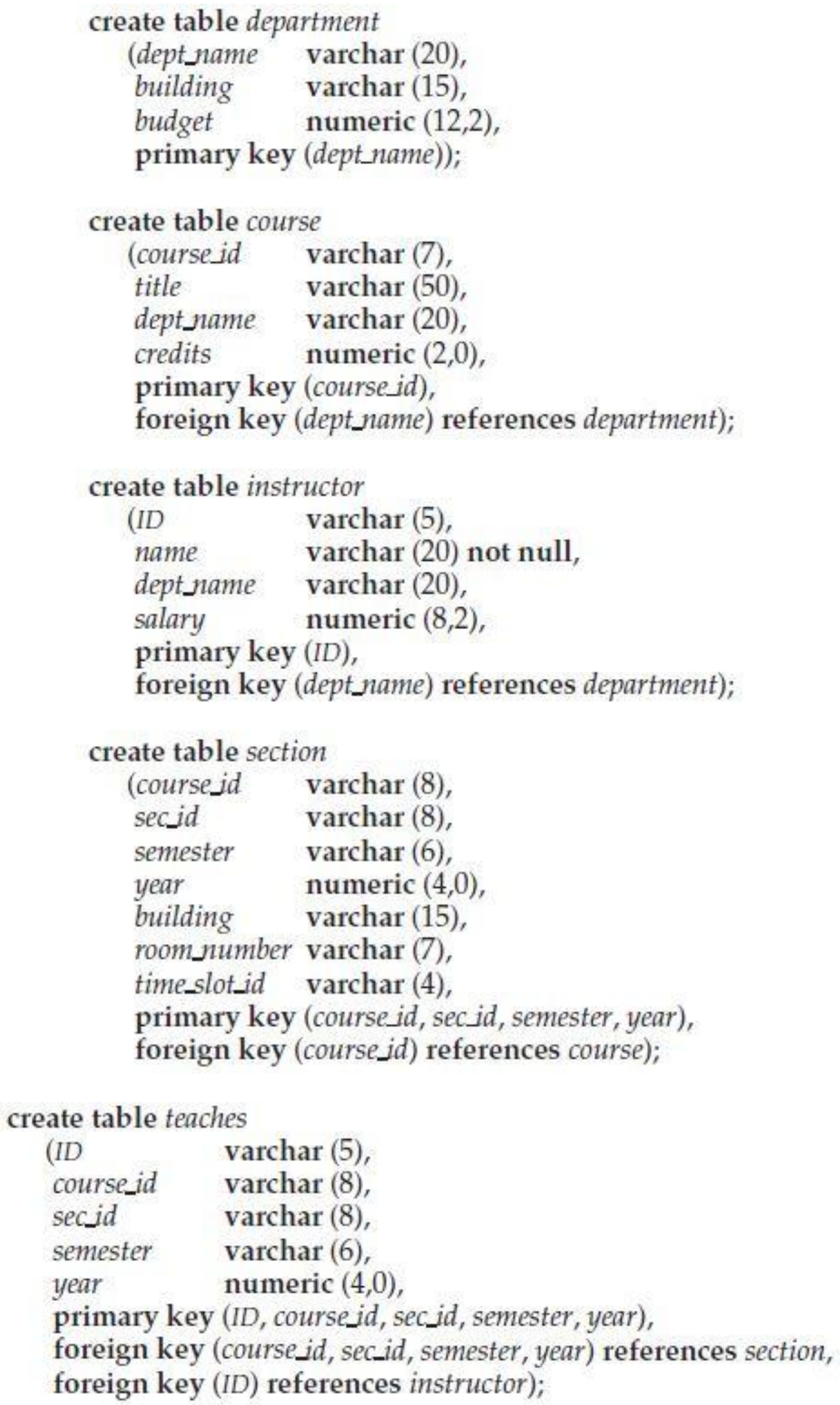
Database Management Systems

Database System Concepts, Sixth Edition,
Abraham Silberschatz, Henry F. Korth, S. Sudarshan



```
create table department
    (dept_name    varchar (20),
     building     varchar (15),
     budget       numeric (12,2),
     primary key (dept_name));

create table course
    (course_id    varchar (7),
     title        varchar (50),
     dept_name    varchar (20),
     credits      numeric (2,0),
     primary key (course_id),
     foreign key (dept_name) references department);

create table instructor
    (ID           varchar (5),
     name         varchar (20) not null,
     dept_name    varchar (20),
     salary       numeric (8,2),
     primary key (ID),
     foreign key (dept_name) references department);

create table section
    (course_id    varchar (8),
     sec_id       varchar (8),
     semester     varchar (6),
     year         numeric (4,0),
     building     varchar (15),
     room_number  varchar (7),
     time_slot_id varchar (4),
     primary key (course_id, sec_id, semester, year),
     foreign key (course_id) references course);

create table teaches
    (ID           varchar (5),
     course_id    varchar (8),
     sec_id       varchar (8),
     semester     varchar (6),
     year         numeric (4,0),
     primary key (ID, course_id, sec_id, semester, year),
     foreign key (course_id, sec_id, semester, year) references section,
     foreign key (ID) references instructor);
```

Figure 3.1 SQL data definition for part of the university database.

