# Selected Past CS319 and CS253 Exam Questions

Hugh Darwen

*These questions from former CS319 and CS253 exams were set on my lectures on Temporal Data, The Askew Wall, and How To Handle Missing Information Without Using NULL.*

## A. Temporal Data

### Question 1

A relation variable (relvar) `RoomAvailability` is declared to record the times during which each of the university's lecture rooms is available for teaching purposes, on which dates. Here is how it is declared (in **Temporal Tutorial D**):

```
VAR RoomAvailability BASE RELATION
    { Room#  CHAR,
      Date   DATE
      During INTERVAL_ToD }
    USING During ◂ KEY { Room#, Date, During } ▸ ;
```

You may assume that a value of type `ToD` represents a given time of day, on a scale of one minute.

(a) Complete the following predicate for `RoomAvailability`:

Room `Room#` is or was available for teaching purposes on `Date` ... [2]

(b) Write down the constraint declarations that need to be added to the relvar declaration below, which records lectures that have been or are scheduled to be given. Your constraints should ensure that:

- no two lectures can be in progress in the same room at the same time;
- no member of staff can be giving more than one lecture at the same time; and
- the room allocated for a lecture is indeed available for the duration of that lecture:

```
VAR Lecture BASE RELATION
    { Room#   CHAR,
      Date    DATE,
      Module# CHAR,
      Staff#  CHAR,
      During  INTERVAL_ToD }
    ... ;
```

[3]

(c) Write a **Temporal Tutorial D** expression to yield a relation giving (`Room#, Date, During`) combinations where `During` gives a maximal interval throughout which room `Room#` is or was available but not allocated for a lecture on `Date`. [4]

(d) Give *three* advantages of using interval-typed attributes rather than separate attributes for the beginnings and ends of time intervals? [3]

(e) With reference to operators involving intervals and relations with interval attributes, discuss the justification for *quantisation*, under which the concept of *unit interval* is defined for each interval type, according to the *scale* of that interval type. [4]

(f) `Lecture` is not in sixth normal form (6NF). Show how it could be decomposed to give an equivalent 6NF design. State, giving your reasons, whether you would recommend this alternative design. [4]

(g) It is proposed to extend the database by showing which members of staff were or are currently appointed as lecturers on which modules. For appointments that are not current, the start and end times are both recorded, but no attempt is to be made to estimate when any current appointment is to terminate. Describe the four methods that have been advanced for recording such information and comment briefly (2-3 sentences) on their respective merits. [5]

**Question 2**

(a) Which of the following comparisons, each of two integer intervals, yields *true*?

(i) `[22:24] OVERLAPS [19:22]`

(ii) `(3:9] MEETS [-1:3)`

(iii) `(-4:9] BEGINS [-3:10)`

(iv) `(-14:-9] BEFORE (-9:-4)`

(b) Write down, using closed-closed notation for each element, the collapsed form of the set consisting of each of the intervals appearing in part (a). Recall that closed-closed notation uses square brackets for both bounds, as shown in (a)(i).

(c) Boris and Cindy, both studying computer science in the same year at Warwick University, live together and have just one computer. They are planning their revision for the various exams they have to take later this term. They decide to use a **Temporal Tutorial D** (**TTD**) database to record their revision schedule. The version of **TTD** they are using supports a type, `HOUR`, whose values are times on a scale of 1 hour; e.g., 13:00 on 16/06/2007. The type `INTERVAL_HOUR` therefore represents intervals of type `HOUR`, such as the 2-hour interval [13:00 on 16/06/2007 : 14:00 on 16/06/2007].

A *revision period* is defined to show exactly when a student (Boris or Cindy) plans to start revising for a particular exam, and when to stop. For example, Boris might plan to start a revision period for the CS319 exam at 6:00 on 24/05/2007, ending in time for

breakfast at 9:00 the same morning. Cindy might plan to revise for CS301 from 8:00 to 10:00 that same day, then at 11:00 switch to revising for CS319.

A student cannot be revising for more than one exam at the same time. Boris and Cindy can be revising at the same time but they cannot both be using the computer at the same time. Therefore the database needs to show when the computer is in use, and by whom, for revision.

