Databases

Course 02807

September 24, 2019

Carsten Witt

Databases

- Database = an organized collection of data, stored and accessed electronically (Wikipedia)
- Different principles for organization of data: navigational, relational, object-oriented, non-relational (noSQL), …
- Focus here: **relational**, accessible via **SQL** (structured query language)
- Elements of relational DB: **tables** consisting of **rows,** where rows consist of **columns** [in the theory of DB, a table is a relation]
- Famous relational database systems: Oracle DB, IBM Db2, MS SQL Server, PostgreSQL, MySQL, MariaDB, SQLite, …
- Today: databases in **SQLite** (public domain, easy to use) and access via SQL, both from command line and in Python

Example: Bank Database

`accounts`

accountId	balance
Filter	Filter
1	68386.73
2	54258.96
3	99564.07
4	43651.95
5	-38743.82
6	21370.64
7	60163.15
8	-37469.4
9	75130.83
10	1337.0

`transactions`

transactionId	date	amount	fromAccountId	toAccountId
Filter	Filter	Filter	Filter	Filter
1	2018-09-15 08:51:39	21112.94	17	11
2	2018-08-25 03:42:12	61412.75	5	6
3	2018-09-10 12:47:47	89210.37	6	16
4	2017-11-25 18:27:51	56416.69	16	9
5	2018-05-30 10:20:53	93294.45	8	3
6	2017-12-31 23:07:49	21611.7	13	3
7	2018-04-13 16:39:02	85871.16	17	3
8	2018-02-04 13:24:04	47534.46	14	7
9	2018-03-19 16:10:39	93104.78	1	9
10	2018-02-13 02:19:46	15309.95	6	8
11	2018-02-15 06:42:42	95467.39	6	2
12	2018-03-22 18:45:58	73057.8	8	2
13	2018-08-25 20:03:15	17404.78	18	19
14	2017-12-29 10:03:19	55757.81	19	8
15	2018-04-10 03:23:49	68476.1	17	10
16	2018-03-16 01:57:08	39337.78	19	11

Essential SQL Commands

- `CREATE TABLE …`
- `INSERT INTO … VALUES …`
- `SELECT … FROM … WHERE … [ORDER BY …]`

  `WHERE` checks a condition, e.g. (in)equality (`"<="` `etc.`), set membership (”IN”), formulated in basic logic (use connectors AND and OR) …

- `UPDATE … SET col = val WHERE …`
- `DELETE FROM … WHERE …`
- `DROP TABLE …`

https://www.sqlite.org/lang.html

SQLite Command Line

- (apt-get install sqlite3)
- sqlite3 bankdb.sqlite
- .tables
- CREATE table accounts(accountId INTEGER PRIMARY KEY, balance REAL);
- .schema accounts
- SELECT * FROM accounts;
- CREATE TABLE transactions(transactionId INTEGER PRIMARY KEY, date TEXT, amount REAL, fromAccountId INTEGER, toAccountId INTEGER);
- INSERT INTO transactions(date,amount,fromAccountID,toAccountID) VALUES (datetime('now'), 999.98, 2, 3);
- .exit

Data Mining with SQL

- **Aggregate functions** `AVG, MIN, MAX, SUM, COUNT` compute statistic from a set of rows

- `SELECT AVG(balance) FROM accounts`

- `SELECT AVG(balance) FROM accounts WHERE balance > 0`

- Results can be split according to another column value:
  `SELECT AVG(amount) FROM transactions GROUP BY toAccountId`

SQLite from Python

(https://docs.python.org/3.7/library/sqlite3.html?highlight=sqlite3)

```
#!/usr/bin/python3
import sqlite3

conn = sqlite3.connect('bankdb.sqlite')
c = conn.cursor()

c.execute("INSERT INTO accounts (balance) VALUES (1337)")
conn.commit()
c.execute("SELECT accountId, balance FROM accounts WHERE balance > 1336")
print("First result: “, c.fetchone())

print("All remaining results: “, c.fetchall())

conn.commit()
conn.close()
```

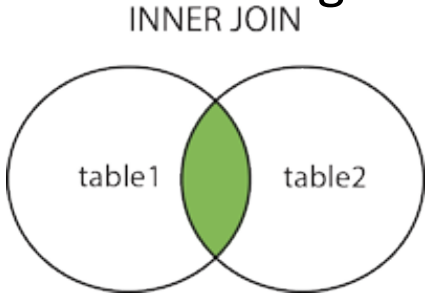
Advanced SQL Queries: Joining Tables (1/5)

- **Problem:** find all existing accounts [i.e. accounts appearing in the `accounts` table] to which there were transferred more than 100000 units of money within the last 2 months and retrieve account ID and the total amount transferred.

- **Subproblem:** find all transactions to existing accounts within the last 2 months, retrieve account ID and the total amount transferred.

