IT UNIVERSITY OF COPENHAGEN

Introduction to Database Systems IDBS - Fall 2024

Lecture 4 - SQL Programming and Python

Functions
Triggers & Constraints
SQL and DBMS in Python
Transactions

Readings: PDBM 9.2, 14.1, 14.2.1 and 14.5

Omar Shahbaz Khan

Last Time in IDBS…

-- TODO -> DONE

- ✓ Division
- ✓ JOIN & NULL
  - ✓ Natural Joins, Cross Joins, Self-Joins
  - ✓ NULL
  - ✓ Outer Joins (LEFT, RIGHT, FULL)
- ✓ Set Operations
  - ✓ UNION, INTERSECT, EXCEPT
- ✓ Subqueries (Nested Queries)
  - ✓ =, IN, EXISTS, ALL, ANY
- ✓ Views: Queries as subroutines

Wake Up Task!

Bills DB

People (PID, pName, pGender, pHeight)
Accounts (AID, PID, aDate, aBalance, aOver)
AccountRecords (RID, AID, rDate, rType, rAmount, rBalance)
Bills (BID, PID, bDueDate, bAmount, bIsPaid)

- How many accounts have never been used before?
- How many accounts have a negative account balance and also have bills due?

This Time…

-- TODO

- Database Functions
- Database Triggers & Constraints
- Transactions
- SQL in Python

$$f(x)$$

Functions in SQL

- What is a function?

- Similar to other programming languages methods/functions

- How is it useful in SQL?

Create a new Person

- In the Sports database create a new person
  - Insert a row

- Can be cumbersome to write INSERT INTO … VALUES … if we want to insert many new rows

- How about a function?

```sql
INSERT INTO
People (name, gender, height)
VALUES ('Terry', 'M', 1.77);
```

Create a new Person

- In the Sports database create a new person
    - Insert a row
- Can be cumbersome to write INSERT INTO … VALUES … if we want to insert many new rows
- How about a function?

```
DROP FUNCTION IF EXISTS NewPerson;

CREATE FUNCTION NewPerson (
     IN pname VARCHAR(50),
     IN pgender CHAR(1),
     IN pheight FLOAT
)
RETURNS INTEGER
AS $$
BEGIN
    INSERT INTO
    Person (name, gender, height)
    VALUES (pname, pgender, pheight);
    RETURN lastval();
END;
$$ LANGUAGE plpgsql;
```

Function Parameters

Return Type

Insert Statement

Return the last value of the table’s SEQUENCE

Using Functions

- From an SQL Script

- Typically used from an ODBC (Java, Python, C#, …)

```
SELECT NewPerson('Terry', 'M', 1.77);


SELECT * FROM NewPerson('Terry', 'M', 1.77);


DO $$
    BEGIN
        PERFORM NewPerson('Terry', 'M', 1.77);
    END
$$;
```

Let’s make a function (I)

- Assume RecordLog table:

```
RecordLog (
    peopleID INT,
    competitionID INT,
    sportID INT,
    oldrecord FLOAT,
    newrecord FLOAT,
    seton DATE
);
```

- Function: BiggestRecordJump
  - Input: ID of sport
  - Output: The largest increase of that sport

```
RecordLog (
    peopleID INT,
    competitionID INT,
    sportID INT,
    oldrecord FLOAT,
    newrecord FLOAT,
    seton DATE
);
```

Let’s make a function (II)

- Assume RecordLog table:

```
RecordLog (
    peopleID INT,
    competitionID INT,
    sportID INT,
    oldrecord FLOAT,
    newrecord FLOAT,
    seton DATE
);
```

- Function: BiggestRecordJump
  - Input: ID of sport
  - Output: The largest increase of that sport

```
DROP FUNCTION IF EXISTS BiggestRecordJump();

CREATE FUNCTION BiggestRecordJump(
    IN sid INT
)
RETURNS FLOAT
AS $$
DECLARE r FLOAT;
BEGIN
    SELECT MAX(newrecord - oldrecord) INTO r
      FROM RecordLog
     WHERE sportID = sid
    RETURN r;
END;
$$ LANGUAGE plpgsql;

SELECT BiggestRecordJump(1);
```

Variable to store value from query

Are Functions Faster?

- May be faster than executing from a client
  - Why?