The last revision period for a particular exam must obviously be over in time for that exam. Therefore the database has to include a relvar showing the schedule for the relevant exams. Here is its **TTD** definition:

```
VAR exam BASE RELATION { ModuleId  CHAR,
                         StartTime HOUR }
                   KEY { ModuleId } ;
```

Predicate for `exam`: The exam for module *ModuleId* starts at *StartTime*.

The relvars for revision periods and computer use look like this:

```
VAR revision BASE RELATION { Sname    CHAR,
                             ModuleId CHAR,
                             during   INTERVAL_HOUR }
                   KEY { Sname, during } ;

CONSTRAINT before_exam IS_EMPTY ( ( revision JOIN exam
) WHERE END(during) >= StartTime ) ;

VAR pcinuse  BASE RELATION { Sname    CHAR,
                             during   INTERVAL_HOUR }
                   KEY { ALL BUT } ;
```

(Recall that `{ ALL BUT }` denotes all the attributes of the relevant relation or relvar.)

Predicate for `revision`: Student *Sname* is revising for module *ModuleId*, starting at the beginning of *during* and finishing at the end of *during*.

Predicate for `pcinuse`: The computer is being used for revision by student *Sname*, starting at the beginning of *during* and finishing at the end of *during*.

(i) What are the so-called problems of *redundancy, circumlocution,* and *contradiction* that are observed in connection with temporal data? You may use Boris and Cindy's database as an example to illustrate your explanations.

(ii) The constraints defined so far do not meet the stated requirements. In particular:

- they permit the same person to be revising more than one module at the same time;
- they permit both students to be using the computer for revision at the same time;

- they permit the computer to be allocated at a time when neither student is revising; and
- they permit the relvars `revision` and `pcinuse` to suffer from both redundancy and circumlocution.

What constraints should be written in place of and in addition to the existing `KEY` constraints? (Indications will do if you can't give exact **TTD** syntax for these.)

(iii) Write **TTD** expressions for the following queries:

- Show for each student *Sname* intervals throughout which *Sname* is revising.
- Show intervals throughout which some student is revising but the computer is not in use.

(d) As exam time draws near Boris and Cindy get a bit panicky and decide to put in some extra, unplanned revision periods. When they start these periods they don't even know how long they will last, but as soon as they start they want the database to record the fact that they are currently revising, and the exam they are revising for. When they finish such an *ad hoc* revision period, they update the database again to reflect that. Suggest:

(i) How the existing database might be used for this purpose without any change at all.

(ii) A suitable extension to the database for the same purpose, addressing any problems that you perceive with your solution (i).

**Question 3**

(a) According to Wikipedia's entry for **temporal database,** a temporal table "gets two extra fields, Valid-From and Valid-To" as compared with its nontemporal (current state only) counterpart. Most authorities actually recommend just one "extra field"—in relational terms, a single attribute. For example, if Wikipedia's extra fields would be of type `DATE`, the declared type of the single attribute preferred by most authorities would be one representing intervals of dates, perhaps named `INTERVAL_DATE`.

(i) Give three advantages of the proposed single attribute over Wikipedia's "two extra fields".

(ii) List the dates contained in the interval

```
INTERVAL_DATE ( [ DATE ('01 Jan 2008'):
                  DATE ('04 Jan 2008') ) )
```

(iii) What is meant by the *point type* of an interval type? What properties must a type have to qualify as a point type?

(b) Consider the relation `Rab` represented by the following table:

| A | B |
|---|---|
| 1 | [1:4] |
| 1 | [4:6) |
| 1 | [7:7] |

Using similar tabular notation, give the result of

(i) `PACK Rab ON B`

(ii) `UNPACK Rab ON B`

(iii)
```
USING(B) ◂ Rab NOT MATCHING
RELATION { TUPLE { A 1, B INTERVAL_INTEGER((3:4]) }
} ▸
```

(c) Consider the following **Tutorial D** relvar definition:

```
VAR xyHistory BASE RELATION { x CHAR, y
INTERVAL_DATE }
                              KEY { x, y } ;
```

representing intervals `y` throughout which person `x` was in full-time education.

