TDA357/DIT621 – Databases

Lecture 9 – JDBC, Database security

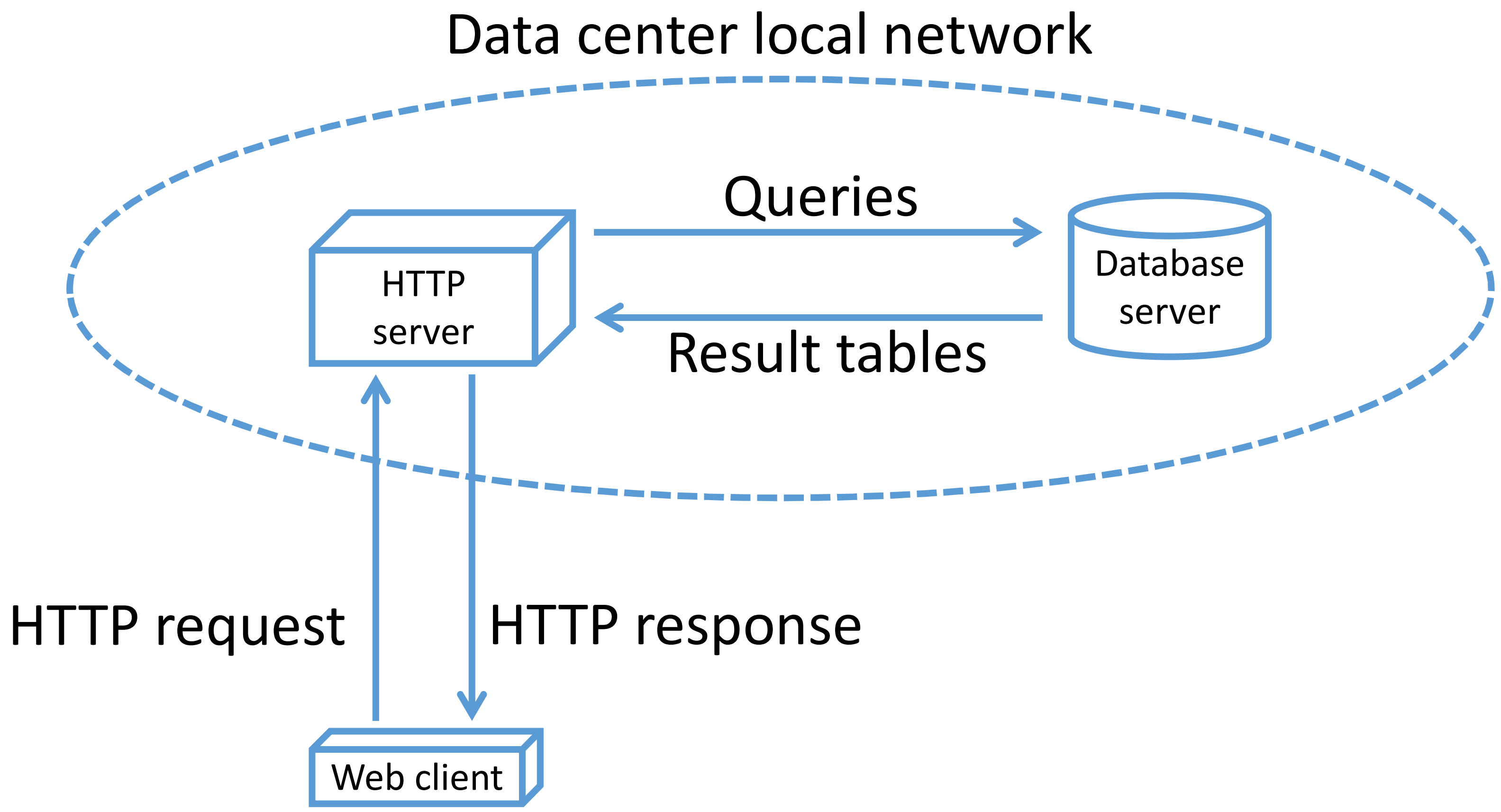
Jonas Duregård

The final piece of the relational database puzzle

- We already know how to:
  - Design a database from an informal domain description
  - Implement constraints: the user can do everything they need to do, and nothing they shouldn't be doing
  - Avoid redundancy and update/deletion anomalies
  - Query the database for information
- Issue: The only way we have communicated with the database so far is through specialized tools (psql/pgAdmin/IDE plugins/…)
- We need to learn to access the database from other applications

