Basi di Dati e
Sistemi Informativi I

SQL

Data modification in SQL
Other definitions of data in SQL
Access control

Data modification in SQL

- Statements for
  - insertion (`insert`)
  - deletion (`delete`)
  - change of attribute values (`update`)
- All the statements can operate on a set of tuples (set-oriented)
- In the condition it is possible to access other relations

Insertions, 1

- Syntax:

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

- Using `values`:

```
insert into Department(DeptName, City)
   values('Production','Toulouse')
```

- Using a subquery:

```
insert into LondonProducts
   (select Code, Description
    from Product
    where ProdArea = 'London')
```

Insertions, 2

- The ordering of the attributes (if present) and of values is meaningful (first value with the first attribute, and so on)
- If *AttributeList* is omitted, all the relation attributes are considered, in the order in which they appear in the table definition
- If *AttributeList* does not contain all the relation attributes, to the remaining attributes it is assigned the default value (if defined) or the null value

Deletions, 1

- Syntax:

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

- Remove the Production department:

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

- Remove the departments without employees:

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

Deletions, 2

- The `delete` statement removes from the table all the tuples that satisfy the condition
- The removal may produce deletions from other tables if a referential integrity constraint with `cascade` policy has been defined
- If the `where` clause is omitted, `delete` removes all the tuples

- To remove all the tuples from DEPARTMENT (keeping the table schema):

```
delete from Department
```

- To remove table DEPARTMENT completely (content and schema):

```
drop table Department cascade
```

Updates, 1

- 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’

update Employee
   set Salary = Salary * 1.1
   where Dept = ‘Administration’
```

Updates, 2

- Since the language is set oriented, the order of the statements is important

```
update Employee
  set Salary = Salary * 1.1
  where Salary <= 30
update Employee
  set Salary = Salary * 1.15
  where Salary > 30
```

- If the statements are issued in this order, some employees may get a double raise

Generic integrity constraints

- The `check` clause can be used to express arbitrary constraints during schema definition
- Syntax:

  `check` (*Condition*)
- *Condition* is what can appear in a `where` clause (including nested queries)
- E.g., the definition of an attribute Superior in the schema of table EMPLOYEE:

```
Superior character(6)
check (RegNo like "1%" or
       Dept = (select Dept
               from Employee E
               where E.RegNo = Superior)
```

Assertions

- Assertions permit the definition of constraints outside of table definitions
- Useful in many situations (e.g., to express generic inter-relational constraints)
- An assertion associates a name to a `check` clause; syntax:

  `create assertion` *AssertionName* `check` (*Condition*)

- There must always be at least one tuple in table EMPLOYEE:

```
create assertion AlwaysOneEmployee
  check (1 <= (select count(*)
               from Employee))
```

Views, 1

- 'Virtual tables' whose contents depend on the contents of the other tables in the database.
- Syntax:

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

`check option` (usable only in the context of views that can be updated) specifies that update operations can be carried out only on rows belonging to the view, and after the update the rows must continue to belong to the view.

When a view is defined in term of other views, the `local` or `cascaded` option specifies whether the control over row removal must be made only at the (local) view or whether it must be propagated to all the views on which the view depends. The default option is `cascaded`

Views, 2

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

create view JuniorAdminEmployee as
select *
from AdminEmployee
where Salary < 50
with check option
```

Views, 3

- SQL views cannot be mutually dependent (no recursion)
- Views can be used to formulate complex queries
  - Views decompose the problem and produce a more readable solution
- Views are sometimes necessary to express certain queries:
  - queries that combine and nest several aggregate operators
  - queries that make a sophisticated use of the union operator

Views and queries, 1

- Find the department with the highest salary expenditure (without a view):

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

- This solution may not be recognized by all SQL systems

Views and queries, 2

- Find the department with the highest salary expenditure (using a view):

```
create view SalaryBudget (Dept,SalaryTotal) as
select Dept, sum(Salary)
from Employee
group by Dept

select Dept
from SalaryBudget
where SalaryTotal = (select max(SalaryTotal)
                     from SalaryBudget)
```

