Tutorial: Creating and Populating Tables

Students at the National University of Ngendipura (NUN) buy books for their studies. They also lend and borrow books to and from other students. Your company, Apasaja Private Limited, is commissioned by NUN Students Association (NUNStA) to implement an online book exchange system that records information about students, books that they own and books that they lend and borrow.

The database records the name, faculty, and department of each student. Each student is identified in the system by her email. The database also records the date at which the student joined the university (`year` attribute) and the date of her graduation (`graduate` attribute, which is `NULL` if the student has not graduated at the time of the database instance.).

The database records the title, authors, publisher, edition, ISBN-10, and ISBN-13 for each book. The International Standard Book Number, ISBN-10 or -13, is an industry standard for the unique identification of books. It is possible that the database records books that are not owned by any students (because the owners of a copy graduated or because the book was advised by a lecturer for a course but not yet purchased by any student.)

The database records the date at which a book copy is borrowed and the date at which it is returned. We refer to this information as a loan record.

The database records information about books, copies and owners as long as the owners are students. Loan records of books that have been returned are deleted at the end each semester. The database records information about graduated students as long as there are loan records concerning books that they owned or borrowed but did not return.

1. You are provided with the relational schema and with an instance of the database dated from the end of academic year 2021–2022 (31^{st} July 2022). You are given the SQL data definition language code to create the schema and the SQL data manipulation language code to create a instance of the database.

   (a) Download the following files from Canvas "`Files > Cases > Book Exchange`".

   `NUNStASchema.sql`,
   `NUNStAStudent.sql`,
   `NUNStABook.sql`,
   `NUNStACopy.sql`,
   `NUNStALoan.sql`,
   and `NUNStAClean.sql`.

   (b) Read the SQL files. What are they doing?

   Solution: `NUNStASchema.sql` creates the tables with the associated domain declarations and constraints.
   `NUNStAStudent.sql`, `NUNStABook.sql`, `NUNStACopy.sql` and `NUNStALoan.sql` populate the tables.
   `NUNStAClean.sql` drops all the tables and their content, if needed.

(c) Use the files to create and populate a database (they may need some "bug" fixing).

Solution: Create a database with a name of your choice in pgAdmin 4. Open the query tool in the database. Load and run the sql files.

The first file to be run is `NUNStASchema.sql`. It creates the different tables. The referential integrity constraints (`FOREIGN KEY`) impose that the table `copy` and the table `loan` are created after the tables `student` and `book` and in that order.

The table `student` is populated using `NUNStAStudent.sql`.

The table `book` is populated using `NUNStABook.sql`.

These two tables can be populated in any order.

The table `copy` is populated using `NUNStACopy.sql`.

The table `loan` is populated using `NUNStALoan.sql`.

The table `copy` and the table `loan` can only be populated after the tables `student` and `book` are populated and in that order because of the referential integrity constraints (`FOREIGN KEY`).

The referential integrity constraints (`FOREIGN KEY`) impose that the table `loan` and the table `copy` are deleted before the tables `student` and `book` and in that order in `NUNStAClean.sql` matters.

2. Insertion, deletion, and update.

(a) Insert the following new book.

```
INSERT INTO book VALUES (
'An Introduction to Database Systems',
'paperback' ,
640 ,
'English' ,
'C. J. Date' ,
'Pearson',
'0321197844' ,
'978-0321197849');
```

Solution:

Notice the implicit order of the fields.

You can check that the insertion was effective with the following query.

```
SELECT *
FROM book;
```

(b) Insert the same book with a different `ISBN13`, for instance `'978-0201385908'`.

Solution:

```
INSERT INTO book VALUES (
'An Introduction to Database Systems',
'paperback',
640,
'English',
'C.J. Date',
'Pearson',
'0321197844',
'978-0201385908');
```

The command yields an error because `ISBN10` must be unique. PostgreSQL returns the following error message.

```
ERROR:  duplicate key value violates unique constraint "book_isbn10_key"
DETAIL:  Key (isbn10)=(0321197844) already exists.
SQL state: 23505
```

All messages emitted by the PostgreSQL server are assigned five-character error codes that follow the SQL standard's conventions for "SQLSTATE" codes. Applications that need to know which error condition has occurred should usually test the error code, rather than looking at the textual error message. See `www.postgresql.org/docs/13/errcodes-appendix.html`

(c) Insert the same book with the original `ISBN13` but with a different `ISBN10`, for instance `'0201385902'`.

Solution:

```
INSERT INTO book VALUES (
'An Introduction to Database Systems',
'hardcover',
938,
'English',
'C.J. Date',
'Addison Wesley Longman',
'0201385902',
'978-0321197849');
```

The command yields an error because `ISBN13` is a primary key and therefore unique. PostgreSQL returns the following error message.

```
ERROR:  duplicate key value violates unique constraint "book_pkey"
DETAIL:  Key (isbn13)=(978-0321197849) already exists.
SQL state: 23505
```

(d) Insert the following new student.

```
INSERT INTO student VALUES (
'TIKKI TAVI' ,
'tikki@gmail.com' ,
'2021-08-01',
'School of Computing',
'CS',
NULL);
```

Solution: Notice that the value of the field `year` is `NULL`. This is because the student has not yet graduated.

(e) Insert the following new student.

```
INSERT INTO student (email, name, year, faculty, department)
VALUES (
'rikki@gmail.com',
'RIKKI TAVI',
'2021-08-01',
'School of Computing',
'CS');
```

Solution:

Notice how we explicitly indicate the order of the fields in the insertion command. In this case, if a field is omitted, the system attempts to insert a null value.

Try the following insertion.

```
INSERT INTO student (name, year, faculty, department)
VALUES (
'RIKKI TAVI',
'2021-08-01',
'School of Computing',
'CS');
```

The command does not work because `email` is a primary key and therefore cannot be null.

```
ERROR:  null value in column "email" of relation "student"
violates not-null constraint
DETAIL:  Failing row contains (RIKKI TAVI, null, 2021-08-01, School of Computing,
CS, null).
SQL state: 23502
```

(f) Change the name of the department 'CS' to 'Computer Science'.

```
UPDATE student
SET department = 'Computer Science'
WHERE department = 'CS';
```

Solution:

You can check that the update was effective with the following queries. The first query has no result.

```
SELECT *
FROM student
WHERE department = 'CS';
```

The second query prints the students from the computer science department.

```
SELECT *
FROM student
WHERE department = 'Computer Science';
```

(g) Delete the students from the 'chemistry' department.

