**Assignment 3**

By Harsh Sharma

**Part I. Database APIs**

Question 1:

Answer: I’m using Python and SQLite for this Assignment and hence have updated the JDBC file accordingly to work in my environment.

The 14 places that was needed to be updated in Java file is changed. Since I’m using Python not all 14 places need to be updated. I’ve made the changes and marked them as necessary, and file is shared as a script.

Screenshots of the Result:

1. There were more movies but I’m sharing a glimpse of the movies. The complete fetch result is available on the jupyter notebook.

A screenshot of a computer screen

Description automatically generated

(ii)

A screenshot of a computer program

Description automatically generated

**Part II. Physical Database Organization and Index**

Q1. (1)

**SQLite does not directly provide a system catalog table equivalent to USER\_USERS in Oracle.**

Since SQLite has different functionalities than Oracle, the SQL queries and the output will be a little different.

Query:

import sqlite3

# Connect to your SQLite database

conn = sqlite3.connect('hw3.db')

# Create a cursor object

cursor = conn.cursor()

# Execute the query to extract information

cursor.execute("SELECT name, type, rootpage, sql FROM sqlite\_master WHERE type='table';")

# Fetch all rows

rows = cursor.fetchall()

# Display the result

for row in rows:

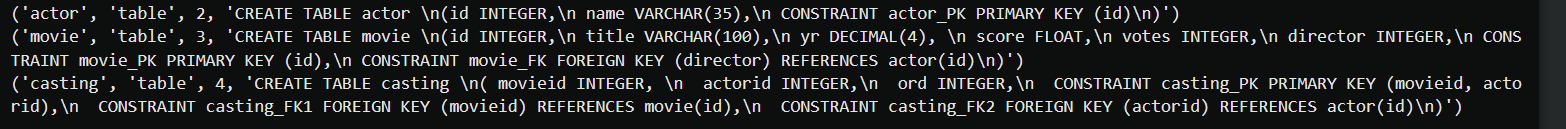
print(row)

# Close the cursor and connection

cursor.close()

conn.close()

Result:



Q2. (1)

For SQLite, three’s no equivalent of Oracle's USER\_TABLESPACES view. However, I used the sqlite\_master table to gather information about the tables created in the database, including the tablespace details if specified.

Query:

import sqlite3

# Connect to your SQLite database

conn = sqlite3.connect('hw3.db')

# Create a cursor object

cursor = conn.cursor()

# Execute the query to gather information about tables

cursor.execute("SELECT tbl\_name, sql FROM sqlite\_master WHERE type='table';")

# Fetch all rows

rows = cursor.fetchall()

# Display the result

for row in rows:

print("Table Name:", row[0])

print("Table Definition:", row[1])

print() # Add a newline for readability

# Close the cursor and connection

cursor.close()

conn.close()

Screenshot of result:

A screenshot of a computer program

Description automatically generated

Q3. (1)

In SQLite, there isn't a direct equivalent to Oracle's DBA\_DATA\_FILES view since SQLite doesn't manage tablespaces or data files in the same way as Oracle. However, I used gather information about the tables and their associated file sizes directly from the file system.

Query::

import os

# Function to get file size in bytes

def get\_file\_size(file\_path):

return os.path.getsize(file\_path)

# Function to convert bytes to Oracle blocks (assuming block size of 8192 bytes)

def bytes\_to\_blocks(file\_size):

return file\_size / 8192

# Directory containing the SQLite database file

db\_directory = 'path\_to\_directory\_containing\_database\_file'

# SQLite database file name

db\_file\_name = 'your\_database.db'

# Full path to the SQLite database file

db\_file\_path = os.path.join(db\_directory, db\_file\_name)

# Get information about the database file

file\_size\_bytes = get\_file\_size(db\_file\_path)

file\_size\_blocks = bytes\_to\_blocks(file\_size\_bytes)

# Display the information

print("Database File Information:")

print("File Name:", db\_file\_name)

print("File Size (Bytes):", file\_size\_bytes)

print("File Size (Oracle Blocks):", file\_size\_blocks)

Screenshot of Result:

A black screen with white text

Description automatically generated

Q4.

In SQLite, there isn't a direct equivalent to Oracle's ANALYZE TABLE statements. However, I used the ANALYZE command in SQLite to gather statistics that the query planner can use to optimize queries.

Query:

import sqlite3

# Connect to your SQLite database

conn = sqlite3.connect('hw3.db')

# Create a cursor object

cursor = conn.cursor()

# Execute ANALYZE command for each table

cursor.execute("ANALYZE movie;")

cursor.execute("ANALYZE actor;")

cursor.execute("ANALYZE casting;")

# Commit the changes

conn.commit()

# Close the cursor and connection

cursor.close()

conn.close()

Q5 (1)

In SQLite, there isn't a direct equivalent to Oracle's USER\_TABLES data dictionary view. I gathered some relevant information about the tables using SQLite's built-in functions and system tables.