Lab Assignment – 1

(Chapter 3: Introduction to SQL)

3.1 Write the following queries in SQL, using the university schema. (We suggest you actually run these queries on a database, using the sample data that we provide on the Web site of the book, db-book.com. Instructions for setting up a database, and loading sample data, are provided on the above Web site.)

a. Find the titles of courses in the Comp. Sci. department that have 3 credits.

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

c. Find the highest salary of any instructor.

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

e. Find the enrollment of each section that was offered in Autumn 2009.

f. Find the maximum enrollment, across all sections, in Autumn 2009.

g. Find the sections that had the maximum enrollment in Autumn 2009.

person (*driver_id*, *name*, *address*)
car (*license*, *model*, *year*)
accident (*report_number*, *date*, *location*)
owns (*driver_id*, *license*)
participated (*report_number*, *license*, *driver_id*, *damage_amount*)

Figure 3.18 Insurance database for Exercises 3.4 and 3.14.

3.2 Suppose you are given a relation *grade_points*(*grade*, *points*), which provides a conversion from letter grades in the *takes* relation to numeric scores; for example an "A" grade could be specified to correspond to 4 points, an "A−" to 3.7 points, a "B+" to 3.3 points, a "B" to 3 points, and so on. The grade points earned by a student for a course offering (section) is defined as the number of credits for the course multiplied by the numeric points for the grade that the student received.

Given the above relation, and our university schema, write each of the following queries in SQL. You can assume for simplicity that no *takes* tuple has the *null* value for *grade*.

a. Find the total grade-points earned by the student with ID 12345, across all courses taken by the student.

b. Find the grade-point average (*GPA*) for the above student, that is, the total grade-points divided by the total credits for the associated courses.

c. Find the ID and the grade-point average of every student.

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%.

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

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.

3.4 Consider the insurance database of Figure 3.18, where the primary keys are underlined. Construct the following SQL queries for this relational database.

a. Find the total number of people who owned cars that were involved in accidents in 2009.

b. Add a new accident to the database; assume any values for required attributes.

c. Delete the Mazda belonging to "John Smith".

branch($\underline{\text{branch\_name}}$, branch_city, assets)
customer ($\underline{\text{customer\_name}}$, customer_street, customer_city)
loan ($\underline{\text{loan\_number}}$, branch_name, amount)
borrower ($\underline{\text{customer\_name}}$, $\underline{\text{loan\_number}}$)
account ($\underline{\text{account\_number}}$, branch_name, balance)
depositor ($\underline{\text{customer\_name}}$, $\underline{\text{account\_number}}$)

Figure 3.19 Banking database for Exercises 3.8 and 3.15.

3.5 Suppose that we have a relation *marks*(ID, *score*) and we wish to assign grades to students based on the score as follows: grade F if $score < 40$, grade C if $40 \leq score < 60$, grade B if $60 \leq score < 80$, and grade A if $80 \leq score$. Write SQL queries to do the following:

a. Display the grade for each student, based on the *marks* relation.

b. Find the number of students with each grade.

3.6 The SQL **like** operator is case sensitive, but the lower() function on strings can be used to perform case insensitive matching. To show how, write a query that finds departments whose names contain the string "sci" as a substring, regardless of the case.

3.7 Consider the SQL query

> **select distinct** $p.a1$
> **from** $p, r1, r2$
> **where** $p.a1 = r1.a1$ **or** $p.a1 = r2.a1$

Under what conditions does the preceding query select values of $p.a1$ that are either in $r1$ or in $r2$? Examine carefully the cases where one of $r1$ or $r2$ may be empty.

3.8 Consider the bank database of Figure 3.19, where the primary keys are underlined. Construct the following SQL queries for this relational database.

a. Find all customers of the bank who have an account but not a loan.

b. Find the names of all customers who live on the same street and in the same city as "Smith".

c. Find the names of all branches with customers who have an account in the bank and who live in "Harrison".

3.9 Consider the employee database of Figure 3.20, where the primary keys are underlined. Give an expression in SQL for each of the following queries.

a. Find the names and cities of residence of all employees who work for "First Bank Corporation".

> *employee* (*employee_name*, *street*, *city*)
> *works* (*employee_name*, *company_name*, *salary*)
> *company* (*company_name*, *city*)
> *manages* (*employee_name*, *manager_name*)

Figure 3.20 Employee database for Exercises 3.9, 3.10, 3.16, 3.17, and 3.20.

b. Find the names, street addresses, and cities of residence of all employees who work for "First Bank Corporation" and earn more than $10,000.

