Databases: The other stuff

Transactions, APIs and the other kinds of database

Jo Hallett

December 1, 2025

The story so far…

Two weeks ago we discussed database design

- We talked about normal forms and doodling designs

Last week we discussed SQL

- We talked about how to query a database

This week…

- Well when was the last time you ever heard of someone programming something in SQL only?
- (…and a bit of logic programming)

Domain specific languages

SQL is a domain specific language

- It's great for one thing (database queries)
- But rubbish for general programming
  - A bit like regular expressions!

You don't want to have to program in SQL all the time

- So how do you get at it from a more normal programming language?

API bindings

Different programming languages do different things!
SQLite provides bindings for C that are roughly portable

- Some programming languages will take the C API and make an interface that's more or less the same as the C one
- Some programming languages will create a brand new API that better fits with the language
- Some will do something else…

For example Eitaro Fukamachi's SXQL for Commonlisp…

```
(select (:id :name :sex)
  (from (:as :person :p))
  (where (:and (:>= :age 18)
               (:< :age 65)))
  (order-by (:desc :age)))
```

Python and SQLite

Python provides SQLite bindings through the `sqlite3` library.

- And it's provided as part of a core Python install

```
import sqlite3
con = sqlite3.connect("chinook.db")
cur = con.cursor()
for row in cur.execute("SELECT␣*␣FROM␣Album␣LIMIT␣3"):
    print(row)
cur.close()
con.close()
```

```
(1, 'For Those About To Rock We Salute You', 1)
(2, 'Balls to the Wall', 2)
(3, 'Restless and Wild', 2)
```

Well except…

What if something goes wrong in the middle?

- I might crash with my database still open

```
import sqlite3
try:
    con = sqlite3.connect("chinook.db")
    try:
        cur = con.cursor()
        for row in cur.execute("SELECT␣*␣FROM␣Album␣LIMIT␣3"):
            print(row)
    finally:
        cur.close()
finally:
    con.close()
```

```
(1, 'For Those About To Rock We Salute You', 1)
(2, 'Balls to the Wall', 2)
(3, 'Restless and Wild', 2)
```

What about if I'm adding data?

What happens if I'm adding multiple bits of data

```
import sqlite3
try:
    con = sqlite3.connect("chinook.db")
    try:
        artist = (276, "Belle␣and␣Sebastian")
        albums = [(348, "Dear␣Catastrophe␣Waitress", 276),
                  (349, "Write␣About␣Love", 276)]
        cur = con.cursor()
        cur.execute("INSERT␣INTO␣Artist(ArtistId,␣Name)␣VALUES␣(?,?)", artist)
        for album in albums:
            cur.execute('''
␣␣␣␣␣␣␣␣␣␣␣␣INSERT␣INTO␣Album(AlbumId,␣Title,␣ArtistId)
␣␣␣␣␣␣␣␣␣␣␣␣VALUES␣(?,?,?)''',
                        album)
        #con.commit() #Uncomment if you want to save changes!
        for row in cur.execute('''
␣␣␣␣␣␣␣␣SELECT␣Artist.Name,␣Album.Title
␣␣␣␣␣␣␣␣FROM␣Album␣JOIN␣Artist␣ON␣Album.ArtistId␣=␣Artist.ArtistId
␣␣␣␣␣␣␣␣WHERE␣Album.ArtistId␣=␣276'''):
            print(row)
    finally:
        cur.close()
finally:
    con.close()
```

None

? ?

