STAT243 Lecture 7 Databases (Reduced)

1 What We Already Know About Handling Big Data

- **UNIX 命令的高效性**
  - UNIX 命令行操作速度极快，适合大文件的行/列筛选
  - 可使用以下命令组合实现高效数据预处理：
    - `grep`：按条件提取行
    - `head` / `tail`：查看文件头尾
    - `awk`：按条件提取行或列
  - 结合 **管道操作 (piping)** 与 **shell 脚本** 可快速缩减数据规模
- **GNU Parallel**
  - 命令行并行化工具
  - 可在 Linux 集群中同时执行多个任务，提高处理效率
- **实践建议**
  - **删除无关列**：若数据有 30 列而仅需 5 列，应先提取子集
  - **抽样分析**：在许多情况下，用随机样本的结果与全量分析几乎一致
  - 通过这些简化策略，可继续使用熟悉的 Python/R 工具，而无需复杂的大数据框架
- **存储格式优化**
  - 二进制格式 (如 `.parquet`、`.feather`) 比文本格式 (如 `.csv`) 更紧凑、高效
  - 推荐在分析前将大型文本数据转换为二进制格式
- **使用数据库**
  - 对许多应用而言，将大型数据集导入 **标准数据库系统** (如 PostgreSQL、SQLite、DuckDB) 是一种稳健且高效的解决方案

2 MapReduce Overview

2.1 MapReduce 范式

- **核心思想**
  将数据分布在多个节点上，每个节点独立处理本地数据，然后再汇总结果。
  适用于超大规模数据集的分布式计算。
- **MapReduce 的四个主要步骤**
  1. **读取数据**：从文件或记录中读入原始数据
  2. **Map 阶段**：将输入转化为键值对 `{key, value}`
  3. **Reduce 阶段**：对每个键的所有值执行聚合操作，输出 `{key, result}`
  4. **输出结果**：写出最终的 `{key, result}` 集合

2.2 MapReduce 的灵活性与限制

- Map 阶段可以执行过滤、转换或扩展操作 (一个输入生成多个输出)。
- Reduce 阶段可执行复杂聚合与统计分析。
- 限制：

- Map 阶段的每个记录必须独立处理
- Reduce 阶段的每个键的聚合也必须独立
- 这确保了可并行执行

> ⚠ **Remark: "Shuffling" data**
>
> 若操作需要跨节点移动大量数据，会显著降低性能
>
> - 例如计算 **中位数 (median)** 时，需移动数据以按组重排。
> - 而计算 **均值或总和 (mean/sum)** 时，只需传输每个节点的部分和，通信量小得多。
>
> 因此，MapReduce 的效率取决于任务是否需要大量数据移动。

2.3 Hadoop

- 基于 Java 的 MapReduce 框架
- 核心组件：
  - **HDFS (Hadoop Distributed File System)**：分布式文件系统，数据自动冗余存储
  - **任务调度系统**：监控节点执行与故障恢复
- 优点：容错性强，可在大规模集群上运行
- 缺点：
  - 配置复杂，维护成本高
  - 主要用于磁盘为主的计算 (IO 密集型)

2.4 Spark

- 可视为“内存化的 Hadoop”
- **将多节点内存视为统一内存池**，支持高速数据操作
- 特点：
  - 数据能放入内存时速度远超 Hadoop
  - 若超出内存则自动使用 HDFS
- 缺点：
  - 集群配置复杂，调优困难
  - 内存溢出错误常见且难以调试

---

3 使用 Dask 进行大数据处理 (Using Dask for Big Data Processing)

3.1 基本特性

- Dask 继承 MapReduce 思想，可：
  - 在单机多核或多节点上**并行化数据处理**
  - **自动将数据分块** (chunks/shards/partitions) 并行处理
- 优势：
  1. **并行化计算与读取**：每个 worker 处理部分数据
  2. **扩展可处理数据规模**：可利用多节点的内存或磁盘

> ⚠ **Remark: 注意事项**

- 如果数据仅在一个磁盘上，多个进程同时读写会受限于磁盘 IO
- 超算系统通常具备并行文件系统，可实现真正的并行 IO
- Hadoop/Spark 通过“多节点多磁盘” (通常每个机器/节点一个磁盘) 实现分布式 IO
- 在单节点上使用 Dask 时，建议使用 **threads 调度器**，避免频繁数据拷贝

