**1.6 LAB - MySQL Workbench review (Sakila)**

**The purpose of this lab is to gain familiarity with MySQL Workbench. The lab also ensures the correct version of the Sakila sample database is installed on your computer, for use in other zyLabs.**

**This lab has three parts:**

**Install the Sakila database.**

**Run a simple query.**

**Recreate a Sakila table in the zyLab environment.**

**Only the third part is graded.**

**Install the Sakila database**

**This lab requires access to MySQL Server via MySQL Workbench. Most students install and access MySQL Server and MySQL Workbench on their personal computer. Installation instructions are available at MySQL Server installation and MySQL Workbench installation.**

**To create Sakila tables in MySQL, download the Sakila schema file, open MySQL Workbench, and click the following menu commands:**

**Click 'File' > 'Open SQL Script…' and open the Sakila schema file.**

**Click 'Query' > ''Execute (All or Selection)'.**

**To load sample data to the Sakila tables, download the Sakila data file and repeat steps 1 and 2 with this file.**

**Run a simple query**

**Refer to the following MySQL Workbench screenshot, taken from a Mac computer. Workbench looks slightly different on Windows.**

**The image is a screenshot of MySQL Workbench running on a Mac computer. The Schemas tab is highlighted in a box. Below Schemas, two circular arrows representing a screen refresh operation are highlighted in a circle. On the left, sakila is highlighted in a box. Below sakila and indented, Tables is highlighted in a box. Below Tables and indented, film is highlighted in a box. A box of commands appears to the right of film. At the top of the box, the phrase Select Rows - limit 1000 is highlighted in a box.**

**If 'sakila' does not appear under 'Schemas', click the refresh icon, in the red circle above. If 'sakila' still does not appear, repeat the installation process or request assistance.**

**Depending on Workbench configuration, a different Limit may appear after 'Select Rows'.**

**When 'sakila' appears under 'Schemas':**

**Click > to expand 'sakila'.**

**Click > to expand 'Tables'.**

**Right-click 'film'.**

**Click 'Select Rows - Limit 1000'.**

**MySQL Workbench executes SELECT \* FROM film; and displays 1000 films:**

**The image is a screenshot of MySQL Workbench running on a Mac computer. On the left is a list of tables in the Sakila database. In the center, within a query panel, is the statement SELECT \* FROM sakila.film;. Below the query panel is a result panel containing the first nine rows of the film table, with a vertical scroll bar so that additional rows can be viewed.**

**Recreate a Sakila table in the zyLab environment**

**To recreate the actor table in the zyLab environment:**

**Right-click 'actor'.**

**Select 'Copy to Clipboard' > 'Create Statement' to copy the CREATE TABLE statement to your clipboard.**

**Paste the CREATE TABLE statement into the zyLab Main.sql box.**

**Delete the following characters for compatibility with the zyLab environment:**

**COLLATE=utf8mb4\_0900\_ai\_ci**

**all apostrophes (`)**

**The CREATE TABLE statement creates actor columns, keys, and indexes. No result is displayed in Develop mode. The tests in Submit mode verify that the zyLab and Sakila actor tables are identical.**

-- Your CREATE TABLE statement goes here

CREATE TABLE actor (

actor\_id smallint unsigned NOT NULL AUTO\_INCREMENT,

first\_name varchar(45) NOT NULL,

last\_name varchar(45) NOT NULL,

last\_update timestamp NOT NULL DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP,

PRIMARY KEY (actor\_id),

KEY idx\_actor\_last\_name (last\_name)

) ENGINE=InnoDB AUTO\_INCREMENT=201 DEFAULT CHARSET=utf8mb4;

**2.13 LAB - Create Movie table**

**Create a Movie table with the following columns:**

**ID - positive integer with maximum value of 50,000**

**Title - variable-length string with up to 50 characters**

**Rating - fixed-length string with 4 characters**

**ReleaseDate - date**

**Budget - decimal value representing a cost of up to 999,999 dollars, with 2 digits for cents**

-- Your SQL statement goes here

CREATE TABLE Movie (

ID SMALLINT(5) UNSIGNED,

Title VARCHAR(50) NOT NULL,

Rating CHAR(4),

ReleaseDate DATE,

Budget DECIMAL (8,2)

);

**2.14 LAB - Alter Movie table**

**The Movie table has the following columns:**

**ID - positive integer**

**Title - variable-length string**

**Genre - variable-length string**

**RatingCode - variable-length string**

**Year - integer**

**Write ALTER statements to make the following modifications to the Movie table:**

**Add a Producer column with VARCHAR data type (max 50 chars).**

**Remove the Genre column.**

**Change the Year column's name to ReleaseYear, and change the data type to SMALLINT.**

-- Your SQL statements go here

ALTER TABLE Movie

ADD COLUMN Producer VARCHAR(50),

DROP COLUMN Genre,

CHANGE COLUMN Year ReleaseYear SMALLINT;

**2.15 LAB - Select horses with logical operators**

**The Horse table has the following columns:**

**ID - integer, primary key**

**RegisteredName - variable-length string**

**Breed - variable-length string**

**Height - decimal number**

**BirthDate - date**

**Write a SELECT statement to select the registered name, height, and birth date for only horses that have a height between 15.0 and 16.0 (inclusive) or have a birth date on or after January 1, 2020.**

-- Your SELECT statement goes here

SELECT RegisteredName, Height, BirthDate

FROM Horse

WHERE (Height >= 15 AND Height <= 16)

OR BirthDate >= '2020-01-01';

