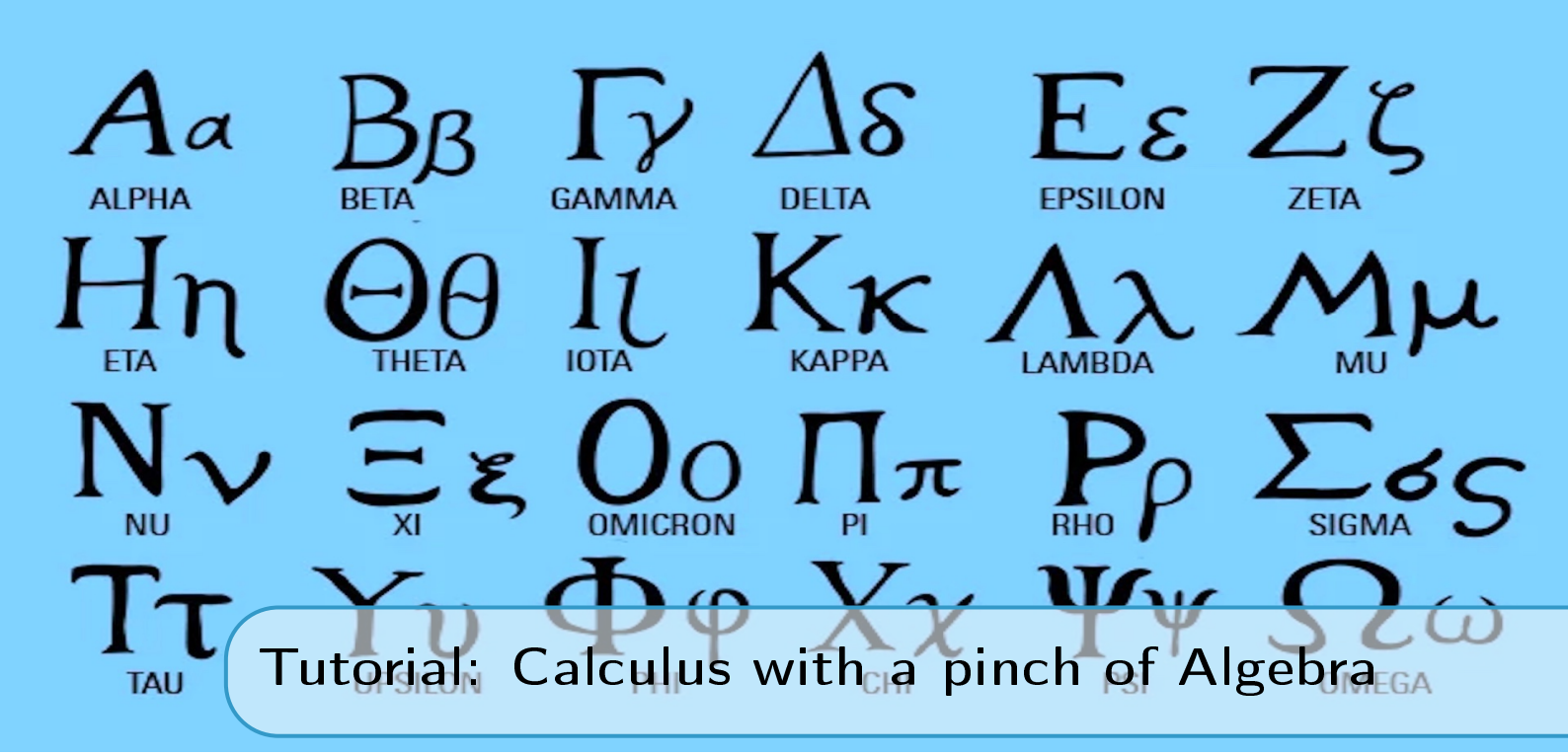


Tutorial: Calculus with a pinch of Algebra

This tutorial uses the schema of the database in Tutorial 1.

1. Translate the following queries into tuple relational calculus (TRC).
   (a) Find the different departments in School of Computing.

   Solution:
   $\{T \mid \exists T_1\ (T_1 \in department \wedge T_1.faculty = \text{'School of Computing'} \wedge T.department = T_1.department)\}$
   Although the following notations and combinations thereof exist. They are not used in this module. They are simpler, but may hide some difficulties (related to quantification) that we want to make explicit.
   $\{T \mid \exists T_1 \in department\ (T_1.faculty = \text{'School of Computing'} \wedge T.department = T_1.department)\}$
   $\{< T_1.department > \mid \exists T_1\ (T_1 \in department \wedge T_1.faculty = \text{'School of Computing'})\}$
   Note that Calculus (and algebra), as we study them, work on sets. Therefore they automatically eliminate duplicates.

   (b) Let us check the integrity of the data. Find the emails of the students who borrowed or lent a copy of a book before they joined the university. There should not be any.

   Solution: $\{T \mid \exists T_1\ \exists T_2\ (T_1 \in student \wedge T_2 \in loan \wedge ((T_1.email = T_2.borrower \wedge T_2.borrowed < T_1.year) \vee (T_1.email = T_2.owner \wedge T_2.borrowed < T_1.year)) \wedge T.email = T_1.email)\}$
   This can be rewritten as follows.
   $\{T \mid \exists T_1\ \exists T_2\ (T_1 \in student \wedge T_2 \in loan \wedge (T_1.email = T_2.borrower \vee T_1.email = T_2.owner) \wedge T_2.borrowed < T_1.year \wedge T.email = T_1.email)\}$
   We can accordingly write the answer in Tutorial 2 as either:

```
SELECT DISTINCT s.email
FROM loan l, student s
WHERE (s.email = l.borrower AND l.borrowed < s.year)
OR (s.email = l.owner AND l.borrowed < s.year);
```

```
SELECT DISTINCT s.email
FROM loan l, student s
WHERE (s.email = l.borrower OR s.email = l.owner)
AND l.borrowed < s.year;
```

   (c) Print the emails of the students who borrowed but did not lend a copy of a book on the day that they joined the university.

Solution: $\{T \mid \exists T_1\ \exists T_2\ (T_1 \in student \wedge T_2 \in loan\ \wedge\ T_1.email = T_2.borrower\ \wedge\ T_2.borrowed = T_1.year\ \wedge \neg(\exists T_3\ (T_3 \in loan\ \wedge\ T_1.email = T_3.owner\ \wedge\ T_3.borrowed = T_1.year))\ \wedge\ T.email = T_1.email)\}$

This translates into SQL as follows.

```
SELECT  DISTINCT s.email
FROM loan l2, student s1
WHERE s1.email = l2.borrower AND l2.borrowed = s1.year
AND NOT EXISTS(
    SELECT  *
    FROM loan l3
    WHERE s.email = l3.owner AND l3.borrowed = s1.year;
```

The query can also be written as follows in calculus.

$\{T \mid \exists T_1\ \exists T_2\ \forall T3\ (T_1 \in student \wedge T_2 \in loan\ \wedge\ T_1.email = T_2.borrower\ \wedge\ T_2.borrowed = T_1.year \wedge \neg(T_3 \in loan \wedge T_1.email = T_3.owner \wedge T_3.borrowed = T_1.year) \wedge T.email = T_1.email)\}$

$\{T \mid \exists T_1\ \exists T_2\ \forall T3\ (T_1 \in student \wedge T_2 \in loan\ \wedge\ T_1.email = T_2.borrower\ \wedge\ T_2.borrowed = T_1.year \wedge (T_3 \notin loan \vee T_1.email \neq T_3.owner \vee T_3.borrowed \neq T_1.year) \wedge T.email = T_1.email)\}$

(d) Find the ISBN13 of the books that have been borrowed by all the students in the computer science department.

Solution: $\{T \mid \exists T_1\ (T_1 \in book \wedge \forall T_2\ ((T_2 \in student \wedge T_2.department = \text{'Computer Science'}) \rightarrow (\exists T_3\ (T_3 \in loan\ \wedge\ T_2.email = T_3.borrower\ \wedge T_1.isbn13 = T_3.book))\ \wedge T.isbn13 = T_1.isbn13)\}$

Which can be rewritten as follows.