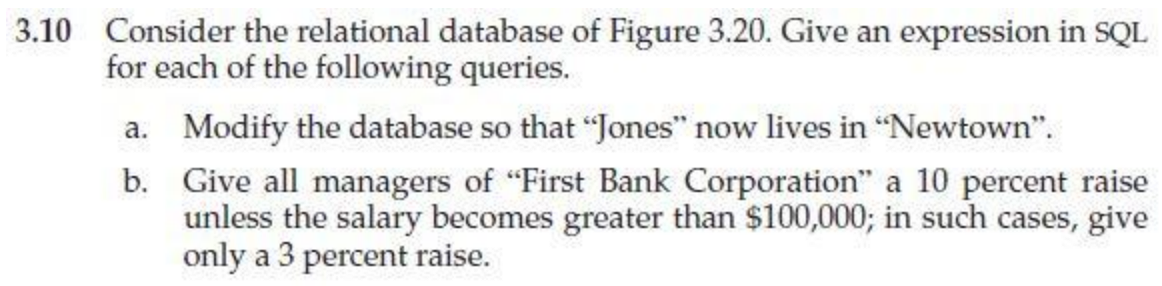
c. Find all employees in the database who do not work for "First Bank Corporation".

d. Find all employees in the database who earn more than each employee of "Small Bank Corporation".

e. Assume that the companies may be located in several cities. Find all companies located in every city in which "Small Bank Corporation" is located.

f. Find the company that has the most employees.

g. Find those companies whose employees earn a higher salary, on average, than the average salary at "First Bank Corporation".



3.10 Consider the relational database of Figure 3.20. Give an expression in SQL for each of the following queries.

a. Modify the database so that "Jones" now lives in "Newtown".

b. Give all managers of "First Bank Corporation" a 10 percent raise unless the salary becomes greater than $100,000; in such cases, give only a 3 percent raise.