- Code may be pre-compiled and optimized
  - Do not need to invoke optimizer again
  - May occur with well written queries using plan caching

- The code runs at the server
  - The server may be more efficient
  - No need to move data

Pros & Cons

- Code shared across ALL applications
- May be used for access control
- May give performance benefits

- Very system-specific
- Code maintenance requires care
  - Versioning is difficult

IDENTITY Columns

- Run 04-sports-schema.sql
- Try inserting a new person
- Due to the ID column not being an IDENTITY column it will fail inserting the row
- Quickfix: Add parameter pid in the NewPerson Function

- Best to avoid this by using `GENERATED ALWAYS AS IDENTITY` when creating the table
- Fixed tables with ID columns correctly set to `GENERATED ALWAYS AS IDENTITY` (run 04-sports-schema-fixed.sql)
- In case you want to insert a row with a specific ID, you can do so by OVERRIDING SYSTEM VALUE. See example in the file.

-- TODO

✓ Database Functions

- Database Triggers & Constraints
- Transactions
- SQL in Python

Triggers in SQL

- Automatically executed function in response to certain events on a table (or view) in a database
    - Events are typically INSERT, UPDATE, DELETE
    - Triggers can execute BEFORE or AFTER the event

- Useful for maintaining the integrity of the information on the Database.
    - For example, in a company database when a new record (representing a new worker) is added to the employees table, new records could also be created in the tables of taxes, vacations and salaries.

- Logging historical data, for example to keep track of employees' previous salaries.

Triggers in PostgreSQL

- Multiple triggers per table per event
  - INSERT / UPDATE / DELETE
  - Run in alphabetical order
  - Per row OR Per statement – focus here on per row

- New data is in the NEW record
  - For INSERT / UPDATE
  - Same schema as modified relation
  - Can refer to NEW.ID, NEW.name, …

- Old data is in the OLD record
  - For UPDATE / DELETE
  - Same schema, refer to OLD.ID, OLD.name, …

- The variable TG_OP says which operation it is

Trigger: Checking Values

- Use triggers to check value semantics
- A result in the Sports DB must not be negative

```
CREATE FUNCTION CheckResult()
RETURNS TRIGGER
AS $$ BEGIN
    IF (NEW.result < 0.0) THEN
        RAISE EXCEPTION
        'CheckResult: Result must be a
         positive'
        USING ERRCODE = '45000';
    END IF;
    RETURN NEW;
END; $$ LANGUAGE plpgsql;
```

Create Trigger based on function

```
CREATE TRIGGER CheckResult
BEFORE INSERT OR UPDATE
ON Results
FOR EACH ROW EXECUTE PROCEDURE CheckResult();
```

Trigger: Ban Updates/Deletes

- Use triggers to not allow updating or deleting rows

```
CREATE FUNCTION BanChanges()
RETURNS TRIGGER
AS $$
BEGIN
    RAISE EXCEPTION
        'BanChanges: Cannot change
         results!'
        USING ERRCODE = '45000';
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER BanChanges
BEFORE UPDATE OR DELETE
ON Results
FOR EACH ROW EXECUTE PROCEDURE
BanChanges();
```

Trigger: Update Records

- Use triggers to update the values after an event
- If a result in Sports DB is a new record, the row for that sport in the sports table need to be updated

```
CREATE FUNCTION UpdateRecord()
RETURNS TRIGGER
AS $$ BEGIN
    IF NEW.result > (
        SELECT s.record
          FROM Sports s
         WHERE s.id = NEW.sportID
    ) THEN
        UPDATE Sports
        SET record = NEW.result
        WHERE s.id = NEW.sportID;
    END IF;
    RETURN NEW;
END; $$ LANGUAGE plpgsql;

CREATE TRIGGER UpdateRecord
AFTER INSERT OR UPDATE
ON Results
FOR EACH ROW EXECUTE PROCEDURE UpdateRecord();
```