Describe the problems of *redundancy, circumlocution,* and *contradiction* exhibited by this definition, and how those problems could be addressed by inclusion of suitable `PACKED ON, WHEN/THEN` and/or "U_key" constraints.

**Question 4**

I've started to design a relational database in which to record my family tree as far back as I can trace it. Here is what I've got so far (in **Temporal Tutorial D**):

```
VAR person BASE RELATION { name CHAR,
                           gender CHAR,
                           during INTERVAL_DATE }
                 KEY { name } ;
```

Predicate for `person`: Person ***name*** of gender ***gender*** was alive from the beginning to the end of ***during***.

```
CONSTRAINT m_or_f person { gender } ⊆
                  RELATION {TUPLE { gender 'F' },
                            TUPLE { gender 'M' } } ;

VAR parent_of BASE RELATION { parent CHAR,
                              child CHAR }
                 KEY { parent, child } ;
```

Predicate for `parent_of`: Person ***parent*** was the mother or father of ***child***.

```
CONSTRAINT one_of_each
   COUNT ( parent_of RENAME ( parent AS name ) JOIN person
           WHERE gender = 'F' ) = 1 AND
   COUNT ( parent_of RENAME ( parent AS name ) JOIN person
           WHERE gender = 'M' ) = 1 ) ;

VAR marriage BASE RELATION { wife CHAR,
                             husband CHAR,
                             during INTERVAL_DATE }
                   KEY { wife, husband } ;
```

Predicate for `marriage`: Female person ***wife*** was married to male person ***husband*** from the beginning to the end of ***during***.

```
CONSTRAINT hetero
   marriage { wife } ⊆
   RELATION { TUPLE { gender 'F' } } AND
   marriage { husband } ⊆
   RELATION { TUPLE { gender 'M' } } ;
```

(a) I notice that the Wikipedia entry for **temporal database** would suggest a `to` and `from` attribute, each of type `DATE`, instead of my single attribute `during` in relvars `person` and `marriage`. Should I switch to Wikipedia's suggestion? If so, why? If not, why not?

(b) I'm not happy with the constraints defined for marriage:

- they permit the same person to be married to more than one other person at the same time (which I regard as impossible);
- they don't restrict marriages to the lifetimes of the people involved; and
- I might discover a couple that got divorced and later remarried—my existing `KEY` constraint wouldn't permit that.

What constraints should I write in place of the existing `KEY` constraint? (Indications will do if you can't give me exact **Temporal Tutorial D** syntax for these.)

(c) How could I write a constraint to make sure everybody's mother was alive on their date of birth? (Again, an indication will do if you can't give me exact **Temporal Tutorial D** syntax.)

(d) I want to extend my definition to permit inclusion of family members who are still alive. Please suggest some ways in which I can achieve that and point out for me their relative advantages and disadvantages.

## Question 5

(a) Assume that type `OneTwoThree` is defined in **Tutorial D** to consist of just three values, the integers 1, 2, and 3.

(i) What are the properties of `OneTwoThree`, in **Tutorial D** with added support for interval types, that enable us to assume that type `INTERVAL_OneTwoThree` also exists? **2 marks**

(ii) How many distinct values are there of type `INTERVAL_OneTwoThree`? **1 mark**

(iii) *i1* and *i2* are values of type `INTERVAL_OneTwoThree` such that *i1* `MERGES` *i2* and *i1* `PRECEDES` *i2* are both false. What are *i1* and *i2*? **1 mark**

(iv) Explain why there is only one value of type `INTERVAL_OneTwoThree` that can be denoted using open-open notation for intervals. **1 mark**

(b) Consider the relation `Rab` represented by the following table:

| A | B |
|---|---|
| 1 | [2:4) |
| 1 | [4:6) |
| 1 | [0:1) |

Using similar tabular notation, and the same closed-open notation for the intervals, give the result of