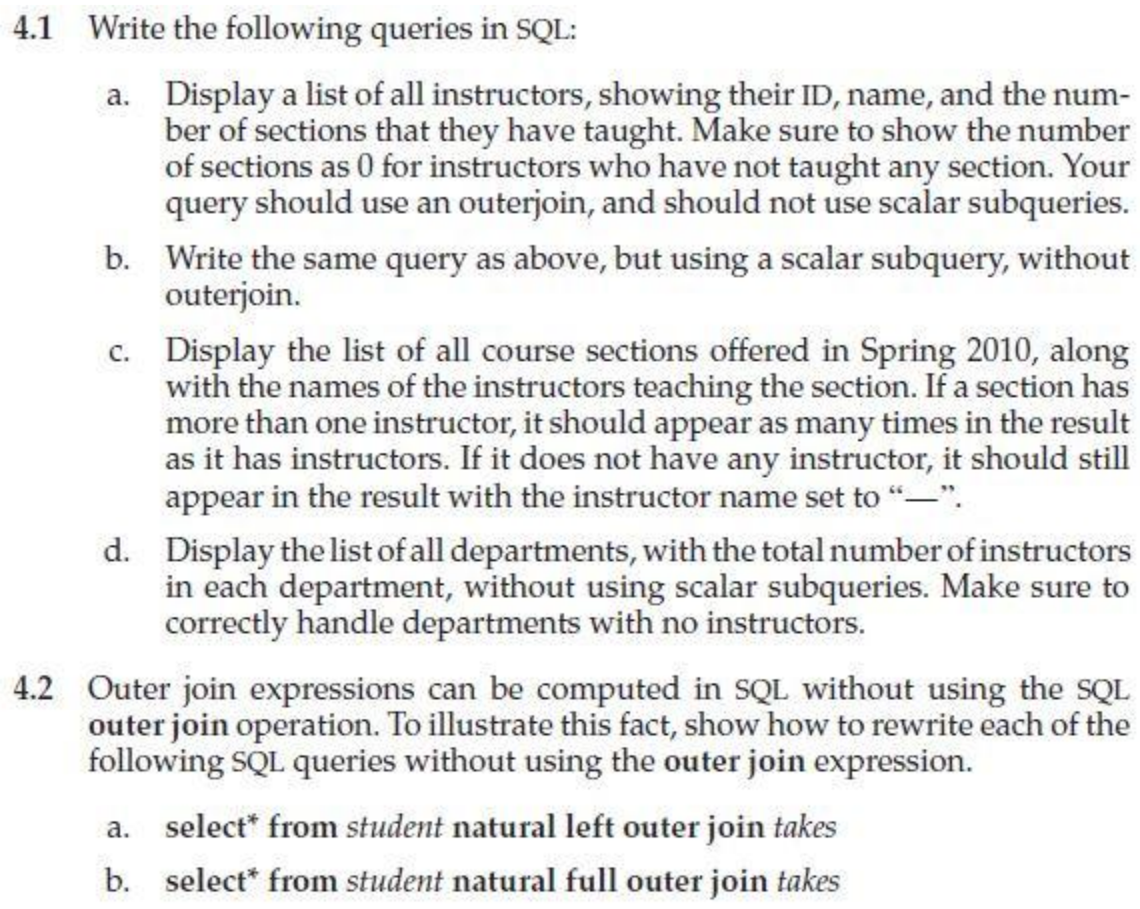
<u>Additional Questions</u>

<u>Basic SQL</u>

1. Find the names of all the instructors from Biology department
2. Find the names of courses in Computer science department which have 3 credits
3. For the student with ID 12345 (or any other value), show all course_id and title of all courses registered for by the student.
4. As above, but show the total number of credits for such courses (taken by that student). Don't display the tot_creds value from the student table, you should use SQL aggregation on courses taken by the student.
5. As above, but display the total credits for each of the students, along with the ID of the student; don't bother about the name of the student. (Don't bother about students who have not registered for any course, they can be omitted)
6. Find the names of all students who have taken any Comp. Sci. course ever (there should be no duplicate names)
7. Display the IDs of all instructors who have never taught a couse (Notesad1) Oracle uses the keyword minus in place of except; (2) interpret "taught" as "taught or is scheduled to teach")
8. As above, but display the names of the instructors also, not just the IDs.
9. You need to create a movie database. Create three tables, one for actors(AID, name), one for movies(MID, title) and one for actor_role(MID, AID, rolename). Use appropriate data types for each of the attributes, and add appropriate primary/foreign key constraints.
10. Insert data to the above tables (approx 3 to 6 rows in each table), including data for actor "Charlie Chaplin", and for yourself (using your roll number as ID).
11. Write a query to list all movies in which actor "Charlie Chaplin" has acted, along with the number of roles he had in that movie.
12. Write a query to list all actors who have not acted in any movie
13. List names of actors, along with titles of movies they have acted in. If they have not acted in any movie, show the movie title as null. (Do not use SQL outerjoin syntax here, write it from scratch.)

Lab Assignment – 2

(Chapter 4: Intermediate SQL)



4.1 Write the following queries in SQL:

a. Display a list of all instructors, showing their ID, name, and the number of sections that they have taught. Make sure to show the number of sections as 0 for instructors who have not taught any section. Your query should use an outerjoin, and should not use scalar subqueries.

b. Write the same query as above, but using a scalar subquery, without outerjoin.

c. Display the list of all course sections offered in Spring 2010, along with the names of the instructors teaching the section. If a section has more than one instructor, it should appear as many times in the result as it has instructors. If it does not have any instructor, it should still appear in the result with the instructor name set to "—".

d. Display the list of all departments, with the total number of instructors in each department, without using scalar subqueries. Make sure to correctly handle departments with no instructors.

4.2 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*

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

4.3 Suppose we have three relations $r(A, B)$, $s(B, C)$, and $t(B, D)$, with all attributes declared as **not null**. Consider the expressions

- r **natural left outer join** (s **natural left outer join** t), and
- (r **natural left outer join** s) **natural left outer join** t

a. Give instances of relations r, s and t such that in the result of the second expression, attribute C has a null value but attribute D has a non-null value.

b. Is the above pattern, with C null and D not null possible in the result of the first expression? Explain why or why not.

4.4 Testing SQL queries: To test if a query specified in English has been correctly written in SQL, the SQL query is typically executed on multiple test databases, and a human checks if the SQL query result on each test database matches the intention of the specification in English.

a. In Section 3.3.3 we saw an example of an erroneous SQL query which was intended to find which courses had been taught by each instructor; the query computed the natural join of *instructor*, *teaches*, and *course*, and as a result unintentionally equated the *dept_name* attribute of *instructor* and *course*. Give an example of a dataset that would help catch this particular error.

b. When creating test databases, it is important to create tuples in referenced relations that do not have any matching tuple in the referencing relation, for each foreign key. Explain why, using an example query on the university database.

c. When creating test databases, it is important to create tuples with null values for foreign key attributes, provided the attribute is nullable (SQL allows foreign key attributes to take on null values, as long as they are not part of the primary key, and have not been declared as **not null**). Explain why, using an example query on the university database.

Hint: use the queries from Exercise 4.1.

4.5 Show how to define the view *student_grades* (*ID*, *GPA*) giving the grade-point average of each student, based on the query in Exercise 3.2; recall that we used a relation *grade_points*(*grade*, *points*) to get the numeric points associated with a letter grade. Make sure your view definition correctly handles the case of *null* values for the *grade* attribute of the *takes* relation.

4.6 Complete the SQL DDL definition of the university database of Figure 4.8 to include the relations *student*, *takes*, *advisor*, and *prereq*.

employee (*employee_name*, *street*, *city*)
works (*employee_name*, *company_name*, *salary*)
company (*company_name*, *city*)
manages (*employee_name*, *manager_name*)

Figure 4.11 Employee database for Figure 4.7 and 4.12.

4.7 Consider the relational database of Figure 4.11. Give an SQL DDL definition of this database. Identify referential-integrity constraints that should hold, and include them in the DDL definition.

