

第 1 章 DuckDB 是什么

"DuckDB 是为分析而生的 SQLite。" —— DuckDB 官方对自身最简洁的总结

本章回答一个问题：在你已经有了 PostgreSQL、MySQL、SQLite、pandas、Spark 的世界里，为什么还需要 DuckDB？

- 要回答它，我们需要先理解 DuckDB 的三个身份：
- 1. 它是一个**嵌入式（进程内）数据库**；
 - 2. 它是一个**列式 OLAP（在线分析处理）引擎**；
 - 3. 它是一个**无需 " 入库 " 就能直接查询文件的数据处理工具**。

三个身份叠加起来，产生了一种此前很少见的组合：**SQLite 一样简单，Spark 一样快的数据处理能力，而且是单机。**

1.1 三分钟理解 DuckDB 的本质

1.1.1 身份一：嵌入式数据库

传统数据库（MySQL、PostgreSQL、Oracle）都是**客户端/服务器架构**：



而 DuckDB 是**进程内（in-process）**的，就像 SQLite：



带来的直接后果：

特性	传统数据库	DuckDB
安装	装服务、建用户、开端口、配权限	<code>pip install duckdb</code>
延迟	一次查询含网络往返（0.1 ~ 10 ms 起）	无网络，纯函数调用
数据传输	结果序列化后跨网络发送	零拷贝 ：直接给出 Arrow/Pandas 对象指针

特性	传统数据库	DuckDB
部署形态	服务器集群	一个库、一个命令行工具、一个 WASM 文件
运维成本	需要 DBA	基本为零

重要提醒：这里的"零拷贝"是关键。当你用 pandas 读取 DuckDB 的结果时，数据不是"SELECT 出来再转成 DataFrame"，而是**内存布局本来就是 Arrow 格式**，转换几乎不需要额外代价。这是 DuckDB 和"用 Python 驱动连 MySQL"的本质差别。

1.1.2 身份二：列式 OLAP 引擎

数据库引擎按存储与处理方式，粗略分为两类：

行式 (OLTP 导向)： 适合"查一条订单的所有字段"。

id	name	age	city	amount
1	Alice	30	Beijing	100
2	Bob	25	Shanghai	50

列式 (OLAP 导向)： 适合"算所有订单的 amount 之和"。

amount 列
100
50
78
210

后者有三个巨大优势：

1. **读得少：** 算 `SUM(amount)` 不需要读入 name/age/city 这三个列的字节；
2. **压缩率高：** 同一列的数据类型一致、取值相似度高，压缩比常常达到 5 ~ 20 倍（行式通常只有 2 ~ 5 倍）；
3. **能被向量化：** 同一列的连续数据可以用 SIMD 指令一条 CPU 指令处理多个值。

DuckDB 的整个引擎就是围绕这三点设计的，详见第 16 章。

1.1.3 身份三：直接查询"文件"的数据工具

这是最容易被低估、但最能改变你日常工作的一点：**DuckDB 可以不需要先把数据 " 装载 " 进来。**

```

1  -- 不需要 CREATE TABLE，也不需要 INSERT/COPY
2  SELECT city, sum(amount) AS total
3  FROM 'sales_2026.parquet'      -- ← 直接把文件名当表名
4  GROUP BY city
5  ORDER BY total DESC;

```

同理，CSV、JSON、Excel、Hive 分区目录、云上的 Parquet、甚至 PostgreSQL 里的一张表，都可以直接“放在 FROM 后面”。这让 DuckDB 同时具备了两顶帽子：

- 一个 **SQL 数据库**（有持久存储、有事务）；
- 一个 **文件处理器**（像 `jq`、`csvkit`、`pandas` 那样的瑞士军刀）。

1.2 DuckDB 的设计哲学

DuckDB 团队（来自荷兰 CWI 数据库研究组，也是 MonetDB/Vectorwise 的传承者）明确提出了几条设计原则，理解它们有助于理解为什么 DuckDB 的 API 长这样：

原则一：简单优先 (Simplicity)

- 一个二进制文件，无外部依赖；
- 没有事务日志同步、没有后台进程、没有连接池；
- 默认参数就能跑得很好（绝大多数用户不需要调任何参数）。

原则二：可移植 (Portability)

- 支持 Linux、macOS、Windows（含 ARM64）；
- 支持 x86_64 和 ARM64 两种指令集；
- 编译到 WebAssembly 可以在浏览器里跑（DuckDB-WASM）；
- 有 Java、Python、Node.js、Go、Rust、C/C++、ODBC、JDBC、ADBC 等客户端。

原则三：面向分析 (Analytics First)

- 列存 + 向量化 + 并行；
- 支持超出内存的数据量（会 spill 到磁盘，见第 17 章）；
- 完整的 SQL 标准：窗口函数、CTE、嵌套数据类型、近似聚合。

原则四：免费且开放源码

- 许可证是 **MIT**（核心）+ 部分组件为 ICU/Apache 等宽松许可证。
- 这意味着可以商业闭源集成，没有 AGPL 的传染性担忧（这是它能被广泛集成进商业产品的关键）。

原则五：正确性不可妥协

- 有一个 `sqlsmith` 扩展专门用于随机生成 SQL 做模糊测试；
- 每个 PR 都要求通过大量回归测试。

1.3 与其他技术的对比

1.3.1 DuckDB vs SQLite

维度	SQLite	DuckDB
存储模型	行式 B-Tree	列式 + 行组 (Row Group)
擅长负载	OLTP：点查、小型事务	OLAP：全表扫描、聚合
聚合 1 亿行	数十秒 ~ 分钟级	通常 < 1 秒
主键点查一行	极快 (微秒级)	较慢 (需扫列组)
并发写	单写多读	单进程写，多进程读 (详见第 18 章)
生态位	手机 App、嵌入式设备、小配置存储	数据分析、ETL、数据科学
二者关系	—	互为补充，甚至可用扩展直接读 SQLite 文件

一句话：如果你要做一个记账 App 存用户数据，用 SQLite；如果你要分析几百万条日志，用 DuckDB。如果要同时做两件事，用 DuckDB + `sqlite` 扩展挂载 SQLite 文件 (第 14 章)。

1.3.2 DuckDB vs PostgreSQL / MySQL

维度	PostgreSQL	DuckDB
架构	C/S，需要服务器	进程内库
适用场景	生产业务库 (OLTP为主，兼顾 OLAP)	分析、ETL、单机数仓
事务与并发写入	完整 MVCC，多写者	单写进程，快照隔离
用户/权限管理	完整 (角色、授权、行级安全)	基本没有 (不适合多租户服务)
大数据量分析 (GB 级)	需要调优，仍受行式 IO 限制	快一个数量级

维度	PostgreSQL	DuckDB
列式后压缩	需要插件 (cstore_fdw 等)	原生

实际 workflow 建议：很多团队的真实用法是"PostgreSQL 做业务库，DuckDB 做分析"。用 DuckDB 的 postgres 扩展定时把业务表拉到 DuckDB 里做宽表聚合，再把结果写回 PostgreSQL (第 14 章有完整例子)。

1.3.3 DuckDB vs pandas / Polars

这是很多人最熟悉的比较维度。

维度	pandas	Polars	DuckDB
语言	Python API	Python API (Rust 内核)	SQL (+ Python API)
执行模型	逐行 / 批式解释执行	向量化 + 并行	向量化 + 并行 + 查询优化器
超出内存数据	直接 OOM	部分支持流式	完整支持 spill 到磁盘
惰性求值	需手动优化	LazyFrame 有优化器	天生就是 (优化器最强)
多表连接	手动 merge	手动 join	优化器自动选连接顺序与算法
学习成本	Python 程序员 0	类似 pandas	需要 SQL 基础
组合方式	—	—	三者可无缝互操作 (第 15 章)

值得强调的是：DuckDB 与 pandas/Polars **并非替代关系**。最常见的高生产力模式是：

```

1 import duckdb
2 # 1. 用 SQL 做重活 (读取、过滤、聚合、连接、写出 Parquet)
3 duckdb.sql("""
4     COPY (
5         SELECT dept, avg(salary) AS avg_sal
6         FROM read_parquet('employees/**/*.parquet')
7         GROUP BY dept
8     ) TO 'dept_salary.parquet'
9 """)
10 # 2. 把结果拉回 pandas 做最后的统计/画图
11 df = duckdb.sql("SELECT * FROM 'dept_salary.parquet']").df()
12 df.plot.bar(x="dept", y="avg_sal")

```

把"数据库擅长的事交给数据库"，是 DuckDB 带来的最大效率提升。

1.3.4 DuckDB vs Spark

这是"用牛刀还是用菜刀"的问题。

数据量	推荐方案
< 100 MB	pandas / polars / DuckDB 皆可
100 MB ~ 50 GB	DuckDB (单机) 通常更快、更省事
50 GB ~ 500 GB	DuckDB 配合 spill 到快速 SSD 依然可行
> 1 TB 且需要集群	Spark / ClickHouse / Trino 等分布式方案

业界有一个被广泛引用的经验法则：**Spark 的启动开销 (JVM 启动、任务调度) 大约在 5 ~ 30 秒量级，这段时间内 DuckDB 通常已经完成了 TB 级以内的大部分分析任务。**

当然这是粗略的。真实结论取决于：磁盘 IO 速度、压缩率、是否发生 shuffle、是否跨网络读取数据等。但**如果你的数据能塞进一台机器的磁盘，先试试 DuckDB 是性价比最高的决策。**

诚实的补充：Spark 的价值不只是"算力"，还有分布式容错、YARN/K8s 调度、与数据湖 (Iceberg/Hudi) 的深度集成。在这些方面 DuckDB 并不试图替代它。DuckDB 的优势在**单机的单位成本**。

1.3.5 DuckDB vs ClickHouse

维度	ClickHouse	DuckDB
架构	分布式列式数据库 (有服务)	嵌入式
写入吞吐与实时摄入	强	一般 (不适合高频写入)
部署复杂度	高 (需要运维集群)	零
适合场景	线上报表、实时大屏、PB 级服务	本地分析、一次性 ETL、嵌入式分析

1.4 DuckDB 的适用场景与"不合适"场景

1.4.1 非常适合的场景

1. 本地数据分析 / 数据科学

Jupyter 里告别 `pd.read_csv` 带来的长时间等待与内存爆炸。

2. 一次性/周期性 ETL

把乱七八糟的 CSV/XML/JSON 清洗成规范 Parquet 落到对象存储。

3. 日志文件分析

按天存放的 JSONL/NDJSON 日志，`read_ndjson('logs/2026-*/**/*.ndjson')` 一条语句搞定。

4. 数据科学家的数据准备 (feature engineering)

窗口函数算滑动窗口特征比 pandas `rolling` 好写得多，且能处理超出内存的数据。

5. 嵌入式分析 (Edge Analytics)

把 DuckDB 编进桌面应用、命令行工具、浏览器 (WASM) 里，提供本地 SQL 查询能力。

6. 数据湖/对象存储的查询客户端

配合 https 直接查 S3 上的 Parquet: **列裁剪 + 谓词下推**使其只读取真正需要的字节 (例如 15 列只用到 3 列时，网络传输量直接降到 1/5)。

7. 临时的"数据中转站"

用 DuckDB 读 PostgreSQL, join 一个 CSV, 再写成 Parquet 上传 S3。

1.4.2 不太适合的场景 (请务必读)

诚实地说，DuckDB 不是万能的：

场景	为什么不合适	建议
高并发在线服务 (OLTP)	单进程写限制、无用户权限体系、行级点查慢	用 PostgreSQL / MySQL
每秒几千次的 INSERT/UPDATE	每次写都有事务开销，行组较大时不适合小批量随机写	用 OLTP 数据库或 Kafka 缓冲
多用户共享的 BI 平台	无权限/配额管理	用 ClickHouse / Trino / StarRocks
PB 级、需要分布式容错	DuckDB 单机，无自动并行到多台机器	Spark / Trino
需要强事务性、多写者、行级锁	DuckDB 只有快照隔离与单写约束	PostgreSQL
长期休眠连接 (长事务)	单进程模型	传统数据库

记住：DuckDB 的杀手锏是**"分析型负载 + 单机 + 零运维"**。超出这三个关键词，请考虑其他工具。

1.5 DuckDB 的发展历程 (一个简短的时间线)

了解历史有助于理解它的设计理念：

年份	事件
2018	项目由 Mark Raasveldt 与 Hannes Mühleisen 在荷兰 CWI 启动
2019	首个公开版本 (0.1)，核心目标是"嵌入式分析"
2020	加入 mit-license 开源，Python 生态开始流行

年份	事件
2021	引入 extension 机制；httpfs、parquet 成为标配
2022	0.5 ~ 0.7, Storage 格式开始稳定，官方网站上线
2023	0.8 起 ：官方 MO 提出"数据科学界的 Python 包"，生态爆发；1.0 的 API 冻结
2024	DuckDB 1.0 发布 （代号 GoldDB），宣布稳定文件格式向后兼容
2025	1.2/1.3 版本引入大量分析能力、UI 扩展、Iceberg/Delta 深度支持
2026	1.5.0 (Variegata) ：全新 CLI、原生 <code>VARIANT</code> 类型、更强的 JSON 处理能力

DuckDB 1.0 的意义在于：**磁盘文件格式保证向后兼容**，从此可以放心把数据存在 DuckDB 文件里长期使用（这在第 18 章会详细讨论）。

1.6 一个"看见差别"的例子

在正式开始安装之前，先用一组伪感受让你理解 DuckDB 到底改变了什么。

假设有一个 3000 万行的销售 CSV（约 2.4 GB）：

传统 pandas 写法：

```

1 import pandas as pd
2 df = pd.read_csv("sales.csv")           # 等待 ~40 秒，内存占用峰值
  ~10 GB
3 df["date"] = pd.to_datetime(df["date"]) # 又拷贝一次
4 df = df[df["region"] == "APAC"]         # 又一次全表拷贝
5 result = df.groupby("city")["amount"].sum() # 又要等
6 # 最终：~2 分钟，可能已经 Swap/OMM

```

DuckDB 写法：

```

1 SELECT city, sum(amount) AS total
2 FROM 'sales.csv'
3 WHERE region = 'APAC'
4 GROUP BY city
5 ORDER BY total DESC;

```

差异不在于"代码少了"（虽然确实少了很多），而在于：

1. **读取时不需要把所有列都物化**（列裁剪，只读 city/region/amount/date）；
2. **过滤是流式进行的**，不需要"先全表再筛"（谓词下推）；
3. **内存不够会自动落盘**（spill to disk）；
4. **自动多线程并行**（默认使用所有 CPU 核）；
5. **结果可以直接是 Arrow**，交给 pandas 时几乎不拷贝。

这就是本章开头那句话的真正含义。

实例演练

本章的 3 个实例全部基于示例数据（先执行 `examples/00-seed.sql`），以下输出均为 DuckDB 1.5.5 实测结果。

实例 1-1 · 一条 SQL 顶掉一整段脚本

传统做法要：读文件 → 建表/建 DataFrame → 过滤 → 分组 → 排序。DuckDB 里只要一句：

```
1 SELECT region, round(sum(amount)) AS gmv
2 FROM read_parquet('sales.parquet')      -- 直接读文件，不需要建表
3 WHERE status = '已完成'
4 GROUP BY region
5 ORDER BY gmv DESC;
```

region	gmv
华中	5721367
西南	5719149
华北	5717483
华南	5700959
东北	5651074
华东	5631328

耗时 **0.012 秒**（20 万行）。

实例 1-2 · Parquet 到底比 CSV 快多少

```
1 ls -lh sales.csv sales.parquet
```

```
1 17M sales.csv
2 2.1M sales.parquet    ← 体积只有 1/8
```

```
1 .timer on
2 SELECT sum(amount) FROM read_parquet('sales.parquet');  -- real
0.009s
3 SELECT sum(amount) FROM read_csv_auto('sales.csv');      -- real
0.118s
```

格式	文件体积	同一查询耗时
CSV	17 MB	0.118 s
Parquet	2.1 MB	0.009 s (快约 13 倍)

结论：**反复查询的数据一定要转 Parquet**（第 13.6 节会讲怎么转）。这也是第 1 章那句"列裁剪 + 压缩 + 谓词下推"最直观的证明。

实例 1-3 · 不建表，直接把文件当表查

```
1 SELECT count(*) AS rows, round(sum(amount)) AS gmv
2 FROM read_csv_auto('sales.csv')
3 WHERE region = '华东';
```

rows	gmv
32980	16847718.0

没有 `CREATE TABLE`，没有 `INSERT`，没有"导入"步骤——**文件本身就是表**。

一句话总结本章：DuckDB 让你把"读数据"和"查数据"合并成一个动作。

1.7 本章小结

- DuckDB = **进程内的列式 OLAP 数据库**，目标是成为"分析领域的 SQLite"。
- 三大核心优势：**零运维部署 + 列式向量化性能 + 直接查询文件**。
- 它**不替代** PostgreSQL (OLTP)、Spark (PB 级分布式)、ClickHouse (高并发在线分析)，而是填补了"单机 GB 级分析"的巨大空隙。
- 最好的使用心态：把它当成**一个能处理超大文件的 SQL 计算器 + 一个可以落盘的数据库**。

思考题

1. 你目前的工作中，有哪些任务是"先用 Python 脚本读一堆 CSV，再做 pandas 变换，最后写出结果"的？尝试列出来，它们中的大多数都可以用一条 DuckDB SQL 替代。
2. 你的数据真的需要分布式集群吗？用 `du -sh` 估算一下你最大的分析数据集有多大，和一台机器的内存/磁盘对比一下。

下一章

[第 2 章 安装与运行](#) —— 我们会在 10 分钟内跑通第一个查询。

第 2 章 安装与运行

本章目标：在你的机器上成功运行第一条 DuckDB 语句，并熟悉基本的启动方式。

本章会覆盖：各种平台上的安装方法、验证安装、第一个真正有意义的查询、以及“退出后数据去哪了”这个新手第一疑问。

2.1 DuckDB 的两种形态

在安装之前，先明确你要装的是哪一种：

形态	用途	典型使用者
命令行工具 (CLI)	交互式写 SQL、跑脚本、做数据探索	数据工程师、DBA、SQL 爱好者
语言绑定的库	在程序里调用 (Python/Java/Node 等)	应用开发者、数据科学家

两者共享同一个文件格式与引擎：你在 CLI 里创建的数据库文件，Python 可以直接打开继续使用。

建议：**两个都装**。CLI 用来探索与调试，Python 库用来写工程代码。第 3 章会详细说明它们的分工。

2.2 安装命令行工具 (CLI)

2.2.1 macOS

```
1 # Homebrew (推荐)
2 brew install duckdb
3
4 # 或 使用官方安装脚本 (会把二进制放到 ~/.duckdb/cli/latest)
5 curl https://install.duckdb.org | sh
```

Homebrew 安装后直接可用：

```
1 duckdb --version
2 # --> v1.5.5
```

2.2.2 Linux

方式一：官方安装脚本（推荐，自动识别发行版与架构）

```
1 curl https://install.duckdb.org | sh
```

安装脚本会把 CLI 放到 `~/.duckdb/cli/latest/duckdb`，并提示你把它加入 `PATH`。如果安装过程中提示缺少 `unzip`，先装上：

```
1 # Debian/Ubuntu
2 sudo apt-get install -y unzip
3 # RHEL/CentOS/Fedora
4 sudo yum install -y unzip # 或 dnf install -y unzip
5 # Alpine
6 apk add --no-cache unzip
```

然后重新执行安装脚本，或手动把目录加入 `PATH`：

```
1 echo 'export PATH="$HOME/.duckdb/cli/latest:$PATH"' >> ~/.bashrc
2 source ~/.bashrc
```

方式二：手动下载压缩包

从官方下载页 <https://duckdb.org/docs/installation/> 选择合适的压缩包，例如 Linux x86_64：

```
1 # 注意把版本号替换成你下载的版本
2 wget
https://github.com/duckdb/duckdb/releases/download/v1.5.5/duckdb_cli-
linux-amd64.zip
3 unzip duckdb_cli-linux-amd64.zip -d duckdb-cli
4 sudo mv duckdb-cli/duckdb /usr/local/bin/
5 duckdb --version
```

常见架构对应关系：

平台	文件名关键词
Linux x86_64	linux-amd64
Linux ARM64	linux-aarch64
macOS Intel	osx-universal (通用二进制)
macOS ARM (M1/M2/M3)	osx-universal 或 osx-arm64
Windows 64 位	windows-amd64

方式三：部分发行版的包管理器

```
1 # Arch Linux
2 sudo pacman -S duckdb
3 # Nix / NixOS
4 nix-shell -p duckdb
```

⚠ 通过发行版仓库安装的有时版本较旧，建议优先使用官方途径。

2.2.3 Windows

方式一：winget (Windows 10/11 推荐)

```
1 | winget install DuckDB.cli
```

方式二：PowerShell 官方脚本

```
1 | iwr https://install.duckdb.org -useb | iex
```

方式三：Chocolatey / Scoop

```
1 | choco install duckdb
2 | # 或
3 | scoop install duckdb
```

安装完成后，在 PowerShell 或 CMD 中运行 `duckdb --version` 验证。

2.2.4 验证安装

```
1 | duckdb --version
2 | duckdb
3 | # 进入交互界面后：
4 | SELECT version() AS v, current_setting('threads') AS threads;
```

如果看到版本号 and 线程数，说明安装成功。

2.3 安装语言绑定

2.3.1 Python (最常用)

```
1 | pip install duckdb
2 | # 想要最新版/预发布版
3 | pip install --pre duckdb
4 | # 使用 uv
5 | uv pip install duckdb
6 | # 使用 conda / mamba
7 | conda install -c conda-forge python-duckdb
```

验证：

```
1 | python -c "import duckdb; print(duckdb.__version__);
print(duckdb.connect().sql('SELECT 42').fetchall())"
```

Python 版本要求： DuckDB 支持 CPython 3.9+ (老版本可能到 3.7，取决于具体发行版)。

2.3.2 Node.js

```
1 # 新版官方推荐的高性能绑定
2 npm install @duckdb/node-api
3 # 经典绑定 (API 类似 sqlite3 包)
4 npm install duckdb
```

2.3.3 Java / Kotlin / Scala (JDBC)

Maven:

```
1 <dependency>
2   <groupId>org.duckdb</groupId>
3   <artifactId>duckdb_jdbc</artifactId>
4   <version>1.5.5</version>
5 </dependency>
```

Gradle:

```
1 implementation 'org.duckdb:duckdb_jdbc:1.5.5'
```

2.3.4 Go

```
1 go get github.com/marcboeker/go-duckdb
```

2.3.5 Rust

```
1 # Cargo.toml
2 [dependencies]
3 duckdb = { version = "1.5.5", features = ["bundled"] }
```

2.3.6 R

```
1 install.packages("duckdb")
2 # 或用 duckplyr / DBI + dbplyr 生态
3 install.packages(c("DBI", "dbplyr"))
```

2.3.7 其他

- **C / C++**: 下载 `libduckdb-src.zip` 或静态库, 包含 `duckdb.h` / `duckdb.hpp` (amalgamation) ;
- **ODBC / ADBC**: 官方提供驱动, 可被 Tableau、Excel、Power BI、Superset 等工具连接;
- **浏览器**: `@duckdb/duckdb-wasm` ;
- **Julia**、**.NET**: 由社区维护。

第 3 章会对其中几种给出完整的"Hello World"。

2.4 你的第一条语句

2.4.1 在 CLI 里

打开终端，输入：

```
1 | duckdb
```

你会看到类似：

```
1 | DuckDB v1.5.5 ...
2 | Enter ".help" for usage hints.
3 | Connected to a transient in-memory database.
4 | Use ".open DBPATH" to reopen a persistent database.
5 | D
```

提示符 `D` 表示 DuckDB 在等你输入。试试：

```
1 | D SELECT 'Hello, DuckDB!' AS greeting;
```

输出：

```
1 | ┌───────────┐
2 | │ greeting │
3 | └───────────┘
4 | │ Hello, DuckDB! │
5 | └───────────┘
```

CLI 使用小贴士

- 语句末尾一定要写 `;`，否则 DuckDB 会认为你还没输入完（会显示 `...>` 续行提示符）；
- 如果卡住了，按 `Ctrl + C` 可以取消当前输入；
- `Ctrl + D` (Windows 是 `Ctrl + Z` 后回车) 或输入 `.exit` / `.quit` 退出；
- 上下方向键可以翻历史命令，支持自动补全 (Tab 键)。

2.4.2 在 Python 里

```
1 | import duckdb
2 |
3 | # 快速 POSIX: 不用显式 connect
4 | duckdb.sql("SELECT 'Hello, DuckDB!' AS greeting").show()
```

输出：

```
1 | [ greeting ]
2 | [         ]
3 | [         ]
4 | [ Hello, DuckDB! ]
5 | [         ]
```

2.5 第一个"有意义的"查询

Hello World 不能让你理解 DuckDB 的能力。我们用一条语句产生 1000 万行数据并在上面做聚合，感受一下性能。

```
1 | SELECT
2 |     range % 100 AS bucket,      -- 生成 0~99 的桶
3 |     count(*) AS cnt,
4 |     sum(range) AS total,
5 |     avg(range) AS avg_v
6 | FROM range(10_000_000)         -- range() 是一个表函数，返回一列名为
   | range 的行
7 | GROUP BY bucket
8 | ORDER BY bucket
9 | LIMIT 5;
```

`range(n)` 是 DuckDB 的**表函数 (table function)**，返回一列整数；该列默认名为 `range`（在 `FROM range(n)` 中可直接引用，也常写成 `FROM range(n) AS t(i)` 来改名）。`10_000_000` 里的下划线是 SQL 数字字面量允许的分隔符，纯为可读性。

在 CLI 里你会在**毫秒到百毫秒级**看到结果。试着改成 `range(100_000_000)` 再感受一次。

观察：DuckDB 会自动并行。用下面这条语句确认线程数：

```
1 | SELECT current_setting('threads') AS threads;
```

2.5.1 让它"慢"下来看看执行计划（预告）

```
1 | EXPLAIN SELECT count(*) FROM range(100_000_000);
```

你会看到一棵执行计划树。看不懂没关系，第 17 章会把每一个算子讲透。

2.6 最重要的问题：数据到底存在哪？

这是新手一定会遇到的困惑。

2.6.1 内存模式 (transient in-memory)

```
1 | duckdb
```

注意启动后的提示: `Connected to a transient in-memory database.`

这意味着:

- 数据只存在于内存中;
- **退出进程后数据全部消失;**
- 适合临时探索。

Python 中等价写法:

```
1 | import duckdb
2 | con = duckdb.connect()           # 等价于 duckdb.connect(":memory:")
```

2.6.2 持久化模式 (persistent file-backed)

```
1 | duckdb my_tutorial.db
```

提示会变成 `Connected to a transient in-memory database.` 之外的内容 (不再有 transient 字样), 此时:

- 数据保存在文件 `my_tutorial.db` 中;
- 退出后仍然存在;
- 下次 `duckdb my_tutorial.db` 可以继续用;
- 目录下会同时出现 `my_tutorial.db.wal` (预写日志, 第 18 章讲)。

Python 中等价写法:

```
1 | con = duckdb.connect("my_tutorial.db")
```

本书约定: 从本章开始, 除非特别说明, 请使用持久化模式打开一个数据库文件 (例如 `~/duckdb-learn.db`), 这样你做的练习可以跨章节保留。

```
1 | mkdir -p ~/duckdb-learn && cd ~/duckdb-learn
2 | duckdb learn.db
```

2.6.3 只读模式

如果你要确保不会误改数据 (很常见于生产只读副本):

```
1 | duckdb -readonly learn.db
```

或 Python:

```
1 | con = duckdb.connect("learn.db", read_only=True)
```

尝试写入时会报错：

```
1 | TransactionContext Error: Cannot execute statement of type
  | "CREATE_TABLE" on database "learn" which is attached in read-only
  | mode!
```

这个错误很常见，第 20 章会解释。

2.6.4 在 CLI 中切换数据库

不需要退出重启：

```
1 | D .open school.db      -- 打开另一个库
2 | D .open                -- 不带参数：回到全新的内存库
3 | D .open :memory:      -- 同上
```

2.7 准备好本书的示例数据

后续章节需要一些数据。我们用一个**完全离线、可复现**的方式生成，不依赖网络下载。

2.7.1 生成一个销售示例表

在 CLI 里执行以下代码（可以保存为 `seed.sql`，然后用 `duckdb learn.db < seed.sql` 或 `.read seed.sql` 执行）：

```
1 | -- 1) 先清空，保证可重复执行
2 | DROP TABLE IF EXISTS sales;
3 |
4 | -- 2) 建一张有真实感的表
5 | CREATE TABLE sales AS
6 | SELECT
7 |     (i % 5_000) + 1                                AS order_id,
8 |     'U' || lpad((i % 800) + 1, 4, '0')              AS user_id,
9 |     CASE (i % 6)
10 |         WHEN 0 THEN '华东' WHEN 1 THEN '华北'
11 |         WHEN 2 THEN '华南' WHEN 3 THEN '西南'
12 |         WHEN 4 THEN '华中' ELSE '东北'
13 |     END                                              AS region,
14 |     CASE (i % 4)
15 |         WHEN 0 THEN '电子产品' WHEN 1 THEN '服饰'
16 |         WHEN 2 THEN '食品'      ELSE '图书'
17 |     END                                              AS category,
18 |     CASE (i % 3)
19 |         WHEN 0 THEN '已支付' WHEN 1 THEN '已发货' ELSE '已完成'
20 |     END                                              AS status,
21 |     round((random() * 980 + 20)::DECIMAL(10,2), 2) AS amount,
22 |     (random() * 9 + 1)::INTEGER                     AS quantity,
```

```

23     DATE '2026-01-01' + (i % 180)                                AS order_date,
24     (TIMESTAMP '2026-01-01 08:00:00'
25       + INTERVAL (i % 180) DAY
26       + INTERVAL (i % 24) HOUR
27       + INTERVAL (i % 60) MINUTE)                                AS created_at
28 FROM range(200_000) AS t(i);

```

小心 repetition: 由于用了取模, `order_id`、`user_id` 会有大量重复, 这正是我们后面演示去重、窗口函数、连接所需要的"脏数据"。如果你想对所有行唯一, 用 `i` 本身做 ID。

检查一下:

```

1  DESCRIBE sales;
2
3  SELECT * FROM sales LIMIT 5;
4
5  SELECT count(*) AS rows, count(DISTINCT user_id) AS users, sum(amount)
   AS gmV
6  FROM sales;

```

2.7.2 生成一个用户维表

```

1  DROP TABLE IF EXISTS users;
2
3  CREATE TABLE users AS
4  SELECT
5      'u' || lpad(i, 4, '0')                                AS
   user_id,
6      'User' || i                                            AS
   user_name,
7      CASE (i % 3) WHEN 0 THEN '个人版' WHEN 1 THEN '专业版' ELSE '企业
   版' END AS plan,
8      DATE '2020-01-01' + (i % 1_500)                        AS
   reg_date,
9      ((random() * 100)::INTEGER)                            AS
   credit_score
10 FROM range(1, 801) AS t(i);  -- 800 个用户

```

2.7.3 导出为文件 (供第 12 章使用)

```

1  -- 导出成不同格式, 后续章节用来演示"如何查询文件"
2  COPY sales TO 'sales.parquet' (FORMAT PARQUET, COMPRESSION ZSTD);
3  COPY sales TO 'sales.csv' (HEADER, DELIMITER ',');
4  COPY users TO 'users.parquet' (FORMAT PARQUET);

```

看看磁盘占用:

```

1  ls -lh sales.parquet sales.csv

```

一般会看到 Parquet 明显小于 CSV，这是列式压缩的直接体现。

2.7.4 验证内容

```
1 SELECT region, category, count(*) AS cnt, sum(amount) AS gmv
2 FROM sales
3 GROUP BY ALL
4 ORDER BY gmv DESC;
```

`GROUP BY ALL` 是 DuckDB 的语法糖（第 4 章详解）：自动把所有非聚合列作为分组键。

更完整的版本：本书提供了统一的示例数据脚本 `examples/00-seed.sql`（生成 7 张表 + 4 个数据文件，结果完全可复现）：

```
1 duckdb examples.db < examples/00-seed.sql
```

第 22 章的所有实战案例都跑在这份数据上，建议直接用它替代本节的手工生成步骤。

2.8 CLI 的基本功

在正式进入 SQL 之前，先掌握几个必须会的点命令（dot command）。

```
1 D .help                -- 查看所有点命令
2 D .databases           -- 列出当前已挂载的数据库
3 D .tables              -- 列出表（等价于 SHOW TABLES）
4 D .schema              -- 查看所有表的建表语句
5 D .schema sales        -- 只看 sales 表
6 D .mode MODE           -- 切换输出格式：
   duckbox/line/csv/json/markdown/list/html/insert
7 D .headers on          -- 是否显示列名
8 D .separator ','       -- 自定义分隔符
9 D .output result.csv   -- 把后续所有查询结果写入文件
10 D .output              -- 不带参数：恢复到屏幕输出
11 D .once out.txt        -- 只把下一条语句的输出写入文件
12 D .read seed.sql       -- 执行一个 SQL 脚本文件
13 D .timer on            -- 显示每条语句的执行耗时
14 D .open learn.db       -- 打开/切换数据库
15 D .exit                -- 退出
```

版本提示：DuckDB 1.5 起 CLI 经过重新设计，支持更好的自动补全、分页与主题；不同版本可用的点命令略有差异，以 `.help` 输出为准。

一次完整的会话示例：

```
1 D .timer on
2 D .mode markdown
3 D SELECT region, sum(amount) AS gmv FROM sales GROUP BY region ORDER
   BY gmv DESC;
```

2.9 用非交互方式运行

DuckDB 也可以像脚本一样被“喂”一段 SQL，这在自动化里非常有用。

```
1 # 方式一：管道
2 echo "SELECT count(*) FROM sales;" | duckdb learn.db
3
4 # 方式二：重定向文件
5 duckdb learn.db < my_query.sql
6
7 # 方式三：-c 参数，直接给 SQL（可多次）
8 duckdb learn.db -c "SELECT count(*) FROM sales;" -c "SELECT
9 max(amount) FROM sales;"
10
11 # 方式四：-init 启动后先执行脚本再进入交互
12 duckdb learn.db -init init.sql
13
14 # 指定输出格式与结果文件
15 duckdb learn.db -csv -c "SELECT * FROM sales LIMIT 3" > out.csv
16 duckdb learn.db -json -c "SELECT * FROM sales LIMIT 3" > out.json
```

常用命令行参数速查：

参数	作用
<code>-c SQL</code>	执行一条 SQL 后退出（可重复）
<code>-csv / -json / -list / -line / -markdown / -html</code>	指定输出格式
<code>-readonly</code>	只读模式打开
<code>-init FILE</code>	启动后先跑脚本
<code>-no-stdin</code>	不要读取标准输入
<code>-unsigned</code>	允许加载未签名的扩展（第 19 章）
<code>-s3_endpoint</code> 等	直接传配置（第 14 章）
<code>--version</code>	输出版本并退出

2.10 常见安装问题排查

现象	原因	解决
<code>command not found: duckdb</code>	二进制不在 PATH 里	把安装目录加入 PATH，或用绝对路径调用

现象	原因	解决
duckdb: error while loading shared libraries: libstdc++.so.6	系统 GCC 运行时库太老	更新系统 libstdc++，或使用 musl 静态构建版本
Python import duckdb 报错	架构/wheel 不匹配	确认 Python 版本 ≥ 要求版本，pip install --upgrade pip 后重装
启动卡住很久	打开了非常大的数据库文件所在的网络盘	把数据库文件放到本地 SSD
Permission denied	没有对数据库文件/目录的写权限	检查权限；或用 -readonly
数据库被锁住 Could not set lock on file	已有另一个进程打开了该库	关闭那个进程；生产上详见第 18 章
install.duckdb.org 脚本失败	缺少 unzip 或网络问题	安装 unzip；或手动下载 zip

实例演练

实例 2-1 · 把 DuckDB 当脚本用（非交互运行）

```

1  # 一条 SQL
2  duckdb examples.db -c "SELECT count(*) AS orders FROM sales;"
3
4  # 多条 SQL 依次执行
5  duckdb examples.db -c "SELECT count(*) AS orders FROM sales;" \
6                      -c "SELECT round(sum(amount)) AS gmv FROM sales;"

```

实测输出：

```

1  |
2  | orders |
3  |
4  | 200000 |
5  |
6  |
7  | gmv    |
8  |
9  | 102177691 |
10 |

```

这适合写进 shell 脚本 / cron / CI：不需要 Python 胶水代码。

实例 2-2 · 让输出直接被程序消费 (`-json` / `-csv`)

```
1 duckdb examples.db -json -c "SELECT region, count(*) n FROM sales
  GROUP BY 1 ORDER BY n DESC LIMIT 2;"
2 duckdb examples.db -csv -c "SELECT region, count(*) n FROM sales
  GROUP BY 1 ORDER BY n DESC LIMIT 2;"
```

实测输出：

```
1 [{"region":"华南","n":33751},
2 {"region":"华北","n":33594}]
```

```
1 region,n
2 "华南",33751
3 "华北",33594
```

配合重定向就能产出交付物：

```
1 duckdb examples.db -csv -c "SELECT * FROM sales LIMIT 1000" >
  sample.csv
```

实例 2-3 · 交互式里看耗时与竖排输出

```
1 D .timer on
2 D .mode line
3 D SELECT region, count(*) AS n FROM sales GROUP BY 1 ORDER BY n DESC
  LIMIT 2;
```

```
1      region = 华南
2          n = 33751
3      region = 华北
4          n = 33594
5 Run Time (s): real 0.008 user 0.027796 sys 0.000000
```