$\{T \mid \exists T_1\ (T_1 \in book \wedge \neg(\exists T_2\ (T_2 \in student \wedge T_2.department = \text{'Computer Science'} \wedge \neg(\exists T_3\ (T_3 \in loan\ \wedge\ T_2.email = T_3.borrower\ \wedge T_1.isbn13 = T_3.book))))\ \wedge T.isbn13 = T_1.isbn13)\}$

This translates into SQL as follows.

```
SELECT   b.isbn13
FROM book b
WHERE NOT EXISTS(
    SELECT  *
    FROM student s
    WHERE s.department = 'Computer Science'
        AND NOT EXISTS(
        SELECT  *
        FROM loan l
        WHERE s.email = l.borrower
        AND b.isbn13 = l.book));
```

2. Translate the following queries into relational algebra.

(a) Find the different departments in School of Computing.

Solution:

$\pi_{d.department}(\sigma_{d.faculty=\text{'School of Computing'}}(\rho(department, d)))$

(b) Let us check the integrity of the data. Find the emails of the students who borrowed or lent a copy of a book before they joined the university. There should not be any.

Solution: $\pi_{s.email}(\sigma_{(s.email=l.borrower\ \vee\ s.email=l.owner)\ \wedge\ l.borrowed<s.year}(\rho(student, s) \times \rho(loan, l)))$

This can also be written with a join.

$\pi_{s.email}(\rho(student, s) \bowtie_{(s.email=l.borrower\ \vee\ s.email=l.owner)\ \wedge\ l.borrowed<s.year} \rho(loan, l))$

or as

$\pi_{s_1.email}(\sigma_{s_1.email=l_1.borrower\ \wedge\ l_1.borrowed<s_1.year}(\rho(student, s_1) \times \rho(loan, l_1)))$

$\cup$

$\pi_{s_2.email}(\sigma_{s_2.email=l_2.owner\ \wedge\ l_2.borrowed<s_2.year}(\rho(student, s_2) \times \rho(loan, l_2)))$

We can accordingly write the answer in Tutorial 2 as either:

```
SELECT  s1.email
FROM loan l1, student s1
WHERE s1.email = l1.borrower
AND l1.borrowed < s1.year
UNION
SELECT  s2.email
FROM loan l2, student s2
WHERE s2.email = l2.owner
AND l2.borrowed < s2.year
```

(c) Print the emails of the students who borrowed but did not lend a copy of a book on the day that they joined the university.

Solution: We use non-symmetric set difference.

$\pi_{s_1.email}(\sigma_{s_1.email=l_1.borrower \wedge l_1.borrowed=s_1.year}(\rho(student, s_1) \times \rho(loan, l_1)))$

$\setminus$

$\pi_{s_2.email}(\sigma_{s_2.email=l_2.owner \wedge l_2.borrowed=s_2.year}(\rho(student, s_2) \times \rho(loan, l_2)))$

This is the query with `EXCEPT` in tutorial 2.

Non-symmetric set difference can be used to implement other universally quantified queries in algebra but these queries are quite complicated to write.

References

[1] S. Bressan and B. Catania. *Introduction to Database Systems.* McGraw-Hill Education, 2006.

[2] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems.* Pearson, 7th edition, 2015.

[3] R. Ramakrishnan and J. Gehrke. *Database Management Systems.* McGraw-Hill, 2002.

1. Consider the table `Exams(sid, cid, score)`, such that:

   - Each `sid` is an integer and represents a student ID.
   - Each `cid` is an integer and represents a course ID.
   - Each `score` is an integer and represents a final exam score of a student in a course.

   Write a function `max_min` with the following properties:

   - It has an input parameter `stu_id`, which is an integer.
   - It has two output parameters `max_cid` and `min_cid`, both of which are integers.
   - It examines the records in `Exams` whose `sid` is equal to `stu_id` and identifies the two records among them with the largest and smallest `score` where *ties are broken arbitrarily*. For the record with the largest `score`, its `cid` is assigned to `max_cid`. For the record with the smallest `score`, its `cid` is assigned to `min_cid` *only if it is smaller than the largest score*. Otherwise, `min_cid` is set to `NULL`.

   A template for `max_min` is provided below:

```
1 CREATE OR REPLACE FUNCTION max_min (IN stu_id INT, OUT max_cid
     INT, OUT min_cid INT)
2 RETURNS RECORD AS $$
3 DECLARE
4   max_score INT;
5   min_score INT;
6 BEGIN
7   /* write your code here */
8 END;
9 $$ LANGUAGE plpgsql;
```

Solution:

```
1  CREATE OR REPLACE FUNCTION max_min (IN stu_id INT, OUT
      max_cid INT, OUT min_cid INT)
2  RETURNS RECORD AS $$
3  DECLARE
4    max_score INT;
5    min_score INT;
6  BEGIN
7    SELECT cid, score INTO max_cid, max_score
8    FROM   Exams
9    WHERE  sid = stu_id
10     AND  score =
11           (SELECT MAX(score) FROM Exams WHERE sid = stu_id);
12
13   SELECT cid, score INTO min_cid, min_score
14   FROM   Exams
15   WHERE  sid = stu_id
16     AND  score =
17           (SELECT MIN(score) FROM Exams WHERE sid = stu_id);
18
```

```
19    IF max_score = min_score THEN
20      min_cid := NULL;
21    END IF;
22  END;
23  $$ LANGUAGE plpgsql;
```

2. Consider the same table `Exams(sid, cid, score)` from the previous question. Write a function `revised_avg` with the following properties:

   - It has an input parameter `stu_id`, which is an integer.
   - It has an output parameter `r_avg`, which is a numeric.
   - It examines the records in `Exams` whose `sid` is equal to `stu_id`. If there are *at least* 3 such records, the function returns the average `score` of these records but *excludes* one record with the highest `score` with *ties broker arbitrarily* as well as one record with the lowest `score` with *ties broker arbitrarily*. If there are fewer than 3 such records, the function returns `NULL`.

   A template for `revised_avg` is provided below:

```
1 CREATE OR REPLACE FUNCTION revised_avg (IN stu_id INT, OUT r_avg
      FLOAT)
2 RETURNS FLOAT AS $$
3   /* write your code here */
4 $$ LANGUAGE plpgsql;
```

Solution:

```
1  CREATE OR REPLACE FUNCTION revised_avg (IN stu_id INT, OUT
       r_avg FLOAT)
2  RETURNS FLOAT AS $$
3  DECLARE
4    max_score INT;
5    min_score INT;
6    sum_score FLOAT;
7    ctx_score INT;
8  BEGIN
9    SELECT MAX(score), MIN(score), SUM(score), COUNT(score)
       INTO
10         max_score , min_score , sum_score , ctx_score
11   FROM   Exams
12   WHERE  sid = stu_id;
13
14   IF ctx_score < 3 THEN
15     r_avg := NULL;
16   ELSE
17     r_avg := (sum_score - max_score - min_score)/(ctx_score
       -2);
18   END IF;
19 END;
20 $$ LANGUAGE plpgsql;
```

3. Consider the same table `Exams(sid, cid, score)` from the first question as well as the concept of "revised average `score`" in the previous question. Write a function `list_r_avg` that returns the `sid` of each student in `Exams` along with their revised average `score`. For simplicity, we assume that all `sid` in `Exams` are non-negative integers.