4.8 As discussed in Section 4.4.7, we expect the constraint "an instructor cannot teach sections in two different classrooms in a semester in the same time slot" to hold.

a. Write an SQL query that returns all (*instructor*, *section*) combinations that violate this constraint.

b. Write an SQL assertion to enforce this constraint (as discussed in Section 4.4.7, current generation database systems do not support such assertions, although they are part of the SQL standard).

4.9 SQL allows a foreign-key dependency to refer to the same relation, as in the following example:

```
create table manager
   (employee_name   varchar(20)  not null
    manager_name    varchar(20)  not null,
    primary key employee_name,
    foreign key (manager_name) references manager
                               on delete cascade )
```

Here, *employee_name* is a key to the table *manager*, meaning that each employee has at most one manager. The foreign-key clause requires that every manager also be an employee. Explain exactly what happens when a tuple in the relation *manager* is deleted.

4.10 SQL provides an n-ary operation called **coalesce**, which is defined as follows: **coalesce**($A_1, A_2, \ldots, A_n$) returns the first nonnull A_i in the list $A_1, A_2, \ldots, A_n$, and returns *null* if all of $A_1, A_2, \ldots, A_n$ are *null*.

Let a and b be relations with the schemas A(*name, address, title*), and B(*name, address, salary*), respectively. Show how to express a **natural full outer join** b using the **full outer-join** operation with an **on** condition and the **coalesce** operation. Make sure that the result relation does not contain two copies of the attributes *name* and *address*, and that the solution is correct even if some tuples in a and b have null values for attributes *name* or *address*.

salaried_worker (*name, office, phone, salary*)
hourly_worker (*name, hourly_wage*)
address (*name, street, city*)

Figure 4.12 Employee database for Exercise 4.16.

4.11 Some researchers have proposed the concept of *marked* nulls. A marked null $\perp_i$ is equal to itself, but if $i \neq j$, then $\perp_i \neq \perp_j$. One application of marked nulls is to allow certain updates through views. Consider the view *instructor_info* (Section 4.2). Show how you can use marked nulls to allow the insertion of the tuple (99999, "Johnson", "Music") through *instructor_info*.

Additional Question

Intermediate SQL

Using the university schema that you have created, write the following queries. In some cases you need to insert extra data to show the effect of a particular feature -- this is indicated with the question. You should then show not only the query, but also the insert statements to add the required extra data.

1. Find the maximum and minimum enrollment across all sections, considering only sections that had some enrollment, don't worry about those that had no students taking that section
2. Find all sections that had the maximum enrollment (along with the enrollment), using a subquery.
3. As in in Q1, but now also include sections with no students taking them; the enrollment for such sections should be treated as 0. Do this in two different ways (and create require data for testing)
    1. Using a scalar subquery
    2. Using aggregation on a left outer join (use the SQL natural left outer join syntax)
4. Find all courses whose identifier starts with the string "CS-1"
5. Find instructors who have taught all the above courses
    1. Using the "not exists ... except ..." structure
    2. Using matching of counts which we covered in class (don't forget the distinct clause!).
6. Insert each instructor as a student, with tot_creds = 0, in the same department
7. Now delete all the newly added "students" above (note: already existing students who happened to have tot_creds = 0 should not get deleted)
8. Some of you may have noticed that the tot_creds value for students did not match the credits from courses they have taken. Write and execute query to update tot_creds based on the credits passed, to bring the database back to consistency. (This query is provided in the book/slides.)
9. Update the salary of each instructor to 10000 times the number of course sections they have taught.
10. Create your own query: define what you want to do in English, then write the query in SQL. Make it as difficult as you wish, the harder the better.

Lab Assignment – 3

(Chapter 5: Advanced SQL)

5.2 Write a Java function using JDBC metadata features that takes a `ResultSet` as an input parameter, and prints out the result in tabular form, with appropriate names as column headings.

5.3 Write a Java function using JDBC metadata features that prints a list of all relations in the database, displaying for each relation the names and types of its attributes.

5.4 Show how to enforce the constraint "an instructor cannot teach in two different classrooms in a semester in the same time slot." using a trigger (remember that the constraint can be violated by changes to the *teaches* relation as well as to the *section* relation).

5.5 Write triggers to enforce the referential integrity constraint from *section* to *time_slot*, on updates to *section*, and *time_slot*. Note that the ones we wrote in Figure 5.8 do not cover the **update** operation.

