Assignment1 (Chap 1, Chap 2)

Chapter 1

1. 7 List four significant differences between a file-processing system and a DBMS.

1. A database management system could coordinate both the physical and the logical access to the data, but a file-processing system coordinates only the physical access.

2. Database management systems allow flexible access to data but file-processing system is designed to allow predetermined access to data

3. Database management systems are designed to coordinate simultaneous access to the same data by multiple users. File processing systems are often designed to allow one or more programs to access different data files simultaneously.

4. A database management system reduces the amount of data duplication by ensuring that a physical piece of data is available to all programs authorized to have access to it, whereas data written by one program in a file-processing system may not be readable by another program.

1. 8 Explain the concept of physical data independence and its importance in database systems.

Physical data independence is the ability to modify the physical scheme without making it necessary to rewrite application programs. Such modifications include changing from unblocked to blocked record storage, or from sequential to random access files. Such a modification might be adding a field to a record; an application program's view hides this change from the program.

2. 14 Consider the employee database of Figure 2.17. Give an expression in the relational algebra to express each of the following queries:

a. Find the ID and name of each employee who works for "BigBank".

b. Find the ID, name, and city of residence of each employee who works for "BigBank".

c. Find the ID, name, street address, and city of residence of each employee who works for "BigBank" and earns more than $10000.

d. Find the ID and name of each employee in this database who lives in the same city as the company for which she or he works.

employee (ID， *person_name*, *street*, *city*)
works (*ID*, *company_name*, *salary*)
company (*company_name*, *city*)
Figure 2.17 Employee database.

a. $\prod_{ID,\ person\_name}(\sigma_{company\_name\ =\ \text{“BigBank”}}(\boldsymbol{employee} \bowtie_{employee.id=works.id} \boldsymbol{works}))$

b. $\prod_{ID,person\_name,city}(\sigma_{company\_name=\text{“BigBank”}(}\boldsymbol{employee} \bowtie_{employee.id=works.id} \boldsymbol{works}))$

c. $\prod_{ID,\ person\_name,\ street,\ city}(\sigma_{company\_name\ =\ \text{“BigBank”}\ \wedge\ salary\ >\ 10000}(\boldsymbol{employee} \bowtie_{employee.id=works.id}\boldsymbol{works}))$

d. $\prod_{ID,person\_name}(\sigma_{employee.city=company.city}(\boldsymbol{employee}\bowtie_{employee.ID=works.ID} \boldsymbol{works)}\bowtie_{works.company\_name=company.company\_name} (\mathbf{company}))$

2. 15 Consider the bank database of Figure 2.18. Give an expression in the relational algebra for each of the following queries:

a. Find each loan number with a loan amount greater than $10000.

b. Find the ID of each depositor who has an account with a balance greater than $6000.

c. Find the ID of each depositor who has an account with a balance greater than $6000 at the “Uptown” branch.

branch(*branch name*, *branch city, assets*)
customer (*ID*, *customer_name*, *customer_street, customer_city*)
loan (*loan_number*, *branch_name, amount*)
borrower (*ID*, *loan_number*)
account (*account_number*, *branch_name, balance*)
depositor (*ID*, *account_number*)

Figure 2.18 Bank database.

a. $\prod_{loan\_number}(\sigma_{amount>\ 10000}(\boldsymbol{loan}))$

b. $\prod_{ID}(\sigma_{\text{balance}>\ 6000}(\boldsymbol{depositor} \bowtie_{depositor.account\_number=account.account\_number} \boldsymbol{account}))$

c. $\prod_{ID}(\sigma_{\text{balance}>6000\ \text{^branch\_name=“Uptown”}}(\boldsymbol{depositor}\bowtie_{depositor.account\_number=account.account\_number} \boldsymbol{account}))$

答案不唯一，仅供参考

3.1 Write the following queries in SQL, using the university schema (as shown in the lecture ppt or the textbook).
a. Find the titles of courses in the Comp. Sci. department that have 3 credits.

select title from course where dept name = 'Comp. Sci.' and credits = 3

b. Find the IDs of all students who were taught by an instructor named Einstein; make sure there are no duplicates in the result.

```
select distinct student.ID from
(student join takes using(ID)) join (instructor join teaches using(ID))
using(course_id, sec_id, semester, year)
where instructor.name = 'Einstein'
```

c. Find the highest salary of any instructor.

```
select max(salary) from instructor
```

d. Find all instructors earning the highest salary (there may be more than one with the same salary).

```
select ID, name from instructor where salary = (select max(salary) from instructor)
```

e. Find the enrollment of each section that was offered in Fall 2017.

```
select course_id, sec_id, count(ID) from
section natural join takes
where semester = 'Fall' and year = 2017
group by course_id, sec_id
```

