STAT243 Lecture 7.3 Databases

1 概述 (Overview)

1.1 SQL 数据库介绍

- **SQL 数据库 (Relational Databases)**
  - 关系型数据库由多个**表 (tables / relations)** 组成
  - 每个表类似于 R 或 Pandas 的 DataFrame：
    - **列 (fields / attributes)**：存储相同类型的数据 (数值、字符、日期、类别等)
    - **行 (records)**：表示单个观测或实体
- **多表设计的意义**
  不同表之间通常共享字段 (如 ID)，从而支持**合并 (join)** 操作。
  例如：
  - `Student` 表：学生信息
  - `Teacher` 表：教师信息
  - `School` 表：学校信息
    可通过字段匹配实现数据联结。
- **核心功能**
  - 快速查询 (query)
  - 高效合并 (join)
  - 数据规范化 (normalization)

1.2 内存与磁盘使用 (Memory and Disk Use)

- **数据库**：数据存储在**磁盘**上
- **Python / R**：数据存储在**内存**中

虽然磁盘访问理论上更慢，但数据库可通过：

- 操作系统的**磁盘缓存 (disk caching)**
- 高效的索引与查询算法

实现高速访问。因此，数据库可以：

- 处理**远超内存容量**的数据集
- 依然保持较快查询性能

---

2 与数据库交互 (Interacting with a Database)

2.1 数据库系统 DBMS

- **常见数据库系统 (DBMS)**
  - SQLite
  - DuckDB
  - MySQL
  - PostgreSQL
  - Oracle
  - Microsoft Access

- **交互方式**
  - 通过 DBMS 提供的命令行接口
  - 通过 Python / R 程序访问
  - 通过图形化工具 (如 DBeaver, DataGrip)
- **客户端-服务器模式 (Client–Server Model)**
  - 服务器端 (server)：存储与管理数据
  - 客户端 (client)：通过认证后发送 SQL 查询
  - 查询结果通常是一个新的表

2.2 SQLite 与 DuckDB

- **SQLite**
  - 单文件数据库，无需服务器
  - 简洁、轻量，适合个人或本地项目
  - 限制：
    - 不能使用 `ALTER TABLE` 修改列类型或删除列
- **DuckDB**
  - 现代替代方案，推荐使用
  - **列式存储 (column-wise storage)** → 对大表的查询速度快
  - 支持直接在磁盘上操作数据，不必全部加载入内存
  - 支持**OLAP (Online Analytical Processing)** 应用

---

3 Database Schema 与 Normalization

- **Schema**：数据库的结构元数据，描述各表及字段类型
- **Normalization (规范化)**：
  将数据分散存储到多个表中，避免冗余与重复信息

3.1 示例：教育数据库

不规范设计 (Denormalized)
单表结构如下：

student ID	name	grade level	class1	grade1	teacher1	department1	age1	...

缺点：

1. 数据冗余 (重复存储教师信息)
2. 空值多 (非所有学生都有相同数量的课程)
3. 更新困难 (修改教师信息需多处同步)
4. 不支持一对多/多对多关系 (如教师属于多个部门)

3.2 改进设计：以注册 (Enrollment) 为中心的结构

student ID	student name	class	grade	grade level	teacher ID	department	age

这种设计避免了空值，但仍存在部分冗余。

3.3 规范化设计 (Normalized)

使用四张表：

3.3.1 Student

ID	name	grade_level

3.3.2 Teacher

ID	name	department	age

3.3.3 Class

ID	topic	class_size	teacher_ID

3.3.4 Enrollment

student_ID	class_ID	grade

> ⚠ **Remark**
>
> - `Enrollment` 表表示每个“学生–课程”组合，解决了不规则 (“ragged”) 数据问题
> - 这种设计也符合 **tidy data** 与 **长格式 (long format)** 的思想

3.4 主键与外键 (Keys)

- **主键 (Primary Key)**
  每行的唯一标识符
  - Student.ID
  - Teacher.ID
  - Class.ID
  - Enrollment.{student_ID, class_ID} (复合主键)
- **外键 (Foreign Key)**
  指向另一张表的主键，用于建立表之间的关系
  - Enrollment.student_ID → Student.ID
  - Class.teacher_ID → Teacher.ID
  - Enrollment.class_ID → Class.ID

3.5 多表联合查询 (Joining Across Tables)

目标：查询所有 **9 年级学生的课程成绩**

需要：

- 从 `Student` 表获取 `grade_level`
- 从 `Enrollment` 表获取 `grade`

SQL 查询如下：

SQL

```sql
SELECT Student.ID, grade
FROM Student, Enrollment
WHERE Student.ID = Enrollment.student_ID
  AND Student.grade_level = 9;
```