(i) `PACK Rab ON B` **2 marks**

(ii) `UNPACK Rab ON B` **2 marks**

(iii)
```
USING(B) ◂ Rab JOIN
RELATION { TUPLE { C 2, B INTERVAL_INTEGER((3:5]) } } ▸
```
**2 marks**

(c) Consider the SQL database fragment defined as follows:

```
CREATE TABLE HireHistory ( E# CHAR(5) NOT NULL,
                           HiredOn DATE NOT NULL,
                           LeftOn DATE,
                PRIMARY KEY ( E#, HiredOn ) ;

CREATE TABLE Absence ( E# CHAR(5) NOT NULL,
                       AbsentFrom DATE NOT NULL,
                       AbsentTo DATE NOT NULL,
                       Reason VARCHAR(30) NOT NULL,
             PRIMARY KEY ( E#, AbsentFrom ) ;
```

No other constraints are defined relevant to these two tables.

`HireHistory` records the hire date, `HiredOn`, of every person, `E#`, that has ever been employed by the company and, where applicable, the date (`LeftOn`) on which that hiring was terminated. The key, `(E#, HiredOn)`, is explained by the fact that sometimes a person who has left the company is subsequently re-employed, so there can be more than one row for the same employee. For people currently employed the `LeftOn` values are recorded as `NULL`.

`Absence` records, for each employee, all of that employee's periods of absence from work during a period of employment. It includes planned future absences of current employees, as well as actual past periods of absence. In each case, the reason for the absence (such as sickness or vacation) is recorded.

(i) The lack of constraints, apart from those shown, exposes this design to several problems concerning integrity. Describe these problems. **6 marks**

(ii) Using **Tutorial D** with temporal extensions, give `VAR` declarations and any additional `CONSTRAINT` declarations that might be needed, to replace those SQL `CREATE TABLE` statements and address the problems you identified in (i). You may assume that type `DATE` is available. **5 marks**

(iii) Explain how your solution to (ii) represents current employees (i.e., those for whom the `LeftOn` value is `NULL` in the SQL implementation). Briefly describe an alternative approach to the same problem. *Note:* For marking we treat the two approaches as equally valid; and you are not asked to give your reasons for preferring the approach you chose. **4 marks**

## B. The Askew Wall

### Question 6

Consider the following list of DBMS features prescribed by E.F. Codd's relational model of data (1970).

(a) The DBMS provides uniformity of data structure and method of access. [2]
(b) There is a value for every attribute in every tuple of every relation. [4]
(c) The same tuple cannot appear more than once in a relation. [4]
(d) There is no significance to the ordering of the attributes of a relation. [4]
(e) There is no significance to the ordering of the tuples of a relation. [2]
(f) No attribute value shall be a pointer. [3]
(g) All queries and constraints are to be expressed declaratively, not procedurally. [3]
(h) Types are orthogonal to relations. [3]

Assess SQL's adherence to or deviation from each of these prescriptions. In each case justify your claim with reference to specific operators and/or statement types and, where SQL deviates, briefly mention any adverse or advantageous consequences of the deviation. Where appropriate, you are encouraged to give at least one complete example in SQL syntax (minor syntax errors will not be penalised).

### Question 7

(a) `T` is an SQL base table with columns `A, B,` and `C.`

(i) Write an SQL query to yield a table equivalent to the relational projection of `T` over `A` and `B` except that the columns of the result are both named `X`.

(ii) Explain in a single sentence why your query represents a deviation from relational theory.

(iii) What problems can arise from the fact that your query is acceptable in SQL? (Think of operations that in **Tutorial D** you could perform on any relation but in SQL could not be performed on this table.)

(b) `T1` is an SQL base table whose definition is not given except that it has columns named `A,` and `B` of type `INTEGER` and `C1` and `C2` of type `VARCHAR(5).` It is updated nightly. Each of the following observations were obtained at 9:00am on different days. In each case give a possible explanation for the phenomenon observed and state any deviation from relational theory by SQL that is involved in your explanation.