5.6 To maintain the *tot_cred* attribute of the *student* relation, carry out the following:

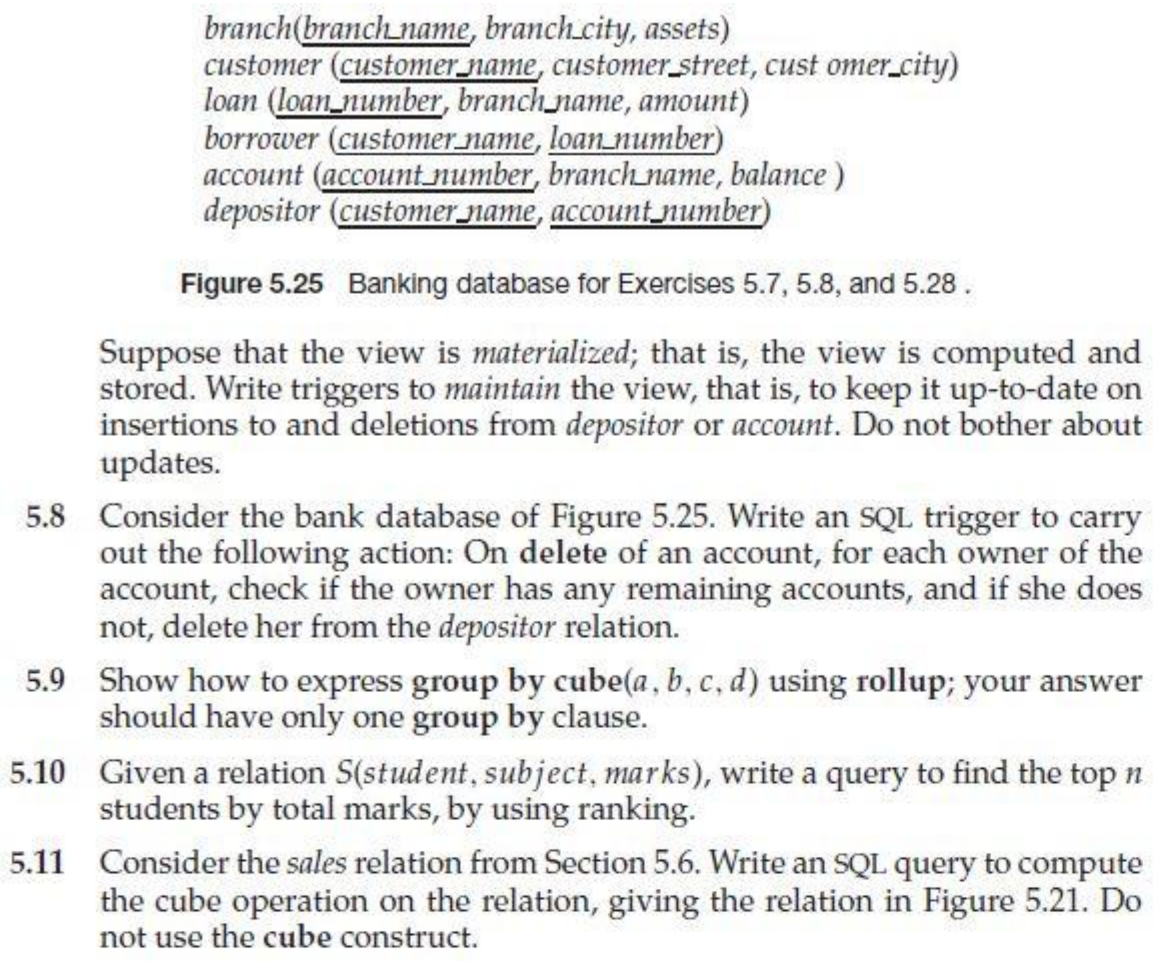
a. Modify the trigger on updates of *takes*, to handle all updates that can affect the value of *tot_cred*.

b. Write a trigger to handle inserts to the *takes* relation.

c. Under what assumptions is it reasonable not to create triggers on the *course* relation?