A typical web-service infrastructure

- The web-server software communicates with the database server
- Impossible to connect directly to the database from the internet



Database connectivity

- To connect to a database from an application, we need software libraries in the language the application is developed in
- All major programming languages have at least one such library
- JDBC (Java Database Connectivity) is a library for connecting to databases from Java
    - Can connect to lots of different DBMS using different driver classes
    - Provides a common interface (classes and methods) for running queries and processing their results in Java programs

Using JDBC – step by step

1. Load the Postgres *driver* (or another DBMS driver)
2. Initiate a single **Connection** object, by providing server URL, username etc.
3. Create one or more **Statement** or **PreparedStatement** objects from the connection (each represents a 'channel' for executing a query or a statement)
4. Executing queries through statements give a **ResultSet** object (represents a query result, that can be iterated row by row)

- Each of these objects is a resource that needs to be closed after use
- Any of these steps can fail for various reasons, throwing a **SQLException**
  - Error handling is important
- All these classes (except the Postgres driver) are described in the Java API https://docs.oracle.com/javase/8/docs/api/java/sql/package-tree.html

The 'boilerplate' code

Typical main program, loading the driver and initiating a connection

```
String DATABASE = "jdbc:postgresql://localhost/portal"
Class.forName("org.postgresql.Driver");
Properties props = new Properties();
props.setProperty("user", "postgres");
props.setProperty("password", "postgres");

// Try-with-resource (requires Java 7) closes connection automatically
try (Connection conn = DriverManager.getConnection(DATABASE, props)) {
    <actual code goes here>
} catch (SQLException e) {
    System.err.println(e);
    System.exit(2);
}
```

Load the driver

Open the connection

Deal with (unexpected) errors here

Connecting to a database on your own machine

Hostname (localhost means the machine running Java)

Name of database

- Set **DATABASE** = `"jdbc:postgresql://localhost/portal";` to connect to a database installed on your own machine (where "portal" is the name of your database, may not be needed for some configurations)
- Alternatively, you can connect to the Chalmers postgres-server using the credentials you got from Task 0, but the advantages of that are limited and requires campus presence

Try-with-resource

- You may not recognize this syntax:

```
try(Connection conn = DriverManager.getConnection(DATABASE, props)){
    <actual code goes here>
} <catch errors here>
```

- It is a relatively new feature of Java (from 2011, so not THAT new)
- Makes sure that conn.close() is called no matter what exceptions are thrown
- Replaces this old trick:

```
Connection conn = null;
try {
   conn = ...
} finally {
      if (conn != null) conn.close();
}
```

Your first challenge: Loading the driver

- For the program to work, a jar file (postgresql-42.2.18.jar) needs to be in your runtime class path
- This can be achieved in a number of ways (which makes it complicated):
  - Add it to your CLASSPATH system variable
  - Set a CLASSPATH variable in your IDE
  - Import the .jar file directly into Eclipse or similar IDE
  - Add it to the classpath when invoking the java command line tool
  - ...
- The details differ depending on operating system and Java IDE
- Good luck, and remember to use the Slack and/or Google (something like "add jar file to runtime classpath eclipse" should help)

Your first JDBC program

- Assuming **conn** is the Connection object we opened earlier
- Queries are written in Java strings
- Use try-with-resource to automatically close Statements

```
String query = "SELECT idnr,name FROM BasicInformation";
try (Statement s = conn.createStatement();){
   ResultSet rs = s.executeQuery(query);

   while(rs.next()){
      String id = rs.getString(1);
      String name = rs.getString(2);
      System.out.println(id + " " + name);
   }
}
```

