DTA (M) Technical Report 2019 2460681S > Wesley Scott

Task 1: Relational Modelling

Initial tables:
STUDENT (SID, SNAME, HCID, HCNAME, TID, TNAME, JYEAR)
TOPIC(SID, ADVNAME, ADVID, TOPIC)
TEXTBOOK(COURSE, ADVID, TEXTBOOK)

Task 1.1:

TABLE: STUDENT
FD1: SID > SNAME, HCID
The unique student ID entry defines the student's name, and as a student may only have one static home city, also defines their HCID.

FD2: HCID > HCNAME
The students home city ID defines home city name.
Therefore, HCNAME is defined by SID via transitive dependency such that:
SID > HCID > HCNAME

FD3: TID > TNAME
The unique team ID (TID) defines the team name.

FD4: SID, JYEAR > TID
As it is possible for a student to join 0 or 1 teams per year, we can also say that a combination of SID and JYEAR can uniquely identify a Student's Team ID (TID). As a student may join the same team again, we cannot with certainty say that {SID, TID} > JYEAR will always hold true. Also, as SID, JYEAR > TID therefore SID, JYEAR > TNAME via transitive dependency.

Since SID, JYEAR is the chosen Primary Key, it can be said that SNAME and HCID become partial dependencies, only requiring SID. This will be considered during normalisation.

TABLE: TOPIC
For each TOPIC a student has their own advisor; the same student can have multiple advisors, depending on which topic is referenced, therefore FDs are as follows:

FD1: SID, TOPIC > ADVID
ADVID is provided by a combination of SID and TOPIC. A student will have multiple advisors, one for each of their topics.

FD2: ADVID > TOPIC
However, it is also true that an advisor may only advise one TOPIC.

FD3: ADVID > ADVNAME
Each unique advisor has their own name.

Therefore <u>SID, TOPIC</u> > ADVID > ADVNAME via transitive dependency.

TABLE: TEXTBOOK
Each course has a set number of TEXTBOOKs, these textbook(s) must ALL be accounted for every time an ADVIOR for a COURSE is referenced in the relation.

To uniquely identify a tuple in this table currently we need all three attributes to act as a primary key.

FD: <u>COURSE, ADVID, TEXTBOOK</u> > COURSE, ADVID, TEXTBOOK

Task 1.2:

TABLE STUDENT: (FD4: <u>SID, JYEAR</u> > TID) states that a student's team ID is defined by both the student ID and the join year. As a student can join zero teams in a year, it is then possible they have joined no team, and thereby on INSERT, a tuple could potentially have unknown and therefore NULL values in both Team ID (TID), as well as JYEAR. This would affect the use of JYEAR as a primary key, which cannot be a null value. During creation of the database in its original form, it would be necessary to not allow NULL values for both attributes or create an edge case default entry for those without teams and therefore no join year. During normalisation these relations should be constructed such that the student parent become separate from the team relation and are only referenced together in a joining table.

TABLE TOPIC: With TOPIC, we are told that an advisor can only cover one topic, so due to this model, on INSERT, a new student with an advisor already present in the table, (see ADVID 1 (McReader)), duplication of information in the topic and advisor name attributes must occur. This means that if an Advisor were to change Topic, we would have to update a large number of tuples. This can be improved upon when normalising by separating the advisee and advisor relations.

TABLE TEXTBOOK: Due to the nature of this table, where each Advisor will use all of the necessary textbooks within a course, every time we insert a new ADVID against a course, we will require multiple tuples for each of the textbooks applicable to that course with the new Advisor. In the same way, a new textbook for either Database or Advanced Database would require new tuples for each of the advisors assigned to that course. This can be improved by separating the course and advisor relations.

Task 1.3:

TABLE: STUDENT
STUDENT (SID, SNAME, HCID, HCNAME, TID, TNAME, JYEAR)

The table STUDENT is currently in **1NF**, due to partial dependencies on the Primary Key.

In BCNF:
STUDENT(SID, SNAME, HCID(FK))
SID and JYEAR are the Primary Key, HCID becomes the foreign key, referencing HOMECITY(HCID).

STUDENT_JOINYEAR(SID(FK), JYEAR, TID(FK))
By moving SID and JYEAR to a new relationship we remove transitive dependencies from STUDENT. TID is a foreign key, referencing TEAM(TID).