(i) The result of `SELECT COUNT(*) AS T1_count FROM T1` is

| T1_count |
|---|
| 100 |

And yet the result of

```
SELECT COUNT(*) AS U_count FROM
  ( SELECT * FROM T1 UNION SELECT * FROM T1 ) AS U
```

is:

| U_count |
|---|
| 1 |

(ii) `SELECT * FROM T1 UNION SELECT * FROM T1`
yields a result containing several rows and yet
`SELECT * FROM ( T1 NATURAL JOIN T1 ) AS J`
yields an empty table.

(iii) `SELECT * FROM T1 UNION SELECT * FROM T1`
yields a result consisting of just row and yet
`SELECT * FROM ( T1 NATURAL JOIN T1 ) AS J`
yields one consisting of four rows.

(iv)
```
SELECT * FROM T1
WHERE C1 = C2 AND (C1 || C2) <> (C2 || C1)
```
yields a table containing several rows.
`||` is the string concatenation operator in SQL
e.g., `'SQ' || 'L' = 'SQL'`.

(v)
```
SELECT SUM( A ) + SUM( B ) AS SUMAB1,
       SUM( A + B ) AS SUMAB2
FROM T1
```
yields:

| SUMAB1 | SUMAB2 |
|---|---|
| 200 | 100 |

(c) `r1` is a relvar whose definition is not given apart from the fact that it has an attribute named `a` and does not have an attribute named `x`.

(i) Write a **Tutorial D** expression to yield a relation that is identical to `r1` except that the attribute name `x` appears in place of the attribute name `a`.

(ii) Now suppose you are asked to write an equivalent SQL expression. Explain how you would approach this task.

(iii) Suggest an improvement to SQL to make this task a little easier.

**Question 8**

(a) Define the terms *heading* and *body,* as applied to relations in relational database theory. Describe the various ways in which tables resulting from evaluation of SQL queries can

have headings or bodies that deviate from these definitions. (It will be helpful to number your deviations, for ease of reference in your solution to part (b)).

(b) For each of the following queries explain whether any of SQL's deviations referred to in part (a) might be exhibited by the result. Assume that `T` is a base table defined by

```
CREATE TABLE T ( A INTEGER NOT NULL,
                 B VARCHAR(10),
                 C INTEGER NOT NULL,
                 K1 CHAR(1),
                 K2 CHAR(1),
       PRIMARY KEY ( K1, K2 ) )
```

(i)
```
SELECT K2, A, C FROM T WHERE A > B
```

(ii)
```
SELECT * FROM T
```

(iii)
```
SELECT SUM(C) FROM T
```

(iv)
```
SELECT * FROM T T1, T T2 WHERE T1.K1 = T2.K2
```

(v)
```
SELECT DISTINCT K2, C FROM T
UNION ALL
SELECT K1 AS K2, COUNT(*) AS C
FROM T
GROUP BY K1
HAVING SUM(C) IS NULL
```

(c) Translate the following **Tutorial D** expression into SQL:

```
( r1{a,b} RENAME ( a AS x, b AS y )
UNION
( r2{a,b} RENAME ( a AS y, b AS x )
```

**Question 9**

(a) Assume that `T` is the only base table in the database and is defined in SQL by

```
CREATE TABLE T ( A INTEGER,
                 PRIMARY KEY ( A ) ) ;
```

Assume also that `T` contains just two rows, with `A = 1` and `A = 2`.

For each of the following, give an example of an SQL query that exhibits the given properties.

(i) The query contains just one `FROM` clause. The `SELECT` clause contains only column references (no literals or operators). The result consists of two identical rows.

(ii) No `FROM` clause contains a comma or the key word `JOIN`. No `SELECT` clause contains a literal or an operator. The result consists two identical rows.

(iii) The query contains just one `FROM` clause. The `FROM` clause does not contain a comma or the key word `JOIN`. The result contains two columns with the same name.

(iv) The query contains just one `FROM` clause, preceded by `SELECT *`. The result contains two columns with the same name.

(v) The result contains a column that has no name.

(vi) The key word `NULL` does not appear in the query. The query contains just one `FROM` clause. The `FROM` clause does not contain a comma or the key word `JOIN`. The result contains a column named `X` such that `X IS NULL` would evaluate to *true* for at least one row.