说明：

- 这是一个 **内连接 (inner join)**
- 类似于 R 中的 `merge()` 或 `dplyr::join()`
- 也可以显式使用 `JOIN` 语句：

SQL

```sql
SELECT Student.ID, grade
FROM Student
JOIN Enrollment ON Student.ID = Enrollment.student_ID
WHERE Student.grade_level = 9;
```

4 Stack Overflow 元数据示例 (Stack Overflow Metadata Example)

4.1 数据集概览 (Overview)

- **数据来源**：Stack Overflow (2021 年所有问答的元数据)
- **数据说明**：
  - 不包含问题与答案的具体文本，仅保留结构化元信息
  - 存储在 SQLite 与 DuckDB 数据库中
- **特征**：
  - 每个问题可以有多个答案
  - 每个问题可对应多个标签 (tags)

4.2 非规范化设计的问题 (Problems with Denormalized Design)

若将所有信息放在一张表中，会出现以下问题：

4.2.1 方案 1：每行对应一个问题

question ID	user ID	title	tag1	tag2	…	answer1 ID	answer1 user	answer1 age	answer1 name	…

- **问题**：

- 存在大量重复字段
- 难以维护多对多关系 (问题 ↔ 标签、问题 ↔ 答案)
- 空值多、扩展性差

4.2.2 方案 2：每行对应一个问答对 (question–answer pair)

question ID	user ID	title	tag1	tag2	…	answer ID	answer user	answer age	answer name

- 依然存在大量冗余与非结构化数据问题
- 无法灵活扩展或高效查询

4.3 规范化方案 (Normalized Schema)

数据库应分为多个表，并通过外键建立关系。示例表设计如下：

表名	字段
questions	questionid, creationdate, score, viewcount, answercount, commentcount, favoritecount, title, ownerid
answers	answerid, questionid, creationdate, score, ownerid
users	userid, creationdate, lastaccessdate, location, reputation, displayname, upvotes, downvotes, age, accountid
questions_tags	questionid, tag

这种结构支持：

- 一对多关系 (一个问题对应多个答案)
- 多对多关系 (问题与标签之间)
- 高效查询、连接与聚合

---

5 在 Python 中访问数据库 (Accessing Databases in Python)

Python 提供多种方式访问数据库 (SQLite, DuckDB, MySQL, PostgreSQL 等)。
核心逻辑统一：

Python

```python
cursor.execute("SOME SQL STATEMENT")
```

5.1 示例：访问 SQLite 数据库

Python

```python
import sqlite3 as sq
import os

dir_path = '/mirror/data/pub/users/paciorek/share'
db_filename = 'stackoverflow-2021.db'

con = sq.connect(os.path.join(dir_path, db_filename))
db = con.cursor()
```

```python
db.execute("select * from questions limit 3")
db.fetchall()
```

返回的结果是一个包含元组的列表，每个元组代表一行记录。

Python

```
[
 (65534165.0, '2021-01-01 22:15:54', 0.0, 112.0, 2.0, 0.0, None, "Can't update a value in sqlite3", 13189393.0),
 ...
]
```

5.2 示例：访问 DuckDB 数据库

Python

```python
import duckdb as dd

dir_path = '../data'
db_filename = 'stackoverflow-2021.duckdb'

con = dd.connect(os.path.join(dir_path, db_filename))
db = con.cursor()
db.execute("select * from questions limit 5")
db.fetchall()
```

5.3 查看数据库结构 (Inspecting the Database)

5.3.1 查看所有表名

Python

```python
def db_list_tables(db):
    db.execute("SELECT name FROM sqlite_master WHERE type='table';")
    return [table[0] for table in db.fetchall()]

db_list_tables(db)
# ['questions', 'answers', 'questions_tags', 'users']
```

DuckDB 示例：

Python

```python
con.execute("show tables").fetchall()
```

5.3.2 查看字段名称

Python

```python
db.execute("select * from questions")
[item[0] for item in db.description]
# ['questionid', 'creationdate', 'score', 'viewcount', 'answercount', 'commentcount', 'favoritecount', 'title', 'ownerid']
```

5.4 基本 SQL 查询 (Basic SQL Queries)

5.4.1 获取表数据

Python

```python
results = db.execute("select * from questions limit 5").fetchall()
type(results)      # <class 'list'>
type(results[0])   # <class 'tuple'>
```

或分步执行：

Python

```python
query = db.execute("select * from questions")
results2 = query.fetchmany(5)
```

断开连接：

Python

```python
db.close()
```

5.4.2 获取 Pandas DataFrame

Python

```python
import pandas as pd
results = pd.read_sql("select * from questions limit 5", con)
```