`.mode line` 在列很多时特别有用（横向放不下时自动竖排）；`.timer on` 是最轻量的对比手段。

实例 2-4 · 持久化 vs 内存（退出后数据还在吗）

```
1 # 内存库：退出即消失
2 duckdb -c "CREATE TABLE t AS SELECT 1 AS x;"
3 duckdb -c "SELECT count(*) FROM t;"      # 报错: Table with name t does
  not exist
4
5 # 持久化库
6 duckdb demo.db -c "CREATE TABLE t AS SELECT 1 AS x;"
7 duckdb demo.db -c "SELECT count(*) FROM t;"  # 1
8 rm -f demo.db
```

2.11 本章小结

- DuckDB 有 **CLI** 和**语言绑定**两种形态，建议都安装。
 - CLI 用 `duckdb [dbfile]` 启动，`duckdb` 不带参数是**内存模式**，程序退出数据就没了。
 - **持久化模式**：`duckdb learn.db`，数据写入文件，跨会话保留。
 - CLI 中的点命令（`.tables`、`.schema`、`.mode`、`.read`、`.timer` 等）是提高效率的关键。
 - 我们准备好了本书贯穿使用的示例数据：`sales`（20 万行）与 `users`（800 行），并导出了 Parquet/CSV 文件备用。
-

动手练习

1. 安装 CLI 与 Python 包，分别运行 `SELECT version();`。
2. 用持久化模式打开 `practice.db`，生成第 2.7 节的两张表，然后**退出**。重新打开，确认数据还在。
3. 尝试 `.mode` 的至少三种输出格式，观察区别（`markdown` 便于粘贴到文档，`json` 便于交给程序）。
4. 用 `duckdb practice.db -c "..."` 的方式查询一次，感受"作为脚本使用"的方式。
5. 把 `sales` 导出为 `csv` 和 `parquet`，比较文件大小，算一下压缩比。

下一章

[第 3 章 交互界面：CLI 与各类客户端](#) —— 深入 CLI，并把 Python / Node.js / Java / Go / Rust 的完整用法讲清楚。

第 3 章 交互界面：CLI 与各类客户端

掌握一个工具，首先要记住它不是只有一个用法。本章把 DuckDB 的"入口"梳理清楚：

- **CLI**：交互式探索、跑脚本、输出各种格式；
- **Python**：数据科学的主战场；
- **其他语言**：Node.js、Java/JDBC、Go、Rust、C/C++、ODBC、WASM。

读完本章，你应该知道在什么场合用哪个入口，并且每个入口都能写出跑得起来的代码。

3.1 CLI 深度指南

CLI 不只是"能打字的地方"。它是一款经过打磨的交互式工具，很多工程师的日常数据探索完全在上面完成。

3.1.1 启动参数总览

```
1 | duckdb [OPTIONS] [FILENAME] [SQL]
```

参数	说明	示例
<code>-c SQL</code>	执行 SQL 后退出，可重复多次	<code>duckdb -c "SELECT 1" -c "SELECT 2"</code>
<code>-csv / -json / -line / -list / -markdown / -html / -insert</code>	指定输出格式	<code>duckdb -json -c "SELECT 1"</code>
<code>-readonly</code>	只读打开	<code>duckdb -readonly prod.db</code>
<code>-init FILE</code>	启动后先执行脚本文件	<code>duckdb -init ~/.duckdbrc</code>
<code>-no-stdin</code>	不读取 stdin（脚本里避免误吞输入）	<code>duckdb -no-stdin -c "..."</code>
<code>-unsigned</code>	允许加载未签名扩展	第 19 章
<code>-s3_*</code> 系列	传递对象存储配置	第 14 章
<code>--version / -v</code>	版本信息	

实用技巧：你可以创建一个 `~/.duckdbrc`（或自定义文件），在里面写上你希望每次启动都执行的配置，例如：

```

1 SET memory_limit = '12GB';
2 SET threads = 8;
3 SET enable_progress_bar = false;
4 INSTALL httpfs; LOAD httpfs;

```

启动时 `duckdb -init ~/.duckdbrc my.db` 即可。（部分版本会自动读取同名 rc 文件，请以各自版本行为为准。）

3.1.2 点命令大全 (Dot Commands)

点命令以 `.` 开头，**不需要也不能加** `;`。

下表是日常最常用的部分，更完整的清单请在 CLI 里运行 `.help`：

命令	作用
<code>.help [pattern]</code>	查看命令帮助，支持模糊匹配
<code>.open [DB]</code>	打开数据库；不带参数则切到新的内存库
<code>.databases</code>	列出已挂载的数据库及其读写模式
<code>.tables [pattern]</code>	列出表与视图
<code>.schema [table]</code>	查看建表/建视图语句（等价于 <code>SHOW CREATE TABLE</code> ）
<code>.mode MODE</code>	输出模式： <code>duckbox</code> （默认）/ <code>line</code> / <code>csv</code> / <code>json</code> / <code>markdown</code> / <code>list</code> / <code>html</code> / <code>insert</code> / <code>trash</code>
<code>.headers on\ off</code>	是否显示列名
<code>.separator SEP</code>	设置列分隔符与行分隔符
<code>.nullvalue STR</code>	NULL 的显示字符串，默认 <code>NULL</code>
<code>.output FILE</code>	后续结果全部重定向到文件；不带参数恢复屏幕
<code>.once FILE</code>	仅把 下一条 语句的输出写入文件
<code>.read FILE</code>	执行 SQL 脚本
<code>.timer on\ off</code>	显示每语句耗时
<code>.changes on\ off</code>	显示受影响的行数
<code>.limit N</code>	每条 SELECT 自动加 <code>LIMIT N</code>
<code>.maxrows N</code>	限制输出的最大行数（防止误刷全表）

命令	作用
<code>.timer on +</code> <code>.mode line</code>	组合起来做基准测试很方便
<code>.dump [table]</code>	导出 SQL 重建脚本
<code>.extensions</code>	查看已加载/可安装的扩展
<code>.version</code>	版本信息
<code>.system CMD</code>	执行 shell 命令，例如 <code>.system ls -lh</code>
<code>.cd DIR</code>	切换工作目录
<code>.exit / .quit</code>	退出
<code>.echo on\ off</code>	回显输入语句（调试脚本时有用）
<code>.print TEXT</code>	打印字符串
<code>.bail on\ off</code>	出错时中止脚本执行

版本差异：1.5 起重做了 CLI，支持更好的语法提示、自动补全候选列表、分页显示和配色主题。老教程里提到的个别命令在新版本可能被移除或改名，请以 `.help` 实际输出为准。

3.1.3 输出模式实操

```

1  .mode line
2  SELECT 1 AS a, 'x' AS b;
3  -- 输出:
4  --      a = 1
5  --      b = x
6
7  .mode json
8  SELECT 1 AS a, 'x' AS b;
9  -- 输出: [{"a":1,"b":"x"}]
10
11 .mode markdown
12 SELECT * FROM users LIMIT 2;
13 -- 输出 markdown 表格，可直接粘贴到文档里
14
15 .mode csv
16 .headers on
17 .separator '|'
18 SELECT * FROM users LIMIT 2;
19 -- 输出管道分隔的 CSV

```

`.once` 的例子——把查询结果存成文件：

```

1 | .mode markdown
2 | .once report.md
3 | SELECT region, count(*) n, sum(amount) gmvt FROM sales GROUP BY ALL
   | ORDER BY gmvt DESC;

```

3.1.4 让终端更好用的几个配置

```

1 | -- 关闭进度条（结果很小的时候进度条反而干扰）
2 | SET enable_progress_bar = false;
3 |
4 | -- 显示更详细的错误信息行号
5 | SET errors_as_json = false;
6 |
7 | -- 让大结果集自动分页（1.5+ 的 CLI 支持 shell 风格分页）
8 | -- 具体开关名请查 .help
9 |
10 | -- 高亮/history 默认已支持，上下键即可翻历史

```

在 CLI 里也可以直接写 SQL 层面的设置：

```

1 | SET memory_limit = '8GB';
2 | SET threads = 4;
3 | SET temp_directory = '/tmp/duckdb_tmp';

```

3.2 Python 客户端（重点）

Python 是 DuckDB 用户最多的场景。它的 API 有两个层次，理解区别非常重要。

3.2.1 两种入口：Module API 与 Connection API

```

1 | import duckdb
2 |
3 | # (A) 模块级便捷 API：内部使用一个默认的内存连接
4 | duckdb.sql("SELECT 1").show()
5 | duckdb.execute("CREATE TABLE t AS SELECT 1 AS x")
6 | print(duckdb.sql("SELECT * FROM t").fetchall())

```

```

1 | # (B) 显式 Connection：工程代码的推荐方式
2 | con = duckdb.connect("my.db")          # 持久化
3 | mem = duckdb.connect(":memory:")       # 内存
4 | ro = duckdb.connect("my.db", read_only=True)
5 | cfg = duckdb.connect(config={"threads": 4, "memory_limit": "8GB"})

```

什么时候用哪个？

- 交互式/Notebook/Jupyter：用 (A)，少打字；
- 写包、写服务、需要并发或多数据源：用 (B)，显式管理连接生命周期。

⚠ 模块级 API 共享同一个全局内存连接，在多线程或库代码中可能引发意料之外的状态，务必避免在生产库代码中使用。

3.2.2 执行与取结果

```
1 con = duckdb.connect()
2
3 con.execute("CREATE TABLE users(id INTEGER, name VARCHAR)")
4 con.execute("INSERT INTO users VALUES (1, 'Alice'), (2, 'Bob')")
5
6 # 方式 1: 取全部 (tuple 列表)
7 rows = con.execute("SELECT * FROM users").fetchall()
8 # [(1, 'Alice'), (2, 'Bob')]
9
10 # 方式 2: 逐条取 (模拟游标, 适合大结果集)
11 cur = con.execute("SELECT * FROM users")
12 print(cur.fetchone())      # (1, 'Alice')
13 print(cur.fetchmany(1))    # [(2, 'Bob')]
14
15 # 方式 3: 只取标量
16 print(con.execute("SELECT count(*) FROM users").fetchone()[0])
17
18 # 方式 4: 关系式 API + 漂亮输出
19 con.sql("SELECT * FROM users").show()
20
21 # 方式 5: 直接转 pandas
22 df = con.execute("SELECT * FROM users").df()
23 # 或 con.sql("SELECT * FROM users").df()
```

`.fetchall()` 会把所有数据拉到 Python 内存——大结果集请改用 `.df()` (Arrow 零拷贝) 或分批 `fetchmany`。

3.2.3 参数绑定 (务必掌握)

永远不要把用户输入拼接到 SQL 字符串里 (哪怕是“内部工具”)：

```
1 # ❌ 危险: SQL 注入
2 email = "a'; DROP TABLE users; --"
3 con.execute(f"SELECT * FROM users WHERE email = '{email}'")
4
5 # ✅ 正确: 参数绑定
6 con.execute("SELECT * FROM users WHERE email = ?", [email])
7
8 # ✅ 也可以用具名参数 (不同版本支持情况略有差异, 见下)
9 con.execute("INSERT INTO users VALUES ($id, $name)",
10             {"id": 3, "name": "Carol"})
```

Python API 支持三种占位符风格 (取决于具体版本的驱动实现)：

风格	示例	说明
----	----	----

风格	示例	说明
qmark	<code>WHERE id = ?</code>	通用推荐
named / dollar	<code>WHERE id = :id</code> 或 <code>WHERE id = \$id</code>	具名

如果传 dict 报错（旧版本），先用 qmark 风格，或查阅对应版本的文档。

3.2.4 关系式 API (Relational API)

这是 DuckDB Python API 的精髓：**构建惰性查询，最后一次性执行。**

```

1  rel = con.table("sales")                # 得到
   Duc.kDBPyRelation
2  rel = rel.filter("amount > 500")        # WHERE
3  rel = rel.project("order_date, amount, region") # SELECT
4  rel = rel.aggregate("region, sum(amount) AS gmv")
5  rel = rel.order("gmv DESC")
6  rel = rel.limit(10)
7
8  rel.show()                             # 打印
9  df = rel.df()                          # pandas
10 arrow_tbl = rel.arrow() # PyArrow Table
11 # Polars (需安装 polars)
12 pl_df = rel.pl()
13
14 print(rel.sql_query()) # 查看最终生成的 SQL (调试神器)

```

关系式 API 的好处：

- **惰性**：只在你调用 `.show()/.df()/.arrow()/.fetchall()` 时才真正执行；
- **可组合**：像写 pandas 链式调用一样写 SQL；
- **与 SQL 等价**：`rel.sql_query()` 可以打印出最终的 SQL，方便学习与迁移。

常用方法：

方法	等价 SQL
<code>.project(cols)</code>	<code>SELECT cols</code>
<code>.select(cols)</code>	<code>SELECT cols</code>
<code>.filter(cond)</code>	<code>WHERE cond</code>
<code>.aggregate(expr)</code>	<code>GROUP BY</code>
<code>.distinct()</code>	<code>SELECT DISTINCT</code>
<code>.order(expr)</code>	<code>ORDER BY</code>
<code>.limit(n, offset=0)</code>	<code>LIMIT / OFFSET</code>

方法	等价 SQL
<code>.union(other)</code>	UNION
<code>.join(other, cond, how=...)</code>	JOIN
<code>.except_(other)</code> / <code>.intersect(other)</code>	EXCEPT / INTERSECT
<code>.describe()</code>	类似 pandas 的 describe
<code>.insert_into(table)</code>	INSERT INTO
<code>.create(name)</code> , <code>.create_view(name)</code>	CREATE TABLE/VIEW AS
<code>.write_parquet(path)</code> / <code>.write_csv(path)</code>	COPY ... TO
<code>.to_arrow_table()</code>	同 <code>.arrow()</code>
<code>.explain()</code>	EXPLAIN
<code>.columns</code> / <code>.types</code> / <code>.dtypes</code>	元数据

3.2.5 直接从 Python 对象建表

```

1 import pandas as pd
2 import pyarrow as pa
3
4 df = pd.DataFrame({"a": [1, 2, 3], "b": ["x", "y", "z"]})
5
6 # 1) replacement scan: 在 SQL 里直接引用变量名（最省事）
7 duckdb.sql("SELECT sum(a) FROM df").show()
8
9 # 2) 显式注册（推荐，避免作用域问题）
10 con.register("df_view", df)
11 con.execute("SELECT * FROM df_view").show()
12 con.unregister("df_view")
13
14 # 3) 从 Arrow Table 创建永久表
15 tbl = pa.table({"x": [1, 2, 3]})
16 con.execute("CREATE TABLE t AS SELECT * FROM tbl")

```

"Replacement Scan" 是什么：DuckDB 执行 SQL 时，如果发现 FROM 里的名字既不是表名也不是视图，就会去 Python 的当前作用域里找同名变量（pandas/Polars/Arrow/甚至 `duckdb.DuckDBPyRelation`），找到就用它代替。这是它能做到零胶水代码的关键机制。

3.2.6 事务控制

```

1 con.execute("BEGIN TRANSACTION")
2 try:
3     con.execute("INSERT INTO sales SELECT * FROM staging")
4     con.execute("COMMIT")
5 except Exception:
6     con.execute("ROLLBACK")
7     raise

```

Python 也支持上下文管理器风格的写法（部分版本提供 `with con.cursor()` 等便捷 API，请以版本文档为准）。

3.2.7 一个完整的 Python 例子

```

1 import duckdb
2
3 con = duckdb.connect("analytics.db", config={"threads": 8,
4 "memory_limit": "12GB"})
5
6 con.execute("""
7     CREATE OR REPLACE TABLE events AS
8     SELECT * FROM read_parquet('events/**/*.*parquet', union_by_name
9     = true)
10 """)
11
12 # 每日活跃 + 次日留存
13 con.execute("""
14     CREATE OR REPLACE TABLE daily_stats AS
15     WITH base AS (
16         SELECT user_id, event_time::DATE AS d
17         FROM events
18         GROUP BY 1, 2
19     )
20     SELECT
21         a.d,
22         count(DISTINCT a.user_id) AS dau,
23         count(DISTINCT CASE WHEN b.d = a.d + INTERVAL 1 DAY
24             THEN b.user_id END) AS retained,
25         count(DISTINCT CASE WHEN b.d = a.d + INTERVAL 1 DAY
26             THEN b.user_id END)::DOUBLE
27             / count(DISTINCT a.user_id) AS retention
28     FROM base a
29     LEFT JOIN base b ON a.user_id = b.user_id AND b.d > a.d
30     GROUP BY a.d
31     ORDER BY a.d
32 """)
33
34 con.sql("SELECT * FROM daily_stats ORDER BY d DESC LIMIT 7").show()
35 con.close()

```

3.3 Node.js

3.3.1 新式 @duckdb/node-api (推荐)

```
1  const { DuckDBInstance } = require("@duckdb/node-api");
2  // 或 import { DuckDBInstance } from "@duckdb/node-api";
3
4  const instance = await DuckDBInstance.create(":memory:");
5  const con = await instance.connect();
6
7  await con.run("CREATE TABLE people (id INTEGER, name VARCHAR)");
8  await con.run("INSERT INTO people VALUES (1, 'Alice'), (2, 'Bob')");
9
10 const reader = await con.runAndReadAll("SELECT * FROM people");
11 console.log(reader.getRowsJS()); // [{ id: 1, name: 'Alice' }, ...]
12
13 // 也可直接转成 JSON / 行对象
14 console.log(JSON.stringify(reader.getRowsJson()));
```

它还支持直接从 JS 对象插入 (Appender 风格) 与 Arrow 互操作。

3.3.2 经典 duckdb 包

```
1  const duckdb = require("duckdb");
2
3  const db = new duckdb.Database(":memory:");
4  const con = db.connect();
5
6  con.all("SELECT 42 AS answer", (err, res) => {
7    if (err) throw err;
8    console.log(res);           // [ { answer: 42 } ]
9  });
10
11 con.run("CREATE TABLE items(a INTEGER, b VARCHAR)");
12 // 推荐用 prepared statement + 批量执行, 避免逐条 INSERT 的开销
13 con.prepare("INSERT INTO items VALUES (?, ?)", (err, stmt) => {
14   for (const [a, b] of [[1, "x"], [2, "y"]]) stmt.run(a, b);
15   stmt.finalize();
16 });
17
18 // 流式读取大结果集 (避免一次性占满内存)
19 con.each("SELECT * FROM items", (err, row) => { console.log(row);
20 });
```

注意：逐条 `con.run(INSERT)` 在 DuckDB 里非常慢（每条都是一个事务）。生产请用 prepared statement 或 Appender。

3.4 Java / JDBC

```
1  import java.sql.*;
2  import java.util.Properties;
```

```

3
4 public class DuckDbDemo {
5     public static void main(String[] args) throws Exception {
6         // 内存:   jdbc:duckdb:
7         // 文件:   jdbc:duckdb:/path/to/mydb.db
8         String url = "jdbc:duckdb:analytics.db";
9         Properties props = new Properties();
10        props.setProperty("duckdb_api_config", "");
11
12        try (Connection conn = DriverManager.getConnection(url,
13        props);
14            Statement stmt = conn.createStatement()) {
15            stmt.execute("CREATE OR REPLACE TABLE t AS SELECT 1 AS
16            a, 'x' AS b");
17            try (ResultSet rs = stmt.executeQuery("SELECT * FROM
18            t")) {
19                while (rs.next()) {
20                    System.out.println(rs.getInt("a") + ", " +
21                    rs.getString("b"));
22                }
23            }
24        }
25    }
26 }

```

要点:

- JDBC URL 格式: `jdbc:duckdb:<path>`, 留空即内存库;
- 支持 `PreparedStatement`、`ResultSet`、`batch()`;
- **Appender API** (`DuckDBAppender`) 用于高速批量写入, 比逐条 `executeUpdate` 快几十倍;
- 支持 Arrow (`ArrowStreamReader` 等绑定, 视版本而定)。

3.5 Go

```

1 package main
2
3 import (
4     "database/sql"
5     "fmt"
6     _ "github.com/marcboeker/go-duckdb"
7 )
8
9 func main() {
10    db, err := sql.Open("duckdb", "analytics.db")
11    if err != nil { panic(err) }
12    defer db.Close()
13

```

```

14     _, err = db.Exec(`CREATE OR REPLACE TABLE t AS SELECT 1 AS a,
15     'x' AS b`)
16     if err != nil { panic(err) }
17
18     rows, err := db.Query(`SELECT a, b FROM t`)
19     if err != nil { panic(err) }
20     defer rows.Close()
21
22     for rows.Next() {
23         var a int
24         var b string
25         if err := rows.Scan(&a, &b); err != nil { panic(err) }
26         fmt.Println(a, b)
27     }

```

- 使用标准 `database/sql`;
- 支持 `NewConnector` 传配置、`Appender` 批量写;
- CGO 依赖较多, 注意交叉编译时使用 musl 版本。

3.6 Rust

```

1 use duckdb::{params, Connection, Result};
2
3 fn main() -> Result<> {
4     let conn = Connection::open("analytics.db")?; // 或
5     open_in_memory()?
6
7     conn.execute_batch(
8         "CREATE TABLE IF NOT EXISTS person (id INTEGER, name TEXT);
9         INSERT INTO person VALUES (1, ?1);",
10    )?;
11
12    let mut stmt = conn.prepare("SELECT id, name FROM person")?;
13    let rows = stmt.query_map([], |row| {
14        Ok((row.get::(<_, i32>(0)?, row.get::(<_, String>(1)?))
15    )?);
16
17    for person in rows {
18        println!("{:?}", person);
19    }
20    ok(())
21 }

```

3.7 C / C++

官方提供 amalgamation (单文件源码) 与静态/动态库, 适合嵌入到自己产品里。

```

1  #include "duckdb.h"
2  #include <stdio.h>
3
4  int main() {
5      duckdb_database db;
6      duckdb_connection con;
7      if (duckdb_open("analytics.db", &db) != DuckDBSuccess) return 1;
8      if (duckdb_connect(db, &con) != DuckDBSuccess) return 2;
9
10     duckdb_result res;
11     duckdb_query(con, "SELECT 42 AS answer", &res);
12     printf("%lld\n", duckdb_value_int64(&res, 0, 0)); // 42
13
14     duckdb_destroy_result(&res);
15     duckdb_disconnect(&con);
16     duckdb_close(&db);
17     return 0;
18 }

```

C++ API 更现代 RAII 风格（详见第 19 章提到 C API 时再展开）。

3.8 ODBC / ADBC / BI 工具连接

如果你希望用 Excel、Tableau、Power BI、Superset、Metabase 连接 DuckDB：

1. 安装官方 ODBC 驱动（Windows/macOS/Linux 均有，注意版本匹配）；
2. 配置 DSN（或直接写连接字符串 `Driver=DuckDB Driver;Database=/path/x.db`）；
3. 在 BI 工具里选 ODBC 数据源。

注意事项：

- DuckDB 是**单进程写**模型，多个 BI 工具同时打开一个 db 文件会遇到锁冲突（第 18 章）；
- 常见做法是：**ETL 过程写库** → **完成后关闭连接**（释放锁）→ BI 只读打开。
- 更稳妥的做法是把数据写成 Parquet/Delta/Iceberg，BI 直接读文件（`read_parquet` 的 ODBC 等价物在 BI 里可能需要先在视图里定义）。

ADBC（Arrow Database Connectivity）也是 DuckDB 支持的现代标准，适合 Python/Java/Go 之间传输 Arrow 数据。

3.9 DuckDB-WASM：在浏览器里跑数据库

```
1 | npm install @duckdb/duckdb-wasm
```

```

1  import * as duckdb from "@duckdb/duckdb-wasm";
2
3  const JSDELIVR_BUNDLES = duckdb.getJsDelivrBundles();
4  const bundle = await duckdb.selectBundle(JSDELIVR_BUNDLES);

```

```

5  const worker = new worker(bundle.mainworker);
6  const logger = new duckdb.ConsoleLogger();
7  const db = new duckdb.AsyncDuckDB(logger, worker);
8  await db.instantiate(bundle.mainModule, bundle.pthreadworker);
9
10 const conn = await db.connect();
11 await conn.query(`
12     CREATE TABLE weather AS
13     SELECT * FROM read_csv_auto('https://.../weather.csv')
14 `);
15 const res = await conn.query("SELECT * FROM weather LIMIT 5");
16 console.log(res.toArray().map(r => r.toJSON()));

```

用途：

- 纯前端的数据分析产品（数据不出浏览器，隐私友好）；
- 静态博客里嵌入交互式 SQL 查询；
- 官方 shell.duckdb.org 就是跑 WASM 版本。

限制：受浏览器沙箱限制，只能访问 HTTP 数据源或用户上传的文件，无法访问本地文件系统（除 File System Access API 场景）。

3.10 如何选择合适的入口

你的身份/场景	推荐入口
探索数据、写一次性查询、临时算指标	CLI
Notebook / 数据科学	Python（模块级 API + <code>.df()</code> ）
数据管道 / 服务 / ETL 脚本	Python 显式 Connection，或 Go/Java/Rust
前端产品或演示	DuckDB-WASM
BI 报表工具	ODBC / JDBC
嵌入自有程序	C/C++ API 或对应语言绑定
协作式数据平台	Python + 写 Parquet/Delta 到对象存储

一个很常见的组合是：Python 写 ETL 与复杂逻辑，CLI 用来调试 SQL，ODBC 给 BI 工具查 prepared 结果表。

实例演练

实例 3-1 · 用 `-init` 预置配置（团队统一环境的好办法）

`init.sql`:

```
1 SET threads = 4;
2 SET memory_limit = '4GB';
3 SET enable_progress_bar = false;
```

运行:

```
1 duckdb examples.db -init init.sql -c "SELECT
  current_setting('threads') t, current_setting('memory_limit') m;"
```

实测输出:

t	m
4	3.7 GiB

可以把这个文件放进项目仓库，让所有人跑同一套参数。

实例 3-2 · 一条命令产出 Markdown 报表

```
1 duckdb examples.db -c ".mode markdown" \
2   -c "SELECT region, count(*) n FROM sales GROUP BY 1 ORDER BY n DESC
  LIMIT 3;"
```

实测输出（可直接粘贴进文档）：

region	n
华南	33751
华北	33594
华中	33400

配合 `.once` 落盘:

```
1 .mode markdown
2 .once report.md
3 SELECT region, count(*) n FROM sales GROUP BY 1 ORDER BY n DESC;
```

实例 3-3 · 查看这个库有多大

```
1 PRAGMA database_size;
```

实测输出（markdown 模式）：

database_name	database_size	block_size	total_blocks	used_blocks	free_blocks	wal_size	memory_usage	memory_limit
examples	8.0 MiB	262144	32	32	0	0 bytes	768.0 KiB	3.7 GiB

`wal_size = 0 bytes` 说明所有改动都已 checkpoint 进主文件（第 18 章）。

实例 3-4 · Python 关系式 API（链式、惰性）

```

1  import duckdb
2
3  con = duckdb.connect("examples.db")
4
5  rel = (con.table("sales")
6         .filter("amount > 500")
7         .project("region, amount, order_date")
8         .aggregate("region, sum(amount) AS gmv")
9         .order("gmv DESC")
10        .limit(3))
11
12  rel.show()                # 漂亮打印
13  print(rel.sql_query())    # 看看最终生成的 SQL（学习/调试神器）
14  df = rel.df()             # 转 pandas（Arrow 零拷贝）

```

⚠ 本机环境缺 pip，无法安装 duckdb 的 Python 包，因此第 3/15 章的 Python 代码未在本机实测，仅有 SQL/CLI 部分是实测的。这些代码依据 Python API 文档编写，如与你的版本有出入，请以 `help(duckdb.DuckDBPyConnection)` 为准。

3.11 本章小结

- CLI 是探索与调试最快的入口；点命令 `.tables/.schema/.mode/.read/.timer/.once` 是效率核心。
- Python 有两套 API：模块级便捷 API 适合交互；显式 `Connection` 才是工程代码应该用的。
- Python 的**关系式 API** 让你像操作 pandas 一样构建惰性 SQL 查询。
- Python 务必使用参数绑定，禁止字符串拼接 SQL。
- 其他语言（Node/Java/Go/Rust/C）都提供标准数据库 API 与高速 Appender 批量写入接口。
- ODBC/ADBC 让 BI 工具也能连 DuckDB，但要注意单进程写的约束。

动手练习

1. 用 `.mode line` 看一行数据，用 `.once out.md + .mode markdown` 导出一份报表。
2. 在 Python 中用关系式 API 重写第 2 章的 `SELECT region, sum(amount) FROM sales GROUP BY ALL`，并用 `rel.sql_query()` 打印它生成的 SQL。
3. 用参数绑定写一个函数：`get_orders_by_region(con, region)`。
4. 如果你熟悉 Java/Go/Rust 中的一种，跑通本章对应的 Hello World。

5. 试一下 `con.execute("SELECT * FROM sales").arrow()` 与 `.df()`, 用 `pyarrow.Table.nbytes` 感受结果集的体积。

下一章

[第4章 好友 SQL: DuckDB 的语法糖](#) —— 从这一章开始, 你会发现写 DuckDB SQL 比标准 SQL 舒服得多。

第 4 章 好友 SQL: DuckDB 的语法糖

DuckDB 官方把自己的方言叫做 **Friendly SQL (友好 SQL)**。这些语法不是新标准，而是**减少重复劳动的写法**：你已经写了 `GROUP BY`，为什么还要把同样的列再抄一遍？

本章把这套“福利”集中讲清。这些内容在第 7~10 章学习核心 SQL 时会被反复使用，先于此加深印象。

贯穿全章的示例数据沿用第 2 章的 `sales` 与 `users` 表。

4.1 可选的 SELECT: FROM 优先的写法

标准 SQL 要求 `SELECT ... FROM ...`。DuckDB 允许省略 **SELECT**，直接从 `FROM` 开始：

```
1  -- 传统写法
2  SELECT * FROM sales LIMIT 3;
3
4  -- DuckDB 的 FROM-first 写法
5  FROM sales LIMIT 3;
```

更常见的用法是 `SELECT` 紧跟在 `FROM` 之后，其余子句再依次展开：

```
1  FROM sales
2  SELECT order_id, amount, order_date
3  WHERE region = '华东'
4  ORDER BY amount DESC
5  LIMIT 5;
```

⚠ 已实测 (DuckDB 1.5.5) 的顺序规则： `FROM` 之后必须立刻写 `SELECT`，然后才是 `WHERE / GROUP BY / HAVING / ORDER BY / LIMIT`。

写成 `FROM sales WHERE ... SELECT ...` 会报 `Parser Error: syntax error at or near "SELECT"`。

也就是：

```
1  FROM <数据源>
2  [SELECT <投影>]
3  [WHERE] [GROUP BY] [HAVING] [ORDER BY] [LIMIT]
```

4.1.1 为什么这样更好？

考虑一个多层嵌套的查询，传统写法必须从里往外读：

```
1  SELECT city, cnt FROM (
2      SELECT city, count(*) AS cnt FROM (
3          SELECT city, amount FROM sales WHERE amount > 100
4      ) GROUP BY city
5  ) ORDER BY cnt DESC;
```

而 FROM-first 写法可以**从上往下读**，像管道一样：

```
1 FROM (FROM sales WHERE amount > 100 SELECT city, amount)
2 SELECT city, count(*) AS cnt
3 GROUP BY city
4 ORDER BY cnt DESC;
```

是不是有点像 Python 的 pandas 链式调用？这是刻意的设计：让 SQL 从“嵌套”变成“线性管道”。

在实际写 Ad-hoc 查询时，尤其是配合 CTE（第 10 章）时，这种写法非常顺手：

```
1 FROM 'logs/2026/**/.*.parquet'
2 SELECT
3     date_trunc('hour', ts) AS hour,
4     count(*)                AS errors,
5     count(DISTINCT host)    AS hosts
6 WHERE level = 'ERROR'
7 GROUP BY hour
8 HAVING errors > 100
9 ORDER BY errors DESC;
```

注意：不同 DuckDB 版本对子句重排的具体容忍度略有差异；最稳妥的是“FROM / WHERE / GROUP BY / HAVING 在前，SELECT / ORDER BY / LIMIT 在后”这个顺序，运行时报语法错的话就换回标准顺序。

4.2 SELECT * 的进阶用法

SELECT * 很方便，但一旦表有 50 列就变成灾难。DuckDB 提供了四个修饰符。

4.2.1 EXCLUDE：排除若干列

```
1 SELECT * EXCLUDE (created_at, quantity) FROM sales LIMIT 2;
2 SELECT * EXCLUDE (amount) FROM sales;
```

典型用途：宽表（几十上百列）里排除大对象列或敏感列（密码 token 等）。

```
1 SELECT * EXCLUDE (password_hash, ssn) FROM users;
```

4.2.2 REPLACE：在原地改写某列

```
1 SELECT * REPLACE (round(amount, 0) AS amount, lower(status) AS status)
2 FROM sales
3 LIMIT 3;
```

等价于：

```

1 SELECT order_id, user_id, region, category, status, round(amount,0) AS
  amount, quantity,
2     order_date, lower(status) AS status, created_at
3 FROM sales LIMIT 3;

```

差别显而易见：**你不用知道这张表究竟有哪些列。**

4.2.3 COLUMNS(): 对"列集合"做运算

这是 DuckDB 最有表现力的语法之一。

```

1 -- 对所有 DOUBLE 类型的列取整数
2 SELECT COLUMNS(lambda c: starts_with(c, 'amount'))::INTEGER FROM sales
  LIMIT 3;
3
4 -- 更常见: 对所有数值列求和
5 SELECT sum(COLUMNS(*)) FROM sales; -- 会作用于所有列 (含非数值会报错)

```

下面是一个干净、可直接运行的示例。先建一张宽表：

```

1 CREATE OR REPLACE TABLE wide AS
2 SELECT
3     i                                AS id,
4     (i % 7)::DOUBLE                 AS amount_a,
5     (i % 13)::DOUBLE                AS amount_b,
6     (i % 3)::DOUBLE                 AS qty_x,
7     'n' || i                        AS note
8 FROM range(5) AS t(i);

```

用法一：**选出满足条件的列**

```

1 -- 只选出名字包含 amount 的列 (用 lambda 表达式筛选列名)
2 SELECT COLUMNS(lambda c: contains(c, 'amount')) FROM wide;
3
4 -- 只选出名字以 qty 开头的列
5 SELECT COLUMNS(lambda c: starts_with(c, 'qty')) FROM wide;

```

用法二：**对一批列做同一个运算** (这才是杀手锏)

```

1 -- 对所有名字包含 amount 的列求和
2 SELECT sum(COLUMNS(lambda c: contains(c, 'amount')))) FROM wide;
3 -- 结果列名会变成 sum(amount_a)、sum(amount_b)
4
5 -- 给结果统一改名前缀
6 SELECT sum(COLUMNS(lambda c: contains(c, 'amount')))) AS "total_\0"
  FROM wide;

```

上面 AS "total_\0" 中的 \0 代表"原列名", 这是 COLUMNS 提供的占位符语法。

用法三：**对所有列做同一个运算**

```

1  -- 把所有数值列取整（需要先用 lambda 排除非数值列，否则会报类型错误）
2  SELECT round(COLUMNS(lambda c: c LIKE 'amount%' OR c LIKE 'qty%'))
   FROM wide;

```

用法四：与 `EXCLUDE` / `REPLACE` 组合

```

1  SELECT COLUMNS(lambda c: c <> 'note') FROM wide;    -- 等价于 SELECT *
   EXCLUDE (note)

```

`COLUMNS(*)` 的经典场景：一张宽表的几十个金额字段都要 `COALESCE(x, 0)` 或都要 `round(x, 2)`，只需写一行。

⚠ 注意：`COLUMNS` 的 lambda 里比较的是列名字符串，不是列的值。写以前请用 `DESCRIBE` 表名 确认列名。

4.2.4 * EXCLUDE/REPLACE 与视图

注意：这些语法在视图定义里同样可用，但视图的列是展开固定的，后续给基表加列不会自动同步到视图（这是通用 SQL 行为，不是 DuckDB 缺陷）。

4.3 GROUP BY ALL / ORDER BY ALL

这是每天都能用到的省事写法。

4.3.1 GROUP BY ALL

```

1  -- 传统：分组键要抄一遍
2  SELECT region, category, count(*) AS cnt, sum(amount) AS gmv
3  FROM sales
4  GROUP BY region, category;
5
6  -- DuckDB
7  SELECT region, category, count(*) AS cnt, sum(amount) AS gmv
8  FROM sales
9  GROUP BY ALL;

```

`ALL` 的含义：所有 `SELECT` 列表中的非聚合表达式自动作为分组键。

如果你误写了聚合与分组：

```

1  SELECT region, count(*) FROM sales GROUP BY ALL;    -- ✓
   region 分组
2  SELECT region, sum(amount) FROM sales GROUP BY ALL; -- ✓
   region 分组
3  SELECT sum(amount), count(*) FROM sales GROUP BY ALL; -- ✓ 全
   局聚合，无分组键

```

注意：`GROUP BY ALL` 不适用于 `SELECT *`（通配符不会被展开成分组键），需要显式指定列。

4.3.2 ORDER BY ALL

```
1 SELECT a, b, c FROM t ORDER BY ALL;      -- 按 a, b, c 依次排序
2 SELECT a, b, c FROM t ORDER BY ALL DESC;
```

4.3.3 按序号/位置聚合

```
1 SELECT region, count(*) FROM sales GROUP BY 1 ORDER BY 2 DESC;
2 -- 1 = 第 1 列 region, 2 = 第 2 列 count(*)
```