   A template for `list_r_avg` is provided below:

```
1  CREATE OR REPLACE FUNCTION list_r_avg ()
2  RETURNS TABLE (stu_id INT, ravg FLOAT) AS $$
3  DECLARE
4    curs CURSOR FOR (SELECT sid, score FROM Exams ORDER BY sid);
5    /* add other variables here */
6  BEGIN
7    /* write your code here */
8  END;
9  $$ LANGUAGE plpgsql;
```

Solution:

```
1  CREATE OR REPLACE FUNCTION list_r_avg ()
2  RETURNS TABLE (stu_id INT, ravg FLOAT) AS $$
3  DECLARE
4    curs CURSOR FOR (SELECT sid, score FROM Exams ORDER BY sid)
       ;
5    r RECORD;
6    max_score INT;
7    min_score INT;
8    sum_score FLOAT;
9    ctx_score INT;
10 BEGIN
11   stu_id = -1;
12   OPEN curs;
13   LOOP
14     FETCH curs INTO r;
15     IF r.sid <> stu_id OR NOT FOUND THEN
16       IF stu_id <> -1 THEN
17         IF (ctx_score < 3) THEN
18           ravg := NULL;
19         ELSE
20           ravg := (sum_score - max_score - min_score)/(
     ctx_score-2);
21         END IF;
22         RETURN NEXT;
23       END IF;
24       EXIT WHEN NOT FOUND;
25       stu_id := r.sid;
26       max_score := r.score;
27       min_score := r.score;
28       sum_score := r.score;
29       ctx_score := 1;
30     ELSE
31       sum_score := sum_score + r.score;
32       ctx_score := ctx_score + 1;
33       IF r.score > max_score THEN max_score := r.score; END
     IF;
```

```
34        IF r.score < min_score THEN min_score := r.score; END
      IF;
35       END IF;
36     END LOOP;
37     CLOSE curs;
38     RETURN;
39 END;
40 $$ LANGUAGE plpgsql;
```

4. Consider the same table `Exams(sid, cid, score)` from the first question. Write a function `list_scnd_highest` that returns the `sid` of each student in `Exams` along with their second highest `score` with *ties broken arbitrarily*. If the student only have one `score`, then the second highest `score` is `NULL`. For simplicity, we assume that all `sid` in `Exams` are non-negative integers.

   A template for `list_scnd_highest` is provided below:

```
CREATE OR REPLACE FUNCTION list_scnd_highest ()
RETURNS TABLE (stu_id INT, scnd_highest INT) AS $$
  /* write your code here */
$$ LANGUAGE plpgsql;
```

Solution:

```
CREATE OR REPLACE FUNCTION list_scnd_highest ()
RETURNS TABLE (stu_id INT, scnd_highest INT) AS $$
DECLARE
  curs CURSOR FOR (SELECT sid, score FROM Exams ORDER BY sid)
   ;
  r RECORD;
  max_score INT;
  ctx_score INT;
BEGIN
  stu_id = -1;
  OPEN curs;
  LOOP
    FETCH curs INTO r;
    IF r.sid <> stu_id OR NOT FOUND THEN
      IF stu_id <> -1 THEN
        IF (ctx_score < 2) THEN
          scnd_highest := NULL;
        END IF;
        RETURN NEXT;
      END IF;
      EXIT WHEN NOT FOUND;
      stu_id := r.sid;
      max_score := r.score;
      scnd_highest := -1;
      ctx_score := 1;
    ELSE
      ctx_score := ctx_score + 1;
      IF r.score > max_score THEN
        scnd_highest := max_score;
        max_score := r.score;
      ELSEIF r.score > scnd_highest THEN
        scnd_highest := r.score;
      END IF;
    END IF;
  END LOOP;
  CLOSE curs;
  RETURN;
END;
$$ LANGUAGE plpgsql;
```

This tutorial uses the relational schema in `T07.sql`.

1. Suppose that no employee can be both an engineer and a manager. Create two triggers to enforce this constraint on the `Engineers` and `Managers` tables. The triggers should run before insert or update and *prevent* changes (*i.e.*, no insertion and no update) when the condition is not met (*i.e.*, when an employee is about to be both engineer and manager).

Solution:

The solution given here is on `Managers`. The solution on `Engineers` is similar.

```
CREATE OR REPLACE FUNCTION not_manager()
RETURNS TRIGGER AS $$
DECLARE
  count NUMERIC;
BEGIN
  SELECT COUNT(*) INTO count
  FROM   Managers
  WHERE  NEW.eid = Managers.eid; /* Engineers.eid */

  IF count > 0 THEN
    RETURN NULL;
  ELSE
    RETURN NEW;
  END IF;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER non_manager
BEFORE INSERT OR UPDATE ON Engineers
FOR EACH ROW EXECUTE FUNCTION not_manager();
```

2. Suppose that we pay every engineers working on a project $100 per hour worked. Since every project has a budget, the total number of hours worked by every engineer multiplied by 100 cannot exceed the project budget. Create a trigger to enforce this constraint such that when an insert or update is performed on `Works` table that violates this constraint, the number of hours worked by the engineer is set to the maximum allowable for that project.

Solution:

```
CREATE OR REPLACE FUNCTION check_budget()
RETURNS TRIGGER AS $$
DECLARE
  hrs INTEGER;
  bgt INTEGER;
  rst INTEGER;
BEGIN
  SELECT COALESCE(SUM(hours), 0) INTO hrs
         /* COALESCE is used to handle NULL values */
  FROM   Works
  WHERE  pid = NEW.pid
    AND  eid <> NEW.eid;  /* for update */

  SELECT pbudget INTO bgt
  FROM   Projects
  WHERE  pid = NEW.pid;

  rst := (bgt - hrs*100)/100;
  IF NEW.hours > rst THEN
    RETURN (NEW.pid, NEW.eid, NEW.wid, rst);
  ELSE
    RETURN NEW;
  END IF;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER budget_check
BEFORE INSERT OR UPDATE ON Works
FOR EACH ROW EXECUTE FUNCTION check_budget();
```

3. As each work how has a type, we have an additional constraint that for a given work, the amount of time spent on the work cannot exceed the maximum hours for that particular type of work. Create a trigger to restrict `Works` table such that the hours worked cannot exceed the maximum hours for the given type. Whenever we want to insert or update such that the hours worked exceed the maximum hours, we set the hours worked to the maximum hours.

Solution:

```
CREATE OR REPLACE FUNCTION max_hour_work()
RETURNS TRIGGER AS $$
DECLARE
  maximal INTEGER; /* cannot be NUMERIC */
BEGIN
  SELECT max_hours INTO maximal
  FROM   WorkType
  WHERE  WorkType.wid = NEW.wid;

  IF NEW.hours > maximal THEN
    RETURN (NEW.pid, NEW.eid, NEW.wid, maximal);
  ELSE
    RETURN NEW;
  END IF;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER hours_max
BEFORE INSERT OR UPDATE ON Works
FOR EACH ROW EXECUTE FUNCTION max_hour_work();
```

4. Consider a case where we have a default work type. For simplicity, we let `wid = 0` to be the default work type. As this is the default, we can neither modify nor delete this work type. Create a trigger to prevent modification or deletion of the default work type. The trigger should raise notice that some users are trying to modify or delete this default work type. Furthermore, the trigger should not raise any notice when some users are trying to modify or delete other type of work.

Solution:

```
CREATE OR REPLACE FUNCTION default_work()
RETURNS TRIGGER AS $$
BEGIN
  RAISE NOTICE 'some user tried to';
  RAISE NOTICE 'modify/delete default';
  RAISE NOTICE 'work type';
  RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER work_default
BEFORE UPDATE OR DELETE ON WorkType
FOR EACH ROW WHEN (OLD.wid = 0) EXECUTE FUNCTION default_work
   ();
```

TUTORIAL for Lecture Week 10

INFORMAL DESIGN GUIDELINES AND FUNCTIONAL DEPENDENCIES

(Prof. Sham Navathe- Lecture on Oct 24, 2023)

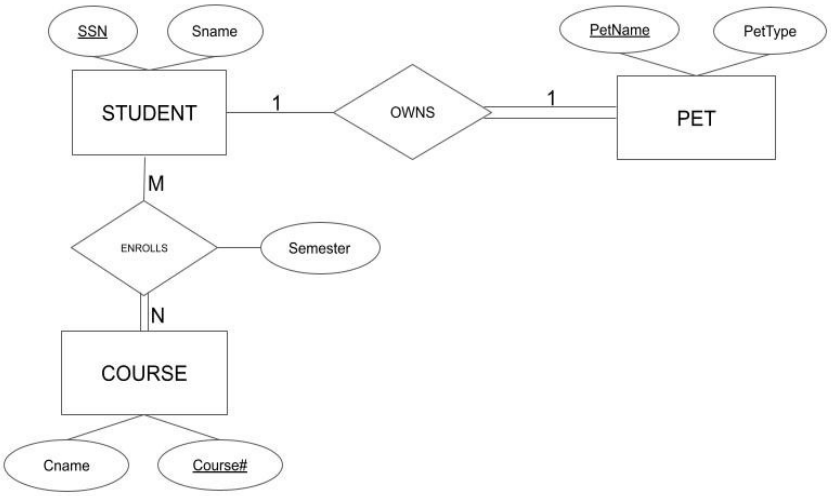
TUTORIAL+SOLUTIONS

This tutorial includes questions on three topics:

1) Informal Guidelines for Relational Design
2) Functional Dependencies, Closure and Equivalence
3) Minimum cover of FDs.