(b) With `T` as defined and populated in (a), consider the following two SQL queries, each resulting in a table with one row and one column:

```
SELECT SUM(X) + SUM(Y) AS SUMXY1
FROM ( SELECT A AS X FROM T ) T1 LEFT JOIN
     ( SELECT A AS Y FROM T WHERE A = 2 ) T2
     ON ( T1.X = T2.Y )
```

and

```
SELECT SUM(X+Y) AS SumXY2
FROM ( SELECT A AS X FROM T ) T1 LEFT JOIN
     ( SELECT A AS Y FROM T WHERE A = 2 ) T2
     ON ( T1.X = T2.Y )
```

What are the values of `SUMXY1` and `SUMXY2`?

(c) Relvars r1 and r2 are defined in **Tutorial D** as follows:

```
VAR r1 BASE RELATION { a INTEGER, b INTEGER } KEY{ a
};
  VAR r2 BASE RELATION { b INTEGER, a INTEGER } KEY{
  b };
```

(i) Write equivalent SQL `CREATE TABLE` statements, including constraints to ensure that equivalence, for `r1` and `r2`.

(ii) Translate the following **Tutorial D** expression into SQL:

```
r1 UNION r2
```

## Question 10

(a) Describe how an SQL user can obtain a table, one of whose columns has no name. **3 marks**

(b) Describe two ways in which an SQL user can obtain a table in which two or more distinct columns have the same name. **4 marks**

(c) Assume that SQL query `Q` results in a table whose columns are all named and no two of those columns have the same name. What SQL operations can be applied to `Q` that could not be applied, or could only be applied with restrictions, if its result columns were not so named? **5 marks**

(d) Describe two ways in which an SQL user can obtain a table in which the same row appears more than once, even when every base table has a primary key defined for it. **4 marks**

(e) Identical base tables, `BT`, are created using two different implementations of SQL. One of the columns of `BT` is defined as `C1 VARCHAR(10) NOT NULL`. Exactly the same rows are inserted into both tables. It is observed that the two implementations give different results for the query `SELECT MAX(C1) FROM BT`, even though both implementations conform to the international standard for SQL. How can this be explained? **5 marks**

(f) Give two possible explanations for the fact that the following two queries do not give the same results: **4 marks**

```
SELECT SUM(x+y) AS Sumxy
FROM T
```

```
SELECT SUM(z) AS Sumxy
FROM (SELECT SUM(x) AS z FROM T
      UNION
      SELECT SUM(y) AS z FROM T) Q
```

## C. How To Handle Missing Information Without Using NULL

### Question 11

(a) In each of the following lists, for each SQL expression *n*>1 state, giving reasons, whether it is equivalent to expression 1.

(i)
1. `SELECT * FROM t WHERE x = y`
2. `SELECT * FROM t WHERE NOT ( x <> y )`

(ii)
1. `SELECT * FROM t`
2. `SELECT * FROM t WHERE x = y OR NOT ( x = y )`

(iii)
1. `SELECT SUM(x) + SUM(y) FROM t`
2. `SELECT SUM(x+y) FROM t`

(iv)
1. `( x, y ) IS NOT NULL`
2. `x IS NOT NULL AND y IS NOT NULL`
3. `NOT ( ( x, y ) IS NULL )`

(v)
1. `( x, y ) IS NULL`
2. `x IS NULL AND y IS NULL`
3. `NOT ( x IS NOT NULL AND y IS NOT NULL )`
4. `NOT ( x IS NOT NULL ) AND NOT ( y IS NOT NULL )`
5. `NOT ( ( x, y ) IS NOT NULL )`

(b) A software development organization has an SQL database concerning its various projects. The database include a table that was created by

```
CREATE TABLE project
             ( project# INTEGER PRIMARY KEY,
               title VARCHAR(30) NOT NULL,
               dept# INTEGER NOT NULL REFERENCES dept,
               completed DATE );
```

NULL is used in the completed column for cases where the completion date is not (yet) known, also for projects that are considered to be indefinitely ongoing. They are unhappy with this design. For one thing they keep making mistakes with their queries because of the strange behaviour of NULL, resulting in bad decisions by misinformed executives. For another, they would like to be able to distinguish among ongoing projects, projects that haven't been completed yet, and completed projects whose completion dates are unknown.