2.16 LAB - Insert rows into Horse table

**The Horse table has the following columns:**

**ID - integer, auto increment, primary key**

**RegisteredName - variable-length string**

**Breed - variable-length string, must be one of the following: Egyptian Arab, Holsteiner, Quarter Horse, Paint, Saddlebred**

**Height - decimal number, must be between 10.0 and 20.0**

**BirthDate - date, must be on or after Jan 1, 2015**

**Insert the following data into the Horse table:**

| RegisteredName | Breed | Height | BirthDate |
| --- | --- | --- | --- |
| Babe | Quarter Horse | 15.3 | 2015-02-10 |
| Independence | Holsteiner | 16.0 | 2017-03-13 |
| Ellie | Saddlebred | 15.0 | 2016-12-22 |
| *NULL* | Egyptian Arab | 14.9 | 2019-10-12 |

-- Your SQL statements goes here

INSERT INTO Horse(RegisteredName, Breed, Height, Birthdate)

VALUES('Babe', 'Quarter Horse', 15.3, '2015-02-10'),

('Independence', 'Holsteiner', 16.0, '2017-03-13'),

('Ellie', 'Saddlebred', 15.0, '2016-12-22'),

(NULL, 'Egyptian Arab', 14.9, '2019-10-12');

**2.17 LAB - Update rows in Horse table**

**The Horse table has the following columns:**

**ID - integer, auto increment, primary key**

**RegisteredName - variable-length string**

**Breed - variable-length string, must be one of the following: Egyptian Arab, Holsteiner, Quarter Horse, Paint, Saddlebred**

**Height - decimal number, must be ≥ 10.0 and ≤ 20.0**

**BirthDate - date, must be ≥ Jan 1, 2015**

**Make the following updates:**

**Change the height to 15.6 for horse with ID 2.**

**Change the registered name to Lady Luck and birth date to May 1, 2015 for horse with ID 4.**

**Change every horse breed to NULL for horses born on or after December 22, 2016.**

-- Your SQL statements goes here

UPDATE Horse

SET Height = 15.6

WHERE ID = 2;

UPDATE Horse

SET RegisteredName = "Lady Luck", BirthDate = '2015-05-01'

WHERE ID = 4;

UPDATE Horse

SET Breed = NULL

WHERE BirthDate >= '2016-12-22';

**2.18 LAB - Delete rows from Horse table**

**The Horse table has the following columns:**

**ID - integer, auto increment, primary key**

**RegisteredName - variable-length string**

**Breed - variable-length string**

**Height - decimal number**

**BirthDate - date**

**Delete the following rows:**

**Horse with ID 5.**

**All horses with breed Holsteiner or Paint.**

**All horses born before March 13, 2013.**

**Note: Your SQL code does not display any results in Develop mode. Use Submit mode to test your code.**

-- Your SQL statements goes here

DELETE FROM Horse

WHERE ID = 5;

DELETE FROM Horse

WHERE Breed = "Holsteiner";

DELETE FROM Horse

WHERE Breed = "Paint";

DELETE FROM Horse

WHERE BirthDate < '2013-03-13';

**2.19 LAB - Create Horse table with constraints**

**Create a Horse table with the following columns, data types, and constraints. NULL is allowed unless 'not NULL' is explicitly stated.**

**ID - integer with range 0 to 65 thousand, auto increment, primary key**

**RegisteredName - variable-length string with max 15 chars, not NULL**

**Breed - variable-length string with max 20 chars, must be one of the following: Egyptian Arab, Holsteiner, Quarter Horse, Paint, Saddlebred**

**Height - number with 3 significant digits and 1 decimal place, must be ≥ 10.0 and ≤ 20.0**

**BirthDate - date, must be ≥ Jan 1, 2015**

**Notes: Not all constraints can be tested due to current limitations of MySQL. Your SQL code does not display any results in Develop mode. Use Submit mode to test your code.**

-- Your SQL statement goes here

CREATE TABLE Horse (

ID SMALLINT UNSIGNED AUTO\_INCREMENT,

RegisteredName VARCHAR(15) NOT NULL,

Breed VARCHAR(20) CHECK (Breed IN ('Egyptian Arab', 'Quarter Horse', 'Holsteiner',

'Paint', 'Saddlebred')),

Height DECIMAL(3,1) CHECK (Height >= 10.0 AND Height <= 20.0),

BirthDate DATE CHECK (Birthdate >= '2015-01-01'),

PRIMARY KEY (ID)

);

**2.20 LAB - Create Student table with constraints**

**Create a Student table with the following column names, data types, and constraints:**

**ID - integer with range 0 to 65 thousand, auto increment, primary key**

**FirstName - variable-length string with max 20 chars, not NULL**

**LastName - variable-length string with max 30 chars, not NULL**

**Street - variable-length string with max 50 chars, not NULL**

**City - variable-length string with max 20 chars, not NULL**

**State - fixed-length string of 2 chars, not NULL, default "TX"**

**Zip - integer with range 0 to 16 million, not NULL**

**Phone - fixed-length string of 10 chars, not NULL**

**Email - variable-length string with max 30 chars, must be unique**

**Note: Your SQL code does not display any results in Develop mode. Use Submit mode to test your code.**

-- Your SQL statements go here

CREATE TABLE Student (

ID SMALLINT UNSIGNED AUTO\_INCREMENT,

FirstName VARCHAR(20) NOT NULL,

LastName VARCHAR(30) NOT NULL,

Street VARCHAR(50) NOT NULL,

City VARCHAR(20) NOT NULL,

State CHAR(2) NOT NULL DEFAULT "TX",

Zip MEDIUMINT UNSIGNED NOT NULL,

Phone CHAR(10) NOT NULL,

Email VARCHAR(30) UNIQUE,

PRIMARY KEY (ID)