HOMECITY(HCID, HCNAME)
HCID becomes the primary key of a new relation HOMECITY.

TEAM(TID, TNAME)
Team ID becomes the primary key of a new relation TEAM.

There are now no partial dependencies on any non-prime attribute within all tables, as well as no transitive dependencies, satisfying Boyce-Codd normal form.

TABLE: TOPIC
TOPIC(SID, ADVNAME, ADVID, TOPIC)

Topic is currently in **2NF**, due to transitive dependencies on the primary key.

3NF:
Consider the alternative, split Topic into two relations:

ADVISEE_TOPIC(SID, TOPIC, ADVID(FK))
A student can have multiple Advisors, but only one for each topic, therefor SID and TOPIC are required to uniquely identify a tuple. Due to the FD ADVID > TOPIC, this table cannot be normalised to BCNF.

ADVISOR(ADVID, ADVNAME)
ADVID uniquely identifies ADVNAME, which would be transitively dependant on primary key (FD4: SID, TOPIC > ADVID > ADVNAME) before split using FD3(ADVID > ADVNAME).

Task 1.4:

Due to the nature of the relationships in this table, where each Advisor will use all of the necessary textbooks within a course, every time we insert a new ADVID against a course, we will require multiple tuples for each of the textbook(s) applicable to that course.

TEXTBOOK

TEXTBOOK(COURSE, ADVID, TEXTBOOK)

Consider the following relations:

COURSE_TEXTBOOK(COURSE, TEXTBOOK)
ADVISOR_COURSE(ADVID, COURSE)

In this ideal split, we create two tables, each with 2 attributes as their respective primary key.

It is desirable to join these tables via courses, reducing the number of unnecessary tuples from each individual textbook, which are added individually for each advisor for each of their courses.

It also allows us to deal with the issue arising when the same textbook could be used in multiple courses, where COURSE_TEXTBOOK(COURSE, TEXTBOOK) accounts and uniquely identifies these occurrences. ADVISOR_COURSE(ADVID, COURSE) accounts for the potential issues which may arise where the same advisor belongs to multiple courses and uniquely identifies these instances.

The original relation causes some issues regarding splitting, due to the fact that no unique foreign key can be determined in any of the possible split states.

Based on this finding, creating an additional intermediary table with just the joining attribute would be required.

…

Therefore, the proposed solution is as follows:

COURSE_TEXTBOOK(<u>COURSE(FK), TEXTBOOK</u>) – references INTERMEDIARY_COURSE

INTERMEDIARY_COURSE(<u>COURSE</u>) – unique, defined within its own table

ADVISOR_COURSE (<u>ADVID, COURSE(FK)</u>) – references INTERMEDIARY_COURSE

Task 2: SQL Statements

FAO marker: SQL statements have been verified using pgAdmin, and as such had included some syntax (e.g. "public."TABLE"."ATTRIBUTE") in proven solutions. This has been edited out where possible but remained in places to avoid introducing errors when editing for this report and clarity.

Task 2.1: <u>STUDENT</u>

TABLE STUDENT:

```
CREATE TABLE STUDENT
(
   SID integer NOT NULL,
   SNAME character varying(20),
   HCID character varying(2),
   CONSTRAINT "PKSTUDENT" PRIMARY KEY (SID),
   CONSTRAINT "FKSTUDENTHCID" FOREIGN KEY (HCID)
      REFERENCES HOMECITY (HCID)
      ON UPDATE RESTRICT
      ON DELETE SET NULL
)
```

FKSTUDENTHCID: A home city ID should not be updated, this attribute should remain static, unique to a city, referenced from a static list of possible cities during tuple insertion - therefore ON UPDATE RESTRICT. However, if a HOMECITY is deleted we want to set a student's HCID to null as not to lose a unique row in STUDENT, thus losing a SID and SNAME.

HCID is set to NOT NULL in the parent relation below as it is the PK. If it is deleted a null value will hold its place in STUDENT.