5.7 Consider the bank database of Figure 5.25. 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
```



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

Figure 5.25 Banking database for Exercises 5.7, 5.8, and 5.28 .

Suppose that the view is *materialized*; 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 and deletions from *depositor* or *account*. Do not bother about updates.

5.8 Consider the bank database of Figure 5.25. Write an SQL trigger to carry out the following action: On **delete** of an account, for each owner of the account, check if the owner has any remaining accounts, and if she does not, delete her from the *depositor* relation.

5.9 Show how to express **group by cube**(a, b, c, d) using **rollup**; your answer should have only one **group by** clause.

5.10 Given a relation $S(student, subject, marks)$, write a query to find the top n students by total marks, by using ranking.

5.11 Consider the *sales* relation from Section 5.6. Write an SQL query to compute the cube operation on the relation, giving the relation in Figure 5.21. Do not use the **cube** construct.

Additional Questions

Advanced SQL

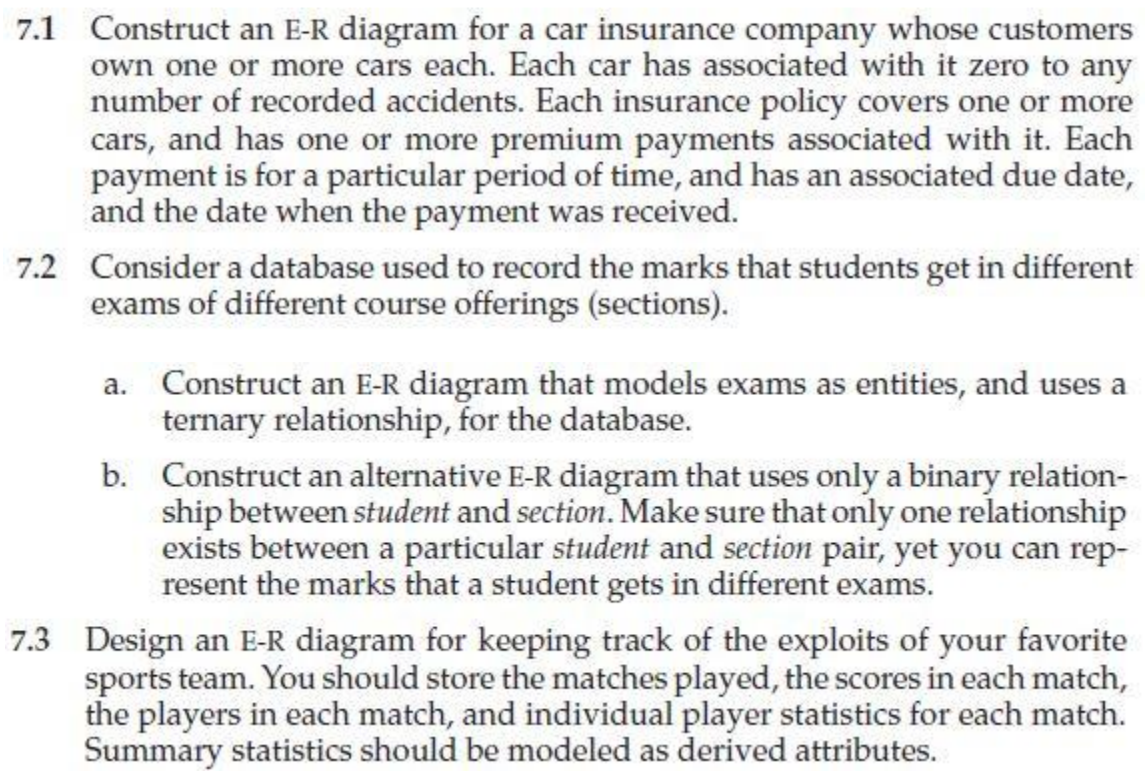
In this assignment, you will write more complex SQL queries, using the usual schema by default. Again, you have to add required data to test your queries, and must include the SQL statements to create the data along with your queries.

1. The university rules allow an F grade to be overridden by any pass grade (A, B, C, D). Now, create a view that lists information about all fail grades that have not been overridden (the view should contain all attributes from the takes relation).
2. Find all students who have 2 or more non-overridden F grades as per the takes relation, and list them along with the F grades.
3. Grades are mapped to a grade point as follows: A:10, B:8, C:6, D:4 and F:0. Create a table to store these mappings, and write a query to find the CPI of each student, using this table. Make sure students who have not got a non-null grade in any course are displayed with a CPI of null.
4. Find all rooms that have been assigned to more than one section at the same time. Display the rooms along with the assigned sections; I suggest you use a with clause or a view to simplify this query.
5. Create a view faculty showing only the ID, name, and department of instructors.

6. Create a view CSinstructors, showing all information about instructors from the Comp. Sci. department.
7. Insert appropriate tuple into each of the views faculty and CSinstructors, to see what updates your database allows on views; explain what happens.
8. Grant permission to one of your friends to view all data in your student relation.
9. Now grant permission to all users to see all data in your faculty view. Conversely, find a friend who has granted you permission on their faculty view, and execute a select query on that view.

