End;
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)
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*. 
Triggers: Example*

Assume one donation has been inserted to indiv, for simplicity

CREATE TRIGGER MYTRIG1
BEFORE INSERT ON Account
FOR EACH STATEMENT
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;

Triggers vs Constraints

- Triggers are harder to understand
  - If multiple triggers apply, their order of execution is unspecified
  - One trigger can activate another, causing a chain of actions
  - A trigger can activate itself
- Triggers are more powerful than constraints
  - They can make changes to tables
  - They can take action before or after a data modification
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.
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)
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
Another Trigger Example

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
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 >= 60000
CREATE TRIGGER minSalary
BEFORE INSERT ON MANAGER
  
for what context ?

BEGIN

check for violation here

END;
Example Trigger

CREATE TRIGGER minSalary
BEFORE INSERT ON Manager
FOR EACH ROW
BEGIN
  Check Violation of Minimum Manager Salary
END;
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;
END;
Details of Trigger Example

- **BEFORE INSERT ON** Manager
  - This trigger is checked before the tuple is inserted in the Target Table Manager
- **FOR EACH ROW**
  - Specifies that trigger is performed for each row inserted
- **:new**
  - **:new variable** 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.
Example trigger

CREATE TRIGGER minSalary
BEFORE INSERT ON Manager
FOR EACH ROW
DECLARE temp int;  -- dummy variable for example
BEGIN
  IF (:new.salary < 60000) THEN
    RAISE_APPLICATION_ERROR(-20004, 'Violation of Minimum Manager Salary');
  END IF;
  temp := 10;  -- to illustrate declared variables
END;

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 (Event) activating the trigger.
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;
Details of Trigger Example

- **BEFORE UPDATE ON** `Emp`
  - This trigger is checked before the tuple is updated in the **Target Table** `Emp`

- **FOR EACH ROW**
  - specifies that trigger is performed for each row inserted

- **:new, :old**
  - **:new variable** refers to the **new tuple after update**
  - **:old variable** refers to the **old tuple before update**

- If `(:new.salary < :old.salary)`
  - then an application error is raised and hence the row is not updated; otherwise the row is updated.
Example Trigger

Ensure that salary does not decrease

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;
Example Trigger

Ensure that salary does not decrease

CREATE TRIGGER minSalary
BEFORE UPDATE ON Emp
FOR EACH ROW

BEGIN
  IF (:new.salary < :old.salary) THEN
    RAISE_APPLICATION_ERROR (-20004, 'Violation of Salary Increase');
  END IF;

END;

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.
Event for Statement and Row Triggers

Example 1: Monitoring Statement Events

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

Execute only once per event even if multiple rows affected

Example 2: Monitoring Row Events

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

Execute for each row of the table affected by the event
Firing Sequence of Database Triggers on a Single Row

**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 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>

- **BEFORE statement trigger**
- **BEFORE row trigger** followed by **AFTER row trigger**
- **BEFORE row trigger** followed by **AFTER row trigger**
- **BEFORE row trigger** followed by **AFTER row trigger**
- **BEFORE row trigger** followed by **AFTER row trigger**
- **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 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
You can declare a **cursor** on a table or query statement (which generates a table as a result).

You can **open** a cursor, and repeatedly **fetch** a tuple then **move** the cursor, until all tuples have been retrieved.

You can modify/delete a tuple pointed to by a cursor.

**SQL must be able to report data-generated errors.**
Cursors

- Cursor points to the current row.

- DECLARE cName CURSOR FOR select statement
  - declares the select statement

- OPEN cName
  - Executes the select statement
Embedding SQL in C: An Example

Void ListAges( int minzip)
{
    char SQLSTATE[6];
    
    EXEC SQL BEGIN DECLARE SECTION
        char c_name[20];
        char c_party[3];
        integer minzip;
    EXEC SQL END DECLARE SECTION

    • SQLSTATE holds SQL error codes
    • EXEC SQL denotes embedded SQL section
    • DECLARE SECTION binds variables into SQL
Cursor that gets names of candidates who have a principal committee, in alphabetical order

EXEC SQL DECLARE cinfo CURSOR FOR
  SELECT N.Candname
  FROM Cand N JOIN Comm M ON (N.Princomm = M.Commid)
  ORDER BY N.Candname;

OPEN cinfo;

FETCH cinfo INTO :c-name;  (probably in a loop in your program)

CLOSE cinfo; Notice the colon in :cname – it refers to a variable that has been declared in the surrounding program
EXEC SQL DECLARE cinfo CURSOR FOR
SELECT N. candname, N.par
FROM cand N
WHERE zip > :minzip
ORDER BY N.candname;

- DECLARE cinfo CURSOR: defines a name for Cursor
- SELECT ... : SQL whose results the cursor will point to
- :minzip : Note the colon referring to a C variable declared previously
Embedding in C: An Example

```c
EXEC SQL OPEN cinfo;
EXEC SQL FETCH cinfo INTO :c_name;
While(SQLSTATE != "02000")
{
    printf("Candidate name is %s\n", candname);
    EXEC SQL FETCH cinfo INTO :c_name;
}
EXEC SQL CLOSE cinfo;
```

- **OPEN cinfo**: Executes the query and positions the cursor before the first row of the output.
- **FETCH cinfo INTO**: assigns the data of the first row (if it exists) into C program variables.
- **CLOSE cinfo**: Free the cursor’s resources.