Q1. ABOUT INFORMAL DESIGN GUIDELINES:

CONSIDER THE FOLLOWING ER SCHEMA about students, courses they enroll in and the pets they have.



A. Suppose that only 30% of the students own a pet. The relational design for this schema was done by mapping into the following table:

STUDENT_PET (Student_Ssn, Sname, Petname, Pettype)

i) What informal guideline(s) is/are violated by this design?
ii) What type of problem would be caused in this table?
iii) Propose a correct design that will eliminate null values.
iv) List the functional dependencies present in the relation STUDENT_PET .
v) List the functional dependencies in your correct design. Can you say that functional dependencies are preserved in the correct design but the nulls are eliminated?
vi) If the OWNS relationship type was modified to be One-to-many , allowing a student to own multiple pets, would your design above change? Why or why not?

ANSWER:

1A.

(i) It violates the guideline that attributes from entities should not be mixed inappropriately and that null values should be avoided as much as possible

(ii) In this table 70% of the tuples will have null values under the columns Petname and Pettype

(iii) The correct design that eliminates the problem has two tables:
STUDENT (SSN, Sname)
PET (Petname, Pettype, Student_SSN)
In PET, Student_SSN is a foreign key.

(iv) The given relation has FDs:
{ *Student_SSN* $\rightarrow$ *Sname, Petname, Pettype, and*
Petname $\rightarrow$ *Pettype, Petname* $\rightarrow$ *Student_SSN* }

(v) Relation STUDENT has FD : *Student_SSN* $\rightarrow$ *Sname.*
Relation PET has FD: *Petname* $\rightarrow$ *Pettype, Student_SSN.*
Yes, the correct design maintains the FDs but eliminates the nulls.

(vi) Making the OWNS relationship type one-to-many does NOT change the above correct design. The foreign key Student_SSN in the PET relation takes care of the one-to-many relationship type.

++++++++++++++++++++++++++++

B. Suppose the ENROLLS relationship type was mapped into the following table:
ENROLLS (SSN, Course#, Student_name, Cname, Semester)

i) What informal guideline(s) are being violated here?

ii) Due to the violation of these guidelines, Insertion, deletion and update anomalies will be present. Give one example of each anomaly.

iii) Propose a correct design that will eliminate these anomalies.

iv) List the functional dependencies present in the relation ENROLLS.

v) List the functional dependencies in your correct design. Can you say that functional dependencies are preserved in the correct design but the anomalies are eliminated?

vi) If each student was allowed to take only one course per semester, would you still maintain your design? Give a proper explanation.

ANSWER:

1B.

(i) It violates the informal guideline that attributes from relationship type and entity types should not be mixed inappropriately. The table should be easy to explain as one that stands for an entity type or a relationship type.

(ii) Insertion anomaly: A student cannot be inserted unless he/she is enrolled in some course
Deletion anomaly: Deletion of a course may result in accidentally deleting student(s) that are enrolled only in that course.
Update anomaly: If a Course_name is changed it will cause updating each row for every student enrolled in that course.

(iii) The correct design to eliminate these anomalies:
STUDENT (Student_SSN, Sname)
COURSE (Course#, Cname)
ENROLLS_CORRECT (Student_SSN, Course#, Semester)

(iv) FDs present in ENROLLS:

{ *Student_SSN, Course#* $\rightarrow$ *Sname, Cname , Semester*

Student_SSN $\rightarrow$ *Sname,* *Course#* $\rightarrow$ *Cname}*

(v) *In the correct design:*

STUDENT has FD: *Student_SSN* $\rightarrow$ *Sname*

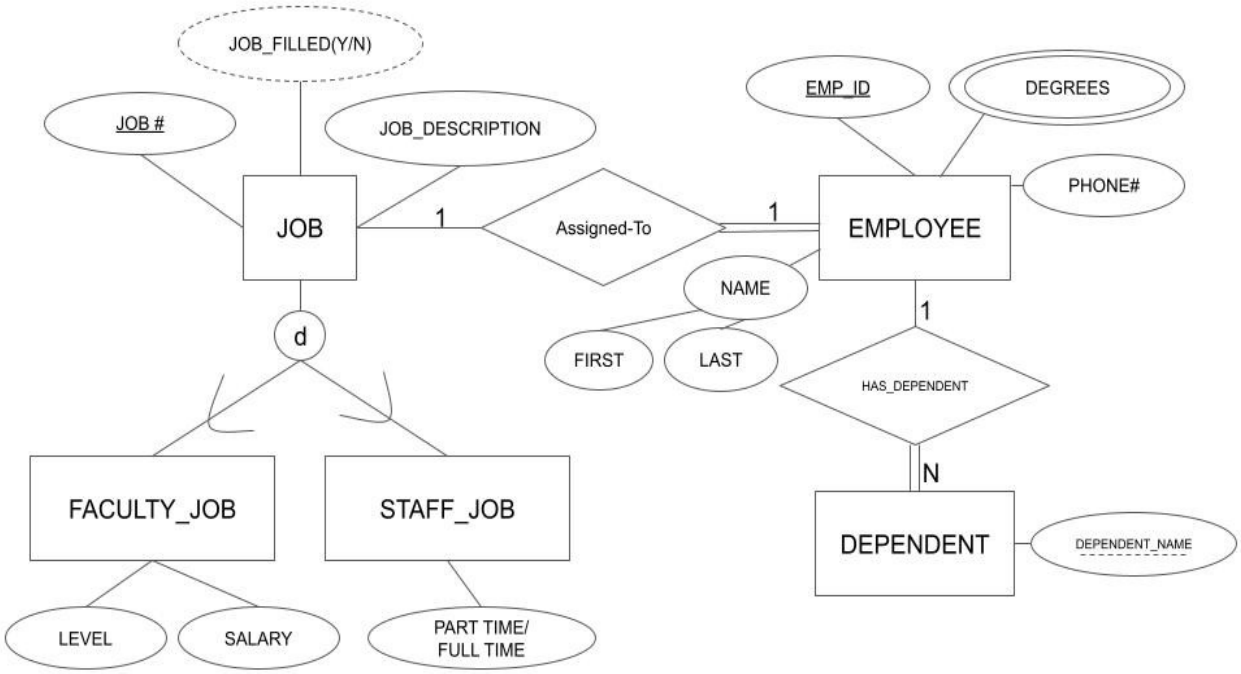
COURSE has FD: *Course#* $\rightarrow$ *Cname*

ENROLLS_CORRECT has FD: *Student_SSN, Course#* $\rightarrow$ *Semester*

By applying the Armstrong Axiom of augmentation, we see that the set of FDs in correct design are equivalent to the FDs in the poorly designed ENROLLS table; yet it avoids all the anomalies.

