Lab 3
Advanced Data Manipulation

CSE 4308

DATABASE MANAGEMENT SYSTEM LAB

SEPTEMBER 22, 2024

Contents

1 Distinct

The DISTINCT statement is used to return only distinct (different) values. Inside a table, a column often contains many duplicate values; and sometimes you only want to list the different (distinct) values. The syntax is as follows:

```
SELECT DISTINCT attributename
FROM tablename;
```

For example,

```
SELECT DISTINCT DEPT_NAME FROM EMPLOYEE;
```

2 Range Operator

2.1 BETWEEN Operator

The BETWEEN operator selects values within a given range. The values can be numbers, strings, or dates. The BETWEEN operator is inclusive: begin and end values will be included. However, you have to provide the lower value first then the upper value.

```
SELECT attribute_1, .....
FROM tablename
WHERE attribute_1 BETWEEN l_value AND u_value;
```

For example,

```
SELECT NAME, DEPT_NAME, SALARY
FROM EMPLOYEE
WHERE SALARY BETWEEN 35000 AND 80000;
```

For excluding a range, we can use the NOT operator before BETWEEN. For example:

```
SELECT NAME, DEPT_NAME, SALARY
FROM EMPLOYEE
WHERE SALARY NOT BETWEEN 35000 AND 80000;
```

2.2 IN Operator

The IN operator allows one to specify multiple values in the WHERE clause. It is a shorthand for multiple OR conditions.

```
SELECT attribute_1, .....
FROM tablename
WHERE attribute_1 IN (value1, value2, ...);
```

For example,

```
SELECT NAME, SALARY
FROM EMPLOYEE
WHERE DEPT_NAME IN ('DEV', 'TESTING');
```

3 Rename

SQL aliases are used to give a table, or a column in a table, a temporary name. We can also use it for renaming an expression or an entire query. The primary purpose of using aliases is to make a column or table more readable.
For example:

```
SELECT ID, NAME, (SALARY *12) AS Y_SALARY
FROM EMPLOYEE;

SELECT E.NAME, P.NAME
```

```
FROM EMPLOYEE E, (SELECT * FROM PROJECT) P
WHERE E.P_ID = P.P_ID;
```

4 String Operator

4.1 LIKE Operator

In simple terms, the LIKE operator is used to match substrings in a query. It is used in a WHERE clause to search for a specified pattern in a column.

```
SELECT attribute_1, attribute_2 .....
FROM tablename
WHERE attribute_1 LIKE pattern;
```

Patterns are case-sensitive. For example,

```
SELECT NAME
FROM EMPLOYEE
WHERE NAME LIKE 'A%';
```

To restrict the name size of the resultant name to 4 characters:

```
SELECT NAME
FROM EMPLOYEE
WHERE NAME LIKE 'A___';
```

4.2 Concatenation

To concatenate multiple columns or any additional string with any column at the time of retrieving data, SQL supports string operator ‘||‘.

```
SELECT 'Employee NAME: ' || NAME
FROM EMPLOYEE;
```

5 Null Value Handling

Null values present special problems in relational operations. Consider the following:

```
SELECT NAME, (SALARY + BONUS) AS TOTAL
FROM EMPLOYEE;
```

Employees who have null values for their bonus will return Null as total salary. Solution: Use ‘NVL()‘ built-in function.

```
SELECT NAME, (SALARY + NVL(BONUS,0)) AS TOTAL
FROM EMPLOYEE;
```

6 Set Operator

6.1 UNION

For example, to find the employee name of the ‘DEV‘ or ‘TESTING‘ department, we can write:

```
SELECT NAME FROM EMPLOYEE WHERE DEPT_NAME = 'DEV'
UNION
SELECT NAME FROM EMPLOYEE WHERE DEPT_NAME = 'TESTING';
```

6.2 INTERSECT

To retrieve employees who work for both ‘DEV‘ and ‘TESTING‘:

```
SELECT NAME FROM EMPLOYEE WHERE DEPT_NAME = 'DEV'
INTERSECT
SELECT NAME FROM EMPLOYEE WHERE DEPT_NAME = 'TESTING';
```

7 Data Sorting

The ORDER BY keyword is used to sort the result-set in ascending or descending order.

```
SELECT attribute_1, attribute_2 .....
FROM tablename
ORDER BY attribute_1 [ASC/DESC];
```

For example,

```
SELECT NAME, SALARY
FROM EMPLOYEE
ORDER BY DEPT, SALARY DESC;
```

8 Aggregation Function

The syntax for aggregate functions is as follows:

```
SELECT [AVG/MIN/MAX/SUM/COUNT](attribute_1)
FROM EMPLOYEE;
```

For example,

```
SELECT AVG(SALARY)
FROM EMPLOYEE;
```

8.1 Group By

The GROUP BY clause is used to apply the aggregate function to groups of sets of tuples.

```
SELECT DEPT, AVG(SALARY)
FROM EMPLOYEE
GROUP BY DEPT;
```

8.2 Having

The HAVING clause is used to specify conditions on groups rather than on tuples.

```
SELECT DEPT, AVG(SALARY)
FROM EMPLOYEE
GROUP BY DEPT
HAVING AVG(SALARY) > 5000;
```

9 Subquery

SQL provides a mechanism for nesting subqueries.