);

**2.21 LAB - Create LessonSchedule table with FK constraints**

**Two tables are created:**

**Horse with columns:**

**ID - integer, primary key**

**RegisteredName - variable-length string**

**Student with columns:**

**ID - integer, primary key**

**FirstName - variable-length string**

**LastName - variable-length string**

**Create the LessonSchedule table with columns:**

**HorseID - integer with range 0 to 65 thousand, not NULL, partial primary key, foreign key references Horse(ID)**

**StudentID - integer with range 0 to 65 thousand, foreign key references Student(ID)**

**LessonDateTime - date/time, not NULL, partial primary key**

**If a row is deleted from Horse, the rows with the same horse ID should be deleted from LessonSchedule automatically.**

**If a row is deleted from Student, the same student IDs should be set to NULL in LessonSchedule automatically.**

**Notes: Table and column names are case sensitive in the auto-grader. Your SQL code does not display any results in Develop mode. Use Submit mode to test your code.**

CREATE TABLE Horse (

ID SMALLINT UNSIGNED AUTO\_INCREMENT,

RegisteredName VARCHAR(15),

PRIMARY KEY (ID)

);

CREATE TABLE Student (

ID SMALLINT UNSIGNED AUTO\_INCREMENT,

FirstName VARCHAR(20),

LastName VARCHAR(30),

PRIMARY KEY (ID)

);

-- Your SQL statements go here

CREATE TABLE LessonSchedule (

HorseID SMALLINT UNSIGNED NOT NULL,

StudentID SMALLINT UNSIGNED,

LessonDateTime DATETIME NOT NULL,

PRIMARY KEY (HorseID, LessonDateTime),

FOREIGN KEY (HorseID) REFERENCES Horse(ID)

ON DELETE CASCADE,

FOREIGN KEY (StudentID) REFERENCES Student(ID)

ON DELETE SET NULL

);

**3.10 LAB - Select number of movies grouped by year**

**The Movie table has the following columns:**

**ID - integer, primary key**

**Title - variable-length string**

**Genre - variable-length string**

**RatingCode - variable-length string**

**Year - integer**

**Write a SELECT statement to select the year and the total number of movies for that year.**

**Hint: Use the COUNT() function and GROUP BY clause.**

-- Your SELECT statement goes here

SELECT Year,

COUNT(ID) Total\_Movies

FROM Movie

GROUP by Year;

**3.11 LAB - Select movie ratings with left join**

**The Movie table has the following columns:**

**ID - integer, primary key**

**Title - variable-length string**

**Genre - variable-length string**

**RatingCode - variable-length string**

**Year - integer**

**The Rating table has the following columns:**

**Code - variable-length string, primary key**

**Description - variable-length string**

**Write a SELECT statement to select the Title, Year, and rating Description. Display all movies, whether or not a RatingCode is available.**

**Hint: Perform a LEFT JOIN on the Movie and Rating tables, matching the RatingCode and Code columns.**

-- Your SELECT statement goes here

SELECT Title, Year, Description

FROM Movie

LEFT JOIN Rating ON RatingCode = Code;

**3.12 LAB - Select employees and managers with inner join**

**The Employee table has the following columns:**

**ID - integer, primary key**

**FirstName - variable-length string**

**LastName - variable-length string**

**ManagerID - integer**

**Write a SELECT statement to show a list of all employees' first names and their managers' first names. List only employees that have a manager. Order the results by Employee first name. Use aliases to give the result columns distinctly different names, like "Employee" and "Manager".**

**Hint: Join the Employee table to itself using INNER JOIN.**

-- Your SELECT statement goes here

SELECT A.FirstName AS 'Employee', B.FirstName AS 'Manager'

FROM Employee A

INNER JOIN Employee B

ON B.ID = A.ManagerID

ORDER BY A.FirstName ASC;

**3.13 LAB - Select lesson schedule with inner join**

**The database has three tables for tracking horse-riding lessons:**

**Horse with columns:**

**ID - primary key**

**RegisteredName**

**Breed**

**Height**

**BirthDate**

**Student with columns:**

**ID - primary key**

**FirstName**

**LastName**

**Street**

**City**

**State**

**Zip**

**Phone**

**EmailAddress**

**LessonSchedule with columns:**

**HorseID - partial primary key, foreign key references Horse(ID)**

**StudentID - foreign key references Student(ID)**

**LessonDateTime - partial primary key**

**Write a SELECT statement to create a lesson schedule with the lesson date/time, horse ID, and the student's first and last names. Order the results in ascending order by lesson date/time, then by horse ID. Unassigned lesson times (student ID is NULL) should not appear in the schedule.**

**Hint: Perform a join on the Student and LessonSchedule tables, matching the student IDs.**

-- Your SELECT statement goes here

SELECT LessonDateTime, HorseID, FirstName, LastName

FROM LessonSchedule

JOIN Student ON LessonSchedule.StudentID = Student.ID

ORDER BY LessonDateTime ASC, HorseID;