位置写法很省事，但**重构时容易出错**（改了 SELECT 列表就错位）。生产代码建议写列名或 `GROUP BY ALL`。

4.4 过滤与比较的人性化写法

4.4.1 LIKE / SIMILAR TO / GLOB

```
1 SELECT DISTINCT status FROM sales WHERE status LIKE '已%';      -- 通
   配: % 任意长度, _ 单个字符
2 SELECT * FROM users WHERE user_name SIMILAR TO 'User(1|2)\d';    -- 正
   则子集 (SQL 标准 SIMILAR TO)
3 SELECT * FROM users WHERE user_name ~ '^User\d+$';              --
   POSIX 正则 (推荐用这个)
4 SELECT * FROM users WHERE user_name GLOB 'User*';                --
   Unix 风格通配
```

DuckDB 中推荐使用 `regexp_matches(str, pattern)` 或 `~` 运算符做正则，功能比 `SIMILAR TO` 完整（后者是 SQL 标准的一个简化子集）。

正则相关的常用函数：

```
1 SELECT regexp_extract('abc-123', '(\d+)', 1) AS num;      -- 123
2 SELECT regexp_replace('a b c', '\s+', ' ') AS s;        -- 'a b
   c'
3 SELECT regexp_matches('hello@x.com', '^[^@]+@[^@]+$') AS ok; -- true
4 SELECT strpos_regex('abc123', '\d') AS pos;              -- 位置
```

4.4.2 IN / ANY / ALL 的列表写法

```
1 SELECT * FROM sales WHERE region IN ('华东', '华北');
2 SELECT * FROM sales WHERE region NOT IN ('华东', '华北');
3
4 -- 从另一个查询结果过滤
5 SELECT * FROM sales WHERE user_id IN (SELECT user_id FROM users WHERE
   plan = '企业版');
```

4.4.3 链式比较

```
1 | SELECT * FROM sales WHERE amount BETWEEN 100 AND 500;
2 | SELECT * FROM sales WHERE 100 <= amount AND amount <= 500;  -- 等价
```

注意 BETWEEN 是闭区间 [100, 500]，且边界顺序不能写反。

4.4.4 字符串好看贴士：单引号是对的

DuckDB 遵循标准：单引号表示字符串，双引号表示标识符（列名/表名）。

```
1 | SELECT 'hello' AS greeting, "a b" AS weird_column FROM t;
```

双引号包裹 "My Table" 可以包含空格；但如果列名本身无需转义，" 也可以用来避免关键字冲突（例如 "date"）。

4.5 抽样：USING SAMPLE

大数据集上"先看一眼"的利器。

```
1 | -- 按行数抽样：1000 行（近似）
2 | SELECT * FROM sales USING SAMPLE 1000;
3 |
4 | -- 按百分比抽样
5 | SELECT * FROM sales USING SAMPLE 1%;
6 |
7 | -- 按行数抽样：结果大致为 1000 行，但需要确定总行数
8 | SELECT * FROM sales USING SAMPLE 1000 ROWS;
9 |
10 | -- 分层抽样
11 | SELECT * FROM sales USING SAMPLE 10 PERCENT (stratify, "region");
```

不同选项的性能代价：

写法	行为	性能
USING SAMPLE n	从每个数据源取前 n 行左右（近似）	最快，几乎不读整个文件
USING SAMPLE n%	按百分比的伯努利系统采样	需要确定总行数/百分比，很快
USING SAMPLE n ROWS	蓄水池采样：结果行数基本确定	需要扫全表，但结果行数可控
USING SAMPLE reservoir(n)	蓄水池：结果恰好 n 行（若总行数 ≥ n）	需要扫全表

写法	行为	性能
<code>USING SAMPLE n% (bernoulli)</code>	每行以 n% 概率独立入选	扫全表，真正随机
<code>USING SAMPLE ... (stratify)</code>	分层采样	见下

探索阶段用 `USING SAMPLE 1000` 最实用：几乎瞬间返回，适合了解列结构、典型值。

4.6 数据修改的友好写法

4.6.1 UPSERT: `INSERT OR REPLACE / ON CONFLICT`

DuckDB 支持标准的 upsert:

```

1 CREATE TABLE t (id INTEGER PRIMARY KEY, v VARCHAR, n INTEGER);
2
3 INSERT INTO t VALUES (1, 'a', 10);
4
5 -- (A) 标准占位语法: 冲突时什么都不做
6 INSERT INTO t VALUES (1, 'b', 20) ON CONFLICT DO NOTHING;
7
8 -- (B) 标准占位语法: 冲突时更新 (upsert)
9 INSERT INTO t VALUES (1, 'b', 20)
10 ON CONFLICT (id) DO UPDATE SET v = excluded.v, n = excluded.n;
11
12 -- (C) DuckDB 的简化写法
13 INSERT OR REPLACE INTO t VALUES (1, 'b', 20);
14 INSERT OR IGNORE INTO t VALUES (1, 'c', 30);

```

前提：必须有主键/唯一约束，否则"冲突"无从判断。

`DO UPDATE` 里用 `excluded.xxx` 引用"试图插入的新值"。

4.6.2 INSERT BY NAME: 按名字对齐列

写 ETL 时最烦人的一件事：两张表的列顺序不一致。

```

1 -- 传统: 必须写出完整列顺序
2 INSERT INTO target (a, b, c) SELECT a, b, c FROM source;
3
4 -- DuckDB: 只要列名能对上即可
5 INSERT INTO target BY NAME SELECT * FROM source;

```

反向也有:

```

1 INSERT OR REPLACE INTO target BY NAME SELECT * FROM source;

```

⚠ 已实测 (1.5.5) : `BY NAME` 只解决"顺序不同", 不解决"源表多出列"。

```

1  -- src 比 tgt 多一列 extra
2  INSERT INTO tgt BY NAME SELECT * FROM src;
3  -- Binder Error: Table "tgt" does not have a column with name
   "extra"
4
5  -- ☒ 只选出目标表有的列
6  INSERT INTO tgt BY NAME SELECT price, sku FROM src;  -- OK

```

也就是说：`SELECT` 出来的每一列都必须在目标表中找得到同名列。

4.6.3 UNION BY NAME

两张结构不同的表纵向拼接：

```
1 | SELECT * FROM sales_2025 UNION BY NAME SELECT * FROM sales_2026;
```

列按名字对齐，缺失的列填 `NULL`。

对比：

写法	行为
<code>UNION</code>	要求两侧列数、顺序、类型一致；去重
<code>UNION ALL</code>	同上但不去重（更快）
<code>UNION BY NAME</code>	按列名对齐，缺失列自动补 <code>NULL</code>
<code>UNION ALL BY NAME</code>	按列名对齐且不去重

`UNION BY NAME` 是数据湖里合并多个版本数据的救命稻草。

4.6.4 DELETE / UPDATE 的 USING

```

1  UPDATE sales
2  SET amount = amount * 0.9
3  WHERE region = '华东';
4
5  -- 带 JOIN 的更新（用 FROM/USING）
6  UPDATE sales
7  SET amount = u.new_amount
8  FROM (VALUES (1, 99.9), (2, 88.8)) AS u(order_id, new_amount)
9  WHERE sales.order_id = u.order_id;
10
11 -- 删除
12 DELETE FROM sales WHERE order_date < DATE '2026-01-15';

```

4.7 CTAS / 视图 / 临时对象

```

1  -- CTAS: CREATE TABLE AS SELECT
2  CREATE OR REPLACE TABLE region_summary AS
3  SELECT region, sum(amount) AS gmv FROM sales GROUP BY ALL;
4
5  -- CREATE TABLE ... LIKE: 只复制结构不要数据
6  CREATE TABLE sales_copy LIKE sales;
7
8  -- CREATE VIEW
9  CREATE OR REPLACE VIEW v_top AS SELECT * FROM region_summary ORDER
10 BY gmv DESC;
11
12 -- 临时对象（连接关闭即消失）
13 CREATE TEMP TABLE tmp AS SELECT 1 AS x;
14 CREATE TEMP VIEW tv AS SELECT * FROM tmp;
15
16 -- 判断存在性（避免报错）
17 CREATE TABLE IF NOT EXISTS t2 (a INTEGER);
18 DROP TABLE IF EXISTS t2;

```

为什么 `CREATE OR REPLACE` 在探索时好用到爆：DDL 变成了可重复执行的脚本，不必每次跑之前手动清理。

4.8 其他小糖果

4.8.1 尾随逗号

```

1  SELECT
2      a,
3      b,          -- ← 允许尾随逗号，方便加列
4  FROM t;

```

4.8.2 INSERT 多行的 VALUES

```

1  INSERT INTO t VALUES
2      (1, 'a'),
3      (2, 'b'),
4      (3, 'c');
5
6  -- VALUES 也可以独立查询（不建表）
7  SELECT * FROM (VALUES (1, 'a'), (2, 'b')) AS v(id, name);
8  -- FROM-FIRST 写法:
9  FROM (VALUES (1, 'a'), (2, 'b')) AS v(id, name);

```

4.8.3 QUALIFY（窗口函数过滤）

标准的 STD SQL 里必须把窗口函数包进子查询才能过滤，DuckDB 支持 `QUALIFY`：

```

1 | SELECT user_id, amount, row_number() OVER (PARTITION BY user_id ORDER
   | BY amount DESC) AS rn
2 | FROM sales
3 | QUALIFY rn = 1;      -- 每个用户金额最大的一笔订单

```

第 10 章详解。

4.8.4 一行求 DRY: `count(*)` vs `count(col)`

```

1 | SELECT count(*) AS rows, count(quantity) AS non_null_qty,
   | count(DISTINCT region) AS regions
2 | FROM sales;

```

`count(*)` 计数所有行 (含 NULL) , `count(col)` 只数非 NULL。这是面试高频考点, 也是无数 bug 的来源 (第 8 章详述)。

4.8.5 表达式取别名后可在 ORDER BY / GROUP BY 使用

```

1 | SELECT
2 |     region,
3 |     sum(amount) AS gmv
4 | FROM sales
5 | GROUP BY region
6 | ORDER BY gmv DESC;      -- 直接用别名, 不必重复写 sum(amount)

```

MySQL/PostgreSQL 也支持 ORDER BY 别名; 但 **GROUP BY/HAVING** 用别名是 DuckDB 的额外宽容 (很多数据库不行)。

4.8.6 布尔 type 更直接

```

1 | SELECT 1=1 AS t, (1>2) AS f, true IS true AS yes;

```

4.8.7 舒服的 `date/interval` 字面量

```

1 | SELECT DATE '2026-01-01', TIMESTAMP '2026-01-01 12:00:00', INTERVAL 1
   | DAY, INTERVAL '3 months';
2 | SELECT DATE '2026-01-01' + INTERVAL 7 DAY AS next_week;

```

第 5 章会展开时间类型。

4.8.8 PIVOT / UNPIVOT

长宽表互转 (版本不同可用性有差异, 较新版本已把它放入了专门的扩展: 详情运行 `.help/` 查文档确认) :

```

1  -- 简化语法：自动探测列
2  PIVOT sales ON status USING sum(amount) GROUP BY region;
3
4  -- UNPIVOT：宽转长
5  UNPIVOT monthly_sales ON jan, feb, mar
6  INTO NAME month VALUE sales;

```

等价的传统写法（逻辑）：

```

1  SELECT region,
2      sum(CASE WHEN status='已支付' THEN amount END) AS "已支付",
3      sum(CASE WHEN status='已发货' THEN amount END) AS "已发货",
4      sum(CASE WHEN status='已完成' THEN amount END) AS "已完成"
5  FROM sales GROUP BY region;

```

如果 PIVOT 报 "syntax error", 说明你的版本把它移到了扩展里, 安装后即可, 或用上面的 CASE WHEN 写法替代 (后者更快、更可控)。

4.9 综合运用：一个"友好 SQL"风格的真实查询

任务：找出每个区域里金额 Top 3 的订单，并把金额按千元展示，同时输出所属客户的套餐。

```

1  FROM sales s
2  JOIN users u USING (user_id)
3  QUALIFY row_number() OVER (PARTITION BY s.region ORDER BY s.amount
4  DESC) <= 3
5  SELECT
6      s.region,
7      s.order_id,
8      u.user_name,
9      u.plan,
10     round(s.amount / 1000, 1) AS k_amount,
11     s.order_date,
12     s.created_at
13 ORDER BY s.region, k_amount DESC;

```

这里用到了：

- FROM-first 写法；
- JOIN ... USING ；
- QUALIFY (窗口过滤) ；
- 表达式 + 别名。

把这些"糖果"组合起来后，SQL 变成一段"声明式的管道"，而不是一堆括号。

实例演练

实例 4-1 · EXCLUDE / REPLACE：不认识全部列也能改表

```
1 SELECT * EXCLUDE (created_at, quantity) FROM sales LIMIT 1;
2
3 SELECT * REPLACE (round(amount) AS amount, lower(status) AS status)
4 FROM sales LIMIT 1;
```

实测 (`.mode line`, 第 2 个查询)：

```
1 order_id = 1
2 user_id = U0583
3 region = 华中
4 category = 食品
5 status = 已完成
6 amount = 851      ← 已被 round 替换 (原为 851.47)
7 quantity = 8
8 order_date = 2026-01-01 00:00:00
9 created_at = 2026-01-01 10:10:00
```

实例 4-2 · COLUMNS：对一批列做同一个运算

```
1 SELECT sum(COLUMNS(lambda c: contains(c, 'amount')))) FROM wide;
```

```
1 |
2 | sum(amount_a) | sum(amount_b) |
3 |
4 | 10.0 | 10.0 |
5 |
```

加 `\0` 占位符给结果改名：

```
1 SELECT sum(COLUMNS(lambda c: contains(c, 'amount')))) AS "total_\0"
FROM wide;
2 -- total_amount_a = 10.0
3 -- total_amount_b = 10.0
```

注意 lambda 的新语法 `lambda c: ...` (旧的 `c -> ...` 在 1.5.5 上会打印弃用警告)。

实例 4-3 · USING SAMPLE：秒看大数据

```
1 SELECT count(*) AS n FROM (SELECT * FROM sales USING SAMPLE 1000);
2 -- 1000
```

20 万行里取 1000 行只用了几毫秒——它是“读前 1000 行”而不是“扫全表再随机”。

实例 4-4 · FROM-first 的正确顺序

```
1 FROM sales
2 SELECT status, round(sum(amount)) AS gmv
3 WHERE region = '华东'
4 GROUP BY status
5 ORDER BY gmv DESC;
```

status	gmv
已完成	5631328
已发货	5622860
已支付	5593529

实例 4-5 · upsert 的三种写法（实测行为）

```
1 CREATE OR REPLACE TABLE dim_product (
2     sku    VARCHAR PRIMARY KEY,
3     price  DECIMAL(10,2),
4     stock  INTEGER
5 );
6 INSERT INTO dim_product VALUES ('A-1', 9.90, 10), ('A-2', 19.90, 5);
7
8 INSERT OR REPLACE INTO dim_product VALUES ('A-1', 12.90, 8); -- 整行
   覆盖
9 INSERT OR IGNORE INTO dim_product VALUES ('A-1', 99.90, 1); -- 冲突
   则跳过
10
11 -- 冲突时按规则更新（可以做累加）
12 INSERT INTO dim_product VALUES ('A-2', 15.00, 3)
13 ON CONFLICT (sku) DO UPDATE
14     SET price = excluded.price,
15         stock = dim_product.stock + excluded.stock;
```

实测结果：

sku	price	stock	说明
A-1	12.90	8	被 REPLACE 覆盖；后面的 INSERT OR IGNORE 未生效
A-2	15.00	8	price 更新为 15, stock = 原 5 + 新 3 = 8

实例 4-6 · UNION BY NAME：列不同也能拼

```
1 SELECT * FROM (SELECT 'x' AS a, 1 AS b)
2 UNION BY NAME
3 SELECT 'y' AS a, 2 AS c;
```

a	b	c
y	NULL	2
x	1	NULL

缺失的列自动补 `NULL`——这在合并"不同版本的同类数据"时极为好用。

4.10 本章小结

语法糖	解决的问题
可选 <code>SELECT</code> / <code>SELECT</code> 后置	让 SQL 从上往下都能读，像管道
<code>* EXCLUDE</code> / <code>REPLACE</code>	不重复列出几十列
<code>COLUMNS(*)</code> + lambda	对一批列做同样的运算
<code>GROUP BY ALL</code> / <code>ORDER BY ALL</code>	不重复抄分组键
<code>USING SAMPLE</code>	快速"瞄一眼"大数据
<code>INSERT BY NAME</code> / <code>UNION BY NAME</code>	结构不同也能对得起
<code>INSERT OR REPLACE</code> / <code>ON CONFLICT</code>	upsert
<code>QUALIFY</code>	窗口函数结果过滤，不用子查询
<code>CREATE OR REPLACE</code>	可重复执行的 DDL
尾随逗号、表达式别名复用	减少低级错误

记住一个判断标准：**当你在写 SQL 时感到"我在重复输入刚才说过的东西"**，DuckDB 多半有一个语法糖可以省掉它。

动手练习

1. 用 `USING SAMPLE 500` 看一眼 `sales` 的数据分布，再用 `USING SAMPLE 0.1%` 比较。
2. 写一句 SQL：给每个分区（region × category）算 GMV 和订单数，`GROUP BY ALL`；再手动写一遍传统 `GROUP BY`，体会区别。
3. 用 `* REPLACE` 把 `sales` 里的 `amount` 四舍五入、`status` 转小写输出。
4. 构造两张列顺序不同的表，用 `INSERT INTO ... BY NAME` 搬运数据。
5. 用 `QUALIFY` 找出每个用户金额最高的两笔订单。

下一章

[第 5 章 数据类型体系](#) —— 类型是所有正确性的起点，也是性能的起点。

第 5 章 数据类型体系

类型是数据库的地基。选错类型会导致三个层面的问题：

1. **正确性**：时间用 VARCHAR 存，排序和比较都可能出错；
2. **性能**：BIGINT 键改成 VARCHAR，连接会从"整数哈希"退化成"字符串哈希"，慢数倍；
3. **存储**：Parquet 里 INT32 和 DOUBLE 的压缩率完全不同。

本章完整梳理 DuckDB 的类型系统，包括它的**嵌套类型**与较新的 `VARIANT` 类型。

5.1 类型全景图

```
1  DuckDB 类型
2  |— 数值
3  |   |— 整数: TINYINT(8) SMALLINT(16) INTEGER(32) BIGINT(64)
      HUGEINT(128)
4  |   |   |— 无符号: UTINYINT USMALLINT UINTEGER UBIGINT UHUGEINT
5  |   |— 浮点: FLOAT(REAL,4) DOUBLE(8)
6  |   |— 定点: DECIMAL/NUMERIC(p,s)
7  |   |— UBIGINT 溢出 → HUGEINT
8  |— 字符
9  |   |— VARCHAR / STRING / TEXT (UTF-8, 动态长度与压缩)
10 |   |— CHAR(n) (等价于 VARCHAR, n 会被忽略)
11 |— 二进制
12 |   |— BLOB / BYTEA / BINARY
13 |— 日期与时间
14 |   |— DATE
15 |   |— TIME
16 |   |— TIME WITH TIME ZONE (TIMETZ)
17 |   |— TIMESTAMP (微秒精度)
18 |   |— TIMESTAMP WITH TIME ZONE (TIMESTAMPTZ)
19 |   |— INTERVAL
20 |— 布尔
21 |   |— BOOLEAN / BOOL
22 |— 半结构化/嵌套
23 |   |— LIST<T>
24 |   |— STRUCT<k T, ...>
25 |   |— MAP<K,V>
26 |   |— UNION<...>
27 |   |— ARRAY / T[n] (定长数组)
28 |   |— JSON / VARIANT
29 |— 枚举 & UUID
30 |   |— ENUM
31 |   |— UUID
32 |— NULL
```

查看当前库里用到的所有类型：

```
1 | SELECT DISTINCT data_type FROM duckdb_types() ORDER BY data_type;
```

5.2 数值类型

5.2.1 整数家族

类型	字节	取值范围	别名
TINYINT	1	-128 ~ 127	INT1, INT8
SMALLINT	2	±32767	INT2, INT16
INTEGER	4	±2.1e9	INT, INT4, INT32, SIGNED
BIGINT	8	±9.2e18	INT8, INT64, LONG
HUGEINT	16	±1.7e38	INT128
UTINYINT	1	0~255	—
USMALLINT	2	0~65535	—
UIINTEGER	4	0~4.29e9	—
UBIGINT	8	0~1.8e19	—
UHUGEINT	16	0~3.4e38	—

HUGEINT 的价值：常年在 Python/R 里被当作 bignum 使用，是精确大整数运算的救星，避免了 JavaScript 风格的精度丢失。

```
1 | SELECT 2::BIGINT, -3::HUGEINT, 18446744073709551615::UBIGINT + 1;
```

注意整数溢出会报错：

```
1 | SELECT 2147483647::INTEGER + 1;
2 | -- Overflow Error: Overflow in addition of INT32
```

若希望在溢出时自动提升到更大类型，可以显式转换源操作数：

```
1 | SELECT 2147483647::BIGINT + 1; -- OK
```

5.2.2 浮点 vs 定点

```
1 | SELECT 0.1::DOUBLE + 0.2::DOUBLE; -- 0.30000000000000004 (二进制浮点误差)
2 | SELECT 0.1::DECIMAL(10,2) + 0.2; -- 0.30 (十进制精确)
```

DECIMAL 是钱的正确选择：

```

1 CREATE TABLE orders (
2     id      BIGINT,
3     price DECIMAL(12, 2),      -- 总共 12 位, 2 位小数
4     tax     DECIMAL(8, 4)
5 );
6
7 INSERT INTO orders VALUES (1, 19.99, 1.2000);
8 SELECT price * quantity FROM ...;

```

性能提示：DECIMAL 的算术比 DOUBLE 慢。在不需要精确十进制（例如科学计算、机器学习特征）时用 DOUBLE 更快；涉及金额、税率、价格务必用 DECIMAL。

DuckDB 的 DECIMAL 支持到 `DECIMAL(38, x)`（内部从 small/int64/int128 自动选择存储宽度）。

5.2.3 数值特殊值

```

1 SELECT 'nan'::DOUBLE AS nan, 'inf'::DOUBLE AS inf, '-inf'::DOUBLE AS
   ninf;
2 SELECT isnan('nan'::DOUBLE), isinf('inf'::DOUBLE), isfinite(1.0);

```

聚合函数遇到 NaN 与 NULL 的行为不同：`sum()` 忽略 NULL，但 NaN 会污染结果。用 `sum(CASE WHEN NOT isnan(x) THEN x END)` 处理。

5.3 字符串类型

5.3.1 VARCHAR 是唯一选项（但不必纠结）

DuckDB 里 `VARCHAR`、`STRING`、`TEXT` 是同一个东西，`CHAR(n)` 的 `n` 会被忽略：

```

1 SELECT 'abc'::VARCHAR, 'abc'::STRING, 'abc'::TEXT, 'abc'::CHAR(10);

```

DuckDB 内部对字符串做了优化：

- **短字符串内联：**≤ 12 字节的字符串直接存在列向量里，不需要指针跳转；
- **常量压缩/字典压缩：**重复值多的列可以极大压缩。

这解释了为什么 DuckDB 里 VARCHAR 的性能常常比想象的好。但仍然：如果一列只有几个取值，`ENUM` 更省；如果需要做高频 JOIN，考虑改成 INTEGER 代理键。

5.3.2 常用字符串函数速览

```

1 SELECT
2     length(' DuckDB ') AS len,      -- 8 (含空格)
3     upper('duck')      AS upper,    -- DUCK
4     lower('DUCK')      AS lower,    -- duck
5     trim(' DuckDB ')   AS trimmed,  -- DuckDB
6     lpad('7', 3, '0')  AS padded,   -- 007
7     substr('abcdef', 2, 3) AS sub,    -- bcd

```

```
8 concat('a', '-', 'b') AS cat, -- a-b
9 string_split('a,b,c', ',') AS parts, -- ['a','b','c']
10 strpos('abc', 'b') AS pos, -- 2
11 replace('a-c', '-', '_') AS rep, -- a_c
12 regexp_extract('x-42', '(\d+)', 1) AS num, -- 42
13 starts_with('duckdb', 'duck') AS sw, -- true
14 contains('duckdb', 'db') AS cont, -- true
15 printf('%05.2f', 3.14159) AS fmt; -- 03.14
```

注意: `substr`/`substring` 在 DuckDB 中是 **1-based** 索引（与 PostgreSQL 一致），不同于某些语言的 0-based。

5.3.3 Unicode 处理与 ICU 扩展

默认字符串函数按字节/码点处理。需要“按语言习惯”排序、大小写转换（如德语 ß、土耳其语 i）时，加载 ICU 扩展：

```
1 INSTALL icu; LOAD icu;
2 SELECT lower('istanbul' COLLATE tr) AS tr_lower; -- 土耳其语规则
3 SELECT * FROM t ORDER BY name COLLATE NOCASE; -- 不区分大小写排序
```

DuckDB 1.5 起对 UTF-8 处理做了大量内部重构，即使不加载 ICU 也能正确处理多字节字符（emoji、中文等）。

5.3.4 BLOB 二进制

```
1 SELECT 'hello'::BLOB AS b, encode('hello') AS b64, decode('aGVsbG8=')
   AS raw;
2 SELECT octet_length('\x00\x01'::BLOB) AS nbytes;
```

BLOB 用于存 protobuf、图像的原始字节、pickle 数据等。**不建议**用来存大数据量对象（数据库会变大且扫描慢）。

5.4 时间类型（重点，坑最多）

DuckDB 的时间类型完全对齐 PostgreSQL，但多了一些细节。

类型	含义	精度	例子
DATE	年-月-日	天	2026-09-17
TIME	一天中的时刻	微秒	14:30:00.123456
TIMETZ	时刻 + 时区偏移	微秒	14:30:00+08
TIMESTAMP	日期 + 时刻（ 无时区 ）	微秒	2026-09-17 14:30:00
TIMESTAMPZ	瞬时时间点（ 有时区 ）	微秒	内部 UTC + 显示时区

类型	含义	精度	例子
<code>INTERVAL</code>	时间间隔	月/天/微秒 三元组	<code>INTERVAL 1 MONTH</code>

5.4.1 TIMESTAMP vs TIMESTAMPTZ：本质区别

```

1 | SELECT '2026-01-01 12:00:00'::TIMESTAMP;      -- 墙上时钟："某地现在是 12
   | 点", 但不知道是哪地
2 | SELECT '2026-01-01 12:00:00+00'::TIMESTAMPTZ; -- 瞬时点：UTC 时间 12 点

```

选用原则：

场景	用哪个
业务发生时间（订单、日志、事件）	<code>TIMESTAMPTZ</code>
没有时区语义的业务时间（例如"排班表上的 09:00"）	<code>TIMESTAMP</code>
只需要日期粒度（例如"注册日"）	<code>DATE</code>

真实世界里 90% 的时间字段应该用 `TIMESTAMPTZ`；用 `TIMESTAMP` 的常见问题是在跨时区分析、夏令时/时区切换时出现偏移。

5.4.2 时区与显示

```

1 | -- 查看/设置会话时区
2 | SELECT current_setting('TimeZone');
3 | SET TimeZone = 'Asia/Shanghai';
4 | SET TimeZone = 'UTC';
5 |
6 | -- 转换
7 | SELECT '2026-01-01 12:00:00+00'::TIMESTAMPTZ AT TIME ZONE
   | 'Asia/Shanghai';

```

`session时区` 只影响 `TIMESTAMPTZ` 的显示，不改变其内部存储的 UTC 值。

5.4.3 字符串→时间：解析

```

1 | SELECT
2 |     strptime('2026/09/17', '%Y/%m/%d')           AS d,
3 |     try_strptime('not-a-date', '%Y-%m-%d')       AS d_bad, --
   | NULL 而不报错
4 |     '2026-09-17'::DATE                           AS cast1,
5 |     '2026-09-17T14:30:00Z'::TIMESTAMPTZ          AS cast2;
6 |
7 | -- dd/mm 与 mm/dd 的歧义: strptime 更可靠
8 | SELECT strptime('03/04/2026', '%d/%m/%Y') AS eu_style; -- 2026-04-03

```

`strftime` 的常用格式符:

符	含义	符	含义
<code>%Y</code>	四位年	<code>%H</code>	24 小时
<code>%y</code>	两位年	<code>%I</code>	12 小时
<code>%m</code>	月	<code>%M</code>	分
<code>%d</code>	日	<code>%S</code>	秒
<code>%j</code>	一年中的第几天	<code>%f / %.f</code>	微秒小数部分
<code>%z</code>	时区偏移 (+0800)	<code>%Z</code>	时区名
<code>%p</code>	AM/PM	<code>%U / %W</code>	周数
<code>%A / %a</code>	星期名 (全/简)	<code>%B / %b</code>	月份名

5.4.4 时间→字符串: 格式化

```
1 SELECT strftime(now(), '%Y年%m月%d日 %H:%M:%S') AS cn_fmt,
2         strftime(now(), '%A') AS weekday,
3         DATE '2026-01-01' + INTERVAL 30 DAY AS plus30,
4         age(TIMESTAMP '2026-12-01', TIMESTAMP '2026-01-01') AS diff;
```

5.4.5 提取与时间维度

```
1 SELECT
2     date_part('year', now()) AS y,           -- 或 extract(year from
3     date_part('month', now()) AS m,          now())
4     date_part('dow', now()) AS dow,         -- 0=周日 (PostgreSQL 语
5     date_part('isoyear', now()) AS isoyear, 义)
6     date_part('epoch', now()) AS unix_ts,   -- 秒
7     date_trunc('hour', now()) AS hour,      -- 截断到小时
8     date_trunc('week', now()) AS week_start,
9     last_day(now()) AS month_end;
```

实践建议: 做按小时/天的报表时, 用 `date_trunc('hour', ts)` 生成"时间桶", 比 `GROUP BY EXTRACT(...)` 快且容易写出多粒度。

5.4.6 INTERVAL 的三个分量

`INTERVAL` 内部是三个字段: **months** (月)、**days** (天)、**micros** (微秒)。它们不会自动互换!

```

1  -- ✅ 已实测 (1.5.5)：相等比较会按绝对时长归一化，结果是 TRUE
2  SELECT INTERVAL 1 MONTH = INTERVAL 30 DAY;      -- true
3  SELECT INTERVAL 1 MONTH = INTERVAL 31 DAY;      -- false
4  SELECT INTERVAL 1 YEAR  = INTERVAL 12 MONTH;    -- true
5  SELECT INTERVAL 1 DAY   = INTERVAL 24 HOURS;    -- true
6
7  -- ⚠️ 但是"加到日期上"的语义不同！
8  SELECT DATE '2026-02-01' + INTERVAL 1 MONTH;    -- 2026-03-01 (按日历月)
9  SELECT DATE '2026-02-01' + INTERVAL 30 DAY;     -- 2026-03-03 (按天数)

```

关键区别：

- **比较时**，DuckDB 把月按 30 天、年按 365 天折算成绝对时长，所以 `1 MONTH = 30 DAY` 为 **true**；
- **运算时**，`+ INTERVAL 1 MONTH` 走**日历语义**（下个月同一天），`+ INTERVAL 30 DAY` 走**天数语义**。

所以：想表达"30 天"就写 `INTERVAL 30 DAY`，想表达"下个月同一天"就写 `INTERVAL 1 MONTH`，两者的结果可能相差一到两天（如上例的 3-01 vs 3-03）。

常用写法：

```

1  SELECT now() - INTERVAL 7 DAY      AS week_ago,
2         now() + INTERVAL 3 HOUR    AS later,
3         INTERVAL (x) DAY           AS dyn,    -- <- 变量相乘写法
4         to_days(30)                AS i2;     --

```

`to_days/to_months/to_hours` 辅助函数

5.4.7 生成时间序列

```

1  SELECT * FROM generate_series(DATE '2026-01-01', DATE '2026-01-10',
2                                INTERVAL 1 DAY) AS t(d);
3  -- 等价于：
4  SELECT unnest(generate_series(DATE '2026-01-01', DATE '2026-01-10',
5                                INTERVAL 1 DAY)) AS d;

```

补缺日期（日历维度表）的经典用法：

```

1  WITH cal AS (
2      SELECT unnest(generate_series(DATE '2026-01-01', DATE '2026-06-
3          30', INTERVAL 1 DAY)) AS d
4  ),
5  daily AS (
6      SELECT order_date, sum(amount) AS gmv FROM sales GROUP BY 1
7  )
8  SELECT cal.d, coalesce(daily.gmv, 0) AS gmv
9  FROM cal LEFT JOIN daily ON cal.d = daily.order_date
10 ORDER BY cal.d;

```

5.4.8 常见时间坑汇总

坑	现象	解决
时区错乱	数据相差 8 小时	统一用 <code>TIMESTAMP TZ</code> ，并显式设置 <code>SET TimeZone</code>
字符串日期排序错误	<code>'2026-1-2' < '2026-1-10'</code> 为 false（字典序）	用 <code>DATE</code> 类型，或 <code>lpad</code> 补零
夏令时/闰秒	重复/缺失小时	<code>TIMESTAMP TZ</code> 已内部处理；避免用 <code>TIMESTAMP</code> 存 UTC
解析失败中断整批导入	一条脏数据导致整个 COPY 失败	用 <code>try_strptime</code> / <code>try_cast</code>
<code>BETWEEN</code> 边界	漏掉当天最后几秒	用半开区间 <code>>= d AND < d+1</code>

5.5 BOOLEAN 与 NULL

```
1 SELECT true, false, NULL::BOOLEAN;
2 SELECT NOT true AS n, true AND false AS a, true OR NULL AS o;
```

5.5.1 三值逻辑（务必理解）

表达式	结果
<code>NULL = NULL</code>	NULL （不是 TRUE！）
<code>NULL IS NULL</code>	TRUE
<code>NULL IS NOT NULL</code>	FALSE
<code>'x' = NULL</code>	NULL
<code>NULL AND true</code>	NULL
<code>NULL AND false</code>	FALSE
<code>NULL OR true</code>	TRUE
<code>NOT NULL</code>	NULL

后果： `WHERE col <> 'abc'` **不会返回** `col IS NULL` **的行**（因为结果是 NULL，被 WHERE 过滤掉）。

处理 NULL 的函数：

```

1 SELECT coalesce(NULL, 0)           AS c,    -- 0
2     ifnull(NULL, 'n/a')           AS i,    -- n/a (coalesce 的两参数简
   写)
3     nullif(a, b)                   AS n;    -- a=b 时返回 NULL

```

更多 NULL 语义陷阱见第 7 章。

5.6 ENUM 与 UUID

5.6.1 ENUM

```

1 CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
2 CREATE TABLE person (name VARCHAR, current_mood mood);
3 INSERT INTO person VALUES ('Alice', 'happy');
4 -- INSERT INTO person VALUES ('Bob', 'angry'); -- 报错
5
6 SELECT enum_range(NULL::mood);           -- 列出所有取值
7 SELECT enum_first(NULL::mood), enum_last(NULL::mood);

```

优点：存储更省（内部 UINTEGER 编码）、防止脏值。

缺点：加一个新取值需要 `ALTER TYPE mood ADD VALUE 'excited';` (DDL)。

什么时候不用 ENUM：当取值会频繁变化时（如业务状态码），用 VARCHAR 反而更好维护。反过来，取值稳定且重复率高时（如星期、状态），ENUM/维度表更优。

5.6.2 UUID

```

1 SELECT uuid() AS generated, 'a0eebc99-9c0b-4ef8-bb6d-
   6bb9bd380a11'::UUID AS u;

```

UUID 存为 128 位，比 VARCHAR(36) 省一半以上空间。

5.7 嵌套类型（预览）

嵌套类型是 DuckDB 的王牌功能，第 11 章专门讲。这里先给一个直观印象：

```

1 SELECT
2     [1, 2, 3]                               AS a_list,
3     {'name': 'Alice', 'age': 30}             AS a_struct,
4     MAP {'a': 1, 'b': 2}                     AS a_map,
5     unnest([10, 20, 30])                     AS flattened;

```

```

1 SELECT struct_extract({'x':1,'y':2}, 'x') AS x;    -- 1
2 SELECT ([ 'x':1, 'y':2 ]).y;                  -- 2 (点语法)
3 SELECT list_transform([1,2,3], lambda x: x * 2) AS doubled; --
   [2,4,6]
4 SELECT list_filter([1,2,3,4], lambda x: x % 2 = 0) AS evens; -- [2,4]

```

5.8 JSON 与 VARIANT (1.5 重点)

5.8.1 JSON 类型

'{"a":1}':::JSON 与 '{"a":1}':::VARCHAR 的区别:

- JSON 类型会被**解析并校验**, 存储为二进制格式;
- 提取值时不需要每次重新 parse, 因此**快得多**;
- 支持路径提取语运算符 `->`、`->>`。

```
1 SELECT '{"name":"Alice","tags":["a","b"]}'::JSON AS j;  
2 SELECT j->>'name' FROM (SELECT '{"...}'::JSON AS j);    -- 提取为字符串
```

JSON 类型的引入改变了常见工作流: 以前日志入库要么存 TEXT 每次 parse, 要么 ETL 成列建模; 现在可以 "存 JSON + 建投影列"。

5.8.2 VARIANT 类型 (1.5 新增)

VARIANT 是一种**自描述的混合类型**: 它的每个值可以是标量、数组或对象, 且可以混合嵌套。相比 JSON 类型的优势:

- 更紧凑的二进制表示;
- 更高效的内部结构体布局, 字段访问更快;
- 支持**自动类型推断的读取**: 嵌套字段可以直接用点语法取出并参与运算。

```
1 SELECT '{"user":{"id":1},"items":[{"sku":"A","qty":2}]}'::VARIANT AS  
   v;  
2 SELECT v.user.id, unnest(v.items).sku FROM t;
```

具体语法请以你所用版本的官方文档为准; 如果你的版本不支持 VARIANT, 用 JSON 类型 + `json_extract` 同样可行 (第 11 章详述)。

