Software Technologies for Data Science

Lecture 20

SQL - API

Ken Cameron

Content

- Transactions
- API
  - Sqlite3
  - MySQL
- Locks

Transactions

- I have previously said that queries have an immediate effect on the database.
- It is possible to prevent this using transactions.
- Transactions provide a means of grouping queries together.

ACID

- Atomicity
    - Ensure either all the queries in the transaction are successful or none are.
- Consistency
    - Ensure the database changes state properly after a successful transaction.
- Isolation
    - Ensure transactions operate independently of each other.
- Durability
    - Ensure successful transactions persist, even if there is a system failure.

Example

- If we're a bank and we're transferring money from one account to another.

  - We want the transfer to happen successfully or not.

    - We don't want to remove the money from one account and not add it to the other.

    - Or worse (for us), add it to the destination but not remove it from the source account.

MySQL Transactions

- By default MySQL treats each query as an independent transaction.
  - It is committed to the database if successful.

- You can disable the behaviour in a session using:

```
SET autocommit = 0;
```

MySQL Transactions

- Alternatively it can be done explicitly using:

```
START TRANSACTION;
```

- All queries after this will be part of the transaction until either

A `COMMIT` or a `ROLLBACK` query is issued.

MySQL Transactions

- `COMMIT`
  - The consequences of queries are committed to the database.
  - The changes are permanent.
- `ROLLBACK`
  - The queries are not committed to the database.

    It is as if all the queries between `START TRANSACTION` and `ROLLBACK` were not executed.

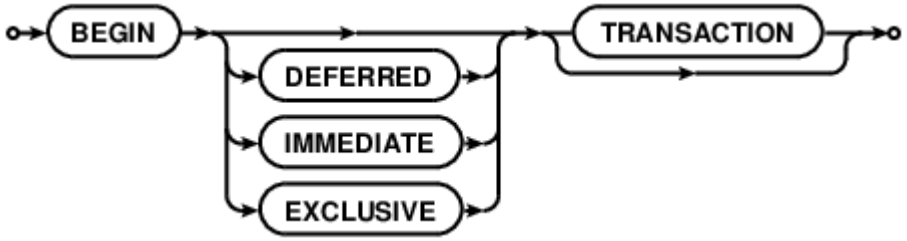
MySQL Transaction

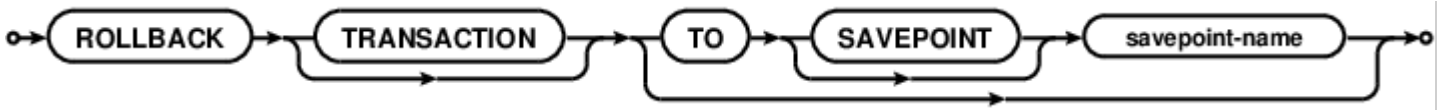
- This is only the basic transaction behaviour.
- We can add modifiers to `START TRANSACTION` to for example,
  - Chain transactions together,
  - Indicate if tables are to be read or written.
- Transactions only work properly if the underlying table database structures support transactions. Not all do.

Sqlite3 Transaction

- Use:

```
BEGIN TRANSACTION;
```





API

- Up until now I've looked at SQL queries in isolation.
  - They can be issued directly to a database at a command line.
  - But you've seen their use within software in Lab 5.
- We can access an SQL server via an API from within most programming languages.
- Today we'll examine accessing Sqlite3 and MySQL from Python in more detail.

Sqlite3 – connect()


```
import sqlite3

# Create a database in RAM
db = sqlite3.connect(':memory:')

# Create/open a file called mydb.
db = sqlite3.connect('data/mydb')
```

Sqlite3 – cursor and commit()

```
# Get a cursor object
cursor = db.cursor()

#execute the query
cursor.execute('''CREATE TABLE users(id INTEGER
PRIMARY KEY, name TEXT, email TEXT)''')

#commit it to the database
db.commit()
```

Sqlite3 – cursor and commit()

```
# Get a cursor object
cursor = db.cursor()

#execute the query
cursor.execute('''CREATE TABLE users(id INTEGER
PRIMARY KEY, name TEXT, email TEXT)''')

#commit it to the database
db.commit()
```

Sqlite3 – cursor and commit()

```
# Get a cursor object
cursor = db.cursor()

#execute the query
cursor.execute('''CREATE TABLE users(id INTEGER
PRIMARY KEY, name TEXT, email TEXT)''')

#commit it to the database
db.commit()
```