Could also be RETURN NULL since result of AFTER trigger is ignored

Trigger: Log Changes

```
CREATE TABLE RecordLog (
    peopleID INT,
    competitionID INT,
    sportID INT,
    oldrecord FLOAT,
    newrecord FLOAT,
    seton DATE
    PRIMARY KEY
    (peopleID, competitionID, sportID)
    FOREIGN KEY
    (peopleID, competitionID, sportID)
    REFERENCES Results
    (peopleID, competitionID, sportID)
);
```

```
CREATE FUNCTION LogRecord()
RETURNS TRIGGER
DECLARE oldRecord FLOAT;
AS $$ BEGIN
    IF NEW.result > (
        SELECT s.record
          FROM Sports s
         WHERE s.id = NEW.sportID
    ) THEN
        SELECT s.record INTO oldRecord
          FROM Sports s
         WHERE s.id = NEW.sportID;
        INSERT INTO RecordLog
        VALUES (NEW.peopleID, NEW.competitionID,
                NEW.sportID, oldRecord,
                NEW.result);
    END IF;
    RETURN NEW;
END; $$ LANGUAGE plpgsql;

CREATE TRIGGER LogRecord
AFTER INSERT ON Results
FOR EACH ROW EXECUTE PROCEDURE LogRecord();
```

Questions

- Why not create the LogRecord on Sports?

- What about Trigger order?
    - LogRecord vs UpdateRecord
    - Order is alphabetical = good in this case
        - Try with different order
    - May be better to merge similar triggers

- What about BEFORE or AFTER?
        - LogRecord and UpdateRecord are AFTER
        - CheckResult and BanUpdates are BEFORE
    - BEFORE and AFTER what? When does each apply? Why?

Merged Trigger (BanUpdates + Check Updates)

```
CREATE FUNCTION MergedTrigger()
RETURNS TRIGGER
AS $$ BEGIN
    IF (TG_OP = 'DELETE' OR TG_OP = 'UPDATE')
    THEN
        RAISE EXCEPTION
        'MergedTrigger: Cannot change results!'
        USING ERRCODE = '45000';
    END IF;
    IF (NEW.result < 0.0) THEN
        RAISE EXCEPTION
        'CheckResult: Result must be a positive!'
        USING ERRCODE = '45000';
    END IF;
END; $$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER MergedTrigger
BEFORE INSERT OR UPDATE OR DELETE
ON Results
FOR EACH ROW EXECUTE PROCEDURE
MergedTrigger();

INSERT INTO Results
VALUES (1,1,3,-1.0);
DELETE FROM Results WHERE sportID = 3;
```

BEFORE vs AFTER

1. Are you only checking the newly inserted/updated entry? -> BEFORE (or AFTER)
   - Checking happens earlier with BEFORE, so less work
   - But if other triggers might modify the values, then prefer AFTER

2. Are you inserting a row to another table
   with a foreign key constraint to the NEW record? -> AFTER
   - Otherwise, the NEW record is NOT in the database, so your insertion will fail!

3. Are you modifying the NEW record? -> BEFORE
   - Otherwise, the record is already in the database and will not be changed

4. Are you doing both 2 and 3? -> BEFORE and AFTER
   - Two different triggers!

Are Triggers Faster?

- May be faster than executing the code from a client
  - Why?

- Code may be pre-compiled and optimized
  - Do not need to invoke optimizer again
  - But this may happen with well written queries using plan caching

- The code runs at the server
  - The server may be more efficient
  - No need to move data to client

Pros & Cons

- Code often runs faster
  - No context switch
  - Compiled code
  - No data transfers
- Useful for security
  - Wrap data and functionality
  - Same access from all clients
- Same code for all applications

- Code is “hidden”
  - Only visible via system tables
  - Easily forgotten
  - Versioning is hard!
  - Schema may be edited using a GUI = no-no!
- Generally not portable

Exercises on SQL Programming

- Exercise 4 are out: Views, functions, triggers
    - Use test script to verify your work
    - Can run parts in pgAdmin – best via command prompt
    - You can extend the script with your own tests!
    - Also has a very nice database cleaning script

```
# once
psql -q Bills < bills-schema.sql

#repeatedly
psql Bills < universal-cleanup.sql
psql Bills < your-solution.sql
psql Bills < test-script.sql > output.txt 2>&1
less output.txt (linux / mac)
cat output.txt (windows) or open in an editor
```

NOTE: Windows uses '-f' instead of '<' and requires the DB name at the end of the command

NOTE 2: You may also be required to add user, -U <username>

Exercises on SQL Programming (Catalog)

- All information about the database is stored in tables!
  - This is the catalog!