5.8.3 JSON 函数概览

```
1 SELECT  
2     json_extract(j, '$.a')           AS val,      -- 或 j->'a'  
3     json_extract_string(j, '$.a')    AS str,      -- 或 j->>'a'  
4     json_valid('{"a":1}')            AS ok,  
5     json_type('[1,2]')               AS kind,  
6     json_keys('{"a":1,"b":2}')        AS keys,  
7     json_array_length('[1,2,3]')     AS n,  
8     to_json({'a': 1})                 AS gen,  
9     json_group_array(name)           AS agg;      -- 聚合为 JSON 数组
```

5.9 类型转换

5.9.1 CAST

```
1 SELECT CAST('123' AS INTEGER);           -- 标准写法
2 SELECT '123'::INTEGER;                   --
   DuckDB/PostgreSQL 简写
3 SELECT try_cast('abc' AS INTEGER);       -- NULL 而不报错 →
   NULL
4 SELECT coalesce(try_cast('abc' AS INTEGER), -1); -- 失败时用默认值 →
   -1
```

⚠ **已实测 (DuckDB 1.5.5)** : DuckDB **不支持**某些数据库里的 `TRY_CAST(x AS T DEFAULT d)` 语法, 会报语法错误。想要"失败兜底"请一律用 `coalesce(try_cast(x AS T), 默认值)`。

导入脏数据时的黄金法则: **能用 `try_cast` 就不用 `cast`**, 避免一条脏数据毁掉整批导入。

5.9.2 隐式转换规则

DuckDB 会自动做一些安全的隐式转换 (例如 `INTEGER` → `BIGINT`、`INTEGER` → `DOUBLE`) , 但不会做有损/有歧义的隐式转换 (如 `VARCHAR` → `DATE`) 。需要时请显式 `CAST`。

检查表达式类型:

```
1 SELECT typeof(1 + 1.0) AS t;           -- DOUBLE
2 SELECT typeof([1,2]) AS t2;           -- INTEGER[]
```

`typeof()` 是调试类型的第一工具。

5.10 与 Python/Pandas/Parquet 的类型对照

DuckDB	Python	pandas (numpy)	Parquet
BOOLEAN	bool	bool	BOOLEAN
TINYINT ... BIGINT	int	int8/16/32/64	INT8/16/32/64
HUGEINT	int	object	FIXED_LEN_BYTE_ARRAY
UBIGINT 等 unsigned	int	uint*	INT(bit-width, unsigned)
FLOAT	float	float32	FLOAT
DOUBLE	float	float64	DOUBLE
DECIMAL(p,s)	Decimal	object	DECIMAL / INT32/64
VARCHAR	str	object	BYTE_ARRAY (UTF8)

DuckDB	Python	pandas (numpy)	Parquet
BLOB	bytes	object	BYTE_ARRAY
DATE	datetime.date	datetime64[ns]	DATE
TIME	datetime.time	object	TIME
TIMESTAMP	datetime.datetime	datetime64[ns]	TIMESTAMP (isAdjustedToUTC=false)
TIMESTAMPTZ	datetime (tz-aware)	datetime64[ns, tz]	TIMESTAMP (isAdjustedToUTC=true)
INTERVAL	timedelta	timedelta64	—
UUID	uuid.UUID	object	FIXED_LEN_BYTE_ARRAY(16)
LIST	list	object / Arrow list	LIST
STRUCT	dict	dict-like	STRUCT
MAP	dict	dict	MAP

5.11 类型选择最佳实践

1. **主键/外键用 INTEGER/BIGINT**，别用 VARCHAR（连接性能差距巨大）；
2. **金额用 DECIMAL**，不要 DOUBLE；
3. **时间用 TIMESTAMPTZ**，除非你有明确理由不用；
4. **小整数**：状态/标志 $\in \{0,1\}$ 用 `BOOLEAN`；人数 ≤ 100 用 `TINYINT / SMALLINT`（虽然 DuckDB 会做内部优化，但显式声明让意图更清晰，Parquet 里也更省）；
5. **高基数的重复字符串**：考虑 ENUM 或维表 + 整数代理键；
6. **半结构化数据**：优先拆成真列；实在不能拆、且 schema 频繁变化，再用 JSON/VARIANT；
7. **导入用 try_cast**：宁可得到 NULL，也不要让整批任务失败；
8. **别随便保留 15 位小数**：ROUND / DECIMAL 精度越小越省。

实例演练

实例 5-1 · 金额到底该用 DOUBLE 还是 DECIMAL

```
1 | SELECT 0.1::DOUBLE + 0.2::DOUBLE      AS float_sum,  
2 |        (0.1::DECIMAL(10,2) + 0.2)     AS dec_sum;
```

float_sum	dec_sum
-----------	---------

float_sum	dec_sum
0.300000000000000004	0.30

订单金额、税率、价格一律用 `DECIMAL`；只有科学计算/机器学习特征才用 `DOUBLE`。

实例 5-2 · 脏数据的安全转换

```
1 SELECT try_cast('abc' AS INTEGER) AS failed,
2    coalesce(try_cast('abc' AS INTEGER), -1) AS fallback;
```

failed	fallback
NULL	-1

前面提醒过：DuckDB 不支持 `try_cast(x AS INT DEFAULT -1)`，用 `coalesce` 兜底。

实例 5-3 · 时区只影响"显示"，不影响"存储"

```
1 SET TimeZone = 'UTC';
2 SELECT TIMESTAMP '2026-01-01 12:00:00'::TIMESTAMPTZ AS utc;
3
4 SET TimeZone = 'Asia/Shanghai';
5 SELECT TIMESTAMP '2026-01-01 12:00:00'::TIMESTAMPTZ AS sh;
```

utc	sh
2026-01-01 12:00:00+00	2026-01-01 12:00:00+08

注意：这里两个值**不是同一时刻**（因为没有写偏移，两个字面量各自被解释成本地时间 12:00）。

真正"同一时刻不同显示"要这么写：

```
1 SET TimeZone='UTC';      SELECT TIMESTAMP '2026-01-01
12:00:00+00'::TIMESTAMPTZ; -- 12:00+00
2 SET TimeZone='Asia/Shanghai'; SELECT TIMESTAMP '2026-01-01
12:00:00+00'::TIMESTAMPTZ; -- 20:00+08
```

实例 5-4 · INTERVAL 的"比较"与"运算"是两套语义

```
1 SELECT INTERVAL 1 MONTH = INTERVAL 30 DAY AS a, -- true
2    INTERVAL 1 MONTH = INTERVAL 31 DAY AS b; -- false
3
4 SELECT DATE '2026-02-01' + INTERVAL 1 MONTH AS m1, -- 2026-03-01（日
   历）
5    DATE '2026-02-01' + INTERVAL 30 DAY AS m2; -- 2026-03-03（天
   数）
```

a	b	m1	m2
true	false	2026-03-01	2026-03-03

实例 5-5 · JSON 提取的两种写法等价

```

1 SELECT j->>'user' AS u,
2        json_extract_string(j, '$.user') AS fn
3 FROM (SELECT '{"user":{"name":"Alice"}}'::JSON AS j);

```

5.12 本章小结

- DuckDB 的类型体系 = PostgreSQL 风格 + **丰富的嵌套类型** + JSON/VARIANT;
- 整数有符号/无符号/128 位三条产品线；金额用 DECIMAL；
- `TIMESTAMP` 无时区、`TIMESTAMPTZ` 存瞬时 UTC，**绝大多数业务用 TIMESTAMPTZ**；
- INTERVAL 的三分量（月/天/微秒）不会互相转换，这是特性不是 bug；
- 用 `try_cast` 应对脏数据，`typeof()` 调试类型；
- 半结构化数据优先考虑列化，其次才用 JSON/VARIANT。

动手练习

1. 把 `sales.created_at` 转成三种类型分别查询，感受 `TIMESTAMP` 与 `TIMESTAMPTZ` 的输出差异：

```

1 SET TimeZone='UTC';
2 SELECT created_at::TIMESTAMP, created_at::TIMESTAMPTZ FROM sales
   LIMIT 3;

```

2. 写一个按周的报表： `date_trunc('week', order_date) + SUM(amount)`。
3. 使用 `generate_series` 生成 2026 年全年日历，与 `sales` 做 LEFT JOIN，找出没有销售额的日期。
4. 用 `try_cast` 演示：包含 `'abc'` 的列表转 INTEGER 不报错。
5. 创建 `CREATE TYPE order_status AS ENUM (...)`，用它建一张表，插入非法值观察报错。

下一章

[第 6 章 表、视图与模式管理](#) —— 怎么把数据结构设计得既规范又高效。

第 6 章 表、视图与模式管理

如果说 SQL 是一座城市，那 DDL（数据定义语言）就是城市规划图。本章讨论如何把数据组织得**规范、清晰、高效**。

内容涵盖：CREATE/ALTER/DROP、主键与各种约束、模式（schema）的使用、临时表、视图、以及 DuckDB 里“为什么不建索引”这个经典疑问。

6.1 创建表

6.1.1 基本语法

```
1 CREATE TABLE employees (  
2     id          BIGINT,  
3     name        VARCHAR,  
4     department  VARCHAR,  
5     salary      DECIMAL(12,2),  
6     hired_at    DATE,  
7     is_active   BOOLEAN  
8 );
```

带上约束的完整版本：

```
1 CREATE TABLE employees (  
2     id          BIGINT,  
3     name        VARCHAR NOT NULL,  
4     email       VARCHAR UNIQUE,  
5     department  VARCHAR DEFAULT '未分配',  
6     salary      DECIMAL(12,2) CHECK (salary >= 0),  
7     hired_at    DATE DEFAULT current_date,  
8     is_active   BOOLEAN DEFAULT true,  
9     PRIMARY KEY (id)  
10 );
```

6.1.2 从查询创建 (CTAS)

这是 DuckDB 里**最常用的建表方式**，因为 OLAP 场景下数据通常从别处来：

```
1 -- CREATE TABLE AS SELECT  
2 CREATE TABLE dept_summary AS  
3 SELECT department, count(*) AS headcount, avg(salary) AS avg_sal  
4 FROM employees  
5 GROUP BY ALL;  
6  
7 -- 可重复执行  
8 CREATE OR REPLACE TABLE dept_summary AS SELECT ...;  
9  
10 -- 存在则跳过，不存在才建  
11 CREATE TABLE IF NOT EXISTS dept_summary AS SELECT ...;
```

```

12
13  -- 只要结构不要数据
14  CREATE TABLE employees_copy LIKE employees;

```

重要：CTAS 出来的表**不会继承源表的约束与主键**。如果你希望新表有主键，必须事后 `ALTER TABLE ... ADD PRIMARY KEY` 或在 CREATE 里显式声明。

6.1.3 临时表

```

1  CREATE TEMP TABLE scratch AS SELECT * FROM sales LIMIT 1000;
2  -- 或 CREATE TEMPORARY TABLE

```

特点：

- 属于当前**连接 (connection)**，连接关闭即自动删除；
- 不写入 WAL，速度快；
- 适合中间结果、调试。

6.1.4 直接从文件建表（其实通常不需要）

```

1  CREATE TABLE sales AS SELECT * FROM read_parquet('sales.parquet');

```

但请记住第 1 章的核心思想：**读文件不一定要“入库”**。你完全可以每都直接 `FROM read_parquet('sales.parquet')`。只有当同一份数据会被反复查询、或者需要做多次增量写入时，才值得物化成表。

6.2 约束：哪些有意义，哪些只是“文档”

DuckDB 支持的约束：

约束	支持	说明
PRIMARY KEY	✓	唯一 + 非空； 用于 upsert 冲突检测 ，但不是物理索引
UNIQUE	✓	唯一性；同样用于 upsert
NOT NULL	✓	强制非空
CHECK	✓	表达式校验，插入/更新时检查
FOREIGN KEY	✓	支持，但 DuckDB 里 主要用于语义声明 ，且会带来写入开销

6.2.1 主键的真实作用

```

1 CREATE TABLE product (
2     sku      VARCHAR PRIMARY KEY,
3     price    DECIMAL(10,2)
4 );
5
6 -- 唯一性会被强制执行
7 INSERT INTO product VALUES ('A-1', 9.9);
8 INSERT INTO product VALUES ('A-1', 8.8);
9 -- Constraint Error: Duplicate key "sku: A-1" violates primary key
   constraint

```

关键点：

1. 唯一性会被真正检查；
2. 但 DuckDB **不会**为它自动创建 B-Tree 索引（见 6.6 节）；
3. 它的第二个作用是为 `ON CONFLICT (sku) DO UPDATE` 提供冲突判定依据。

6.2.2 CHECK 与 NOT NULL

```

1 CREATE TABLE orders (
2     amount DECIMAL(10,2) CHECK (amount > 0),
3     qty     INTEGER      CHECK (qty BETWEEN 1 AND 1000),
4     status  VARCHAR      NOT NULL
5 );
6
7 INSERT INTO orders VALUES (-1, 1, 'x');
8 -- Constraint Error: CHECK constraint failed: orders

```

性能权衡：CHECK 约束在每次插入时都要求值。对于导入海量数据的原始表，建议**先不建约束**，在清洗后的结果表上再约束（或者用 ETL 时显式过滤）。

6.2.3 复合主键

```

1 CREATE TABLE daily_sales (
2     dt      DATE,
3     region  VARCHAR,
4     gmv     DECIMAL(14,2),
5     PRIMARY KEY (dt, region)
6 );

```

在 OLAP 里，复合主键通常表达的是“**粒度** (grain)”这个概念——这张表一行代表一天一个区域。

6.3 修改表结构

```

1 ALTER TABLE employees ADD COLUMN manager_id BIGINT;
2 ALTER TABLE employees ADD COLUMN IF NOT EXISTS level INTEGER DEFAULT
  1;
3
4 ALTER TABLE employees DROP COLUMN manager_id;
5 ALTER TABLE employees DROP COLUMN IF EXISTS level;
6
7 ALTER TABLE employees RENAME COLUMN level TO emp_level;
8 ALTER TABLE employees RENAME TO staff;
9
10 -- 改类型（可能失败，需谨慎）
11 ALTER TABLE employees ALTER COLUMN salary SET DATA TYPE
  DECIMAL(14,2);
12 ALTER TABLE employees ALTER COLUMN department SET DEFAULT '未知';
13 ALTER TABLE employees ALTER COLUMN name DROP DEFAULT;

```

大表改结构的代价：DuckDB 是列存的。`ADD COLUMN` 基本是元数据操作（几乎瞬时）；但 `DROP COLUMN` / 改类型 在老数据上可能触发**整个表的重写**。生产上遇到大表改结构的稳妥做法是 CTAS 到新表再重命名：

```

1 CREATE OR REPLACE TABLE employees_new AS
2 SELECT id, name, salary, department FROM employees;
3 DROP TABLE employees;
4 ALTER TABLE employees_new RENAME TO employees;

```

6.4 删除与清空

```

1 DROP TABLE IF EXISTS tmp_table;
2 DROP VIEW IF EXISTS v_top;
3 DROP TABLE IF EXISTS t CASCADE;      -- 同时删掉依赖它的视图
4
5 DELETE FROM sales WHERE order_date < DATE '2026-02-01';  -- 支持条件
  删除
6 TRUNCATE TABLE staging;                -- 清空表，
  不可回滚（DDL）
7
8 -- 想要"可回滚的清空"：
9 BEGIN TRANSACTION;
10 DELETE FROM staging;
11 -- 检查后 COMMIT，否则 ROLLBACK
12 COMMIT;

```

`TRUNCATE` 属于 DDL，在某些版本中会隐式提交事务，请留意。

6.5 模式（Schema）：库内的命名空间

随着表变多，把所有表平铺会很混乱。DuckDB 支持标准 schema：

```

1 CREATE SCHEMA IF NOT EXISTS staging;
2 CREATE SCHEMA IF NOT EXISTS mart;
3
4 -- 在指定 schema 建表
5 CREATE TABLE staging.raw_events (id BIGINT, payload VARCHAR);
6 CREATE SCHEMA mart;
7 CREATE TABLE mart.daily_active (d DATE, dau BIGINT);
8
9 -- 访问
10 SELECT * FROM staging.raw_events;
11 SELECT * FROM mart.daily_active;
12
13 -- 设置默认搜索路径（不必每次写前缀）
14 SET search_path = 'staging';
15 SELECT * FROM raw_events;    -- 会解析到 staging.raw_events
16
17 -- 移动/重命名
18 ALTER TABLE staging.raw_events RENAME TO mart.raw_events; -- 跨
    schema 移动（视版本支持）
19 DROP SCHEMA IF EXISTS staging CASCADE;

```

元数据查询：

```

1 SHOW SCHEMAS;                                -- 或 SELECT * FROM
    duckdb_schemas();
2 SELECT * FROM duckdb_tables();
3 SELECT * FROM duckdb_views();
4 SELECT * FROM duckdb_columns();

```

6.5.1 推荐的 schema 分层

数据仓库经典分层，在 DuckDB 里完全适用：

schema	用途	内容
raw / landing	原始落地	未经处理的 Parquet/CSV 映射视图
staging	清洗层	类型转换、重命名、去重后的中间表
intermediate	中间层（可选）	复杂业务逻辑的中间结果
mart	应用层	面向报表/API 的宽表

一条典型的 pipeline：

```

1 CREATE OR REPLACE VIEW raw.events AS
2 SELECT * FROM read_parquet('events/*.parquet', union_by_name =
    true);
3
4 CREATE OR REPLACE TABLE staging.events AS
5 SELECT
6     try_cast(user_id AS BIGINT) AS user_id,

```

```

7      lower(event_type)                AS event_type,
8      try_cast(ts AS TIMESTAMPTZ)      AS event_time,
9      json_extract_string(payload, '$.page') AS page
10 FROM raw.events
11 WHERE try_cast(ts AS TIMESTAMPTZ) IS NOT NULL;
12
13 CREATE OR REPLACE TABLE mart.daily_metrics AS
14 SELECT
15     event_time::DATE AS d,
16     count(DISTINCT user_id) AS dau,
17     count(*) AS events
18 FROM staging.events
19 GROUP BY 1
20 ORDER BY 1;

```

6.6 索引：DuckDB 的特殊答案

这是很多从 PostgreSQL/MySQL 转过来的用户最困惑的问题。

6.6.1 DuckDB 默认不用传统 B-Tree 索引

原因：

- OLAP 查询几乎都是**全列扫描 + 聚合**，不是点查；
- 列存 + 行组 + **Zone Map (每个行组的 min/max 统计)** 天然起到"跳过无关数据"的作用；
- 真的需要点查时，全表扫描也可能比 B-Tree 更快（因为列存压缩率高、可以向量化）。

尝试创建：

```

1 CREATE INDEX idx ON sales(amount);
2 -- 部分版本会报 "Not implemented" 或明确告知该 API 是实验性的

```

6.6.2 你真正应该用的"加速器"

需求	方案
按某列过滤很慢	让数据按该列 排序存储 （用 <code>COPY ... TO</code> 后 <code>ORDER BY</code> ），Zone Map 生效
大量重复值列过滤	该列用 ENUM 或字典编码紧凑存储
分区裁剪	用 Hive 分区目录 （第 12 章），按日期分区
精确点查为主	考虑加载到内存后用 appender/其它结构；或换 OLTP 数据库
复杂条件多行过滤 + 需要精确索引	DuckDB 有实验性的 ART 索引/全文索引（fts 扩展）

最重要的实践：用数据布局代替索引。把经常过滤的列（如日期）作为**排序键**写入 Parquet 或 DuckDB 表，让 Zone Map 能跳过 90% 的行组。

示例：

```
1  -- 建立一个"按日期有序"的结果表，此后按日期范围过滤会非常快
2  CREATE OR REPLACE TABLE sales_sorted AS
3  SELECT * FROM sales ORDER BY order_date;
```

6.7 视图 (View)

6.7.1 普通视图

```
1  CREATE VIEW v_region_gmv AS
2  SELECT region, sum(amount) AS gmv, count(*) AS orders
3  FROM sales
4  GROUP BY region;
5
6  SELECT * FROM v_region_gmv ORDER BY gmv DESC;
```

特性：

- **不存储数据**，每次查询时执行底层 SQL；
- 自动"看到"基表的最新数据；
- 可以像表一样被 JOIN、过滤。

6.7.2 什么时候用视图 vs 物化视图/表

需求	选择
总是要最新数据、每次计算代价可接受	视图
计算很贵、希望结果缓存	表 (CTAS) ， 或 DuckDB 的实验性物化视图
只想封装一段 SQL 供多处复用	视图
想给 BI 工具一个稳定的表接口	表 (由定时 ETL 重建)

DuckDB 也支持 PostgreSQL 风格的物化视图（视版本）：

```
1  CREATE MATERIALIZED VIEW IF NOT EXISTS mv_region AS
2  SELECT region, sum(amount) AS gmv FROM sales GROUP BY region;
3
4  REFRESH MATERIALIZED VIEW mv_region;
5  DROP MATERIALIZED VIEW IF EXISTS mv_region;
```

如果版本不支持，用 `CREATE OR REPLACE TABLE ... AS SELECT` 手动刷新即可。

6.7.3 视图的注意事项

- 视图的列列表在定义时确定；给基表 `ADD COLUMN` 后视图不会自动包含新列（需 `CREATE OR REPLACE VIEW`）；
- 视图也可以定义临时版本：`CREATE TEMP VIEW`；
- 视图可以嵌套，但过深会影响可读性与优化（建议 ≤ 3 层）。

6.8 元数据查询 ("鸭子版的 information_schema")

DuckDB 提供一组 `duckdb_*` 的元数据函数，比标准的 `information_schema` 更好用：

```
1  SHOW TABLES;                -- 列出表
2  SHOW ALL TABLES;            -- 含内置表
3  DESCRIBE sales;              -- 列与类型（等价 SHOW
    COLUMNS FROM sales）
4  SHOW CREATE TABLE sales;     -- 完整建表语句
5  SHOW TABLESPACE;            -- 存储信息
6
7  SELECT * FROM duckdb_tables() ORDER BY table_name;
8  SELECT * FROM duckdb_columns() WHERE table_name = 'sales';
9  SELECT * FROM duckdb_views();
10 SELECT * FROM duckdb_schemas();
11 SELECT * FROM duckdb_constraints() LIMIT 20;
12 SELECT * FROM duckdb_indexes();
13 SELECT * FROM duckdb_types();
14 SELECT * FROM duckdb_functions() WHERE function_name LIKE 'list%';
15 SELECT * FROM duckdb_settings() WHERE name = 'threads';
16 SELECT * FROM duckdb_extensions();
17 SELECT * FROM duckdb_databases();
18 SELECT * FROM duckdb_dependencies() LIMIT 20;
19 SELECT * FROM duckdb_temporary_files(); -- 是否有 spill 临时文件
```

6.8.1 实用小查询

```
1  -- 1) 每张表有多大
2  SELECT
3      t.table_name,
4      t.estimated_size,
5      t.column_count,
6      t.index_count
7  FROM duckdb_tables() t
8  ORDER BY t.estimated_size DESC;
9
10 -- 2) 某表所有列及其类型
11 SELECT column_name, data_type, is_nullable
12 FROM duckdb_columns()
13 WHERE table_name = 'sales'
14 ORDER BY column_index;
```

```

15
16 -- 3) 统计某库里有哪些 VARCHAR 列（找字段时可以省事）
17 SELECT table_name, count(*) AS varchar_cols
18 FROM duckdb_columns()
19 WHERE data_type = 'VARCHAR'
20 GROUP BY table_name
21 ORDER BY varchar_cols DESC;

```

6.9 注释 (Comments)

工程化很重要的一环：给表和列写文档。

```

1 COMMENT ON TABLE employees IS '员工主数据，每日凌晨全量刷新';
2 COMMENT ON COLUMN employees.salary IS '税前月薪，单位 CNY';
3
4 -- 查看
5 SELECT table_name, comment FROM duckdb_tables();
6 SELECT column_name, comment FROM duckdb_columns() WHERE
   table_name='employees';

```

6.10 表设计实务：OLAP 与 OLTP 的差别

6.10.1 不要过度规范化

OLTP 数据库推崇第三范式 (3NF)，因为目标是“减少更新异常”。

OLAP (DuckDB 所在的世界) 恰恰相反：**宽表更少 JOIN，扫描更快。**

	OLTP 设计	DuckDB (OLAP) 设计
维度表	独立表，靠外键关联	可以直接冗余进事实表，或保留维表靠 JOIN (DuckDB 的哈希连接很快)
更新	频繁单行	批量覆盖/追加
冗余	尽量避免	适度冗余可换性能
粒度	一行一笔业务	一行一个聚合单元

DuckDB 的哈希连接性能很强，因此**不一定要做完全的事实表冗余**。折中方案：保留星型模型，但把频繁使用、取值稳定的维度（如“区域名称”）内联到事实表里。

6.10.2 一个具体的宽表例子

```

1 CREATE OR REPLACE TABLE mart.order_wide AS
2 SELECT
3     s.order_id,
4     s.order_date,
5     date_trunc('month', s.order_date) AS month,
6     s.region,

```

```

7      s.category,
8      s.status,
9      s.amount,
10     s.quantity,
11     u.user_name,
12     u.plan,
13     u.reg_date,
14     (s.amount / NULLIF(s.quantity, 0))::DECIMAL(10,2) AS unit_price
15 FROM sales s
16 LEFT JOIN users u USING (user_id);
17
18 COMMENT ON TABLE mart.order_wide IS '订单宽表，包含用户维度，按
    order_date 排序存储';

```

宽表的代价是**存储冗余 + 更新一致性**。因为 DuckDB 常常是"批量重建"，这个代价可控。

6.10.3 列的顺序有意义吗？

对行式数据库有意义；对列式数据库**基本没意义**（每列单独存储）。所以你可以任意加列（`ADD COLUMN`）而不必担心性能。

但**行的顺序**（即数据如何写入）非常有意义——它决定了 Zone Map 能否跳过数据。这就是为什么 `ORDER BY` 写入能大幅提速。

实例演练

实例 6-1 · 约束真的会被执行（实测报错原文）

```

1 CREATE OR REPLACE TABLE pk_demo (id INTEGER PRIMARY KEY, v VARCHAR);
2 INSERT INTO pk_demo VALUES (1, 'a');
3 INSERT INTO pk_demo VALUES (1, 'b');  -- 主键冲突

```

```

1 Constraint Error: Duplicate key "id: 1" violates primary key
  constraint.

```

```

1 CREATE OR REPLACE TABLE chk (amount DECIMAL(10,2) CHECK (amount > 0));
2 INSERT INTO chk VALUES (-1);

```

```

1 Constraint Error: CHECK constraint failed on table chk with expression
  CHECK((amount > 0))

```

实例 6-2 · CTAS 不会继承约束（重要）

```

1 CREATE OR REPLACE TABLE src_pk (id INTEGER PRIMARY KEY, v VARCHAR);
2 CREATE OR REPLACE TABLE ctas_copy AS SELECT * FROM src_pk;
3
4 SELECT (SELECT count(*) FROM duckdb_constraints() WHERE table_name =
   'src_pk') AS src_constraints,
5       (SELECT count(*) FROM duckdb_constraints() WHERE table_name =
   'ctas_copy') AS ctas_constraints;

```

src_constraints	ctas_constraints
2	0

想让新表也有主键，必须显式 `CREATE TABLE ... (... PRIMARY KEY ...)` 或事后 `ALTER TABLE ... ADD PRIMARY KEY (...)`。

实例 6-3 · schema 分层的最小骨架

```

1 CREATE SCHEMA IF NOT EXISTS staging;
2
3 CREATE OR REPLACE TABLE staging.orders AS
4 SELECT order_id, region, amount FROM sales LIMIT 100;
5
6 SELECT count(*) AS staging_rows FROM staging.orders; -- 100

```

推荐的四层：raw（视图，直读文件）→ staging（清洗）→ intermediate（中间）→ mart（应用宽表）。

实例 6-4 · 临时表与事务内 DDL

```

1 CREATE TEMP TABLE tmp_x AS SELECT 1 AS a;
2 SELECT count(*) FROM tmp_x; -- 1（连接关闭后自动消失）
3
4 BEGIN;
5 CREATE TABLE will_rollback (x INTEGER);
6 ROLLBACK;
7 SELECT count(*) FROM duckdb_tables() WHERE table_name =
   'will_rollback'; -- 0

```

DDL 能回滚是 DuckDB 的一大优势（MySQL 做不到）。这让“建表 + 灌数据”可以作为一个原子操作。

实例 6-5 · 列的顺序无所谓，行的顺序很关键

```

1 -- 列式存储：加列几乎零成本
2 ALTER TABLE staging.orders ADD COLUMN note VARCHAR;
3
4 -- 但"按常用过滤列排序写入"会让后续查询快得多：
5 COPY (SELECT * FROM sales ORDER BY order_date)
6 TO 'sales_sorted.parquet' (FORMAT PARQUET, COMPRESSION ZSTD);

```

6.11 本章小结

- CTAS 是 DuckDB 里最常用的建表方式；`CREATE OR REPLACE` 让脚本幂等；
 - 约束里真正重要的是 PRIMARY KEY/UNIQUE（配合 upsert）与 CHECK（数据质量）；
 - **不要用传统索引思维理解 DuckDB**：用"排序存储 + 分区 + Zone Map"替代；
 - schema 分层（`raw/staging/mart`）非常适合 DuckDB 的 ETL 场景；
 - 视图不存数据、始终最新；贵的结果是物化成表；
 - `duckdb_tables()` / `duckdb_columns()` 等元数据函数是日常必备。
-

动手练习

1. 建三个 schema：`raw`、`staging`、`mart`，按 6.5.1 的例子搭建一条 pipeline。
2. 给 `sales` 加一个 `unit_price` 列并回填（`amount / quantity`）。
3. 创建视图 `v_region_daily`（区域 × 日期聚合），然后创建 `CREATE OR REPLACE TABLE t_region_daily AS SELECT * FROM v_region_daily` 对比两者对同一查询的表现。
4. 给一张表加 PRIMARY KEY 并测试 `INSERT OR REPLACE` 的效果。
5. 用 `duckdb_columns()` 找出 `sales` 中所有的 VARCHAR 列，思考哪些可以改成 ENUM。

下一章

[第 7 章 查询基础](#) —— 投影、过滤、排序、分页与 NULL 陷阱。

第 7 章 查询基础

本章夯实 SELECT 的地基。即使你已经写过几年 SQL，也建议读一遍第 7.5 节（NULL 陷阱）与 7.6 节（谓词写法与性能）。

7.1 一条 SELECT 的全貌

```
1 SELECT [DISTINCT] select_expr [, ...]           -- 5. 投影
2 FROM table_expr [, ...]                         -- 1. 数据源
3 WHERE condition                                 -- 2. 行过滤
4 GROUP BY group_expr [, ...]                    -- 3. 分组
5 HAVING condition                               -- 4. 组过滤
6 WINDOW window_def [, ...]                      -- 窗口定义
7 QUALIFY condition                             -- 窗口结果过滤
8 ORDER BY order_expr [ASC|DESC] [NULLS ...]     -- 6. 排序
9 LIMIT n [OFFSET m];                            -- 7. 截断
```

逻辑执行顺序（注意与书写顺序不同）正是上面标注的 1→7。理解这一点能解释很多“为什么这里不能用别名”的问题。

7.2 投影：选择你需要的列

7.2.1 基本投影与别名

```
1 SELECT
2     region,
3     amount,
4     amount * 0.9 AS discounted,
5     round(amount, 0)::BIGINT AS amount_int,
6     upper(category) || '-' || status AS tag
7 FROM sales
8 LIMIT 3;
```

7.2.2 去重 DISTINCT

```
1 SELECT DISTINCT region FROM sales;
2 SELECT DISTINCT region, category FROM sales;           -- 组合去重
3 SELECT DISTINCT ON (region) region, order_id, amount
4 FROM sales
5 ORDER BY region, amount DESC;                           -- 每个 region 取
   amount 最大的一行
```

`DISTINCT ON` 是 PostgreSQL 语法，DuckDB 支持。它比窗口函数写 Top-1 更短，但**必须配 ORDER BY**。现代写法更推荐 `QUALIFY row_number() OVER (...) = 1`（第 10 章）。

7.2.3 常量与表达式

```
1 SELECT
2     '常量'                AS c,
3     1 + 2 * 3             AS calc,
4     now()                 AS ts,
5     current_date           AS today,
6     current_setting('threads') AS threads,
7     version()              AS ver,
8     uuid()                 AS random_id
9 LIMIT 1;
```

注意：不带 FROM 的 SELECT 返回一行。这是测试函数的好办法。

7.3 过滤：WHERE 的艺术

7.3.1 比较运算符

```
1 SELECT * FROM sales WHERE amount > 500;
2 SELECT * FROM sales WHERE amount <> 500;      -- 不等于 (!= 也可以)
3 SELECT * FROM sales WHERE amount >= 100 AND amount <= 900;
4 SELECT * FROM sales WHERE amount BETWEEN 100 AND 900;  -- 闭区间
5 SELECT * FROM sales WHERE region IN ('华东', '华南');
6 SELECT * FROM sales WHERE category NOT IN ('图书');
```

7.3.2 字符串匹配

```
1 -- 前缀/包含/后缀
2 WHERE name LIKE 'A%'      -- 以 A 开头
3 WHERE name LIKE '%DB%'    -- 包含 DB
4 WHERE name LIKE 'user_'   -- user + 恰好一个任意字符
5 WHERE name ILIKE '%db%'   -- 不区分大小写 (Postgres 风格, DuckDB 支持)
6
7 -- 正则 (推荐这种方式)
8 WHERE name ~ '^U.*d$'      -- 匹配 POSIX 正则
9 WHERE name !~ 'test'       -- 不匹配
10 WHERE regexp_matches(email, '^[@@]+@.+$') -- 函数形式
11
12 -- 其他
13 WHERE starts_with(name, 'A')
14 WHERE contains(lower(name), 'duck')
15 WHERE ends_with(file_path, '.csv')
```

7.3.3 布尔组合优先级

优先级从高到低: `NOT` > `AND` > `OR`。永远用括号明确意图:

```
1  -- 危险: 会被解析为 (region='华东' AND amount>500) OR status='已完成'
2  WHERE region = '华东' AND amount > 500 OR status = '已完成'
3
4  -- 明确
5  WHERE region = '华东' AND (amount > 500 OR status = '已完成')
```

7.3.4 日期过滤的写法性能差异

```
1  -- ❌ 慢: 对列做了函数运算, 无法下推 (sargable 原则)
2  WHERE strftime(order_date, '%Y-%m') = '2026-03'
3
4  -- ✅ 快: 直接对列做范围比较
5  WHERE order_date >= DATE '2026-03-01' AND order_date < DATE '2026-04-01'
6
7  -- ✅ 也快: date_trunc 后的比较
8  WHERE date_trunc('month', order_date) = DATE '2026-03-01'
```

SARGable 原则 (Search ARGument ABLE): 永远不要让索引/统计信息失效于 "把列包在函数里"。虽然 DuckDB 没有 B-Tree, 但这个原则依然重要——列包在函数里会导致 Zone Map 无法跳过数据。

7.4 排序与分页

7.4.1 ORDER BY

```
1  SELECT * FROM sales ORDER BY amount DESC;
2  SELECT * FROM sales ORDER BY region ASC, amount DESC;  -- 多列, 依次
   优先
3
4  -- NULL 的排序位置
5  SELECT * FROM t ORDER BY x NULLS FIRST;
6  SELECT * FROM t ORDER BY x DESC NULLS LAST;
7
8  -- 按位置
9  SELECT region, amount FROM sales ORDER BY 2 DESC;
10
11 -- 按表达式
12 SELECT region, amount FROM sales ORDER BY abs(amount - 500);
13
14 -- Ducks: 按所有列排序
15 SELECT region, category, sum(amount) gm FROM sales GROUP BY ALL
   ORDER BY ALL DESC;
```

7.4.2 LIMIT / OFFSET 的陷阱

```
1 SELECT * FROM sales ORDER BY amount DESC LIMIT 10;
2 SELECT * FROM sales ORDER BY amount DESC LIMIT 10 OFFSET 20;
```

⚠ **OFFSET 的代价是线性增长的：** `OFFSET 100000` 需要真正跳过 10 万行。深分页请用“游标” (keyset pagination)：

```
1 -- 第一页
2 SELECT * FROM sales ORDER BY (order_id, amount) LIMIT 20;
3 -- 下一页（用上一页最后一行的值继续）
4 SELECT * FROM sales
5 WHERE amount < last_amount OR (amount = last_amount AND order_id
6 > last_id)
7 ORDER BY amount DESC, order_id
8 LIMIT 20;
```

7.4.3 交互式查看的安全习惯

CLI 里有两招防止误刷全表：

```
1 .maxrows 100      -- 限制每次输出行数
2 .limit 100        -- 自动给每条 SELECT 加 LIMIT
```

Python 里可以用 `.limit(20).show()` 或 `.df()` 前先看 `.arrow().num_rows`。

7.5 NULL：最常见的 bug 来源

7.5.1 三值逻辑复习

操作	结果
<code>NULL = 1</code>	NULL
<code>NULL <> 1</code>	NULL
<code>NULL IS NULL</code>	TRUE
<code>NULL IS DISTINCT FROM 1</code>	TRUE (推荐写法)

```
1 -- ❌ 陷阱：NULL 的行不会被返回
2 SELECT count(*) FROM sales WHERE status <> '已完成';
3
4 -- ✅ 正确写法
5 SELECT count(*) FROM sales WHERE status IS DISTINCT FROM '已完成';
6 -- 等价：WHERE status <> '已完成' OR status IS NULL
```

7.5.2 聚合函数与 NULL

聚合	NULL 行为
<code>count(*)</code>	计数所有行 (含全 NULL 行)
<code>count(col)</code>	只数非 NULL
<code>sum/avg/min/max</code>	忽略 NULL
<code>sum(col)</code> 全为 NULL 时	返回 NULL (不是 0!)
<code>string_agg</code>	忽略 NULL

```
1 SELECT count(*) AS rows, count(quantity) AS qty_nonnull
2 FROM sales;
3 -- 两者不同 → 说明 quantity 有 NULL
```