You, the organization's DBA for this database, are tasked with the necessary redesign. You decide on an approach based on proposals you were taught in CS253, but you discover that SQL still lacks certain features without which the redesigned database will be exposed to a grave risk of inconsistency.

Draft some notes for use in a requirements statement to be submitted to the committee responsible for the SQL international standard. The notes should include `CREATE TABLE` statements for the new design, explanation of the perceived integrity exposure resulting from it, and suggestions for extensions to the language by which that exposure could be addressed. (You are not expected to suggest precise syntax for these extensions. Just give the committee sufficient information for them to devise such syntax.)

**Question 12**

(a) Give a possible explanation for

(i) `SELECT * FROM t WHERE x = y OR NOT ( x = y )`
gives a result that is empty even though `t` contains 100 rows. **2 marks**

(ii) `SELECT * FROM t WHERE ( x, y ) IS NOT NULL`
and
`SELECT * FROM t WHERE NOT ( ( x, y ) IS NULL )`
where `t` contains just one row, give different results. **2 marks**

(iii) `SELECT SUM(x) + SUM(y) AS SumXY FROM t`
and
`SELECT SUM(x+y) AS SumXY FROM t`
where `t` contains just two rows, give different results, neither of which contains an appearance of `NULL` for the column `SumXY`. **2 marks**

(iv)
```
SELECT * FROM t WHERE 0 = ( SELECT SUM(1) FROM t
                            WHERE y > y )
```
gives an empty result, whereas
```
SELECT * FROM t WHERE 0 = ( SELECT COUNT(*) FROM t
                            WHERE y > y )
```
gives the same result as `SELECT * FROM t`. **2 marks**

(b) The CS252 course work assignment in November, 2007, required you to create a single table for all transactions against bank accounts. Each transaction is of one of the four kinds: payment in, payment by cheque, payment by direct debit, payment by debit card. A possible solution was something like this:

```
CREATE TABLE Transaction ( Transaction# INTEGER,
                           Account# CHAR(8),
                           DateReceived DATE NOT NULL,
                           TimeReceived TIME NOT NULL,
                           Amount DECIMAL(9,2) NOT NULL,
                           Source VARCHAR(100),
                           Cheque# CHAR(6),
                           Payee VARCHAR(50),
                           DateWritten DATE,
                           Card# CHAR(9),
              PRIMARY KEY ( Transaction#, Account# ),
              FOREIGN KEY ( Account# ) REFERENCES Account,
                   UNIQUE ( Cheque#, Account# ),
              FOREIGN KEY ( Card#, Account# )
                   REFERENCES DebitCard( Card#, Account#
),
              CHECK ( expression to check consistency of
                      nullable columns ) ;
```

A payment in, identified by the `Amount` being negative, has a `Source` value and requires the other nullable columns to be `NULL`. A payment by cheque has `Cheque#`, `Payee`, and `DateWritten` values and requires the other nullable columns to be `NULL`. A payment by direct debit has a `Payee` value and requires the other nullable columns to be `NULL`. A payment by debit card has `Payee` and `Card#` values and requires the other nullable columns to be `NULL`.

Recall that the `DebitCard` table has `Card#` as primary key but includes the redundant constraint `UNIQUE ( Card#, Account# )` as a workaround to enable the foreign key referencing `DebitCard` to include a check that the debit card used for the payment really is for use with the relevant account.

You would like to redesign this part of the database to avoid using `NULL`. You would like your redesign to consist of a single table having all the columns that are common to all transactions plus a separate table for each of the four transaction types, with just the columns needed for the type in question. You would also like each of these five tables to be in fifth normal form (5NF). Unfortunately, you are using an SQL implementation that does not support constraints containing subqueries.