What are all those ?'s about?

```
cur.execute("INSERT␣INTO␣Artist(ArtistId,␣Name)␣VALUES␣(?,?)", artist)
```

Why not?

```
cur.execute("INSERT␣INTO␣Artist(ArtistId,␣Name)␣" +
          "VALUES␣("+artist[0]+","artist[1]")")
```

Little Johnny Drop Tables

Suppose you have a login system on a website.

```
cur.execute("SELECT␣FROM␣users␣WHERE␣username='"+username+"'␣" +
          "AND␣password='"+password+"'")
```

If your database can find a row that matches their username and password they get to log in!
My password is `fish' OR 'fish' = 'fish`

- Good secure password
- What will happen if I try and login?

```
cur.execute("SELECT␣FROM␣users␣WHERE␣username='jo'␣" +
          "AND␣password='fish'␣OR␣'fish'␣=␣'fish'")
```

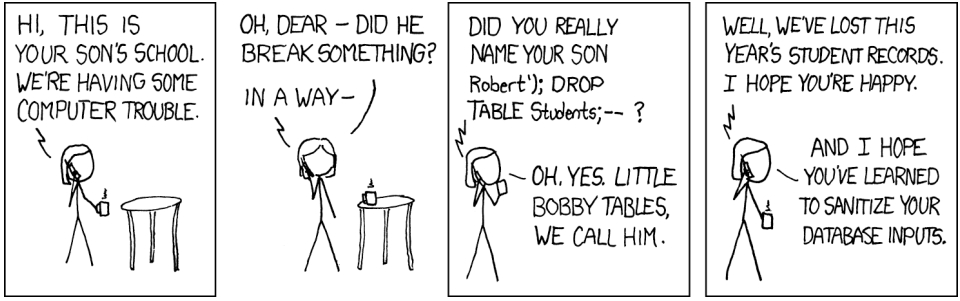
What would happen if Matt tried to log in with my password?

Prepared Statements

SQL Injection vulnerabilities

Prepared statements are a safety measure

- When I insert data it goes through what I've added and escapes it



Don't get your database hacked

- Use prepared statements
- …and don't store passwords like this

I'll commit to that

```
#con.commit() #Uncomment if you want to save changes!
```

Are my changes not normally saved?

What if something goes wrong?

Suppose I'm adding lots of albums and artists…

- I get halfway through and then my program crashes
- What happens if I try and restart?

```
.schema Artist
```

```
CREATE TABLE [Artist]
(
    [ArtistId] INTEGER NOT NULL,
    [Name] NVARCHAR(120),
    CONSTRAINT [PK_Artist] PRIMARY KEY ([ArtistId])
);
```

A UNIQUE sort of pain

If I restart I have to:

- Work out how far I got and undo all bits I've already done
- Work out how far I got and not skip some of the entries to do

Both sound like a pain.

Transactions

- Let me define a bunch of database queries as a single atomic operation
- If any fail I can roll back all the changes and start again

Until I `commit()` my work nothing is saved

Well, technically you can do this from SQL directly…

```
BEGIN TRANSACTION;

INSERT INTO Artist(ArtistId, Name)
VALUES (4949, "Geordie␣Greep");
INSERT INTO Album(AlbumId, Title, ArtistId)
VALUES (12345, "The␣New␣Sound", 4949);

ROLLBACK TRANSACTION; -- or...
COMMIT;
```

But unless you work with SQL directly, it's not that common…

Java and the JDBC

With Python, we loaded SQLite's API

- Java tries something different

Rather than implementing SQL API's for every database engine…

- Implement a general framework for database access
  - The Java Database Connectivity or JDBC
- Database engines provide they're own bridge to the JDBC API

So how do I use it?

```
import java.sql.*;

try (
    final var connection = DriverManager.getConnection("jdbc:sqlite:chinook.db");
    final var statement = connection.prepareStatement("SELECT␣name␣FROM␣Artist␣WHERE␣name␣LIKE␣?");
) {
   statement.setString(1, "B%");
   final var results = statement.executeQuery();
   while (results.next()) {
      System.out.println(results.getObject(0));
   }
} catch (SQLException err) {
   err.printStackTrace(System.err);
}
```

```
java.sql.SQLException: No suitable driver found for jdbc:sqlite:chinook.db
```