想要 NULL 当 0 处理: `coalesce(sum(x), 0)`。

7.5.3 NULL 与排序/去重/连接

- 排序时 ASC 默认 `NULLS LAST` (DuckDB 的默认可能与某些库不同, 建议显式写);
- `DISTINCT` 认为多个 NULL 是相同的 (只保留一个);
- JOIN 时 `NULL = NULL` 不成立, 所以 NULL 键**不会匹配**。

7.5.4 处理 NULL 的工具

```
1 SELECT
2     coalesce(a, b, c, 0)      AS first_non_null,
3     ifnull(a, 0)              AS a_or_zero,
4     nullif(a, '')             AS empty_to_null,
5     CASE WHEN x IS NULL THEN '缺失' ELSE '有值' END AS flag,
6     x IS NULL                 AS is_missing
7 FROM t;
```

7.6 表达式、运算符与类型转换

7.6.1 常用运算

```
1 SELECT
2     7 / 2          AS int_div,      -- 3 (整数除法)
3     7 / 2.0        AS real_div,    -- 3.5
4     7 % 3          AS modulo,      -- 1
5     2 ^ 10         AS power,       -- 1024 (注意 ^ 是指数, 不是异或)
6     7 // 2         AS floor_div,   -- 3
7     abs(-5)        AS abs,
8     'a' || 'b'     AS concat      -- 'ab'
```

⚠ 在 SQL 里 `/` 对整数做整数除法（和 Python 3 不同）。想要浮点结果，至少要有有一个操作数是浮点：`amount / 3.0`。

7.6.2 条件表达式

```
1  -- CASE WHEN (标准)
2  SELECT
3      CASE
4          WHEN amount >= 900 THEN '高'
5          WHEN amount >= 500 THEN '中'
6          WHEN amount >= 100 THEN '低'
7          ELSE '极低'
8      END AS level
9  FROM sales;
10
11 -- 简单 CASE
12 SELECT CASE status WHEN '已支付' THEN 1 WHEN '已发货' THEN 2 ELSE 3
13      END AS code FROM sales;
14
15 -- IIF (简洁)
16 SELECT iif(amount > 500, 'big', 'small') AS tag FROM sales;
17
18 -- COALESCE / NULLIF (前面讲过)
```

7.6.3 类型转换 safty

```
1  SELECT
2      CAST(amount AS INTEGER)          AS c1,
3      try_cast('abc' AS INTEGER)       AS c2,    -- NULL
4      try_cast('abc' AS INTEGER DEFAULT -1) AS c3, -- -1
5      amount::VARCHAR                  AS c4;
```

7.7 数据抽样与探查（探索型查询）

当你拿到一张陌生表，建议按以下顺序：

```
1  -- 1) 有多少行
2  SELECT count(*) FROM sales;
3
4  -- 2) 表结构
5  DESCRIBE sales;
6
7  -- 3) 看几条
8  SELECT * FROM sales USING SAMPLE 5;
9
10 -- 4) 数值列的分布
11 SELECT
12     min(amount) AS min, quantile_cont(amount, 0.25) AS p25,
13     median(amount) AS p50, quantile_cont(amount, 0.75) AS p75,
14     max(amount) AS max, avg(amount) AS mean, stddev(amount) AS sd
```

```

15 FROM sales;
16
17 -- 5) 类别列的取值分布
18 SELECT region, count(*) AS n, round(count(*) * 100.0 / sum(count(*))
OVER (), 2) AS pct
19 FROM sales GROUP BY region ORDER BY n DESC;
20
21 -- 6) 缺失值统计
22 SELECT
23     count(*) AS total,
24     count(*) - count(quantity) AS miss_quantity,
25     count(*) - count(category) AS miss_category,
26     count(*) - count(amount) AS miss_amount
27 FROM sales;
28
29 -- 7) 一键 describe (Python 里)

```

Python 里的一键profile:

```
1 | duckdb.sql("FROM sales").describe().show()
```

或 SQL:

```

1 | SELECT column_name, data_type, null_count
2 | FROM duckdb_columns() WHERE table_name='sales';
3 | -- 注: DuckDB 的 SUMMARIZE 语句也很方便
4 | SUMMARIZE sales;

```

`SUMMARIZE` 是 DuckDB 提供的便利语句, 输出每列的 min/max/null/approx_unique/avg 等统计。探索数据时第一条就写它。

7.8 子查询入门

7.8.1 FROM 里的子查询 (派生表)

```

1 | SELECT t.region, t.cnt
2 | FROM (SELECT region, count(*) AS cnt FROM sales GROUP BY region) AS t
3 | WHERE t.cnt > 30000
4 | ORDER BY t.cnt DESC;

```

7.8.2 WHERE 里的子查询

```

1  -- IN
2  SELECT * FROM sales
3  WHERE user_id IN (SELECT user_id FROM users WHERE plan = '企业版');
4
5  -- EXISTS (通常更快, 且 NULL 语义更安全)
6  SELECT * FROM sales s
7  WHERE EXISTS (SELECT 1 FROM users u WHERE u.user_id = s.user_id AND
8                u.plan = '企业版');
9
10 -- 相关子查询 (每行执行一次, 注意性能)
11 SELECT s.order_id,
12        (SELECT plan FROM users u WHERE u.user_id = s.user_id) AS
13        plan
14 FROM sales s;

```

性能提示:

- `IN` (子查询) 会被优化器转换成半连接 (semi join), 通常没问题;
- `NOT IN` 遇到子查询含 NULL 会**返回空结果** (经典陷阱)。用 `NOT EXISTS` 更安全;
- 相关子查询在标量子查询形式下 DuckDB 往往能去相关化 (decorrelate), 但不要指望它。

7.9 常用单行函数速查

数值

```

1  round(x, n)  ceil(x)  floor(x)  trunc(x)  abs(x)  sqrt(x)  ln(x)
   log2(x)  exp(x)
2  power(a,b)  mod(a,b)  sign(x)  greatest(a,b,...)  least(a,b,...)
3  random()  setseed(0.42)

```

字符串

```

1  length(s)  upper(s)  lower(s)  trim(s)  ltrim/rtrim  lpad/rpad
   substr(s, from, len)
2  concat(a,b,...)  concat_ws(sep, ...)  string_split(s, sep)
   string_to_array
3  strpos(s, sub)  contains(s, sub)  starts_with  ends_with  replace(s,
   from, to)
4  regexp_extract(s, pat, grp)  regexp_replace  regexp_matches
   regexp_split_to_array
5  md5(s)  sha256(s)  encode(b)  decode(s)  format('{}', x)  printf('%d',
   x)

```

日期

```
1 now() current_date current_timestamp today()
2 date_part(part, ts) date_trunc(part, ts) strptime(s, fmt)
  strftime(ts, fmt)
3 age(a, b) date_diff(part, a, b) epoch(ts) to_timestamp(x)
4 greatest/least (也可用于时间)
```

条件/空值

```
1 coalesce() ifnull() nullif() iif() CASE WHEN isnull(x)
  try_cast()
```

实例演练

实例 7-1 · <> 会悄悄吞掉 NULL (实测)

先造一批 NULL:

```
1 CREATE OR REPLACE TABLE t_null AS SELECT * FROM sales;
2 UPDATE t_null SET status = NULL WHERE order_id % 10 = 0; -- 2 万行置
  为 NULL
```

对比两种写法:

```
1 SELECT count(*) AS total,
2        count(*) FILTER (WHERE status <> '已完成') AS
ne_rows,
3        count(*) FILTER (WHERE status IS DISTINCT FROM '已完成') AS
distinct_rows,
4        count(*) - count(status) AS
null_rows
5 FROM t_null;
```

total	ne_rows	distinct_rows	null_rows
200000	119834	139834	20000

整整 2 万行 (正好等于 NULL 行数) 被 <> 漏掉了。用 IS DISTINCT FROM 才是正确写法。

实例 7-2 · 深分页: OFFSET vs keyset

```

1  -- OFFSET (要真的跳过 10 万行)
2  SELECT order_id, amount FROM t_null ORDER BY amount DESC LIMIT 5
   OFFSET 100000;
3
4  -- keyset (用上一页最后一行的值继续, 代价恒定)
5  SELECT order_id, amount FROM t_null
6  WHERE (amount, order_id) < (上一页最后一行的 amount, order_id)
7  ORDER BY amount DESC, order_id LIMIT 5;

```

在只有 5 行的分页上两者结果一致：

order_id	amount
6	149.51
7	936.82
8	211.29
9	454.61
10	207.65

差距要翻到“第 1 万页”才明显：OFFSET 的代价随页码**线性增长**，keyset 恒定。

实例 7-3 · SARGable：不要把列包在函数里

```

1  -- ❌ 慢：对列做了运算，Zone Map 失效
2  SELECT count(*) FROM sales WHERE strftime(order_date, '%Y-%m') =
   '2026-03';
3
4  -- ✅ 快：直接对列做范围比较
5  SELECT count(*) FROM sales
6  WHERE order_date >= DATE '2026-03-01' AND order_date < DATE '2026-04-
   01';

```

(两条都返回同样结果，但后者能跳过整个 row group。)

实例 7-4 · 探索陌生表的四步

```

1  DESCRIBE sales;           -- 1) 结构
2  SELECT * FROM sales USING SAMPLE 5;  -- 2) 看几条
3  SUMMARIZE sales;         -- 3) 全列统计
4  SELECT count(*) FROM sales;  -- 4) 规模

```

7.10 本章小结

- SELECT 的**书写顺序** ≠ **执行顺序**，逻辑顺序是 FROM→WHERE→GROUP BY→HAVING→SELECT→ORDER BY→LIMIT；

- 让列保持"SARGable": 不要把列包在函数里去做比较;
 - **OFFSET 深分页很贵**, 用 keyset 分页;
 - NULL 不是 0 也不是空字符串: `<>` 会漏掉 NULL, 请用 `IS DISTINCT FROM`;
 - `count(*)` 与 `count(col)` 不同; `sum()` 全 NULL 返回 NULL;
 - 探索陌生数据时先用 `SUMMARIZE`、`DESCRIBE`、`USING SAMPLE`;
 - 小心 SQL 的整数除法: `/` 对整数会取整, 写成 `x / 1.0` 才能得到小数结果。
-

动手练习

1. 用 `SUMMARIZE sales` 看统计摘要, 找出 amount 的中位数与最大值。
2. 统计每天订单数与 GMV, 用半开区间日期范围过滤 2026 年 3 月。
3. 找出 `quantity` 为 NULL 的行数 (如果都是非 NULL, 先用 UPDATE 造几个 NULL 练习)。
4. 对比: `WHERE amount <> 500` 与 `WHERE amount IS DISTINCT FROM 500` 在有 NULL 时的行数差异。
5. 用 keyset 分页写一个"取第 5 页 (每页 20 行, 按 amount desc) "的查询。

下一章

[第 8 章 聚合与分组的艺术](#) —— OLAP 的核心。

第 8 章 聚合与分组的艺术

OLAP 的本质就是“把几亿行压成几百行”。本章把这件事讲透。

8.1 聚合的本质

```
1 | SELECT count(*) FROM sales;      -- 20 万行 → 1 行
```

聚合有两个隐藏的关键属性：

1. **内存占用**：大部分聚合需要把中间状态放内存（哈希表）；高基数的 `GROUP BY`（比如按 `user_id` 分组）会很吃内存。**DuckDB 会自动落盘**（spill），第 17 章会讲怎么判断与调优。
2. **并行模型**：DuckDB 会拆分数据到多个线程各自做局部聚合，最后合并（第 16 章讲原理）。

8.2 聚合函数全景

8.2.1 基础统计

```
1 | SELECT
2 |     count(*)                AS rows,
3 |     count(amount)           AS non_nulls,
4 |     count(DISTINCT user_id) AS users,
5 |     sum(amount)              AS total,
6 |     avg(amount)              AS mean,
7 |     min(amount), max(amount),
8 |     median(amount)           AS p50,
9 |     quantile_cont(amount, 0.95) AS p95,
10 |    quantile_disc(amount, 0.95) AS p95_exact_value,
11 |    stddev(amount), variance(amount),
12 |    skewness(amount), kurtosis(amount)
13 | FROM sales;
```

函数	说明
<code>count(x)</code>	非 NULL 行数
<code>count(*)</code>	所有行数
<code>sum / avg / min / max</code>	标准聚合
<code>median(x)</code>	中位数（ <code>quantile_cont(x, 0.5)</code> 别名）
<code>quantile_cont(x, p)</code>	连续分位数（插值）
<code>quantile_disc(x, p)</code>	离散分位数（返回一个实际存在的值）

函数	说明
<code>stddev</code> / <code>variance</code>	样本标准差/方差
<code>stddev_pop</code> / <code>var_pop</code>	总体标准差/方差
<code>skewness</code> / <code>kurtosis</code>	偏度/峰度

`quantile_cont` vs `quantile_disc`: 前者在相邻值之间插值 (比如 [1,2] 的中位数是 1.5), 后者返回实际存在的某个值 ([1,2] 的中位数是 1 或 2)。做"P95 响应时间"分析通常用 `cont`; 做"某个订单金额"则用 `disc`。

8.2.2 近似 (Approximate) 聚合 —— 超大数据集的利器

当你有 10 亿行、只需要"大概多少个不同的用户", 精确去重会很慢。DuckDB 提供 HyperLogLog 系列:

```
1 SELECT
2     approx_count_distinct(user_id) AS approx_users,
3     count(DISTINCT user_id)       AS exact_users,
4     approx_quantile(amount, 0.95) AS approx_p95
5 FROM sales;
```

函数	用途	误差
<code>approx_count_distinct(x)</code>	近似去重计数 (HLL)	~2% (可调)
<code>approx_quantile(x, p)</code>	近似分位数 (t-digest/GK 类结构)	小
<code>approx_top_k(x, k)</code>	近似 Top-K 频次	有概率遗漏

近似聚合的价值: 内存占用从 $O(\text{去重基数})$ 降到 $O(\log)$, 可以处理天文数字级别的基数。

典型场景: UV 统计、日志里的独立 IP 数。

```
1 -- 自定义精度 (部分版本支持第三个参数)
2 SELECT approx_count_distinct(user_id, 0.01) AS uv FROM sales;
```

8.2.3 去重计数的三种写法对比

```
1 -- (A) 精确但昂贵
2 SELECT count(DISTINCT user_id) FROM sales;
3
4 -- (B) 先 GROUP BY 再 count (有时更快, 因为可以并行 + spill)
5 SELECT count(*) FROM (SELECT user_id FROM sales GROUP BY user_id);
6
7 -- (C) 近似
8 SELECT approx_count_distinct(user_id) FROM sales;
```

经验：(A) 在小/中基数时最优；(B) 在超高基数且内存不足时更稳；(C) 在可以接受误差时最快、最省内存。

8.2.4 字符串/数组聚合

```
1 SELECT
2     string_agg(category, ',')                AS all_cats,
3     string_agg(DISTINCT category, '|')        AS uniq_cats,
4     string_agg(category, ', ' ORDER BY amount DESC) AS sorted_cats,
5     list(user_id)                            AS user_list,
6     list(DISTINCT user_id)                   AS uniq_list,
7     array_agg(amount)                        AS amounts,
8     histogram(region)                       AS region_hist,
9     -- MAP<值, 计数>
10    sum(amount) FILTER (WHERE region='华东') AS east_gmv
FROM sales;
```

FILTER 子句是 SQL 标准的一大福利，让**条件聚合**异常清晰：

```
1 SELECT
2     count(*)                AS total,
3     count(*) FILTER (WHERE status = '已支付') AS paid,
4     count(*) FILTER (WHERE status = '已完成') AS done,
5     sum(amount) FILTER (WHERE category='图书') AS book_gmv,
6     round(100.0 * count(*) FILTER (WHERE status='已完成') / count(*),
7     2) AS done_pct
FROM sales;
```

FILTER 比 `sum(CASE WHEN ... THEN 1 ELSE 0 END)` 更清晰，且优化器能更好地处理。写条件聚合优先用 **FILTER**。

8.2.5 布尔聚合

```
1 SELECT
2     bool_and(x)  -- 全部为 true 才 true
3     bool_or(x)   -- 任一为 true 即 true
4     every(x)     -- 同 bool_and
5 FROM t;
```

8.2.6 统计 Titans: 相关性与协方差

```
1 SELECT
2     corr(amount, quantity) AS pearson_r,
3     covar_pop(amount, quantity) AS cov_pop,
4     covar_samp(amount, quantity) AS cov_samp,
5     regr_slope(y, x) AS slope,      -- 线性回归斜率
6     regr_intercept(y, x) AS intercept,
7     regr_r2(y, x) AS r2
8 FROM sales;
```

这些函数让 DuckDB 能做"数据库内的机器学习特征分析"。

8.3 GROUP BY 深入

8.3.1 分组键的几种写法

```
1  -- 列名
2  GROUP BY region
3  -- 表达式
4  GROUP BY date_trunc('month', order_date)
5  -- 位置
6  GROUP BY 1, 2
7  -- ALL (自动展开非聚合列)
8  GROUP BY ALL
9  -- 组合/空集
10 GROUP BY GROUPING SETS ((region), (category), ()) -- 见下
```

8.3.2 HAVING: 过滤分组结果

```
1  SELECT region, sum(amount) AS gmv
2  FROM sales
3  GROUP BY region
4  HAVING sum(amount) > 4_000_000
5  ORDER BY gmv DESC;
6
7  -- DuckDB 允许在 HAVING 里使用别名
8  SELECT region, sum(amount) AS gmv
9  FROM sales
10 GROUP BY region
11 HAVING gmv > 4_000_000;
```

WHERE vs HAVING: WHERE 在聚合前过滤行（减少工作量，优先用）；HAVING 在聚合后过滤组。

```
1  -- ✅ 好: 先过滤再聚合（大多数时候）
2  SELECT region, sum(amount) FROM sales WHERE order_date >= DATE '2026-03-01' GROUP BY region;
3
4  -- ❌ 差: 先聚合所有再过滤（除非你本来就要全量结果）
5  SELECT region, sum(amount) FROM sales GROUP BY region HAVING region IN ('华东', '华南');
```

8.3.3 多级分组: ROLLUP / CUBE / GROUPING SETS

这是做报表的神器。一次性产生多个粒度的小计与合计。

```
1  -- ROLLUP: 层级小计 (region → region, category → 总计)
2  SELECT region, category, sum(amount) AS gmv
3  FROM sales
```

```

4  GROUP BY ROLLUP (region, category)
5  ORDER BY region NULLS LAST, category NULLS LAST;
6
7  -- CUBE: 所有组合的小计
8  SELECT region, category, sum(amount) AS gmv
9  FROM sales
10 GROUP BY CUBE (region, category);
11
12 -- GROUPING SETS: 精确指定要哪几种粒度
13 SELECT region, category, sum(amount) AS gmv
14 FROM sales
15 GROUP BY GROUPING SETS ((region), (category), (region, category),
    ());

```

使用 `GROUPING()` 区分"真实 NULL"与"小计产生的 NULL":

```

1  SELECT
2      CASE WHEN GROUPING(region) = 1 THEN '全部区域' ELSE region
3      END AS region_label,
4      CASE WHEN GROUPING(category) = 1 THEN '全部品类' ELSE category
5      END AS cat_label,
6      sum(amount) AS gmv
7  FROM sales
8  GROUP BY ROLLUP (region, category)
9  ORDER BY GROUPING(region), region, GROUPING(category), category;

```

8.3.4 GROUP BY ALL 的边界

```

1  SELECT region, count(*) FROM sales GROUP BY ALL;           -- 
2  SELECT * FROM sales GROUP BY ALL;                           --  通配
   符不会展开
3  SELECT *, count(*) FROM sales GROUP BY ALL;                --  同上

```

混沌写法: `SELECT *, row_number() OVER () FROM sales GROUP BY ALL` 也不行。需要显式写列。

8.4 聚合的经典陷阱

8.4.1 陷阱一: sum 溢出

```

1  SELECT sum(id) FROM huge_table;  -- 若 id 是 INTEGER, sum 会溢出
2  -- 解决: 显式提升
3  SELECT sum(id::BIGINT) FROM huge_table;

```

DuckDB 对 `sum(INTEGER)` 内部通常用更大累加器 (或部分版本自动提升到 `HUGEINT`) , 但显式 `CAST` 是最保险的做法。

8.4.2 陷阱二：avg 的整数除法

```
1 SELECT avg(quantity) FROM sales;    -- 返回 DOUBLE, OK
2 SELECT sum(qty)/sum(n) FROM t;      -- 若都是整数会得到整数结果!
3 SELECT sum(qty)::DOUBLE / sum(n)   -- ✓
```

8.4.3 陷阱三：多次 COUNT(DISTINCT) 在一个查询里

```
1 SELECT count(DISTINCT user_id), count(DISTINCT order_id) FROM sales;
```

这会产生两个独立的去重哈希表，内存翻倍。超大数据集上考虑近似版本或拆成两个查询。

8.4.4 陷阱四：聚合后行数你没有预期

```
1 SELECT count(*) FROM sales;          -- 200000
2 SELECT count(*) FROM sales JOIN users USING (user_id);
3 -- 如果 users.user_id 不唯一，结果会变多（多对多膨胀）
```

JOIN 后计数前，请先确认维表的连接键是唯一的：

```
1 SELECT user_id, count(*) FROM users GROUP BY user_id HAVING
   count(*) > 1;
```

8.4.5 陷阱五：NULL 组

```
1 SELECT coalesce(region, '未知'), sum(amount) FROM sales GROUP BY 1;
```

GROUP BY 会把所有 NULL 归为一组。若需要区分来源，用 coalesce 显式标注。

8.5 直方图与分桶

8.5.1 用 CASE WHEN 手工分桶

```
1 SELECT
2     CASE
3         WHEN amount < 100 THEN '0-100'
4         WHEN amount < 300 THEN '100-300'
5         WHEN amount < 600 THEN '300-600'
6         WHEN amount < 900 THEN '600-900'
7         ELSE '900+'
8     END AS bucket,
9     count(*) AS cnt,
10    round(avg(amount),2) AS avg_amount
11 FROM sales
12 GROUP BY bucket
13 ORDER BY min(amount);
```

8.5.2 等宽分桶（自己算）

⚠ 已实测 (DuckDB 1.5.5) : DuckDB 没有 `width_bucket` 函数 (部分其他数据库有)。用表达式自己算最清楚:

```
1  -- 0~1000 元按 100 元一档分成 10 档，超出范围的归到第 9 档
2  SELECT
3      least(floor(amount / 100)::INTEGER, 9) AS b,
4      count(*)                               AS cnt,
5      min(amount)                             AS lo,
6      max(amount)                             AS hi
7  FROM sales
8  GROUP BY b
9  ORDER BY b;
```

8.5.3 等频分桶（用 ntile）

如果你希望每桶包含同样多的行数（而不是同样宽的值域），用窗口函数 `ntile`：

```
1  SELECT ntile(10) OVER (ORDER BY amount) AS bucket, amount
2  FROM sales
3  LIMIT 5;
4
5  -- 每个十分位的边界值
6  SELECT bucket, min(amount) AS lo, max(amount) AS hi
7  FROM (SELECT ntile(10) OVER (ORDER BY amount) AS bucket, amount FROM
8  sales)
9  GROUP BY bucket
10 ORDER BY bucket;
```

等宽适合做直方图；等频适合做“把用户按消费额平均分成 10 组”。

8.5.3 用 histogram 得到 MAP

```
1  SELECT histogram(region) FROM sales;
2  -- {'华东': 33334, '华北': 33333, ...}
```

结合 `unnest` 展开：

```
1  SELECT unnest(histogram(region)) AS kv FROM sales;
```

8.6 Top-N 与排行

8.6.1 Top-N by 聚合值

```
1 SELECT region, sum(amount) AS gmv
2 FROM sales
3 GROUP BY region
4 ORDER BY gmv DESC
5 LIMIT 3;
```

8.6.2 组内 Top-N (用窗口函数, 第 10 章详述)

```
1 SELECT region, category, gmv
2 FROM (
3     SELECT region, category, sum(amount) AS gmv,
4         row_number() OVER (PARTITION BY region ORDER BY sum(amount)
5             DESC) AS rn
6     FROM sales
7     GROUP BY region, category
8 )
9 WHERE rn <= 2
10 ORDER BY region, gmv DESC;
```

或用 DuckDB 的 `QUALIFY` :

```
1 SELECT region, category, sum(amount) AS gmv
2 FROM sales
3 GROUP BY region, category
4 QUALIFY row_number() OVER (PARTITION BY region ORDER BY sum(amount)
5     DESC) <= 2
6 ORDER BY region, gmv DESC;
```

注意: `QUALIFY` 里可以直接引用聚合结果 (`sum(amount)`), 无需别名, 这比 PostgreSQL 舒服。

8.6.3 百分比/累计占比

```
1 WITH agg AS (
2     SELECT category, sum(amount) AS gmv FROM sales GROUP BY category
3 )
4 SELECT
5     category, gmv,
6     round(100.0 * gmv / sum(gmv) OVER (), 2)
7 AS pct,
8     round(100.0 * sum(gmv) OVER (ORDER BY gmv DESC
9         ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
10     / sum(gmv) OVER (), 2)
11 AS cum_pct
12 FROM agg
13 ORDER BY gmv DESC;
```

这是典型的 **帕累托分析 (80/20)** 写法。

8.7 透视报表：PIVOT / UNPIVOT

有些查询本质就是"把行变成列"：

```
1 SELECT
2     region,
3     sum(amount) FILTER (WHERE category='电子产品') AS 电子产品,
4     sum(amount) FILTER (WHERE category='服饰')      AS 服饰,
5     sum(amount) FILTER (WHERE category='食品')      AS 食品,
6     sum(amount) FILTER (WHERE category='图书')      AS 图书,
7     sum(amount)                                     AS 合计
8 FROM sales
9 GROUP BY region
10 ORDER BY 合计 DESC;
```

这其实是手工 PIVOT。推荐优先掌握这种写法，因为：

1. 它不依赖特定版本是否包含 PIVOT 扩展；
2. 它显式可控（列名、缺省值、多个聚合都清楚）；
3. 优化器处理 CASE WHEN 已经很成熟。

如果想用简写：（部分版本提供 PIVOT 语句/扩展）

```
1 PIVOT sales ON category USING sum(amount) GROUP BY region;
```

反向（宽转长）：

```
1 UNPIVOT region_gmv
2 ON 电子产品, 服饰, 食品, 图书
3 INTO NAME category VALUE gmv;
```

等价的标准写法：

```
1 SELECT region, '电子产品' AS category, 电子产品 AS gmv FROM region_gmv
2 UNION ALL SELECT region, '服饰', 服饰 FROM region_gmv
3 UNION ALL SELECT region, '食品', 食品 FROM region_gmv
4 UNION ALL SELECT region, '图书', 图书 FROM region_gmv;
```

8.8 实战：一个完整的分析脚本

任务：生成一份"区域月度经营报表"。

```
1 CREATE OR REPLACE TABLE mart.monthly_report AS
2 WITH base AS (
3     SELECT
4         s.order_date,
5         date_trunc('month', s.order_date) AS month,
6         s.region,
```

```

7      s.category,
8      s.status,
9      s.amount,
10     s.quantity,
11     u.plan
12 FROM sales s
13 LEFT JOIN users u USING (user_id)
14 WHERE s.order_date >= DATE '2026-01-01'
15 )
16 SELECT
17     month,
18     region,
19     count(*) AS orders,
20     count(DISTINCT quantity) AS
distinct_qty,
21     sum(amount) AS gmv,
22     round(avg(amount), 2) AS aov,
23     sum(amount) FILTER (WHERE status = '已完成') AS
done_gmv,
24     round(100.0 * sum(amount) FILTER (WHERE status='已完成') /
sum(amount), 2) AS done_pct,
25     approx_count_distinct(plan) AS
approx_plans,
26     string_agg(DISTINCT plan, ',') AS
plan_list
27 FROM base
28 GROUP BY month, region
29 HAVING sum(amount) > 100000
30 ORDER BY month, region;

```

查看结果:

```

1 | SELECT * FROM mart.monthly_report LIMIT 10;

```

加上 ROLLUP 小计:

```

1 | SELECT
2 |     month,
3 |     coalesce(region, '合计') AS region,
4 |     sum(gmv) AS gmv
5 | FROM mart.monthly_report
6 | GROUP BY ROLLUP (month, region)
7 | ORDER BY month, GROUPING(region);

```

实例演练

实例 8-1 · FILTER：一行算出多口径

```
1 SELECT count(*) AS total,
2        count(*) FILTER (WHERE status = '已支付') AS paid,
3        count(*) FILTER (WHERE status = '已完成') AS done,
4        round(100.0 * count(*) FILTER (WHERE status = '已完成')
5              / count(*), 2) AS done_pct
6 FROM sales;
```

total	paid	done	done_pct
200000	66493	66813	33.41

比 `sum(CASE WHEN ... THEN 1 ELSE 0 END)` 清晰得多，优化器也更喜欢。

实例 8-2 · ROLLUP 一行给出合计

```
1 SELECT CASE WHEN GROUPING(region) = 1 THEN '合计' ELSE region END AS
   region_label,
2        sum(amount) AS gmv
3 FROM sales
4 GROUP BY ROLLUP (region)
5 ORDER BY GROUPING(region), gmv DESC;
```

region_label	gmv
华南	17176471.53
华北	17172024.94
华中	17109098.41
西南	16992234.30
东北	16880143.73
华东	16847717.73
合计	102177690.64

`GROUPING(region) = 1` 区分“真实的小计行”与“原始数据里的 NULL”。

实例 8-3 · 自己做直方图（没有 width_bucket）

```
1 SELECT least(floor(amount / 100)::INTEGER, 9) AS bucket,
2        count(*) AS cnt
3 FROM sales
4 GROUP BY bucket
5 ORDER BY bucket
6 LIMIT 5;
```

bucket	cnt
0	16361
1	20347
2	20069
3	20438
4	20382

(金额 20~1000 元均匀分布，所以每档约 2 万行；第 0 档只覆盖 0~100 元，行数偏少符合预期。)

实例 8-4 · GROUP BY ALL 的边界

```

1 SELECT region, count(*) FROM sales GROUP BY ALL;  --  region 作为分
   组键
2 SELECT * FROM sales GROUP BY ALL;                --  通配符不会被展
   开

```

需要"全部列 + 聚合"时，请显式列出列或改用窗口函数。

8.9 本章小结

- 聚合函数分四类：基础统计、近似（HLL）、字符串/列表、相关性统计；
- FILTER 子句是写条件聚合的最佳方式；**
- 过滤优先放 **WHERE**（聚合前），少放 **HAVING**（聚合后）；
- ROLLUP/CUBE/GROUPING SETS** 一次生成多粒度小计，配合 **GROUPING()** 区分小计行；
- GROUP BY ALL** 不支持 **SELECT ***；
- 小心：整数 sum 溢出、整数除法、JOIN 膨胀、NULL 分组；
- PIVOT 优先用 **CASE WHEN** / **FILTER** 手工实现，可控且兼容性好。

动手练习

- 用 **SUMMARIZE** 观察 **sales** 的数值列分布，再用手工 **CASE WHEN** 分桶。
- 用 **FILTER** 写出每个 region 的各 status 订单占比（PIVOT 风格）。
- 比较 **count(DISTINCT user_id)** 与 **approx_count_distinct(user_id)** 的差值与耗时。
- 用 **ROLLUP (region, category)** 生成包含区域小计与总计的报表，并用 **GROUPING()** 打标签。
- 做一个 ABC 分析：按 category 的 GMV 累计占比，标记 A（前 80%）、B（80~95%）、C（其他）。

下一章

[第9章 连接与集合运算](#) —— 从 INNER 到 ASOF, 一个都不能落下。

第 9 章 连接与集合运算

连接 (JOIN) 是让 DuckDB 有别于 pandas 的核心能力之一：优化器会自动选择连接算法、连接顺序并动态调整内存使用。本章把所有的 JOIN 讲透，包括很多人没用过的 SEMI/ANTI/LATERAL/ASOF。

9.1 连接的心智模型

```
1 SELECT ...
2 FROM 左表
3 [INNER|LEFT|RIGHT|FULL|CROSS|SEMI|ANTI] JOIN 右表
4 ON 条件
```

本质：按某种规则把两张表的行匹配起来，形成一张更宽的结果。

DuckDB 支持的连接方式一览：

类型	保留左表全部？	保留右表全部？	输出行数	典型用途
INNER (默认)	否 (只留匹配)	否	匹配数	关联维表、过滤
LEFT OUTER	是	否	$\geq$ 左表行数	保留主表全部 + 补维度
RIGHT OUTER	否	是	$\geq$ 右表行数	少用 (可改写成 LEFT)
FULL OUTER	是	是	并集	数据对账、差异比对
CROSS	笛卡尔积	笛卡尔积	行数相乘	生成组合、交叉报表
SEMI	只留匹配的行，不添加列	—	$\leq$ 左表行数	存在性过滤
ANTI	只留不匹配的行	—	$\leq$ 左表行数	找孤儿/缺失数据
LATERAL	逐行调用子查询	—	视子查询	Top-N per group、行式函数
ASOF	时间最近匹配	—	$\leq$ 左表行数	时序对齐、事件归因
POSITIONAL	按位置配对	—	max(两边)	已排好序数据的快速拼接

9.2 基本连接

9.2.1 INNER JOIN

```
1 SELECT s.order_id, u.user_name, u.plan, s.amount
2 FROM sales s
3 JOIN users u ON s.user_id = u.user_id
4 LIMIT 5;
```

等价写法:

```
1 FROM sales s, users u WHERE s.user_id = u.user_id      -- 古老的隐式连接（不推荐）
2 FROM sales s JOIN users u USING (user_id)              -- 同名列，推荐
3 FROM sales s NATURAL JOIN users u                      -- 自动匹配所有同名列（危险）
```

不要用 NATURAL JOIN：它会自动匹配所有同名列。一旦表结构变化（例如两边都新增了 `created_at`），结果会悄悄变错。

9.2.2 JOIN USING 的好处

```
1 SELECT order_id, user_name, amount
2 FROM sales JOIN users USING (user_id);
```

好处：`user_id` 只出现一列，不会有两份同名列导致 `* SELECT` 混乱。

9.2.3 LEFT JOIN

```
1 SELECT count(*) AS total,
2         count(u.user_id) AS matched,
3         count(*) - count(u.user_id) AS unmatched
4 FROM sales s
5 LEFT JOIN users u USING (user_id);
```

重要：在 LEFT JOIN 里判断“是否匹配到”，用右表的连接键是否为 NULL (`u.user_id IS NULL`)，因为右表未匹配时所有列都填 NULL。

9.2.4 RIGHT / FULL OUTER

```
1 -- 两边有哪些 user 没有订单 & 哪些订单没有 user
2 SELECT user_id, s.order_id, u.user_name
3 FROM sales s FULL OUTER JOIN users u USING (user_id)
4 WHERE s.order_id IS NULL OR u.user_id IS NULL;
```

这是数据对账 (reconciliation) 的标准写法。

9.2.5 CROSS JOIN

```
1  -- 所有 region × category 组合
2  SELECT DISTINCT r.region, c.category
3  FROM (SELECT DISTINCT region FROM sales) r
4  CROSS JOIN (SELECT DISTINCT category FROM sales) c;
```

用途：生成"完整笛卡尔骨架"，再 LEFT JOIN 实际数据，从而补出 0 值的行（报表常见需求）。

```
1  WITH skeleton AS (
2      SELECT d.day, r.region
3      FROM (SELECT unnest(generate_series(DATE '2026-01-01', DATE
4          '2026-01-31', INTERVAL 1 DAY)) AS day) d
5      CROSS JOIN (SELECT DISTINCT region FROM sales) r
6  ),
7  actual AS (
8      SELECT order_date, region, sum(amount) AS gmv FROM sales GROUP
9      BY 1, 2
10 )
11 SELECT s.day, s.region, coalesce(a.gmv, 0) AS gmv
12 FROM skeleton s
13 LEFT JOIN actual a ON s.day = a.order_date AND s.region = a.region
14 ORDER BY s.day, s.region;
```

9.3 SEMI / ANTI：只做过滤，不做拼接

9.3.1 SEMI JOIN

"找出在另一张表中存在匹配的左表行"（不添加任何右表列）。

```
1  -- 找出至少有一个企业版用户的订单
2  SELECT *
3  FROM sales s
4  SEMI JOIN users u ON s.user_id = u.user_id AND u.plan = '企业版';
5
6  -- 等价写法（优化器通常会生成同样的计划）
7  SELECT * FROM sales s
8  WHERE EXISTS (SELECT 1 FROM users u WHERE u.user_id = s.user_id AND
9      u.plan='企业版');
```

相比 INNER JOIN 的优势：**不会行数膨胀**。即使右表的 user_id 有重复，左表行数也不会变多。

9.3.2 ANTI JOIN

"找出**不在**另一张表中存在的左表行"

```
1  -- 找出没有匹配用户记录的孤儿订单
2  SELECT * FROM sales s
3  ANTI JOIN users u ON s.user_id = u.user_id;
4
5  -- 等价
6  SELECT * FROM sales s
7  WHERE NOT EXISTS (SELECT 1 FROM users u WHERE u.user_id = s.user_id);
```

⚠ 注意 NULL 语义: `NOT IN` 在子查询含 NULL 时**返回空结果**。所以"找缺失数据"请一律用 ANTI JOIN 或 NOT EXISTS。

```
1  -- ❌ 危险
2  SELECT * FROM sales WHERE user_id NOT IN (SELECT user_id FROM users);
3
4  -- ✅ 安全
5  SELECT * FROM sales s ANTI JOIN users u ON s.user_id = u.user_id;
```