<u>Lab Assignment – 4</u>

<u>(Chapter 7: E-R Model)</u>



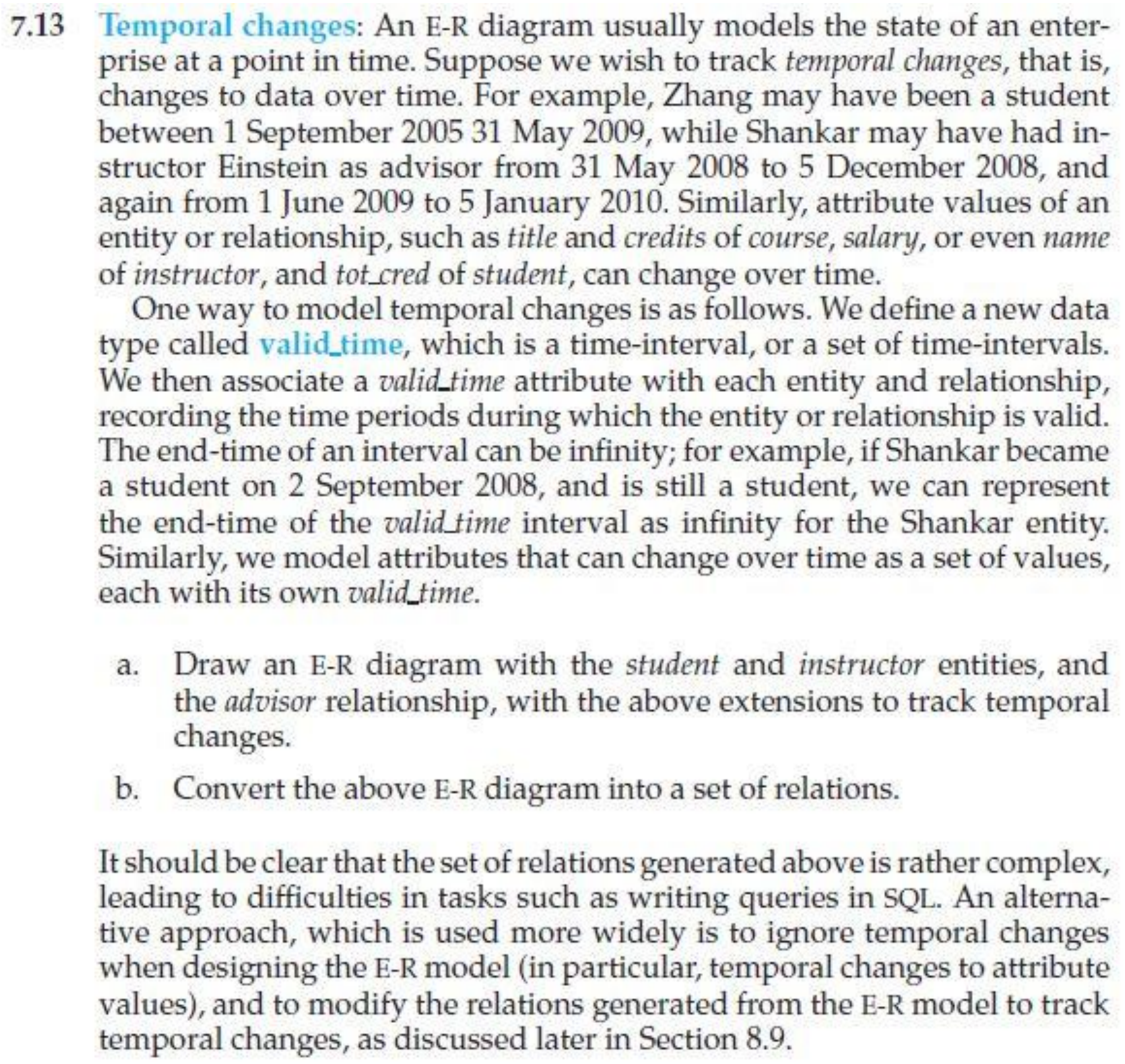
7.1 Construct an E-R diagram for a car insurance company whose customers own one or more cars each. Each car has associated with it zero to any number of recorded accidents. Each insurance policy covers one or more cars, and has one or more premium payments associated with it. Each payment is for a particular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

7.2 Consider a database used to record the marks that students get in different exams of different course offerings (sections).

a. Construct an E-R diagram that models exams as entities, and uses a ternary relationship, for the database.

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.

7.3 Design an E-R diagram for keeping track of the exploits of your favorite sports team. You should store the matches played, the scores in each match, the players in each match, and individual player statistics for each match. Summary statistics should be modeled as derived attributes.



7.13 **Temporal changes**: An E-R diagram usually models the state of an enterprise at a point in time. Suppose we wish to track *temporal changes*, that is, changes to data over time. For example, Zhang may have been a student between 1 September 2005 31 May 2009, while Shankar may have had instructor Einstein as advisor from 31 May 2008 to 5 December 2008, and again from 1 June 2009 to 5 January 2010. Similarly, attribute values of an entity or relationship, such as *title* and *credits* of *course*, *salary*, or even *name* of *instructor*, and *tot_cred* of *student*, can change over time.

One way to model temporal changes is as follows. We define a new data type called **valid_time**, which is a time-interval, or a set of time-intervals. We then associate a *valid_time* attribute with each entity and relationship, recording the time periods during which the entity or relationship is valid. The end-time of an interval can be infinity; for example, if Shankar became a student on 2 September 2008, and is still a student, we can represent the end-time of the *valid_time* interval as infinity for the Shankar entity. Similarly, we model attributes that can change over time as a set of values, each with its own *valid_time*.

a. Draw an E-R diagram with the *student* and *instructor* entities, and the *advisor* relationship, with the above extensions to track temporal changes.

b. Convert the above E-R diagram into a set of relations.

It should be clear that the set of relations generated above is rather complex, leading to difficulties in tasks such as writing queries in SQL. An alternative approach, which is used more widely is to ignore temporal changes when designing the E-R model (in particular, temporal changes to attribute values), and to modify the relations generated from the E-R model to track temporal changes, as discussed later in Section 8.9.