CS-A1153 Databases, Tutorial session 4

Overview

- Python: Create tables and Import data from a sql & csv file
- Views: example of creating view
- Indices: calculating the cost of implementing indices
- Triggers: example
- Transactions: example of ACID property in SQLite database transaction.

NOTE:

The examples for VIEWS and TRIGGERS are based on the same Student table created with Python. You can try creating views and triggers using Python code. The example queries for VIEWS and TRIGGERS should work for SQLite. To make it work for Postgres, here are a few links for PostgreSQL syntax:

- CREATE TRIGGER
- Trigger functions

Python

Create tables and Import data from a sql & csv file

→ This part is covered inside the code examples that you can find on MyCourse this week. The examples cover both SQLite and PostgreSQL databases (code and `.sql` file for each database reside in their separate directories, `./postgresql` and `./sqlite`. As most groups chose to work with PostgreSQL, we can cover that tutorial. The tutorial for SQLite can be left as extra material

→ Once you had the file downloaded, setup Python virtual environment and run `pip install -r requirements.txt`.

→ For the PostgreSQL example, it assumes that you have installed PostgreSQL in your local machine. Thus, the host is "localhost". Make sure that you have the right password for user "postgres".

Views (works on Python example for sqlite)

Example: Consider the relation:

`Student (studID, name, DOB, program, credits)`

Create a view of all DataScience students (program = 'DS')

```
CREATE VIEW DSstudent AS
    SELECT studID, name, dob, program, credit
        FROM Student
        WHERE program = ' DS'; ---The space in front of value seems to be inherit from the dataset.
```

Once a view is created, you can query a view as well as modify it using:

```
SELECT * FROM DSstudent;

DROP VIEW DSstudent;
```

Indices

Example: Consider the relation:

```
Appointment (patient, doctor, date, description)
```

Suppose that we perform two kinds of queries on this relation

In Q_1, we look for patients who have had an appointment with a given doctor, assuming that the fraction of these queries of all database operations on this relation is p_1.

In Q_2, we look for doctors who have had an appointment with a given patient, assuming that the fraction of these queries of all database operations on this relation is p_2.

The fraction of insertions of all database operations on this relation is $1 - p_1 - p_2$.

The relation occupies 100 disk pages. Each doctor has 50 appointments on average, and each patient has 2 appointments on average.

If there is an index on doctor, Q_1 requires 1 disk access for the index and 50 for the tuples with the given doctor. If there is an index on patient, Q_2 requires 1 disk access for the index and 2 for the tuples with the given patient. In the case of insertion, each index must be updated. Two disk accesses are needed for each index (one to read the disk page into main memory and another to write the updated page back to the disk) and two disk accesses are needed for the page where the tuple is inserted.

Action	No Index	doctor Index	patient Index	both Index
Q1	100	51	100	51
Q2	100	100	3	3
I	2	4	4	6
Average	2 + 98p_1 + 98p2	4 + 47p_1 + 96p_2	4 + 96p_1 - p_2	6 + 45p_1 - 3p_2

Triggers (works on Python example for sqlite)

Example: Consider the relation:

```
Student (studID, name, DOB, program, credits)
```

When updating a student's number of credits, the new number must be larger than the old one. If the new number of credits is lower, keep the old one.

```
CREATE TRIGGER MinCredits
     AFTER UPDATE OF credit
        ON Student
  FOR EACH ROW
      WHEN (NEW.credit <= OLD.credit)
        BEGIN
            UPDATE Student
               SET credit = OLD.credit
             WHERE studID = NEW.studID;
        END;
```

For testing:

```
SELECT * FROM Student WHERE credit = 1;

UPDATE Student SET credit = credit - 1 WHERE credit = 1;

UPDATE Student SET credit = 0 WHERE studid = 2;

SELECT * FROM Student WHERE credit = 1;
```

```
    DROP TRIGGER MinCredits;
```

Transactions

Example: Consider the tables:

```
Product(prodID, stock)
Order(orderID, customer)
OrderContent(orderID,prodID, amount)
```

Atomicity

"All the commands inside a transaction should take place or none of them."

Consider the transaction consisting of the following commands

- Add a new tuple to OrderContent: ("001", "A", 3).
- Decrease the value of "stock" for product "A" by 3.

We should always complete the whole transaction or not at all. In practice this is enforced using a log.

Consistency

"The database must be consistent before and after the transaction."

If we had the commands from earlier as separate transactions, the database would be inconsistent after running the first one.

Isolation

"When two or more transactions occur simultaneously, the consistency should remain maintained."

Let us break the earlier transaction into smaller pieces as a pseudo code

```
read(stack)
if stack >= m
    stack = stack - m
```

Consider that we have two transactions of this type:

- T_1 decreases stack by 5 if stack is large enough
- T_2 decreases stack by 3 if stack is large enough

Let the stack value be initially 6. Consider that the commands appear in the following order

```
T1 reads the stack value: stack = 6
T2 reads the stack value: stack = 6
T1 writes the stack value: stack = 1
T2 writes the stack value: stack = 3
```

Now the transaction T_2 overrides the changes made by T_1 which leads to inconsistency.

Durability

Durability ensures, that the changes will remain in the database even in the case of failure.

Creating a transaction in SQL

Each independent statement is by default a transaction in a database. If we want to group statements, we want to define a transaction separately.

```
CREATE TABLE Product(
    prodID TEXT PRIMARY KEY,
    stack INTEGER);
```

```
CREATE TABLE OrderContent(
    orderID TEXT PRIMARY KEY,
    prodID TEXT REFERENCES Product(prodID),
    amount INTEGER
);

INSERT INTO Product VALUES ("A", 5);

BEGIN TRANSACTION;
INSERT INTO OrderContent VALUES ("001", "A", 3);
UPDATE Product
SET stack = stack - 3
WHERE prodID = "A";
COMMIT;

BEGIN TRANSACTION;
INSERT INTO OrderContent VALUES ("002", "A", 5);
UPDATE Product
SET stack = stack - 3
WHERE prodID = "A";
ROLLBACK;
```

By inspecting the tables at different phases, one can see in SQLiteStudio, that the changes from the second transaction have been rolled back.