Solution:

```
DELETE FROM student
WHERE department = 'chemistry';
```

'chemistry' is misspelled with a lower case 'c'. There is no error but nothing is deleted because there is no department 'chemistry'.

(h) Delete the students from the 'Chemistry' department.

Solution:

```
DELETE FROM student
WHERE department = 'Chemistry';
```

Nothing is deleted because a constraint is violated. It is not a programming error. It is part of the control of the access to the data.

```
ERROR:  update or delete on table "student"
violates foreign key constraint "loan_borrower_fkey"
on table "loan"
DETAIL:  Key (email)=(xiexin2011@gmail.com) is still referenced from table "loan".
SQL state: 23503
```

3. Integrity constraints (to be discussed in class.)

(a) Some constraints in PostgreSQL are `DEFERRABLE`. What does it mean?

Solution: Upon creation, a `UNIQUE`, `PRIMARY KEY`, or `FOREIGN KEY` constraint is given one of three characteristics: `DEFERRABLE INITIALLY IMMEDIATE`, `DEFERRABLE INITIALLY DEFERRED`, and `NOT DEFERRABLE`.

By default the above constraints and all others are `NOT DEFERRABLE`. A `NOT DEFERRABLE` constraint is always `IMMEDIATE`. By default a `DEFERRABLE` constraint is `INITIALLY IMMEDIATE`.

These qualifications refers to when the constraint is checked: immediately after each operation (`INSERT`, `DELETE`, `UPDATE`), or at the end of the transaction executing the operation.

Although this is not the default setting, it is preferable that all constraints be deferred. Unfortunately, this is only possible for `UNIQUE`, `PRIMARY KEY`, and `FOREIGN KEY` constraints and not for `CHECK` constraints in the current version of PostgreSQL.

(b) Insert the following copy of 'An Introduction to Database Systems' owned by Tikki.

```
INSERT INTO copy
VALUES (
'tikki@gmail.com',
'978-0321197849',
1);
```

What is the difference between the following two SQL programs?

```
BEGIN TRANSACTION;
SET CONSTRAINTS ALL IMMEDIATE;
DELETE FROM book
WHERE ISBN13 = '978-0321197849' ;
DELETE FROM copy
WHERE book = '978-0321197849' ;
END TRANSACTION;
```

```
BEGIN TRANSACTION;
SET CONSTRAINTS ALL DEFERRED;
DELETE FROM book WHERE ISBN13 = '978-0321197849' ;
DELETE FROM copy WHERE book = '978-0321197849' ;
END TRANSACTION;
```

Solution: In the first transaction, the statement `SET CONSTRAINTS ALL IMMEDIATE;` checks the reference from the table `copy` to the table `book` checkable after each operation. The first deletion of the transaction violates the constraint.

```
BEGIN TRANSACTION;
DELETE FROM book
WHERE ISBN13 = '978-0321197849';
DELETE FROM copy
WHERE book = '978-0321197849';
END TRANSACTION;
```

This is the default, so the code below has the same issue.

```
BEGIN TRANSACTION;
DELETE FROM book
WHERE ISBN13 = '978-0321197849';
DELETE FROM copy
WHERE book = '978-0321197849';
END TRANSACTION;
```

When the execution is interrupted, you need to rollback the transaction explicitely and manually `ROOLLBACK;` manually.

You can also use commands like `BEGIN`, `COMMIT` as well as `SAVEPOINT my_savepoint` and `ROLLBACK TO my_savepoint;` to control the flow of transactions.

In the second transaction, the constraints are checked at the end of the transaction and are not violated. Namely, `SET CONSTRAINTS ALL DEFERRED` makes the reference from the table `copy` to the table `book` checkable at the end of transactions. The combined effect of the two deletions in the transaction does not violate the FOREIGN KEY constraint. The transaction is committed.

Note that SQLite has a pragma called `defer_foreign_keys` to control deferred foreign keys.

4. Modifying the Schema (to be discussed in class.)
   (a) Add a field `available` to the table `copy` that indicates whether the copy is currently borrowed, `'false'`, or is available, `true'`. Make the necessary changes. Is it needed?

Solution: Add the field with a default value set to true.

```
ALTER TABLE copy
ADD COLUMN available BOOLEAN DEFAULT 'true';
```

```
UPDATE copy
SET available = 'false'
WHERE (owner, book, copy) IN
(SELECT owner, book, copy
FROM loan
WHERE returned ISNULL);
```

The availability of a copy can be derived. The following query finds the copies loaned but not returned and therefore not available.

```
SELECT owner, book, copy, returned
FROM loan
WHERE returned ISNULL;
```

We drop the field `available` in the table `copy`

```
ALTER TABLE copy
DROP COLUMN available;
```

We create a view `copy_view` with the field restored.

```
CREATE OR REPLACE VIEW copy_view (owner, book, copy, available)
AS
(SELECT DISTINCT c.owner, c.book, c.copy,
       CASE
       WHEN EXISTS (SELECT * FROM loan l
       WHERE l.owner = c.owner
       AND l.book = c.book
       AND l.copy = c.copy AND l.returned ISNULL) THEN 'FALSE'
       ELSE 'TRUE'
       END
FROM copy c);

SELECT * FROM copy_view;
```

Can the view be updated?

```
UPDATE copy_view
SET owner = 'rikki@gmail'
WHERE owner = 'tikki@gmail.com'
```

In principle we should be able to update all fields except `available`. It is however not directly possible but it can be programmed using `INSTEAD OF UPDATE` triggers or unconditional `ON UPDATE DO INSTEAD` rules. In addition, in this case, all fields are foreign keys and should be updated accordingly.

```
DROP VIEW copy_view;
```

(b) Argue that the table `student` should not contain both the fields `department` and `faculty`. Make the necessary changes.

