Database Lab

MySQL

SQL

- SQL stands for **S**tructured **Q**uery **L**anguage
- A special purpose programming language designed for managing data in a relational database system
- Initially developed at IBM by Donald D. Chamberlin and Raymond F. Boyce in early 1970s
- Initially it was known as SEQUEL and later changed to SQL

MYSQL

- MySQL is an open source relational database management system
- It runs as a server to provide multi user access to a number of databases
- MySQL is a popular choice of database for use in web application

Some useful links-
https://www.tutorialspoint.com/mysql/index.htm
https://www.w3schools.com/sql/default.asp
https://www.techonthenet.com/mysql/index.php

Basic MySQL Commands

- At first, we will learn the commands to do the followings-
  - How to create a database?
  - How to add users?
  - How to create tables?
  - How to insert data into the table?
  - How to check the contents of a table?

Create Database

- It is used to create a database
- A name will be associated with it
  - Example, if we want to create a database named *dblab* then we may use the following command-
  - sql> CREATE DATABASE dblab;
- To check all databases, use *SHOW databases* command
  - sql> SHOW DATABASES;
- To create tables under a specific database, then we have to provide *USE databaseName* command
  - sql> USE DATABASE dblab;

CREATE USER

- It is used to create a new user
- For each user-
  - username and password are required
  - Example, if we want to create a user named *bob* with password *devil* then we may use the following command-
  - sql> CREATE USER 'bob'@'localhost' IDENTIFIED BY 'devil';

User bob can login locally where the server is installed

- But the new user has no permission to access anything

Granting permission

- Type of permissions
  - ALL PRIVILEGES: provides all access
  - CREATE: allows to create new tables and databases
  - DROP: allows user to delete tables or databases
  - DELETE: allows user to delete rows of tables
  - INSERT: allows user to insert rows into tables
  - SELECT: allows user to use the SELECT command
  - UPDATE: allows user to update table rows
  - GRANT OPTION: allows user to grant or remove other users' privileges

Grant & Revoke command

- Grant command is used to provide permission
- GRANT [type of permission] ON [database name].[table name] TO '[username]'@'localhost';
  - To give access to any database or any table, (*) can be used in place of database name and table name
- Revoke command is used to revoke permission
- REVOKE [type of permission] ON [database name].[table name] FROM '[username]'@'localhost';

Create Table

- To create table under a database name, we have to use *CREATE TABLE* command
- For example, to create a table for land entity-

```
sql> CREATE TABLE land(
        land_id char(10),
        address varchar(30),
        land_type varchar(15),
        region varchar(15));
```

Data types

- Frequently used text data types-
  - char(size): holds a fixed length upto 255 character
  - varchar(size): holds a variable length upto 255 character
  - text: holds a maximum length of 65,535 characters

Data types

- Frequently used number data types-
  - int(size):
    - range from -2147483648 to 2147483647 normal. 0 to 4294967295 unsigned
    - Size is maximum number of digits
  - smallint(size):
    - range from32768 to 32767 normal. 0 to 65535 unsigned
  - float(size,d):
    - A small number with a floating decimal point.
    - The maximum number of digits may be specified in the size parameter.
    - The maximum number of digits to the right of the decimal point is specified in the *d* parameter

Date types

- Frequently used date/time data types-
  - date():
    - A date. Format: YYYY-MM-DD
  - datetime():
    - A date and time combination. Format: YYYY-MM-DD HH:MM:SS
  - time():
    - A time. Format: HH:MM:SS

Describe command

- Once a table has been created, the schema definition can be seen using **DESC** (or describe) command
- Check the schema description of table *land*
  - sql> **DESC** land;

Insertion

- Newly created table is empty
- Add a new tuple to *account*

insert into *account*
values (9732, 'Park Road', 1200)

  - Insertion fails if any integrity constraint is violated

INSERTION

- Multiple rows can also be inserted using single insert command
- Example:
  - Sql> **insert into** *account*
    values (9732, 'Park Road', 1200),
    (1332, 'Buchtel Blvd', 1560),
    (1991, 'Exhibition Rd', 2560)

UPDATE

- The *UPDATE* command is used to update records in a table
- For example, if we want to update the mobile_no of a student with roll_no = '2009CS01'
  - Sql> **update** student
    set mobile_no=9876543210
    where roll_no = '2009CS01'
- When using update operation care must be taken to specify appropriate condition in *where* clause

DELETION

- The *DELETE* command is used to delete rows a table
- Example 1: to delete record from *student* table with roll_no '2009CS01'
  - Sql> **delete from** student
    where roll_no ='2009CS01'
- Example 2: to delete all records from student table
  - sql> **delete from** student