https://www.postgresql.org/docs/current/catalogs.html

- See also: System information functions

https://www.postgresql.org/docs/current/functions-info.html

```sql
SELECT *
  FROM information_schema.tables
 WHERE table_schema = 'public';
```

```sql
SELECT current_database();

SELECT current_user;

SELECT lastval();
```

-- TODO

✓ Database Functions

✓ Database Triggers & Constraints

- Transactions

- SQL in Python

II

Transfer 50 DKK from Bob to Alice

Step 1: Get balance of Bob

```
SELECT Balance INTO x
  FROM accounts
 WHERE AccountName = 'Bob'
```

Account Name	Balance
Bob	200
Alice	0

Account Name	Balance
Bob	150
Alice	0

Step 2: Reduce balance of Bob

```
UPDATE accounts
   SET Balance = x-50
 WHERE AccountName = 'Bob'
```

WHAT IF SOMETHING CRASHES?

Step 3: Get balance of Alice

```
SELECT Balance INTO y
  FROM accounts
 WHERE AccountName = 'Alice'
```

Account Name	Balance
Bob	150
Alice	0

Account Name	Balance
Bob	150
Alice	50

Step 4: Increase balance of Alice

```
UPDATE accounts
   SET Balance = y+50
 WHERE AccountName = 'Alice'
```

What is a Transaction?

- A group of related operations to the database
- Wish for "all or nothing" execution
- Wish for isolation from other transactions

Basic SQL Syntax:

```
BEGIN;
COMMIT;
ROLLBACK;
```

Savepoints:

```
SAVEPOINT <name>;
ROLLBACK TO SAVEPOINT <name>;
```

Transactions in PostgreSQL

- By default, every statement is a transaction
  - To override this behaviour:
  - BEGIN; … COMMIT; / ROLLBACK;
  - Some (DDL) statements implicitly COMMIT transactions

- Calling a function starts a transaction
  - Can assume that the function has transactional properties!
  - Errors abort the transaction, erase all previous operations!

- Errors inside functions:
  - Cannot simply say: ROLLBACK
  - Need to raise and handle exceptions!
  - See example in exercise code

Transactions and Testing

- Transactions are useful in test scripts
  - Multiple examples in Exercise 4

- Simple pattern to test changing the database
  - … without actually changing the database

```
BEGIN
-- Make changes
-- Run test queries
ROLLBACK
```