Run the query

next() moves to the next row, returns false if there are no more rows

fetches column 2 (name) of the current result row

loop through whole result

The next() method in ResultSet

- Each ResultSet has an internal cursor pointing at the current row in the result
- Initially the cursor is "above" the table, pointing at no row
- If there is a row below the current one, next() will move to it and return true
  - Otherwise it closes the ResultSet and returns false
- If next() returns false or has not been called, calls to get* will throw exceptions

For this result, next() is called 3 times

```
while(rs.next()){
  System.out.println(rs.getString(1)+" "+rs.getString(2));
}
```

Result in rs

idnr	name
1111111111	S1
2222222222	S2

Call 1: cursor "above" table, move to first row and return true
Call 2: cursor at first row, move to second and return true
Call 3: cursor at second row, close and return false
(this terminates the loop!)

Single row query

- Replace the while-loop with an if-else for queries that should give a single row
- The else-clause deals with the case when no row is found

```
String query =
  "SELECT idnr,name FROM BasicInformation WHERE idnr='2222222222'";
try (Statement s = conn.createStatement();){
   ResultSet rs = s.executeQuery(query);
   if(rs.next())
      System.out.println(rs.getString(2));
   else
      System.out.println("error, no such student!");

}
```

Always a single call to next()

We could use `if(rs.next())` again to check that there are no more rows

Updates (includes deletes and inserts!)

- The executeUpdate method in Statement is for UPDATE/INSERT/DELETE
- Also for creating/dropping tables etc, but that's rarely done from applications
- Does not give a ResultSet, instead gives an int (the number of affected rows)

```
String query="DELETE FROM Registered WHERE student='1111111111'";
try (Statement s = conn.createStatement();){
    int r = s.executeUpdate(query);
    System.out.println("Deleted "+r+" registrations.");
}
```

r will be the number of rows deleted
(or sometimes more like "the number of times a trigger was executed"...)

The hassle of writing queries as Strings

- In JDBC, queries are just strings
- Requires escaping special characters: If I want to write a query like
  ```
  INSERT INTO Notes VALUES ('The "root" is C:\')
  ```
  it will look like this in Java:
  ```
  "INSERT INTO Notes VALUES ('The \"root\" is C:\\')"
  ```
- Things like line breaks in the definition are annoying (use + operator)
- Syntax errors and type errors in SQL are not discovered until runtime ☹
- Some database libraries have features for building queries using methods
  - Can catch (some) errors at compile time
  - Downside: Code is less portable and requires more knowledge of the library

String operations

- Queries can but shouldn't be built using + and similar String operations
- In the code below, if the user inputs "ccc111", **query** will be:
  DELETE FROM Registered WHERE student='111111111' AND course='CCC111'
- Common mistakes: Forgetting the last single quote, or a space after a value

```
String sid = "111111111";
String code = <request user input>;
String query = "DELETE FROM Registered WHERE student='"+sid+
               "' AND course='"+code+"'";
try (Statement s = conn.createStatement();){
   int r = s.executeUpdate(query);
   System.out.println("Deleted "+r+" registrations.");
}
```

USING THIS CODE IS A BAD IDEA – see subsequent slides

A most nefarious student

```
String sid = "1337";
String code = <request user input>;
String query = "DELETE FROM Registered WHERE student='"+sid+
               "' AND course='"+code+"'";
```

- What happens if the student inputs this course code: `"x' OR 'a'='a"`?
- The query will be:
  DELETE FROM Registered WHERE student='1337' AND course='x' OR 'a'='a'
- Ooops, that query just deleted all our registrations...

