

STAT243 Lecture 7 Databases

1 关于“大数据”的思考 (An Editorial on “Big Data”)

• 热度的变化

- “大数据 (big data)”曾经是热门词汇，但随着 AI/ML 革命的到来，它不再是主要的流行语。
- 然而，AI/ML 的发展本身正是建立在**大规模可获取数据集**的基础上。

• 理性看待“大数据”

- 部分“大数据”热潮是合理的，因为：
 - 大规模数据能支持更复杂、更非线性的模型
 - 能回答传统小样本无法探索的问题

- 但“大数据”并不能解决以下根本问题：

- **相关不代表因果 (correlation ≠ causation)**

例如：即使拥有全美人口的健康数据，也无法仅凭观测得出“盐摄入导致高血压”

- **非随机样本 ≠ 统计样本**

非代表性样本无法推断总体特征

如 COVID-19 数据虽多，但因不完整/不均衡而难以得出可靠结论

- **多重检验 (multiple testing)**

在庞大数据上执行大量分析会提高“伪显著”概率，纯属噪声的结果可能看起来“有意义”

• 结论

- 一个**小而精的代表性数据集**往往比一个庞大但杂乱的样本更有信息价值
- 大数据仍有价值：可用于筛选、构造更合理的样本，或支持因果推断设计
- 但必须搭配严谨的统计思考与实验设计使用

• “大”的不同定义

- **技术定义**：指数据体量或采集速度极大
- **社会定义**：指数据与实证分析在社会中的普遍应用，而非数据量本身

延伸阅读：

文章讨论电子健康记录 (EHR) 分析中的偏差问题

- <https://doi.org/10.1093/jrsssa/qnae039>
- <https://doi.org/10.1093/jrsssa/qnae005>

2 数据规模与运算限制 (Logistics and Data Size)

• 内存瓶颈

- Python 与 R 的主要限制：所有对象都存储在内存中
- 因此单机处理的数据集通常受限于 **1–20 GB**，具体取决于系统内存

• 适用范围

- 本单元介绍的技术主要针对：
 - **GB 到数十 GB** 的数据规模
 - 若机器拥有足够内存或磁盘，也可扩展至更大规模
- 若数据规模达到：
 - **数百 GB – TB 级别** → 需使用数据库或云平台
 - **PB 级别** → 建议采用 Spark、AWS、GCP 等大数据框架

• 数据存储建议

- 将大文件存放于**本地磁盘**而非网络驱动器
- 避免网络传输带来的延迟与带宽瓶颈

| 3 我们已掌握的“大数据”处理基础 (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

| 4 Overview

| 4.1 背景与发展 (Background)

- **传统高性能计算 (HPC)**

传统 HPC 关注基于消息传递的并行计算技术，如 **MPI (Message Passing Interface)**，以及如何通过优化算法提高效率。

- **现代趋势**

在过去二十年中，计算重点转向了**大规模分布式数据集**的处理：

数据分布在多个机器上，但用户可像操作单个数据集一样进行分析。

- **常见工具**

- **Spark**：在 2010 年代广受欢迎
- **Dask**：Python 原生的分布式计算框架 (本单元主要讲解)

| 4.2 MapReduce 范式 (The MapReduce Paradigm)

- **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

- **类比范式**

Pandas 与 R 的 **dplyr** 中的 **split-apply-combine** 策略与 MapReduce 思想一致。

4.3 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。

4.4 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)

4.5 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

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

5.1 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

5.2 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

5.3 Dask DataFrame (类 Pandas 对象)

- **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

5.3.1 示例：读取航班延误数据 (约 11 GB)


Python

```

1 import dask
2 dask.config.set(scheduler='threads', num_workers=4)
3 import dask.dataframe as ddf
4
5 path = '/scratch/users/paciorek/243/AirlineData/csvs/'
6 air = ddf.read_csv(path + '*.csv.bz2',
7                     compression='bz2', encoding='latin1',
8                     dtype={'Distance': 'float64', 'CRSElapsedTime': 'float64',
9                            'TailNum': 'object', 'CancellationCode': 'object',
10                           'DepDelay': 'float64', 'ActualElapsedTime': 'float64',
11                           'ArrDelay': 'float64', 'ArrTime': 'float64', 'DepTime': 'float64'})
12 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**)

5.3.2 示例：基本计算


Python

```

1 max_delay = air.DepDelay.max().compute()
2 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)

5.3.3 示例：分组聚合 (Split-Apply-Combine)