⚠ Remark

DuckDB 返回时可能出现警告，但结果可正常使用。

---

6 Basic SQL for choosing rows and fields from a table

6.1 SQL 属于 declarative language

SQL 属于声明式语言

- 命令式语言 (imperative)：指定执行步骤
- 声明式语言 (declarative)：描述目标结果，由系统决定执行方式

6.2 示例：Stack Overflow 查询案例

6.2.1 高浏览量问题

Python

```python
db.execute('select title, viewcount from questions where viewcount > 100000').fetchall()[:3]
```

6.2.2 前 10 名浏览量问题

Python

```
db.execute('select title, viewcount from questions order by viewcount desc limit 10').fetchall()
```

6.3 SQL 查询结构

SQL

```sql
SELECT <columns>
FROM <table>
WHERE <conditions>
ORDER BY <columns>
```

常用关键字：

关键词	功能
SELECT	选择列
FROM	指定表
WHERE	过滤行
LIKE / IN / < / > / = / !=	条件操作符
ORDER BY	排序结果

其他有用关键字包括：
`DISTINCT`, `JOIN`, `GROUP BY`, `AS`, `HAVING`, `UNION`, `INTERSECT` 等。

6.4 查询结果类型 (Result of a Query)

- SQL 查询返回的本质是一个表
- 例如：

SQL

```sql
SELECT COUNT(*) FROM questions;
```

返回一行一列的结果。
在 Python 中为：

- `fetchall()` → list of tuples
- `pandas.read_sql()` → DataFrame

6.5 分组与分层 (Grouping / Stratifying)

6.5.1 语法结构

SQL

```sql
SELECT <aggregate>(<column>)
FROM <table>
GROUP BY <column>
```

6.5.2 示例：统计标签出现次数

Python

```python
db.execute("""
select tag, count(*) as n
from questions_tags
group by tag
order by n desc
limit 25
""").fetchall()
```

返回：

```
('python', 255614), ('javascript', 182006), ('java', 89097), ('reactjs', 83180), ...
```

常见聚合函数：`COUNT`，`MIN`，`MAX`，`AVG`，`SUM`

6.5.3 使用 HAVING 过滤分组结果

若在 `GROUP BY` 之后过滤结果，需使用 `HAVING` 而非 `WHERE`：

SQL

```sql
SELECT tag, COUNT(*) as n
FROM questions_tags
GROUP BY tag
HAVING n > 10000;
```

6.6 去重 (DISTINCT)

`DISTINCT` 可去除重复行或重复值。

Python

```python
# 获取所有唯一标签
tag_names = db.execute("select distinct tag from questions_tags").fetchall()

# 统计唯一标签数量
db.execute("select count(distinct tag) from questions_tags").fetchall()
```

输出：

```
[('sorting',), ('visual-c++',), ('mfc',), ('cgridctrl',), ('css',)]
[(42137,)]
```

6.7 Simple SQL joins

要从多张表中提取数据，可以使用 **JOIN** 操作。常见语法：

SQL

```sql
SELECT <columns>
FROM <table1>
JOIN <table2>
```

```
ON <table1.column> = <table2.column>
WHERE <conditions>
ORDER BY <columns>
```

等价的简化写法：

SQL

```sql
SELECT *
FROM table1, table2
WHERE table1.id = table2.id
```

6.7.1 示例：查询带有 "python" 标签的问题

Python

```python
result1 = db.execute("""
SELECT *
FROM questions
JOIN questions_tags
ON questions.questionid = questions_tags.questionid
WHERE tag = 'python'
""").fetchall()
```

等价形式（不使用 JOIN）：

Python

```python
result2 = db.execute("""
SELECT *
FROM questions, questions_tags
WHERE questions.questionid = questions_tags.questionid
AND tag = 'python'
""").fetchall()
```

6.7.2 示例: 三表连接 (Three-Way Join)

Python

```python
result = db.execute("""
SELECT *
FROM questions Q
JOIN questions_tags T ON Q.questionid = T.questionid
JOIN users U ON Q.ownerid = U.userid
WHERE tag = 'python' AND viewcount > 1000
""").fetchall()
```

解释：

- 选取所有 **Python 标签的问题**（`tag = 'python'`）
- 同时返回 **提问者的用户信息**（`users` 表）
- 限定 **浏览量大于 1000** 的问题（`viewcount > 1000`）

6.7.3 Challenge 1

返回所有带有 Python 标签的问题的答案

SQL

```sql
SELECT A.*
FROM answers A
JOIN questions Q ON A.questionid = Q.questionid
JOIN questions_tags T ON Q.questionid = T.questionid
WHERE T.tag = 'python';
```

解释：

- 首先将 `answers` 表与 `questions` 表匹配（根据 `questionid`）
- 再将问题与 `questions_tags` 匹配，筛选出标签为 Python 的问题
- 返回这些问题的所有答案