Solution: The faculty is determined by the department. This information can be stored once and for all in a separate table. The student table needs only to store the department.
The changes can be done with the following SQL code.

```
CREATE TABLE department (
department VARCHAR(32) PRIMARY KEY,
faculty VARCHAR(62) NOT NULL);

INSERT INTO department
SELECT DISTINCT department, faculty
FROM student;

ALTER TABLE student
DROP COLUMN faculty;

ALTER TABLE student
ADD FOREIGN KEY (department) REFERENCES department(department);
```

References

[1] W3schools online web tutorials. `www.w3schools.com`. Visited on 21 July 2022.

[2] S. Bressan and B. Catania. *Introduction to Database Systems*. McGraw-Hill Education, 2006.

[3] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Pearson, 7th edition, 2015.

[4] H. Garcia-Molina, J.D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Pearson international edition. Pearson Prentice Hall, 2009.

[5] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 2002.

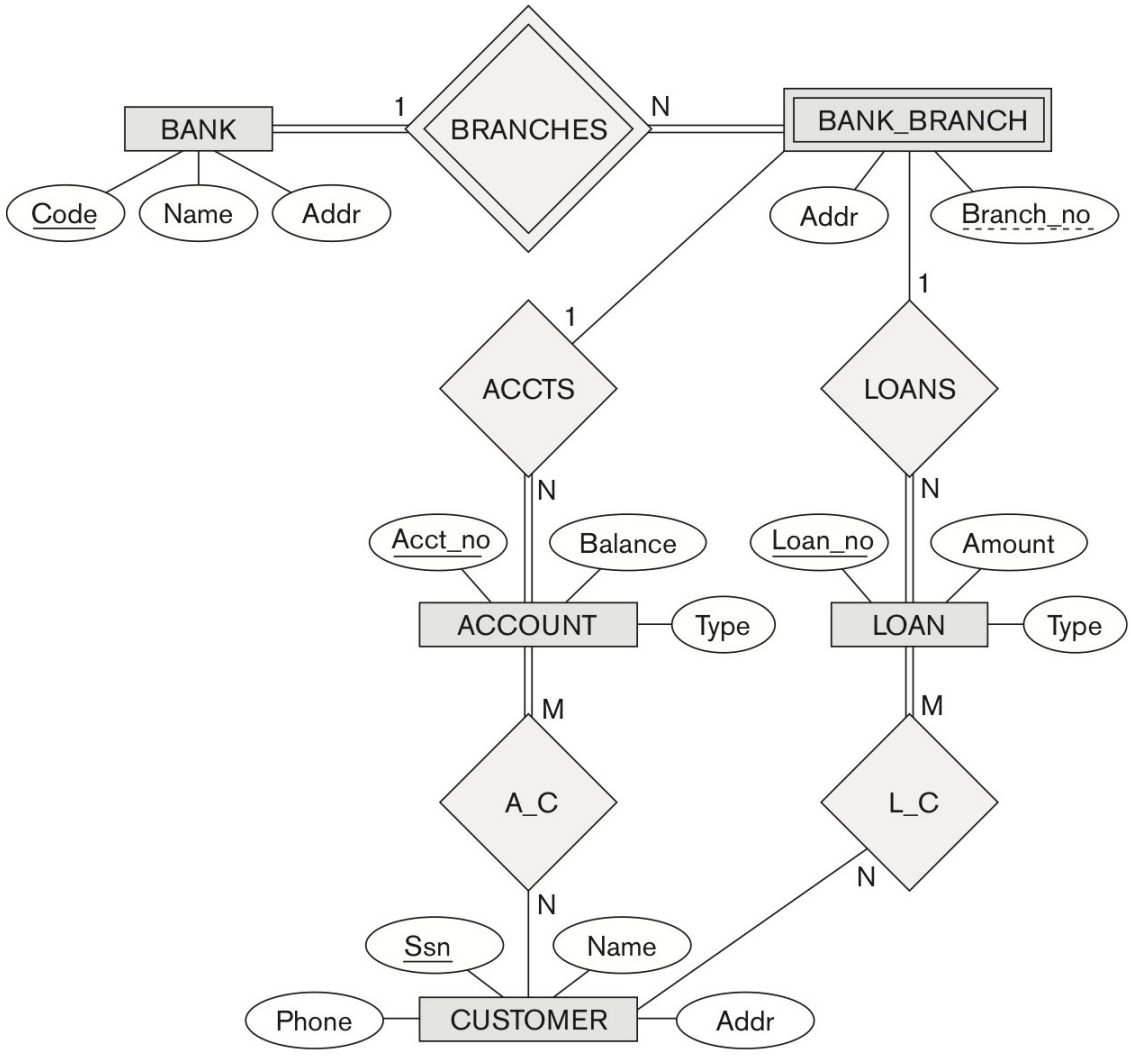
LECTURE 3 TUTORIAL (ER MODELING AND MAPPING ER SCHEMAS TO A RELATIONAL DESIGN)

THREE QUESTIONS

(WITH SOLUTION)

QUESTION 1:

Consider the ER schema of a Bank Database:



Q1. WITH SOLUTION

1A. Answer the following questions for this ER Schema:

a. List the strong entity types in the schema
   ANSWER: BANK, ACCOUNT, LOAN, CUSTOMER
b. Is there a weak entity type in the schema? If so, name it, name the identifying relationship that identifies it and state the partial identifier and complete identifier of this entity type.
   ANSWER: BANK_BRANCH is the Weak Entity type. The identifying relationship type is BRANCHES . The partial identifier for BANK_BRANCH is Branch_no and the full identifier is (Bank_Code,Branch_no).
c. Can a LOAN be assigned to multiple customers
   ANSWER: YES
d. Can there be customers in the database that have multiple accounts?
   ANSWER: YES
e. Can there be customers in the database that have neither an account, nor a loan?
   ANSWER: YES
f. What constraints did you use to give the above answer in (e) ?
   ANSWER: THE MINIMUM CARDINALITY CONSTRAINTS FOR CUSTOMER ENTITY IN RELATIONSHIPS A_C AND L_C ARE ZERO. IN OTHER WORDS, THE PARTICIPATION OF A Customer (entity) in the relationship type A_C and L_C IS OPTIONAL.

