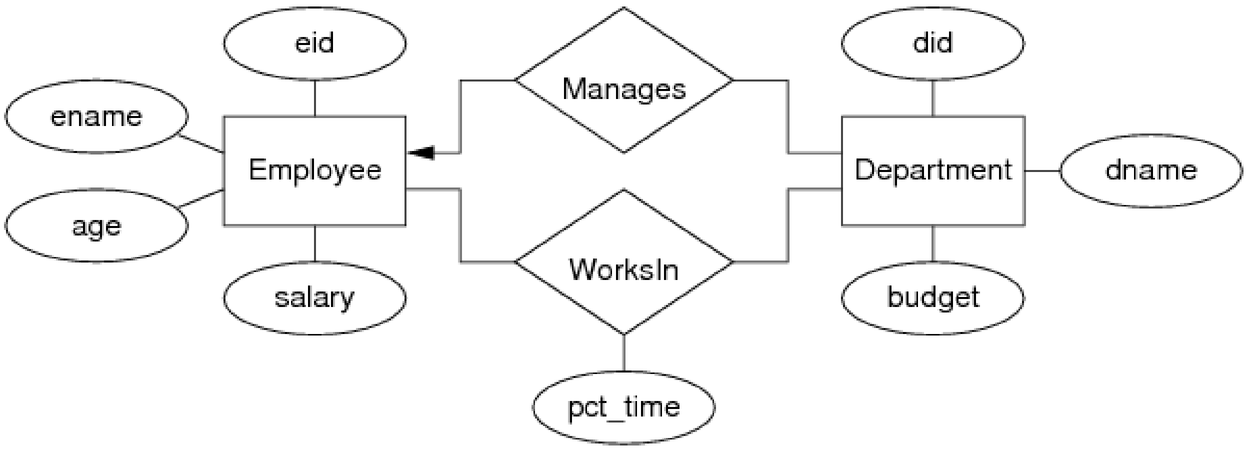
COMP3311 19T3 Database Systems

Exercises 03
Constraints and Data Modification

Consider the following data model for a a business organisation and its employees:



Employees are uniquely indentified by an id (`eid`), and other obvious information (name,age,...) is recorded about each employee. An employee may work in several departments, with the percentage of time spent in each department being recorded in the `WorksIn` relation. The percentages for a given employee may not sum to one if the employee only works part-time in the organisation. Departments are also uniquely identified by an id (`did`), along with other relevant information, including the id of the employee who manages the department.

Based on the ER design and the above considerations, here is a relational schema to represent this scenario:

```
create table Employees (
      eid       integer,
      ename     varchar(30),
      age       integer,
      salary    real,
      primary key (eid)
);
create table Departments (
      did       integer,
      dname     varchar(20),
      budget    real,
      manager   integer,
      primary key (did),
      foreign key (manager) references Employees(eid)
);
create table WorksIn (
      eid       integer,
      did       integer,
      pct_time  real,
      primary key (eid,did),
      foreign key (eid) references Employees(eid),
      foreign key (did) references Departments(did)
);
```

Answer each of the following questions for this schema:

1. Does the order of table declarations above matter?

   Answer:

   Yes table declarations order in the schema above matters. We can not insert a Department tuple until there is an Employee tuple available to be the manager of the department. We cannot also insert any

WorksIn tuple until you have both the Employee tuple and the Department tuple where the employee works.

2. A new government initiative to get more young people into work cuts the salary levels of all workers under 25 by 20%. Write an SQL statement to implement this policy change.

   Answer:

   SQL statement to reduce pay for people under 25 by 20%:

   ```
   update Employees
   set    salary = salary * 0.8
   where  age < 25;
   ```

   A straightforward applicatiom of the SQL `UPDATE` statement.

3. The company has several years of growth and high profits, and considers that the Sales department is primarily responsible for this. Write an SQL statement to give all employees in the Sales department a 10% pay rise.

   Answer:

   SQL statement to give Sales employees a 10% pay rise:

   ```
   update Employees e
   set    e.salary = e.salary * 1.10
   where  eid in
              (select eid
               from   Departments d, WorksIn w
               where  d.dname = 'Sales' and d.did = w.did
              );
   ```

   This query requires that we know which department an employee works for before updating their pay. The only information we have in the `Employees` relation to help with this is the employee id. Thus the subquery gives a list of ids for all employees working in the Sales department.