-- TODO

✓ Database Functions

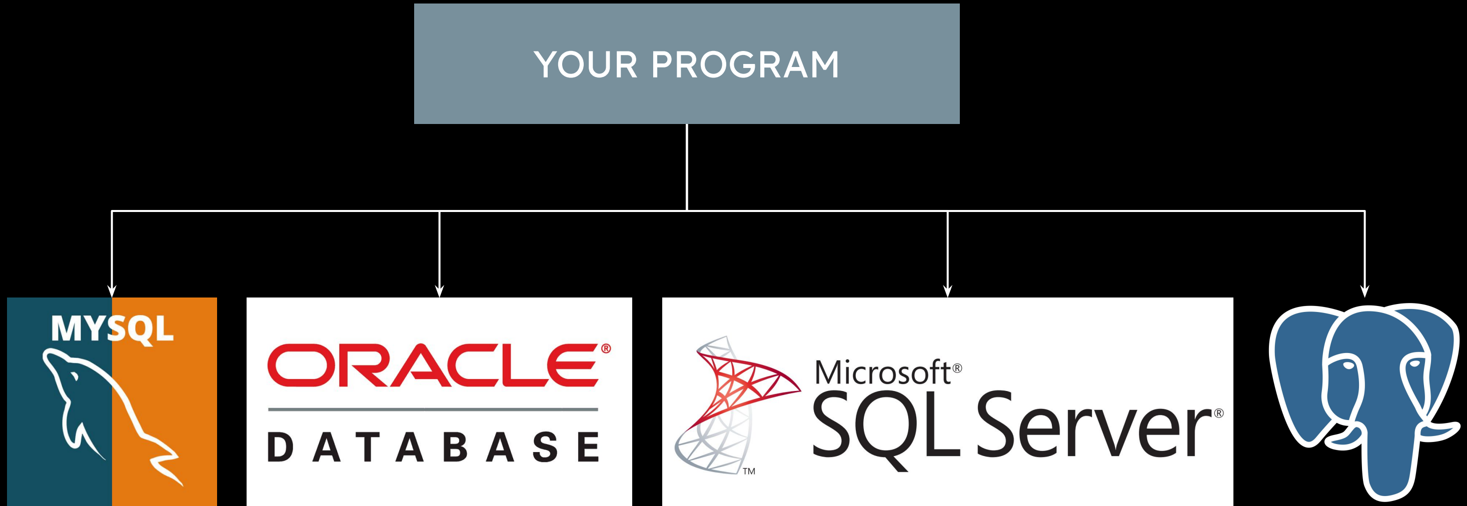
✓ Database Triggers & Constraints

✓ Transactions

- SQL in Python

Problem: Vendor Lock-in

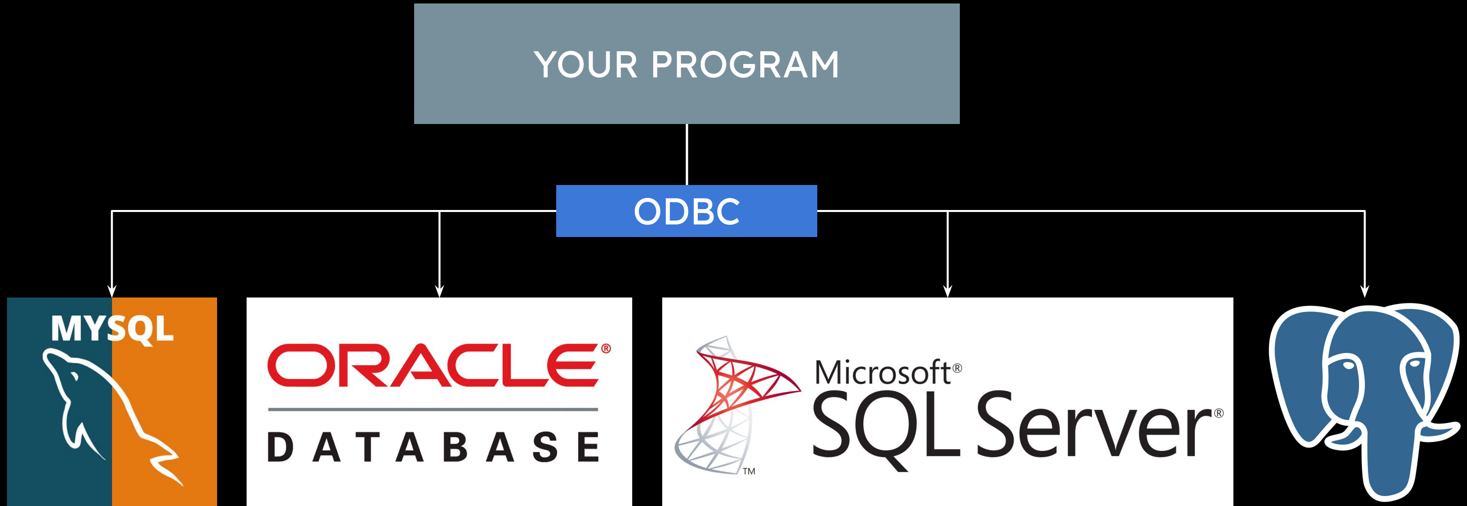
- They’re all compatible with standard SQL…
- …and mutually incompatible due to vendor-specific functionality!



YOUR PROGRAM

Open Database Connectivity (ODBC)

- In-between translation layer
- Vendors not so happy as lock-in is profitable!



Python and PostgreSQL

- pyodbc is a package that implements ODBC and can connect to any databases

- The psycopg package is a database adapter specifically for PostgreSQL
  - Reliable, documented, references
  - To install: python -m pip install psycopg or pip install psycopg
    - import psycopg

- A simple database API - Similar to ODBC
  - connect(): Establish connection to specified database
  - cursor(): Object to manage context of an SQL operation
  - execute(): Execute SQL queries
  - fetchone(), fetchmany(), fetchall(): Get results from queries
  - Much more, but these are the most used functions

