EDAF75
Database Technology

Lecture 5

Christian.Soderberg@cs.lth.se

February 4, 2020

Today's lecture

- A few words about keeping track of state
- Using `SQLite3` from the command line
- Connecting to a database from Java using `JDBC`
- Connecting to a database from Python using the `sqlite3` package

Administration

- Lab 1 this week, lab 2 and lab 3 the following two weeks
- The Thursday lab sessions will be *packed*, so bring something else to do while you wait

Handling state

- How do we keep track of the balance of a bank account?
- Two possible solutions:
  - using a mutable attribute (`balance`)
  - by saving all updates (i.e., deposits and withdrawals)
- Not having a mutable balance requires us to do additional computations, but it brings some really nice benefits:
  - We can now explain *why* we have the current balance
  - We can see what the balance has been at different points of time in the past
  - We don't have to lock the database while updating the balance
- Saving changes instead of state is sometimes called *event sourcing*

Running SQLite3

- DBMS's such as PostgreSQL, MariaDB, Oracle, MySQL, etc., run as servers, and offer clients which talks to the servers
- SQLite3 doesn't run as a server, but it has a simple text based client which lets us manipulate its databases
- There are also some nice GUIs for SQLite3, just as there are for most DBMS's

SQLite3– command line client

Some useful SQLite3 commands:

- `.help`: makes this slide futile
- `.tables`: shows all tables in the current database
- `.schema <table>`: shows how a table is defined
- `.dump <table>`: gives INSERT statements for creating the specified table
- `.import <filename> <table>`: imports data into a table
- `.read <filename>`: reads a script from a file
- `.save <filename>`: writes the current database to file

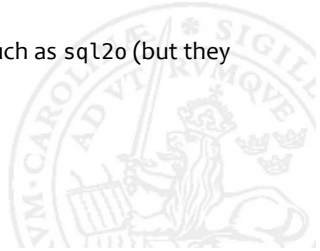
SQLite3 format

We can get various output formats, using `.mode` – some examples (there are more):

- `csv`: Comma-separated values
- `column`: Left-aligned columns
- `html`: HTML code
- `insert`: SQL insert statements
- `line`: One value per line
- `list`: Values delimited by some separator
- `tabs`: Tab-separated values

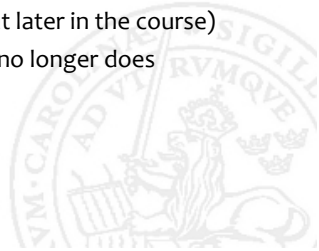
Java Database Connectivity (JDBC)

- Standard classes for handling database connections
- Can handle all relevant relational databases
- Based upon *connections*, *statements*, and *result sets*
- Lots of things can go wrong when we connect to databases, so JDBC requires lots of exception handling
- There are also some alternative, non-standard libraries, such as `sql2o` (but they often depend on JDBC)



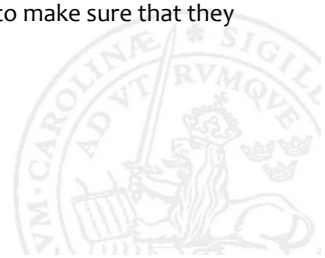
Connection

- Used to set up a *session* to a database (in the case of `SQLite3`, we don't really need to connect, the database is in a file on our hard drive, or even in memory)
- Creates `Statement`-objects, which we can use to send SQL statements to our database
- Connections also handle *transactions* (we'll talk about that later in the course)
- Used to require some strange incantations to set up, but no longer does



Statements

- `Statement`: a simple but unsafe kind of statement (amenable to *SQL injection*)
- `PreparedStatement`: a precompiled statement, safer, and more efficient when executed multiple times
- We *always* use '*try-with-resources*' to create statements, to make sure that they are closed properly when we finish

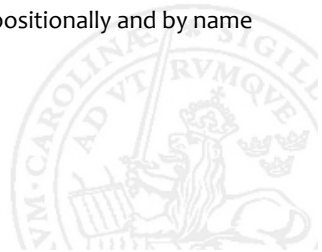


PreparedStatement

- Created with `prepareStatement(str)` on a connection, where `str` is a `String` containing a query or statement
- The query or statement can contain parameters, marked as `?` – they get their values with various `setXXX`-methods
- We typically call `boolean execute()` if we have an update statement
- For queries, we call `ResultSet executeQuery()`

ResultSet

- A `ResultSet` which represents the table of data returned from an SQL query
- It's a kind of iterator (but doesn't implement any iterator interface), we call `next()` to jump to the next row, and it returns `false` if there is no next row
- We can use various `getXXX`-methods to fetch data, both positionally and by name



JDBC, code sample

```
var found = new ArrayList<ReturnType>();
var query =
    "SELECT  ...\n" +
    "FROM    ...\n" +
    "WHERE   ... = ?\n" +
    "...\n";
try (var ps = conn.prepareStatement(query)) {
    ps.setString(1, ...);    // set parameter value
    var rs = ps.executeQuery();
    while (rs.next()) {
        found.add(ReturnType.fromRS(rs));
    }
} catch (SQLException e) {
    e.printStackTrace();
}
return found;
```

Keys, superkeys, and invented keys

- A *key* is a set of attributes which uniquely identifies each row in a table
- In our college application database, `s_id` is unique in the `students` table, but `s_name` isn't
- Of course, the tuple `(s_id, s_name)` is also unique, but here `s_name` is redundant, so we call `(s_id, s_name)` a *superkey* (which, despite its name, is way less fancy than a regular key)
- The `s_id` is an artificial value which we associate with each student, and we call it an *invented key* (or *surrogate key*)

Natural keys, and composite keys

- In the `colleges` table, `c_name` is unique in our small example, so we can use it as a key
- It is a value which occurs naturally in our problem domain, so we call it a *natural key*
- If we had a database with more colleges, we would soon end up with universities with the same name, but in different states – in that case we could use `(c_name, state)` as a key, it would be a *composite key*

Foreign keys

- In the `applications` table, we have two attributes, `s_id` and `c_name` which refers to keys in other tables, they are called *foreign keys*
- The key for the `applications` table is `(s_id, c_name, major)`, it is a composite key
- If we needed to refer to the `applications` table from other tables, we'd need the composite key as a foregin key

More about invented keys

- To avoid having big composite keys being dragged around our databases, we often create invented keys
- If the value of a key ever changes, we may have to update in many places in our database – so we normally avoid natural keys which might change, and use an invented key instead
- On the other hand: having an invented key requires us to do more joins

Different kinds of invented keys

- We can use an increasing sequence of integers as invented key – it is very common, but it's predictible, and reveals something about the state of the database
- A *uuid* (universally unique identifier) is a ≈ 128-bit random number, and it's a good choice for an invented key – we can safely assume it will be unique, and it doesn't reveal anything about the state of our system
- In `SQLite3` we can use `lower(hex(randomblob(16)))` to create something akin to a uuid (but it takes up more memory)
- The most recent version of `SQLite3` (Sqlite 3.31) has a `uuid()`-function

Generating invented keys

- In `SQLite3` we can get a uuid-lookalike using:

```
CREATE TABLE students (
  s_id     TEXT DEFAULT (lower(hex(randomblob(16)))),
  s_name   TEXT,
  gpa      DECIMAL(3,1),
  size_hs  INT,
  PRIMARY KEY (s_id);
);
```

- The database doesn't have to check if the generated value is unique, since the chance of a collision is ridiculously low