EDAF75
Database Technology

Lecture 10

Feb 27, 2020

Administation

- On Tuesday next week, March 3, we'll have a peer review session where you'll discuss your project designs with other groups – *participation is required from everyone who wants credits for the project*, and you must sign up before 23:59 on Sunday
- To report the progress of your project, you have to use a git repository on `gitlab.com`, please let me know it you want to learn how to use git
- Today we'll have a quick look at the project, and then go through the remainder of the course material

Today

- Short intro to the project
- More on foreign keys
- SQL injection
- Indexes and query planning
- Limitations of SQL
- Stored procedures
- Alternatives to SQL (NoSQL)
- Scaling and Map/Reduce

Project

- Design a database for a small bakery (Krusty)
- Participate in a design meeting, where your group will discuss your design with two other groups – you'll also give feedback on the other groups designs
- You must bring an ER-diagram showing your design to the meeting, in three copies (one for each group) – the design doesn't have to be perfect, but should be reasonably good (so, spend at least a couple of hours on it beforehand)
- The ER-diagram *must* be in proper UML (i.e., no "Crows foot", "Chen", or other notations)

Project

- Each group will 'report' to a project supervisor (PS)
- Your code and report should be kept in a *private* repository on `gitlab.com`, and you'll share it with your PS
- The report should be a Markdown file called `README.md` – the problem text describes what should be in it (we try to keep things simple, so it isn't much)

Project

- As soon as you're finished, push your code and report to `gitlab.com`, and notify your TA
- Each project supervisor will have *lots* of project to keep track of, and they will at the same time be busy with other courses, so unfortunately the feedback may sometimes take several days
- Deadline: you must have pushed your final (approvable) version to `gitlab.com` no later than 23:59 on April 30

Project

- Some of you have programmed a lot before taking this course, many of you haven't – we'll require that all of you do your best (but no more), *and that you follow common coding guidelines*, such as:
  - Never, ever, use tabs in your Java or Python code!
  - Always use correct indentation (4 spaces, in both Java and Python), and place braces where the standard guidelines put them
  - Functions/methods should generally do only one thing each – a function/method which does many different things should be broken into several functions/methods
  - Use proper names – functions/methods/parameters should have descriptive names, local variables can be shortened (i.e., how descriptive a name should be depends on how big its scope is)

Transactions and triggers

- Last time we looked into transactions and triggers
- The project will be much easier to implement if you use transactions and triggers properly!
- See some examples of transactions and triggers in the notebook

Example

Example

Write a program which keeps track of the names of your friends, and write code to insert new friends

SQL Injection

- We must be careful before putting user data into our database – never concatenate parameters into a query!
- SQL injection is when a user sends a string which alters the intended meaning of our SQL statement
- A classic example is the statement

```
stmt = "SELECT * FROM users WHERE name = '"
       + user_name + "';"
```

  and `user_name` gets the value `"' OR '1'='1"`
- Using `PreparedStatement` instead of `Statement` makes our code safer
- In the project, we'll require that you use a `PreparedStatement` where a `Statement` would have been dangerous

SQL injection

- On wikipedia there is a long list of known SQL injections (and we probably don't hear about most of the successful ones)
- So, SQL injection is a real thing, and we'd better be safe than sorry – using a `PreparedStatement` is a very simple protective measure
- In languages such as C/C++, there are relatives to SQL injection – one well known example is "buffer overruns", in which someone crafts a message which overflows the buffer created to hold the reply (this was one of the exploits used by the "Morris worm")
- Very few programmers have regretted writing code which was 'too safe'

Indicies

- When we define a primary key, the database creates a special *index* to make searches for the key fast – indexes can also speed up joins and `ORDER BY` statements
- We can create our own indexes, with as many columns as we want

```
CREATE INDEX names_and_ages
ON employees(last_name, first_name, birth_year);
```

- This index will speed up searches for:
  - last name
  - last name and then first name
  - last name and birth year (with some fiddling)
- The index above will not help much if we're just searching for first name, or birth year
- Indicies make some things faster, but insertions and deletions will become slower
- We can sometimes create a *covering index*, it is an index which includes the value we're normally looking for – using a covering index the DBMS will not even have to look at the table itself

Indicies and the Query Planner

- For each SQL statement, there might be thousands of ways to perform the operation
- Before a DBMS executes a statement, its *query planner* takes a good look at it, to find a way to execute the statement efficiently
- Indicies are very important during the planning – when searching for a value which isn't indexed properly, the DBMS might have to do a linear search through a table
- Sometimes big joins (especially cross joins) gives the query planner so many alternatives that the planning itself takes substantial time
- In `SQLite3` there is a command `EXPLAIN QUERY PLAN` which explains what will happen during a given query

Exercise

Exercise

Define a table with employees of a company, and information about their immediate supervisors.