Drop command

- The **drop table** command is used to delete a table from the database
  - sql> **drop table** table_name
- The **drop database** command is used to delete a database
  - sql> **drop database** database_name

SQL Query

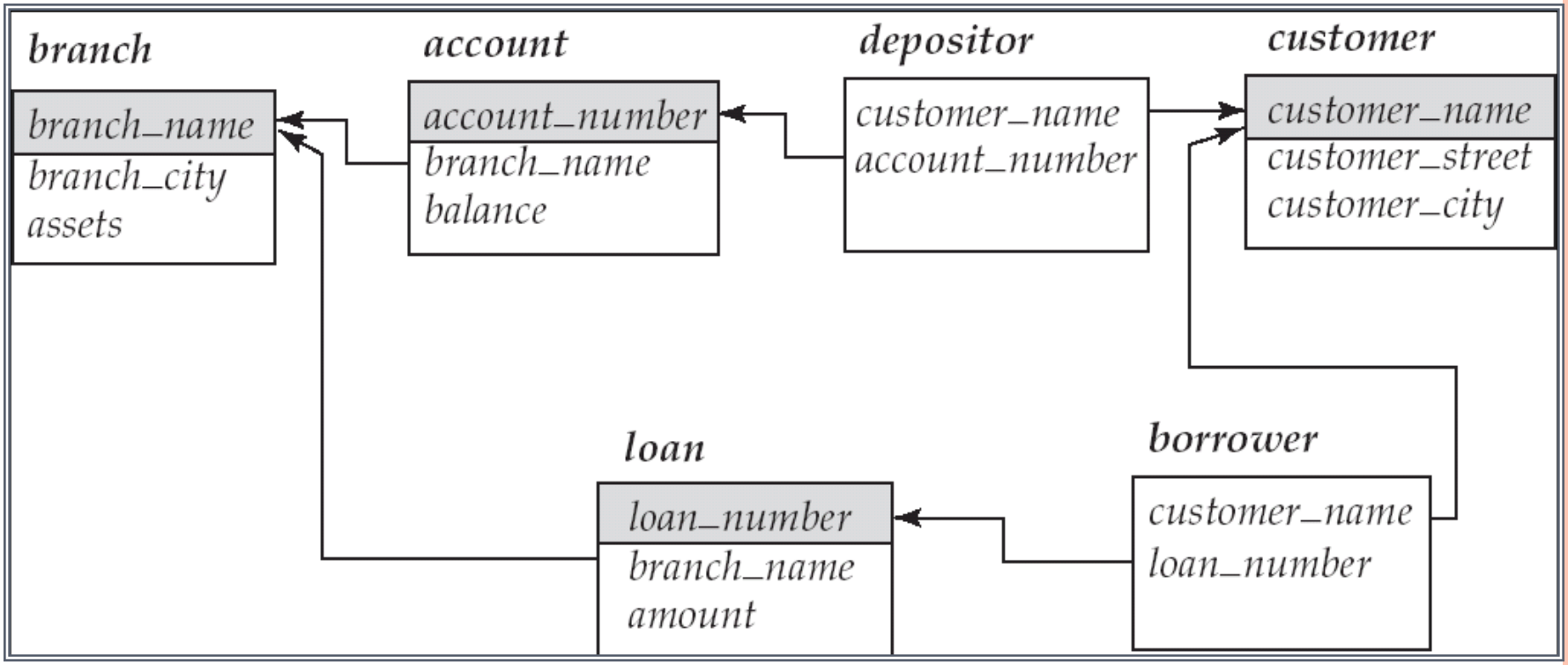
- Basic form

```
SELECT <attributes>
FROM   <one or more relations>
WHERE  <conditions>
```

SQL **commands** are case insensitive

Call this a **SFW** query.

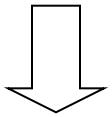
Bank Example



Simple SQL Query: Selection

Selection is the operation of filtering a relation's tuples on some condition

account_number	branch_name	balance
1009	Boring Rd	19.99
1233	PU	29.99
2124	Patliputra	149.99
8721	PU	203.99



```
SELECT *
FROM   account
WHERE  branch_name = 'PU'
```

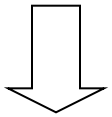
account_number	branch_name	balance
1233	PU	29.99
8721	PU	203.99

In SELECT clause, '*' indicates all attributes

Simple SQL Query: Projection

Projection is the operation of producing an output table with tuples that have a subset of their prior attributes

account_number	branch_name	balance
1009	Boring Rd	19.99
1233	PU	29.99
2124	Patliputra	149.99
8721	PU	203.99



```
SELECT account_number, balance
FROM   account
WHERE  branch_name = 'PU'
```