4. Add a constraint to the `CREATE TABLE` statements above to ensure that every department must have a manager.

   Answer:

   ```
      create table Departments (
         did        integer,
         dname      varchar(20),
         budget     real,
         manager    integer not null,
         primary key (did),
         foreign key (manager) references Employees(eid)
      );
   ```

   Change the definition of `Departments` to ensure that the foreign key `manager` is not null. The foreign key constraint already ensures that the value must be a primary key in the `Employees` relation but, without the `NOT NULL` declaration, a foreign key value is allowed to be null.

5. Add a constraint to the `CREATE TABLE` statements above to ensure that no-one is paid less than the minimum wage of $15,000.

   Answer:

   This is a straightforward example of a single-attribute constraint. It is effectively a constraint on the domain of the `salary` attribute. In fact, it would been sensible to have an initial constraint to ensure that the `salary` was at least positive. A similar constraint should probably be applied to the `pct_time` attribute in the `WorksIn` relation.

```
create table Employees (
   eid       integer,
   ename     varchar(30),
   age       integer,
   salary    real check (salary >= 15000),
   primary key (eid)
);
```

6. Add a constraint to the `CREATE TABLE` statements above to ensure that no employee can be committed for more than 100% of his/her time. Note that the SQL standard allows queries to be used in constraints, even though DBMSs don't implement this (for performance reasons).

   Answer:

   We have expressed this as a tuple-level constraint, even though it's really a constraint on the `eid` attribute. Note the use of scoping in the sub-query: the `eid` refers to the id of the employee that is being inserted or modified, while `w.eid` is bound by the tuple variable in the subquery. The subquery itself computes the total `pct_time` that the employee `eid` works.

   This query is valid according to Ullman/Widom's description of the SQL2 standard. However, neither PostgreSQL nor Oracle supports subqueries in `CHECK` conditions. The condition can only be an expression involving the attributes in the updated row.

```
create table WorksIn (
   eid         integer,
   did         integer,
   pct_time    real,
   primary key (eid,did),
   foreign key (eid) references Employees(eid),
   foreign key (did) references Departments(did)
   constraint  MaxFullTimeCheck
               check (1.00 >= (select sum(w.pct_time)
                                from   WorksIn w
                                where  w.eid = eid)
                      )
);
```

   In most relational database management systems, the constraint checking required here would need to be implemented via a trigger.

7. Add a constraint to the `CREATE TABLE` statements above to ensure that a manager works 100% of the time in the department that he/she manages. Note that the SQL standard allows queries to be used in constraints, even though DBMSs don't implement this (for performance reasons).

   Answer:

   We have expressed this as a tuple-level constraint, even though it's really a constraint on the `manager` attribute.

```
create table Departments (
   did         integer,
   dname       varchar(20),
   budget      real,
   manager     integer,
   primary key (did),
   foreign key (manager) references Employees(eid)
   constraint  FullTimeManager
               check (1.0 = (select w.pct_time
                              from   WorksIn w
                              where  w.eid = manager)
                     )
);
```

As in the previous question, this kind of constraint is allowed by the SQL standard, but DBMSs typically don't implement cross-table constraints like this; a trigger is required and the check has to be programmed in the trigger.

8. When an employee is removed from the database, it makes sense to also delete all of the records that show which departments he/she works for. Modify the `CREATE TABLE` statements above to ensure that this occurs.

   Answer:

   The `ON DELETE CASCADE` clause ensures that when the `Employees` record for `eid` is removed, then any `WorksIn` tuples that refer to `eid` are also removed.

   ```
   create table WorksIn (
      eid         integer,
      did         integer,
      pct_time    real,
      primary key (eid,did),
      foreign key (eid) references Employees(eid) on delete cascade,
      foreign key (did) references Departments(did)
   );
   ```

   Of course, this immediately reaises the issue of references to the `Departments` relation; this is considered in the next question.

9. When a manager leaves the company, there may be a period before a new manager is appointed for a department. Modify the `CREATE TABLE` statements above to allow for this.

   Answer:

   If the department has no manager, we indicate this by putting a value of `NULL` for the `manager` field. However, in one of the questions above, we added a NOT NULL contraint to ensure that every department *does* have a manager. To solve this question, we need to remove that constraint.

   An alternative would be to always appoint a temporary manager, which could be accomplished via an `UPDATE` statement, e.g.

   ```
   update department set manager = SomeEID where did = OurDeptID;
   ```

10. Consider the deletion of a department from a database based on this schema. What are the options for dealing with referential integrity between `Departments` and `WorksIn`? For each option, describe the required behaviour in SQL.

    Answer:

    Three possible approaches to referential integrity between between `Departments` and `WorksIn`:

    a. Disallow the deletion of a `Departments` tuple if any `Works` tuple refers to it. This is the default behaviour, which would result from the `CREATE TABLE` definition in the previous question.

    b. When a `Departments` tuple is deleted, also delete all `WorksIn` tuples that refer to it. This requires adding an `ON DELETE CASCADE` clause to the definition of `WorksIn`.