9.4 自连接

```
1  -- 找出同区域内，订单金额比自己高的其他订单的数量
2  SELECT a.order_id, count(b.order_id) AS higher_cnt
3  FROM sales a
4  LEFT JOIN sales b
5      ON a.region = b.region AND b.amount > a.amount
6  GROUP BY a.order_id
7  ORDER BY higher_cnt DESC
8  LIMIT 5;
```

自连接常用于：层级结构（员工-经理）、序列比较（前后行对比 → 但用窗口函数更好，见第 10 章）。

9.5 不等值连接与范围连接

```

1  -- 把每个订单映射到"价格区间"维表
2  WITH ranges AS (
3      SELECT * FROM (VALUES
4          (0, 200, '低'),
5          (200, 600, '中'),
6          (600, 1000, '高')
7      ) AS t(lo, hi, label)
8  )
9  SELECT s.order_id, s.amount, r.label
10 FROM sales s
11 JOIN ranges r ON s.amount >= r.lo AND s.amount < r.hi
12 LIMIT 5;

```

不等值连接 (non-equi join) 无法走哈希连接, DuckDB 会用 nested loop / IEJoin 等算法, 代价随行数增长较快。大表上要谨慎, 第 17 章会讲优化手段。

9.6 LATERAL JOIN: 每行执行一次子查询

LATERAL 让右侧子查询引用左侧的列, 实现"对每一行做一次计算"。

9.6.1 经典场景: 每个分组取 Top-N

```

1  SELECT r.region, top.category, top.gmv
2  FROM (SELECT DISTINCT region FROM sales) AS r,
3  LATERAL (
4      SELECT category, sum(amount) AS gmv
5      FROM sales s
6      WHERE s.region = r.region
7      GROUP BY category
8      ORDER BY gmv DESC
9      LIMIT 2
10 ) AS top
11 ORDER BY r.region, top.gmv DESC;

```

等价 (通常更推荐) 写法:

```

1  SELECT region, category, gmv
2  FROM (
3      SELECT region, category, sum(amount) AS gmv,
4          row_number() OVER (PARTITION BY region ORDER BY sum(amount)
5              DESC) rn
6      FROM sales GROUP BY region, category
7  ) QUALIFY rn <= 2;

```

LATERAL vs 窗口函数: 窗口函数通常更快 (一次扫描 + 分区排序); LATERAL 更灵活 (可以做 LIMIT N、调用表函数等)。

9.6.2 用 LATERAL 调用函数/表函数

```
1  -- 把每个用户的标签字符串拆开
2  SELECT u.user_id, t.tag
3  FROM users u,
4  LATERAL (SELECT unnest(string_split(u.plan, '_')) AS tag) t;
```

9.6.3 LATERAL 与 LEFT

```
1  -- 即使某区域没有数据也保留该行
2  FROM dim d
3  LEFT JOIN LATERAL (SELECT ... LIMIT 3) x ON true
```

9.7 ASOF JOIN：时间序列的"就近连接"

这是 DuckDB 的强大功能之一。它解决"事件对齐"问题。

问题：有一张"价格快照表"（价格变动时才记录），另一张"成交表"。想知道**每笔成交发生时的最新价格**。

```
1  CREATE TABLE price_snapshots (sym VARCHAR, ts TIMESTAMP, price
   DECIMAL(10,2));
2  CREATE TABLE trades (trade_id BIGINT, sym VARCHAR, ts TIMESTAMP, qty
   INTEGER);
3
4  INSERT INTO price_snapshots VALUES
5      ('AAPL', TIMESTAMP '2026-01-01 09:30:00', 100.00),
6      ('AAPL', TIMESTAMP '2026-01-01 10:00:00', 101.50),
7      ('AAPL', TIMESTAMP '2026-01-01 11:30:00', 99.80);
8
9  INSERT INTO trades VALUES
10     (1, 'AAPL', TIMESTAMP '2026-01-01 09:45:00', 10),
11     (2, 'AAPL', TIMESTAMP '2026-01-01 10:30:00', 5),
12     (3, 'AAPL', TIMESTAMP '2026-01-01 12:00:00', 7);
```

```
1  SELECT t.trade_id, t.ts, p.price AS price_at_trade
2  FROM trades t
3  ASOF JOIN price_snapshots p
4      ON t.sym = p.sym AND t.ts >= p.ts
5  ORDER BY t.trade_id;
```

结果每笔成交匹配"不晚于成交时间的最近一次价格"：

trade_id	ts	price_at_trade
1	09:45	100.00
2	10:30	101.50

trade_id	ts	price_at_trade
3	12:00	99.80

9.7.1 其他变体

```

1  -- LEFT ASOF: 找不到也不丢行
2  SELECT * FROM trades t ASOF LEFT JOIN prices p ON t.sym = p.sym AND
   t.ts >= p.ts;
3
4  -- 用 < 取"之后的第一个"
5  SELECT * FROM trades t ASOF JOIN prices p ON t.sym = p.sym AND t.ts <=
   p.ts;
6
7  -- 用 ||> 之类取得最近（不论前后）的写法视版本而定

```

ASOF JOIN 条件中：只能有一个不等式（>=、>、<=、< 之一），等值条件可以有多个。

9.7.2 典型应用

- 成交价格归因；
- 用户操作日志关联"当时的配置版本"；
- IoT 传感器数据对齐到最近的校准记录；
- 汇率换算（按当天生效汇率）。

9.8 POSITIONAL JOIN

如果两张表已经按相同顺序排好，可以按"位置"配对，避免任何匹配开销：

```

1  SELECT a.x, b.y
2  FROM (SELECT * FROM range(5) AS t(x)) a
3  POSITIONAL JOIN (SELECT * FROM range(5) AS t(y)) b;

```

用途有限，但在特定已有排序数据的场景最快。

9.9 集合运算

9.9.1 UNION / UNION ALL

```

1  -- UNION: 去重（较慢，需要去重步骤）
2  SELECT region FROM sales WHERE category='图书'
3  UNION
4  SELECT region FROM sales WHERE category='食品';
5
6  -- UNION ALL: 不去重（快）
7  SELECT region FROM sales WHERE category='图书'
8  UNION ALL
9  SELECT region FROM sales WHERE category='食品';

```

经验：默认用 **UNION ALL**；只有在真的需要去重时才用 **UNION**，并且要知道它会带来一次哈希去重。

9.9.2 UNION BY NAME (DuckDB 特色)

```

1  -- 两表列不同也能合并，缺失补 NULL
2  SELECT * FROM sales_2025 UNION BY NAME SELECT * FROM sales_2026;

```

9.9.3 INTERSECT / EXCEPT

```

1  -- 交集：两个集合都有的
2  SELECT user_id FROM sales WHERE category='图书'
3  INTERSECT
4  SELECT user_id FROM sales WHERE category='食品';
5
6  -- 差集：在第一个但不在第二个
7  SELECT user_id FROM sales WHERE category='图书'
8  EXCEPT
9  SELECT user_id FROM sales WHERE category='食品';

```

`INTERSECT` / `EXCEPT` 也有 `ALL` 变体（`INTERSECT ALL`），保留重复计数。

9.9.4 用 JOIN 实现集合运算（大数据集更快）

```

1  -- INTERSECT 等价
2  SELECT DISTINCT a.k FROM a SEMI JOIN b USING (k);
3  -- EXCEPT 等价
4  SELECT DISTINCT a.k FROM a ANTI JOIN b USING (k);

```

9.10 连接的性能：优化器在背后做了什么

DuckDB 会自动：

1. **选择连接算法**：哈希连接（等值）、嵌套循环/IEJoin（不等值）、**Piecewise Merge Join**（有些情况）、ASOF 专用算法；
2. **决定连接顺序**：多表连接时用动态规划/贪婪策略重排；
3. **决定是否构建端**：通常把**小的一侧构建哈希表**，大的一侧流式探测；

- 4. **谓词下推**：先过滤再连接；
- 5. **分区并行**：按哈希把数据分到多个线程各自处理。

你可以用 EXPLAIN 看到这些（第 17 章）：

```
1 | EXPLAIN SELECT s.amount, u.plan FROM sales s JOIN users u USING
   | (user_id);
```

9.10.1 让连接更快的可控手段

手段	说明
先过滤再 JOIN	<code>FROM (SELECT * FROM big WHERE dt >= '2026-01-01') JOIN small</code> 通常更快
小表在前（某些情况）	DuckDB 会自动选择，但显式 CTE 有时更好
避免字符串连接键	改成 INTEGER 代理键，哈希成本大幅下降
避免函数包住连接键	<code>ON lower(a.k)=lower(b.k)</code> 无法哈希到同一个桶
用 SEMI/ANTI 代替 JOIN+DISTINCT	减少中间结果
维表去重	确保 JOIN 右表连接键唯一，避免行数膨胀

9.11 常见的连接错误

错误	症状	原因/修复
行数暴增	结果远大于左表	右表连接键不唯一 → 先 <code>GROUP BY</code> 去重
<code>NOT IN</code> 查不到任何东西	返回 0 行	子查询里有 NULL → 用 <code>ANTI JOIN</code> / <code>NOT EXISTS</code>
同名列歧义	<code>Binder Error: Ambiguous column</code>	用表别名限定 <code>s.created_at</code> 或改用 <code>USING</code>
左连接条件写在 WHERE 里	<code>LEFT JOIN</code> 变成了 <code>INNER</code>	<code>WHERE u.plan='企业版'</code> 过滤掉了 NULL 行 → 条件应写在 <code>ON</code> 里
连接很慢	—	检查连接键类型是否一致（VARCHAR vs INTEGER 会隐式转换导致无法优化）
内存暴涨	报 OOM	大表 JOIN 大表会建巨大哈希表；考虑分批或加 spill 空间（第 17 章）

最容易犯的错误示例：

```

1  -- ✗ 这会退化成 INNER JOIN!
2  SELECT * FROM sales s LEFT JOIN users u ON s.user_id = u.user_id
3  WHERE u.plan = '企业版';
4
5  -- ✓ 条件放 ON 里才保留所有销售行
6  SELECT * FROM sales s LEFT JOIN users u
7  ON s.user_id = u.user_id AND u.plan = '企业版';

```

实例演练

实例 9-1 · LEFT JOIN 的条件写在哪儿，行数差 3 倍

```

1  -- (A) 条件在 ON 里：保留所有订单
2  SELECT count(*) FROM sales s LEFT JOIN users u
3  ON s.user_id = u.user_id AND u.plan = '企业版';
4
5  -- (B) 条件在 WHERE 里：退化成 INNER JOIN
6  SELECT count(*) FROM sales s LEFT JOIN users u
7  ON s.user_id = u.user_id WHERE u.plan = '企业版';

```

cond_in_on	cond_in_where
200000	66888

少了 13 万行！因为 B 里未匹配的行 `u.plan` 为 NULL，被 `WHERE` 直接过滤掉了。这是最常见的 JOIN bug。

实例 9-2 · SEMI / ANTI 不会让行数膨胀

```

1  SELECT (SELECT count(*) FROM sales)
2  AS sales_rows,
3  (SELECT count(*) FROM sales s SEMI JOIN users u USING
4  (user_id)) AS semi_rows,
5  (SELECT count(*) FROM sales s JOIN users u USING (user_id))
6  AS inner_rows,
7  (SELECT count(*) FROM sales s ANTI JOIN users u USING
8  (user_id)) AS anti_rows;

```

sales_rows	semi_rows	inner_rows	anti_rows
200000	200000	200000	0

本例 `users.user_id` 唯一，所以 INNER 也没膨胀；SEMI 无论如何都不会膨胀。ANTI = 0 表示没有“孤儿订单”。

实例 9-3 · 维表不唯一会怎样

```
1  -- 造一张 user_id 重复的维表
2  CREATE OR REPLACE TABLE dup_user AS
3  SELECT user_id, plan FROM users
4  UNION ALL
5  SELECT user_id, plan FROM users
6  LIMIT 100;
7
8  SELECT count(*) AS exploded_rows FROM sales s JOIN dup_user d USING
   (user_id);
```

exploded_rows
24936

这 100 行维表对应了 sales 里 12468 笔订单，每笔被复制 2 倍 → 24936 行。
JOIN 前永远先确认维表连接键唯一：

```
1  SELECT user_id, count(*) FROM dup_user GROUP BY 1 HAVING count(*)
   > 1;
```

实例 9-4 · LATERAL 做"每组 Top-N"

```
1  SELECT r.region, top.category, top.gmv
2  FROM (SELECT DISTINCT region FROM sales) r,
3  LATERAL (
4      SELECT category, round(sum(amount)) AS gmv
5      FROM sales s WHERE s.region = r.region
6      GROUP BY category ORDER BY gmv DESC LIMIT 2
7  ) AS top
8  ORDER BY r.region LIMIT 4;
```

region	category	gmV
东北	图书	4228939
东北	服饰	4272939
华东	服饰	4218357
华东	食品	4247464

实例 9-5 · ASOF JOIN 归因 (含"早于首个报价"的坑)

见第 22 章案例 7，其中 trade_id = 5 (09:20 成交) 因为早于 MSFT 第一个快照 (09:30) 而被普通 ASOF JOIN 丢弃——用 ASOF LEFT JOIN 才能保留。

9.12 本章小结

- 连接类型：INNER / LEFT / RIGHT / FULL / CROSS / SEMI / ANTI / LATERAL / ASOF / POSITIONAL；
 - **SEMI/ANTI 不添加列、不膨胀行数**，是存在性判断的最佳选择；替代 `IN / NOT IN`；
 - `NOT IN` 遇 NULL 会返回空集 → 用 ANTI JOIN 或 NOT EXISTS；
 - **LEFT JOIN 的过滤条件必须写在 ON 里**，写在 WHERE 里会退化成 INNER JOIN；
 - LATERAL 实现"逐行子查询"，对 Top-N per group 很有用（但窗口函数常常更快）；
 - ASOF JOIN 是时序对齐的杀手锏，条件是"等值 + 唯一不等式"；
 - 性能优化核心：先过滤、用整数键、避免在键上套函数、维表去重。
-

动手练习

1. 统计每个 `plan` 的订单数与 GMV，并找出**没有任何**订单的 plan（用 ANTI JOIN）。
2. 用 `LATERAL` 找出每个 category 金额最高的 2 笔订单，再用窗口函数写一遍，比较二者计划差异（`EXPLAIN`）。
3. 构造一张价格快照表与交易表，用 ASOF JOIN 完成价格归因。
4. 用 skeleton + LEFT JOIN 生成"区域 × 日期"的完整矩阵，缺失 GMV 填 0。
5. 故意构造一个 LEFT JOIN 条件写在 WHERE 里的 bug，观察行数如何变化。

下一章

[第 10 章 CTE、窗口函数与 QUALIFY](#) —— SQL 里最"高级"、也最能少写代码的武器。

第 10 章 CTE、窗口函数与 QUALIFY

窗口函数是 SQL 的"分水岭": 会用的人, 写出来的 SQL 和不会用的人完全不是一个量级的简洁度。

本章覆盖:

1. CTE (`WITH`) 与递归 CTE;
2. 窗口函数的完整语法;
3. 帧 (Frame) 语义;
4. `QUALIFY`;
5. 常见分析模式 (排名、累计、同环比、会话化、漏斗)。

10.1 CTE: 给子查询起个名字

10.1.1 基本用法

```
1 WITH monthly AS (  
2     SELECT date_trunc('month', order_date) AS month,  
3           sum(amount) AS gmv  
4     FROM sales  
5     GROUP BY 1  
6 )  
7 SELECT month, gmv, lag(gmv) OVER (ORDER BY month) AS prev_month  
8 FROM monthly  
9 ORDER BY month;
```

10.1.2 链式 CTE (推荐的最佳实践)

```
1 WITH step1 AS (  
2     SELECT * FROM sales WHERE order_date >= DATE '2026-01-01'  
3 ),  
4 step2 AS (  
5     SELECT date_trunc('month', order_date) AS month, region,  
6           sum(amount) AS gmv  
7     FROM step1 GROUP BY 1, 2  
8 ),  
9 step3 AS (  
10    SELECT month, region, gmv,  
11          rank() OVER (PARTITION BY month ORDER BY gmv DESC) AS rk  
12    FROM step2  
13 )  
14 SELECT * FROM step3 WHERE rk <= 3 ORDER BY month, rk;
```

这种"管道式 CTE"让 SQL 从上往下线性可读, 是**推荐的复杂查询写法**。比嵌套子查询好维护得多。

10.1.3 CTE 的执行语义（重要）

在 DuckDB 里，CTE 是**内联展开**的（类似视图），不是预先物化成临时表：

- 优点：能被优化器整体优化（谓词下推到 CTE 内部）；
- 缺点：如果一个 CTE 被引用多次，**可能会被计算多次**。

如果发现多次引用导致重复计算，可以手动物化：

```
1 CREATE TEMP TABLE agg AS SELECT region, sum(amount) gmvl FROM sales
  GROUP BY 1;
2 SELECT ... FROM agg JOIN ... USING (region);
```

10.1.4 CTE 里的数据修改

```
1 WITH to_delete AS (
2     SELECT order_id FROM sales WHERE amount < 10
3 )
4 DELETE FROM sales WHERE order_id IN (SELECT order_id FROM to_delete);
```

10.2 递归 CTE

递归 CTE 用于**树形/图遍历**、**序列生成**。

```
1 WITH RECURSIVE t(n) AS (
2     SELECT 1 -- 锚定成员（起点）
3     UNION ALL
4     SELECT n + 1 FROM t WHERE n < 10 -- 递归成员（必须能收敛）
5 )
6 SELECT * FROM t;
```

10.2.1 经典场景一：生成阶乘/数列

```
1 WITH RECURSIVE fact(n, f) AS (
2     SELECT 1, 1
3     UNION ALL
4     SELECT n + 1, f * (n + 1) FROM fact WHERE n < 10
5 )
6 SELECT * FROM fact;
```

10.2.2 经典场景二：遍历组织架构树

```
1 CREATE TABLE emp (id INTEGER, name VARCHAR, mgr INTEGER);
2 INSERT INTO emp VALUES
3     (1, 'CEO', NULL),
4     (2, 'VP-A', 1),
5     (3, 'VP-B', 1),
6     (4, 'Dev-1', 2),
```

```

7      (5, 'Dev-2', 2);
8
9  WITH RECURSIVE org_tree AS (
10     SELECT id, name, mgr, 0 AS level, [name] AS path
11     FROM emp WHERE mgr IS NULL
12     UNION ALL
13     SELECT e.id, e.name, e.mgr, t.level + 1, list_prepend(e.name,
14        t.path)
15     FROM emp e JOIN org_tree t ON e.mgr = t.id
16 )
17 SELECT id, level, name, path FROM org_tree ORDER BY path;

```

10.2.3 经典场景三：日期序列补齐

```

1  WITH RECURSIVE cal(d) AS (
2     SELECT DATE '2026-01-01'
3     UNION ALL
4     SELECT d + INTERVAL 1 DAY FROM cal WHERE d < DATE '2026-01-31'
5 )
6  SELECT count(*) FROM cal;    -- 31

```

大多数情况下用 `generate_series` 更快更简单；递归 CTE 适合需要携带状态的场景（如上面的 path）。

10.2.4 注意事项

- 递归成员必须引用 CTE 自身一次，且**必须**在 `UNION ALL` 之后；
- 一定要有收敛条件，否则死循环直到爆内存；
- 深层递归会消耗内存，树深度 > 数千时考虑其他方案。

10.3 窗口函数：语法全景

```

1  function_name ( [expression] ) OVER (
2     [PARTITION BY partition_expression [, ...]]
3     [ORDER BY order_expression [ASC|DESC] [NULLS {FIRST|LAST}] [,
4     ...]]
5     [frame_clause]
6 )

```

核心概念：

- `PARTITION BY`：把数据切成互不干扰的“分区”，每个分区独立计算（类似 GROUP BY，但不减少行数）；
- `ORDER BY`：定义分区内的顺序（决定累计、排名的依据）；
- `frame_clause`：在已排序的分区内，进一步限定“窗口帧”（看哪些行）。

窗口函数不改变行数——这是它和 GROUP BY 的根本区别。每一行都保留，同时附带一个跨行计算出的值。

10.4 排名类函数

函数	同值时	结果特点
<code>row_number()</code>	给不同编号	1,2,3,4 (无并列)
<code>rank()</code>	并列后跳号	1,1,3,4
<code>dense_rank()</code>	并列不跳号	1,1,2,3
<code>percent_rank()</code>	相对排名	$(\text{rank}-1)/(\text{总行数}-1)$, 0~1
<code>cume_dist()</code>	累积分布	$\leq$ 当前值的行数占比
<code>ntile(n)</code>	分桶	均分成 n 个桶 (用于分位数切片)

```
1 SELECT
2     order_id, region, amount,
3     row_number() OVER (PARTITION BY region ORDER BY amount DESC) AS
   rn,
4     rank()      OVER (PARTITION BY region ORDER BY amount DESC) AS
   rk,
5     dense_rank() OVER (PARTITION BY region ORDER BY amount DESC) AS
   drk,
6     ntile(4)    OVER (PARTITION BY region ORDER BY amount DESC) AS
   quartile
7 FROM sales
8 QUALIFY rn <= 3
9 ORDER BY region, rn;
```

什么时候用哪个：需要"取每组的第 N 条"用 `row_number()` (因为它唯一)；做名次排行榜用 `rank()` 或 `dense_rank()`。

10.5 偏移类 (前后对比)

函数	说明
<code>lag(expr [, offset [, default]])</code>	前 offset 行的值
<code>lead(expr [, offset [, default]])</code>	后 offset 行的值
<code>first_value(expr)</code>	窗口第一行的值
<code>last_value(expr)</code>	窗口 当前帧 最后一行 (注意 frame 陷阱, 见 10.7)
<code>nth_value(expr, n)</code>	第 n 行

```

1  -- 环比：每个区域每月 GMV 与前月对比
2  WITH monthly AS (
3      SELECT region, date_trunc('month', order_date) AS month,
4             sum(amount) AS gmv
5      FROM sales GROUP BY 1, 2
6  )
7  SELECT
8      region, month, gmv,
9      lag(gmv, 1) OVER (PARTITION BY region ORDER BY month) AS
prev_gmv,
10     lag(gmv, 12) OVER (PARTITION BY region ORDER BY month) AS
same_month_last_year,
11     round(100.0 * (gmv - lag(gmv, 1) OVER (PARTITION BY region ORDER
BY month)))
12     / nullif(lag(gmv, 1) OVER (PARTITION BY region ORDER
BY month), 0), 2) AS mom_pct
13 FROM monthly
14 ORDER BY region, month;

```

用 `default` 参数避免首行 NULL: `lag(gmv, 1, 0)`。

10.6 聚合类窗口函数

任何聚合函数都可以加 `OVER`：

```

1  SELECT
2      order_id, region, amount,
3      sum(amount) OVER (PARTITION BY region)
4      AS region_total,
5      avg(amount) OVER (PARTITION BY region)
6      AS region_avg,
7      count(*) OVER (PARTITION BY region)
8      AS region_orders,
9      round(100.0 * amount / sum(amount) OVER (PARTITION BY region),
10     2) AS pct_of_region,
11     sum(amount) OVER (PARTITION BY region ORDER BY order_date
12     ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT
13     ROW) AS running_total
14 FROM sales
15 ORDER BY region, order_date
16 LIMIT 10;

```

10.7 窗口帧 (Frame)：最容易忽略的细节

10.7.1 帧 vs 分区

- **PARTITION BY**：把行分组（静态）；
- **ORDER BY**：组内排序；
- **frame**：在排序后，决定"当前行看到哪些行"（动态窗口）。

```
1 默认帧：  
2    - 有 ORDER BY: RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW  
3      （从分区开头到当前行——**累计**语义）  
4    - 无 ORDER BY: RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED  
5      FOLLOWING  
      （整个分区——**整体聚合**语义）
```

10.7.2 语法

```
1  {ROWS | RANGE} BETWEEN frame_start AND frame_end  
2  
3  frame_start: UNBOUNDED PRECEDING | n PRECEDING | CURRENT ROW  
4  frame_end:   CURRENT ROW | n FOLLOWING | UNBOUNDED FOLLOWING
```

10.7.3 ROWS vs RANGE 的区别（关键）

- **ROWS**：按物理行数计算；
- **RANGE**：按值域（相同 ORDER BY 值的行算一组）。

```
1  -- 演示数据  
2  CREATE TABLE demo AS SELECT * FROM (VALUES  
3    (1, 10), (2, 10), (3, 20), (4, 30)  
4  ) AS t(id, v);
```

```
1  SELECT id, v,  
2    sum(v) OVER (ORDER BY v ROWS BETWEEN UNBOUNDED PRECEDING AND  
3    CURRENT ROW) AS rows_sum,  
4    sum(v) OVER (ORDER BY v RANGE BETWEEN UNBOUNDED PRECEDING AND  
5    CURRENT ROW) AS range_sum  
6  FROM demo ORDER BY id;
```

结果：

id	v	rows_sum	range_sum
1	10	10	20 ← 因为 id=2 也是 10，RANGE 把它们算在一起
2	10	20	20
3	20	40	40
4	30	70	70

结论：有重复 ORDER BY 值时，`RANGE` 的累计值会"跳跃"，`ROWS` 则逐行累加。做**同行并列**的分析要注意这点。

10.7.4 移动平均

```
1 WITH daily AS (  
2     SELECT order_date, sum(amount) AS gmv FROM sales GROUP BY 1  
3 )  
4 SELECT  
5     order_date, gmv,  
6     avg(gmv) OVER (ORDER BY order_date ROWS BETWEEN 6 PRECEDING AND  
7     CURRENT ROW) AS ma7,  
8     avg(gmv) OVER (ORDER BY order_date ROWS BETWEEN 29 PRECEDING AND  
9     CURRENT ROW) AS ma30  
10 FROM daily  
11 ORDER BY order_date;
```

10.7.5 时间窗口：RANGE 与 INTERVAL（日期/时间戳专用）

```
1 -- 过去 7 天（按天数为窗口，自动处理缺失日期）  
2 SELECT ts, value,  
3     sum(value) OVER (ORDER BY ts RANGE BETWEEN INTERVAL 7 DAYS  
4     PRECEDING AND CURRENT ROW) AS rolling_7d  
5 FROM metrics  
6 ORDER BY ts;
```

这在处理**不规则时间序列**（有的天没有数据）时特别有用。`ROWS 7 PRECEDING` 会取前 7 行而非前 7 天。

10.7.6 last_value 的经典陷阱

```
1 -- ❌ 错误：默认帧到 CURRENT ROW, last_value 拿到的是当前行自己  
2 SELECT id, last_value(v) OVER (ORDER BY id) FROM demo;  
3  
4 -- ✅ 正确：把帧扩展到整个分区  
5 SELECT id, last_value(v) OVER (ORDER BY id  
6     ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS  
7     last_v  
8 FROM demo;  
9  
10 -- 或者更简单：倒序取 first_value  
11 SELECT id, first_value(v) OVER (ORDER BY id DESC) AS last_v FROM  
12 demo;
```

10.8 QUALIFY：窗口结果的过滤器

标准 SQL 中，不能在 WHERE 里用窗口函数（因为 WHERE 先于窗口计算）。传统做法要包一层子查询：

```

1  -- 传统：包一层
2  SELECT * FROM (
3      SELECT *, row_number() OVER (PARTITION BY region ORDER BY amount
4      DESC) rn
5      FROM sales
6  ) WHERE rn <= 3;

```

DuckDB 支持 `QUALIFY`：

```

1  SELECT *
2  FROM sales
3  QUALIFY row_number() OVER (PARTITION BY region ORDER BY amount DESC)
4  <= 3;

```

或者先在 `SELECT` 里算别名，再 `QUALIFY`：

```

1  SELECT region, order_id, amount,
2      row_number() OVER (PARTITION BY region ORDER BY amount DESC) AS
3      rn
4  FROM sales
5  QUALIFY rn <= 3
6  ORDER BY region, rn;

```

⚠ 已实测的规则 (DuckDB 1.5.5)：`QUALIFY` 里用别名 (如 `rn <= 3`) 有一个前提——同一个查询块的 `SELECT` 列表里必须出现窗口函数。

下面这样会报错：

```

1  SELECT * FROM (SELECT *, row_number() OVER (...) AS rn FROM
2  sales) QUALIFY rn = 1;
3  -- Binder Error: at least one window function must appear in the
4  SELECT column or QUALIFY clause

```

两种正确写法：

```

1  -- (A) 外层用 WHERE
2  SELECT * FROM (SELECT *, row_number() OVER (...) AS rn FROM
3  sales) WHERE rn = 1;
4  -- (B) 把窗口函数直接写在 QUALIFY 里
5  SELECT * FROM sales QUALIFY row_number() OVER (...) = 1;

```

执行顺序：FROM → WHERE → GROUP BY → HAVING → 窗口计算 → **QUALIFY** → SELECT 投影/ORDER BY → LIMIT。

10.8.1 QUALIFY 与聚合 + DISTINCT ON 对比

```

1  -- 每组 GMV 最高的品类，三种写法
2  -- (A) QUALIFY (推荐，最清晰)
3  SELECT region, category, sum(amount) AS gmv
4  FROM sales GROUP BY region, category

```

```

5  QUALIFY row_number() OVER (PARTITION BY region ORDER BY sum(amount)
   DESC) = 1;
6
7  -- (B) DISTINCT ON
8  SELECT DISTINCT ON (region) region, category, sum(amount) AS gmv
9  FROM sales GROUP BY region, category
10 ORDER BY region, gmv DESC;
11
12 -- (C) 子查询 + JOIN 回原聚合
13 SELECT r.region, c.category, c.gmv
14 FROM ...

```

10.9 命名窗口：少写重复代码

```

1  SELECT
2      order_id, amount,
3      sum(amount) OVER w      AS region_total,
4      rank() OVER w          AS rk,
5      lag(amount) OVER w      AS prev
6  FROM sales
7  WINDOW w AS (PARTITION BY region ORDER BY amount DESC);

```

也可以在 OVER 里基于已命名窗口追加：

```

1  SELECT count(*) OVER (w ROWS BETWEEN 5 PRECEDING AND CURRENT ROW)
2  FROM sales
3  WINDOW w AS (PARTITION BY region ORDER BY order_date);

```

10.10 实战模式集锦

10.10.1 累计 (Running Total) 与占比

```

1  WITH cat AS (
2      SELECT category, sum(amount) AS gmv FROM sales GROUP BY 1
3  )
4  SELECT category, gmv,
5      sum(gmv) OVER (ORDER BY gmv DESC ROWS UNBOUNDED PRECEDING) AS
   cum_gmv,
6      round(100.0 * sum(gmv) OVER (ORDER BY gmv DESC ROWS UNBOUNDED
   PRECEDING)
7          / sum(gmv) OVER (), 2) AS cum_pct
8  FROM cat ORDER BY gmv DESC;

```

注：ROWS UNBOUNDED PRECEDING 是 ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW 的简写。

10.10.2 同环比 (MoM / YoY)

```
1 WITH monthly AS (  
2     SELECT date_trunc('month', order_date) AS m, sum(amount) AS gmv  
3     FROM sales GROUP BY 1  
4 )  
5 SELECT m, gmv,  
6     lag(gmv, 1) OVER w AS prev_m,  
7     lag(gmv, 12) OVER w AS prev_y,  
8     round(100.0*(gmv - lag(gmv,1) OVER w) / nullif(lag(gmv,1) OVER  
9 w,0), 2) AS mom,  
10    round(100.0*(gmv - lag(gmv,12) OVER w) / nullif(lag(gmv,12) OVER  
11 w,0), 2) AS yoy  
12 FROM monthly  
13 WINDOW w AS (ORDER BY m)  
14 ORDER BY m;
```

10.10.3 会话化 (Sessionization)

问题：把同一用户间隔不超过 30 分钟的连续行为合并成一个 session。

```
1 WITH events AS (  
2     SELECT user_id, created_at AS ts  
3     FROM sales  
4 ),  
5 gaps AS (  
6     SELECT user_id, ts,  
7         lag(ts) OVER (PARTITION BY user_id ORDER BY ts) AS  
8 prev_ts  
9     FROM events  
10 ),  
11 flags AS (  
12     SELECT user_id, ts,  
13         CASE WHEN prev_ts IS NULL  
14             OR ts > prev_ts + INTERVAL 30 MINUTE  
15             THEN 1 ELSE 0 END AS is_new_session  
16     FROM gaps  
17 ),  
18 labeled AS (  
19     SELECT user_id, ts,  
20         sum(is_new_session) OVER (PARTITION BY user_id ORDER BY  
21 ts  
22                                     ROWS UNBOUNDED PRECEDING) AS  
23 session_id  
24     FROM flags  
25 )  
26 SELECT user_id, session_id,  
27     min(ts) AS session_start,  
28     max(ts) AS session_end,  
29     count(*) AS events  
30 FROM labeled  
31 GROUP BY user_id, session_id
```

```
29 ORDER BY user_id, session_start;
```

这是数据分析面试的经典题。核心套路：** lag 求间隔 → 打新会话标记 → 累加标记得到会话号**。

10.10.4 漏斗分析 (Funnel)

```
1 WITH steps AS (  
2     SELECT user_id,  
3         max(CASE WHEN status='已支付' THEN created_at END) AS  
step1_at,  
4         max(CASE WHEN status='已发货' THEN created_at END) AS  
step2_at,  
5         max(CASE WHEN status='已完成' THEN created_at END) AS  
step3_at  
6     FROM sales GROUP BY user_id  
7 )  
8 SELECT  
9     count(*)  
AS total_users,  
10    count(step1_at)  
AS reached_step1,  
11    count(CASE WHEN step2_at > step1_at THEN 1 END)  
AS reached_step2,  
12    count(CASE WHEN step3_at > step2_at AND step2_at > step1_at THEN  
1 END) AS reached_step3  
13 FROM steps;
```

10.10.5 去重保留最新一条 (SCD / 拉链表)

```
1 SELECT *  
2 FROM (  
3     SELECT *,  
4         row_number() OVER (PARTITION BY user_id ORDER BY updated_at  
DESC) AS rn  
5     FROM user_snapshots  
6 )  
7 QUALIFY rn = 1;
```

10.10.6 首末次行为

```
1 SELECT user_id,  
2     first_value(amount) OVER (PARTITION BY user_id ORDER BY  
created_at) AS first_amount,  
3     last_value(amount) OVER (PARTITION BY user_id ORDER BY created_at  
4         ROWS BETWEEN UNBOUNDED PRECEDING AND  
UNBOUNDED FOLLOWING) AS last_amount,  
5     count(*) OVER (PARTITION BY user_id) AS orders  
6 FROM sales;
```

10.10.7 相邻行差值与异常检测

```
1 WITH daily AS (  
2     SELECT order_date, sum(amount) gmv FROM sales GROUP BY 1  
3 ),  
4 d AS (  
5     SELECT order_date, gmv,  
6           lag(gmv) OVER (ORDER BY order_date) AS prev,  
7           avg(gmv) OVER (ORDER BY order_date ROWS BETWEEN 7  
8 PRECEDING AND 1 PRECEDING) AS recent_avg,  
9           stddev(gmv) OVER (ORDER BY order_date ROWS BETWEEN 7  
10 PRECEDING AND 1 PRECEDING) AS recent_sd  
11     FROM daily  
12 )  
13 SELECT order_date, gmv, recent_avg,  
14        round((gmv - recent_avg) / nullif(recent_sd, 0), 2) AS  
15        z_score  
16 FROM d  
17 WHERE ABS((gmv - recent_avg) / nullif(recent_sd, 0)) > 3    -- 3σ 异常  
18 ORDER BY order_date;
```

10.11 性能注意事项

关注点	说明
分区基数	高基数 <code>PARTITION BY</code> (如 <code>user_id</code>) 需要大量内存排序/哈希; 必要时分批
排序开销	每个窗口的 <code>ORDER BY</code> 都可能触发一次排序; 尽量复用同一个 <code>WINDOW</code> 定义
帧大小	<code>UNBOUNDED FOLLOWING</code> 需要缓冲整个分区; 尽量用有限帧
多窗口不同排序	会触发多次排序 (DuckDB 有时能复用), 可用 CTE 拆分控制
QUALIFY 不等于提前裁剪	窗口先算出来再过滤; 要减少数据量请在 <code>WHERE</code> 里先过滤行

```
1 -- ✅ 先过滤, 再做窗口计算  
2 SELECT * FROM (  
3     SELECT * FROM sales WHERE order_date >= DATE '2026-03-01'  
4 ) QUALIFY row_number() OVER (PARTITION BY region ORDER BY amount DESC)  
5 <= 3;
```

实例演练

实例 10-1 · 累计、移动平均、环比一次算完

```
1 WITH daily AS (  
2     SELECT order_date, sum(amount) AS gmv FROM sales GROUP BY 1  
3 )  
4 SELECT order_date, gmv,  
5     lag(gmv) OVER w  
6     AS prev_gmv,  
7     round(100.0*(gmv - lag(gmv) OVER w) / nullif(lag(gmv) OVER  
8 w,0), 2) AS mom_pct,  
9     round(avg(gmv) OVER (ORDER BY order_date ROWS BETWEEN 6  
10 PRECEDING AND CURRENT ROW), 2) AS ma7,  
11     sum(gmv) OVER (ORDER BY order_date ROWS UNBOUNDED PRECEDING)  
12     AS cum_gmv  
13 FROM daily WINDOW w AS (ORDER BY order_date) ORDER BY order_date LIMIT  
14 3;
```

order_date	gmv	prev_gmv	mom_pct	ma7	cum_gmv
2026-01-01	560142.74	NULL	NULL	560142.74	560142.74
2026-01-02	561437.64	560142.74	0.23	560790.19	1121580.38
2026-01-03	561609.71	561437.64	0.03	561063.36	1683190.09