3.2 Dask DataFrame (类 Pandas DataFrame)

- **结构**：每个 Dask DataFrame 由多个小型 Pandas DataFrame 构成
- **默认调度器**：threads
- **功能**：支持大部分 Pandas 操作，但不完全兼容

> ⚠ **Remark**
>
> 多次调用 `.compute()` 会重复读取数据，应尽量一次计算完成

> ⚠ **Remark: 结果内存占用**
>
> - `.compute()` 返回的结果是普通 Python 对象
> - 若结果数据过大，可能导致内存溢出

3.2.1 示例：读取航班延误数据 (约 11 GB)

Python

```python
import dask
dask.config.set(scheduler='threads', num_workers=4)
import dask.dataframe as ddf

path = '/scratch/users/paciorek/243/AirlineData/csvs/'
air = ddf.read_csv(path + '*.csv.bz2',
      compression='bz2', encoding='latin1',
      dtype={'Distance': 'float64', 'CRSElapsedTime': 'float64',
             'TailNum': 'object', 'CancellationCode': 'object',
             'DepDelay': 'float64', 'ActualElapsedTime': 'float64',
             'ArrDelay': 'float64', 'ArrTime': 'float64', 'DepTime': 'float64'})
air.npartitions
```

- Dask 会并行读取多个压缩文件
- 但实际读取在 `.compute()` 调用前不会发生 **(lazy evaluation)**

3.2.2 示例：基本计算

Python

```python
max_delay = air.DepDelay.max().compute()
mean_delay = air.DepDelay.mean().compute()
```

- 计算中位数 (median) 会很慢甚至不可行，因为它涉及大量数据移动 (shuffle)

3.2.3 示例：分组聚合 (Split-Apply-Combine)

Python

```python
sub = air[(air.UniqueCarrier == 'UA') & (air.Origin == 'SFO')]
byDest = sub.groupby('Dest').DepDelay.mean()
results = byDest.compute()
```

3.3 Dask Bag (类 Python 列表)

- **结构**：类似 Python 的 list，但无固定顺序
- **默认调度器**：processes
- 适合执行类似并行 map 的操作

3.3.1 示例：Wikipedia 日志数据

Python

```python
import dask.multiprocessing
dask.config.set(scheduler='processes', num_workers=4)
import dask.bag as db

path = '/scratch/users/paciorek/wikistats/dated_2017_small/dated/'
wiki = db.read_text(path + 'part-0*gz')
n = wiki.count().compute()
```

3.3.2 示例：过滤操作

Python

```python
import re

def find(line, regex='Armenia'):
    vals = line.split(' ')
    if len(vals) < 6:
        return False
    tmp = re.search(regex, vals[3])
    return tmp is not None

armenia = wiki.filter(find)
result = armenia.count().compute()
```

⚠ Remark

应避免多次 `.compute()` 调用，应将所有操作构建完毕后统一执行

3.3.3 示例：转换为 DataFrame

Python

```python
def make_tuple(line):
    return tuple(line.split(' '))

dtypes = {'date': 'object', 'time': 'object', 'language': 'object',
          'webpage': 'object', 'hits': 'float64', 'size': 'float64'}

df = armenia.map(make_tuple).to_dataframe(dtypes)
result = df.compute()
```

3.4 Dask Array (类 NumPy 数组)

- 结构：数组被分块 (by rows and columns) 存储为多个 NumPy 子数组
- 默认调度器：threads
- 适用于大规模矩阵计算，但非所有 NumPy 操作都支持

3.4.1 示例：单节点数组操作

Python

```python
import dask
dask.config.set(scheduler='threads', num_workers=4)
import dask.array as da

x = da.random.normal(0, 1, size=(40000, 40000), chunks=(10000, 10000))
mycalc = da.mean(x, axis=1)
rs = mycalc.compute()
```