And with the CLASSPATH fixed…

Go and fetch the driver you need (or use Maven) and stick it in your classpath

```
import java.sql.*;

try (
    final var connection = DriverManager.getConnection("jdbc:sqlite:chinook.db");
    final var statement = connection.prepareStatement("SELECT␣name␣FROM␣Artist␣WHERE␣name␣LIKE␣?");
) {
   statement.setString(1, "Q%");
   final var results = statement.executeQuery();
   while (results.next()) {
      System.out.println(results.getObject(1));
   }
} catch (SQLException err) {
   err.printStackTrace(System.err);
}
```

```
Queen
```

That's all folks

We've covered:

- Some database theory
- SQL syntax
- Interacting with Databases from programming languages

Well except…

Lets make a tree in SQL!

```
CREATE TABLE tree
( id INTEGER PRIMARY KEY AUTOINCREMENT
, value TEXT NOT NULL
, left INTEGER
, right INTEGER
);
```

Assuming your table contains multiple trees:

- Can you write a query to search if a given value is in a tree or not?

Recursive data structures

Turns out you can't.

- Well you can, but you have to write out queries for each level of nesting.

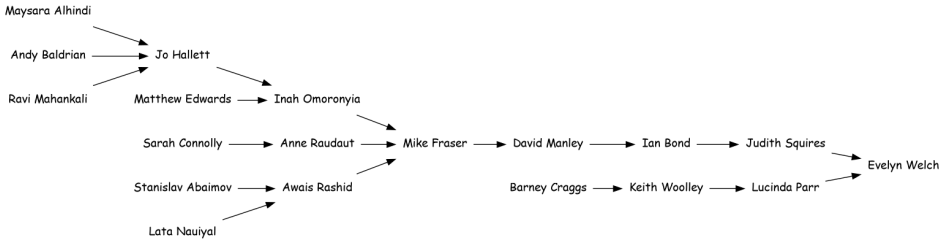
```
SELECT *
FROM tree
JOIN tree AS tree2
ON tree.left = tree2.id
JOIN tree AS tree3
ON tree2.left = tree3.id
JOIN tree AS tree4
ON tree3.left = tree4.id
JOIN tree AS tree5
ON tree4.left = tree5.id
JOIN tree AS tree6
ON tree5.left = tree6.id
JOIN tree AS tree7
ON tree6.left = tree7.id
-- and so on...
```

This is tedious.

How do we express these in SQL

Trees and other graph structures turn up a lot in real life; but SQL can't handle them in general For example:

- Suppose you are storing a database of employees supervisors…
- To have your holiday approved you need approval from your boss, or their boss or their boss's boss…



I told Barney and Awais that I'm going on holiday today; and they said it was fine…

- Am I allowed to take the day off?

I could do this in SQL…

```
import sqlite3
person = "Jo Hallett"
boss1 = "Awais Rashid"
boss2 = "Barney Craggs"

def canAuthorizeLeave(person, boss):
    con = sqlite3.connect("bosses.db")
    cur = con.cursor()
    for row in cur.execute("SELECT manager FROM manager WHERE employee IS ?", [person]):
        if row is None:
            return False
        elif row[0] == boss:
            return True
        else:
            return canAuthorizeLeave(row[0], boss)
    cur.close()
    con.close()

if canAuthorizeLeave(person, boss1) or canAuthorizeLeave(person, boss2):
    print("Leave approved!")
else:
    print("Go ask someone else!")
```

```
Go ask someone else!
```

Yuk

But it feels clunky…

- Can't do from SQL alone
- Have to make repeated queries
- Surprisingly large amount of boilerplate for one line of database query

The thing is if we were to write this out in formal logic its relatively trivial…

$$\text{R1}\quad \frac{\Gamma \vdash \mathsf{manages}(\mathsf{Person}, \mathsf{Boss})}{\Gamma \vdash \mathsf{canAuthorizeLeave}(\mathsf{Person}, \mathsf{Boss})}$$

$$\text{R2}\quad \frac{\Gamma \vdash \mathsf{manages}(\mathsf{Person}, \mathsf{Boss}) \qquad \Gamma \vdash \mathsf{manages}(\mathsf{Boss}, \mathsf{BossesBoss})}{\Gamma \vdash \mathsf{manages}(\mathsf{Person}, \mathsf{BossesBoss})}$$

Proof!