Connecting to a Database

```
import psycopg as pg
from psycopg.rows import dict_row

conn_string = "host=localhost dbname=Sports user=postgres password=******"

# Connection option one
conn = pg.connect(conn_string, autocommit=True, row_factory=dict_row)
# Execute Queries using Cursors
conn.close()


# Connection option two
with pg.connect(conn_string, autocommit=True, row_factory=dict_row) as conn:
    # Execute Queries using Cursors
# Closes connection
```

Optional: Rows are returned as dict, better readability

Tips

```
try:
    # Connection option one
    conn = pg.connect(conn_string, autocommit=True, row_factory=dict_row)
    # Execute Queries using Cursors
except Exception as error:
    print(error)
finally:
    conn.close()
```

This block always run

```
try:
    # Connection option two
    with pg.connect(conn_string, autocommit=True, row_factory=dict_row) as conn:
        # Execute Queries using Cursors
    # Closes connection
except Exception as error:
    print(error)
```

Querying the DB

```
# Get all athletes
athletes: list[dict] = conn.execute("SELECT * FROM Athletes").fetchall()
```

Creates a cursor object and executes the query

```
# Get five athletes
athletes: list[dict] = conn.execute("SELECT * FROM Athletes").fetchmany(5)
```

What if I want more items later from the same query?

```
# Get all athletes
athletes: list[dict] = conn.execute("SELECT * FROM Athletes").fetchone()
```

Cursors

```
# Get query cursor
athletes_cursor = conn.execute("SELECT * FROM Athletes")

# Get five athletes
athletes: list[dict] = athletes_cursor.fetchmany(5)


# Get sixth athlete
athlete_six = athletes_cursor.fetchone()


# Get the next 10 athletes
athlete_next_ten = athletes_cursor.fetchmany(10)
```

Cursors

```
# Get query cursor
athletes_cursor = conn.execute("SELECT * FROM Athletes")

# Get five athletes
athletes: list[dict] = athletes_cursor.fetchmany(5)

sports_cursor = conn.execute("SELECT * FROM Sports")

# Get sixth athlete
athlete_six = athletes_cursor.fetchone()

sports: list[dict] = sports_cursor.fetchall()

# Get the next 10 athletes
athlete_next_ten = athletes_cursor.fetchmany(10)

athletes_cursor.close()
sports_cursor.close()
```

Advantage: Multiple cursors active at the same time, which can fetch or execute queries

Remember to close the cursors (context block, finally block)

Creating a Table

```
with pg.connect(conn_string, autocommit=True, row_factory=dict_row) as conn:
    conn.execute("DROP TABLE IF EXISTS Quotes")
    conn.execute(
        """
        CREATE TABLE Quotes (
            x INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
            y VARCHAR(250) NOT NULL,
        )
        """
    )
```

Better than SERIAL and follows SQL Standard.
Still a sequence under the hood.
PostgreSQL Identity Columns

INSERT

```
conn = pg.connect(conn_string, autocommit=True, row_factory=dict_row)
with conn:
    conn.execute("INSERT INTO Quotes (y) VALUES ('I am Vengeance')")

nope = conn.execute("SELECT 42 as universe").fetchone()
```

Won’t run since the connection is closed

INSERT

```
conn = pg.connect(conn_string, autocommit=True, row_factory=dict_row)
with conn:
    conn.execute("INSERT INTO Quotes (y) VALUES ('I am Vengeance')")

nope = conn.execute("SELECT 42 as universe").fetchone()
```

Won’t run since the connection is closed

```
with conn.cursor() as cur:
    cur.execute("INSERT INTO Quotes (y) VALUES ('I am the Night')")

yes = conn.execute("SELECT 42 as universe").fetchone()
```

Will run since only the cursor closed

```
conn.close()
```

Querying with variables

```
in_text = 'I am Batman'
sel_text = "'%Batman%'"


with pg.connect(conn_string, autocommit=True, row_factory=dict_row) as conn:
    conn.execute("INSERT INTO Quotes (y) VALUES (%s)", [in_text])
    conn.execute("SELECT * FROM Quotes WHERE y LIKE %s" % sel_text)
```

Question: Is this good or bad?