OR

```
select course_id, sec_id, (select count(ID)
from takes where takes.year = section.year and takes.semester = section.semester and
takes.course_id = section.course_id and takes.sec_id = section.sec_id)
from section where semester = 'Fall' and year = 2017
```

f. Find the maximum enrollment, across all sections, in Fall 2017.

```
select max(enrollment) from (select count(ID) as enrollment from
section natural join takes
where semester = 'Fall' and year = 2017
 group by course_id, sec_id)
```

g. Find the sections that had the maximum enrollment in Fall 2017.

with sec_enrollment as (select course_id, sec_id, count(ID) as enrollment from section natural join takes
where semester = 'Fall' and year = 2017
group by course_id, sec_id)
select course_id, sec_id from sec_enrollment where enrollment = (select max(enrollment) from sec_enrollment)

3.3 Write the following inserts, deletes, or updates in SQL, using the university schema.
a. Increase the salary of each instructor in the Comp. Sci. department by 10%.

update instructor
set salary = salary * 1.10
where dept name = 'Comp. Sci.'

b. Delete all courses that have never been offered (i.e., do not occur in the *section* relation).

delete from course
where course_id not in (select course_id from section)

c. Insert every student whose *tot cred* attribute is greater than 100 as an instructor in the same department, with a salary of $10,000.

insert into instructor
select ID, name, dept_name, 10000 from student
where tot_cred > 100

3.8 Consider the bank database of Figure 3.18, where the primary keys are underlined. Construct the following SQL queries for this relational database.
a. Find the ID of each customer of the bank who has an account but not a loan.

(select ID from depositor)
except
(select ID from borrower)

b. Find the ID of each customer who lives on the same street and in the same city as customer '12345'.

select F.ID from customer F join customer S
using(customer_street, customer_city) where S.ID = '12345'

c. Find the name of each branch that has at least one customer who has an account in the bank and who lives in "Harrison".

select distinct branch_name from

account natural join depositor natural join customer
where customer_city = 'Harrison'

branch(*branch_name*, *branch city, assets*)
customer (*ID*, *customer_name*, *customer_street, customer_city*)
loan (*loan_number*, *branch_name, amount*)
borrower (*ID*, *loan number*)
account (*account_number*, *branch_name, balance*)
depositor (*ID*, *account_number*)
Figure 3.18 Banking database.

3.9 Consider the relational database of Figure 3.19, where the primary keys are underlined. Give an expression in SQL for each of the following queries.
a. Find the ID, name, and city of residence of each employee who works for "First Bank Corporation".

select e.ID, e.person_name, e.city from employee e, works w
where w.company_name = 'First Bank Corporation' and w. ID = e. ID

b. Find the ID, name, and city of residence of each employee who works for First Bank Corporation" and earns more than $10000.

select ID, person_name, city from employee where ID in (select ID from works where company name = 'First Bank Corporation' and salary > 10000)

c. Find the ID of each employee who does not work for "First Bank Corporation".

select ID from works where company name <> 'First Bank Corporation'

OR

select ID from employee where ID not in (select ID from works where company name = 'First Bank Corporation')

d. Find the ID of each employee who earns more than every employee of "Small Bank Corporation".

select ID from works where salary > all (select salary from works where company name = 'Small Bank Corporation')

e. Assume that companies may be located in several cities. Find the name of each company that is located in every city in which “Small Bank Corporation”is located.

Exists A = True → A is not null
A Except B → A – B

P. 26 of Chapter 3:

- Find courses that ran in Fall 2017 but not in Spring 2018 (MySQL not support)

```
(select course_id from section where sem = 'Fall' and year = 2017)
except
(select course_id from section where sem = 'Spring' and year = 2018)
```

Assume: company S is located in every city in which “Small Bank Corporation”is located.

↓

City(SBC) ⊂ City(S): city set of SBC included by city set of S

↓

City(SBC) - City(S) is null: every city appears in city(SBC) also appears in city(S)

↓

Not Exists City(SBC) except City(S)

select S.company_name from company S
where not exists ((select city from company where company_name = ’Small Bank Corporation’)
except
(select city from company T where T.company_name = S.company_name))

f. Find the name of the company that has the most employees (or companies, in the case where there is a tie for the most).

select company_name from works
group by company_name
having count (distinct ID) >= all (select count(distinct ID) from works group by company_name)

g. Find the name of each company whose employees earn a higher salary, on average, than the average salary at “First Bank Corporation”.

select company_name from works
group by company_name
having avg (salary) > (select avg (salary) from works where company_name = 'First Bank Corporation')

employee (*ID*, *person_name*, *street*, *city*)
works (*ID*, *company_name*, *salary*)
company (*company_name*, *city*)
manages (*ID*, *manager_id*)
Figure 3.19 Employee database.