**3.14 LAB - Select lesson schedule with multiple joins**

**The database has three tables for tracking horse-riding lessons:**

**Horse with columns:**

**ID - primary key**

**RegisteredName**

**Breed**

**Height**

**BirthDate**

**Student with columns:**

**ID - primary key**

**FirstName**

**LastName**

**Street**

**City**

**State**

**Zip**

**Phone**

**EmailAddress**

**LessonSchedule with columns:**

**HorseID - partial primary key, foreign key references Horse(ID)**

**StudentID - foreign key references Student(ID)**

**LessonDateTime - partial primary key**

**Write a SELECT statement to create a lesson schedule for Feb 1, 2020 with the lesson date/time, student's first and last names, and the horse's registered name. Order the results in ascending order by lesson date/time, then by the horse's registered name. Make sure unassigned lesson times (student ID is NULL) appear in the results.**

**Hint: Perform a join on the LessonSchedule, Student, and Horse tables, matching the student IDs and horse IDs.**

-- Your SELECT statement goes here

SELECT LessonDateTime, FirstName, LastName, RegisteredName

FROM LessonSchedule

LEFT JOIN Student ON Student.ID = StudentID

INNER JOIN Horse ON Horse.ID = HorseID

WHERE LessonDateTime LIKE "%2020-02-01%"

ORDER BY LessonDateTime ASC, RegisteredName;

**3.15 LAB - Select tall horses with subquery**

**The Horse table has the following columns:**

**ID - integer, primary key**

**RegisteredName - variable-length string**

**Breed - variable-length string**

**Height - decimal number**

**BirthDate - date**

**Write a SELECT statement to select the registered name and height for only horses that have an above average height. Order the results by height.**

**Hint: Use a subquery to find the average height.**

-- Your SQL statements go here

SELECT RegisteredName, Height

FROM Horse

WHERE Height > (SELECT AVG (Height) FROM Horse)

ORDER BY Height;

**4.12 LAB - Multiple joins with aggregate (Sakila)**

**Refer to the film, actor, and film\_actor tables of the Sakila database. The tables in this lab have the same columns and data types but fewer rows.**

**Write a query that:**

**Computes the average length of all films that each actor appears in.**

**Rounds average length to the nearest minute and renames the result column "average".**

**Displays last name, first name, and average, in that order, for each actor.**

**Sorts the result in descending order by average, then ascending order by last name.**

**The query should exclude films with no actors and actors that do not appear in films.**

**Hint: Use the ROUND() and AVG() functions.**

SELECT a.last\_name, a.first\_name,

round(avg(f.length)) as "average"

FROM actor a

JOIN film\_actor fa

ON a.actor\_id = fa.actor\_id

JOIN film f ON fa.film\_id = f.film\_id

GROUP BY a.last\_name, a.first\_name

ORDER BY average DESC, a.last\_name ASC;

**4.13 LAB - Nested aggregates - MIN of COUNT (Sakila)**

**Refer to the film and inventory tables of the Sakila database. The tables in this lab have the same columns and data types but fewer rows.**

**Write a query that lists the titles of films with the fewest rows in the inventory table.**

**This query requires a subquery that computes the minimum of counts by film\_id:**

**SELECT MIN(count\_film\_id)**

**FROM ( SELECT COUNT(film\_id) AS count\_film\_id**

**FROM inventory**

**GROUP BY film\_id )**

**AS temp\_table;**

**This subquery is provided in the template.**

-- Your SELECT statement goes here

Select film.title

From film

Right Join inventory

On film.film\_id = inventory.film\_id

Group by film.film\_id

Having count(\*) =

-- Use the following subquery:

( SELECT MIN(count\_film\_id) FROM

( SELECT COUNT(film\_id) AS count\_film\_id

FROM inventory

GROUP BY film\_id )

AS temp\_table );

**4.13 LAB - Implement strong entity (Sakila)**

**Implement a new strong entity phone in the Sakila database. Attributes and relationships are shown in the following diagram:**

**The phone entity appears on the right. The phone entity contains four attributes, each followed by cardinality information: phone\_id 1-1(1), country\_code M-1(1), phone\_numer M-1(1), and phone\_type M-1(0). Three entities appear on the left: store, staff, and customer, connected to the phone entity by three identical relationships. The three relationships are named 'has' and have cardinality 1(0) on both sides.**

**The diagram uses Sakila naming conventions. Follow the Sakila conventions for your table and column names:**

**All lower case**

**Underscore separator between root and suffix**

**Foreign keys have the same name as referenced primary key**

**Write CREATE TABLE and ALTER TABLE statements that:**

**Implement the entity as a new phone table.**

**Implement the has relationships as foreign keys in the Sakila customer, staff, and store tables.**

**Remove the existing phone column from the Sakila address table.**

**Step 2 requires adding a foreign key constraint to an existing table. Ex:**

**ALTER TABLE customer**

**ADD FOREIGN KEY (phone\_id) REFERENCES phone(phone\_id)**

**ON DELETE SET NULL**

**ON UPDATE CASCADE;**

**Specify data types as follows:**

**phone\_id, phone\_number, and country\_code have data type INT.**

**phone\_type has date type VARCHAR(12) and contains strings like 'Home', 'Mobile', and 'Other'.**

**Apply these constraints:**

**NOT NULL constraints correspond to cardinalities on the diagram above.**

**Foreign key actions are SET NULL for delete rules and CASCADE for update rules.**

**Specify a suitable column as the phone table primary key.**

-- Your CREATE TABLE and ALTER TABLE statements go here

CREATE TABLE phone (

phone\_id INT NOT NULL PRIMARY KEY,

country\_code INT NOT NULL,

phone\_number INT NOT NULL,

phone\_type VARCHAR(12) CHECK (phone\_type IN('Home', 'Mobile', 'Other'))

);

ALTER TABLE customer

ADD phone\_id INT,

ADD FOREIGN KEY(phone\_id) REFERENCES phone(phone\_id)