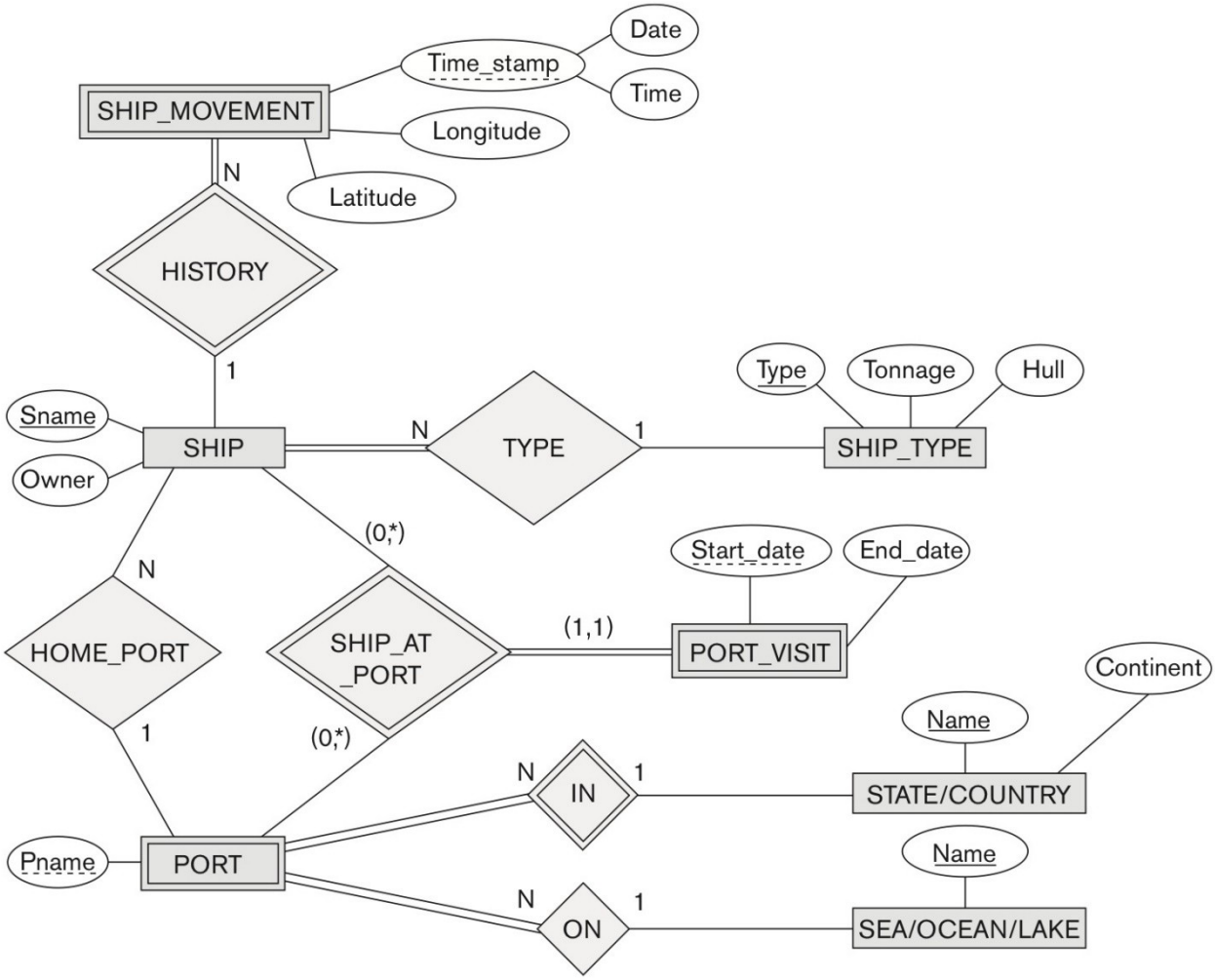
1B. Questions about mapping to a set of relations:

a. Show the mapping of the weak entity type from this schema into a relation.
   ANSWER: BANK_BRANCH (Bank_code, Branch_no, Addr)
b. Consider the mapping of the ACCOUNT entity type into the following relation:
   ACCOUNT (Account#, ……., Balance, Account-type, Customer_SSN).
   What attribute is missing from the key?
   ANSWER: Bank_code, Branch_no.
   What attribute is incorrect in this relation? Why?
   ANSWER: Customer_SSN included here is incorrect. It cannot be included here because an account may be owned by multiple customers and therefore a separate relation is needed to capture the many-to-many relationship between CUSTOMER and ACCOUNT.
c. If we map the L_C relationship type into a relation, what will that relation look like?
   ANSWER: L_C (Loan#, Customer_SSN)
d. How is the LOANS relationship type taken into consideration while mapping to the relational schema?
   ANSWER: By incorporating the foreign key (Branch_no, Bank_code) into the relation for LOAN.

e. The A_C relationship type is mapped into the relation:
CUSTOMER_ACCOUNT (Acct#, SSN, Opening_date).
What is the primary key of this relation?
ANSWER: {Acct#, SSN}
Opening_date should have been in the ER schema if this relation was correct. Where would this attribute be placed in the ER schema?
ANSWER: As an attribute of the relationship type A_C.

Q2. ER TO RELATIONAL SCHEMA MAPPING

Consider a database kept by a shipping company HONG_KONG_LINES. They use it to keep track of the movement of ships and track ships periodically as they move. They do this to log data on all ships in their possession for reporting to maritime authorities. Your task is to map this schema into a set of relations. Identify Primary and Foreign keys appropriately.