4.2 Write the following queries in SQL:

a. Display a list of all instructors, showing each instructor's ID and the number of sections taught. Make sure to show the number of sections as 0 for instructors who have not taught any section. Your query should use an outer join, and should not use subqueries.

```
select ID, count(sec_id)
from instructor natural left outer join teaches
group by ID;
```

b. Write the same query as in part a, but using a scalar subquery and not using outer join.

```
select id, (select count(sec_id)
            from teaches
            where teaches.id=instructor.id) as num_of_course
from instructor;
```

c. Display the list of all course sections offered in Spring 2018, along with the ID and name of each instructor teaching the section. If a section has more than one instructor, that section should appear as many times in the result as it has instructors. If a section does not have any instructor, it should still appear in the result with the instructor name set to "—".

```
select course_id, sec_id, ID, coalesce(name, '—') as name
from (section natural left outer join teaches) natural left outer join instructor
where semester = 'Spring' and year = 2018;
```

d. Display the list of all departments, with the total number of instructors in each department, without using subqueries. Make sure to show departments that have no instructors, and list those departments with an instructor count of zero.

```
select dept_name, count(ID) as instructor_num
from department natural left outer join instructor
group by dept_name;
```

4.3 Outer join expressions can be computed in SQL without using the SQL outer join operation. To illustrate this fact, show how to rewrite each of the following SQL queries without using the outer join expression.

a. select * from *student* **natural left outer join** *takes*

```
select * from student natural join takes
union
select ID, name, dept_name, tot_cred, NULL, NULL, NULL, NULL, NULL
from student S1 where not exists
```

```
(select ID from takes T1 where T1.id = S1.id);
```

b. select * from *student* **natural full outer join** *takes*

```
select * from student natural join takes
union
(select ID, name, dept_name, tot_cred, NULL, NULL, NULL, NULL, NULL
from student S1
where not exists
(select ID from takes T1 where T1.id = S1.id))
union
(select ID, NULL, NULL, NULL, course_id, sec_id, semester, year, grade
from takes T1
where not exists
(select ID from student S1 where T1.id = S1.id));
```

Assignment 4 (Chapter 4, Chapter5)

4.13 Consider a view v whose definition references only relation r.

(1) If a user is granted select authorization on v, does that user need to have select authorization on r as well? Why or why not?

(2) If a user is granted update authorization on v, does that user need to have update authorization on r as well? Why or why not?

1. If a user is granted select authorization on view v, they do not necessarily need select authorization on relation r. This is because the view is simply a virtual table that presents data from relation r in a certain way. The user only needs to be able to see the data presented in the view, which does not necessarily require access to the underlying table.
2. If a user is granted update authorization on view v, they may need update authorization on relation r as well, depending on the view definition and the type of update being performed. If the view definition includes all columns from relation r and the update operation modifies data in those columns, the user will need update authorization on r as well. However, if the view definition only includes a subset of columns from r, and the update operation only modifies data in those columns, the user may not need update authorization on r. Additionally, if the update operation is performed through a trigger or stored procedure that has the necessary update authorization on r, the user may not need direct update authorization on r.

5.6 Consider the bank database of Figure 5.21. Let us define a view branch_cust as follows:

```
create view branch_cust as (
select branch_name, customer_name
from depositor, account
where depositor.account_number = account.account_number);
```

Suppose that the view is materialized (so you can regard branch_cust as a base table); that is, the view is computed and stored. Write triggers to maintain the view, that is, to keep it up-to-date on insertions to depositor or account. It is not necessary to handle deletions or updates. Note that, for simplicity, we have not required the elimination of duplicates.

branch (branch_name, branch_city, assets)
customer (customer_name, customer_street, customer_city)
loan (loan_number, branch_name, amount)
borrower (customer_name, loan_number)
account (account_number, branch_name, balance)
depositor (customer_name, account_number)

Figure 5.21 Banking database for Exercise 5.6.

Assuming that the DBMS supports triggers on views, the following triggers can be written to maintain the branch_cust view:

Trigger to update the view on insertion to the depositor table:

```
CREATE TRIGGER insert_depositor_trigger AFTER INSERT ON depositor
FOR EACH ROW
BEGIN
  INSERT INTO branch_cust (branch_name, customer_name)
  VALUES ((SELECT branch_name FROM account WHERE account_number =
NEW.account_number), NEW.customer_name);
END;
```

This trigger inserts a new row into the branch_cust view for every new row inserted into the depositor table. The branch_name is obtained by looking up the account_number of the new depositor in the account table.