WHERE-clause is always true ☹

SQL injection attacks

- The trick on the last slide is called SQL injection (because we "inject" code into user inputs)
- In the youth of the WWW, this could be used to hack almost any website
- The counter is to sanitize input data, making sure reserved characters (like single quotes) are properly escaped
- Still, lots of programmers are too lazy to do this...

In the worst cases, you can even run arbitrary statements (that is why ; is not allowed at all in JDBC queries)

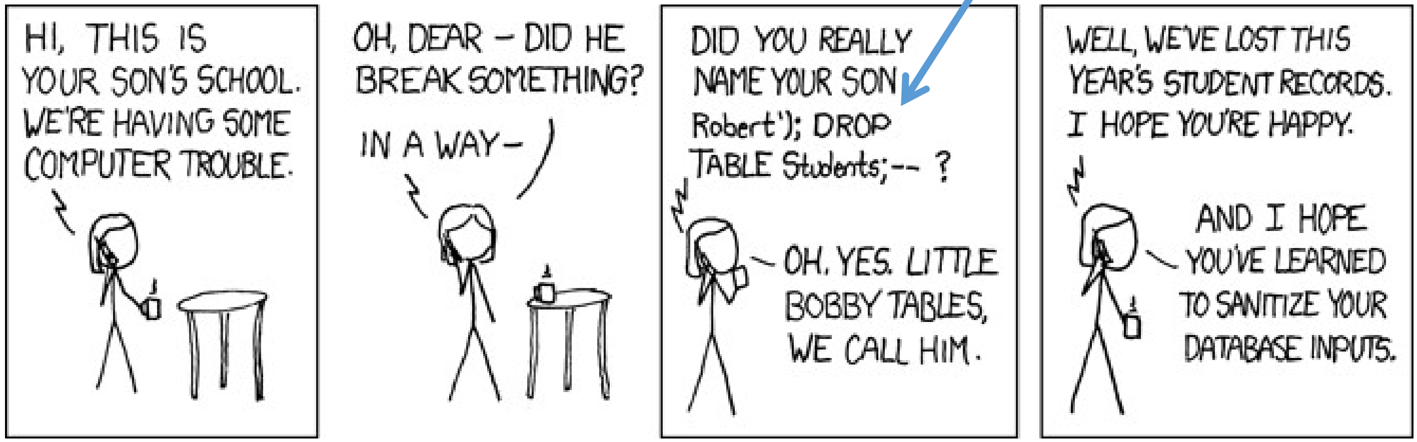

HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE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

image credit: xkcd.com

SELECTS are also vulnerable

- A query like this may seem harmless:
  ```
  "SELECT code FROM Registered WHERE student='"+student+"'"
  ```
- But for the wrong value of student it will give this query:
  ```
  SELECT code FROM Registered WHERE student='hacker'
  UNION SELECT password FROM users WHERE uname='admin'
  ```
- We just selected a list of courses ending with the password of the admin user
- Can be used to automatically extract the whole contents of the database
- ☹

An unusual SQL injection example

- In the 2010 Swedish election, someone wrote "DROP TABLE VALJ;" on their voting ballot
- The text of the ballot was then manually entered into a computer system by election workers (as a non-registered party name)
- The attack was not successful, but the vote can still be found in public records: ~~https://data.val.se/val/val2010/statistik/index.html#handskrivna~~ (link no longer works)
- There are also several examples of people writing JavaScript code on their ballots, presumably attempting to run it in the browsers of those reading the vote results (another kind of code injection)

SQL injection wins again!

- The OWASP (Open Web Application Security Project) categorizes and assess security vulnerabilities, including a "top ten vulnerabilities list"
- To absolutely no ones surprise, injection attacks remains the most common and impactful category of security vulnerabilities of the Web
- If there is ever a situation where you should have security in mind, this is it
    - Consider every user an attacker
    - There is no such thing as being paranoid about this

Do NOT do this at home

Now you know what SQL Injection is! Please use this knowledge responsibly

DO NOT EXPLORE POTENTIAL SECURITY FLAWS IN ANY SYSTEM WITHOUT EXPLICIT PERMISSION

It is illegal, unethical and possibly extremely harmful

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.