I could even generate a proof tree to prove it…
What if I had asked Ian…

- Read $\Gamma \vdash$ as we know that…
- (Or more accurately: Γ contains statements from which we can derive that…)

$$\dfrac{\dfrac{}{\Gamma \vdash \mathit{manages}(\mathrm{jo}, \mathrm{inah})}\,\text{R1} \quad \dfrac{\dfrac{}{\Gamma \vdash \mathit{manages}(\mathrm{inah}, \mathrm{mike})}\,\text{R2} \quad \dfrac{\dfrac{}{\Gamma \vdash \mathit{manages}(\mathrm{mike}, \mathrm{david})}\,\text{R2} \quad \dfrac{}{\Gamma \vdash \mathit{manages}(\mathrm{david}, \mathrm{ian})}}{\Gamma \vdash \mathit{manages}(\mathrm{mike}, \mathrm{ian})}}{\Gamma \vdash \mathit{manages}(\mathrm{inah}, \mathrm{ian})}}{\Gamma \vdash \mathit{canAuthorizeLeave}(\mathrm{jo}, \mathrm{ian})}$$

(I was gonna ask Evelyn to authorize my leave but the tree got too big to fit on the slide :-S)

If only we had a database language based on first order logic…

Datalog

Datalog is a database language based on first order logic

- Simplified version of the logic programming language Prolog
- All facts/assertions are written as Horn clauses
  - `predicate(arguments)`
- Variables are capitalized, constants are not
- Rules separated with a `:-`, conjunctions with `,`
  - `canAuthorizeLeave(P,B) :- manages(P,B).`
  - `manages(P,BB) :- manages(P,B), manages(B,BB).`
- All variables in the left hand of a rule must be referred to in the right hand side.
  - Typing is usually enough to always meet this rule…
  - `canRead(U,F) :- user(U), file(F), otherReadBitSet(F).`
  - Cannot express infinite sized sets (e.g. integers)
- Closed world assumption
  - The database contains all the facts and rules you might ever need to prove something
  - If you can't prove it with all the facts then it's false

Pros and Cons

Datalog is good because:

- Can represent trees
- Can provably find all possibilities in a reasonable time
- Can produce proofs given the right implementation
- With the right implementation can find all values of a variable satisfying a query
- With extensions can run quite complex queries

Datalog is ~~bad~~ a pain because:

- No infinite sets (no numbers)
- Research-grade implementations
- Somewhat old fashioned
  - A tool from the dark ages of AI…

But its making a bit of a comeback?

- AI is cool again?

Applications

Most databases are written in SQL

- But sometimes Datalog is a really powerful tool

Applications include:

Program analysis can you prove this property about a program?

Access control can this person do this action

Machine learning and AI automated reasoning

Google/Facebook-scale data you do not have yottabytes of data…

Access control and Datalog

I love access control papers and spent my PhD working on them

- (Well a variant of an authorization logic called SecPAL)

If you read an access control paper you will see inference rules

- Turns out knowing that you'll be able to make a decision in a reasonable period of time and with a proof of why it is correct is really good for access control…

Let's implement the UNIX DAC!

- You can read a file if you own it and the user read permission is set
- You can read a file if its owned by a group you're in and the group read permission is set
- You can read a file if the other read permission is set

First order logic

You can read a file if you own it and the user read permission is set

User

$$\frac{\Gamma \vdash \text{U owns F} \qquad \Gamma \vdash \text{F has user read bit set}}{\Gamma \vdash \text{U can read F}}$$

You can read a file if its owned by a group you're in and the group read permission is set

Group

$$\frac{\Gamma \vdash \text{G owns F} \qquad \Gamma \vdash \text{U member of G} \qquad \Gamma \vdash \text{F has group read bit set}}{\Gamma \vdash \text{X can read F}}$$