(vi) If the student is allowed to take only one course per semester, the relationship ENROLLS still remains many-to-many and the correct design still remains as shown above. The constraint acts as a semantic constraint on the table ENROLLS_CORRECT to make sure that for a given semester there are no two distinct Course#s for a given Student_SSN. That will need to be enforced externally by an application.
An alternate solution is to change the design to make it an all-key relation so that the new key includes Semester:
ENROLLS_CORRECT (Student_SSN, Semester, Course#)

C. Consider the EER schema below and answer the following questions:



C1. About relational design based on the JOB Superentity type

i) Show a possible design where the super entity type JOB and the sub-entity types FACULTY_JOB and STAFF_JOB are all accommodated in one table called JOBS.

ii) If only 10% of jobs are faculty jobs and 90% jobs are staff jobs, what potential problem exists?

iii) Propose a design where null values will not occur and jobs will have a discriminating attribute that defines it as a faculty or staff job.

ANSWER:

C1.

(i) Table design using one table:
JOBS (Job#, Job_Descript, Faculty_level, Faculty_salary, Staff_full_parttime, Job_type)

(ii) The attributes < Faculty_level, Faculty_salary> will have null values for 90% rows and the attribute Staff_full_parttime will have null values for 10% rows.

(iii) The correct design will have 3 tables:
JOB (Job#, Job_Descript, Job_type)
FACULTY_JOB (Job#, Fac_level, Fac_salary)
STAFF_JOB (Job#, Staff_full_parttime)

C2. About the ASSIGNED_To relationship type:

Suppose there are 1000 Jobs in the populated database and a table was stored as:
JOB_ASSIGNMENT (Job#, Job_Description, Emp_id, Degree, Phone#)
80% of jobs have been filled and 50% of employees have two degrees.

i) Assume that this table has been populated with the 1000 jobs announced including those that have been filled. What is the nature of redundancy in the above design?

ii) How many tuples will get stored in the JOB_ASSIGNMENT table?

iii) How many null values will get stored?

iv) How many duplicate values will get stored?

v) Assuming that the JOB table is stored as follows:
JOBS (Job#, Job_description, Job_type).
Propose a proper design that takes care of the Assigned-to relationship type and the multivalued attribute Degrees correctly and point out the estimated number of tuples in each table.

vi) List the functional dependencies in your correct design. Can you say that functional dependencies among the attributes in the original EER schema are preserved in the correct design?

ANSWER:

C2.

(i) The redundancy in this table arises from the fact that the multivalued attribute Degrees has been accommodated as a column in this table causing Employee and Job information to be repeated for the Employees with multiple degrees. A large number of null values will be the result.

(ii) JOB_ASSIGNMENT table will have 1000 tuples for the 1000 jobs and 400 extra tuples for 50% of the employees who have 2 degrees. Thus 1400 total tuples.

(iii) 200 unfilled jobs result in 400 null values under the two columns Emp_id and Degree.

(iv) 400 tuples for employees with 2 degrees will have duplicate values stored under first 3 columns of the JOB_ASSIGNMENT table giving 1200 duplicate values.

(v) A proper design of relations to account for EMPLOYEE entity type and ASSIGNED_TO relationship type is:
EMPLOYEE (Emp_id, Phone#, Job#) – 800 tuples; Job# foreign key takes care of the Assigned-to relationship type.
EMP_DEGREES (Emp_id, Degree) – 1200 tuples; 800 employees have 800 tuples; 400 with 2 degrees have 400 additional tuples.

(vi) The EMPLOYEE table has FD: $Emp\_id \rightarrow Phone\#, Job\#$
EMP_DEGREES table has NO FD.
JOBS table has FDs: $Job\# \rightarrow Phone\#, Job\_Description, Job\_type$
Yes, the FDs among the attributes of the EER schema are preserved.
An additional FD : $Job\# \rightarrow Job\_type$ has been added in this design to take care of the classification of Jobs.

++

Q2. ABOUT CLOSURES of attributes and equivalence of sets of FDs :

A. 1. Consider the relation: $R(A, B, C, D)$
with $F = \{A \rightarrow BCD, C \rightarrow D\}$

What are the closures of attributes A, B, C, D?

2. Consider the set of FDs :

$G = \{ A \rightarrow BC, C \rightarrow D\}$.

Compute the closures of attributes A, B, C, D in the set G.

Are F and G equivalent?

3. Consider the set of FDs:

$H = \{ A \rightarrow BD , A \rightarrow C\}$

Compute the closures of attributes A, B, C, D in the set H.

Are F and H equivalent?

ANSWER:

1. Closures of attributes in F:

- $\{A\}^+ = \{ABCD\}$
- $\{B\}^+ = \{B\}$
- $\{C\}^+ = \{CD\}$
- $\{D\}^+ = \{D\}$

2. Closures of attributes in G:

- Stepwise: $\{A\}^+ = \{ABC\}$
- But given that $C \rightarrow D$, $\{A\}^+ = \{ABCD\}$
- $\{B\}^+ = \{B\}$
- $\{C\}^+ = \{CD\}$
- $\{D\}^+ = \{D\}$

Thus the closures of attributes A, B, C, D in F and G are identical.

F can be derived from G and G can be derived from F.

Thus F and G are equivalent.

3. Closures of attributes in H:

- The Union of $A \rightarrow BD$ and $A \rightarrow C$ gives $\{A\}^+ = \{ABCD\}$
- But $\{B\}^+ = \{B\}$
- $\{C\}^+ = \{C\}$
- $\{D\}^+ = \{D\}$

The closure of C is different in F and H.

The FD , $C \rightarrow D$ in F cannot be derived from H.

Hence F and H are not equivalent.

++

Q3. About Minimum Cover of FDs

Find the minimum cover for the set G of f.d.s:

G: **{ A → C, AC → D, AD→ B, CD → EF, E→F }**

Show your work step by step and explain how you arrive at the final min cover.

ANSWER:

Step 1: Decomposing so that every FD has RHS with only one attribute:

CD → EF can be expressed as { CD → E, CD → F } .

Step2: Extraneous attribute removal

(i) From A→ C, we get AA→ AC by augmentation, which means A→AC ;

Now, A→AC and AC → D, Hence, by transitivity, we conclude A→ D.

Thus the **extraneous attribute C in AC → D is not needed.**

Hence LHS of AC →D can be reduced to just A. Hence A→ D.

So the set becomes:

G' : **{ A → C, A → D, AD→ B, CD → E, CD →F, E→F }**

(ii) From A→ D and AD →B, we find A→ B; hence LHS of AD →B can be reduced to just A with D being extraneous. Hence A→ B.

Thus, the **extraneous attribute D in AD → B is not needed.**

Hence we now get:

G'' : **{ A → C, A → D, A→ B, CD → E, CD →F, E→F }**

Step3: Removal of redundant FDs.
Because CD → E and E→F, by transitivity, CD →F.
Hence, **CD →F can be eliminated**. Then we get:

G''' : **{ A → C, A → D, A→ B, CD → E, E→F }**

No further reduction is possible of any FDs from the above set. Hence, it is the minimum cover.

G'''' = G_{min}

REGROUPING STEP:
Taking the union of first three FDs and next two FDs, we can write:

G_{min} : **{ A → BCD, CD → E, E→F }**

This is the minimum cover of the given dependencies.