- 可在单机上高效处理 13 GB 数组
- Dask 通过 lazy evaluation 和 chuck，仅在需要结果时才加载数据

> ⚠ Remark
>
> 对于 mean 这种 row-based operation, 我们可能会考虑只按行进行分块, 即 `x = da.random.normal(0, 1, size=(40000,40000), chunks=(2500, 40000))`, 但实际的运行速度并没有明显差别, 因为均值计算可以分段进行, 并且只有少量的 summary statistics 需要在 workers 之间 move

3.4.2 示例：更大数组 (80 GB)

Python

```python
x = da.random.normal(0, 1, size=(100000,100000), chunks=(10000,10000))
mycalc = da.mean(x, axis=1)
rs = mycalc.compute()
```

> ⚠ Remark
>
> 尽管数组理论上占用 80 GB，实际内存占用仅数 GB，因为 Dask 动态生成块而非一次性加载全部

3.5 各个 Dask 数据类型在 compute 过后会转换成什么？

Dask 类型	构造时是什么	`.compute()` 之后变成什么?	说明
`dask.array.Array`	分块存储的多个 NumPy 子数组（laz…	NumPy ndarray	Dask Array 就是“chunked NumPy array”，最终必须合并为一个 NumPy 数组
`dask.dataframe.DataFrame`	分区的多个 Pandas DataFrame（lazy）	Pandas DataFrame	所有 partitions concat 成一个完整的 pandas.DataFrame
`dask.dataframe.Series`	多个 pandas.Series partition	Pandas Series	同理 concat
`dask.bag.Bag`	多个 Python 对象组成的 bag（lazy list）	Python list	bag 是 list 的懒惰版本；compute 后返回列表
`dask.delayed`	任意 Python 任务（lazy）	该任务实际返回的 Python 对象	delayed 相当于“lazy 的 Python 函数调用”

Dask 类型	构造时是什么	`.compute()` 之后变成什么?	说明
`client.compute()` future	分布式 future	Python 对象（.result()）	需要 `.result()` 取值

---

4 数据库概述 (Overview)

4.1 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

4.2 内存与磁盘使用 (Memory and Disk Use)

数据库的数据存储在**磁盘**上, 但数据库可通过实现高速访问：

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

4.3 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**
  - **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

---

5 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

5.1 Denormalized 的缺点

不规范设计 (Denormalized) 的缺点：

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)

5.2 主键与外键 (Keys)

- **主键 (Primary Key)**: 每行的唯一标识符
- **外键 (Foreign Key)**: 指向另一张表的主键，用于建立表之间的关系

5.3 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**

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)**
- 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;
```

5.4 示例: Stack Overflow Metadata Example

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

---

6 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

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

6.1 示例：访问 SQLite 数据库

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()
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),
 ...
]
```

6.2 示例：访问 DuckDB 数据库

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()
```

6.3 查看数据库结构 (Inspecting the Database)

6.3.1 查看所有表名

SQLite 示例:

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

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

6.3.2 查看字段名称

Python

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

6.4 基本 SQL 查询 (Basic SQL Queries)

6.4.1 获取表数据

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

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

断开连接：

Python

```
db.close()
```

6.4.2 获取 Pandas DataFrame

Python

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

---

7 Basic SQL for choosing rows and fields from a table

Logic

SQL 属于声明式语言 **(declarative)**：描述目标结果，由系统决定执行方式

7.1 SQL 查询结构

Logic

Data Query Language 是数据查询语言, 用来查询数据库中表的记录; DQL 的语法结构大致如下:

SQL