Trigger to update the view on insertion to the account table:

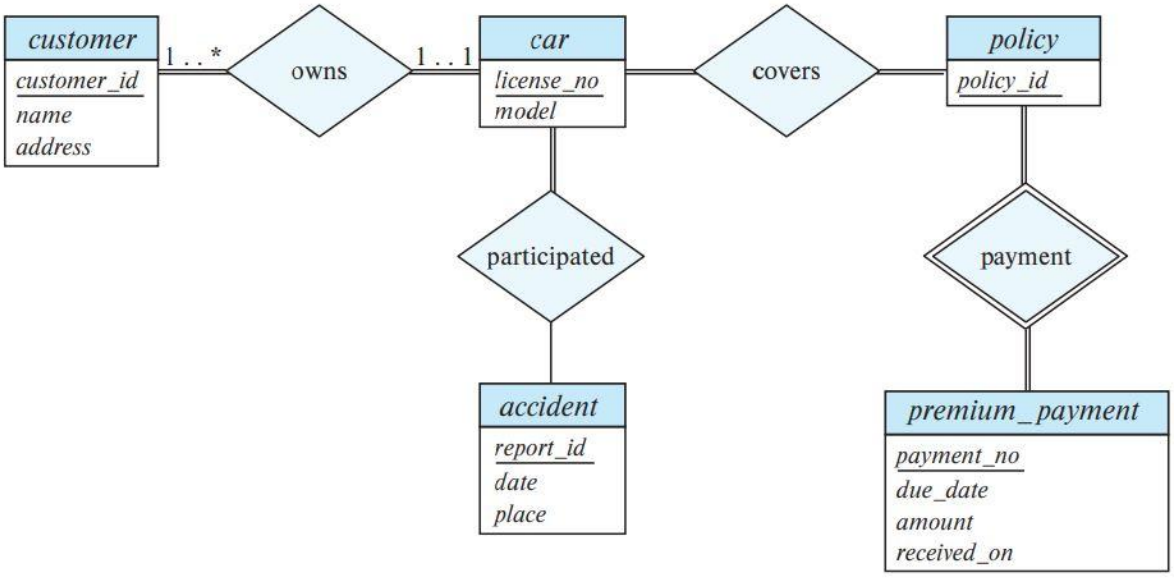
```
CREATE TRIGGER insert_account_trigger AFTER INSERT ON account
FOR EACH ROW
BEGIN
  INSERT INTO branch_cust (branch_name, customer_name)
  SELECT NEW.branch_name, depositor.customer_name
  FROM depositor
  WHERE depositor.account_number = NEW.account_number;
END;
```

This trigger inserts a new row into the branch_cust view for every new row inserted into the account table. The branch_name is taken from the new account row, and the customer_name is obtained by looking up the account_number in the depositor table.

Note that these triggers do not handle updates or deletions, as stated in the problem. Also, as mentioned in the problem statement, these triggers do not eliminate duplicates, so the branch_cust view may contain duplicates if multiple customers have accounts at the same branch.

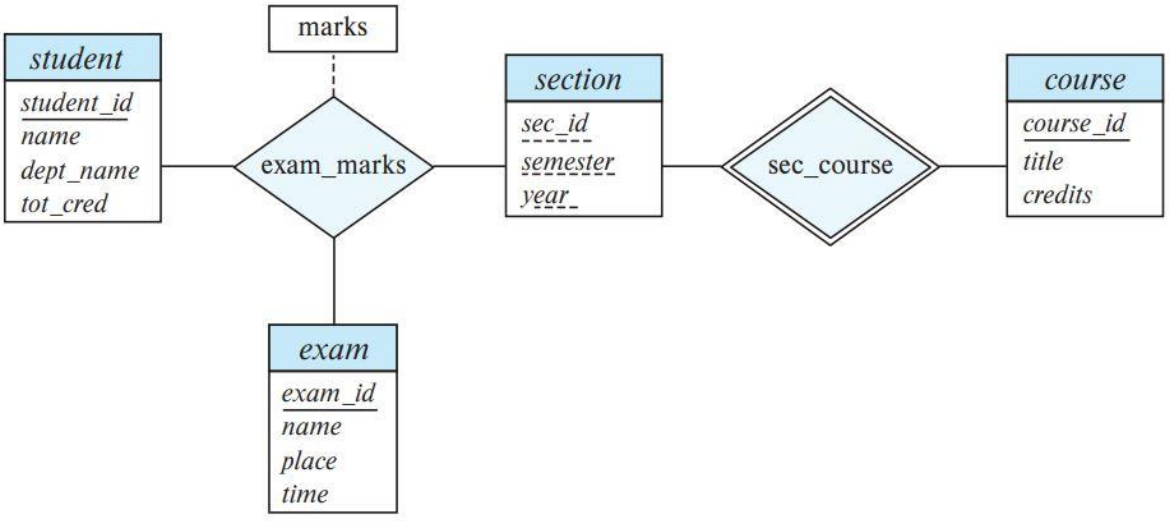
Assignment 5 (Chapter6)

6.1 Construct an E-R diagram for a car insurance company whose customers own one or more cars each. Each car has associated with it zero to any number of recorded accidents. Each insurance policy covers one or more cars and has one or more premium payments associated with it. Each payment is for a particular period of time, and has an associated due date, and the date when the payment was received.