ON DELETE SET NULL

ON UPDATE CASCADE;

ALTER TABLE staff

ADD phone\_id INT,

ADD FOREIGN KEY(phone\_id) REFERENCES phone(phone\_id)

ON DELETE SET NULL

ON UPDATE CASCADE;

ALTER TABLE store

ADD phone\_id INT,

ADD FOREIGN KEY(phone\_id) REFERENCES phone(phone\_id)

ON DELETE SET NULL

ON UPDATE CASCADE;

ALTER TABLE address

DROP COLUMN phone;

4.14 LAB - Implement supertype and subtype entities (Sakila)

Refer to the customer and staff tables of the Sakila database. These tables have many columns in common and represent similar entities. Convert the customer and staff entities into subtypes of a new supertype person:

In the center is the person entity, with primary key person\_id and additional attributes first\_name, last\_name, email, active, and last\_update. The person entity contains subtype entities staff and customer. The staff entity has primary key person\_id and additional attributes picture, username, and password. The customer entity has primary key person\_id and additional attribute create\_date. Cardinality does not appear after the primary keys and attributes. On the left is the address entity, connected to the person entity by the belongs\_to relationship. Belongs\_to has cardinality 1(1) on the address side and M(0) on the person side. On the right is the store entity, connected to the person entity by the works\_at relationship. Works\_at has cardinality 1(1) on the store side and M(0) on the person side.

The diagram uses Sakila naming conventions. Follow the Sakila conventions for your table and column names:

All lower case

Underscore separator between root and suffix

Foreign keys have the same name as referenced primary key

Implement supertype and subtype entities as person, customer, and staff tables with primary key person\_id.

Implement attributes as columns:

All columns are NOT NULL.

The person\_id columns have data type SMALLINT UNSIGNED.

The last\_update and create\_date columns have data type TIMESTAMP.

The picture column has data type BLOB.

All other columns have data type VARCHAR(20).

Implement the dependency relationships between subtype and supertype entities as foreign keys:

The person\_id columns of customer and staff become foreign keys referring to person.

Specify CASCADE actions for both relationships.

Implement the belongs\_to and works\_at relationships as foreign keys:

belongs\_to becomes an address\_id foreign key in person referring to address.

works\_at becomes a store\_id foreign key in staff referring to store.

Specify RESTRICT actions for both relationships.

The address and store tables, with primary keys address\_id and store\_id, are pre-defined in the zyLab environment. Foreign keys must have the same data types as the referenced primary keys:

address\_id has data type SMALLINT UNSIGNED.

store\_id has data type TINYINT UNSIGNED.

If you execute your solution with the Sakila database, you must first drop customer, staff, and all constraints that refer to these tables. Use the following statements with Sakila only, not in the zyLab environment:

DROP TABLE customer, staff;

ALTER TABLE payment

DROP CONSTRAINT fk\_payment\_customer,

DROP CONSTRAINT fk\_payment\_staff;

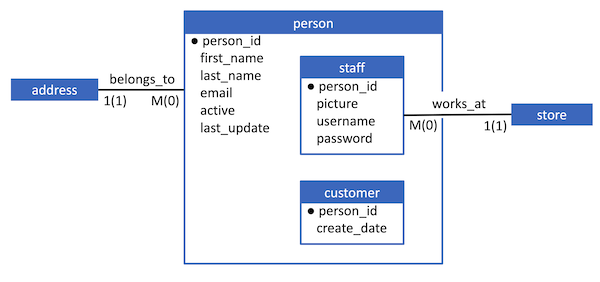
ALTER TABLE rental

DROP CONSTRAINT fk\_rental\_customer,

DROP CONSTRAINT fk\_rental\_staff;

ALTER TABLE store

DROP CONSTRAINT fk\_store\_staff;



-- Your CREATE TABLE statements go here

CREATE TABLE person (

person\_id SMALLINT UNSIGNED PRIMARY KEY,

first\_name VARCHAR(20) NOT NULL,

last\_name VARCHAR(20) NOT NULL,

email VARCHAR(20) NOT NULL,

address\_id SMALLINT UNSIGNED NOT NULL,

active VARCHAR(20) NOT NULL,

last\_update TIMESTAMP NOT NULL,

FOREIGN KEY (address\_id) REFERENCES address(address\_id)ON UPDATE RESTRICT ON DELETE RESTRICT

);

CREATE TABLE staff (

person\_id SMALLINT UNSIGNED PRIMARY KEY,

picture BLOB NOT NULL,

username VARCHAR(20) NOT NULL,

password VARCHAR(20) NOT NULL,

store\_id TINYINT UNSIGNED NOT NULL,

FOREIGN KEY (store\_id) REFERENCES store(store\_id) ON UPDATE RESTRICT ON DELETE RESTRICT,

FOREIGN KEY (person\_id) REFERENCES person(person\_id) ON UPDATE CASCADE ON DELETE CASCADE

);

CREATE TABLE customer (

person\_id SMALLINT UNSIGNED PRIMARY KEY,

create\_date TIMESTAMP NOT NULL,

FOREIGN KEY (person\_id) REFERENCES person(person\_id) ON UPDATE CASCADE ON DELETE CASCADE

);

# 5.8 LAB - Create index and explain (Sakila)

**Refer to the film table of the Sakila database. This lab loads film with 100 rows from Sakila. Consequently, SELECT \* FROM film; generates too many characters to display in the zyLab environment. However, statements with less output, such as SELECT title FROM film;, execute successfully.**