Python

```

1 sub = air[(air.UniqueCarrier == 'UA') & (air.Origin == 'SFO')]
2 byDest = sub.groupby('Dest').DepDelay.mean()

```

```
3 results = byDest.compute()
```

⚠ 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

5.4 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

5.4.1 示例: Wikipedia 日志数据

```
Python

1 import dask.multiprocessing
2 dask.config.set(scheduler='processes', num_workers=4)
3 import dask.bag as db
4
5 path = '/scratch/users/paciorek/wikistats/dated_2017_small/dated/'
6 wiki = db.read_text(path + 'part-0*gz')
7 n = wiki.count().compute()
```

- 计算记录数约需 136 秒 (完整数据集)

5.4.2 示例: 过滤操作

```
Python

1 import re
2
3 def find(line, regex='Armenia'):
4     vals = line.split(' ')
5     if len(vals) < 6:
6         return False
7     tmp = re.search(regex, vals[3])
8     return tmp is not None
9
10 armenia = wiki.filter(find)
11 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

5.4.3 示例: 转换为 DataFrame

```
Python
```

```
1 def make_tuple(line):
2     return tuple(line.split(' '))
3
4 dtypes = {'date': 'object', 'time': 'object', 'language': 'object',
5           'webpage': 'object', 'hits': 'float64', 'size': 'float64'}
6
7 df = armenia.map(make_tuple).to_dataframe(dtypes)
8 result = df.compute()
```

5.5 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

5.5.1 示例: 单节点数组操作

```
Python
1 import dask
2 dask.config.set(scheduler='threads', num_workers=4)
3 import dask.array as da
4
5 x = da.random.normal(0, 1, size=(40000, 40000), chunks=(10000, 10000))
6 mycalc = da.mean(x, axis=1)
7 rs = mycalc.compute()
```

- 可在单机上高效处理 13 GB 数组
- Dask 通过 lazy evaluation 和 chunk, 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

5.5.2 示例: 更大数组 (80 GB)

```
Python
1 x = da.random.normal(0, 1, size=(100000, 100000), chunks=(10000, 10000))
2 mycalc = da.mean(x, axis=1)
3 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

5.6 分布式数组 (Distributed Arrays)

- 使用 Dask Distributed 可将数组分布到多台机器上计算
- 但需避免从单一进程分发数据, 否则会触发大量网络复制

6 概述 (Overview)

6.1 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)

6.2 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

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

7.1 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

7.2 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

8 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

8.1 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)

8.2 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。

8.3 规范化设计 (Normalized)

使用四张表：

8.3.1 Student

ID	name	grade_level

8.3.2 Teacher

ID	name	department	age

8.3.3 Class

ID	topic	class_size	teacher_ID

8.3.4 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

8.4 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

8.5 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：



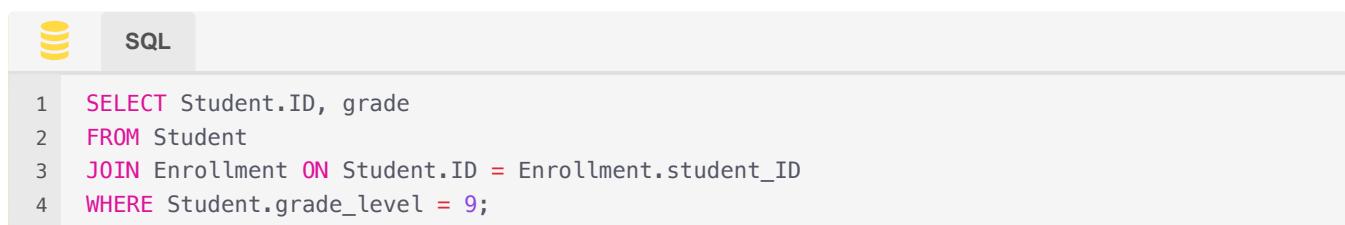
```

1 SELECT Student.ID, grade
2 FROM Student, Enrollment
3 WHERE Student.ID = Enrollment.student_ID
4 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：



```

1 SELECT Student.ID, grade
2 FROM Student
3 JOIN Enrollment ON Student.ID = Enrollment.student_ID
4 WHERE Student.grade_level = 9;

```

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

9.1 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)

9.2 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：

9.2.1 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

9.2.2 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

9.3 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

10 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

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

10.1 示例：访问 SQLite 数据库



Python