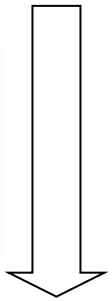
account_number	balance
1233	29.99
8721	203.99

Notation

Input schema

account(account_number, branch_name, balance)

```
SELECT account_number, balance
FROM   account
WHERE  branch_name = 'PU'
```



Output schema

Answer(account_number, balance)

SQL Query: duplicate elimination

- SQL allows duplicates in relations as well as in query results
- To force the elimination of duplicates, insert the keyword **distinct** after SELECT clause.
- Find the names of all branches in the *account* relations, and remove duplicates

  SELECT **DISTINCT** *branch_name*
  FROM *account*

More features of Select clause

- The **SELECT** clause can contain arithmetic expressions involving the operation, +, –, *, and /, and operating on constants or attributes of tuples.
- E.g.:

SELECT *account_number, branch_name, balance* * 100
FROM *account*

The where Clause

- The **where** clause can include multiple conditions
- To find all loan number for loans made at the "Patliputra" branch with loan amounts greater than1200.

```
SELECT loan_number
FROM loan
WHERE branch_name = 'Patliputra'  and amount > 1200
```

- Comparison results can be combined using the logical connectives **and, or,** and **not.**

The from Clause

- The **from** clause can also include multiple relations involved in the query
- Find all the possible combinations of *borrower and loan* relations
  - **SELECT** *
    FROM *borrower, loan*
- Find the name, loan number and loan amount of all customers having a loan at the Patliputra branch.
  - **SELECT** *customer_name, borrower.loan_number, amount*
    FROM *borrower, loan*
    WHERE *borrower.loan_number = loan.loan_number*
    AND *branch_name* = 'Patliputra'

The Rename Operation

- SQL allows renaming relations and attributes using the **as** clause:

  old-name **as** *new-name*

- E.g. Find the name, loan number and loan amount of all customers; rename the column name *loan_number* as *loan_id*.

SELECT *customer_name, borrower.loan_number* **AS** *loan_id, amount*
FROM *borrower, loan*
WHERE *borrower.loan_number = loan.loan_number*

End of First Part

Integrity Constraints on Tables

- **not null**
- **unique**
- **primary key**
- **foreign key**

Specifying primary key

- Example:

```
create table branch(
    branch_name   char(15) primary key,
    branch_city         char(30) not null,
    assets              integer);
```

- Example:

```
create table branch(
    branch_name   char(15),
    branch_city         char(30),
    assets              integer,
   primary key (branch_name));
```

Specifying Primary Key

```
create table branch(
        branch_name   char(15),
        branch_city   char(30),
        assets        integer,
        constraint branch_pk
        primary key (branch_name));
```

Explicitly mentioning the constraint name

SPECIFYING FOREIGN KEY

- Example:
  - **Sql> create table** *account(*

account_no **integer primary key**,
branch_name **char**(15)**,**
branch_city **char(**30),
assets **float(8,2),**

constraint account_fk1 foreign key (branch_name)
references branch(branch_name))

SPECIFYING UNIQUE CONSTRAINT

- Like primary key, *UNIQUE* constraint uniquely identifies each record in a database table
- Unlike primary key, a table can have more than one *UNIQUE* constraint
- Example:
  - Sql> **create table** student(
    roll_no **char(10)** ,
    name **varchar(20) not null**,
    mobile_no **int(10) unique**,
    primary key (roll_no));

ALTER TABLE

- The **alter table** command is used to add attributes to an existing relation:
  - **Sql> alter table** *student*
    add *dob date*;
  - All tuples in the relation are assigned *null* as the value for the new attribute.
  - **Sql> alter table** *student*
    add *dob date* **first**;

  Positions the new column at the beginning
  - **Sql> alter table** *student*
    add *dob date* **after stud_name**;

  Positions the new column after a specific column

Alter Table

- The **alter table** command is used to drop attributes from an existing relation:
  - **Sql> alter table** *student*
    drop *dob;*

  Drops the specified column

  However, this drop clause will not work if the column is the only one left in the table

Alter Table

- The **alter table** command can also be used to modify existing attributes of a relation:
- Suppose we want to increase the size of name attribute of student relation from 15 to 30
- Using modify clause
  - **Sql> alter table** *student*

    modify *name varchar(30)*
- Using change clause
  - **Sql> alter table** *student*

    change *name stud_name varchar(30)*

  Allows to change the column name as well

Alter Table

- Changing the column's default value
  - **Sql> alter table** *account*
    alter *balance* **set default** *100;*