6.7.4 Challenge 2

返回所有回答过 Python 标签问题的用户

SQL

```sql
SELECT DISTINCT U.displayname
FROM users U
JOIN answers A ON U.userid = A.ownerid
JOIN questions Q ON A.questionid = Q.questionid
JOIN questions_tags T ON Q.questionid = T.questionid
WHERE T.tag = 'python';
```

解释：

- 通过 `answers` → `questions` → `questions_tags` 的关联确定问题标签
- 过滤出所有回答了 Python 标签问题的用户
- 使用 `DISTINCT` 去除重复用户

6.8 临时表与视图 (Temporary Tables and Views)

创建视图以便重复使用：

SQL

```sql
CREATE VIEW questionsAugment AS
SELECT questionid, questions.creationdate, score, viewcount,
       title, ownerid, age, displayname
FROM questions
JOIN users ON questions.ownerid = users.userid;
```

视图可以像表一样被查询：

SQL

```sql
SELECT * FROM questionsAugment WHERE viewcount > 1000 LIMIT 3;
```

6.8.1 Challenge 3

创建一个视图，包含每个问题与其标签的组合（包括没有标签的问题）

SQL

```sql
CREATE VIEW question_tag_view AS
SELECT Q.questionid, T.tag
FROM questions Q
LEFT JOIN questions_tags T ON Q.questionid = T.questionid;
```

解释：

- `LEFT JOIN` 可保留所有问题
- 若某问题没有标签，`tag` 字段为 `NULL`

6.8.2 Challenge 4

查询从未发布过问题的用户名称

SQL

```sql
SELECT U.displayname
FROM users U
LEFT JOIN questions Q ON U.userid = Q.ownerid
WHERE Q.ownerid IS NULL;
```

解释：

- 左连接用户与问题表
- 若用户未发布任何问题，则 `Q.ownerid` 为 `NULL`
- 筛选出这些用户

6.9 索引 (Indexes)

索引是一种加速查询的机制，用于快速查找行：

SQL

```sql
CREATE INDEX count_index ON questions(viewcount);
```

- 查询性能可从线性时间 `O(n)` 提升为对数或常数时间
- 适合频繁执行过滤或连接操作的列
- 创建索引需要额外时间与存储空间，因此应仅在必要时使用

6.10 集合操作 (Set Operations)

6.10.1 语法

- `UNION`：并集
- `INTERSECT`：交集
- `EXCEPT`：差集

示例：查询既提问又回答过问题的用户

SQL

```sql
SELECT displayname, userid
FROM questions Q JOIN users U ON U.userid = Q.ownerid
INTERSECT
SELECT displayname, userid
FROM answers A JOIN users U ON U.userid = A.ownerid;
```

6.10.2 Challenge 5

查询既没有提问也没有回答过问题的用户

SQL

```sql
SELECT displayname, userid
FROM users
WHERE userid NOT IN (
  SELECT ownerid FROM questions
  UNION
  SELECT ownerid FROM answers
);
```

解释：

- 先取出所有曾经提问或回答的用户 ID
- 使用 `NOT IN` 排除这些用户，得到完全未参与者

6.11 子查询 (Subqueries)

6.11.1 子查询在 FROM 子句中

子查询可作为临时表参与连接。

Challenge 6
解释下列语句的作用：

SQL

```sql
SELECT *
FROM questions
JOIN answers A ON questions.questionid = A.questionid
JOIN (
    SELECT ownerid, COUNT(*) AS n_answered
    FROM answers
    GROUP BY ownerid
    ORDER BY n_answered DESC
    LIMIT 1000
) most_responsive
ON A.ownerid = most_responsive.ownerid;
```

解释：

- 内层子查询：统计每个用户的回答数量（`n_answered`），并取前 1000 名
- 外层查询：将这些“最活跃回答者”的信息与他们的答案和对应的问题匹配

- 结果：显示最活跃的 1000 位回答者的回答与对应问题

6.11.2 WHERE 子句中的子查询

SQL

```sql
SELECT AVG(upvotes)
FROM users
WHERE userid IN (
  SELECT DISTINCT ownerid
  FROM questions
  JOIN questions_tags ON questions.questionid = questions_tags.questionid
  WHERE tag = 'python'
);
```

作用：
计算所有曾提问 Python 标签问题的用户的平均 upvote 数。

6.12 创建数据库表 (Creating Tables)

使用 Pandas 在 Python 中创建表：

Python

```python
con = sq.connect(db_path)
student_data.to_sql('student', con, if_exists='replace', index=False)
```

此方法常用于：

- 导入整理好的 DataFrame
- 替换或新增数据库表