实例 10-2 · QUALIFY 取每组 Top-N（两种写法等价）

```
1 -- (A) 聚合 + 窗口 + QUALIFY 一气呵成  
2 SELECT region, category, sum(amount) AS gmv  
3 FROM sales GROUP BY region, category  
4 QUALIFY row_number() OVER (PARTITION BY region ORDER BY sum(amount)  
5 DESC) <= 2  
6 ORDER BY region, gmv DESC;  
7  
8 -- (B) SELECT 里有窗口函数时, QUALIFY 可以用别名  
9 SELECT region, category, sum(amount) AS gmv,  
10 row_number() OVER (PARTITION BY region ORDER BY sum(amount)  
11 DESC) AS rn  
12 FROM sales GROUP BY region, category  
13 QUALIFY rn <= 2 ORDER BY region, gmv DESC;
```

实例 10-3 · 去重保留最新一条（三种写法实测都返回 800 行）

```
1 -- (A) 子查询 + WHERE（最通用）  
2 SELECT count(*) FROM (  
3     SELECT *, row_number() OVER (PARTITION BY user_id ORDER BY  
4 created_at DESC) rn  
5     FROM sales  
6 ) WHERE rn = 1;
```

```

7  -- (B) 窗口函数写在 QUALIFY 里（最简洁）
8  SELECT count(*) FROM (
9      SELECT * FROM sales
10     QUALIFY row_number() OVER (PARTITION BY user_id ORDER BY
11        created_at DESC) = 1
12 );
13 -- (C) DISTINCT ON (PostgreSQL 风格)
14 SELECT count(*) FROM (
15     SELECT DISTINCT ON (user_id) user_id, order_id, amount FROM
16     sales
17     ORDER BY user_id, created_at DESC
18 );

```

三者都返回 **800** (= 用户数)，说明结果一致。

实例 10-4 · 递归 CTE 生成日历

```

1  WITH RECURSIVE cal(d) AS (
2      SELECT DATE '2026-01-01'
3      UNION ALL
4      SELECT d + INTERVAL 1 DAY FROM cal WHERE d < DATE '2026-01-10'
5  )
6  SELECT count(*) AS days FROM cal;  -- 10

```

大多数场景用 `generate_series` 更快；递归 CTE 适合需要携带状态（如组织路径）的场景。

10.12 本章小结

- CTE 让复杂查询线性可读；DuckDB 会内联展开它（所以多次引用可能重复计算）；
- 递归 CTE 处理树/图/序列生成，务必设收敛条件；
- 窗口函数**不减少行数**，`PARTITION BY` 分区、`ORDER BY` 排序、`frame` 限定窗口范围；
- `row_number()` 取第 N 条、`rank()/dense_rank()` 排名、`lag/lead` 前后对比、聚合 + `OVER` 做累计；
- 帧默认规则要记牢，**RANGE 看值、ROWS 看行**；不要忘记 `last_value` 的帧陷阱；
- `QUALIFY` 让窗口过滤不必再包一层子查询；
- 性能：先用 `WHERE` 减行、复用同一 `WINDOW`、注意高基数分区的内存。

动手练习

1. 用窗口函数给每个 region 的订单按金额排名，并用 `QUALIFY` 取出每组前 3。
2. 计算每个 region 每月 GMV 的环比 (MoM)。
3. 计算 7 日移动平均 (MA7)，比较 `ROWS 6 PRECEDING` 与 `RANGE INTERVAL 6 DAYS PRECEDING` 的区别。

4. 用会话化套路，把同一 user 间隔 < 5 天的订单合并为一次购买批次。
5. 写一条 SQL 找出"日 GMV 偏离近 7 天均值 2 倍标准差以上"的异常日期。

下一章

[第 11 章 嵌套数据类型的世界](#) —— LIST、STRUCT、MAP、JSON/VARIANT：现代数据处理的必修课。

第 11 章 嵌套数据类型的世界

传统 SQL 只有“扁平表”，而现实世界的数据（JSON、Protobuf、事件流、宽表）天生嵌套。DuckDB 的嵌套类型支持是它与 SQLite、MySQL 拉开差距的重要能力。

读完本章你会掌握：LIST、STRUCT、MAP、UNION、ARRAY，以及 JSON / VARIANT 的处理方式。

11.1 为什么需要嵌套类型

一个典型的订单：

```
1 {
2   "order_id": 1001,
3   "customer": {"id": 5, "name": "Alice", "vip": true},
4   "items": [
5     {"sku": "A1", "qty": 2, "price": 19.9},
6     {"sku": "B7", "qty": 1, "price": 99.0}
7   ],
8   "tags": ["promo", "flash-sale"]
9 }
```

三种建模方式：

方式	做法	优点	缺点
A 拆表	订单表 + 明细表 + 标签表	规范、易更新	查询要多次 JOIN，ETL 复杂
B 嵌套表	一行 = 一个订单，items 是 LIST<STRUCT>	读快、无 JOIN、贴近源结构	更新单条明细麻烦
C 折中	保留扁平主表 + 少量嵌套列 (tags/customer)	平衡	—

DuckDB 支持第三种，且在分析场景下它通常最快（少 JOIN、少随机 IO）。

11.2 LIST（数组）

11.2.1 创建与查询

```

1  -- 字面量: [...]
2  SELECT [1, 2, 3] AS nums;
3  SELECT list_value(1, 2, 3) AS nums;      -- 等价
4
5  -- 带类型标注
6  SELECT [1, 2, 3]::BIGINT[] AS nums;
7  SELECT ['a','b']::VARCHAR[] AS strs;
8
9  -- 从聚合产生
10 SELECT list(amount) AS amounts FROM sales LIMIT 3;
11 SELECT list(DISTINCT region) FROM sales;

```

11.2.2 访问元素

```

1  SELECT list_extract([10, 20, 30], 2) AS v1;  -- 20
2  SELECT ([10, 20, 30])[2] AS v2;  -- 20 (下标语法)
3  SELECT [10,20,30][-1] AS v3;  -- 30 (负索引, 从尾部)
4
5  -- 切片
6  SELECT list_slice([1,2,3,4,5], 2, 4) AS sub;  -- [2,3,4]
7  SELECT list_slice([1,2,3,4,5], 2, 5, 2) AS st;  -- 步长 2: [2,4]

```

⚠ **LIST 下标是 1-based** (与 SQL/PostgreSQL 一致), 区别于 Python 的 0-based。这是最常见的坑。

11.2.3 列表运算

```

1  SELECT
2      len([1,2,3]) AS n,      -- 长度
   (也可用 array_length)
3      array_length([1,2,3], 1) AS n2,
4      list_concat([1,2], [3,4]) AS cat,  --
   [1,2,3,4]
5      list_contains([1,2,3], 2) AS has,  -- true
6      list_position([1,2,3], 3) AS pos,  -- 3
7      list_distinct([1,1,2]) AS dd,  --
   [1,2]
8      list_sort([3,1,2]) AS sorted,  --
   [1,2,3]
9      reverse([1,2,3]) AS rev,  --
   [3,2,1]
10     flatten([[1,2],[3]]) AS flat,  --
   [1,2,3]
11     list_slice(lst, 1, len(lst)) AS all
12 FROM (SELECT [1,2,3] AS lst);

```

11.2.4 lambda 三件套（重点）

```
1  -- 变换
2  SELECT list_transform([1, 2, 3], lambda x: x * 2) AS doubled;
   -- [2,4,6]
3  SELECT list_transform([1, 2, 3], lambda x, i: x + i) AS with_idx;
   -- [2,4,6]
4
5  -- 过滤
6  SELECT list_filter([1,2,3,4], lambda x: x % 2 = 0) AS evens;
   -- [2,4]
7
8  -- 归约
9  SELECT list_reduce([1,2,3,4], lambda a, b: a + b) AS sum_all;
   -- 10
10 SELECT list_reduce([1,2,3,4], lambda a, b: greatest(a,b)) AS
    max_all; -- 4
```

lambda 语法（已实测 1.5.5）：推荐新写法 `lambda x: 表达式`，多参数写 `lambda x, i: ...` 或 `lambda a, b: ...`。

旧写法 `x -> ...` 仍可运行，但会打印弃用警告：

```
Deprecated lambda arrow (->) detected. Please transition to the new
lambda syntax.
```

11.2.5 unnest: 把列表展开成行

```
1  SELECT unnest([10, 20, 30]) AS v;
2  -- 结果 3 行: 10, 20, 30
3
4  -- 带上序号
5  SELECT unnest([10,20,30]) AS v, generate_subscripts([10,20,30], 1)
   AS idx;
6
7  -- 从表里的 list 列展开（保留其他列）
8  SELECT user_id, tag
9  FROM (VALUES (1, ['a','b','c']), (2, ['x'])) AS t(user_id, tags),
10       unnest(t.tags) AS tag;
```

11.2.6 行 → 列表（聚合回来）

```
1  SELECT list(tag) AS tags, list(DISTINCT tag) AS uniq
2  FROM (VALUES ('a'),('b'),('a')) AS t(tag);
```

经典“标签聚合”：

```

1 SELECT user_id,
2     string_agg(DISTINCT category, ',') AS cats_csv,
3     list(DISTINCT category)           AS cats_arr,
4     len(list(DISTINCT category))      AS n_cats
5 FROM sales GROUP BY user_id LIMIT 5;

```

11.2.7 列表作为"多行 → 单行"的桥

这是 Parquet 中最常见的模式之一：

```

1 -- 每行：一个用户，其所有订单金额组成的价格序列
2 SELECT user_id,
3     list(amount ORDER BY created_at) AS amounts_in_order,
4     list_sum(list(amount))           AS total_check
5 FROM sales GROUP BY user_id;

```

11.3 STRUCT（结构体）

11.3.1 创建与访问

```

1 SELECT {'name': 'Alice', 'age': 30, 'vip': true} AS person;
2
3 -- 点语法
4 SELECT person.name FROM (SELECT {'name': 'Alice'} AS person); --
   Alice
5
6 -- struct_extract
7 SELECT struct_extract({'x':1, 'y':2}, 'x');
8
9 -- 下标
10 SELECT ({'x':1, 'y':2}).y;

```

注意：点语法访问 struct 字段时，如果 struct 是列而字段名与关键字冲突，用双引号：
s."year"。

11.3.2 构造

```

1 SELECT {'name': 'Alice', 'age': 30} AS s;
2
3 -- 动态构造（把多行/多列合成 struct）
4 SELECT {'region': region, 'gmv': sum(amount)} AS kv
5 FROM sales GROUP BY region;
6
7 -- 用 struct_pack
8 SELECT struct_pack(name := 'Alice', age := 30) AS s;
9
10 -- 从多列组合成一行
11 SELECT row(first_name, last_name) AS full FROM customers;

```

11.3.3 struct.* 展开为多列

```
1 SELECT s.* FROM (SELECT {'a':1,'b':2} AS s);
2 -- 得到两列: a, b
```

11.3.4 嵌套 struct

```
1 SELECT {'outer': {'inner': 42}} AS deep;
2 SELECT (('outer' : {'inner': 42})).outer.inner; -- 42 (链式访问)
```

常用便利: `struct.*`、`unnest(struct)` (把 struct 展开成横着的多列) 等。

11.4 MAP (键值映射)

```
1 SELECT MAP{'a': 1, 'b': 2} AS m;
2
3 -- 访问 (返回 list, 因为是可能有多个 value)
4 SELECT (MAP{'a':1,'b':2})['a']; -- [1]
5
6 -- 得到单值
7 SELECT element_at(MAP{'a':1,'b':2}, 'a'); -- 1
8
9 -- 键 / 值列表
10 SELECT map_keys(m), map_values(m)
11 FROM (SELECT MAP{'a':1,'b':2} AS m);
12
13 -- 从两列构造
14 SELECT map(list(k), list(v)) AS m
15 FROM (VALUES ('x',1),('y',2)) AS t(k, v);
16
17 -- element_at 找不到返回 NULL
18 SELECT element_at(MAP{'a':1}, 'zzz'); -- NULL
```

MAP 的内部表示其实是 `LIST<STRUCT<key, value>>`, 所以它的很多操作需要 `list_*` 函数。

MAP 的典型用途:

- 动态属性 (不同用户有不同的字段) ;
- 计数型 map (`histogram()` 返回值就是 MAP) ;
- 多语言字典。

```
1 SELECT histogram(region) AS h FROM sales;
2 SELECT unnest(histogram(region)) AS kv FROM sales;
```

11.5 UNION (联合类型)

UNION 类似 C 的 union / TypeScript 的联合类型：一个值可以是几种类型之一。

```
1 SELECT union_value(a := 1) AS u_int;
2 SELECT union_value(b := 'hello') AS u_str;
3
4 -- 取值
5 SELECT union_extract(u, 'a') FROM t;    -- 若 u 存的是 tag=a 则返回 1,
      否则 NULL
6 SELECT union_tag(u) FROM t;             -- 'a' / 'b'
```

定义类型：

```
1 CREATE TABLE mixed (
2     v UNION(a INTEGER, b VARCHAR, c DATE)
3 );
```

UNION 在 DuckDB 中相对少用。处理"半结构化 JSON 字段"时，更多用 `JSON` 或 `VARIANT`。

11.6 ARRAY（定长数组）

（较新版本提供，早期实验版本需要通过扩展或设置开启。）

```
1 SELECT ARRAY[1, 2, 3] AS a;
2 SELECT [1,2,3]::INTEGER[3] AS fixed;
3 SELECT array_length(a), a[1] FROM (SELECT ARRAY[1,2,3] AS a);
```

与 LIST 的区别：ARRAY 维度固定并写入模式，适合向量/嵌入（embedding）这类固定长度数据。

VSS 扩展里做向量相似度检索时就用 `ARRAY<FLOAT>` 存 embedding（第 19 章）。

11.7 JSON / VARIANT 的处理

11.7.1 从字符串解析

```
1 SELECT json('{ "a":1, "b":[1,2] }') AS j;
2 SELECT '{ "a":1 }'::JSON AS j;
3 SELECT json_valid('{ "a":1 }') AS ok;
4 SELECT try_cast(bad_str AS JSON) AS maybe_null;    -- 脏数据安全
```

11.7.2 提取

两种运算符：

运算符	返回类型
-----	------

运算符	返回类型
<code>-></code>	JSON
<code>->></code>	VARCHAR (去引号的字符串)

```

1 SELECT
2     j->'user'                AS user_obj,
3     j->'user'->>'name'        AS name,
4     j->'items'->0->>'sku'     AS first_sku,
5     json_extract(j, '$.user.name') AS fn_style,
6     json_extract_string(j, '$.user.name') AS fn_str
7 FROM (SELECT '{"user":{"name":"Alice"},"items":[{"sku":"A1"}]}':JSON
      AS j);

```

11.7.3 数组

```

1 SELECT
2     json_array_length('[1,2,3]') AS n,
3     json_extract('[1,2,3]', '$[1]') AS second,
4     json_contains('[1,2,3]', '2') AS has
5 FROM t;

```

11.7.4 生成 JSON

```

1 SELECT to_json({'a': 1, 'b': [1,2]}) AS obj;
2 SELECT json_object('k1', 1, 'k2', 'two') AS obj2;
3 SELECT json_array(1, 'two', 3.0) AS arr;
4 SELECT json_group_array(name) AS names FROM
  users;
5 SELECT json_group_object(key, value) AS map1 FROM
  config;

```

11.7.5 JSON → 关系 (宝藏功能)

把 JSON 数组变成真正的行:

```

1 SELECT json_each(j) FROM t;           -- 每行一对 key/value
2 SELECT json_tree(j) FROM t;          -- 递归展开整棵树

```

```

1 SELECT key, value
2 FROM json_each('{"region":"华东","gmv":1000}');

```

(不同版本返回结构可能是 STRUCT, 需要用 `.key` / `.value` 取出, 请先用 `DESCRIBE` 确认列名。)

11.7.6 从JSON生成表: read_json与json_transform

```
1 SELECT * FROM read_json('events/*.json');
2 SELECT type_of->>'x' FROM t;
```

把JSON字符串批量解析成 struct:

```
1 SELECT json_transform(payload,
2     '{"user_id":"BIGINT","amount":"DOUBLE"}') AS rec
3 FROM raw_events;
```

11.7.7 VARIANT (1.5 新增)

```
1 SELECT '{"a":1,"b":{"c":2}}'::VARIANT AS v;
2 SELECT v.a, v.b.c FROM t;                -- 直接用点语法取出嵌套字段参
    与运算
3 SELECT variant_type(v) FROM t;
```

VARIANT 适合模式变化剧烈、但**每次只访问少数几个字段**的场景。如果需要大量分析，还是应该把它们投影成真正的列（见 11.9）。

11.8 嵌套数据的查询模式总结

需求	写法
列表 → 多行	<code>unnest(list_col)</code>
多行 → 列表	<code>list(x)</code> / <code>array_agg(x)</code>
结构体 → 多列	<code>struct_col.*</code>
多列 → 结构体	<code>{'a': c1, 'b': c2}</code> 或 <code>row(c1,c2)</code>
列表 → 标量	<code>list_sum(x)</code> 、 <code>list_avg(x)</code> 、 <code>len(x)</code> 、 <code>list_aggregate(x, 'max')</code>
列表 → 列表	<code>list_transform</code> / <code>list_filter</code> / <code>list_sort</code> / <code>flatten</code>
Map → 行	<code>unnest(map)</code>
JSON → 行	<code>json_each</code> / <code>json_tree</code>
判断包含	<code>list_contains(l, x)</code> / <code>json_contains(j, 'k')</code>

通用的聚合形式:

```

1 SELECT list_aggregate(list_col, 'sum') AS s,
2        list_aggregate(list_col, 'median') AS m
3 FROM t;

```

11.9 实践：把嵌套数据"降维"为分析友好模型

真实工作流推荐 **bronze / silver / gold** 三层：

```

1  -- bronze: 原样保留（一行一条原始事件，payload 是 JSON/VARIANT）
2  CREATE OR REPLACE TABLE bronze_events AS
3  SELECT json(payload_str) AS payload, ingest_time
4  FROM read_csv_auto('raw/*.csv');
5
6  -- silver: 投影出常用字段（保留少量嵌套）
7  CREATE OR REPLACE TABLE silver_events AS
8  SELECT
9      payload->>'event_type' AS event_type,
10     try_cast(payload->>'user_id' AS BIGINT) AS user_id,
11     try_cast(payload->>'amount' AS DOUBLE) AS amount,
12     try_cast(payload->>'ts' AS TIMESTAMPTZ) AS ts,
13     payload->'items' AS items_json    -- 保留嵌套
14  FROM bronze_events;
15
16  -- gold: 彻底扁平化 + 聚合
17  CREATE OR REPLACE TABLE gold_user_daily AS
18  SELECT
19      user_id,
20      ts::DATE AS d,
21      count(*) AS events,
22      sum(amount) AS amount_sum,
23      list(DISTINCT event_type) AS event_types
24  FROM silver_events
25  WHERE user_id IS NOT NULL
26  GROUP BY 1, 2;

```

原则：原始数据保留嵌套（忠实、灵活）；分析层尽量扁平（快、易用）。两者之间的转换就是 ETL。

11.10 嵌套类型的性能注意事项

1. 嵌套**不等于**慢：LIST/STRUCT 在列存里按子元素逐列存储，常常比 JSON 字符串快 10 倍以上；
2. 但是**反复访问深层嵌套**会有解引用成本，热点字段请在 silver 层提取出来；
3. `unnest` 会产生行膨胀，警惕在已经很大的表上做 `unnest` 之后再 JOIN；
4. `list_transform` 的 lambda 会对每行执行，复杂 lambda 有 CPU 成本；
5. 把 JSON 列持久化为 `::JSON` 而非 VARCHAR，查询时省去重复 parse；

6. 写入 Parquet 时，嵌套结构会被完整保留（Parquet 原生支持）。

实例演练

实例 11-1 · 行 → 列表（把每个用户的行为收进一行）

```
1 SELECT user_id,  
2         len(list(DISTINCT category)) AS n_cats,  
3         list(DISTINCT category)      AS cats,  
4         round(list_sum(list(amount)), 2) AS total  
5 FROM sales  
6 GROUP BY user_id  
7 ORDER BY user_id  
8 LIMIT 1;
```

```
1 user_id = U0001  
2 n_cats = 4  
3 cats = [图书, 电子产品, 服饰, 食品]  
4 total = 129382.88
```

实例 11-2 · 列表 → 行（unnest）

```
1 SELECT unnest([10, 20, 30]) AS v;      -- 3 行: 10 / 20 / 30  
2  
3 -- 从表里的 list 列展开，保留其它列  
4 SELECT unnest(list(DISTINCT category)) AS cat  
5 FROM sales  
6 GROUP BY user_id  
7 LIMIT 2;
```

实例 11-3 · lambda 三件套（新语法）

```
1 SELECT list_transform([1,2,3], lambda x: x*2)      AS doubled,  
2        list_filter([1,2,3,4], lambda x: x%2=0)     AS evens,  
3        list_reduce([1,2,3,4], lambda a, b: a+b)    AS sum_all;
```

doubled	evens	sum_all
[2, 4, 6]	[2, 4]	10

旧写法 `x -> x*2` 在 1.5.5 仍可运行，但会打印弃用警告，建议全面改用 `lambda`。

实例 11-4 · STRUCT 与 MAP

```
1 SELECT s.* FROM (SELECT {'region':'华东','gmv':1000} AS s);
2 -- region = 华东 ; gmv = 1000
3
4 SELECT map_keys(h) AS k, map_values(h) AS v
5 FROM (SELECT histogram(region) AS h FROM sales);
```

k	v
[东北, 华东, 华中, 华北, 华南, 西南]	[33098, 32980, 33400, 33594, 33751, 33177]

`histogram()` 的返回值就是 `MAP`，正好演示“聚合结果本身就是嵌套类型”。

实例 11-5 · 嵌套建模 vs 拆表（选型对照）

```
1 -- 嵌套：一行一个用户（读快、无 JOIN）
2 CREATE OR REPLACE TABLE user_profile AS
3 SELECT user_id,
4         list(DISTINCT category) AS categories,
5         sum(amount) AS gmv
6 FROM sales GROUP BY user_id;
7
8 -- 拆表：一行一个用户×品类（便于增量更新）
9 CREATE OR REPLACE TABLE user_category AS
10 SELECT user_id, category, sum(amount) AS gmv
11 FROM sales GROUP BY user_id, category;
```

	行数	适合
嵌套 <code>user_profile</code>	800	报表、特征、API 输出
拆表 <code>user_category</code>	3200	增量更新、按品类过滤

11.11 本章小结

- `LIST`（1-based 下标）、`STRUCT`（点语法访问）、`MAP`（键值）、`UNION`（带标签的联合）、`ARRAY`（定长）构成 DuckDB 的嵌套体系；
- **lambda 三件套** `list_transform` / `list_filter` / `list_reduce` 是处理 `LIST` 的核心武器；
- `unnest` 把嵌套拆成行；聚合函数 `list()` 把行聚成嵌套；两者互为逆运算；
- `JSON` 用 `->`（`JSON`）与 `->>`（字符串）提取，`avours json_each/json_tree` 做结构化展开；
- `VARIANT`（1.5+）点语法访问更快，适合半结构化场景；
- 实战套路：bronze 保留嵌套 → silver 投影 → gold 扁平聚合。

动手练习

1. 把 `sales` 按 `user_id` 聚合成一行一个用户（含订单列表、总额、最常购品类）。
2. 用 `list_transform` + `list_filter` 算出每个用户"金额大于 500 的订单金额之和"。
3. 构造含 JSON 字段的表，用 `->>` 提取字段，再把它 `unnest` 成多行。
4. 用 `unnest` 把 `histogram(region)` 的 MAP 展开为 (key, value) 两列。
5. 挑战：实现"购物篮分析"——找出每个用户的品类组合
(`list_sort(list_distinct(...))`)，再统计每种组合出现的次数。

下一章

[第 12 章 读取外部文件](#) —— CSV、Parquet、JSON、Excel、Hive 分区、多文件合并、HTTP/S3。这是 DuckDB 最实用的能力。

第 12 章 读取外部文件

这是 DuckDB 最能改变你日常工作的能力：**不需要写任何导入代码，直接 SQL 查询文件。**

本章覆盖 CSV、Parquet、JSON/NDJSON、Excel、多文件/分区目录、HTTP(S)、压缩文件，以及每个读取函数的常用参数与踩坑点。

12.1 三种等价写法

```
1  -- (A) replacement scan: 直接把文件名当表名（最省事，类型自动推断）
2  SELECT * FROM 'data.parquet';
3  SELECT * FROM 'data.csv';
4  SELECT * FROM 'https://example.com/data.parquet';
5
6  -- (B) 显式读取函数（可以传参数，推荐在正式代码中使用）
7  SELECT * FROM read_parquet('data.parquet');
8  SELECT * FROM read_csv_auto('data.csv');
9
10 -- (C) 显式 + 完整参数
11 SELECT * FROM read_csv('data.csv', header = true, delim = '|',
12                          columns =
13                          {'id': 'BIGINT', 'name': 'VARCHAR', 'ts': 'DATE'});
```

何时用哪种？

- 探索/临时查询：(A)；
- 需要控制参数（分隔符、类型、压缩等）：(B)/(C)；
- 注意 (A) 对 CSV 是**采样推断类型**（见 12.2），若推断不准请改用 (C) 显式指定。

12.2 CSV：看起来简单，坑最多

12.2.1 基本读取

```
1  SELECT * FROM read_csv('data.csv');           -- 默认 header=true,
2  auto_detect=true（不同版本默认可能不同）
3  SELECT * FROM read_csv_auto('data.csv');       -- 明确开启自动推断
4  SELECT * FROM read_csv('data.csv', header = false, columns =
5  {'c0': 'BIGINT', 'c1': 'VARCHAR'});
```

如果是压缩 CSV，**自动识别** gzip/zstd：

```
1  SELECT * FROM 'data.csv.gz';
2  SELECT * FROM read_csv('data.csv.zst', compression = 'zstd');
```

12.2.2 自动推断的原理与陷阱

DuckDB 会：

1. 抽取 head（若干行/样本）做采样；
2. 为每个字段计算"可能的类型候选"；
3. 统一选择一个能容纳所有样本的型别。

问题：采样可能导致后续行溢出。

```
1  -- 前 1000 行是整数，第 1001 行出现 "abc"
2  SELECT * FROM 'mixed.csv';
3  -- Conversion Error: Could not convert string 'abc' to INT32
```

解决方案：

```
1  -- (1) 增加采样量
2  SELECT * FROM read_csv('mixed.csv', sample_size = -1);      -- 全量推断
   (慢但准)
3
4  -- (2) 直接指定类型
5  SELECT * FROM read_csv('mixed.csv', columns =
   {'a': 'VARCHAR', 'b': 'DOUBLE'});
6
7  -- (3) 全部当作 VARCHAR 读入，之后自己处理（最保险）
8  SELECT try_cast(a AS BIGINT) AS a FROM read_csv('d.csv', all_varchar =
   true);
```

12.2.3 常用参数一览

参数	说明	示例
<code>header</code>	是否含表头	<code>header = true</code>
<code>names</code>	无表头时提供列名	<code>names = ['id', 'name']</code>
<code>columns</code>	显式指定列名与类型	<code>columns =</code> <code>{'id': 'BIGINT', 'd': 'DATE'}</code>
<code>delim / sep</code>	分隔符	<code>delim = ';' </code>
<code>quote</code>	引号字符	<code>quote = '"' </code>
<code>escape</code>	转义字符	<code>escape = '\\' </code>
<code>nullstr</code>	NULL 的表示串	<code>nullstr = 'NULL' </code> （可传多个）
<code>skip</code>	跳过开头 N 行	<code>skip = 2</code>

参数	说明	示例
<code>sample_size</code>	类型推断采样行数	<code>sample_size = -1</code> 全量
<code>all_varchar</code>	全部读成字符串	<code>all_varchar = true</code>
<code>dateformat</code> / <code>timestampformat</code>	日期格式	<code>dateformat = '%d/%m/%Y'</code>
<code>decimal_separator</code>	小数分隔符	<code>decimal_separator = ','</code>
<code>thousands_separator</code>	千位分隔符	<code>thousands_separator = ','</code>
<code>ignore_errors</code>	坏行跳过（谨慎）	<code>ignore_errors = true</code>
<code>encoding</code>	字符编码	<code>encoding = 'ISO-8859-1'</code> (ICU 扩展)
<code>max_line_size</code>	单行最大字节	超大行时需要调大
<code>parallel</code>	是否并行读	<code>parallel = true</code> (默认)
<code>union_by_name</code>	多文件时按列名合并	见 12.6
<code>filename</code>	增加来源文件名列	<code>filename = true</code>
<code>hive_partitioning</code>	是否解析 hive 目录	见 12.6
<code>normalize_names</code>	列名规范化	<code>normalize_names = true</code>
<code>types</code> / <code>dtypes</code>	只指定部分列类型	<code>types = {'a': 'BIGINT'}</code>

12.2.4 真实场景示例

工资单 CSV（含千位符、管道分隔、日期为 dd/mm/yyyy）：

```

1 SELECT *
2 FROM read_csv('payroll.csv',
3     delim           = '|',
4     thousands_separator = ',',
5     dateformat       = '%d/%m/%Y',
6     columns          = {'emp_id': 'BIGINT', 'name': 'VARCHAR',
7
8     'salary': 'DECIMAL(12,2)', 'hire_date': 'DATE'});

```

脏日期列先当字符串，再安全转换：

```

1 SELECT
2     order_id,
3     try_cast(strptime(ts_str, '%Y-%m-%d %H:%M:%S') AS TIMESTAMPTZ) AS
  ts
4 FROM read_csv('events.csv', all_varchar = true);

```

跳过前 3 行的说明性头部:

```

1 SELECT * FROM read_csv('report.csv', skip = 3, header = true);

```

12.2.5 诊断 CSV 的利器: sniff_csv

```

1 -- 先"看一眼"推断结果再决定要不要覆盖
2 SELECT * FROM ( sniff_csv('data.csv') );
3 DESCRIBE SELECT * FROM sniff_csv('data.csv');

```

它会给出: 分隔符、列名、推断类型、是否含表头、是否含 BOM、行分隔符等元信息。

建议流程: 拿到一个陌生 CSV → `SELECT * FROM sniff_csv('x.csv')` → 用它的输出修正 `read_csv` 参数。

12.2.6 CSV 性能提示

- 并行默认是开启的, 但**单文件并且结构简单时并行收益有限**;
- `all_varchar = true` 比推断类型**快** (因为省去了转换), 适合"读进来马上再加工"的场景;
- 若一个文件被查询多次, 考虑先 `COPY ... TO Parquet` (见第 13 章) 再反复查询, 速度快很多;
- 超大 CSV (> 10 GB): 要么转 Parquet, 要么用 `read_csv` + 谓词下推 (DuckDB 会尽量跳过不匹配的行, 但仍需扫描字节)。

12.3 Parquet: 分析的首选格式

12.3.1 基本读取

```

1 SELECT * FROM 'sales.parquet';
2 SELECT * FROM read_parquet('sales.parquet');
3 SELECT count(*) FROM read_parquet('data/**/*.*.parquet'); -- glob 递归

```

12.3.2 为什么 Parquet 快

Parquet 文件内部结构:

```

1 File (Parquet)
2 |— Row Group 0 (建议 100MB~1GB)
3 |   |— Column Chunk: user_id (含 Statistics: min/max/null_count)
4 |   |— Column Chunk: amount (压缩: Dictionary + RLE)
5 |   |— Column Chunk: event_time
6 |— Row Group 1
7 |— ...
8 |— Footer (Schema + RowGroup 元数据 + Key/Value metadata)

```

三点含义:

1. **列裁剪**: `SELECT amount FROM ...` 只 IO amount 列;
2. **谓词下推 + Row Group 跳过**: `WHERE dt = '2026-01-01'` 先看每个 Row Group 的 min/max, 不符合的整组跳过;
3. **压缩 + 编码**: 字典编码 (重复值)、RLE、Delta、Bit-packing。

12.3.3 探查 Parquet 元数据

```

1  -- Schema
2  SELECT * FROM parquet_schema('sales.parquet');
3
4  -- 文件级元数据
5  SELECT * FROM parquet_file_metadata('sales.parquet');
6
7  -- key/value metadata (很多工具在这里藏信息, 比如 pandas 版本)
8  SELECT * FROM parquet_key_value_metadata('sales.parquet');
9
10 -- 所有 row group 的统计信息 (最关键的诊断工具)
11 SELECT
12     row_group_id,
13     column_id,
14     stats_min, stats_max, stats_null_count,
15     num_values, total_compressed_size
16 FROM parquet_metadata('sales.parquet')
17 WHERE column_id = 1
18 ORDER BY row_group_id;

```

`parquet_metadata` 是了解“为什么这个查询慢”的第一工具: 如果 row group 里的 min/max 范围覆盖了整个值域 (数据没有排序), 那就无法跳过任何 row group。

12.3.4 常用参数

```

1 SELECT * FROM read_parquet('logs/**/*.parquet',
2     union_by_name      = true,      -- 不同文件列名不同时按名字合并
3     hive_partitioning  = true,      -- 解析 dt=2026-01-01/ 目录名作为列
4     filename           = true,      -- 增加一列记录来源文件名
5     binary_as_string   = true,      -- BYTE_ARRAY 当作 VARCHAR
6     file_row_number     = true      -- 增加行号列 (file_row_number)
7 );

```

12.3.5 单个表被多个 Parquet 文件表达

```
1  -- 目录 + glob
2  SELECT count(*) FROM read_parquet('warehouse/sales/*.parquet');
3
4  -- 显式列表
5  SELECT * FROM read_parquet(['a.parquet', 'b.parquet']);
6
7  -- 递归所有子目录
8  SELECT * FROM read_parquet('warehouse/**/*.*.parquet');
```

DuckDB 会并行读取它们并做元数据合并。

12.4 JSON / NDJSON

12.4.1 NDJSON (每行一个 JSON)

```
1  SELECT * FROM read_ndjson('events.ndjson');
2  -- 或 read_json_auto
3  SELECT * FROM read_json('events.ndjson', format =
   'newline_delimited');
```

12.4.2 数组形式的 JSON

```
1  [ {"a":1}, {"a":2} ]
```

```
1  SELECT * FROM read_json('arr.json', format = 'array');
```

12.4.3 常用参数

```
1  SELECT * FROM read_json_auto('events.ndjson',
2    columns          =
3    {'id': 'BIGINT', 'ts': 'TIMESTAMPZ', 'payload': 'JSON'},
4    maximum_object_size = 33554432,      -- 单个 JSON 对象最大字节（默认约
   16MB）
5    ignore_errors      = true,           -- 忽略解析失败的行
6    union_by_name       = true,
7    dateformat          = '%Y-%m-%dT%H:%M:%SZ'
   );
```

常见坑：嵌套很深的 JSON 会超过 `maximum_object_size`，调大即可。另一个坑是某些日志每行 JSON 里字段名不一致——用 `union_by_name = true` 自动补齐。

12.4.4 JSON 到列的转换流程

```
1  -- 1) 原样读成 JSON 列
2  CREATE TABLE raw AS SELECT json(line) AS j FROM
   read_ndjson('app.ndjson');
3
4  -- 2) 提取字段 (silver)
5  CREATE OR REPLACE TABLE clean AS
6  SELECT
7      try_cast(j->>'user_id' AS BIGINT)           AS user_id,
8      j->>'action'                                AS action,
9      try_cast(j->>'ts' AS TIMESTAMPTZ)           AS ts,
10     j->'properties'                             AS props
11  FROM raw;
```

12.5 Excel / 其他格式

12.5.1 Excel (xlsx)

较新版本通过 `excel` 扩展支持:

```
1  INSTALL excel;    -- 若你的版本把它作为内置扩展则不需要
2  LOAD excel;
3
4  SELECT * FROM read_xlsx('report.xlsx');
5  SELECT * FROM read_xlsx('report.xlsx', sheet = 'Sheet2');
6  SELECT * FROM read_xlsx('report.xlsx', range = 'A1:D100');
7  SELECT * FROM read_xlsx('report.xlsx', header = false);
```

版本差异: Excel 的支持方式随版本变化较大 (早期是 `st_read` + spatial 扩展, 后来是社区扩展, 近期进入核心扩展)。如果报错, 请查阅对应版本的扩展文档, 或用 Python + pandas/calamine 读完后 `con.register()`。

12.5.2 其他格式一览

格式	方式
Avro	<code>avro</code> 扩展 → <code>read_avro(...)</code>
Delta Lake	<code>delta</code> 扩展 → <code>delta_scan('s3://path')</code>
Apache Iceberg	<code>iceberg</code> 扩展 → <code>iceberg_scan(...)</code>
SQLite 文件	<code>sqlite</code> 扩展 → <code>sqlite_attach('app.db')</code>
MySQL / PostgreSQL	<code>mysql</code> / <code>postgres</code> 扩展 → <code>ATTACH</code> (第 14 章)
GeoJSON / Shapefile	<code>spatial</code> 扩展 → <code>ST_Read(...)</code>
Arrow IPC / Feather	<code>read_feather</code> ? — 通常用上层库转换即可

格式	方式
Remote HTTP	https 扩展 → 直接写 URL

12.6 多文件与分区目录

12.6.1 多文件合并: union_by_name

```

1  -- 三个文件列名部分重叠
2  SELECT * FROM read_parquet(['old_sales.parquet', 'new_sales.parquet'],
3                               union_by_name = true);

```

不开启 `union_by_name` 时, DuckDB 假定所有文件列顺序一致, 不同则出错。

12.6.2 Hive 分区

数据湖标准布局:

```

1  warehouse/
2    dt=2026-01-01/region=华东/part-0.parquet
3    dt=2026-01-01/region=华北/part-0.parquet
4    dt=2026-01-02/region=华东/part-0.parquet