You can read a file if the other read permission is set

Other

$$\frac{\Gamma \vdash \text{F has other read bit set}}{\Gamma \vdash \text{X can read F}}$$

Given…

User

$$\frac{\Gamma \vdash \text{U owns F} \qquad \Gamma \vdash \text{F has user read bit set}}{\Gamma \vdash \text{U can read F}}$$

Group

$$\frac{\Gamma \vdash \text{G owns F} \qquad \Gamma \vdash \text{U member of G} \qquad \Gamma \vdash \text{F has group read bit set}}{\Gamma \vdash \text{U can read F}}$$

Other

$$\frac{\Gamma \vdash \text{F has other read bit set}}{\Gamma \vdash \text{U can read F}}$$

$\Gamma \vdash$ Matt owns `.ssh/id_ed25519`

$\Gamma \vdash$ `.ssh/id_ed25519` has other read bit set

Can I read Matt's private key?

And in action?

```
file('.ssh/id_ed25519').
file('jos-diary').
user('matt'). user('jo'). group('users').

canRead(U,F):- user(U), file(F),
   owns(U,F), userReadBitSet(F).

canRead(U,F):- user(U), group(G), file(F),
   member(U,G), owns(G,F), groupReadBitSet(F).

canRead(U,F):- user(U), file(F),
   otherReadBitSet(F).

owns('matt','.ssh/id_ed25519').
owns('jo','jos-diary').
member('matt','users').
member('jo','users').
userReadBitSet('.ssh/id_ed25519').
userReadBitSet('jos-diary').
otherReadBitSet('.ssh/id_ed25519').
```

I said in action?

```
$ swipl output.pl
?- trace.
true.

[trace] ?- canRead('jo','.ssh/id_ed25519').
   Call: (12) canRead(jo, '.ssh/id_ed25519') ? creep
   Call: (13) user(jo) ? creep
   Exit: (13) user(jo) ? creep
   Call: (13) file('.ssh/id_ed25519') ? creep
   Exit: (13) file('.ssh/id_ed25519') ? creep
   Call: (13) owns(jo, '.ssh/id_ed25519') ? creep
   Fail: (13) owns(jo, '.ssh/id_ed25519') ? creep
   Redo: (12) canRead(jo, '.ssh/id_ed25519') ? creep
   Call: (13) user(jo) ? creep
   Exit: (13) user(jo) ? creep
   Call: (13) group(_21294) ? creep
   Exit: (13) group(users) ? creep
   Call: (13) file('.ssh/id_ed25519') ? creep
   Exit: (13) file('.ssh/id_ed25519') ? creep
   Call: (13) member(jo, users) ? creep
   Exit: (13) member(jo, users) ? creep
   Call: (13) owns(users, '.ssh/id_ed25519') ? creep
   Fail: (13) owns(users, '.ssh/id_ed25519') ? creep
   Redo: (12) canRead(jo, '.ssh/id_ed25519') ? creep
   Call: (13) user(jo) ? creep
   Exit: (13) user(jo) ? creep
   Call: (13) file('.ssh/id_ed25519') ? creep
   Exit: (13) file('.ssh/id_ed25519') ? creep
   Call: (13) otherReadBitSet('.ssh/id_ed25519') ? creep
   Exit: (13) otherReadBitSet('.ssh/id_ed25519') ? creep
   Exit: (12) canRead(jo, '.ssh/id_ed25519') ? creep
true.
```

Are you ever really going to use Datalog?

Probably not

- But it's a cool tool
- And what's the point of teaching you Software Tools if I can't occasionally show you the weird ones?

But maybe…

- Maybe one day you'll be building a proof tool
- Or a recursive database
- And you'll be finding SQL a pain
- And you'll remember this weird little tool…

Recap

- You can talk to SQL from normal programming languages
- Math sometimes gets implemented as a programming language…
- Man I hate computers…