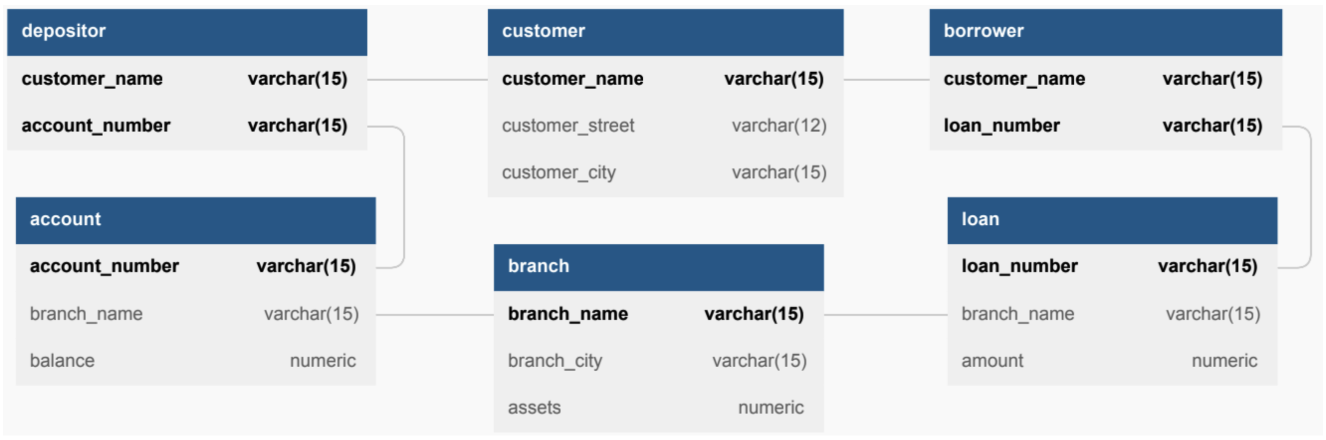
```
SELECT DEPT
FROM EMPLOYEE
WHERE SALARY >
(SELECT AVG(SALARY)
 FROM EMPLOYEE);
```

Or in the FROM clause:

```
SELECT DEPT
FROM EMPLOYEE E,
(SELECT AVG(SALARY) AS A_SALARY FROM EMPLOYEE) A
WHERE E.SALARY > A.A_SALARY;
```

10 Tasks

Execute the banking.sql script using the command. It creates a set of tables along with values that maintain the following schema:



Here, the boldfaces denote the primary keys and the arcs denote the foreign key relationships.
In this lab, you have to write all SQL statements in an editor first and save them with a `.sql` extension. Then, execute the SQL script. Write SQL statements for the following queries:

1. Find all customer names and their cities who have a loan but not an account.
2. Find all customer names who have an account as well as a loan (with and without set operator).
3. Find all customer-related information who have an account or a loan (with and without set operator).
4. Find the total assets of all branches.
5. Find the total number of accounts at each branch city.
6. Find the average balance of accounts at each branch and display them in descending order of average balance.
7. Find the average loan amount at each branch. Do not include any branch which is located in a city that has the substring `'Horse'` in its name.
8. Find the customer name and account number of the account which has the highest balance.
9. Find all customer-related information who have an account in a branch located in the same city as they live.
10. For each branch city, find the average amount of all the loans opened in a branch located in that branch city. Do not include any branch city in the result where the average amount of all loans opened in a branch located in that city is less than 1500.
11. Find those branch names which have a total account balance greater than the average total balance among all the branches.
12. Find the name of the customer who has at least one loan that can be paid off by his/her total balance.
13. Find the branch information for cities where at least one customer lives who does not have any account or any loans. The branch must have given some loans and have accounts opened by other customers.

Submission Guidelines

You are required to submit a report that includes your code, a detailed explanation, and the corresponding output. Rename your report as `<StudentID_Lab_1.pdf>`.

- Your lab report must be generated in LaTeX. Recommended tool: Overleaf. You can use some of the available templates in Overleaf.
- Include all code/pseudo code relevant to the lab tasks in the lab report.
- Please provide citations for any claims that are not self-explanatory or commonly understood.
- It is recommended to use vector graphics (e.g., `.pdf`, `.svg`) for all visualizations.
- In cases of high similarities between two reports, both reports will be discarded.