Advanced SQL Queries: Joining Tables (2/5)

- **Subsubproblem:** find all transactions to existing accounts, retrieve account id and *individual* amount transferred.
- Note: `toAccountId` in `transactions` must show up in `accountid` of `accounts` table
- Concept: join results from several tables using INNER JOIN



- `SELECT transactions.amount,accounts.accountId FROM accounts INNER JOIN transactions ON accounts.accountId = transactions.toAccountId`

- May want to add `ORDER BY accounts.accountId`

Advanced SQL Queries: Joining Tables (3/5)

- Solution to subproblem:
- `SELECT transactions.amount,accounts.accountId FROM accounts INNER JOIN transactions ON accounts.accountId = transactions.toAccountID WHERE transactions.date >= date('now','-2 months');`
- Not yet! Missing the aggregation:
- `SELECT SUM(transactions.amount), accounts.accountId FROM accounts INNER JOIN transactions ON accounts.accountId = transactions.toAccountID WHERE transactions.date >= date('now','-2 months') GROUP BY accounts.accountId`

Advanced SQL Queries: Joining Tables (4/5)

- Solution to full problem: nested SQL and use of alias (”AS”)

```
SELECT mysum,myid FROM (SELECT
SUM(transactions.amount) AS mysum, accounts.accountId
AS myid FROM accounts INNER JOIN transactions ON
accounts.accountId = transactions.toAccountID WHERE
transactions.date >= date('now','-2 months') GROUP BY
accounts.accountId) WHERE mysum > 100000;
```

Advanced SQL Queries: Joining Tables (5/5)

- Alternative: grouping including additional `HAVING` condition

```
SELECT SUM(transactions.amount) AS mysum,
accounts.accountId FROM accounts INNER JOIN
transactions ON accounts.accountId =
transactions.toAccountID WHERE transactions.date >=
date('now','-2 months') GROUP BY accounts.accountId
HAVING mysum > 100000;
```

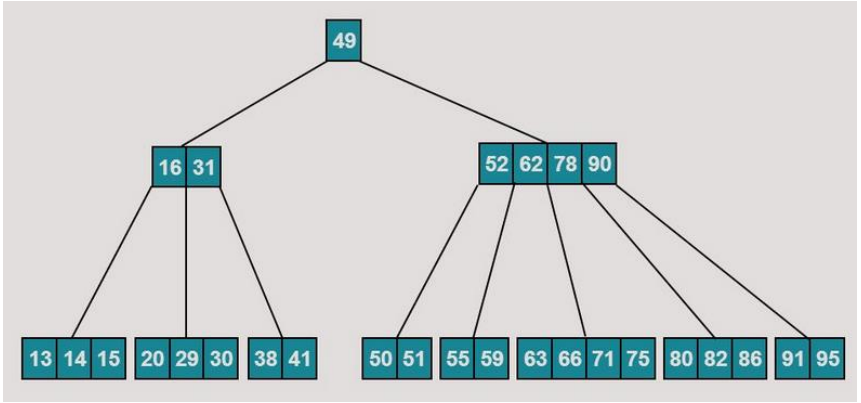
Visual SQL Tools

- DB browser for SQLite: http://sqlitebrowser.org/ available for Windows, Mac and Linux

Indexing: Concept

- Usually, contents of columns are internally stored in a list of rows.
- Disadvantages?

- Table columns can be searched efficiently by building a search tree structure on them: b-trees (extensions of binary search trees, more to come next week)



- Syntax: `CREATE INDEX indname ON table(column)`
- Extensible to multi-column indices, e.g., `CREATE INDEX indname ON table(column1, column2)`: nested search tree structure

Indexing: Example

- Python script that creates 100 000 000 accounts with random balance in 1,…,100 000 000 -> 1.4 GB SQLite database

- `SELECT * FROM accounts WHERE balance > 99999990`

  slowly reveals about 10 entries

- `CREATE INDEX balInd on accounts(balance);`

- Database file grows by 98%.
- However, the above ”select” statement now yields instantaneous results.

Indexing: Pros and Cons

- Pros: fast search on column
- Cons:
  - Additional space consumption
  - Operations such as insertion and updates take longer
    (b-trees have to be updated)
  - Correct indexing can be very complex (e.g. if multiple columns involved)

    Even if all columns have been indexed, can you quickly find all accounts where `balance + accountId = 999991`?

Summary

- SQLite databases via SQL and Python
- SQLite command line: .tables, .schema … etc.
- Python: sqlite3 library, db connection, cursor object, commit
- Basic SQL: CREATE TABLE, SELECT … FROM … WHERE, …
- Advanced queries: inner joins of two tables, aggregation, WHERE, HAVING
- Indexing to speed up search on columns

Questions?