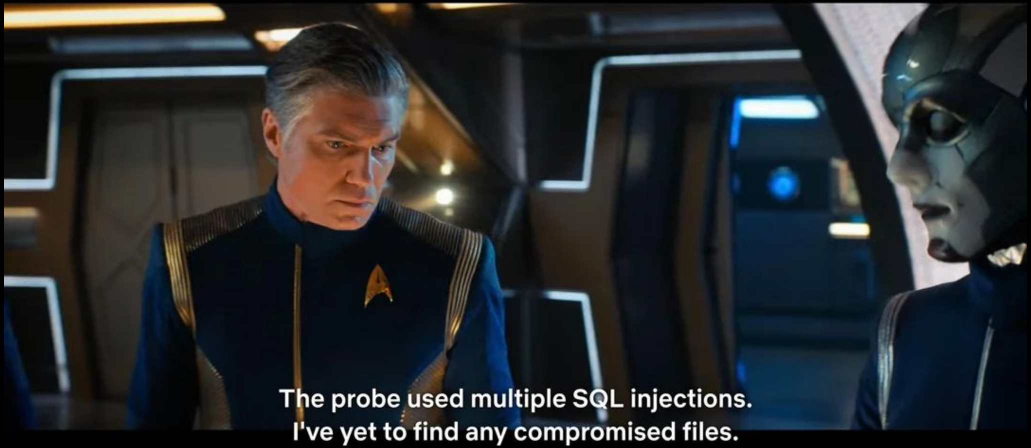
SELECT Query is Prone to SQL Injections!

SQL Injection Example

```
in_text = 'I am Batman'
sel_text = "'%Batman%'; DELETE FROM Quotes WHERE 1=1; INSERT INTO Quotes (y)
VALUES 'Joker was here';--"


with pg.connect(conn_string, autocommit=True, row_factory=dict_row) as conn:
    conn.execute("INSERT INTO Quotes (y) VALUES (%s)", [in_text])
    conn.execute("SELECT * FROM Quotes WHERE y LIKE %s" % sel_text)
```

Even relevant in the future:



The probe used multiple SQL injections.
I've yet to find any compromised files.

Star Trek Discovery Season 2
Episode 8

Year 2255+

Prepared Statements

- Separate the query string and variables
- Prepared Statements
  - Pre-compiled SQL statement at DBMS
  - Client supplied data is treated as content of a parameter and not an SQL statement

```
PREPARE my_query AS
 SELECT *
   FROM Quotes
  WHERE y = "$1"

EXECUTE my_query ("Probe used SQL
Injection... it’s not very effective")
```

Prepared Statements in Python

```
in_text = 'I am Batman'
sel_text = "'%Batman%'; DELETE FROM Quotes WHERE 1=1; INSERT INTO Quotes (y)
VALUES 'Joker was here';--"


with pg.connect(conn_string, autocommit=True, row_factory=dict_row) as conn:
    conn.execute("INSERT INTO Quotes (y) VALUES (%s)", [text])
    conn.execute("SELECT * FROM Quotes WHERE y LIKE %s", [sel_text])
```

✓

Not safe against all cases!

- Some elements can’t be prepared
- Most prominent is the ORDER BY clause
- Check the input in python before using the variable

```
sort_key = "name"

if sort_key in columns:

    cur = conn.execute(
        f"""
        SELECT *
        FROM Athletes
        ORDER BY {sort_key}
        """)

    athletes = cur.fetchall()
```

Autocommit

- If not specified in `connect()` autocommit is OFF by default
- Using `with` connection context commits at the end of the block

```
with pg.connect(conn_string, row_factory=dict_row) as conn:
    conn.execute("INSERT INTO Quotes (y) VALUES ('I am the one who knocks')")
    # BEGIN;
    # INSERT...;
# COMMIT; (default) / ROLLBACK; (if error is raised)
```

- Good for a small number of queries
- But closes connection after, so this may not always be desired

Autocommit (Cursors)

```
conn = pg.connect(conn_string, row_factory=dict_row)
with conn.cursor() as cur:
    tests = cur.execute("SELECT * FROM Quotes")
    # BEGIN;
    # SELECT...;
# Not committed, manually call conn.commit() inside or after the block
conn.commit()
```

- Even a SELECT statement begins a transaction
    - Easy to forget, making it more dangerous to not set autocommit on