Query:

import sqlite3

# Connect to your SQLite database

conn = sqlite3.connect('hw3.db')

# Create a cursor object

cursor = conn.cursor()

# Execute the query for the actor table

cursor.execute("""

SELECT

'actor' AS table\_name,

COUNT(\*) AS total\_rows,

avg(length(name)) AS avg\_row\_length

FROM

actor;

""")

# Fetch and print the result

row = cursor.fetchone()

print("Table Name:", row[0])

print("Total Rows:", row[1])

print("Average Row Length:", row[2])

print() # Add a newline for readability

# Execute the query for the casting table

cursor.execute("""

SELECT

'casting' AS table\_name,

COUNT(\*) AS total\_rows,

avg(length(actorid || ord)) AS avg\_row\_length

FROM

casting;

""")

# Fetch and print the result

row = cursor.fetchone()

print("Table Name:", row[0])

print("Total Rows:", row[1])

print("Average Row Length:", row[2])

print() # Add a newline for readability

# Execute the query for the movie table

cursor.execute("""

SELECT

'movie' AS table\_name,

COUNT(\*) AS total\_rows,

avg(length(title)) AS avg\_row\_length

FROM

movie;

""")

# Fetch and print the result

row = cursor.fetchone()

print("Table Name:", row[0])

print("Total Rows:", row[1])

print("Average Row Length:", row[2])

print() # Add a newline for readability

# Close the cursor and connection

cursor.close()

conn.close()

Screenshot Result:

A computer screen with numbers and letters

Description automatically generated

Q6.

In SQLite, you can query the sqlite\_master table to gather information about indexes on tables.

Query:

import sqlite3

# Connect to your SQLite database

conn = sqlite3.connect('hw3.db')

# Create a cursor object

cursor = conn.cursor()

# Define a function to extract index details

def get\_index\_details(table\_name):

cursor.execute(f"PRAGMA index\_list({table\_name});")

indexes = cursor.fetchall()

for index in indexes:

index\_name = index[1]

cursor.execute(f"PRAGMA index\_info({index\_name});")

index\_info = cursor.fetchall()

print("Index Name:", index\_name)

print("Table Name:", table\_name)

print("Index Type: (assuming B-tree)")

print("Uniqueness Status: UNIQUE" if index[2] else "Non-Unique")

print("Index Height: (not applicable in SQLite)")

print("Number of Leaf Blocks: (not directly accessible in SQLite)")

print("Average Leaf Blocks per Key: (not applicable in SQLite)")

print("Average Data Blocks per Key: (not applicable in SQLite)")

print() # Add a newline for readability

# Retrieve index details for actor table

get\_index\_details("actor")

# Retrieve index details for casting table

get\_index\_details("casting")

# Retrieve index details for movie table

get\_index\_details("movie")

# Close the cursor and connection

cursor.close()

conn.close()

Screenshot of result:

A screen shot of a computer program

Description automatically generated

Q7. Consider the following normalized relations for a sports league.

Answer:

**1. Identify all possible foreign keys:**

- In the given schema, foreign keys are typically columns that reference the primary key of another table. Based on the provided relations, the following foreign keys can be identified:

- `TEAM.TeamLeague` references `LEAGUE.LeagueID`.

- `TEAM.TeamManger` references `MANAGER.ManagerID`.

- `PLAYER.PlayerSpecialtyCode` references `SPECIALTY.SpecialtyCode`.

- There could potentially be more foreign keys depending on relationships not explicitly stated in the schema.

**2. What column(s) in a table do you recommend for indexes:**

- Indexes are typically recommended for columns that are frequently used in search conditions or join operations. Based on the operations described, the following columns could benefit from indexes:

- `TEAM.TeamLeague` and `TEAM.TeamManger` in the `TEAM` table if there are frequent queries joining `TEAM` with `LEAGUE` or `MANAGER`.

- `PLAYER.PlayerSpecialtyCode` in the `PLAYER` table if there are frequent queries involving player specialties.

- `LOCATION.CityName` and `LOCATION.CityCountry` in the `LOCATION` table if there are frequent queries involving city populations or teams/players by city.

- `SPECIALTY.SpecialtyCode` in the `SPECIALTY` table if there are frequent queries involving player specialties.

**3. Specify the types of indexes you recommend for them:**

- For the foreign key columns, primary key columns, and frequently queried columns, you can consider using B-tree indexes, which are the default and most commonly used type of index in relational databases like SQLite. B-tree indexes provide efficient lookup and range queries.

- Additionally, you may consider using unique indexes for columns that enforce uniqueness constraints.

**4. Explanation:**

- The recommendation for indexes is based on the expected queries and operations described in the scenario. For example:

- Adding new players would require efficient lookup by foreign key (e.g., `PlayerSpecialtyCode`).

- Reporting players by team and specialty would require efficient joins and filtering on the `TEAM` and `PLAYER` tables based on team and specialty columns.

- Reporting teams and their players by city would involve joining the `TEAM`, `PLAYER`, and `LOCATION` tables based on city-related columns.

- By identifying the frequent query patterns and considering the foreign keys and primary keys, we can make informed decisions about which columns to index and what types of indexes to use to optimize query performance.