CSCC43H: Introduction to Databases

Lecture 7

Wael Aboulsaadat

Acknowledgment: these slides are partially based on Prof. Garcia-Molina & Prof. Ullman slides accompanying the course's textbook.

Database Management System (DBMS)

- A collection of programs that enable:

  Defining (describing the structure),

  Populating by data (Constructing),

  → Manipulating (querying, updating),

  – Preserving consistency,

  – Protecting from misuse,

  – Recovering from failure, and

  – Concurrent using

  of a database.

Example Database

EMPLOYEE

FirstName	Surname	Dept	Office	Salary	City
Mary	Brown	Administration	10	45	London
Charles	White	Production	20	36	Toulouse
Gus	Green	Administration	20	40	Oxford
Jackson	Neri	Distribution	16	45	Dover
Charles	Brown	Planning	14	80	London
Laurence	Chen	Planning	7	73	Worthing
Pauline	Bradshaw	Administration	75	40	Brighton
Alice	Jackson	Production	20	46	Toulouse

DEPARTMENT

DeptName	Address	City
Administration	Bond Street	London
Production	Rue Victor Hugo	Toulouse
Distribution	Pond Road	Brighton
Planning	Bond Street	London
Research	Sunset Street	San José

Views

- Views are "virtual tables" whose rows are computed from other tables (*base relations*).

- Syntax:

  `create view` *ViewName* [(*AttributeList*)] **`as`** *SelectSQL*
  [**`with`** [**`local`**|**`cascaded`**] **`check option`**]

Views

- Examples:

```
create view AdminEmployee
   (RegNo,FirstName,Surname,Salary) as
 select RegNo,FirstName,Surname,Salary
 from Employee
 where Dept = ‘Admin’ and Salary > 10;

create view JuniorAdminEmployee as
 select * from AdminEmployee
 where Salary < 50;
```

Views: why?

- Simplifies SQL
- More efficient
- Usage:

```
create view X ...;
select ...;
drop view X;
```

Views and Queries

- "Find the department with highest salary expenditures"

- *without using a view:*

```
select Dept from Employee
 group by Dept
 having sum(Salary) >= all
   (select sum(Salary) from Employee
     group by Dept)
```

Views and Queries

- "Find the department with highest salary expenditures"

- using a view:

```
create view SalBudget (Dept,SalTotal) as
   select Dept,sum(Salary) from Employee
   group by Dept;

select Dept from SalBudget
 where SalTotal =
  (select max(SalTotal) from SalBudget);
```

Views and Queries

- "Find the average number of offices per department":

  Incorrect solution (SQL does not allow a cascade of aggregate operators):

```
select avg(count(distinct Office))
from Employee group by Dept
```

  Correct solution (using a view):

```
create view  DeptOff(Dept,NoOfOffices) as
   select Dept,count(distinct Office)
   from Employee group by Dept;
```

```
select avg(NoOfOffices)from DeptOffice;
```

Data Modification in SQL

- Modification statements include:
  - Insertions (**`insert`**);
  - Deletions (**`delete`**);
  - Updates of attribute values (**`update`**).
- All modification statements operate on a set of tuples (no duplicates.)
- In the ***`condition`*** part of an update statement it is possible to access other relations.

Insertions

- Syntax:

  insert into *TableName* [(*AttributeList*)]
  < **values** (*ListOfValues*) | *SelectSQL* >

- Example using **values**:

```
insert into Department(DeptName,City)
   values(‘Production’,’Toulouse’);
```

- Example using a subquery:

```
insert into LondonProducts values
   (select Code, Description
    from Product
    where ProdArea = ‘London’);
```

Deletions

- Syntax:

```
delete from TableName [where Condition ]
```

- "Remove the Production department":

```
delete from Department
   where DeptName = 'Production'
```

- "Remove departments with no employees":

```
delete from Department
   where DeptName not in
      (select Dept from Employee)
```

Updates

- Syntax:

  update *TableName*
  set *Attribute* = < *Expression* | *SelectSQL* | **null** | **default** >
  {, *Attribute* = < *Expression* | *SelectSQL* | **null** | **default** >}
  [**where** *Condition*]

- Examples:

```
update Employee set Salary = Salary + 5
  where RegNo = ‘M2047’;
```

Database Triggers

- Triggers (also known as ECA rules) are element of the database schema.
- General form:

  on <*event*> **when** <*condition*> **then** <*action*>

  - *Event-* request to execute database operation
  - *Condition* - predicate evaluated on database state
  - *Action* – execution of procedure that might involve database updates
- Example:

  on "updating maximum enrollment limit"

  if "# registered > new max enrollment limit "

  then "deregister students using LIFO policy"

Trigger Details

- Activation — occurrence of the *event* that activates the trigger.
- Consideration — the point, after activation, when *condition* is evaluated; this can be *immediate* or *deferred*.
  - *Deferred* means that *condition* is evaluated when the database operation (*transaction*) currently executing requests to commit.
- *Condition* might refer to both the state before and the state after *event* occurs.

Trigger Execution

- This is the point when the *action* part of the trigger is carried out.
- With deferred consideration, execution is also deferred.
- With immediate consideration, execution can occur immediately after consideration or it can be deferred
  - If execution is immediate, execution can occur before, after, or instead of triggering event.
  - Before triggers adapt naturally to maintaining integrity constraints: violation results in rejection of event.

Event Granularity

Event granularity can be:

- ***Row-level***: the event involves change of a single row,
  - This means that a single **`update`** statement might result in multiple events;
- ***Statement-level***: here events result from the execution of a whole statement; for example, a single **`update`** statement that changes multiple rows constitutes a single event.

Triggers in SQL-3

- Events: **`insert`**, **`delete`**, or **`update`** statements or changes to individual rows caused by these statements.
- Condition: Anything allowed in a **`where`** clause.
- Action: An individual SQL statement or a program written in the language of Procedural Stored Modules (PSM or PL/SQL) -- which can contain embedded SQL statements.

Before-Trigger with Row Granularity

```
CREATE TRIGGER  Max_EnrollCheck
  BEFORE INSERT ON Transcript
       REFERENCING NEW AS N   --row to be added
  FOR EACH ROW
  WHEN
  ((SELECT  COUNT (T.StudId) FROM Transcript T
     WHERE  T.CrsCode = N.CrsCode
                 AND T.Semester = N.Semester)
   >=
   (SELECT C.MaxEnroll FROM Course C
     WHERE C.CrsCode = N.CrsCode ))
  THEN ABORT TRANSACTION
```

Check that enrollment ≤ limit

Action

After-Trigger with Row Granularity

```
CREATE TRIGGER LimitSalaryRaise
  AFTER UPDATE OF Salary ON Employee
  REFERENCING OLD AS O
                      NEW AS N
  FOR EACH ROW
  WHEN (N.Salary - O.Salary > 0.05 * O.Salary)
  THEN UPDATE Employee          -- action
     SET Salary = 1.05 *  O.Salary
     WHERE Id = O.Id
```

No salary raises greater than 5%

[Note: The action itself is a triggering event; however, in this case a chain reaction is not possible.]

After-Trigger with Statement Granularity

```
CREATE TRIGGER RecordNewAverage
  AFTER UPDATE OF Salary ON Employee
  FOR EACH STATEMENT
  THEN   INSERT INTO Log
    VALUES  (CURRENT_DATE,
                 SELECT AVG (Salary)
                 FROM  Employee)
```

Keep track of salary averages in the log