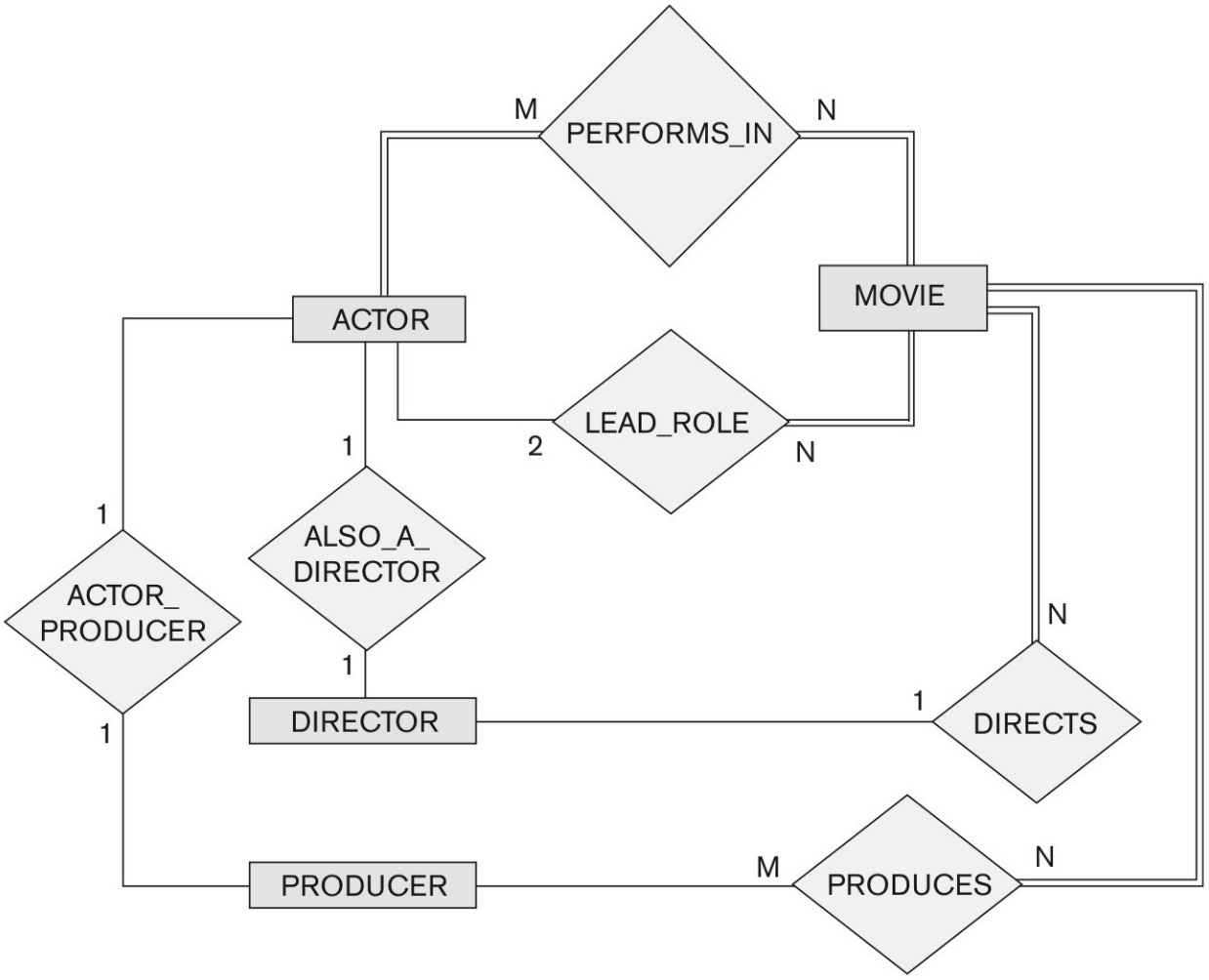
SOLUTION TO Q2 (RELATIONAL DESIGN):

We get 4 relations for the 4 strong entity types and 3 more relations for the three weak entity types. There are no many-to-may relationships; hence no additional relations are required. All 1:N relationships are handled by incorporating the key of the entity on N side of the relationship into the relation on the 1 side as a foreign key.

SHIP (Sname, Owner, Ship_type, Home_portname, Home_countryname)
SHIP_TYPE (Ship_type,Tonnage, Hull)
STATE_COUNTRY (Country_name, Continent)
SEA_OCEAN (Sea_name)

Weak Entity types:
SHIP_MOVEMENT (Sname, Date, Time, Latitude, Longitude)
PORT (Portname, Countryname, Seaname)
PORT_VISIT (Portname, Countryname, Sname, Start_date, Enddate)