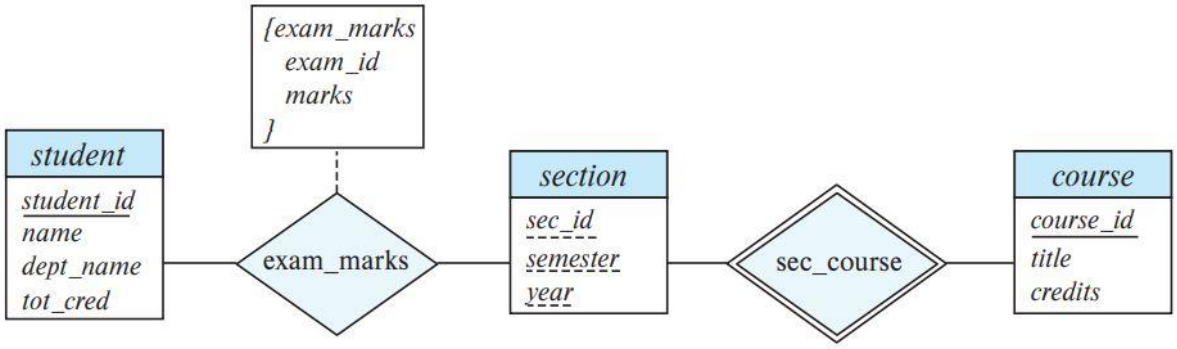


6.2 Consider a database that includes the entity sets student, course, and section from the university schema and that additionally records the marks that students receive in different exams of different sections.

a. Construct an E-R diagram that models exams as entities and uses a ternary relationship as part of the design.



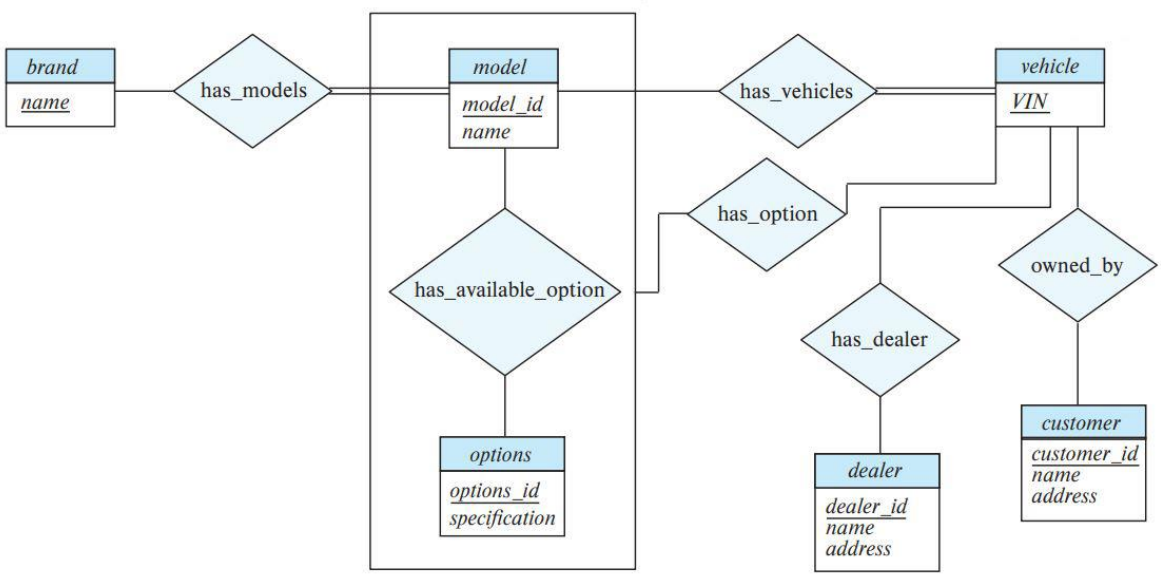
b. Construct an alternative E-R diagram that uses only a binary relationship between student and section. Make sure that only one relationship exists between a particular student and section pair, yet you can represent the marks that a student gets in different exams.



6.22 Design a database for an automobile company to provide to its dealers to assist them in maintaining customer records and dealer inventory and to assist sales staff in ordering cars.

Each vehicle is identified by a vehicle identification number (VIN). Each individual vehicle is a particular model of a particular brand offered by the company (e.g., the XF is a model of the car brand Jaguar of Tata Motors). Each model can be offered with a variety of options, but an individual car may have only some (or none) of the available options. The database needs to store information about models, brands, and options, as well as information about individual dealers, customers, and cars. Your design should include an E-R diagram, a set of relational schemas, and a list of constraints, including primary-key and foreign-key constraints.