TABLE HOMECITY:

```
CREATE TABLE HOMECITY
(
   HCID character varying(2) NOT NULL,
   HCNAME character varying(20),
   CONSTRAINT "PKHOMECITY" PRIMARY KEY (HCID)
)
```

TABLE STUDENT_JOINYEAR:

```
CREATE TABLE STUDENT_JOINYEAR
(
   SID integer NOT NULL,
   JYEAR integer NOT NULL,
   TID integer NOT NULL,
   CONSTRAINT "PKJOINYEAR" PRIMARY KEY (SID, JYEAR),
   CONSTRAINT "FKSID" FOREIGN KEY (SID)
      REFERENCES STUDENT (SID)
      ON UPDATE RESTRICT
      ON DELETE CASCADE,
   CONSTRAINT "FKTID" FOREIGN KEY (TID)
      REFERENCES TEAM (TID)
      ON UPDATE RESTRICT
      ON DELETE RESTRICT
)
```

FKSID: Where a SID must be deleted, it is desirable to remove those referencing rows within this child table. We should RESTRICT update of unique student IDs, which are unique and always static after insertion for each student.

FKTID: Consider the following approaches to the way the foreign key is dealt with for FKTID. Having a tuple in this table with a student's join year to a team, but a null team ID, as would be the case with DELETE SET NULL, seems useless to the user, and we would not lose student information from the database by deleting the tuple using DELETE CASCADE as the STUDENT table holds this, however we would lose a join year for a team, albeit it that the team would no longer exist.

The way the database is designed, there is a separate entry for each student's joining of a team. Therefore, it is advisable to RESTRICT Team ID deletions being pushed to the child, so that team transfer history is maintained in this table, even after the team is no longer active – as long as the student is active within the database.

Updating of team unique IDs should also be restricted, as a team ID should be static and unique to all teams upon insertion, only changed in strenuous cases by a database admin.

TABLE TEAM:

```
CREATE TABLE TEAM
(
   TID integer NOT NULL,
   TNAME character varying(7),
   CONSTRAINT "PKTEAM" PRIMARY KEY (TID)
)
```

TASK 2.1: <u>TOPIC</u>

TABLE ADVISEE_TOPIC:

```
CREATE TABLE ADVISEE_TOPIC
(
   SID integer NOT NULL,
   TOPIC character varying(20) NOT NULL,
   ADVID integer,
   CONSTRAINT "PKADVISEE_TOPIC" PRIMARY KEY (SID, TOPIC),
   CONSTRAINT "FKADVID" FOREIGN KEY (ADVID)
      REFERENCES ADVISOR (ADVID)
      ON UPDATE RESTRICT
      ON DELETE SET NULL
)
```

FKADVID: An ADVID should be static and unique to all advisors upon insertion, therefore updates to this value should be restricted. However, as the table holds unique rows regarding a student's attachment to a topic, alongside their advisor, if an advisor is deleted, we wish to retain the rest of the information, allowing future designation of a new advisor to the student for that topic, therefore ON DELETE of an advisor ID SET NULL.

TABLE ADVISOR:

```
CREATE TABLE ADVISOR
(
   ADVID integer NOT NULL,
   ADVNAME character varying(20),
   CONSTRAINT "PKADVISOR" PRIMARY KEY (ADVID)
)
```

Task 2.2:

SQL1: SELECT COUNT(public."STUDENT"."SID"), public."HOMECITY"."HCID"
```
        FROM public."STUDENT", public."HOMECITY"
        WHERE public."HOMECITY"."HCID" = public."STUDENT"."HCID"
        GROUP BY public."HOMECITY"."HCID"
```

SQL2: SELECT COUNT(public."STUDENT"."SID"), public."TEAM"."TNAME"
```
        FROM public."STUDENT", public."STUDENT_JOINYEAR", public."TEAM"
        WHERE public."STUDENT"."SID" = public."STUDENT_JOINYEAR"."SID"
        AND public."TEAM"."TID" = public."STUDENT_JOINYEAR"."TID"
        AND public."STUDENT_JOINYEAR"."JYEAR" = 2001
        GROUP BY public."TEAM"."TNAME"
```

SQL3: SELECT public."ADVISOR"."ADVNAME"
```
        FROM public."ADVISOR", public."ADVISEE_TOPIC"
        WHERE public."ADVISOR"."ADVID" = public."ADVISEE_TOPIC"."ADVID"
        GROUP BY public."ADVISOR"."ADVNAME"
        HAVING COUNT(DISTINCT public."ADVISEE_TOPIC"."SID") > 10
```