**This lab illustrates the use of indexes and EXPLAIN to optimize query performance. Refer to** [**EXPLAIN documentation**](https://dev.mysql.com/doc/refman/8.0/en/explain-output.html) **for information about EXPLAIN result columns.**

**Write and run seven SQL statements:**

1. **Explain the query SELECT \* FROM film WHERE title = 'ALONE TRIP';.  
   *In the EXPLAIN result, column key is null, indicating no index is available for the query. Column rows is 100, indicating all rows are read. The query executes a table scan and is slow.***
2. **Create an index idx\_title on the title column.**
3. **Explain the query of step 1 again.  
   *In the EXPLAIN result, column key has value idx\_title, indicating the query uses the index on title. Column rows is 1, indicating only one table row is read. The query is fast.***
4. **Explain the query SELECT \* FROM film WHERE title > 'ALONE TRIP';.  
   *In the EXPLAIN result, column key is null, indicating the query does not use the idx\_title index. Column rows is 100, indicating all rows are read. Since the query has > in the WHERE clause rather than =, the query executes a table scan and is slow.***
5. **Explain the query SELECT rating, count(\*) FROM film GROUP BY rating;  
   *In the EXPLAIN result, column key is null, indicating no index is available for the query. Column rows is 100, indicating all rows are read. The query executes a table scan and is slow.***
6. **Create an index idx\_rating on the rating column.**
7. **Explain the query of step 5 again.  
   *In the EXPLAIN result, column key has value idx\_rating, indicating the query reads rating values from the index. The query uses an index scan, which is faster than a table scan (step 5).***

**For submit-mode testing, all seven statements must appear in Main.sql in the correct order.**

**NOTE: In submit-mode tests that generate multiple result tables, the results are merged. Although the tests run correctly, the results appear in one table.**

-- Your SQL statements go here

EXPLAIN SELECT \* FROM film

WHERE title = 'ALONE TRIP';

CREATE INDEX idx\_title on film(title);

--

EXPLAIN SELECT \* FROM film

WHERE title = 'ALONE TRIP';

EXPLAIN SELECT \* FROM film

WHERE title > 'ALONE TRIP';

EXPLAIN SELECT rating, Count(\*) FROM film

GROUP BY rating;

CREATE INDEX idx\_rating on film(rating);

EXPLAIN SELECT rating, Count(\*) FROM film

GROUP BY rating;

--

# 5.9 LAB - Query execution plans (Sakila)

This lab illustrates how minor changes in a query may have a significant impact on the execution plan.

# MySQL Workbench exercise

Refer to the film, actor, and film\_actor tables of the Sakila database. This exercise is based on the initial Sakila installation. If you have altered these tables or their data, your results may be different.

Do the following in MySQL Workbench:

1. Enter the following statements:

USE sakila;

SELECT last\_name, first\_name, ROUND(AVG(length), 0) AS average

FROM actor

INNER JOIN film\_actor ON film\_actor.actor\_id = actor.actor\_id

INNER JOIN film ON film\_actor.film\_id = film.film\_id

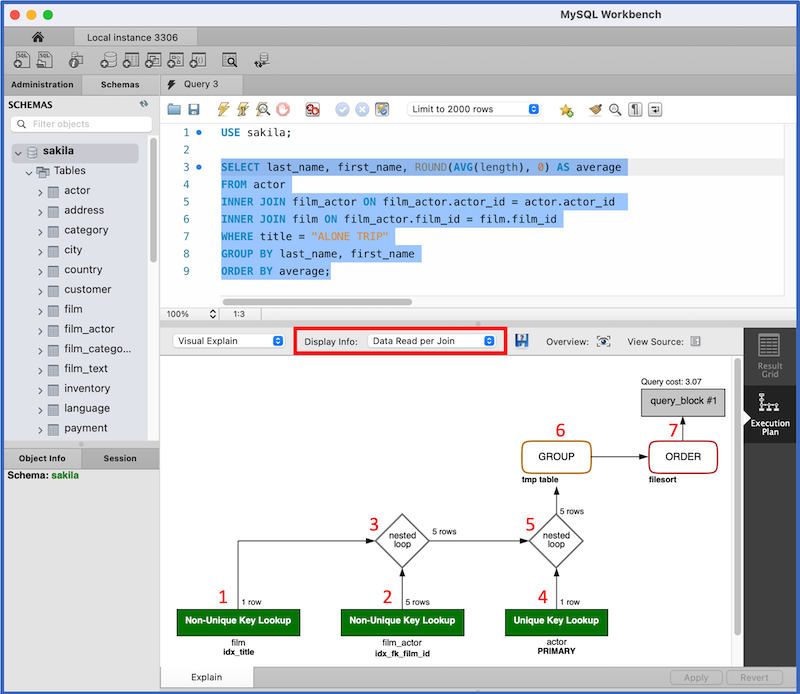
WHERE title = "ALONE TRIP"

GROUP BY last\_name, first\_name

ORDER BY average;

1. Highlight the SELECT query.
2. In the main menu, select Query > Explain Current Statement.
3. In the Display Info box, highlighted in red below, select Data Read per Join.

Workbench displays the following execution plan:



The execution plan depicts the result of EXPLAIN for the SELECT query. The execution plan has seven steps, corresponding to the red numbers on the screenshot:

1. Access a single film row using the idx\_title index on the title column.
2. Access matching film\_actor rows using the idx\_fk\_film\_id index on the film\_id foreign key.
3. Join the results using the nested loop algorithm.
4. Access actor rows via the index on the primary key.
5. Join actor rows with the prior join result using the nested loop algorithm.
6. Store the result in a temporary table and compute the aggregate function.
7. Sort and generate the result table.