    ```
    create table WorksIn (
       eid       integer,
       did       integer,
       pct_time  real,
       primary key (eid,did),
       foreign key (eid) references Employees(eid) on delete cascade,
       foreign key (did) references Departments(did) on delete cascade
    );
    ```

    In this solution, we've added the same functionality to the `eid` field as well (see previous question).

c. For every `WorksIn` tuple that refers to the deleted department, set the `did` field to the department id of some existing 'default' department. Unfortunately, Oracle doesn't appear to implement this functionality. If it did, the definition of `WorksIn` would change to:

```
create table WorksIn (
   eid       integer,
   did       integer default 1,
   pct_time  real,
   primary key (eid,did),
   foreign key (eid) references Employees(eid) on delete cascade,
   foreign key (did) references Departments(did) on delete set default
);
```

11. For each of the possible cases in the previous question, show how deletion of the Engineering department would affect the following database:

```
  EID ENAME                AGE     SALARY
----- --------------- ----- ----------
    1 John Smith          26      25000
    2 Jane Doe            40      55000
    3 Jack Jones          55      35000
    4 Superman            35      90000
    5 Jim James           20      20000

  DID DNAME                BUDGET  MANAGER
----- --------------- ---------- --------
    1 Sales               500000        2
    2 Engineering        1000000        4
    3 Service             200000        4

  EID   DID  PCT_TIME
----- ----- ---------
    1     2      1.00
    2     1      1.00
    3     1      0.50
    3     3      0.50
    4     2      0.50
    4     3      0.50
    5     2      0.75
```

Answer:

a. Disallow ... The database would not change. The DBMS would print an error message about referential integrity constraint violation.

b. `ON DELETE CASCADE` ... All of the tuples in the `WorksIn` relation that have `did = 2` are removed, giving:

```
  DID DNAME                BUDGET  MANAGER
----- --------------- ---------- --------
    1 Sales               500000        2
    3 Service             200000        4

  EID   DID  PCT_TIME
----- ----- ---------
    2     1      1.00
    3     1      0.50
    3     3      0.50
    4     3      0.50
```

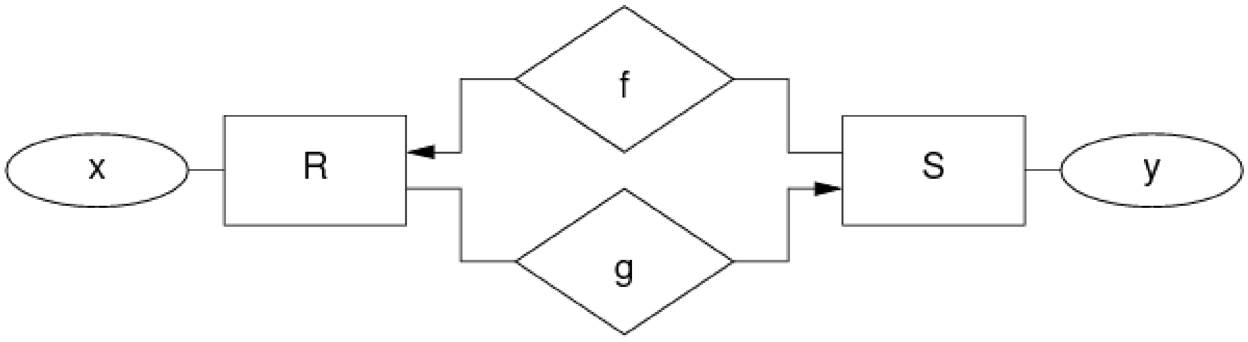
c. `ON DELETE SET NULL` ... All of the tuples in the `WorksIn` relation that have `did = 2` have that attribute modified to `NULL`, giving:

```
  DID DNAME                BUDGET   MANAGER
----- --------------- ---------- --------
    1 Sales               500000        2
    3 Service             200000        4

  EID   DID  PCT_TIME
----- ----- ---------
    1  NULL      1.00
    2     1      1.00
    3     1      0.50
    3     3      0.50
    4  NULL      0.50
    4     3      0.50
    5  NULL      0.75
```

d. `ON DELETE SET DEFAULT` ... All of the tuples in the `WorksIn` relation that have `did = 2` have that attribute modified to the default department (`1`), giving:

```
  DID DNAME                BUDGET   MANAGER
----- --------------- ---------- --------
    1 Sales               500000        2
    3 Service             200000        4

  EID   DID  PCT_TIME
----- ----- ---------
    1     1      1.00
    2     1      1.00
    3     1      0.50
    3     3      0.50
    4     1      0.50
    4     3      0.50
    5     1      0.75
```