E-R diagram:



Schema:

brand(brand_name),

model(model_id, model_name)

vehicle(VIN, dealer_id, customer_id)

option(option_id, specification)

customer(customer_id, customer_name, address)

dealer(dealer_id, dealer_name, address)

has_model(brand_name, model_id ,

foreign key brand_name references brand,

foreign key model_id references model)

has_vehicle(model_id, VIN,

foreign key VIN references vehicle,

foreign key model_id references model)

has_available_option(model_id, option_id,

foreign key option_id references option,

foreign key model_id references model)

has_option(VIN, model_id, option_id,

foreign key VIN references vehicle,

foreign key (model_id, option_id) references available_option)

has_dealer(VIN, dealer_id ,

foreign key dealer_id references dealer,

foreign key VIN references vehicle)

owned_by(VIN, customer_id,

foreign key customer_id references customer,

foreign key VIN references vehicle)

The list of constraints:

brand:

Primary Key: brand_name

model:

Primary Key: model_id

vehicle:

Primary Key: VIN

Foreign Key: dealer_id REFERENCES dealer(dealer_id)

Foreign Key: customer_id REFERENCES customer(customer_id)

option:

Primary Key: option_id

customer:

Primary Key: customer_id

dealer:

Primary Key: dealer_id

has_model:

Primary Key: (brand_name, model_id)

Foreign Key: brand_name REFERENCES brand(brand_name)

Foreign Key: model_id REFERENCES model(model_id)

has_vehicle:

Primary Key: (model_id, VIN)

Foreign Key: model_id REFERENCES model(model_id)

Foreign Key: VIN REFERENCES vehicle(VIN)

has_available_option:

Primary Key: (model_id, option_id)

Foreign Key: model_id REFERENCES model(model_id)

Foreign Key: option_id REFERENCES option(option_id)

has_option:

Primary Key: (VIN, model_id, option_id)

Foreign Key: VIN REFERENCES vehicle(VIN)

Foreign Key: (model_id, option_id) REFERENCES has_available_option(model_id, option_id)

has_dealer:

Primary Key: (VIN, dealer_id)

Foreign Key: dealer_id REFERENCES dealer(dealer_id)

Foreign Key: VIN REFERENCES vehicle(VIN)

owned_by:

Primary Key: (VIN, customer_id)

Foreign Key: customer_id REFERENCES customer(customer_id)

Foreign Key: VIN REFERENCES vehicle(VIN)

Assignment 6 (Chapter7, part1)

7.1 Suppose that we decompose the schema *R* = (*A*, *B*, *C*, *D*, *E*) into

(*A*, *B*, *C*)

(*A*, *D*, *E*).

Show that this decomposition is a lossless decomposition if the following set *F* of functional dependencies holds:

$A \rightarrow BC$

$CD \rightarrow E$

$B \rightarrow D$

$E \rightarrow A$

Answer: A decomposition $\{R_1,\ R_2\}$ is a lossless-join decomposition if $R_1 \cap R_2 \rightarrow R_1$ or $R_1 \cap R_2 \rightarrow R_2$. Let $R_1 = (A, B, C)$, $R_2 = (A, D, E)$, and $R_1 \cap R_2 = A$. Since A is a candidate key (see Exercise 7.11), Therefore $R_1 \cap R_2 \rightarrow R_1$.

7.2 List all nontrivial functional dependencies satisfied by the relation of Figure 7.18.

A	B	C
a1	b1	c1
a1	b1	c2
a2	b1	c1
a2	b1	c3

Figure 7.18 Relation of Exercise 7.2.

Answer: The nontrivial functional dependencies are: $A \rightarrow B$ and $C \rightarrow B$, and a dependency they logically imply: $AC \rightarrow B$. There are 19 trivial functional dependencies of the form $\alpha \rightarrow \beta$, where $\beta \subseteq \alpha$. C does not functionally determine A because the first and third tuples have the same C but different A values. The same tuples also show B does not functionally determine A. Likewise, A does not functionally determine C because the first two tuples have the same A value and different C values. The same tuples also show B does not functionally determine C.

7.13 Show that the decomposition in Exercise 7.1 is not a dependency-preserving decomposition.

Answer: The dependency $B \rightarrow D$ is not preserved. F_1, the restriction of F to (A, B, C) is $A \rightarrow ABC$, $A \rightarrow AB$, $A \rightarrow AC$, $A \rightarrow BC$, $A \rightarrow B$, $A \rightarrow C$, $A \rightarrow A$, $B \rightarrow B$, $C \rightarrow C$, $AB \rightarrow AC$, $AB \rightarrow ABC$, $AB \rightarrow BC$, $AB \rightarrow AB$, $AB \rightarrow A$, $AB \rightarrow B$, $AB \rightarrow C$, AC (same as AB), BC (same as AB), ABC (same as AB). F_2, the restriction of F to (C, D, E) is $A \rightarrow ADE$, $A \rightarrow AD$, $A \rightarrow AE$, $A \rightarrow DE$, $A \rightarrow A$, $A \rightarrow D$, $A \rightarrow E$, $D \rightarrow D$, E (same as A), AD, AE, DE, ADE (same as A). $(F_1 \cup F_2)^+$ is easily seen not to contain $B \rightarrow D$ since the only FD in $F_1 \cup F_2$ with B as the left side is $B \rightarrow B$, a trivial FD. We shall see in Exercise 7.22 that $B \rightarrow D$ is indeed in F^+. Thus $B \rightarrow D$ is not preserved. Note that $CD \rightarrow ABCDE$ is also not preserved.