(i) Give an SQL `CREATE TABLE` statement for each of the five tables required in your redesign. You may omit the data types of the columns and the words `CREATE TABLE`. For each table, include a `PRIMARY KEY` declaration and other constraints to meet as many of the requirements as possible. **5 marks**

(ii) Describe the required constraints that you could not express in your solution to (ii). **4 marks**

(iii) Describe any additional enhancements to SQL that you would need to support your redesign. **3 marks**

**Question 13**

(a) Give a possible explanation for

(i)
```
SELECT COUNT(*) AS number_of_rows1
FROM (SELECT c1 FROM t UNION
      SELECT c1 FROM t) x
```
gives a result of 1 for `number_of_rows1`, whereas
```
SELECT COUNT(*) AS number_of_rows2
FROM (SELECT c1 FROM t) x1 NATURAL JOIN
     (SELECT c1 FROM t) x2
```
gives a result of 0 for `number_of_rows2` and
```
SELECT COUNT(*) AS number_of_rows3 FROM t
```
gives a result of 100 for `number_of_rows3`. **2 marks**

(ii) In addition to the phenomena observed in (i),
```
SELECT * FROM t WHERE c2 = c2
```
where column `c2` is of type `ROW(a INTEGER, b INTEGER)`, gives a result that is empty, whereas the result of
```
SELECT * FROM t WHERE c2 IS NOT NULL
```
contains 100 rows. **2 marks**

(iii)
```
SELECT CASE WHEN c3 = c4 THEN 0 ELSE 1 END AS x FROM t
```
and
```
SELECT CASE WHEN c3 <> c4 THEN 1 ELSE 0 END AS x FROM t
```
give different results. **2 marks**

(iv) In addition to the phenomena observed in (i),
```
SELECT * FROM t WHERE c5 > c6
```
gives an empty result, whereas
```
ALTER TABLE t ADD CONSTRAINT c5gtc6 CHECK (c5 > c6);
```
is accepted (does not give rise to an error). **2 marks**

(c) My collection of classical music consists of various kinds of pieces—symphonies, concertos, string quartets, sonatas, and so on—on CDs (usually several pieces per CD). Here, in a single table, is my first attempt to devise an SQL database in which to record my collection:

```
CREATE TABLE Piece
     ( CDid  VARCHAR(8),
       Title VARCHAR(100) NOT NULL,
       Type VARCHAR(30) NOT NULL
            CHECK Type IN ( 'symphony',
                            'concerto',
                            'chamber',
                            'sonata',
                            'other'),
       Composer VARCHAR(30) NOT NULL,
       Opus# INTEGER NOT NULL,
       Orchestra VARCHAR(30),
       Conductor VARCHAR(30),
       Leader VARCHAR(30),
       Soloist VARCHAR(30),
       Accompanist VARCHAR(30),
       Ensemble VARCHAR(30),
       Movements INTEGER NOT NULL,
       PRIMARY KEY ( CDid, Composer, Opus# ),
       FOREIGN KEY ( CDid ) REFERENCES CD,
       CONSTRAINT difficult_one CHECK ( ... ) ;
```

Note that the code for `CONSTRAINT difficult_one` is missing. I tried to write it but it proved too difficult for me. Here is what it is required to express:

- every symphony or concerto has an orchestra
- no sonata or piece of chamber music has an orchestra
- every piece that has an orchestra has a leader
- every piece that has a leader has an orchestra
- every piece that has a conductor has an orchestra
- every sonata has a soloist
- every concerto has either a soloist or an ensemble, but not both
- every piece of chamber music has an ensemble

I have decided to avoid all use of `NULL` and instead to replace `Piece` by several tables, using horizontal and/or vertical decomposition. But I need your help.

(i) Clearly, each of the tables in the new design will have `PRIMARY KEY ( CDid, Composer, Opus# ).`
For each table that you think I will need:

- suggest a table name
- if it has any non-key columns, list their names, using the names given in the definition of `Piece`
- list the names of any tables it should be referencing by foreign keys

For example, if you think I need a table called Symphony with extra columns Conductor and Soloist and a foreign key referencing a table named Piece, write `Symphony(Conductor, Soloist) FK to Piece` **5 marks**

(ii) Describe the additional constraints that are required. **3 marks**

(iii) Suggest enhancements to SQL that might be needed to support your all the constraints needed in your redesign. **4 marks**

**End of past exam questions**