Sqlite3 – rollback()

```
# Get a cursor object
cursor = db.cursor()

#execute the query
cursor.execute('''CREATE TABLE users(id INTEGER
PRIMARY KEY, name TEXT, email TEXT)''')

#rollback the execution
db.rollback()
```

Sqlite3 – fetchone()

```
# obtains some rows from a table
cursor.execute('''SELECT name,email FROM users''')

#retrieve the first row
user1 = cursor.fetchone()

#Print the first column retrieved(user's name)
print(user1[0])
```

Sqlite3 – fetchall()

```
# obtains some rows from a table
cursor.execute('''SELECT name, email FROM users''')

#retrieve all the rows
all_rows = cursor.fetchall()

for row in all_rows:
    print('{0} : {1}'.format(row[0], row[1]))
```

Sqlite3 - Exceptions

```
try:

    db = sqlite3.connect('data/mydb’)
    cursor = db.cursor()
    cursor.execute('''CREATE TABLE IF NOT EXISTS users(id INTEGER PRIMARY KEY, name TEXT,
                      email TEXT unique, password TEXT)'‘’)
    db.commit()

except Exception as e:

    db.rollback()
    raise e

finally:

    # Close the db connection
    db.close()
```

Sqlite3 - Exceptions

```
try:

    db = sqlite3.connect('data/mydb’)
    cursor = db.cursor()
    cursor.execute('''CREATE TABLE IF NOT EXISTS users(id INTEGER PRIMARY KEY, name TEXT,
                      email TEXT unique, password TEXT)'‘’)
    db.commit()

except Exception as e:

    db.rollback()
    raise e

finally:

    # Close the db connection
    db.close()
```

Sqlite3 - Exceptions

```
try:

    db = sqlite3.connect('data/mydb’)
    cursor = db.cursor()
    cursor.execute('''CREATE TABLE IF NOT EXISTS
                      users(id INTEGER PRIMARY KEY, name TEXT,
                      email TEXT unique, password TEXT)'‘’)
    db.commit()

except Exception as e:

    db.rollback()
    db.close()
```

MySQL - API


```
import _mysql

db = _mysql.connect(”hostname”,
                    ”username”, ”password”,
                    ”database”)

db.query('''CREATE TABLE IF NOT EXISTS
            users(id INTEGER PRIMARY KEY,
                  name VARCHAR(255), email VARCHAR(255))''')
```

In python, no return value from query(), but exceptions possible.

MySQL - results

```
res = db.store_result() #held in client

res = db.use_result() #held in server

res.fetchrow(max_rows)

res.fetchrow()
```

fetchrow() will never return more than max_rows.
Without a parameter the number is 1.

MySQL – cursor version

```
import MySQLdb

db = MySQLdb.connect(…)

c = db.cursor()

c.execute(…) # etc
```

The cursor based interface is also available for MySQL

Multiple client issues

- In full SQL servers, multiple clients can connect simultaneously.
- To ensure database integrity
  - It is sometime necessary to ensure only one client is accessing the database at any time.
  - And that if multiple queries are necessary that they are completed without interference.

Locks

- The solution to this problem is locks.
  - It is possible to lock both whole tables or a set of rows.
  - We'll deal only with whole table locking.

```
LOCK TABLES table1 WRITE, table2 READ, …;
```

Locks

- Tables can be locked for read or for read and write.
- All the tables to be locked by a session must be listed in a single `LOCK TABLES` query.
  - Any locks already held by the session will be released before the new locks are applied.
- Multiple sessions can lock the same table for reading, but only one may lock it for writing.
- The `LOCK TABLES` query will not complete until all the locks are obtained.

Locks

- Once locked, multiple queries can be issued that access the tables that are locked.
- Once the set of queries are complete, the tables should be unlocked.
- All the tables will be unlocked at once.
  - There is no scope to release the locks on a subset of the tables held.

```
UNLOCK TABLES;
```

Pros and Cons of Locks

- Locks can help ensure database consistency when accessed by multiple sessions.
- It may be more efficient.
    - A locked table may not be written to memory until it is unlocked.
- But write locks will prevent other sessions accessing the database.
    - It’s possible to create SELECT queries that take a long time to execute.
    - My personal record is 47 minutes for a POS/booking system monthly summary.
    - But I know of one local government system that was over three days.
        - They executed it on a snapshot of the database.

Summary

- Transactions
- API
  - Sqlite3
  - MySQL
- Locks