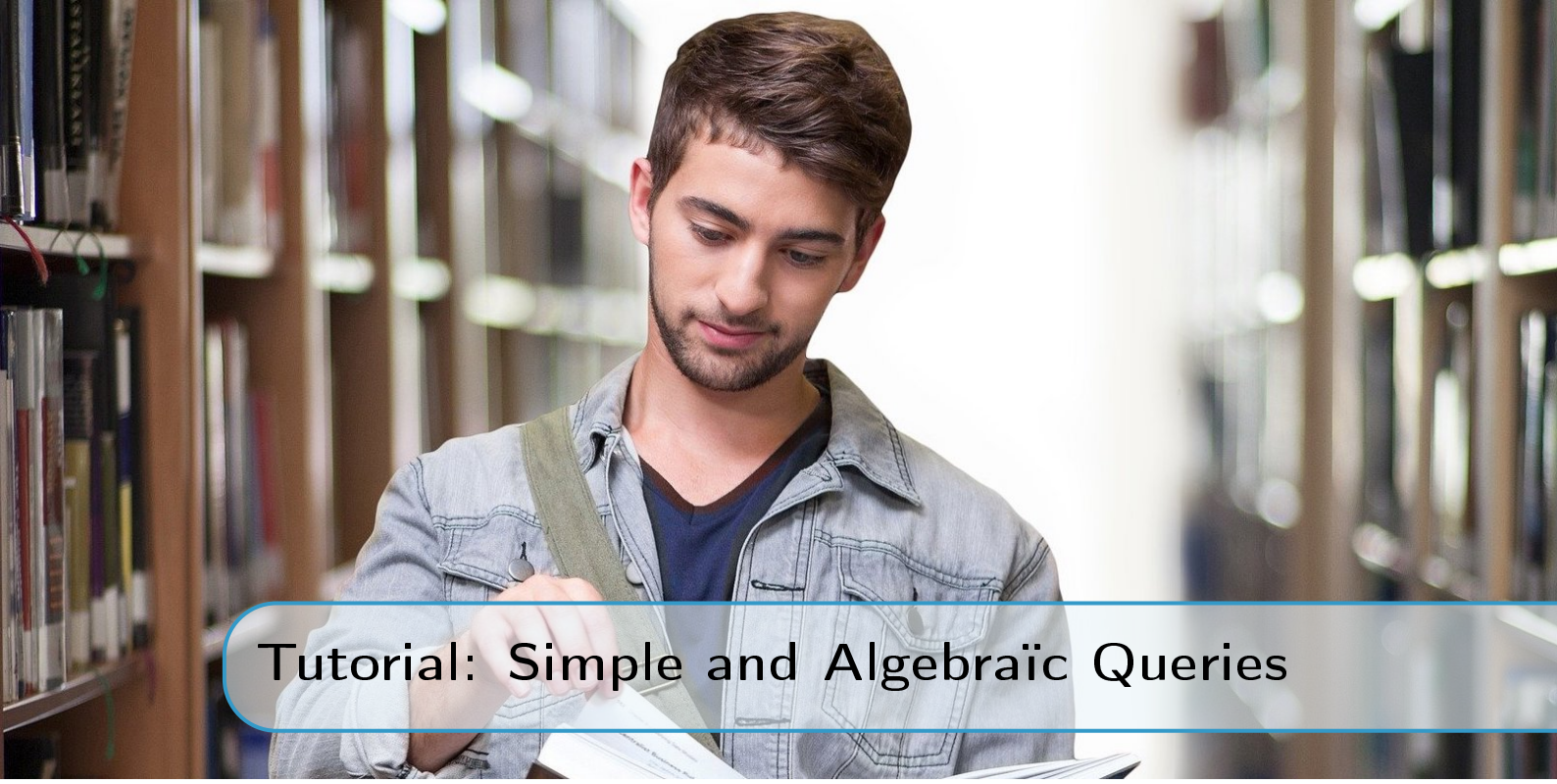
Q3. ADDITIONAL Question for ER Modeling. Consider the MOVIES ER Schema that represents a database that has been populated with appropriate data consistent with this schema. ACTOR is used as a generic term and includes both males and female actresses. Attributes are not shown to keep the schema simple.



QUESTION 3 – A POPULATED DATABASE FOR ABOVE SCHEMA with SOLUTION:

Assume that MOVIES is a populated database. Given the constraints shown in the ER schema, respond to the following statements with True, False, or Maybe. Your answer should be based on what populated database will be consistent with the given schema. Assign a response of Maybe to statements that, although not explicitly shown to be True, cannot be proven False based on the schema as shown. Justify each answer.

a. There are no actors in this database that have been in no movies. NO
b. There are some actors who have acted in more than ten movies. YES
c. Some actors have done a lead role in multiple movies. YES
d. A movie can have only a maximum of two lead actors. YES
e. Every director has been an actor in some movie. NO
f. No producer has ever been an actor. MAYBE (POSSIBLE, BUT CANNOT BE GURANTEED)
g. There are movies with more than a dozen actors. YES.
h. There are some producers who have also been a director. MAYBE (POSSIBLE, BUT CANNOT BE GURANTEED)
I. All movies have one director and one producer. NO
J. Some movies have one director but several producers. YES.
k. There are some actors who have done a lead role, directed a movie, and produced a movie. YES



Tutorial: Simple and Algebraïc Queries

This tutorial uses the schema and data of the database created in the previous Tutorial. All questions will be discussed in class.

1. Simple Queries.
   (a) Print the different departments.

   Solution:

   ```
   SELECT d.department
   FROM department d;
   ```

   Notice that the query does not require `DISTINCT` to eliminate duplicates. Duplicates are guaranteed not to occur because `department` is the `PRIMARY KEY` of the table `department`

   (b) Print the different departments in which students are enrolled.

   Solution:

   There could be departments in which no student is enrolled. This is the case of the department of `Undecidable Computations`. We need to look into the `student` table.

   ```
   SELECT DISTINCT s.department
   FROM student s;
   ```

   Notice that the query requires `DISTINCT` to eliminate duplicates since it is very likely that there is more than one student in most departments.

   (c) Let us check the integrity of the data. Print the emails of the students who borrowed or lent a copy of a book before they joined the university. There should not be any. Use a simple query.

   Solution:

   ```
   SELECT DISTINCT s.email
   FROM loan l, student s
   WHERE (s.email = l.borrower AND l.borrowed < s.year)
   OR (s.email = l.owner AND l.borrowed < s.year);
   ```

   (d) For each copy that has been borrowed and returned, print the duration of the loan. Order the results in ascending order of the ISBN13 and descending order of duration.

Solution:

```
SELECT l.book, l.returned - l.borrowed + 1 AS duration
FROM loan l
WHERE NOT (l.returned ISNULL)
ORDER BY l.book ASC, duration DESC;
```

`ASC` is the default, but it is strongly recommended to indicate it for clarity.

Notice that the duration can be null if the book has not been returned yet. For a complete answer, you need to calculate the duration until July 31, 2022 to include the books that have not been returned yet.

```
SELECT l.book,
  ((CASE
  WHEN l.returned ISNULL
  THEN '2022-07-31'
  ELSE l.returned
  END) - l.borrowed + 1) AS duration
FROM loan l
ORDER BY l.book ASC, duration ASC;
```

(e) For each loan of a book published by Wiley that has not been returned, print the title of the book, the name and faculty of the owner and the name and faculty of the borrower. Use CROSS JOIN.