A simpler argument is as follows: F_1 contains no dependencies with D on the right side of the arrow. F_2 contains no dependencies with B on the left side of the arrow. Therefore for $B \rightarrow D$ to be preserved there must be an FD $B \rightarrow \alpha$ in F_1^+ and $\alpha \rightarrow D$ in F_2^+ (so $B \rightarrow D$ would follow by transitivity). Since the intersection of the two schemes is A, $\alpha = A$. Observe that $B \rightarrow A$ is not in F_1^+ since $B^+ = BD$.

Assignment 7 (Chapter7, part2)

7.14 Show that there can be more than one canonical cover for a given set of functional dependencies, using the following set of dependencies:
$X \rightarrow YZ$, $Y \rightarrow XZ$, and $Z \rightarrow XY$.

Let's consider the set of functional dependencies: `X → YZ`, `Y → XZ`, and `Z → XY`. We can derive two different canonical covers for this set of dependencies.

Canonical Cover 1:

- `X → Y`
- `Y → Z`
- `Z → X`

Canonical Cover 2:

- `X → Z`
- `Y → X`
- `Z → Y`

Both of these covers are minimal representations of the original set of dependencies and satisfy the definition of a canonical cover. Hence, there can be more than one canonical cover for a given set of functional dependencies.

7.15 The algorithm to generate a canonical cover only removes one extraneous attribute at a time. Use the functional dependencies from Exercise 7.14 to show what can go wrong if two attributes inferred to be extraneous are deleted at once.

Sure! Let's consider the set of functional dependencies from the previous exercise: `X → YZ`, `Y → XZ`, and `Z → XY`. If we apply the algorithm to generate a canonical cover and only remove one extraneous attribute at a time, we get the following:

1. We can see that `Z` is extraneous in `X → YZ` because `X → Y` and `Y → Z` (by transitivity). So we can remove `Z` from the right-hand side of `X → YZ` to get `X → Y`.
2. Similarly, we can see that `X` is extraneous in `Y → XZ` because `Y → Z` and `Z → X`. So we can remove `X` from the right-hand side of `Y → XZ` to get `Y → Z`.
3. Finally, we can see that `Y` is extraneous in `Z → XY` because `Z → X` and `X → Y`. So we can remove `Y` from the right-hand side of `Z → XY` to get `Z → X`.

Thus, the canonical cover for this set of functional dependencies is `{X → Y, Y → Z, Z → X}`.

However, if we try to remove two attributes inferred to be extraneous at once, we might end up with an incorrect result. For example, let's say we try to remove both `Z` from the right-hand side of `X → YZ` and `X` from the right-hand side of `Y → XZ` at once. We would end up with `{X → Y, Y → Z, Z → XY}`. But this is not a correct canonical cover because it does not preserve all the dependencies in the original set. Specifically, it does not preserve the dependency `Y → XZ`.

So it's important to only remove one extraneous attribute at a time when generating a canonical cover.

7.21 Give a lossless decomposition into BCNF of schema *R* of Exercise 7.1.

(ABCE) (BD)

7.22 Give a lossless, dependency-preserving decomposition into 3NF of schema *R* of Exercise 7.1.

(ABC) (CDE) (B D) (A E)

7.30 Consider the following set *F* of functional dependencies on the relation schema (*A*, *B*, *C*, *D*, *E*, *G*):

$$A \rightarrow BCD$$
$$BC \rightarrow DE$$
$$B \rightarrow D$$
$$D \rightarrow A$$

a. Compute B^+.

B+ is the set of all attributes that can be functionally determined by B. Given the functional dependencies you provided, we can derive the following:

1. B → D (given)
2. D → A (given)
3. A → BCD (given)
4. B → BCD (using 2 and 3)
5. BC → DE (given)
6. B → DE (using 4 and 5)

Therefore, B+ = {B, D, A, C, E}. Is there anything else you would like to know?

(A,B,C,D,E)

b. Prove (using Armstrong's axioms) that *AG* is a superkey.

Sure! To prove that `AG` is a superkey using Armstrong's axioms, we need to show that the closure of `AG` with respect to the given set `F` of functional dependencies includes all attributes in the relation schema. We start with `AG+ = {A, G}` and apply the dependencies in `F` one by one:

- From `A → BCD`, we can add `{B, C, D}` to `AG+`, so now `AG+ = {A, B, C, D, G}`.
- From `BC → DE`, we can add `{D, E}` to `AG+`, so now `AG+ = {A, B, C, D, E, G}`. No more dependencies can be applied and all attributes in the relation schema are included in the closure of AG. Hence AG is a superkey.