NOT a valid excuse

Prepared Statements

- Prepared statements simplify query writing, and prevents SQL injection ☺
- Each user input is replaced by '?', and set using library methods before the query is executed

```
try(PreparedStatement ps = conn.prepareStatement(
     "DELETE FROM Registered WHERE student=? AND course=?");){
    String sid = "111111111";
    String code = <request user input>;

    ps.setString(1,sid);
    ps.setString(2,code);
    ps.executeUpdate();
}
```

Two parameters (1 and 2)

Turns the Java string into an SQL string, escaping as needed and adding enclosing single quotes, placing it in parameter 2

Use prepared statements

- You should use prepared statements for all queries and statements that contain any kind of user input
- Good rule of thumbs: Always use **`prepareStatement()`** instead of **`createStatement()`** unless you have a compelling reason not to (which you will never have in this course)

Debugging JDBC code

- Getting syntax errors? Query running but not getting the result you expected?
- Add some debug printing! (Or use a proper debugger)
- Run the printed query in psql to get a better idea of what's wrong
- Always remove your debugging code before submitting!

```
try(PreparedStatement ps = conn.prepareStatement(
   "DELETE FROM Registered WHERE student=? AND course=?");){
   System.out.println(ps);
   ps.setString(1,sid);
   ps.setString(2,code);
   System.out.println("query is: " + ps);
   int r = ps.executeUpdate();
}
```

Prints the actual query being executed, including set parameters

Sanitized string

```
try(PreparedStatement ps = conn.prepareStatement(
     "DELETE FROM Waiting WHERE position=? AND course=?");){
    int pos = 1;
    String code = "x' OR 'a'='a";

    ps.setInt(1,pos);
    ps.setString(2,code);
    System.out.println(ps);
}
```

No single qoutes around numbers

Double single-quotes (☺) is how single quotes are escaped in SQL (this SQL-string contains four single quotes, like the Java string)

Output:

```
DELETE FROM Waiting WHERE position=1 AND course='x'' OR ''a''=''a'
```

Debugging JDBC, part 2

- Remember: Changes to the database are persistent!
- If you accidentally or deliberately make JDBC run a query that deletes a registration, that registration will be gone even if you recompile and re-run your program
- Solution: Re-run your "setup.sql" file in psql now and then (including creating triggers), to delete the whole database and recreate it with prepared inserts
- If you need more/different test data to test your program, add it to your setup
- A special case: If both members of the group connect to the same database, be mindful that you may interfere with one another

Your first JDBC program – done right

- The first example, using prepared statement
- No set* operations required on s in this case

```
try (PreparedStatement s = conn.prepareStatement(
       "SELECT idnr,name FROM BasicInformation");){
   ResultSet rs = s.executeQuery();
   while(rs.next()){
      String id = rs.getString(1);
      String name = rs.getString(2);
      System.out.println(id + " " + name);
   }
}
```

Example:
Check if a student is registered on a course

```
try (PreparedStatement ps = conn.prepareStatement(
     "SELECT * FROM Registered WHERE student=? AND course=?");){
   ps.setString(1, "1111111111");
   ps.setString(2, "CCC111");
   ResultSet rs = ps.executeQuery();
   if(rs.next())
      System.out.println("Yes, you are registered :)");
   else
      System.out.println("No, you are not registered :(");
}
```

Replace with user input

Repeating prepared statements

- Sometimes you need to work with several prepared statements simultaneously
- If a prepared statement is executed several times, it is generally a good idea to reuse it (just changing the parameters)
- Example: Delete all grades for everyone named Jonas and print how many where deleted for each such student (if more than 0)
  - See next slide

```
try (PreparedStatement ps1 = conn.prepareStatement(
        "SELECT idnr FROM BasicInformation WHERE name=?");
      PreparedStatement ps2 = conn.prepareStatement(
        "DELETE FROM Taken WHERE student=?");
      ){

    ps1.setString(1,"Jonas");
    ResultSet rs1 = ps1.executeQuery();
    while(rs1.next()){
       String id = rs1.getString(1);
       ps2.setString(1,id);
       int n = ps2.executeUpdate();
       if (n > 0)
         System.out.println("Deleted "+n+" grades for "+id);
    }
}
```