- Write a query which finds the name of the immediate supervisor of all employees
- Write a query which finds the names of the supervisors of the supervisors of all employees
- Write a query which finds all the supervisors (transitively) of an employee

Limitations of SQL

- For a long time, SQL lacked recursion (and loops), and many DBMS's still do
- That means there are many simple things we can't do easily in SQL, such as traversing a varying number of steps in some kind of list-like structure (e.g., a line of ascendants in a tree)
- That shortcoming can be dealt with in several ways:
  - by moving the iterative code to the clients (so we write our recursive calls and loops in another language, like Java or Python)
  - by using *stored procedures*
  - by using a graph database
- Many DMBS's now have some kind of recursion (`SQLite3` is one of them), but using it is somewhat difficult and error prone

Recursion in PostgreSQL/SQLite

- In PostgreSQL and `SQLite3`, as an example, we can 'loop' to create a table with the numbers 1..10:

```
WITH RECURSIVE cnt(x) AS (
  VALUES(1)
  UNION
  SELECT x+1
  FROM   cnt
  WHERE  x<10
)
SELECT x FROM cnt;
```

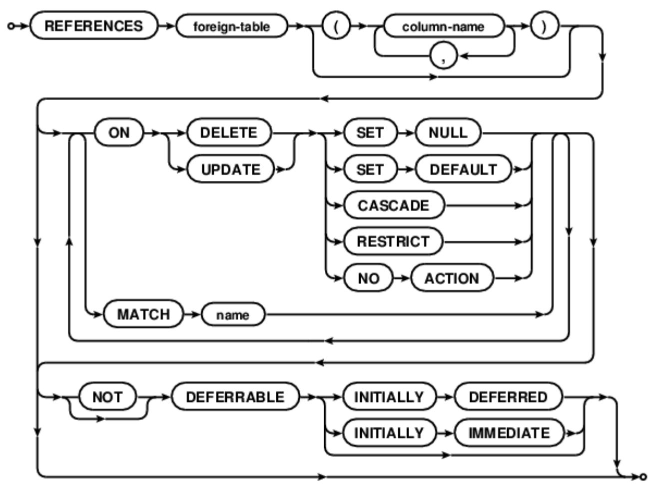
- A detailed understanding of recursive queries is beyond the scope of the course, but I want you to be aware of them

Foreign key checking

- When we use foreign key constraints, how do we handle changes in the 'foreign' table?
- In `CREATE TABLE` we have several ways to define what should happen when a foreign key changes: on both `UPDATE` and `DELETE` we can do:
  - `NO ACTION`: doesn't do anything (obviously...)
  - `RESTRICT`: makes sure that the parent key can't change when there are foreign keys mapped to them
  - `SET NULL`: when a parent key is updated or deleted, the attribute is set to `NULL`
  - `SET DEFAULT`: when a parent key is updated or deleted, the attribute is set to its default value
  - `CASCADE`: propagates the change from the parent table to the referencing tables
- Some of the actions may put the database in a corrupt state
- In `SQLite3` we must enable foreign key checking with:

```
PRAGMA foreign_keys = ON;
```

Foreign key checking



Computer times adjusted to human scale (2017)

One CPU cycle	0.4 ns	1 s
Level 1 cache access	0.9 ns	2 s
Level 2 cache access	2.8 ns	7 s
Level 3 cache access	28 ns	1 min
Main memory access (DDR DIMM)	100 ns	4 min
Intel Optane DC SSD I/O	<10 µs	7 hrs
NVMe SSD I/O	25 µs	17 hrs
SSD I/O	50-150 µs	1.5-4 days
Rotational disk I/O	1-10 ms	1-9 months
Internet call: San Francisco to New York City	65 ms	5 years
Internet call: San Francisco to Hong Kong	141 ms	11 years

Practical consequences of the timescale

- We try to keep the number of network calls to a minimum – `WITH`-statements, stored procedures and triggers can help us
- We try to minimize the number of disk accesses – DMBS's often do this by using B-trees
- We try to write code which has a memory footprint which is as 'local' as possible (this is not really database related)