Refer to [MySQL nested loop documentation](https://dev.mysql.com/doc/refman/8.0/en/nested-loop-joins.html) for an explanation of the nested loop algorithm.

Now, replace = in the WHERE clause with < and generate a new execution plan. Step 1 of the execution plan says Index Range Scan. The index scan accesses all films with titles preceding "ALONE TRIP", rather than a single film.

Finally, replace < in the WHERE clause with > and generate a third execution plan. Step 1 of the execution plan says Full Table Scan and accesses actor rather than film.

# zyLab coding

In the zyLab environment, write EXPLAIN statements for the three queries, in the order described above. Submit the EXPLAIN statements for testing.

The zyLab execution plans do not exactly match the Workbench execution plans, since this lab uses a subset of film, actor, and film\_actor rows from the Sakila database.

NOTE: In submit-mode tests that generate multiple result tables, the results are merged. Although the tests run correctly, the results appear in one table.

-- Your EXPLAIN statements go here

EXPLAIN SELECT title, release\_year, last\_name FROM film

JOIN film\_actor using(film\_id)

JOIN actor using(actor\_id)

WHERE title = "ALONE TRIP"

GROUP BY actor\_id, film\_id

ORDER BY release\_year DESC;

EXPLAIN SELECT last\_name, first\_name, ROUND(AVG(length), 0) AS average FROM actor

INNER JOIN film\_actor ON film\_actor.actor\_id = actor.actor\_id

INNER JOIN film ON film\_actor.film\_id = film.film\_id

WHERE title < "ALONE TRIP"

GROUP BY last\_name, first\_name

ORDER BY average;

EXPLAIN SELECT last\_name, first\_name, ROUND(AVG(length), 0) AS average FROM actor

INNER JOIN film\_actor ON film\_actor.actor\_id = actor.actor\_id

INNER JOIN film ON film\_actor.film\_id = film.film\_id

WHERE title > "ALONE TRIP"

GROUP BY last\_name, first\_name

ORDER BY average;

# 6.6 LAB - Rollback and savepoint (Sakila)

Refer to the actor table of the Sakila database. The table in this lab has the same columns and data types but fewer rows.

Start a transaction and:

1. Insert a new actor with values 999, 'NICOLE', 'STREEP', '2021-06-01 12:00:00'
2. Set a SAVEPOINT.
3. Delete the actor with first name 'CUBA'.
4. Select all actors.
5. Roll back to the savepoint.
6. Select all actors a second time.

The actor with first name 'CUBA' should appear in the second SELECT but not the first.

NOTE: In submit-mode tests that generate multiple result tables, the results are merged. Although the tests run correctly, the results appear in one table.

-- Your SQL statements go here

START TRANSACTION;

INSERT INTO actor VALUES(999, 'NICOLE', 'STREEP', '2021-06-01 12:00:00');

SAVEPOINT S1;

DELETE FROM actor WHERE first\_name = 'CUBA';

SELECT \* FROM actor;

ROLLBACK TO S1;

SELECT \* FROM actor;

# 9.8 LAB - Database programming with Python (SQLite)

Complete the Python program to create a Horse table, insert one row, and display the row. The main program calls four functions:

1. create\_connection() creates a connection to the database.
2. create\_table() creates the Horse table.
3. insert\_horse() inserts one row into Horse.
4. select\_all\_horses() outputs all Horse rows.

Complete all four functions. Function parameters are described in the template. Do not modify the main program.

The Horse table should have five columns, with the following names, data types, constraints, and values:

| Name | Data type | Constraints | Value |
| --- | --- | --- | --- |
| Id | integer | primary key, not null | 1 |
| Name | text |  | 'Babe' |
| Breed | text |  | 'Quarter horse' |
| Height | double |  | 15.3 |
| BirthDate | text |  | '2015-02-10' |

The program output should be:

All horses:

(1, 'Babe', 'Quarter Horse', 15.3, '2015-02-10')

This lab uses the SQLite database rather than MySQL. The Python API for SQLite is similar to MySQL Connector/Python. Consequently, the API is as described in the text, with a few exceptions:

* Use the import library provided in the program template.
* Create a connection object with the function sqlite3.connect(":memory:").
* Use the character ? instead of %s as a placeholder for query parameters.
* Use data type text instead of char and varchar.

SQLite reference information can be found at [SQLite Python Tutorial](https://www.sqlitetutorial.net/sqlite-python/), but is not necessary to complete this lab.

import sqlite3

from sqlite3 import Error

# Creates connection to sqlite in-memory database

def create\_connection():

"""

Create a connection to in-memory database

:return: Connection object

"""

# YOUR CODE HERE

# Use sqlite3.connect(":memory:") to create connection object

try:

conn = sqlite3.connect(":memory:")

return conn

except Error as err:

print(err)

return None

# Creates Horse table

def create\_table(conn):

"""

Create Horse table

:param conn: Connection object

:return: Nothing

"""

# YOUR CODE HERE

sql\_create\_horse\_table = """

CREATE TABLE Horse (

Id integer PRIMARY KEY NOT NULL,

Name text,

Breed text,

Height double,

BirthDate text

); """

try:

c = conn.cursor()

c.execute(sql\_create\_horse\_table)

except Error as err:

print(err)

# Inserts row to Horse table given data tuple

def insert\_horse(conn, data):

"""

Create a new row in Horse table

:param conn: Connection object

:param data: tuple of values for new row

:return: Nothing

"""