Create each statement only once (once per method call at least)

Run multiple times

I could run a single DELETE, but then I would not get this detailed output

Important limitation

- Any (prepared) Statement object can only have a single open ResultSet at a time
- If you need to process several results simultaneously, make sure they are from different statement objects

Avoid doing what SQL does best in Java

- Just like in PL/SQL (triggers) it is often possible to use SQL more, and procedural code less
- For instance if doing something like "list all students and for each student also list their unread mandatory courses"
    - It sounds like a nested loop, but could it be done using a join and a single loop?
    - Not always desirable, but keep the possibility in mind
- For efficiency, fewer queries are better, so push as much work as possible into the DBMS to lessen communication

Avoid repeating your constraints

- If your database already has e.g. a unique constraint, don't run a query to make sure your insert does not violate it
- Instead, just run the insert and catch the exception you would get from violating the unique constraint!
- In particular: Don't re-implement the checks your triggers do!
- There is a **`getErrorCode()`** method in SQLException that returns (DBMS-specific) error codes for different kinds of errors
  - See the PostgreSQL documentation for error code values
  - Or just provoke the error you want to identify, and check the error code you get!

Writing complex queries in JDBC?

- It's possible to write the whole PathToGraduation query directly in Java/JDBC
- It's incredibly annoying, since you will be writing all your code in a String literal
- Rule of thumb: Stick to using simple queries in Java, and write complex queries by creating views in .sql files
    - Less risk of runtime errors (views are syntax/typechecked when created)
    - Easier to test the queries
    - Easier to write the queries with syntax highlighting and without quoting
    - Easier for the DBMS to optimize

Another security issue

- Do you plan on making a web service with user logins?
- Never, ever create a table that contains passwords in plain text
- Sooner or later, someone will hack your database and they will (most likely) have the default password of all your users ☹
  - Remember: Everyone is an attacker
- Instead, you should store a *cryptographic hash* of the password
  - A hash function reduces an arbitrarily large string to a fixed size number
  - You may get the same hash value for different strings, but only rarely
  - Similar to hashcodes in Java objects, but cryptographic hash functions are a lot harder to reverse (hard to find a string that gives a certain hash value)

Google does not know your google password

- Fun fact: Google may know a lot about you, but they may not know your google login password
  - This is because they do not store the password, only its hash value
  - When you login, the password you enter is hashed and the hash is compared to the one stored in their database
  - This is one reason why password resets do not send you your password
- Fun fact 2: You do not have a single password, you have infinitely many*
  - Because of how hashes work, there are infinitely many passwords that give the same hash, and any one of them works for login**

  * most (possibly all) of those passwords exceed the limit for password length

  ** things like salting makes this claim more dubious

BONUS SLIDES!

That I will probably not have time for

Efficiency of databases

In this course we do not talk a lot about efficiency, primarily for two reasons:

1. Predicting database performance is difficult due to automatic optimizations
   - Writing a more complicated query for efficiency may make no difference
   - Worse: It may degrade performance because the DBMS fails to optimize it
2. Premature optimization is a problem
   - A lot of people worry about performance when they should be worrying about correctness and ease of use (and productivity)

- A good approach to efficiency in most cases: Write a simple and elegant solution. If it is too slow, write a messy but hopefully efficient solution
    - Use the simple solution as a reference to test the messy one

A famous quote (my emphasis)

"Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We *should* forget about small efficiencies, say about 97% of the time: **premature optimization is the root of all evil**. Yet we should not pass up our opportunities in that critical 3%."