12. Consider the following data model, not related to the workplace schema above:



and its associated relational schema:

```
create table R (
        x       integer,
        y       char(1),
        primary key (x),
        foreign key (y) references S(y)
);
create table S (
        y       char(1),
        x       integer,
        primary key (y),
        foreign key (x) references R(x)
);
```

As it stands, it is not possible to load this schema into PostgreSQL. While processing the definition of table `R`, PostgreSQL needs to know about table `S`, which does not yet exist. The standard behaviour when a referenced object doesn't exist, is to fail, and so the creation of table `R` fails, which then causes the subsequent attempt to create table `S` to fail.

Describe how might define this schema so that it *is* possible to load it into PostgreSQL.

Answer:

The trick here is to create the tables first without the referential integrity constraints, and then add them once all the tables exist:

```
create table R (
        x       integer,
        y       char(1),
        primary key (x)
);
create table S (
        y       char(1),
        x       integer,
        primary key (y)
);
alter table R add foreign key (y) references S(y);
alter table S add foreign key (x) references R(x);
```

13. Once the above schema is loaded, we want to create the following two relations:

```
db=# select * from R;
 x | y
---+---
 1 | a
(1 row)

db=# select * from S;
 y | x
---+---
 a | 1
(1 row)
```

The following doesn't work:

```
insert into R values (1,'a');
insert into S values ('a',1);
```

The first insert fails because there's not yet a tuple in `S` with a primary key value of `'a'`. The the second insert fails because the first insert failed and there's no tuple in `R` with a primary key value of `1`. Reversing the order of the `insert` statements doesn't solve the problem.

Describe how we can insert these tuples into the database.

Answer:

One possibility, analogous to the above creation of tables, is to create tuples with null foreign key values and then update them to fill in the foreign keys:

```
insert into R values (1,null);
insert into S values ('a',null);
update R set y = 'a' where x = 1;
update S set x = 1 where y = 'a';
```

This works because when we come to assign the foreign keys, the correpsonding primary key values exist in the database. This approach relies on us being able to assign `NULL` to foreign key attributes, which is the default behaviour unless the foreign key attributes are constrained to be `NOT NULL`.

14. Now consider a variation on the above database where the foreign keys are not permitted to have `NULL` values. Conceptually, the schema would look like:

```
create table R (
        x       integer,
        y       char(1) not null,
        primary key (x),
        foreign key (y) references S(y)
);
create table S (
        y       char(1),
        x       integer not null,
        primary key (y),
        foreign key (x) references R(x)
);
```

but as we know from above, this won't load because of the mutual cross-references between the tables.

Describe how you would load the schema and then how you would insert the same two tuples as above under the new schema.

Answer:

The same technique as above can be used to load the schema:

```
create table R (
        x       integer,
        y       char(1) not null,
        primary key (x)
);
create table S (
        y       char(1),
        x       integer not null,
        primary key (y)
);
alter table R add foreign key (y) references S(y) deferrable;
alter table S add foreign key (x) references R(x) deferrable;
```

Note, however, that there's an extra keyword attached to the foreign key definitions: `DEFERRABLE`. Its use is explained below.

The problem with inserting the tuples is that PostgreSQL's default behaviour for SQL is to check constraints after every attempt insert or delete or update a tuple. What's needed here is to somehow delay the constraint checking until all of the desired tuples are in the database. If all the tuples are consistent and have valid foreign keys once they're all "added", then we can leave them in the database. Of course, if we discover that one of the tuples really does have an invalid foreign key, then we can't leave it there, and we have to delete the invalid tuple. This might make some of the other "added" tuples also invalid, so what PostgreSQL does is to remove *all* of the tuples that were temporarily added before the constraints were checked.

The question is, of course, how to get PostgreSQL to have the behaviour above: temporarily adding a set of tuples, then checking the constraints, and then either adding them all permanently, or removing them all. This behaviour is provided by transactions, which are introduced by the `BEGIN` statement and completed by the `COMMIT` statement. However, even within a transaction, constraints will be checked after each statement unless they are explicitly delayed via the `DEFERRED` keyword.

Note that constraint checking cannot be deferred unless the constraint itself was initially declared as *deferrable*.

This is how it all works in PostgreSQL:

```
begin;    -- start a transaction
set constraints all deferred;
```

```
insert into R values (1,'a');
insert into S values ('a',1);
commit;  -- finish the transaction
```