- Removing the column's default constraint
  - **Sql> alter table** *account*
    alter *balance* **drop default**;

Alter Table

- Renaming a table
  - **Sql> alter table** *account*
    rename to *acounts;*

Adding primary key

- Command **alter table** can be used to add primary key in an existing table
- Example:

```
alter table ac_type
  add primary key (account_no)
```

or

```
alter table ac_type
  add constraint ac_type_pk primary key
(account_no)
```

Adding foreign key

- Foreign key can also be added in an existing table using **alter table** command
- Example:

alter table ac_type

add foreign key (account_no) **references** account (account_no)

or

alter table ac_type

add constraint ac_type_fk1 foreign key (account_no) **references** account (account_no)

Dropping Primary key

- One relation can have at most one primary key. So without using the primary key name, the constraint can be dropped.
- Example:

```
alter table ac_type
  drop primary key
```

Dropping foreign key

- Foreign key needs to be dropped using the foreign key name
- Example:

```
alter table ac_type
  drop foreign key ac_type_fk1;
```

To check the constraint name

- The following command can be used to check the constraint names of a table
- SHOW CREATE TABLE account;

Operator 'like'

- 'like' operator can be used to compare a pattern
- For character data types, two wild cards are frequently used-
  - '%': allows to match any string of length (including zero)
  - '_' :allows to match a single character
- Example:

```
SELECT * FROM customer
WHERE name LIKE '_avi%'
```

Operator 'not like'

- Similar to like operator but in this case we are interested in not selecting some specific pattern using NOT LIKE operator
- Example:

```
SELECT * from CUSTOMER
WHERE name NOT LIKE '_avi%'
```

REGEXP

- regexp can be used to catch some specific patterns
- Following characters can be used
  - ^ to match beginning
  - $ to match end
  - . to match each instance
  - a* matches 0 or more instances of a
  - a+ matches 1 or more instances of a
  - [abc] matches patterns containing a,b,or c
  - [a-e] matches characters say a to e
  - [^a-e] not matches a-e characters
  - ^…$ matches exactly 3 characters
  - ^.{3}$ also matches exactly 3 characters

REGEXP

- Case insensitive pattern
- Example: Find the names which contain ‘w’ (w or W)
  - **SELECT** cust_name
    FROM customer
    WHERE cust_name REGEXP ‘w‘
- Case sensitive pattern
- Use ‘binary’ option
- Example: Find the names which contain ‘w’ (lowercase w)
  - **SELECT** cust_name
    FROM customer
    WHERE cust_name REGEXP binary ‘w‘

Operator 'in'

- Allows to specify multiple values in a 'where' clause
- Example:

```
SELECT *
FROM customers
WHERE city IN ('Dhanbad', 'New Delhi')
```

Operator 'between'

- To select a range of data between two values in a 'where' clause
- The values can be number, text or dates
- Example:

```
SELECT * FROM account
WHERE account_no BETWEEN 1234 and 5555
```

BETWEEN operator is inclusive, i.e., includes both start and end values

Operator 'not between'

- Not to select a range of data between two values in a 'where' clause
- Example:

```
SELECT * FROM account
WHERE account_no NOT BETWEEN 1234 and 5555
```

Aggregate functions

- SQL aggregate functions return a single value, calculated from values in a column.
- Some useful functions are
  - AVG() - Returns the average value
  - COUNT() - Returns the number of rows
  - MAX() - Returns the largest value
  - MIN() - Returns the smallest value
  - SUM() - Returns the sum

Aggregate Function

- AVG function returns the average value of a numeric column
  - SELECT AVG(amount) as avg
    FROM account
- COUNT function returns the number of rows that match a given criterion
  - SELECT COUNT(*) FROM account; //counts the number of rows in the table
  - SELECT COUNT(account_number) FROM account;

Aggregate Function

- SUM function returns the total sum of a numeric column
  - SELECT SUM(amount) as total
    FROM account
- MAX function returns the largest value of a selected column
  - SELECT MAX(balance) FROM account;
- MIN function returns the smallest value of a selected column
  - SELECT MIN(balance) FROM account;

ORDER BY CLAUSE

Used to sort the results

```
SELECT   account_number, balance
FROM     account
WHERE    branch_name='PU'
ORDER BY balance, account_number
```

Ties are broken by the second attribute on the ORDER BY list, etc.

Ordering is ascending, unless you specify the DESC keyword.