Therefore, we have proved using Armstrong's axioms that `AG` is a superkey for the given relation schema and set of functional dependencies.

c. Compute a canonical cover for this set of functional dependencies *F*; give each step of your derivation with an explanation.

A canonical cover for a set of functional dependencies F is a minimal set of dependencies equivalent to F. To compute a canonical cover for the given set F = {A → BCD, BC → DE, B → D, D → A}, we can follow these steps:

1. **Split the right-hand side** of all dependencies in F that have more than one attribute on the right-hand side. This gives us F' = {A → B, A → C, A → D, BC → D, BC → E, B → D, D → A}.
2. **Remove extraneous attributes** from the left-hand side of each dependency in F'. An attribute X is extraneous in a dependency YZ → W if YZ → W is equivalent to (Y - X)Z → W. In this case, we can see that C is extraneous in the dependency BC → D because B+ = {B, D, A, C, E} and thus B determines D. So we can remove C from the left-hand side of BC → D to get B → D. This gives us F'' = {A → B, A → C, A → D, B → D, BC → E, B → D, D → A}.
3. **Remove redundant dependencies** from F''. A dependency YZ → W is redundant in F'' if (F'' - {YZ → W})+ = F''+. In this case, we can see that the dependency B → D is redundant because (F'' - {B → D})+ = {A+, B+, C+, D+, E+, G+} = {A+, BCD+, C+, AD+, E+, G+} = F''+. So we can remove the dependency B → D from F''. This gives us F''' = {A → B, A → C, A → D, BC → E, B → D, D → A}.
4. **Combine dependencies** with the same left-hand side in F'''. This gives us the final canonical cover for F: {A → BCD, BC → E, B → D, D → A}.

As B->E (from a), BC->E will be B->E.

A->BC, B->DE, D->A

d. Give a 3NF decomposition of the given schema based on a canonical cover.

A relation schema R is in third normal form (3NF) if for every nontrivial functional dependency X → Y in a canonical cover for F, either X is a superkey or Y is a prime attribute. To decompose R into 3NF based on a canonical cover for F, we can follow these steps:

1. **Compute a canonical cover** for F. In this case, the canonical cover for F is {A → BCD, BC → E, B → D, D → A}.
2. **Create a relation schema** Ri for each dependency in the canonical cover. This gives us the following relation schemas: R1(A, B, C, D), R2(B, C, E), R3(B, D), and R4(D, A).
3. **Check if any relation schema** contains a candidate key for R. If not, create a new relation schema that contains a candidate key for R. In this case, none of the relation schemas contain a candidate key for R. A candidate key for R is {B, C}, so we create a new relation schema R5(B, C).
4. **Remove any redundant relation schemas** from the decomposition. A relation schema Ri is redundant if all its dependencies are also dependencies of another relation schema Rj. In this case, we can see that the dependencies of R3 are also dependencies of R1 and R2, so we can remove R3 from the decomposition.

This gives us the final 3NF decomposition of R: {R1(A, B, C, D), R2(B, C, E), R4(D, A), R5(B, C)}.

As R1 still have A->BCD and B->CDE, so R1 will be (A,B,C)

(A,B,C) (B,D,E), (A,D), (A,G)

e. Give a BCNF decomposition of the given schema using the original set *F* of functional dependencies.

A relation schema R is in Boyce-Codd normal form (BCNF) if for every nontrivial functional dependency X → Y in F, X is a superkey. To decompose R into BCNF using the original set F of functional dependencies, we can follow these steps:

1. **Find a dependency** in F that violates the BCNF condition. In this case, the dependency D → A violates the BCNF condition because D is not a superkey.
2. **Decompose R** into two relation schemas R1 and R2 such that the intersection of R1 and R2 is the determinant of the violating dependency and the union of R1 and R2 is equal to R. In this case, we can decompose R into R1(D, A) and R2(D, B, C, E, G).
3. **Repeat the process** for each relation schema in the decomposition until all relation schemas are in BCNF. In this case, we can see that R1 is already in BCNF because D → A is the only nontrivial functional dependency in F+ that holds on R1 and D is a superkey for R1. For R2, we can see that the dependency B → D violates the BCNF condition because B is not a superkey for R2. So we can decompose R2 into R3(B, D) and R4(B, C, E, G). Both R3 and R4 are now in BCNF because there are no nontrivial functional dependencies in F+ that hold on either relation schema.

This gives us the final BCNF decomposition of R: {R1(D, A), R3(B, D), R4(B, C, E, G)}.

(A,B,C,D) (A,E) (A,G)