```SQL
SELECT
    字段列表
FROM
    表名列表
WHERE
    条件列表
GROUP BY
    分组字段列表
HAVING
    分组后条件列表
ORDER BY
    排序字段列表
LIMIT
    分页参数
```

- 基础查询
  - 按字段查询 `SELECT ... FROM ...`
  - 给字段设置别名 `SELECT ... [AS] ... FROM ...`
  - 去除重复记录 `SELECT DISTINCT ... FROM ...`
- 条件查询
  - 条件查询 `SELECT ... FROM ... WHERE`
  - 比较运算符列表
  - 条件运算符列表
- 聚合查询
  - 聚合查询 `SELECT FUN(...) FROM ...`
  - 聚合函数列表
- 分组查询 `SELECT ... FROM ... GROUP BY ... [HAVING ...]`
- 排序查询 `SELECT ... FROM ... ORDER BY ...`
- 分页查询 `SELECT ... FROM ... LIMIT ...`
- 执行顺序 & 例子

常用关键字:

关键词	功能
SELECT	选择列
FROM	指定表
WHERE	过滤行
ORDER BY	排序结果

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

常用的**比较运算符**如下:

比较运算符	功能
`>`	大于
`>=`	大于等于
`<`	小于
`<=`	小于等于
`=`	等于
`<>` 或 `!=`	不等于
`BETWEEN ... AND ...`	在某个范围内 (包含端点)
`IN(...)`	属于 `IN` 之后的列表中的某一个值
`LIKE ...`	模糊匹配 (`_` 匹配单个字符, `%` 匹配任意个字符)
`IS NULL`	是 `NULL`

常用的**逻辑运算符**如下:

逻辑运算符	功能
`AND` 或 `&&`	并且
`OR` 或 ||	或者
`NOT` 或 `!`	非

SQL 查询的**逻辑处理顺序**如下:

1. `FROM` 子句 (包括 `JOIN`)
   首先确定数据来源, 加载表并处理表连接
2. `WHERE` 子句
   对基础数据进行行级过滤
3. `GROUP BY` 子句
   将过滤后的数据按照指定列分组
4. `聚合函数` (如 `SUM`, `COUNT`, `AVG` 等)
   对每个分组计算聚合值
5. `HAVING` 子句
   对分组后的结果进行过滤
6. `SELECT` 子句
   选择最终显示的列 (包括计算列)
7. `ORDER BY` 子句
   对最终结果排序
8. `LIMIT` 子句
   限制返回的行数

7.1.1 示例：高浏览量问题

Python

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

7.1.2 示例: 前 10 名浏览量问题

Python

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

7.2 分组与分层 (Grouping / Stratifying)

7.2.1 语法结构

SQL

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

7.2.2 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`

⚠ Remark

统计员工总数

SQL

```sql
SELECT COUNT(*) FROM emp;    # 16
SELECT COUNT(idcard) FROM emp;    # 15（有一个 NULL）
SELECT COUNT(1) FROM emp;    # 16
```

三者的区别在于

- `COUNT(*)`：统计时包括 `NULL` 行，且在大多数数据库中经过优化，执行效率较高
- `COUNT(列名)`：统计时不包括 `NULL` 行，且执行效率较低
- `COUNT(1)`：统计时包括 `NULL` 行，且在大多数数据库中经过优化，执行效率较高

7.2.3 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;
```

Remark

- `WHERE` 和 `HAVING` 的区别:
  - 执行时机不同: `WHERE` 是在分组前进行筛选, `HAVING` 是在分组后对结果进行筛选
  - 判断条件不同: `WHERE` 不能对聚合函数进行判定, `HAVING` 可以对聚合函数进行判定
- 执行顺序: `WHERE` > `GROUP BY` > 聚合函数 > `HAVING`

7.3 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,)]
```

7.3.1 何时需要 DISTINCT

1. 结果中可能出现重复行，且你希望去重

通常出现在 join 或数据本身存在重复。

Example: 去重后的课程列表

```sql
SELECT DISTINCT course
FROM takes;
```

如果不加 DISTINCT，每个学生选同一门课都会重复出现。

2. join 导致重复，而你只要唯一值

典型为一对多、多对多 join。

Example: 查所有选过课的唯一学生 ID

```sql
SELECT DISTINCT s.id
FROM student s
JOIN takes t ON s.id = t.id;
```

不加 DISTINCT 会因为学生有多门课而重复出现。

3. 需要 unique count

如统计不重复的个体数量。

Example: 选课学生人数

```
SELECT COUNT(DISTINCT id)
FROM takes;
```

如果不加 DISTINCT，会统计成“选课记录数”。

7.3.2 何时不需要 DISTINCT

1. 查询主键（primary key），天然不重复

主键列本身具有唯一性。

Example: student 表中的 id

```
SELECT id, name
FROM student;
```

2. GROUP BY 已经保证每个分组唯一

GROUP BY 的输出中，每组只出现一次，因此不需要 DISTINCT

Example

```
SELECT major, COUNT(*)
FROM student
GROUP BY major;
```

这里的 major 已经按组唯一

3. 聚合函数返回单行

例如统计总数、总和，不会产生重复。

Example