'GROUP BY' CLAUSE

- Used to group the result set
- Used in conjunction with aggregate functions
- Example: suppose we want to count the number of accounts owned by a customer from depositor relation
  - SELECT customer_name, COUNT(account_no)
    FROM depositor
    GROUP BY customer_name

'HAVING' CLAUSE

- 'where' clause could not be used with the aggregate functions. Alternatively, 'having' clause can be used
- In the previous example, if we have to show the customer names who have more than one account
  - SELECT customer_name, COUNT(account_no)
    FROM depositor
    GROUP BY customer_name
    HAVING COUNT(account_no)>1

Having clause: Another example

- Both where and having can be used in one query
- Suppose, we have to find those customers who have more than one account in 'Boring Rd' branch
  - SELECT customer_name, COUNT(depositor.account_no)
    FROM depositor, account
    WHERE depositor.account_no = account.account_no AND account.branch_name = 'Boring Rd'
    GROUP BY customer_name
    HAVING COUNT(depositor.account_no)>1;

Inner Join

- A join clause is used to combine rows of two tables based on a related column between them
- INNER JOIN or JOIN both are same
- E.g.
  - SELECT depositor.*customer_name*
    FROM *depositor*
    INNER JOIN *borrower*
    ON *depositor.customer_name=borrower.customer_name*;

LEFT JOIN

- The LEFT JOIN returns all records from the left table (first table), and the matched records from the right table (second table). The result is NULL from the right side, if there is no match.
- SELECT depositor.*customer_name,*
  borrower.*customer_name*
  FROM *depositor*
  LEFT JOIN *borrower*
  ON *depositor.customer_name=borrower.customer_name*;

RIGHT JOIN

- The RIGHT JOIN returns all records from the right table (second table), and the matched records from the left table (first table). The result is NULL from the left side, if there is no match.

- SELECT depositor.*customer_name,*
  borrower.*customer_name*
  FROM *depositor*
  RIGHT JOIN *borrower*
  ON *depositor.customer_name=borrower.customer_name*;

UNION OPERATION

- Used to combine two result sets
- In UNION operation,
  - Select clause must have the same number of columns.
  - The columns must also have similar data types.
  - Also, the columns in each SELECT statement must be in the same order.
  - E.g

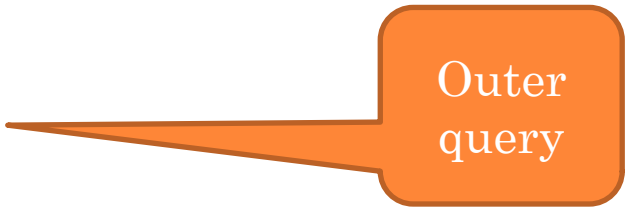
SELECT *customer_name* FROM *depositor* UNION
SELECT *customer_name* FROM *borrower*;

UNION OPERATION

- UNION by default select the distinct values
- UNION ALL will select the duplicate values also
- Mysql does not support INTERSECT and MINUS operation
  - Nested query can be used to implement these functions

Nested Query

- A subquery or nested query is a select statement inside another statement
- SELECT *
  FROM t1
  WHERE column1 in (
  SELECT column1
  FROM t2);



A sub query can be nested within another sub query

For joining two relations

- **SELECT** customer_name
  FROM depositor
  WHERE customer_name **IN** (
  SELECT customer_name
  FROM borrower)
- An equivalent to set intersection operation

Equivalent minus operation

- **SELECT** cust_name
  FROM depsositor
  WHERE cust_name **NOT IN** (
  SELECT cust_name
  FROM borrower)
- An equivalent to minus operation

EXISTS/ NOT EXISTS

- SELECT column1 FROM t1 WHERE EXISTS (SELECT * FROM t2);
- If a subquery returns any rows at all then the EXISTS subquery is true and NOT EXISTS subquery is false

- **SELECT** customer_name
  FROM depositor
  WHERE EXISTS (
  SELECT *
  FROM borrower
  WHERE borrower.customer_name = depositor.customer_name)

A query is called correlated nested query when both the inner query and the outer query are interdependent.

Subqueries with the INSERT Statement:

- INSERT INTO cust_bkp
  SELECT * FROM customers
  WHERE customer_name IN (
  SELECT customer_name FROM depositor) ;

Subqueries with the UPDATE Statement:

- UPDATE customers
  SET salary = salary * 0.25
  WHERE age IN (
      SELECT age FROM cust_bkp
      WHERE age >= 27);

Subqueries with the DELETE Statement:

- DELETE FROM customers
  WHERE age IN (
  SELECT age FROM cust_bkp
  WHERE age > 27);