Solution:

We join primary keys and foreign keys to stitch tables together properly.

```
SELECT b.title,
  s1.name AS ownername,
  d1.faculty AS ownerFaculty,
  s2.name AS borrowername,
  d2.faculty AS  borrowerfaculty
FROM loan l, book b,  copy c,
  student s1, student s2,
  department d1, department d2
WHERE l.book=b.ISBN13
  AND c.book = l.book
  AND c.copy = l.copy
  AND c.owner = l.owner
  AND l.owner = s1.email
  AND l.borrower = s2.email
  AND s1.department = d1.department
  AND s2.department = d2.department
  AND b.publisher ='Wiley'
  AND l.returned ISNULL;
```

You can omit the table `copy` and the `copy` column since the existence of the corresponding rows and values is guaranteed by design and by the foreign and primary key constraints.

```
SELECT b.title,
  s1.name AS ownername,
  d1.faculty AS ownerFaculty,
  s2.name AS borrowername,
  d2.faculty AS  borrowerfaculty
FROM loan l, book b,
  student s1, student s2,
  department d1, department d2
WHERE l.book=b.ISBN13
  AND l.owner = s1.email
  AND l.borrower = s2.email
  AND s1.department = d1.department
  AND s2.department = d2.department
  AND b.publisher ='Wiley'
  AND l.returned ISNULL;
```

2. Algebraïc Queries.

(a) For each loan of a book published by Wiley that has not been returned, print the title of the book, the name and faculty of the owner and the name and faculty of the borrower. Use INNER JOIN.

Solution:

```
SELECT b.title,
  s1.name AS ownername,
  d1.faculty AS ownerFaculty,
  s2.name AS borrowername,
  d2.faculty AS  borrowerfaculty
FROM loan l
  INNER JOIN book b ON l.book=b.ISBN13
  INNER JOIN copy c ON c.book = l.book
    AND c.copy = l.copy
    AND c.owner = l.owner
  INNER JOIN student s1 ON l.owner = s1.email
  INNER JOIN student s2 ON l.borrower = s2.email
  INNER JOIN department d1 ON s1.department = d1.department
  INNER JOIN department d2 ON s2.department = d2.department
WHERE  b.publisher ='Wiley'
  AND l.returned ISNULL;
```

You can omit the table `copy` and the `copy` column since the existence of the corresponding rows and values is guaranteed by design and by the foreign and primary key constraints.

```
SELECT b.title,
  s1.name AS ownername,
  d1.faculty AS ownerFaculty,
  s2.name AS borrowername,
  d2.faculty AS  borrowerfaculty
FROM loan l
  INNER JOIN book b ON l.book=b.ISBN13
  INNER JOIN student s1 ON l.owner = s1.email
  INNER JOIN student s2 ON l.borrower = s2.email
  INNER JOIN department d1 ON s1.department = d1.department
  INNER JOIN department d2 ON s2.department = d2.department
  WHERE  b.publisher ='Wiley'
  AND l.returned ISNULL;
```

(b) Print the emails of the different students who borrowed or lent a copy of a book on the day that they joined the university. Use an algebraïc query.

Solution:

```
SELECT  s.email
FROM loan l, student s
WHERE s.email = l.borrower AND l.borrowed = s.year
UNION
SELECT  s.email
FROM loan l, student s
WHERE s.email = l.owner AND l.borrowed = s.year;
```

`DISTINCT` is not needed because `UNION` eliminates duplicates (so do `INTERSECT`, `EXCEPT` and `MINUS`). `UNION ALL` keeps the duplicates.

```
SELECT  s.email
FROM loan l, student s
WHERE s.email = l.borrower AND l.borrowed = s.year
UNION ALL
SELECT  s.email
FROM loan l, student s
WHERE s.email = l.owner AND l.borrowed = s.year;
```

The corresponding simple query is generally preferable.

```
SELECT DISTINCT s.email
FROM loan l,  student s
WHERE (s.email = l.borrower OR s.email = l.owner)
AND l.borrowed = s.year;
```

The simple query requires an explicit `DISTINCT`.

(c) Print the emails of the different students who borrowed and lent a copy of a book on the day that they joined the university. Use an algebraïc query.

Solution:

```
SELECT  s.email
FROM loan l, student s
WHERE s.email = l.borrower AND l.borrowed = s.year
INTERSECT
SELECT  s.email
FROM loan l, student s
WHERE s.email = l.owner AND l.borrowed = s.year;
```

Note that the corresponding simple query is more complicated. It needs two `loan` tables.

```
SELECT DISTINCT s.email
FROM loan l1, loan l2, student s
WHERE s.email = l1.borrower AND l1.borrowed = s.year
AND s.email = l2.owner AND l2.borrowed = s.year;
```

(d) Print the emails of the students who borrowed but did not lend a copy of a book on the day that they joined the university. Use an algebraïc query.

Solution:

```
SELECT  s.email
FROM loan l, student s
WHERE s.email = l.borrower AND l.borrowed = s.year
EXCEPT
SELECT  s.email
FROM loan l, student s
WHERE s.email = l.owner AND l.borrowed = s.year;
```

There is no corresponding simple query. We would need to use nested or aggregate queries for this type of questions.

(e) Print the ISBN13 of the books (not the copies) that have never been borrowed. Use an algebraïc query.

Solution:

```
SELECT  b.ISBN13
FROM book b
EXCEPT
SELECT  l.book
FROM loan l
```

or, using an `OUTER JOIN`, which introduces `NULL` values,

```
SELECT  b.ISBN13
FROM book b LEFT OUTER JOIN loan l ON b.isbn13 = l.book
WHERE l.book ISNULL;
```

There is no corresponding simple query. We would need to use nested or aggregate queries for this type of questions.

References

[1] W3schools online web tutorials. `www.w3schools.com`. Visited on 21 July 2022.

[2] S. Bressan and B. Catania. *Introduction to Database Systems*. McGraw-Hill Education, 2006.

[3] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Pearson, 7th edition, 2015.

[4] H. Garcia-Molina, J.D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Pearson international edition. Pearson Prentice Hall, 2009.

[5] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 2002.