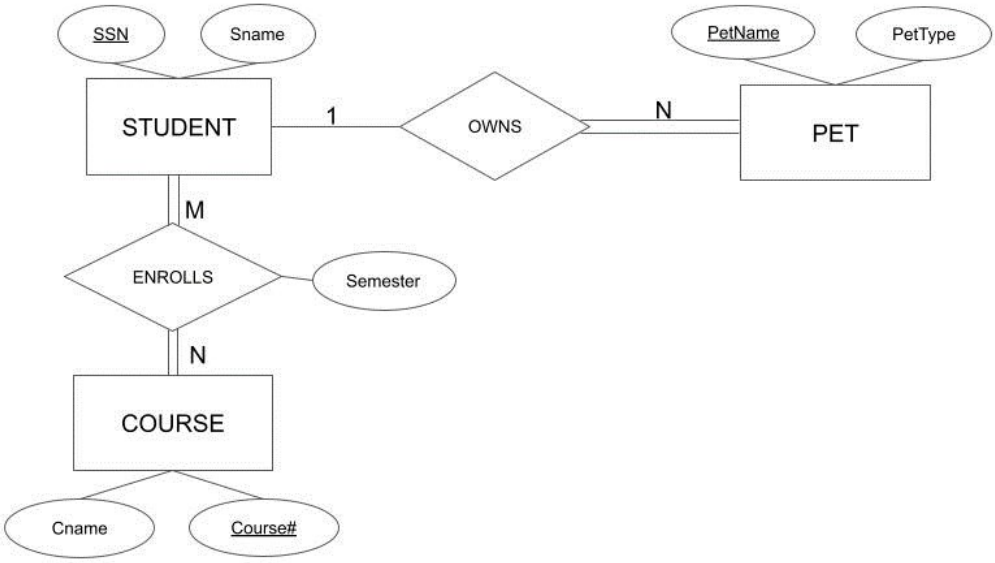
++++++++++++++++++++ +++++++++++++++ **END** ++++++++++++++++++++++++++++++++++++

TUTORIAL for Lecture Week 11 (QUESTIONS AND ANSWERS)

NOMALIZATION in Relational Database Design

(Prof. Sham Navathe- Lecture on Oct 31, 2023)

Q1. CONSIDER THE FOLOWING ER SCHEMA about students, courses they enroll in and the pets they have.



A. Consider a table created from this schema :
STUDENT_PET (Student_Ssn, Sname, Petname, Pettype)
Suppose 70% students own a pet and we are tolerating 30% nulls in the last two columns for students that do not own a pet.

i) What are the functional dependencies in this table?
ii) Is this a good design? What is/are the keys of this table?.
iii) Can Petname be considered a candidate key ?
iv) What normal form is this table in? Do you approve of this design?
v) Consider decomposing it into
STUDENT1 (Student_Ssn, Sname)
PET (Petname, Pettype, Student_SSN)
What FDs are present? Does this design meet BCNF? Is this a better design?

SOLUTION:

Q1 A.

(i) FDs: Student_ssn → Sname; Petname → Pettype

(ii) The relation is poorly designed and has no keys. Student_ssn repeats if a student has multiple Pets. For some students, the Petname is Null. Hence (Student_ssn, Petname) cannot be considered a key.

(iii) Petname cannot be considered a candidate key either because it has a null value for 30% of students. Relational model disallows a key to have null value.
NOTE: Postgres will allow the statement UNIQUE (Petname) because whatever non-null values of Petname exist, they will be unique. However, that still does NOT make it a candidate key.

(iv) The table is in a bad state. It is not even in 1NF.

(v) The new design with STUDENT1 and PET has FDs: Student_ssn → Sname and Petname → Pettype, Student_ssn. It meets the requirements of BCNF and is a better design.

B. Suppose the ENROLLS relationship type was mapped into the following table with primary key (SSN, Course#)

ENROLLS (SSN, Course#, Student_name, Cname, Semester)

i) What are the FDs in this table?

ii) What normal form is it in? Why?

iii) Do successive decomposition of this table.

iv) Test and justify that the result is in BCNF

v) Now suppose we introduce an FD Semester → Course#. Does the design remain in BCNF? If not, convert it into BCNF.

SOLUTION

Q1B.

(i) ENROLLS has the following FDs: SSN → Student_name; Course# → Cname; (SSN, Course#) → Semester

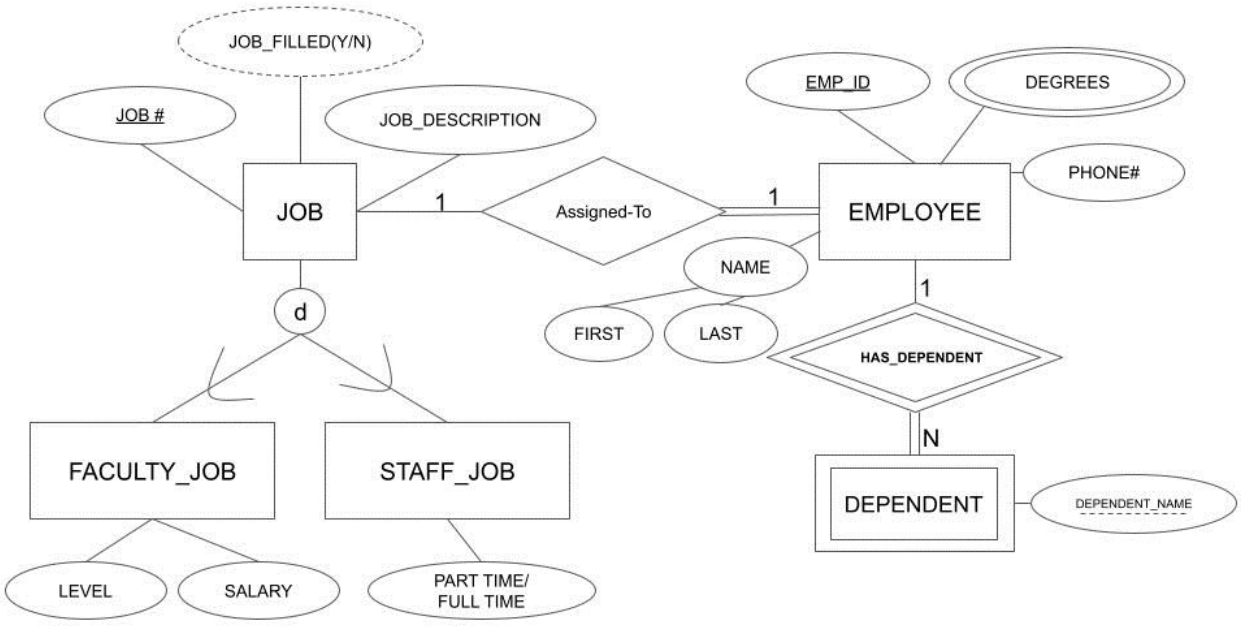
(ii) It is in 1NF. Attributes Student_name and Cname are NOT fully functionally dependent on the entire key: SSN, Course#. Hence, 2NF is violated.

(iii) Second Normalization:
ENROLLS1 (SSN, Course#, Semester)
STUDENT (SSN , Student_name)
COURSE (Course# , Cname)

(iv) Yes, the result is in BCNF because in each relation the LHS of every FD is a superkey.

(v) With the new FD Semester → Course#, ENROLLS1 fails BCNF because the LHS Semester is not a superkey. It needs to be decomposed.
Using the result from Algorithm 15.5, if X → Y violates BCNF in a relation R, we replace it by two relations: R1 (R-Y) and R2 (XUY). By applying this, we convert ENROLLS into:
ENROLLS1 (SSN, Semester)
ENROLLS2 (Semester, Course#). This decomposition meets the NJB property and is the correct decomposition.

Q2. Consider the EER schema below about Employees and Dependents and answer the following questions:



2A. Suppose the designer designed the following table for Employees:

EMPLOYEE (Empid, Phone#, Degrees)

i) What are the FDs in this table?

ii) What normal form is it in?

iii) Show two possible first normalizations of this table – one that may have redundancy and one that will avoid any redundancy. Show the FDs in each case.

SOLUTION:

Q2A.

(i) FDs in the table EMPLOYEE: Empid → Phone#.

(ii) EMPLOYEE is not even in 1 NF

(iii) First Normalizations of table EMPLOYEE:

ONE: EMP_REPEAT (Empid, Degree, Phone#). FDs present are:

Empid, Degree → Phone#; Empid → Phone#.

This design redundantly stores Empid when there are multiple degrees for that employee.

TWO: EMP_PHONE (Empid, Phone#), **EMP_DEGREE** (Empid, Degree)

The table EMP_PHONE has the FD: Empid → Phone#.

The table EMP_DEGREE is an all key table and has NO FDs.

This is the preferred design with no redundancy.

