**NYU Tandon School of Engineering** **April 21, 2017**
**Computer Science and Engineering**
**CS6083, Spring 2017**

# Problem Set #3

**Problem 1**
N/A

**Problem 2**
a.
All candidate keys: {D, E, F}
b.
Because of F → B, B is extraneous in FB → C.
Because of D → A, F is extraneous in DF → A.
So the canonical cover will be:
Fc = { D→A, E→AG, F→BCG}

c.
It is not in BCNF because there is nontrivial functional dependency: F → C and F is not the superkey. Decomposing relation R into BCNF we get:
Step 1 : R1 = {A, D} and R2 = {B, C, D, E, F, G}
Step 2 : R2 can be further decomposed into R2 and R3 because F→BCG is nontrivial functional dependency and D is not a superkey.
Hence BCNF form is:
R1 = {A, D}
R2 = {B, C, F, G}
R3 = {D, E, F}

d.
No. The BCNF form in c is not dependency preserving because we cannot check E → AG in the result of (c).
Converting into 3NF we get:
Fc = { D→A, E→AG, F→BCG} and Candidate key = {D, E, F}
Therefore, 3NF form is:
R1 = {A, D}
R2 = {A, E, G}
R3 = {B, C, F, G} and
R4 = {D, E, F}

**Problem 3:**

a.
This is not a good design for following reasons:

- Current schema does not reflect a lot of functional dependencies. For instance, as per assumptions in the question, *{cakeid →cakename, cakeprice, address, city, zip },* but since the {*cakeid}* is not candidate key, we will end up repeating this information in several tuples.
- Inserting values for the separate entities in one table like Customer or Cake will lead to storing NULL and duplicate values at several places. This will increase the storage and our data will be inconsistent.
- We have to access this one table for querying even minimal data, this will be time-consuming as everything is under this table.
- Lastly, this schema does not adhere to the normalization rules. As everything is under one table, it will make maintenance and querying from the database difficult and it will be hard to see relation between each attribute. The relational database system works better with several small tables than a big table.

b.
Candidate key: {custid, cakeid, ingredid, orderdate}

c. Following are all non-trivial functional dependencies:

{custid} → {custname, cphone, address, city, zip}
{cakeid} → {cakename, cakeprice, slices, status}
{ingredid} → {iname, iprice}
{cakeid, ingredid} → {qty}
{custid, cakeid, orderdate} → {pricepaid, pickupdate}

d.
The canonical cover of the functional dependencies in F is same as part c of this problem because there is no extraneous attribute and there are no two dependencies such that a→b, b→c so that a→c.

e.
No, it is not in BCNF. Because for schema to be BCNF for each non-trivial dependency A→B, A should be a superkey which is not the case here.

Below is the decomposition of each of the non-trivial dependencies to get the BCNF form:

For nontrivial dependency {custid} → {custname, cphone, address, city, zip},
{custid} is not superkey or {custname, cphone, address, city, zip} is not subset of {custid}.
Therefore, decomposing to BCNF gives us:
CUSTOMER (custid, custname, cphone, address, city, zip)

In this way, we finally get the decomposition:
CUSTOMER (**custid**, custname, cphone, address, city, zip)
CAKE (**cakeid**,cakename, cakeprice, slices, status)
INGREDIENT (**ingredid**, iname, iprice)
CONTAINS (**cakeid**, **ingredid**, qty)
ORDER (**custid, cakeid, orderdate**, pickupdate, pricepaid)
RELATION (**custid, cakeid, ingredid, orderdate**)

f.
Yes, the schema in section e is dependency preserving because each relation carries one of the relationships in canonical cover.

g. If we had an additional constraint that that a certain kind of cakes ordered on a certain day have the same price then the changes from b to f will be as follows:

b. Candidate key: {custid, cakeid, ingredid, orderdate}

c.

{custid} → {custname, cphone, address, city, zip}

{cakeid} → {cakename, cakeprice, slices, status}

{ingredid} → {iname, iprice}

{cakeid, ingredid} → {qty}

{custid, cakeid, orderdate} → {pricepaid, pickupdate}

{cakeid, orderdate} → {pricepaid}

d.

{custid} → {custname, cphone, address, city, zip}

{cakeid} → {cakename, cakeprice, slices, status}

{ingredid} → {iname, iprice}

{cakeid, ingredid} → {qty}

{custid, cakeid, orderdate} → {pickupdate}

{cakeid, orderdate} → {pricepaid}

Becasue {cakeid, orderdate} → {pricepaid}, so there is no need for {custid, cakeid, orderdate} → {pricepaid}

e.

It is not BCNF.

Decomposition:

CUSTOMER (custid, custname, cphone, address,city, zip)

CAKE (cakeid,cakename, cakeprice, slices, status)

INGREDIENT (ingredid, iname, iprice)

CONTAINS (cakeid, ingredid, qty)

ORDER (custid, cakeid, orderdate, pickupdate)

RELATION (custid, cakeid, ingredid, orderdate)
COST (cakeid, orderdate, pricepaid)

f.

The schema in section e is dependency preserving because each relation carries one of the relationships in canonical cover.

**Problem 4:**

a.

```
SELECT table_name, count(column_name) as colcount
FROM information_schema.columns natural join information_schema.tables
WHERE TABLE_SCHEMA = 'hw1'
GROUP BY TABLE_NAME;
```

b.

```
CREATE TABLE temp AS
SELECT table_name as tname, count(column_name) as colcount
FROM information_schema.columns natural join information_schema.tables
WHERE TABLE_SCHEMA = 'hw1'
GROUP BY TABLE_NAME;

SELECT tname
FROM temp
WHERE colcount = (SELECT MAX(colcount) FROM temp);
```

c.

```
SELECT table_name,column_name
FROM information_schema.columns natural join information_schema.tables
WHERE table_schema = 'hw1' and DATA_TYPE = 'int';
```

d.

```
CREATE TABLE temp2 AS
SELECT table_name, column_name, data_type
FROM information_schema.columns natural join information_schema.tables
where table_schema = 'hw1';

SELECT distinct t1.table_name, t2.table_name
FROM temp2 as t1, temp2 as t2
WHERE t1.table_name < t2.table_name and t1.column_name = t2.column_name and
t1.data_type = t2.data_type;
```