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
```

Access control

- Every component of the schema can be protected (tables, attributes, views, domains, etc.)
- The owner of a resource (the creator) assigns privileges to the other users
- A predefined user `_system` represents the database administrator and has complete access to all the resources
- A privilege is characterized by:
  - the resource
  - the user who grants the privilege
  - the user who receives the privilege
  - the action that is allowed on the resource
  - whether or not the privilege can be passed on to other users

Types of privilege

- SQL offers six types of privilege
  - `insert`: to insert a new object into the resource
  - `update`: to modify the resource content
  - `delete`: to remove an object from the resource
  - `select`: to access the resource content in a query
  - `references`: to build a referential integrity constraint with the resource (may limit the ability to modify the resource)
  - `usage`: to use the resource in a schema definition (e.g., a domain)

`grant` and `revoke`

- To grant a privilege to a user:

  `grant` < *Privileges* | `all privileges` > `on` *Resource*
  `to` *Users* [`with grant option`]

  - `grant option` specifies whether the privilege of propagating the privilege to other users must be granted
- E.g.:

  `grant select on Department to Stefano`

- To take away privileges:

  `revoke` *Privileges* `on` *Resource* `from` *Users*
  [`restrict` | `cascade`]

Embedded SQL

- Traditional applications often need to “embed” SQL statements inside the instructions of a procedural programming language (C, COBOL, etc.)
- Programs with embedded SQL use a precompiler to manage SQL statements
- Embedded statements are preceded by ‘$’ or ‘EXEC SQL’
- Program variables may be used as parameters in the SQL statements (preceded by ‘:’)
- `select` producing a single row and update commands may be embedded easily
- The SQL environment offers a predefined variable `sqlcode` which describes the status of the execution of the SQL statements (zero if the SQL statement executed successfully)

Cursors

- Fundamental problem: Impedance mismatch
  - traditional programming languages manage records one at a time (tuple-oriented)
  - SQL manages sets of tuples (set-oriented)
- Cursors solve this problem
- A cursor:
  - accesses the result of a query in a set-oriented way
  - returns the tuples to the program one by one

- Syntax of cursor definition:

  `declare` *CursorName* [`scroll`] `cursor for` *SelectSQL*
  [`for` < `read only` | `update` [`of` *Attribute* {, *Attribute*}]>]

Operations on cursors

- To execute the query associated with a cursor:

  `open` *CursorName*

- To extract one tuple from the query result:

  `fetch` [*Position* `from`] *CursorName* `into` *FetchList*

- To free the cursor, discarding the query result:

  `close` *CursorName*

- To access the current tuple (when a cursor reads a relation, in order to update it):

  `current of` *CursorName*   (in the `where` clause)

Example of embedded SQL

```
void DisplayDepartmentSalaries(char DeptName[])
{
  char FirstName[20], Surname[20];
  long int Salary;

$ declare DeptEmp cursor for
    select FirstName, Surname, Salary
    from Employee
    where Dept = :DeptName;
$ open DeptEmp;
$ fetch DeptEmp into :FirstName, :Surname, :Salary;
  printf("Department %s\n",DeptName);
  while (sqlcode == 0)
  {
    printf("Name: %s %s ",FirstName,Surname);
    printf("Salary: %d\n",Salary);
$   fetch DeptEmp into :FirstName, :Surname, :Salary;
  }
$ close DeptEmp;
}
```

Dynamic SQL

- When applications do not know at compile-time the SQL statement to execute, they need dynamic SQL
- Major problem: managing the transfer of parameters between the program and the SQL environment

- For direct execution:

  `execute immediate` *SQLStatement*
- For execution preceded by the analysis of the statement:

  `prepare` *CommandName* `from` *SQLStatement*
  - followed by:

    `execute` *CommandName* [`into` *TargetList*]
    [`using` *ParameterList*]

Procedures

- SQL-2 allows for the definition of procedures, also known as stored procedures
- Stored procedures are part of the schema

```
procedure AssignCity(:Dep char(20),
                     :City char(20))
update Department
set City = :City
where Name = :Dep
```

- SQL-2 does not handle the writing of complex procedures
- Most systems offer SQL extensions that permit to write complex procedures (e.g., Oracle PL/SQL)

Procedure in Oracle PL/SQL

```
Procedure Debit(ClientAccount char(5),Withdrawal integer) is
  OldAmount integer;
  NewAmount integer;
  Threshold integer;
begin
  select Amount, Overdraft into OldAmount, Threshold
    from BankAccount
    where AccountNo = ClientAccount
    for update of Amount;
  NewAmount := OldAmount - WithDrawal;
  if NewAmount > Threshold
    then update BankAccount
           set Amount = NewAmount
           where AccountNo = ClientAccount;
    else
      insert into OverDraftExceeded
        values(ClientAccount,Withdrawal,sysdate);
  end if;
end Debit;
```