```
1 import sqlite3 as sq
2 import os
3
4 dir_path = '/mirror/data/pub/users/paciorek/share'
5 db_filename = 'stackoverflow-2021.db'
6
7 con = sq.connect(os.path.join(dir_path, db_filename))
8 db = con.cursor()
9 db.execute("select * from questions limit 3")
10 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

```
1  [
2    (65534165.0, '2021-01-01 22:15:54', 0.0, 112.0, 2.0, 0.0, None, "Can't update a value in
3     sqlite3", 13189393.0),
4  ...
```

10.2 示例：访问 DuckDB 数据库



Python

```
1 import duckdb as dd
2
3 dir_path = '../data'
4 db_filename = 'stackoverflow-2021.duckdb'
5
6 con = dd.connect(os.path.join(dir_path, db_filename))
7 db = con.cursor()
8 db.execute("select * from questions limit 5")
9 db.fetchall()
```

10.3 查看数据库结构 (Inspecting the Database)

10.3.1 查看所有表名



Python

```
1 def db_list_tables(db):
2     db.execute("SELECT name FROM sqlite_master WHERE type='table';")
3     return [table[0] for table in db.fetchall()]
4
5 db_list_tables(db)
6 # ['questions', 'answers', 'questions_tags', 'users']
```

DuckDB 示例：



Python

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

10.3.2 查看字段名称



Python

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

10.4 基本 SQL 查询 (Basic SQL Queries)

10.4.1 获取表数据



Python

```
1 results = db.execute("select * from questions limit 5").fetchall()
```

```
2 type(results)      # <class 'list'>
3 type(results[0])   # <class 'tuple'>
```

或分步执行：

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

断开连接：

```
Python
1 db.close()
```

10.4.2 获取 Pandas DataFrame

```
Python
1 import pandas as pd
2 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。

11 Basic SQL for choosing rows and fields from a table

11.1 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

11.2 示例：Stack Overflow 查询案例

11.2.1 高浏览量问题

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

11.2.2 前 10 名浏览量问题

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

11.3 SQL 查询结构



SQL

```
1 SELECT <columns>
2 FROM <table>
3 WHERE <conditions>
4 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。

11.4 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

```
1 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

11.5 分组与分层 (Grouping / Stratifying)

11.5.1 语法结构



SQL

```
1 SELECT <aggregate>(<column>)
2 FROM <table>
3 GROUP BY <column>
```

11.5.2 示例：统计标签出现次数



Python

```
1 db.execute("""
2     select tag, count(*) as n
3 """)
```

```
3 from questions_tags
4 group by tag
5 order by n desc
6 limit 25
7 """.fetchall()
```

返回：

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

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

11.5.3 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

```
1 SELECT tag, COUNT(*) as n
2 FROM questions_tags
3 GROUP BY tag
4 HAVING n > 10000;
```

11.6 去重 (DISTINCT)

`DISTINCT` 可去除重复行或重复值。



Python

```
1 # 获取所有唯一标签
2 tag_names = db.execute("select distinct tag from questions_tags").fetchall()
3
4 # 统计唯一标签数量
5 db.execute("select count(distinct tag) from questions_tags").fetchall()
```

输出：

```
1 [('sorting',), ('visual-c++',), ('mfc',), ('cgridctrl',), ('css',)]
2 [(42137,)]
```

11.7 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

```
1 SELECT <columns>
2 FROM <table1>
3 JOIN <table2>
4 ON <table1.<column> = <table2.<column>
5 WHERE <conditions>
6 ORDER BY <columns>
```

等价的简化写法：



SQL

```
1 SELECT *
2 FROM table1, table2
3 WHERE table1.id = table2.id
```

11.7.1 示例：查询带有 "python" 标签的问题



Python

```
1 result1 = db.execute("""
2     SELECT *
3     FROM questions
4     JOIN questions_tags
5     ON questions.questionid = questions_tags.questionid
6     WHERE tag = 'python'
7     """).fetchall()
```

等价形式（不使用 JOIN）：



Python

```
1 result2 = db.execute("""
2     SELECT *
3     FROM questions, questions_tags
4     WHERE questions.questionid = questions_tags.questionid
5     AND tag = 'python'
6     """).fetchall()
```

11.7.2 示例：三表连接 (Three-Way Join)



Python

```
1 result = db.execute("""
2     SELECT *
3     FROM questions Q
4     JOIN questions_tags T ON Q.questionid = T.questionid
5     JOIN users U ON Q.ownerid = U.userid
6     WHERE tag = 'python' AND viewcount > 1000
7     """).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`)