2B. Suppose the designer proposed the design of the Employee and Dependent information as a nested relation:

EMP_NESTED (Empid, Phone#, (Degree), (Dependent_name))

i) What normal form is this design in?

ii) Suppose the following table EMP_FLAT was presented to you as a relational design:

Empid	Phone#	Degree	Dependent_name
101	6146	BS	Peter
101	6146	MS	Mary
102	6234	Mtech	Jill
102	6234	PhD	Janet
102	6234	<null>	John

What are the problems with this design?

iii) How will you convert this to a proper design?

iv) Justify that your final design meets BCNF.

SOLUTION:

Q2B.

(i) EMP_NESTED is not even in 1NF. The key Emp# does NOT functionally determine the nested multi-valued attributes.

(ii) The table EMP_FLAT has the problem that it has mixed two multi-valued attributes in the same table creating a wrong association between the attributes Degree and Dependent. The pairing <BS, Peter> and <MS, Mary> creates a non-existent relationship between degree BS and dependent Peter and degree MS and dependent Mary for Employee 101. It creates FDs
Empid, Degree → Dependent_name and
Empid, Dependent_name → Degree
Both of which are meaningless.

(iii) The correct design is to decompose into 3 tables:
EMP1 (Empid, Phone)
EMP2 (Empid, Degree) and
EMP3 (Empid, Dependent_name)
The above design has only one FD: Empid → Phone#. EMP2 and EMP3 have NO FD.
This design meets BCNF.

Q3. SUCCESSIVE NORMALIZATION

Consider the following relation:

REFRIG_MODELS (Model#, Option_type, Option_Listtprice, Model_color, Discount%, Quantity, Sticker_price)

It refers to an inventory of Refrigerators at an appliance store. It includes options available on refrigerators (e.g. – ice maker, soda-water maker) that are sold and the list-prices and discounted prices etc.

The known f.d.s:

FD1:Model# → Model_color;

FD2:Option_type→ Option_Listprice;

FD3:Model#, Option_type → Quantity, Sticker_price;

FD4: Model_color → Discount%

Argue **using the generalized definition of the BCNF** that this relation is not in BCNF; further argue why it is not even in 3NF by the generalized definition of 3NF. Finally point out which dependency (or dependencies) violate the 2NF. Carry out successive normalization until you reach a BCNF design.

SOLUTION:

Q3. Successive Normalization

The key for this relation is (Model#,Option_type).
The Fds Fd1, FD2, FD4 all have LHS that is NOT a superkey.
Hence this relation is not in BCNF.
Applying the generalized definition of 3NF also the above 3 FDs violate the 3NF definition; hence it is not in 3NF.
The Fds FD1 and FD2 violate the 2NF because Model_color and Option_Listprice are not fully functionally dependent on the key (Model#,Option_type).
Hence the successive normalization proceeds as follows:
2nd Normalization:
MODEL (Model#, Model_color, Discount%)
OPTION (Option_type, Option_listprice)
REFRIG1(Model#, Option_type, Quantity, Sticker_price)
Now, the MODEL relation has a transitive dependency of Discount% via Model_color).
Hence it undergoes third normalization:
MODEL1 (Model#, Model_color)
MODEL2 (Model_color, Discount%)
The final design includes the tables: REFRIG1, OPTION, MODEL1, MODEL2.
All of them meet the BCNF property and hence the design is in BCNF.

Q4. NON-ADDITIVE BINARY DECOMPOSITION

Consider the following set of functional dependencies F in the STUDENT relation (it has obvious meaning of data related to students who enrolled in courses and received a grade):

STUDENT (**Stud_SSN, Course_no, Grade, Course_name, Stud_name, Course_Instr)**

F: { FD1: (Stud_SSN, Course_no) → Grade, Course_name, Stud_name, Course_Instr;

FD2: Stud_SSN → Stud_name;

FD3: Course_no → Course_name, Course_Instr }

The designer proposed to decompose this relation into

STUDENT1 (Stud_SSN, Course_no, Stud_name, Grade) and

COURSE (Course_no, Course_name, Course_Instr)

i) Apply the Non-additive join (NJB) test to determine if this decomposition is lossless
ii) What are the highest normal forms the above relations are in?
iii) Normalize the design further if possible.

SOLUTION:

Q4. Non-additive Join Binary Decomposition

- **PROPERTY NJB (non-additive join test for binary decompositions):**
  A decomposition D = {R1, R2} of R has the lossless join property with respect to a set of functional dependencies F on R *if and only if* either
  - The f.d. ((R1 ∩ R2) → (R1- R2)) is in F^+, or
  - The f.d. ((R1 ∩ R2) → (R2 - R1)) is in F

(i) Applying the NJB property to the given two relations:
STUDENT1 ∩ COURSE = { Course#}
COURSE – STUDENT1 = { Course_name, Course_Instr}.
Since { Course#} → { Course_name, Course_Instr}.
The given decomposition is lossless (non-additive).
(ii) STUDENT1 is in 1NF because Student_ssn → Student_name is a 2NF violation.
and COURSE is in BCNF.
(iii) STUDENT1 should be decomposed as : STUDENT2 (Student_ssn ,Student_name)
and ENROLL ((Stud_SSN, Course_no, Grade)
Hence the final decomposition of the given relation STUDENT as a BCNF design is the following set of relations:

COURSE (Course_no, Course_name, Course_Instr)

STUDENT2 (Student_ssn ,Student_name)

ENROLL ((Stud_SSN, Course_no, Grade)

++++++++++++++++++ +++++++++++++++ **END** ++++++++++++++++++++++++++++++++++

TUTORIAL FOR WEEK 12 (Prof. Navathe)

RELATIONAL DESIGN ALGORITHMS

Q1. Binary Decomposition with Non-additive decomposition

Consider the relation
SUPPLY (Supplier#, Part#, Date, Project, Quantity, Supp_name, Part_name)
The FDs are:
Fd1: Supplier#, Part#, Date → SUPPLY
Fd2: Supplier# → Supp_name
Fd3: Part# → Part_name.
If we decompose SUPPLY into:
A. SUPPLY1 (Supplier#, Part#, Date, Project, Quantity, Part_name)
SUPPLIER (Supplier#, Supp_name)
- Have we preserved the Fds?
- Is this decomposition non-additive (lossless) – why?
- What NF is SUPPLY1 in?

B. Show a further decomposition of SUPPLY 1 and show that the decomposition is non-additive and achieves BCNF.

SOLUTION:
A.
(i) Yes, all of Fd1 , Fd2 and Fd3 are preserved in the design.
(ii) To test for non-additive decomposition, we apply the NJB test.
R1 = SUPPLY1; R2 = SUPPLIER.
R1 ∩ R2 = Supplier#
R2 – R1 = Supp_name
Because R1 ∩ R2 → (R2 – R1), we conclude that the decomposition is non-additive.
(iii) SUPPLY1 is still only in 1NF because of the Fd3 present in it.

B.

Further 2nd normalization of SUPPLY1 produces:
SUPPLY11 (Supplier#, Part#, Date, Project, Quantity) which preserves Fd1

PART (Part#, Part_name).

Now, R1 = SUPPLY11 and R2 = PART
R1 ∩ R2 = Part#
R2 – R1 = Part_name
Because R1 ∩ R2 → (R2 – R1), we conclude that the decomposition is non-additive.
Hence the **final design contains SUPPLIER, PART and SUPPLY11** where successive decompositions were non-additive.
Hence the final set of relations also posseses the non-additive join

property. Since all of them contain only FDs with LHS as the key, the design is in BCNF.

Q2. Relational Synthesis into 3NF relations.

Assume that we are given a universal relation corresponding to the data we have on students and courses which looks like:

STUDENT_COURSE (Stud#, Course#, St_name, Course_name, Course_credit_hr, Grade, Major_dept, Dept_phone_no)

The given set of F.d.s for the universal relation are:

F: {Stud#, Course# → St_name, Course_name, Course_credit_hr, Grade, Major_dept, Dept_phone_no;

Stud# → St_name, Major_dept, Dept_phone_no;

Course# → Course_name, Course_credit_hr

Major_dept → Dept_phone_no.}.