SQL4: SELECT
```
public."ADVISEE_TOPIC"."TOPIC", COUNT(DISTINCT public."ADVISOR"."ADVID")
        FROM public."ADVISOR", public."ADVISEE_TOPIC"
        WHERE public."ADVISOR"."ADVID" = public."ADVISEE_TOPIC"."ADVID"
        GROUP BY public."ADVISEE_TOPIC"."TOPIC"
        LIMIT 1
```

SQL5: SELECT public."STUDENT"."SNAME", COUNT(*)
```
        FROM public."STUDENT", public."STUDENT_JOINYEAR", public."TEAM"
        WHERE public."STUDENT"."SID" = public."STUDENT_JOINYEAR"."SID"
        AND public."TEAM"."TID" = public."STUDENT_JOINYEAR"."TID"
        AND public."STUDENT_JOINYEAR"."JYEAR" >= 2001
        GROUP BY public."STUDENT"."SNAME"
        HAVING COUNT (*) =
(SELECT MAX(TEAMCOUNT) FROM
(SELECT COUNT(DISTINCT public."STUDENT_JOINYEAR"."TID") AS TEAMCOUNT FROM
"STUDENT_JOINYEAR") AS SJY)
```

TASK 3:

3.1:

```
SELECT "EMPLOYEE"."FName"
FROM "EMPLOYEE"
GROUP BY "EMPLOYEE"."FName"
HAVING COUNT("EMPLOYEE"."SSN") > 1
```

3.2:

```
SELECT COUNT("EMPLOYEE"."FName")
FROM "EMPLOYEE"
GROUP BY "EMPLOYEE"."FName"
HAVING COUNT("EMPLOYEE"."SSN") > 1
```

3.3:

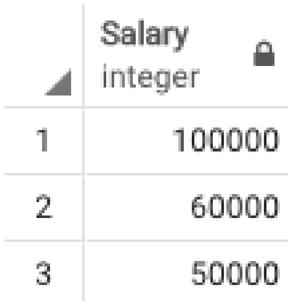
FAO marker: Regarding task 3.3, I was unsure based on the comments in red if the task was to show purely the 3 distinct top 3 salary values within the Salary column only, which is what I understood from the black text, or rather to retrieve all tuples related to entries with top 3 salaries. I have provided a solution to both interpretations:

"provide a SQL query which shows the top-3 highest salary values without using the SQL LIMIT operator. "

Solution provided below:

```
SELECT "Salary"
FROM(SELECT DISTINCT("Salary"),
        DENSE_RANK() OVER(ORDER BY "Salary" desc) SalaryRank
        FROM "EMPLOYEEPK") requiredalias
WHERE SalaryRank <= 3;
```

<u>RESULT</u> (new tuples have been introduced for testing):

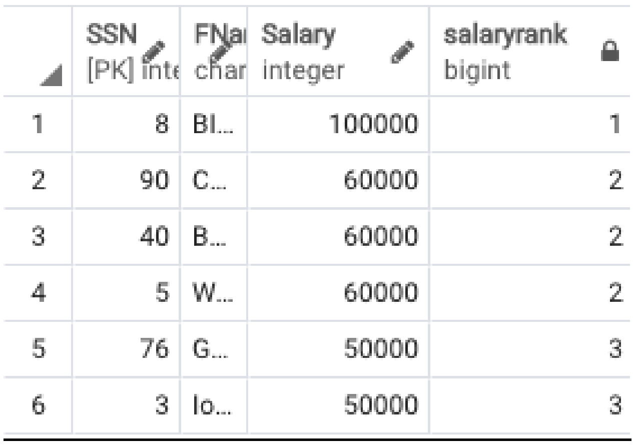


	Salary integer
1	100000
2	60000
3	50000

“tuples corresponding to the top-3 highest salaries.”

```
SELECT *
FROM(SELECT "SSN", "FName" , "Salary",
        DENSE_RANK() OVER(ORDER BY "Salary" desc) SalaryRank
        FROM "EMPLOYEEPK") requiredalias
WHERE SalaryRank <= 3;
```

<u>RESULT</u> (with the same data as previous solution)



	SSN [PK] inte	FNa char	Salary integer	salaryrank bigint
1	8	Bl...	100000	1
2	90	C...	60000	2
3	40	B...	60000	2
4	5	W...	60000	2
5	76	G...	50000	3
6	3	Io...	50000	3