11.7.3 Challenge 1

返回所有带有 Python 标签的问题的答案



SQL

```
1 SELECT A.*
```

```
2   FROM answers A
3   JOIN questions Q ON A.questionid = Q.questionid
4   JOIN questions_tags T ON Q.questionid = T.questionid
5 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

11.7.4 Challenge 2

返回所有回答过 Python 标签问题的用户

SQL

```
1   SELECT DISTINCT U.displayname
2   FROM users U
3   JOIN answers A ON U.userid = A.ownerid
4   JOIN questions Q ON A.questionid = Q.questionid
5   JOIN questions_tags T ON Q.questionid = T.questionid
6 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

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

创建视图以便重复使用：

SQL

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

视图可以像表一样被查询：

SQL

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

11.8.1 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

```
1 CREATE VIEW question_tag_view AS
2 SELECT Q.questionid, T.tag
3 FROM questions Q
4 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`

11.8.2 Challenge 4

查询从未发布过问题的用户名称



SQL

```
1 SELECT U.displayname
2 FROM users U
3 LEFT JOIN questions Q ON U.userid = Q.ownerid
4 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

11.9 索引 (Indexes)

索引是一种加速查询的机制, 用于快速查找行:



SQL

```
1 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

11.10 集合操作 (Set Operations)

11.10.1 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

```
1 SELECT displayname, userid
2 FROM questions Q JOIN users U ON U.userid = Q.ownerid
3 INTERSECT
4 SELECT displayname, userid
5 FROM answers A JOIN users U ON U.userid = A.ownerid;
```

11.10.2 Challenge 5

查询既没有提问也没有回答过问题的用户



SQL

```
1 SELECT displayname, userid
2 FROM users
3 WHERE userid NOT IN (
4     SELECT ownerid FROM questions
5     UNION
6     SELECT ownerid FROM answers
7 );
```

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

11.11 子查询 (Subqueries)

11.11.1 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

```
1 SELECT *
2 FROM questions
3 JOIN answers A ON questions.questionid = A.questionid
4 JOIN (
5     SELECT ownerid, COUNT(*) AS n_answered
6     FROM answers
7     GROUP BY ownerid
8     ORDER BY n_answered DESC
9     LIMIT 1000
10 ) most_responsive
11 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

11.11.2 WHERE 子句中的子查询



SQL

```
1 SELECT AVG(upvotes)
2 FROM users
3 WHERE userid IN (
4     SELECT DISTINCT ownerid
5     FROM questions
6     JOIN questions_tags ON questions.questionid = questions_tags.questionid
7     WHERE tag = 'python'
8 );
```

作用：

计算所有曾提问 Python 标签问题的用户的平均 upvote 数。

11.12 创建数据库表 (Creating Tables)

使用 Pandas 在 Python 中创建表：



Python

```
1 con = sq.connect(db_path)
2 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

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

- 近年来，出现了大量新的文件格式与工具，用于高效处理超大规模数据集。
- 相比传统数据库系统，这些工具常以 **速度与并行处理性能** 为优化目标。

12.1 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**。

12.2 Apache Arrow

12.2.1 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。

12.2.2 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。

12.2.3 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。

12.3 Polars

12.3.1 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。

13 稀疏性 (Sparsity)

13.1 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。

13.2 常见应用

- **图与网络分析**: 邻接矩阵或条件依赖矩阵
- **时间序列与空间模型**: 自回归 (AR) 或空间邻接矩阵
- **高维回归模型**: LASSO 等方法产生稀疏参数估计
- **协方差选择模型**: 稀疏精度矩阵估计

13.3 软件支持

- Python、R、MATLAB 均提供稀疏矩阵的专用数据结构与计算函数。
- 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)

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

14.1 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。

14.2 分布式自助法 (Distributed Bootstrap)

- 提出者：Michael Jordan 等人 (Berkeley Statistics/EECS)。
- 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。
- 参考：Kleiner et al., *Journal of the Royal Statistical Society*, 2014.

14.3 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。
- 参考文献：相关算法综述可见于 *arXiv: randomized numerical linear algebra* 系列论文。

14.4 小结

- **Apache Arrow** 与 **Polars** 是现代高性能数据处理的重要基础。
- **稀疏矩阵结构** 能有效减少大规模计算的空间与时间复杂度。
- **统计抽样与随机化算法** 提供应对计算瓶颈的理论框架。
- 结合这些思想，可在不牺牲统计精度的前提下显著提升大数据分析的可行性和效率。