```
SELECT COUNT(*)
FROM employee;
```

4. 使用 set operations（UNION / INTERSECT / EXCEPT/IN）

这三个运算符本身就会自动去重

7.4 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
```

7.4.1 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()
```

7.4.2 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`）

7.4.3 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

7.4.4 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

7.5 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
SELECT * FROM questionsAugment WHERE viewcount > 1000 LIMIT 3;
```

7.5.1 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
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`

7.5.2 Challenge 4

查询从未发布过问题的用户名称

```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

7.6 索引 (Indexes)

索引是一种加速查询的机制，用于快速查找行：

```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

7.7 集合操作 (Set Operations)

7.7.1 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;
```

7.7.2 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

7.8 子查询 (Subqueries)

7.8.1 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

7.8.2 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。

7.9 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

8 高性能数据存储工具 (Recent Tools and Data Formats)

8.1 数据湖 (Data Lake)

- 大型数据往往不再存储于单一数据库中，而是分散为多个文件，常使用：
  - **Parquet**（面向列的高效存储格式）
  - **CSV**（传统文本格式）
- 当这些文件存储于云端（如 AWS S3、Google Cloud Storage），整体结构被称为 **数据湖**。

8.2 Apache Arrow

8.2.1 核心特点

- 提供高效的内存数据结构，用于跨语言、跨系统的数据分析。
- 在 Python 中可通过 **PyArrow** 包使用（R 中为 **arrow** 包）。
- 采用 **列式存储**：同一列的值顺序存储，可实现 **O(1)** 时间复杂度的单值访问。

8.2.2 优势

- 能够从多种文件格式读取数据（如 Parquet、Arrow 原生格式、文本文件）。
- 仅在需要时才从磁盘读取数据，避免将整个数据集加载至内存。
- 通过按需读取与内存映射（memory mapping）显著减少 I/O 开销。

8.2.3 与其他格式的比较

- 原生 Arrow 格式在按列访问与懒加载方面性能最佳。
- Parquet 在磁盘压缩与云端共享方面更具优势。

8.3 Polars

8.3.1 特点

- 一款 **内存中超高速 DataFrame 库**，是 Pandas 的替代方案。
- 基于 **Apache Arrow** 的列式数据结构。
- 支持 **惰性执行 (lazy execution)** 模型，类似 Spark 或 Dask，可自动优化查询计划。
- 常用于需要高效分组、聚合与 join 操作的分析场景。

---

9 稀疏性 (Sparsity)

9.1 概念

- 在统计与机器学习中，许多矩阵包含大量零元素。
- 稀疏矩阵 (Sparse Matrix) 仅存储非零元素，从而减少存储空间与计算时间。

9.2 软件支持

- Python 中常用模块：
  - `scipy.sparse`
  - `sklearn.utils.sparsefuncs`
- 稀疏矩阵的存储方式包括：
  - COO（坐标形式）
  - CSR（压缩行格式）
  - CSC（压缩列格式）

---

10 利用统计思想解决计算瓶颈 (Statistical Tools for Computational Bottlenecks)

10.1 采样思想 (Sampling for Computation)

- **传统动机**：无法收集全体数据。
- **大数据场景**：无法处理全部数据，可对数据进行采样分析。
- 使用统计不确定性估计 (e.g., 置信区间) 评估采样误差。
- 若结果精度不够，可通过增大样本量改进。

10.2 分布式自助法 (Distributed Bootstrap)

- 方法：
  1. 从大数据中抽取多个较小的自助样本 (bootstrap samples)。
  2. 在每个样本上独立计算估计量。
  3. 综合结果以获得近似全数据估计。
- 优点：计算可完全并行化，内存占用显著减少。

10.3 随机化算法 (Randomized Algorithms)

- 基于随机选择的计算加速策略。
- 应用示例：
  - **随机梯度下降 (SGD)**：优化时仅使用部分样本。
  - **随机线性代数**：在回归中随机选择设计矩阵的部分行。
  - **统计杠杆值 (Leverage Scores)**：用以加权采样高影响点。
- 优点：
  - 降低计算复杂度。
  - 提高算法可扩展性与鲁棒性。