A. Apply the synthesis algorithm 15.4 that constructs 3NF relations from a given set of F.d.'s, on the universal relation. What 3NF design will be produced by this algorithm? Show how you get the answer
B. Evaluate your 3NF design and evaluate if it meets BCNF design.

SOLUTION:

PART A:

Given F:

{FD1: Stud#, Course# → St_name, Course_name, Course_credit_hr, Grade, Major_dept, Dept_phone_no;

FD2: Stud# → St_name, Major_dept, Dept_phone_no;

FD3: Course# → Course_name, Course_credit_hr

FD4: Major_dept → Dept_phone_no.},

Computing Min. Cover of F:

Note: We are skipping the second step of algo. 15.2 where each FD is decomposed to have only one attribute on RHS.

The minimal cover is:

F_{min} = {Stud# → St_name, Major_dept,

Course# → Course_name, Course_credit_hr,

Major_dept → Dept_phone_no,

Stud#, Course# → Grade }

Why? Because,

(i) The part of FD1 based on the key (Stud#, Course#) on LHS **has extraneous attribute Course#** in order to functionally determine Stud#, Course# . Hence it is identical to FD2 and can be eliminated.

(ii) The part of FD1 based on the key (Stud#, Course#) on LHS **has extraneous attribute Stud#** in order to functionally determine Course_name, Course_credit_hr . Hence it is identical to FD3 and can be eliminated.

(iii) After removing these attributes on the RHS of FD1, it reduces to
Stud#, Course# → Grade.
That results in the above min. cover.

NOTE Also that the min cover in standard form contains FDs as follows:

Stud# → St_name,

Stud# → Major_dept,

Course# → Course_name,

Course# → Course_credit_hr.

We **REGROUP them into**

Stud# → St_name, Major_dept

And

Course# → Course_name, Course_credit_hr

Before going to form the 3NF design in step 3 of the .

Once, the above min.cover is obtained, the second step in algorithm 15.4 constructs one relation per FD in the min. cover, giving:

R1(Stud#, St_name, Major_dept)

R2 (Course#, Course_name, Course_credit_hr)

R3 (Major_dept , Dept_phone_no)

R4(Stud#, Course# , Grade)

PART B: Each of R1, R2, R3 and R4 contains an FD where the LHS is the key for that relation. There are no other known FDs. Hence, it meets the requirement of BCNF and hence the design is in BCNF.

Q3. BCNF Decomposition and n-ary non-additive decomposition test:

Consider a relation:

PATIENT_PROC (Patient#, Doctor#, Date, Doctor_name, Doctor_specialty, Procedure, Charge)

The Fds are:
FD1:Patient#, Doctor#, Date → PATIENT_PROC
FD2:Doctor# → Doctor_name, Doctor_specialty
FD3:Procedure → Doctor

A. Successive Normalization

(i) Evaluate and explain the normal form status of the given relation.

(ii) Follow the practice of successive normalization upto BCNF. For converting 3Nf to BCNF, apply the decomposition as per the decomposition in algorithm 15.5.

B. Direct testing and application of Algo 15.5

(i) Argue using the BCNF general definition that PATIENT_PROC does not meet BCNF ‘

(iii) . By applying decomposition in algorithm 15.5 successively produce the BCNF design.

C. Applying n-ary non-additive decomposition test in Algo 15.3

Show that the resulting relations you produced as BCNF design in A meet the non-additive join property using this algorithm.

SOLUTION:

(i) The FD2 shows non-full functional dependencies of the attributes Doctor_name and Doctor_specialty on Doctor# and hence is only in 1 NF.

(ii) Successive decomposition:

Second normalization:

PP1 (Patient#, Doctor#, Date, Procedure, Charge)

DOCTOR (Doctor#, Doctor_name, Doctor_specialty)

Third Normalization:

PP1 and DOCTOR are in 3NF .

PP1 is in 3 NF because it has 2 Fds:

FD11: Patient#, Doctor#, Date → Procedure, Charge

FD12: Procedure → Doctor#

FD11 meets the condition that LHS is superkey

FD12 meets the condition that RHS is prime attribute.

BCNF Normalization

However, FD12 violates BCNF.

Hence apply Algo 15.5 to PP1 and decompose it into:

PP11(Patient#, Date, Procedure, Charge) and

PP12 (Procedure, Doctor#).

Hence the final design is: DOCTOR, PP11 and PP12 tables.

NOTE: The dependency FD1 which is primary-key based FD **has been lost**.

B.

FIRST ATTEMPT (CASE X):

By Applying the BCNF definition directly, we see that PATIENT_PROC is not in BCNF because of
FD2 and FD3 which do not meet the requirement of LHS being a superkey. Suppose we fix the FD2 first. Then the first decomposition will produce:

PX1 (Patient#, Doctor#, Date, Procedure, Charge)
DOCTOR (Doctor#, Doctor_name, Doctor_specialty)

Now PX1 has the FDX1: Procedure → Doctor where LHS is not a superkey.
Hence we apply 15.5 and decompose into:
PX11 (Patient#, Date, Procedure, Charge) and
PX12 (Procedure, Doctor#)

Thus the final design is DOCTOR, PX11, PX12 which is identical with A.

SECOND ATTEMPT (CASE Y):
Suppose we fix FD3 first. Then the first decomposition produces:
PY1 (Patient#, Date, Doctor_name, Doctor_specialty, Procedure, Charge) and
PY2 ((Procedure, Doctor#)

PY1 now has the FD:
FY11: Patient#, Date, Procedure → Doctor_name, Doctor_specialty, Charge
And
FY12: Procedure → Doctor_name, Doctor_specialty.
Now, FY11 has an LHS that is a superkey;
However, FY12 does not meet BCNF. Hence we apply 15.5 to decompose PY1 as:
PY11(Patient#, Date, Procedure, Charge) and
PY12 (Procedure, Doctor_name, Doctor_specialty).
Thus the final BCNF design contains:

PY2, PY11,PY12. Notice that it is almost identical to the first case X except the Doctor# from first design X is replaced by Procedure in PY12 in the second design Y.

C.
TESTING FOR non-additive decomposition.

Set up the initial tableau for the 3 decomposed relations as 3 rows and one column for each attribute. A's represent columns that are present in the relation. B's represent columns that are absent in the relation.

The goal is to see if we can get any row of the tableau to have all a's after the algorithm is applied. That proves that the n-ary decomposition is non-additive.

(iv)	P#	Doc#	Date	Dname	Doctor_	Procedu	Charge

					Specialty	re	
R1	b	a2	b	a4	a5	b	b
R2	a1	b	a 3	a4	b	a 6	a 7
R3	b	a2	b	b	b	a 6	b

R1: DOCTOR (Doctor#, Doctor_name, Doctor_specialty)

R2: PP11(Patient#, Date, Procedure, Charge)

R3: PP12 (Procedure, Doctor).

FIRST STEP: Because Procedure → Doctor and a6 → a2 is present in row3,
We can set Doc# column from b to a in R2

(v)

	P#	Doc#	Date	Dname	Doctor_ Specialty	Procedure	Charge
R1	b	a2	b	a4	a5	b	b
R2	a1	~~b~~ a2	a 3	b	b	a 6	a 7
R3	b	a2	b	b	b	a 6	b

SECOND STEP: Because Doctor# → Doctor_name, Doctor_specialty
and a2 → a4,a5 is present in row1, and now we have a2 present in row2,
we can set Doc# column from b to a in R2

(vi)

	P#	Doc#	Date	Dname	Doctor_ Specialty	Procedure	Charge
R1	b	a2	b	a4	a5	b	b
R2	a1	~~b~~ a2	a 3	~~b~~ a4	~~b~~ a 5	a 6	a 7
R3	b	a2	b	b	b	a 6	b

Now, entire row 2 for R2 has been set to a's. That concludes the procedure and determines that the R1, R2, R3 decomposition has the non-additive join property.

++++++++++++++++ END OF TUTORIAL 12 WITH SOLUTIONS +++++++++
++++++++++++++++++++