Transactions with Autocommit ON

```
conn = pg.connect(conn_string, autocommit=True, row_factory=dict_row)

conn.execute("INSERT INTO Quotes (y) VALUES ('I am inevitable')")
# BEGIN; SELECT...; COMMIT/ROLLBACK;
with conn.transaction():
    # BEGIN;
    with conn.cursor() as cur:
        cur.execute("INSERT INTO Quotes (y) VALUES ('I am Groot')")
        cur.execute("INSERT INTO Quotes (y) VALUES ('I am Iron Man!')")
# COMMIT; (default) / ROLLBACK; (if error is raised)
```

-- TODO -> DONE

✓ Database Functions

✓ Database Triggers & Constraints

✓ Transactions

✓ SQL in Python

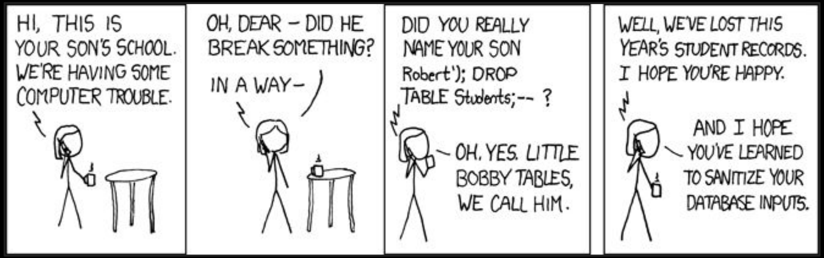
Takeaways

Functions
Set of database operations, performance benefits

Triggers
Execute a function in response to a database event

ODBC (Python psycopg, Java JDBC)
API for database operations

Transactions
Atomicity, Consistency, Isolation, Durability


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

Next Time in IDBS…

IT UNIVERSITY OF COPENHAGEN

Introduction to Database Systems IDBS - Fall 2024

Lecture 5 - Designing Databases

ER Diagrams

Translation to SQL DDL

Readings: PDBM 3.0-3.3, 6.3-6.4

Omar Shahbaz Khan

Extra bits

Roles in Databases

- Multiple users interacting with the database
    - Should everyone use the postgres role?
    - Roles are the same as Users

- Access Control
    - We can define roles with different privileges
    - Avoid accidents or fatal errors (database related and business related)

Roles in SQL

- CREATE

```
CREATE ROLE <role> WITH LOGIN ENCRYPTED PASSWORD <password>;
CREATE ROLE viewer WITH LOGIN ENCRYPTED PASSWORD '1337';
```

- GRANT

```
GRANT <privilege> ON <schema_elements> TO <role>;
GRANT SELECT ON Sports, Results, Competitions, Athletes,
Gender TO viewer;
```

- REVOKE

```
REVOKE SELECT ON Sports, Results, Competitions, Athletes,
Gender FROM viewer;
```

- DROP OWNED BY
- DROP

```
DROP OWNED BY viewer;
DROP ROLE viewer;
```

Using Roles

- To use roles in Python and in general we use the SQL statement SET ROLE

- SET ROLE <role>
- SET LOCAL ROLE <role>

- Best to execute in a transaction so we do not accidentally allow others to execute as the set role

- Check the add functions in the exercise

Roles and Functions I

- Creating a function in PostgreSQL grants execute privileges on public
  - Many roles could have access to public!
  - Solution: Revoke access from public and then grant access to specific roles
  - REVOKE ALL ON FUNCTION <function> FROM public;
  - GRANT EXECUTE ON FUNCTION <function> TO <role>;

Roles and Functions II

- By default a function is executed with the privileges of the calling role
    - A function accessing a schema element which the calling role does not have privileges to, an InsufficientPrivileges error is raised

- Resolution: SECURITY DEFINER
    - Add this after LANGUAGE plpgsql
    - The function is run with the privileges of the user that created it
    - Be careful!

A C I D

Transaction Properties

Recovery

ATOMICITY

A Transaction is "one operation"

Constraints, Triggers

CONSISTENCY

Each Transaction moves the DB from one consistent state to another

Concurrency Control

ISOLATION

Each Transaction is alone in the world

Recovery

DURABILITY

Persistence of successful transactions even through system failure

Transaction Implementation Methods (Lecture 10)

- Consistency
  - PRIMARY and FOREIGN KEY = limited DBMS support

- Isolation
  - Historically locking = (strict/rigorous) two-phase locking
  - Recently multi-version concurrency control

- Atomicity / Durability
  - Logging all changes to disk
  - Write Ahead Logging protocol (WAL)