# YOUR CODE HERE

# Use the ? character as placeholder for SQLite query parameters

ins\_horse = "INSERT INTO Horse VALUES(?, ?, ?, ?, ?)"

curr\_conn = conn.cursor()

curr\_conn.execute(ins\_horse, data)

# Selects and prints all rows of Horse table

def select\_all\_horses(conn):

"""

Query all rows in the Horse table

:param conn: the Connection object

:return: Nothing

"""

# YOUR CODE HERE

curr\_conn = conn.cursor()

curr\_conn.execute("SELECT \* FROM Horse")

rows = curr\_conn.fetchall()

for i in rows:

print(i)

# DO NOT MODIFY main

if \_\_name\_\_ == '\_\_main\_\_':

# Create connection to sqlite in-memroy database

conn = create\_connection()

if conn is None:

print("Error! cannot create the database connection.")

# Create Horse table

create\_table(conn)

# Insert row to Horse table

horse\_data = (1, "Babe", "Quarter Horse", 15.3, "2015-02-10")

insert\_horse(conn, horse\_data)

# Select and print all Horse table rows

print("All horses:")

select\_all\_horses(conn)

# 9.9 LAB - Database programming with Java (SQLite)

Complete the Java program to create a Horse table, insert one row, and display the row. The main program calls four methods:

1. createConnection() creates a connection to the database.
2. createTable() creates the Horse table.
3. insertHorse() inserts one row into Horse.
4. selectAllHorses() outputs all Horse rows.

Complete all four methods. Method parameters are described in the template. Do not modify the main program.

The Horse table should have five columns, with the following names, data types, constraints, and values:

| Name | Data type | Constraints | Value |
| --- | --- | --- | --- |
| Id | integer | primary key, not null | 1 |
| Name | text |  | 'Babe' |
| Breed | text |  | 'Quarter horse' |
| Height | double |  | 15.3 |
| BirthDate | text |  | '2015-02-10' |

The program output should be:

All horses:

(1, 'Babe', 'Quarter Horse', 15.3, '2015-02-10')

This lab uses the SQLite database rather than MySQL. Both SQLite and MySQL Connector/J implement the JDBC API. Consequently, the API is as described in the text, with a few exceptions:

* Use the connection string "jdbc:sqlite::in-memory" to connect to an in-memory database.
* Use the text data type instead of char and varchar.

SQLite reference information can be found at [SQLite Java Tutorial](https://www.sqlitetutorial.net/sqlite-java/), but is not necessary to complete this lab.

import java.sql.\*;

public class LabProgram {

// Create a connection to a sqlite in-memory database

// Returns Connection object

public static Connection createConnection() {

// YOUR CODE HERE

// Use connection string "jdbc:sqlite::memory:"

Connection conn = null;

try {

conn = DriverManager.getConnection("jdbc:sqlite::memory");

}

catch (SQLException e) {

e.printStackTrace();

}

return conn;

}

// Create Horse table

// Parameter conn is database connection created in createConnection()

public static void createTable(Connection conn) {

// YOUR CODE HERE

Statement horseSt = null;

String createTable =

"CREATE TABLE Horse(Id INT NOT NULL, Name TEXT NULL, Breed TEXT NULL, Height DOUBLE NULL, BirthDate TEXT NULL, PRIMARY KEY ( Id ));";

try {

horseSt = conn.createStatement();

horseSt.executeUpdate(createTable);

}

catch (SQLException e) {

e.printStackTrace();

}

}

// Insert row into Horse table using a parameterized query

// Parameter conn is database connection created in createConnection()

// Parameters id, name, breed, height, and birthDate contain values to be inserted

public static void insertHorse(Connection conn, int id, String name, String breed, double height, String birthDate) {

// YOUR CODE HERE

try {

PreparedStatement horseStatement;

horseStatement = conn.prepareStatement("INSERT INTO horse VALUES (?, ?, ? ,?,?) ");

int i = 1;

horseStatement.setInt(i++, id);

horseStatement.setString(i++, name);

horseStatement.setString(i++, breed);

horseStatement.setDouble(i++, height);

horseStatement.setString(i++, birthDate);

int x = horseStatement.executeUpdate();

}

catch (SQLException e) {

e.printStackTrace();

}

}

// Select and print all rows of Horse table

// Parameter conn is database connection created in createConnection()

public static void selectAllHorses(Connection conn) {

// YOUR CODE HERE

PreparedStatement horseStatement;

try {

horseStatement = conn.prepareStatement("SELECT \* FROM horse");

ResultSet resultSet = horseStatement.executeQuery();

System.out.println("All horses:");

if(resultSet != null){

while(resultSet.next()) {

System.out.print("(");

System.out.print(resultSet.getInt("id")+", ");

System.out.print("'"+resultSet.getString("Name") + "', ");

System.out.print("'"+resultSet.getString("Breed") + "', ");

System.out.print(resultSet.getDouble("Height")+ ", ");

System.out.print("'"+resultSet.getString("BirthDate")+ "')");

}

}

else {

System.out.println("Horse table does not exist");

}

}

catch (SQLException e) {

e.printStackTrace();

}

}

// DO NOT MODIFY main

public static void main(String[] args) {

// Create connection to sqlite in-memory database

Connection conn = createConnection();

// Create Horse table

createTable(conn);

// Insert row into Horse table

insertHorse(conn, 1, "Babe", "Quarter Horse", 15.3, "2015-02-10");

// Select and print all Horse table rows

selectAllHorses(conn);

}

}