Stored procedures

- We've seen that SQL lacks some simple features, like user defined functions, and loops
- Often this can be alleviated by writing code in other languages, and have them call SQL queries/statements 'remotely' (like we've done in Java and Python)
- Some DBMS's have *stored procedures*, which are user defined functions, and in which we can use regular programming features such as parameters and loops
- One of the main advantages with stored procedures is that it reduces the need for calls over a network connection
- The way `SQLite3` operates makes stored procedures almost pointless – there are no network calls, since the database code is linked into our program

Using a stored procedure in MySQL

```
CREATE FUNCTION getTopLevelSupervisorName(IN emp_name VARCHAR(10))
       RETURN VARCHAR(10)
BEGIN
    DECLARE b_id INT;
    DECLARE b_name VARCHAR(10);

    SET b_name = emp_name;
    SELECT supervisor_id into b_id
    FROM employees
    WHERE name = emp_name;
    WHILE b_id <> 0 DO
        SELECT name, supervisor_id
        INTO   b_name, b_id
        FROM   employees
        WHERE  nbr = b_id;
    END WHILE;
    RETURN b_name;
END;
```

Big Data

- The cost of storing 1 GB of data has gone down from ca SEK 1 000 000 in the 1980ies, to less than SEK 1 today
- Today, there are many databases on the petabyte scale (that is 1 000 000 000 000 000 bytes)
- Data mining is a very active field today, and we mine for:
  - Scientific discoveries (CERN invented the web for this purpose)
  - User behavior (customize advertising)
  - Economic data (algorithmic trading)
  - ...
- To find interesting patters in our data, we often want to store many kinds of data:
  - text
  - urls
  - images
  - sound
  - video

SQL vs. NoSQL

- Relational databases have been a phenomenal success, for several reasons:
  - ACID
  - Well tested technology
  - An enormous amount of money invested in them
- It is not a panacea for all data, though
  - They require that we convert all data into tables
  - They may be more complex than necessary
  - They're sometimes not fast enough
  - They don't scale very well (more about that soon)

NoSQL

- In 2000-2016, many alternatives to relational databases sprung up – for some reason they got the moniker NoSQL (it should really have been NonRelational)
- Initially it meant literally "No SQL", but it has evolved into meaning "Not Only SQL", since many of them embed some SQL like query language themselves
- NoSQL databases allow users to save and access all sorts of data in simple ways
- NoSQL databases often sacrifice some of the ACID properties, sometimes they replace "Consistency" with "Eventual consistency"

NoSQL

- There are different kinds of NoSQL databases:
  - *Key-value store*: this is the simplest kind of database, it's basically a `Map[K,V]`, where the database has no way of querying on the contents – Redis is a popular key-value store
  - *Document store*: this is an enhanced key-value store, where we can seach and manipulate data based on the contents, not only the keys – MongoDB and CouchDB are popular document stores
  - *Column store*: here we have tables with rows containing columns of data, but unlike relational databases, the names and formats of the columns can vary between rows – BigTable and Apache Cassandra are popular column stores
  - *Graph databases*: Neo4j (from Malmö!) is the most popular example

NoSQL

Exercise

Use MongoDB to insert some student with their applications, and to make some simple queries.

Scaling

- Handling bigger amounts of data requires that we add computing power and storage space, we can do it in at least two ways:
  - *Vertical scaling*: updating our processor and expanding our memory – this is also called *scaling up*
  - *Horizontal scaling*: adding more processors/computers and more hard drives – this is also called *scaling out*
- Horizontal scaling is normally cheaper, and has greater potential

Capitalizing on horizontal scaling

- If we scale horizontally, we need a way to use our resources in parallel
- Google used *map-reduce* for a long time – a simplistic description:
  1. We distribute our data to many servers
  2. On each server, we ’`map`’ a function to our data
  3. We then ’`reduce`’ the results from each server into one final result
- `map` and `reduce` are staples of functional programming
- Hadoop is an open source framework based on map-reduce

Horizontal scaling of databases

- Databases are sometimes split row-wise into non-overlapping partitions, this is called *horizontal partitioning*
- If the partitions are put on separate servers, we call it *sharding*
- A global company could potentially use sharding to put European customers on European servers, and American customers on American servers – this could make some queries faster
- Horizontal partitioning (and sharding in particular) enables us to use map-reduce-like algorithms
- The cost of sharding is increased complexity, and a reliance on the connections between the servers
- MongoDB and Spanner are examples of DBMS’s using sharding