Tutorial: Aggregate and Nested Queries

This tutorial uses the schema and data of the database created in Tutorial 1. All queries will be discussed in class.

1. Aggregate Queries.
   (a) How many loans involve an owner and a borrower from the same department?

Solution:

```
SELECT COUNT(*)
FROM loan l, student s1, student s2
WHERE l.owner = s1.email
  AND l.borrower = s2.email
  AND s1.department = s2.department;
```

   (b) For each faculty, print the number of loans that involve an owner and a borrower from this faculty?

Solution:

```
SELECT d1.faculty, COUNT(*)
FROM loan l, student s1, student s2, department d1, department d2
WHERE l.owner = s1.email
  AND l.borrower = s2.email
  AND s1.department = d1.department
  AND s2.department = d2.department
  AND d1.faculty = d2.faculty
GROUP by d1.faculty;
```

   (c) What are the average and the standard deviation of the duration of a loan? Round the results to the nearest integer.

Solution:

```
SELECT ROUND(AVG((CASE
  WHEN l.returned ISNULL
  THEN '2022-07-31'
  ELSE l.returned
  END) - l.borrowed + 1),0),
  ROUND(STDDEV_POP ((CASE
  WHEN l.returned ISNULL
    THEN '2022-07-31'
    ELSE l.returned
    END) - l.borrowed + 1),0)
```

```
FROM loan l;
```

or

```
SELECT ROUND(AVG(temp.duration),0),
ROUND(STDDEV_POP (temp.duration),0)
FROM (SELECT ((CASE
  WHEN l.returned ISNULL
  THEN '2022-07-31'
  ELSE l.returned
  END) - l.borrowed + 1) AS duration FROM loan l) AS temp;
```

2. Nested Queries

(a) Print the titles of the different books that have never been borrowed. Use a nested query.

Solution:

```
SELECT  b.title
FROM book b
WHERE b.ISBN13 NOT IN (
  SELECT  l.book
  FROM loan l);
```

or, equivalently,

```
SELECT  b.title
FROM book b
WHERE b.ISBN13 <> ALL (
  SELECT  l.book
  FROM loan l);
```

Always use one of the quantifiers `ALL` or `ANY` in front of subqueries wherever possible even though some systems may be lenient with this requirement.

Note that there could be several time the same title (since there could be different books with the same title) but not the same book. There is no need to use `DISTINCT` since the query ask for the different books but not for the different titles.

(b) Print the name of the different students who own a copy of a book that they have never lent to anybody.

Solution:

```
SELECT  s.name
FROM student s
WHERE s.email IN (
  SELECT c.owner
  FROM copy c
  WHERE NOT EXISTS (
    SELECT *
    FROM loan l
    WHERE l.owner = c.owner
      AND l.book = c.book
      AND l.copy = c.copy));
```

or, equivalently,

```
SELECT  s.name
FROM student s
WHERE s.email = ANY (
  SELECT c.owner
  FROM copy c
  WHERE NOT EXISTS (SELECT *
    FROM loan l
    WHERE l.owner = c.owner
      AND l.book = c.book
      AND l.copy = c.copy));
```

The query can also be written as follows but the highlighted tuple construction does not always work on other systems than PostgreSQL.

```
SELECT  s.name
FROM student s
WHERE s.email IN (
  SELECT c.owner
  FROM copy c
  WHERE (c.owner, c.book, c.copy) NOT IN
      (SELECT l.owner, l.book, l.copy
      FROM loan l));
```

The following query prints several time the name of those students who own several copies that have never been borrowed. We would not be able to differentiate the repeated names of students who own several copies that have never been borrowed from the repeated names of different students with the same name.

```
SELECT  s.name
FROM student s, copy c
WHERE s.email = c.owner
  AND NOT EXISTS (SELECT *
  FROM loan l
    WHERE l.owner = c.owner
      AND l.book = c.book
      AND l.copy = c.copy);
```

We can eliminate the duplicate students using `GROUP BY`

```
SELECT  s.name
FROM student s, copy c
WHERE s.email = c.owner
  AND NOT EXISTS (SELECT *
  FROM loan l
    WHERE l.owner = c.owner
      AND l.book = c.book
      AND l.copy = c.copy)
GROUP BY s.email, s.name;
```

(c) For each department, print the names of the students who lent the most.

Solution:

```
SELECT  s.department, s.name, count(*)
FROM student s, loan l
WHERE l.owner = s.email
GROUP BY s.department, s.email, s.name
HAVING count(*) >= ALL
  (SELECT  count(*)
  FROM student s1, loan l1
  WHERE l1.owner = s1.email
    AND s.department = s1.department
  GROUP BY s1.email);
```

Notice that there are two such students in the Chemistry department (that is why one should almost never use TOP N queries).

If we create a new department called Undecidable Computations with some students who never borrowed any book, what would happen? If there were students in the department of Undecidable Computations, should we print all of them or none of them? They would all have borrowed zero book, which would be the maximum in the department... We should print them all (using `OUTER JOIN`, `CASE` and `ISNULL` to consider the cases of 0 loan). Are there students who never borrowed a book?

Note that we need to group by department in order to print the department although there is no ambiguity. Some systems, like PostgreSQL relax this rule. It is recommended not to use this relaxation for the sake of portability.

(d) Print the emails and the names of the different students who borrowed all the books authored by Adam Smith.

Solution:

```
SELECT s.email, s.name
FROM student s
WHERE NOT EXISTS (
  SELECT *
  FROM book b
  WHERE authors = 'Adam Smith'
    AND NOT EXISTS (
      SELECT *
      FROM loan l
      WHERE l.book = b.isbn13
        AND l.borrower = s.email));
```

References

[1] W3schools online web tutorials. `www.w3schools.com`. Visited on 21 July 2022.

[2] S. Bressan and B. Catania. *Introduction to Database Systems*. McGraw-Hill Education, 2006.

[3] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Pearson, 7th edition, 2015.

[4] H. Garcia-Molina, J.D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Pearson international edition. Pearson Prentice Hall, 2009.

[5] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 2002.