- Donald Knuth (1974)

About complexity of database operations

- Analyzing database operations using asymptotic complexity (O-notation) can be misleading
- The reason is that most work of a typical persistent database is spent on:
  - Disk access (reading/writing large blocks of data from disk)
  - Network communication (receiving queries and sending results)
- The former means that rather than the number of rows, we should discuss how many different disk locations we fetch rows from
- The latter means running fewer queries is better

General techniques for efficient databases

- Today I will show you two techniques that can be used to make databases more efficient:
  - Indexes
  - Materialized views
- Neither of them are "silver bullets"
  - All obvious optimizations are done automatically by the DBMS
  - Both techniques can inadvertently degrade performance
  - Require careful consideration

Indexes

- I have a large printed collection of karaoke tracks, organized by producing label
- Users typically want to look up tracks by artist or title
- Problem: Searching through the whole collection linearly is time consuming
- Solution: Print two indexes, one where all tracks are ordered by artist and one where they are ordered by title
  - Users can binary search (most of them without knowing it)
  - Whenever we add new tracks to the collection, we also need to update both the indexes

Indexes in SQL

- An index is separate from, but connected to a table
- It is created on a set of columns on that table
- It allows us to quickly find all rows with given values for those columns
  - An index on (artist,album) would allow us to quickly find all karaoke tracks with a certain artist and album name
- How lookups work is DBMS-specific, but hash-tables are typically used
- Created automatically for primary keys and UNIQUE-constraints
- Automatically used when selecting/updating/deleting using WHERE-clauses
- Updated automatically when a row is deleted/updated/inserted
  - May make inserts a lot slower
- Uses disk space, often more than the table itself

Why not create all indexes automatically?

- For a table with 10 columns, there are 2^{10}-1=1023 possible indexes
- Updating a thousand indexes with each insert/update/delete would be extremely detrimental to performance
- Disk space usage could suddenly become an issue if you use about 1000 times more space
  - Megabytes become gigabytes, gigabytes become terabytes...

When should indexes be used?

- These conditions give a hint that using indexes may be a good idea:
  - You have many rows (if you have a few hundred rows, the extra disk access for reading the index is more time consuming than a linear lookup)
    - Why are you even worrying about performance for hundreds of rows?!
  - You are frequently doing lookups/joins etc. on a non-key
  - You are not worried about inserts being slower or disk space issues

Creating indexes

- Most DBMS support the statement:
  CREATE INDEX index_name ON table(attributes);
  - Not part of the SQL standard
- Can be done on existing tables with data (may take some time to create)
- Existing SQL queries do not need to be changed in any way ☺

Materialized views

- The views you have been writing are *virtual*
    - They are just a name for a query
    - Using a view in a query FROM-clause is the same as using a subquery
- Obvious opportunity for performance gain:
  Caching the result of the view could save a lot of time
- Obvious new performance problem:
  When the table data is changed, the query result needs to be updated

Materialized views in SQL

- Replace CREATE VIEW by CREATE MATERIALIZED VIEW
- Instead of just writing a query, you create a special kind of table that is automatically updated to reflect the result of the query
    - May have to recompute the whole query when a table is updated
    - If updates are more frequent than selections, the materialized view will be less efficient than a virtual view

Materialized views in postgres

In postgres, things work a bit differently:

- Materialized views are NOT automatically updated
- They reflect only the data that is in the tables when it is created
- User needs to run REFRESH MATERIALIZED VIEW; to update it
  - Can be done via a trigger on underlying tables to emulate the standard behavior (not a FOR EACH ROW trigger though!)

When should materialized views be used?

- You have a very costly query in a view
- One of two situations:
  - The underlying tables are rarely modified and view is often selected from
  - You are OK with the view showing slightly outdated data (postgres specific)
- The latter could be implemented by scheduling a REFRESH to be done e.g. once per hour/day/week
  - Example: We have a view that shows the number of members in all Facebook groups, but it doesn't have to be up to this minute