11 Code for Inserting in Tables

```
drop table depositor;
drop table borrower;
drop table account;
drop table loan;
drop table customer;
drop table branch;
```

Code 1: Drop Tables

```
create table branch
   (branch_name     varchar(15) not null,
    branch_city     varchar(15) not null,
    assets      number      not null,
    primary key(branch_name));

create table customer
   (customer_name   varchar(15) not null,
    customer_street     varchar(12) not null,
    customer_city   varchar(15) not null,
    primary key(customer_name));

create table account
   (account_number  varchar(15) not null,
    branch_name     varchar(15) not null,
    balance         number      not null,
    primary key(account_number),
    foreign key(branch_name) references branch(branch_name));

create table loan
   (loan_number     varchar(15) not null,
    branch_name     varchar(15) not null,
    amount      number      not null,
    primary key(loan_number),
    foreign key(branch_name) references branch(branch_name));

create table depositor
   (customer_name   varchar(15) not null,
    account_number  varchar(15) not null,
    primary key(customer_name, account_number),
    foreign key(account_number) references account(account_number),
    foreign key(customer_name) references customer(customer_name));

create table borrower
   (customer_name   varchar(15) not null,
    loan_number     varchar(15) not null,
    primary key(customer_name, loan_number),
    foreign key(customer_name) references customer(customer_name),
    foreign key(loan_number) references loan(loan_number));
```

Code 2: Create Tables

```
insert into customer    values ('Jones',    'Main',     'Harrison');
insert into customer    values ('Smith',    'Main',     'Rye');
insert into customer    values ('Hayes',    'Main',     'Harrison');
insert into customer    values ('Curry',    'North',    'Rye');
insert into customer    values ('Lindsay',  'Park',     'Pittsfield');
insert into customer    values ('Turner',   'Putnam',   'Stamford');
insert into customer    values ('Williams', 'Nassau',   'Princeton');
insert into customer    values ('Adams',    'Spring',   'Pittsfield');
insert into customer    values ('Johnson',  'Alma',     'Palo Alto');
```

```sql
insert into customer    values ('Glenn',    'Sand Hill',    'Woodside');
insert into customer    values ('Brooks',   'Senator',   'Brooklyn');
insert into customer    values ('Green',    'Walnut',    'Stamford');
insert into customer    values ('Jackson',  'University',    'Salt Lake');
insert into customer    values ('Majeris',  'First',     'Rye');
insert into customer    values ('McBride',  'Safety',    'Rye');
```

Code 3: Insert Data into Customer Table

```sql
insert into branch  values ('Downtown', 'Brooklyn',  900000);
insert into branch  values ('Redwood',  'Palo Alto',    2100000);
insert into branch  values ('Perryridge',   'Horseneck',    1700000);
insert into branch  values ('Mianus',   'Horseneck',     400200);
insert into branch  values ('Round Hill',   'Horseneck',    8000000);
insert into branch  values ('Pownal',   'Bennington',    400000);
insert into branch  values ('North Town',   'Rye',       3700000);
insert into branch  values ('Brighton', 'Brooklyn', 7000000);
insert into branch  values ('Central',  'Rye',        400280);
```

Code 4: Insert Data into Branch Table

```sql
insert into account values ('A-101',     'Downtown', 500);
insert into account values ('A-215',     'Mianus',   700);
insert into account values ('A-102',     'Perryridge',   400);
insert into account values ('A-305',     'Round Hill',   350);
insert into account values ('A-201',     'Perryridge',   900);
insert into account values ('A-222',     'Redwood',  700);
insert into account values ('A-217',     'Brighton', 750);
insert into account values ('A-333',     'Central',  850);
insert into account values ('A-444',     'North Town',   625);
```

Code 5: Insert Data into Account Table

```sql
insert into depositor values ('Johnson','A-101');
insert into depositor values ('Smith',  'A-215');
insert into depositor values ('Hayes',  'A-102');
insert into depositor values ('Hayes',  'A-101');
insert into depositor values ('Turner', 'A-305');
insert into depositor values ('Johnson','A-201');
insert into depositor values ('Jones',  'A-217');
insert into depositor values ('Lindsay','A-222');
insert into depositor values ('Majeris','A-333');
insert into depositor values ('Smith',  'A-444');
```

Code 6: Insert Data into Depositor Table

```sql
insert into loan    values ('L-17',     'Downtown', 1000);
insert into loan    values ('L-23',     'Redwood',  2000);
insert into loan    values ('L-15',     'Perryridge',   1500);
insert into loan    values ('L-14',     'Downtown', 1500);
insert into loan    values ('L-93',     'Mianus',   500);
insert into loan    values ('L-11',     'Round Hill',   900);
insert into loan    values ('L-16',     'Perryridge',   1300);
insert into loan    values ('L-20',     'North Town',   7500);
insert into loan    values ('L-21',     'Central',  570);
```

Code 7: Insert Data into Loan Table

```sql
insert into borrower values ('Jones',   'L-17');
insert into borrower values ('Smith',   'L-23');
insert into borrower values ('Hayes',   'L-15');
insert into borrower values ('Jackson', 'L-14');
insert into borrower values ('Curry',   'L-93');
insert into borrower values ('Smith',   'L-11');
insert into borrower values ('Williams','L-17');
insert into borrower values ('Adams',    'L-16');
insert into borrower values ('McBride', 'L-20');
insert into borrower values ('Smith',    'L-21');
```

Code 8: Insert Data into Borrower Table

```sql
commit;
```

Code 9: Commit Changes

The `COMMIT` command in SQL is used to save the changes made during the current transaction to the database. A transaction is a sequence of SQL statements that are executed as a single unit of work, and the COMMIT marks the successful completion of that unit.