```

```

1  SELECT * FROM read_parquet('warehouse/**/*.*.parquet', hive_partitioning
2                               = true)
3  WHERE dt = DATE '2026-01-01';

```

分区列会从目录名解析出来, 并且分区裁剪会直接跳过整个目录 → 极大提速。

可选参数:

```

1  SELECT * FROM read_parquet('warehouse/**/*.*.parquet',
2                               hive_partitioning = true,
3                               hive_types_autocast = 1,      -- 把 dt 自动转成 DATE (0=都当字符串, 1=自动, 2=按 hive 推断)
4                               hive_partition_names = ['dt', 'region'] -- 显式指定 (目录名缺少 key 时必须)
5  );

```

12.6.3 来源文件名

```

1  SELECT filename, count(*) AS n
2  FROM read_parquet('logs/**/*.*.parquet', filename = true)
3  GROUP BY filename;

```

这是排查“哪个文件有问题”的好办法。

12.7 通过 HTTP(S) / 对象存储读取

需要 `httpfs` 扩展（较新版本内置且会自动安装加载）：

```
1 INSTALL httpfs;
2 LOAD httpfs;
3
4 SELECT * FROM 'https://raw.githubusercontent.com/.../data.csv' LIMIT
5 5;
6 SELECT count(*) FROM read_parquet('https://bucket/path/file.parquet');
```

如果是 S3 私有对象，需要配置认证（第 14 章详述）：

```
1 CREATE SECRET my_s3 (TYPE S3, KEY_ID 'AKIA...', SECRET '...', REGION
2 'ap-northeast-1');
3 SELECT * FROM read_parquet('s3://my-bucket/data.parquet');
```

HTTP 的范围请求（Range Request）使 DuckDB **不需要下载整个文件**——只读需要的字节范围（依赖服务端的 Range 支持）。

 注意：对 HTTP 源做多次重复的复杂查询时，最好先 `COPY` 下载到本地，避免重复的网络往返。

12.8 选型指南：什么时候该转 Parquet

场景	建议
源数据是 CSV，且要反复查询	转成 Parquet （通常 5~10 倍压缩 + 10 倍查询提速）
源数据是 CSV，只查一次	直接读，不必转
数据要给别人用通用工具打开	CSV（可读性好）或 Parquet + 文档
需要 ACID 事务、增量写	DuckDB 数据库文件或 Delta/Iceberg
数据量非常大但只用其中几列	必须 Parquet （列裁剪是刚需）

转换命令（详见第 13 章）：

```
1 COPY (SELECT * FROM read_csv_auto('big.csv'))
2 TO 'big.parquet' (FORMAT PARQUET, COMPRESSION ZSTD, ROW_GROUP_SIZE
3 1000000);
```

12.9 常见错误处理

错误信息	原因	处理
------	----	----

错误信息	原因	处理
Conversion Error: Could not convert string 'x' to INT32	类型推断基 于采样, 脏 数据在后	sample_size=-1、all_varchar=true 后 try_cast、或显式 types
CSV Error: Line ... exceeds maximum line size	单行太大	max_line_size = <更大值>
Invalid Input Error: ... Unrecognized date syntax	日期格式不 匹配	dateformat='%d/%m/%Y' 或先按字符串 读
Invalid Input Error: File ... too many columns in line	列数与表头 不一致	ignore_errors=true 或 columns 显 式指定
Catalog Error: Table with name x does not exist	忘了引号 (把文件名 当表名要加 引号)	用 'file.csv'
IO Error: Cannot open file	路径错误 / 没权限 / 需 要 httpfs	检查路径; 远程则 INSTALL httpfs
Invalid Input Error: Duplicate column name	CSV 有重复 列名	normalize_names=true
Not implemented Error: Unsupported compression	压缩格式不 支持	先解压或用对应 compression 参数

12.10 实战：把一堆脏 CSV 变成干净 Parquet

```

1  -- 1) 先看结构
2  SELECT * FROM sniff_csv('orders_2026_01.csv');
3
4  -- 2) 建一个"稳定的读取视图" (把脏活封在这里)
5  CREATE OR REPLACE VIEW staging.orders AS
6  SELECT
7      try_cast(order_id AS BIGINT) AS
8      order_id,
9      trim(user_id) AS
10     user_id,
11     coalesce(region, '未知') AS
12     region,

```

```

10     try_cast(replace(amount, ',', '') AS DECIMAL(12,2))      AS
    amount,
11     try_cast(strptime(order_ts, '%Y/%m/%d %H:%M') AS TIMESTAMP) AS
    order_ts,
12     filename                                                  AS
    src_file
13 FROM read_csv('landing/orders_*.csv',
14     union_by_name = true,
15     filename      = true,
16     all_varchar   = true,          -- 先全字符串，避免推断失败
17     nullstr       = ['NULL', '', 'N/A'],
18     ignore_errors = false
19 );
20
21 -- 3) 校验
22 SELECT count(*) AS total,
23        count(*) - count(order_id) AS miss_id,
24        count(*) - count(amount)   AS miss_amount,
25        min(order_ts), max(order_ts)
26 FROM staging.orders;
27
28 -- 4) 落地为 Parquet (后续所有分析都基于它)
29 COPY (SELECT * FROM staging.orders WHERE order_id IS NOT NULL)
30 TO 'warehouse/orders.parquet'
31 (FORMAT PARQUET, COMPRESSION ZSTD, ROW_GROUP_SIZE 500000);

```

实例演练

实例 12-1 · 用 parquet_metadata 看出"能不能跳过数据"

```

1 SELECT row_group_id, row_group_num_rows, path_in_schema,
2        stats_min, stats_max, compression
3 FROM parquet_metadata('sales.parquet')
4 WHERE path_in_schema = 'order_id'
5 ORDER BY row_group_id;

```

row_group_id	row_group_num_rows	path_in_schema	stats_min	stats_max	compression
0	122880	order_id	1	122880	ZSTD
1	77120	order_id	122881	200000	ZSTD

每个 row group 都带着 min/max (即 Zone Map) 。

如果执行 `WHERE order_id = 5`, 第 2 个 row group (122881~200000) 会被整块跳过——这就是"排序写入能加速"的物理原因。

文件级概览:

```

1 SELECT count(DISTINCT row_group_id) AS num_row_groups,
2       count(DISTINCT column_id) AS num_columns
3 FROM parquet_metadata('sales.parquet');
4 -- 2 个 row group、9 列

```

实例 12-2 · sniff_csv：先诊断再读取

```
1 SELECT * FROM sniff_csv('dirty_sales.csv');
```

实测（节选关键列）：

Delimiter	Quote	Escape	HasHeader	NumColumns
\t	"	"	true	5

推断出的列：order_id BIGINT、user_id VARCHAR、region VARCHAR、amount DOUBLE、order_date_txt VARCHAR。

注意到 amount 被推断成了 DOUBLE 而不是 DECIMAL —— 金额场景请手动指定 DECIMAL(12,2)。

实例 12-3 · 脏 CSV 的安全读法

```

1 SELECT * FROM read_csv('dirty_sales.csv',
2   delim = '|', header = true, all_varchar = true,
3   nullstr = ['', 'NULL', 'N/A']
4 ) LIMIT 3;

```

order_id	user_id	region	amount	order_date_txt
1	U0583	华中	851.47	2026-01-01
7	U0678	华南	(空)	2026-01-07
11	U0123	华东	123.45	03/15/2026

第 7 行金额为空（种子脚本故意造的），第 11 行日期是美式格式。完整清洗方案见第 22 章案例 10。

实例 12-4 · 一次读完整个目录

```

1 -- 同一 schema 的多个文件（最常见）
2 SELECT count(*) FROM read_parquet('warehouse/sales/**/*.parquet',
3   hive_partitioning = true);
4 -- 200000

```

配合 filename = true 排查“哪个文件有问题”：

```
1 SELECT filename, count(*) AS n
2 FROM read_parquet('warehouse/sales/**/*.*.parquet', filename = true)
3 GROUP BY filename ORDER BY n DESC;
```

⚠ **schema 不同的文件**：直接 glob 会按第一个文件的 schema 解析（容易丢列或报错）。

实测：把 `sales.parquet` / `users.parquet` / `events.parquet` 混在一起读，

不加参数返回 20 万（只按第一个文件的 schema 对齐），

加上 `union_by_name = true` 后返回 **860980**（把所有文件的行都算进来，缺失列补 NULL）。

列不一致时一定要显式写 `union_by_name = true`。

12.11 本章小结

- 三种写法：文件名当表名（快） / `read_*` 函数（可控） / 带参数函数（精确）；
- CSV 的关键是**类型推断**：用 `sniff_csv` 诊断、`all_varchar + try_cast` 是最稳的脏数据策略；
- Parquet 的性能来自列裁剪 + Row Group 跳过（Zone Map）+ 压缩，用 `parquet_metadata` 诊断；
- 多文件用 `union_by_name`、Hive 目录用 `hive_partitioning` 获得分区裁剪；
- 远程文件靠 `httpfs + Range` 请求；
- **重复查询的数据请转成 Parquet**，这是最大的性能杠杆。

动手练习

1. 对第 2 章导出的 `sales.csv` 跑 `sniff_csv`，看看它推断出的类型是否正确。
2. 用 `all_varchar=true` 再读一次，观察 `typeof()` 的结果差异。
3. 生成一个多层 Hive 分区目录（按 region），再用 `hive_partitioning=true` 查询并验证 `.count` 是否被裁剪（用 `EXPLAIN ANALYZE` 看扫描行数）。
4. 把 `sales.parquet` 的 row group 数量、每列的 min/max 打印出来。
5. 尝试读取一个远程 CSV（公共 URL），确认 `httpfs` 自动加载的行为。

下一章

[第 13 章 写出数据与 ETL](#) —— COPY TO、分区导出、EXPORT DATABASE，以及一条完整的 ETL 流水线。

第 13 章 写出数据与 ETL

读完本章，你能用 DuckDB 独立完成"从原始文件到可交付数据集"的完整流水线。

13.1 写出的四种方式

方式	适用	例子
<code>COPY ... TO</code>	导出到文件（推荐）	<code>COPY (SELECT...) TO 'a.parquet'</code>
<code>CREATE TABLE AS SELECT</code>	落到数据库表	<code>CREATE TABLE t AS SELECT ...</code>
<code>INSERT INTO ... SELECT</code>	追加数据	<code>INSERT INTO t SELECT ...</code>
<code>EXPORT DATABASE</code>	整个库的迁移/备份	<code>EXPORT DATABASE 'dump'</code>

13.2 COPY TO：导出到文件

13.2.1 基本语法

```
1  -- 导出整表
2  COPY sales TO 'sales.parquet' (FORMAT PARQUET);
3
4  -- 导出查询结果（必须加括号）
5  COPY (SELECT region, sum(amount) AS gmv FROM sales GROUP BY 1)
6  TO 'region_gmv.parquet' (FORMAT PARQUET);
7
8  -- 根据扩展名自动推断格式（推荐）
9  COPY (SELECT * FROM sales) TO 'sales_export.parquet';
10
11 COPY (SELECT * FROM sales) TO 'sales_export.csv' (HEADER, DELIMITER
    ',');
```

13.2.2 各格式的常用参数

Parquet:

```

1 COPY (SELECT * FROM sales ORDER BY order_date)
2 TO 'sales.parquet'
3 (FORMAT PARQUET,
4  COMPRESSION ZSTD,          -- SNAPPY(默认) / ZSTD / GZIP / BROTLI /
                                LZ4 / NONE
5  ROW_GROUP_SIZE 500_000,    -- 每个 row group 的行数
6  ROW_GROUPS_PER_FILE 1,     -- 每个文件包含几个 row group
7  COMPRESSION_LEVEL 3,
8  OVERWRITE_OR_IGNORE
9 );

```

压缩算法怎么选：ZSTD 压缩率高、解压快，适合长期存储；SNAPPY 速度最快但压缩率略低；GZIP 兼容性最好，适合交给外部伙伴。

CSV:

```

1 COPY results TO 'out.csv'
2 (FORMAT CSV, HEADER true, DELIMITER '|', QUOTE '', ESCAPE '',
3  NULLSTR 'NA', DATEFORMAT '%Y-%m-%d', COMPRESSION GZIP);

```

JSON / NDJSON:

```

1 COPY (SELECT * FROM t) TO 'out.json' (FORMAT JSON, ARRAY true); --
  数组形式
2 COPY (SELECT * FROM t) TO 'out.ndjson' (FORMAT JSON, ARRAY false); --
  每行一个 JSON

```

SQL:

```

1 COPY (SELECT * FROM sales) TO 'insert.sql'
2 (FORMAT SQL, table_name = 'sales_copy'); -- 生成 INSERT 语句，便于迁移

```

13.2.3 分区导出 (重要)

```

1 COPY (SELECT *, date_part('year', order_date) AS year FROM sales)
2 TO 'warehouse/sales'          -- 注意：目标是一个**目录**
3 (FORMAT PARQUET, PARTITION_BY (region, year), OVERWRITE_OR_IGNORE,
  COMPRESSION ZSTD);

```

生成结构:

```

1 warehouse/sales/
2   region=华东/year=2026/part-0.parquet
3   region=华北/year=2026/part-0.parquet
4   ...

```

要点:

- `PARTITION_BY` 的列必须是**低基数**（否则会生成无数小目录）；

- 目标路径必须是一个目录；
- 分区目录名是 Hive 风格，读取时用 `hive_partitioning = true`；
- 通常按**日期**（年/月/日）或**业务线**分区。

13.2.4 覆盖与追加行为

参数	行为
(默认)	目标已存在则 报错
<code>OVERWRITE</code>	覆盖同名文件或分区目录
<code>OVERWRITE_OR_IGNORE</code>	目录存在时先删除再重写（分区写出的常用选项）
<code>APPEND</code>	追加（追加到 parquet 需结构兼容）
<code>PER_THREAD_OUTPUT</code>	每个线程输出一个文件（并行写出更快）

13.2.5 控制输出文件大小

```
1 COPY (SELECT * FROM huge_table)
2 TO 'chunks/data.parquet'
3 (FORMAT PARQUET, PER_THREAD_OUTPUT true, ROW_GROUPS_PER_FILE 8,
  FILE_SIZE_BYTES '256MB');
```

输出**太多小文件**会严重影响后续读取（每个文件的 footer 都要单独解析）。建议每个 Parquet 文件在 **100 MB ~ 1 GB** 之间。

13.3 写入数据库表

13.3.1 CTAS

```
1 CREATE OR REPLACE TABLE sales_copy AS SELECT * FROM sales;
```

13.3.2 INSERT

```
1 INSERT INTO sales SELECT * FROM staging_sales WHERE order_date >= DATE
  '2026-01-01';
2
3 INSERT INTO users BY NAME SELECT * FROM new_users_stg;      -- 按列名对
  齐
4
5 INSERT OR REPLACE INTO dim_product BY NAME SELECT * FROM
  product_updates;      -- upsert
```

13.3.3 高效批量写入：Appender

逐条 INSERT 在 DuckDB 中很慢（每行一个事务开销）。三种提速方式：

方式一：一条多值 INSERT

```
1 | INSERT INTO t VALUES (1, 'a'), (2, 'b'), (3, 'c');
```

方式二：Python Appender（最快）

```
1 | import duckdb
2 |
3 | con = duckdb.connect("warehouse.db")
4 | con.execute("CREATE TABLE IF NOT EXISTS events(id BIGINT, payload
   | VARCHAR)")
5 |
6 | appender = con.appender("events")
7 | for row in rows:
8 |     appender.append_row(*row)
9 | appender.flush()
10 | appender.close()
```

方式三：参数化 executemany

```
1 | con.executemany("INSERT INTO t VALUES (?, ?)", list_of_tuples)
```

性能量级参考：循环逐条 INSERT 大约几千行/秒；Appender 可达数十万到百万行/秒。
务必避免 Python for 循环里调 `con.execute("INSERT ...")`。

方式四：从文件批量装载

```
1 | INSERT INTO events SELECT * FROM
   | read_parquet('batch_2026_09_*.parquet');
```

这通常是最快的：让数据库自己读，而不是 Python 逐行喂给它。

13.4 EXPORT / IMPORT DATABASE：整库搬迁

```
1 | EXPORT DATABASE 'dump_2026';
```

生成目录：

```
1 | dump_2026/
2 |   schema.sql      -- CREATE TABLE / VIEW / TYPE / MACRO 等 DDL
3 |   load.sql        -- COPY 语句，引用 data/ 下的 Parquet
4 |   data/
5 |     table1.parquet
6 |     table2.parquet
```

恢复：

```
1 | IMPORT DATABASE 'dump_2026';
```

也可以在另一个环境里先执行 `schema.sql` 再执行 `load.sql`：

```
1 | duckdb new.db < dump_2026/schema.sql
2 | duckdb new.db < dump_2026/load.sql
```

用途：

- **跨版本迁移**（虽然 1.0 之后文件格式向后兼容，导出/导入仍是最保险的方式）；
- 备份与分发；
- 把库搬到另一台机器。

自定义导出参数：

```
1 | EXPORT DATABASE 'dump' (FORMAT PARQUET, COMPRESSION ZSTD,
   | ROW_GROUP_SIZE 500000);
```

⚠ 导出期间请勿并发写入；这是一个**逻辑备份**，不是一致性快照（第 18 章会讲备份策略）。

13.5 增量 / 幂等的 ETL 模式

DuckDB 的单机批处理通常不需要复杂调度框架，以下是几种常见模式。

13.5.1 模式一：全量重建（最简单，优先推荐）

```
1 | CREATE OR REPLACE TABLE mart.daily_metrics AS
2 | SELECT dt, count(*) AS events FROM staging.events GROUP BY 1;
```

优点：逻辑简单、无状态、永远正确。

缺点：数据量大时每次都全算。

适用：**绝大多数中小规模场景**（几 GB 到几十 GB 完全可行，几十秒到几分钟）。

13.5.2 模式二：按分区重跑（常见）

思路：要更新某一天，就重新计算并覆盖那一天的分区。

```
1 | -- 把目标分区写到独立的 Parquet 目录中，覆盖旧目录内容
2 | COPY (SELECT * FROM staging.events WHERE dt = DATE '2026-09-17')
3 | TO 'warehouse/events' (FORMAT PARQUET, PARTITION_BY (dt),
   | OVERWRITE_OR_IGNORE);
```

如果是数据库表，可以用先删后插的方式：

```

1 BEGIN TRANSACTION;
2 DELETE FROM mart.daily_metrics WHERE dt = DATE '2026-09-17';
3 INSERT INTO mart.daily_metrics
4 SELECT dt, count(*) FROM staging.events WHERE dt = DATE '2026-09-17'
   GROUP BY 1;
5 COMMIT;

```

用事务包起来，保证"要么全成功，要么全回滚"，不会出现中间状态。

13.5.3 模式三：Upsert 增量同步

```

1 CREATE TABLE IF NOT EXISTS dim_user (
2     user_id    BIGINT PRIMARY KEY,
3     name       VARCHAR,
4     plan       VARCHAR,
5     updated_at TIMESTAMPTZ
6 );
7
8 INSERT OR REPLACE INTO dim_user BY NAME
9 SELECT user_id, name, plan, max(updated_at) AS updated_at
10 FROM staging.users
11 GROUP BY user_id, name, plan;

```

13.5.4 模式四：Watermark（处理到哪一天）

```

1 CREATE TABLE IF NOT EXISTS etl_state (k VARCHAR PRIMARY KEY, v
   VARCHAR);
2
3 INSERT OR REPLACE INTO etl_state VALUES ('last_processed_date', '2026-
   09-17');
4
5 -- 下次运行时读取起点
6 SELECT v FROM etl_state WHERE k = 'last_processed_date';

```

13.5.5 用 Python 编排

```

1 #!/usr/bin/env python3
2 """每日运营 ETL。用法: python etl.py"""
3 import duckdb
4 from datetime import date, timedelta
5
6 def main():
7     target = date.today() - timedelta(days=1)
8     con = duckdb.connect("warehouse.db")
9
10    # 1) 装载当天原始数据
11    con.execute("""
12        INSERT INTO staging_events
13        SELECT * FROM read_parquet(?, union_by_name = true)
14        """, [f"landing/{target}.parquet"])

```

```

15
16     # 2) 重算指标
17     con.execute("""
18         CREATE OR REPLACE TABLE mart_events AS
19         SELECT dt, count(*) AS events, count(DISTINCT user_id) AS uv
20         FROM staging_events
21         GROUP BY dt
22     """)
23
24     # 3) 导出
25     con.execute("""
26         COPY (SELECT * FROM mart_events)
27         TO 'out/metrics.parquet'
28         (FORMAT PARQUET, COMPRESSION ZSTD, OVERWRITE_OR_IGNORE)
29     """)
30
31     con.execute("INSERT OR REPLACE INTO etl_state VALUES
32     ('last_processed_date', ?)",
33                 [str(target)])
34     con.close()
35
36 if __name__ == "__main__":
37     main()

```

13.6 完整 ETL 实战：从原始日志到运营报表

输入：按天分区的 NDJSON 日志，`landing/events/dt=2026-*/part-*.ndjson`。

输出：

1. 清洗后的 Parquet（按天分区）；
2. 用户维度表；
3. 每日指标宽表；
4. 提供给 BI 的单一导出文件。

步骤 0：准备

```

1 CREATE SCHEMA IF NOT EXISTS staging;
2 CREATE SCHEMA IF NOT EXISTS mart;

```

步骤 1：定义稳定的读取契约（视图）

把“脏活”（类型转换、兼容不同的列名）封在一个视图里。

```

1 CREATE OR REPLACE VIEW staging.raw_events AS
2 SELECT
3     try_cast(user_id AS BIGINT)           AS user_id,
4     lower(event_type)                     AS event_type,
5     try_cast(ts AS TIMESTAMPTZ)          AS ts,
6     try_cast(amount AS DECIMAL(12,2))    AS amount,
7     page,
8     dt,
9     filename
10 FROM read_ndjson('landing/events/**/*.*.ndjson',
11                 hive_partitioning = true,
12                 filename = true,
13                 union_by_name = true);

```

如果你的 JSON 每行字段名嵌套（例如 `{"user":{"...}}`），可以先用 `read_ndjson` 把整行读成 JSON 列，再用 `->>` 提取，详见第 11 章。

步骤 2：清洗（silver 层）

```

1 CREATE OR REPLACE TABLE staging.events AS
2 SELECT
3     user_id,
4     event_type,
5     ts,
6     ts::DATE                AS dt,
7     coalesce(amount, 0)     AS amount,
8     page
9 FROM staging.raw_events
10 WHERE user_id IS NOT NULL
11     AND ts IS NOT NULL;
12
13 -- 数据质量检查（每步 ETL 都应该有）
14 SELECT
15     count(*)                AS total,
16     count(DISTINCT user_id) AS uv,
17     min(ts)                 AS from_ts,
18     max(ts)                 AS to_ts,
19     count(*) FILTER (WHERE event_type IS NULL) AS miss_type
20 FROM staging.events;

```

步骤 3：落成可用的数据湖（分区 Parquet）

```

1 COPY (
2     SELECT * FROM staging.events
3     ORDER BY dt                -- 排序写入，提升后续按 dt 过滤的效率
4 )
5 TO 'warehouse/events'
6 (FORMAT PARQUET, PARTITION_BY (dt), OVERWRITE_OR_IGNORE, COMPRESSION
  ZSTD);

```

`ORDER BY dt` 写入可以让每个分区内的数据有序，结合 Zone Map 显著加速。

步骤 4: 维度表 (gold 层)

```
1 CREATE OR REPLACE TABLE mart.dim_user AS
2 SELECT
3     user_id,
4     min(ts) AS first_seen,
5     max(ts) AS last_seen,
6     count(*) AS total_events,
7     sum(amount) AS total_amount,
8     list(DISTINCT event_type) AS event_types
9 FROM staging.events
10 GROUP BY user_id;
```

步骤 5: 指标宽表

```
1 CREATE OR REPLACE TABLE mart.daily_metrics AS
2 WITH daily AS (
3     SELECT
4         dt,
5         count(*)
6     AS events,
7         count(DISTINCT user_id)
8     AS dau,
9         sum(amount)
10    AS gm,
11         sum(amount) / nullif(count(DISTINCT user_id), 0)
12    AS arpu
13 FROM staging.events
14 GROUP BY dt
15 )
16 SELECT
17     dt, events, dau, gm, arpu,
18     lag(dau) OVER (ORDER BY dt)
19    AS prev_dau,
20     avg(dau) OVER (ORDER BY dt ROWS BETWEEN 6 PRECEDING AND CURRENT
21 ROW) AS dau_ma7
22 FROM daily
23 ORDER BY dt;
```

步骤 6: 导出给下游

```

1 COPY (SELECT * FROM mart.daily_metrics)
2 TO 'out/daily_metrics.parquet' (FORMAT PARQUET, COMPRESSION ZSTD,
  OVERWRITE_OR_IGNORE);
3
4 COPY mart.dim_user
5 TO 'out/dim_user.parquet' (FORMAT PARQUET, COMPRESSION ZSTD,
  OVERWRITE_OR_IGNORE);
6
7 -- 也给用户一份 CSV 便于 Excel 打开
8 COPY (SELECT * FROM mart.daily_metrics ORDER BY dt)
9 TO 'out/daily_metrics.csv' (HEADER, DELIMITER ',',
  OVERWRITE_OR_IGNORE);

```

步骤 7：加上校验断言

```

1 -- 断言 1: 指标行数应该等于不重复日期数
2 SELECT CASE WHEN count(*) = count(DISTINCT dt) THEN 'OK' ELSE 'FAIL'
  END AS check_dt
3 FROM mart.daily_metrics;
4
5 -- 断言 2: 没有负 GMV
6 SELECT CASE WHEN count(*) = 0 THEN 'OK' ELSE 'FAIL' END AS
  check_negative
7 FROM mart.daily_metrics WHERE gmv < 0;

```

真实项目里应把这些断言写成脚本，失败时中止流水线（第 20 章的生产实践会再展开）。

13.7 ETL 的性能要点

要点	做法
别用循环 INSERT	Appender / 批量 INSERT / 直接读文件
先过滤再 JOIN	让参与连接的数据尽量小
分区列要低基数	日期、业务线，不是 user_id
输出文件不要太小	≥ 100 MB
写入前 ORDER BY 排序	让 Zone Map 生效，后续读取快数倍
中间结果用临时表	CREATE TEMP TABLE，减少日志开销
大任务分批	按日期循环处理，避免单次事务过大
注意内存	SET memory_limit + SET temp_directory（第 17 章）

实例演练

实例 13-1 · 压缩算法怎么选（实测 20 万行）

```
1 .timer on
2 COPY (SELECT * FROM sales) TO 'c_snappy.parquet' (FORMAT PARQUET,
3   COMPRESSION SNAPPY);
4 COPY (SELECT * FROM sales) TO 'c_zstd.parquet' (FORMAT PARQUET,
5   COMPRESSION ZSTD);
6 COPY (SELECT * FROM sales) TO 'c_gzip.parquet' (FORMAT PARQUET,
7   COMPRESSION GZIP);
8 COPY (SELECT * FROM sales) TO 'c_plain.csv' (HEADER);
```

格式	体积	写入耗时
CSV	17 MB	0.079 s
Parquet + SNAPPY	3.5 MB	0.064 s
Parquet + ZSTD	2.1 MB	0.076 s
Parquet + GZIP	2.4 MB	0.339 s

结论：ZSTD 是默认最优（体积最小，写速度与 SNAPPY 接近）；GZIP 慢 4~5 倍，只在需要兼容性时用。

实例 13-2 · 分区导出 + 读回

```
1 COPY (SELECT *, strftime(order_date, '%Y-%m') AS ym FROM sales)
2 TO 'warehouse/sales' (FORMAT PARQUET, PARTITION_BY (ym),
3   OVERWRITE_OR_IGNORE);
4 -- ⚠ 目录必须已存在: shell 里先 mkdir -p warehouse/sales
5 SELECT count(*) FROM read_parquet('warehouse/sales/**/*.*parquet',
6   hive_partitioning = true); --
7 200000
```

目录	说明
warehouse/sales/ym=2026-01/...	1 月分区
warehouse/sales/ym=2026-06/...	6 月分区

之后 WHERE ym = '2026-01' 会触发**分区裁剪**，只读第一个目录。

实例 13-3 · 整库搬迁

```
1 EXPORT DATABASE 'dump_dir';      -- 生成 schema.sql / load.sql /
2 data/*.parquet
3 IMPORT DATABASE 'dump_dir';      -- 在另一个环境恢复
```

```
1 | ls dump_dir
2 | # schema.sql load.sql data/
```

实例 13-4 · 别用循环 INSERT（三种批量写入）

```
1 | -- ✅ 推荐：一条多值
2 | INSERT INTO t VALUES (1,'a'), (2,'b'), (3,'c');
3 |
4 | -- ✅ 推荐：从查询/文件灌
5 | INSERT INTO t SELECT * FROM read_parquet('batch.parquet');
6 |
7 | -- ❌ 反例：Python 里 for 循环单条 INSERT（几千行/秒）
```

Python 侧最快的写法是 Appender（第 15/19 章有代码）。

13.8 本章小结

- 导出主要靠 `COPY (查询) TO '文件' (参数)`；
- Parquet 建议用 `COMPRESSION ZSTD`、`ROW_GROUP_SIZE` 适当、`OVERWRITE_OR_IGNORE` 保证幂等；
- 分区导出用 `PARTITION_BY`（必须是低基数列），目标是一个目录；
- 写入表时务必用批量方式（Appender / `INSERT SELECT` / `read_*` 直读），**杜绝循环单条 `INSERT`**；
- `EXPORT DATABASE` / `IMPORT DATABASE` 是整库搬迁与备份的利器；
- ETL 模式四选一：**全量重建**（首选）、**分区重跑**、**upsert**、**watermark**；
- 每步加数据质量断言，让流水线可信。

动手练习

1. 把 `sales` 按 `region` 分区导出为 Parquet 目录，再用 `hive_partitioning=true` 读回来。
2. 对比三种写表方式的速度：循环 `INSERT` vs Appender vs `INSERT ... SELECT FROM read_parquet`。
3. 用 `EXPORT DATABASE` 导出第 2 章创建的 `learn.db`，然后在一个新目录里 `IMPORT DATABASE` 恢复。
4. 写一个幂等的 Python ETL 脚本：把 `sales` 按 `order_date` 分成 6 个 Parquet 文件导出，重复运行结果一致。
5. 给自己的 ETL 加三条断言（行数、非空关键列、非负金额）。

下一章

[第 14 章 挂载数据库与接入云存储](#) —— ATTACH 多个数据库、Secret 管理器、S3/R2/Azure/GCS、直接读写 PostgreSQL/MySQL/SQLite。

第 14 章 挂载数据库与接入云存储

DuckDB 可以像查询本地表一样查询：另一个 DuckDB 文件、PostgreSQL/MySQL/SQLite 数据库、以及 S3/OSS/R2/Azure/GCS 上的对象。

本章的核心是两个关键字：`ATTACH`（挂载）与 `CREATE SECRET`（凭据管理）。

14.1 ATTACH：把外部数据源挂进来

14.1.1 基本用法

```
1 ATTACH 'other.db' AS other;           -- 挂载另一个 DuckDB 库
2 ATTACH 'other.db' AS other_readonly (READ_ONLY); -- 只读挂载
3 ATTACH ':memory:' AS scratch;        -- 挂载一块额外的内存库
4
5 USE other;                           -- 切换默认库
6 SELECT * FROM other.my_table;        -- 用限定名访问
7 SHOW DATABASES;                     -- 看看挂了哪些
8 DETACH other;                        -- 卸载
```

挂载后，被挂载库里的表会自动出现在查询可见范围内：

```
1 SELECT a.x, b.y
2 FROM main.local_table a
3 JOIN other.remote_table b USING (id);
```

`main` 是什么？你最初打开的那个库它的默认名就叫 `main`。`.databases` 或 `SHOW DATABASES` 可以看到全部。

14.1.2 实用场景

1. **跨库搬运**：把旧库的表拷到新库；
2. **只读共享**：同事给你的分析库用 `READ_ONLY` 挂载，避免误改；
3. **内存 workspace**：把 `staging` 放 `:memory:`，中间结果不入磁盘；
4. **多源联邦查询**：PostgreSQL 的业务表 + 本地 Parquet + S3 上的历史数据，一条 SQL join。

示例：把另一个库的表直接搬过来

```
1 ATTACH 'old.db' AS old;
2 CREATE TABLE main.sales AS SELECT * FROM old.sales;
3 DETACH old;
```

读同一个库的多个文件时，也要留意**单写多读**限制（第 18 章）。

14.2 Secret 管理器：现代认证方式

以前 DuckDB 用 `SET s3_access_key_id=...` 这种全局设置。现在推荐使用 **Secret Manager**：

```
1 CREATE SECRET my_secret (  
2     TYPE    S3,  
3     KEY_ID  'AKIAIOSFODNN7EXAMPLE',  
4     SECRET  'wJalrXUtnFEMI/K7MDENG/bPxrFiCYEXAMPLEKEY',  
5     REGION  'ap-northeast-1'  
6 );
```

核心优势：

- 可以同时存在多个凭据，作用于不同 scope；
- 支持**持久化**（`CREATE PERSISTENT SECRET`）到数据库文件；
- 统一管理各类服务（S3/Azure/GCS/HTTP/MySQL/Postgres...）。

14.2.1 查看与管理

```
1 SELECT * FROM duckdb_secrets();           -- 列出当前 secret（不会显示明文）  
2 SELECT name, type, provider, scope FROM duckdb_secrets();  
3  
4 DROP SECRET my_secret;  
5 DROP SECRET IF EXISTS my_secret PERSISTENT;  
6 ALTER SECRET my_secret SET (REGION = 'us-east-1');
```

14.2.2 作用域 scope：让不同路径用不同凭据

```
1 CREATE SECRET prod_s3 (  
2     TYPE S3, KEY_ID '...', SECRET '...',  
3     SCOPE 's3://prod-bucket/'  
4 );  
5 CREATE SECRET dev_s3 (  
6     TYPE S3, KEY_ID '...', SECRET '...',  
7     SCOPE 's3://dev-bucket/'  
8 );
```

路径匹配采用**最长前缀优先**原则。

14.2.3 持久化 Secret

```
1 CREATE PERSISTENT SECRET my_s3 (TYPE S3, KEY_ID '...', SECRET '...');
```

持久化 secret 会**加密存储**在数据库文件里（基于明文密码或直接存储，取决于是否设置了 `secret_directory` / 是否加密数据库）。重启后仍然存在。

⚠ 注意：把含凭证的数据库文件分享给别人，等于分享凭证。团队场景建议用环境变量注入而不是持久化。

14.2.4 从环境变量/授权接口取得凭据

常见云服务商的做法：

- **AWS 默认凭据链**：不配置任何 secret 时，DuckDB 会尝试读取环境变量、配置文件、实例 IAM Role 等（某些 provider 支持 S3 类型的 CREDENTIAL_CHAIN provider）；
- **Azure**：PROVIDER CREDENTIAL_CHAIN；
- **GCS**：可以走 S3 兼容的互操作接口，或先用其他工具取 token。

这类能力随版本变化较快，请以当前版本的官方文档为准，通常写法类似：

```
1 CREATE SECRET aws_default (  
2     TYPE S3,  
3     PROVIDER CREDENTIAL_CHAIN  
4 );
```

14.3 对象存储实战

14.3.1 Amazon S3

```
1 INSTALL httpfs;  
2 LOAD httpfs;  
3  
4 CREATE SECRET s3_prod (  
5     TYPE S3,  
6     KEY_ID 'AKIA...',  
7     SECRET '...',  
8     REGION 'ap-northeast-1'  
9 );  
10  
11 -- 读  
12 SELECT count(*) FROM read_parquet('s3://my-  
    bucket/data/2026/**/*.parquet');  
13  
14 -- 写  
15 COPY (SELECT * FROM sales) TO 's3://my-bucket/out/sales.parquet'  
    (FORMAT PARQUET, COMPRESSION ZSTD);
```

常用参数：

参数	说明
REGION	必需，否则可能报 400
ENDPOINT	自定义端点（MinIO/R2/OSS/COS）
URL_STYLE	'vhost'（默认）或 'path'（MinIO 通常用 path）
USE_SSL	是否 https

参数	说明
SCOPE	作用域前缀
SESSION_TOKEN	临时凭证 (STS)

14.3.2 S3 兼容服务 (MinIO / Cloudflare R2 / 阿里云 OSS / 腾讯云 COS)

```

1  -- Cloudflare R2
2  CREATE SECRET r2 (
3      TYPE      S3,
4      KEY_ID    'xxx',
5      SECRET    'yyy',
6      ENDPOINT  'xxx.r2.cloudflarestorage.com',
7      URL_STYLE 'vhost'
8  );
9
10 -- MinIO (本地)
11 CREATE SECRET minio (
12     TYPE      S3,
13     KEY_ID    'minioadmin',
14     SECRET    'minioadmin',
15     ENDPOINT  'localhost:9000',
16     URL_STYLE 'path',
17     USE_SSL   false
18 );
19
20 -- 阿里云 OSS
21 CREATE SECRET oss (
22     TYPE      S3,
23     KEY_ID    'LTAI...',
24     SECRET    '...',
25     REGION    'oss-cn-hangzhou',
26     ENDPOINT  'oss-cn-hangzhou.aliyuncs.com',
27     URL_STYLE 'vhost'
28 );

```

14.3.3 Azure Blob Storage

```

1 CREATE SECRET az (
2     TYPE AZURE,
3     PROVIDER CREDENTIAL_CHAIN          -- 走 Azure SDK 的默认凭据链（环
    境变量/managed identity）
4 );
5
6 -- 或显式串。.account 形式:
7 CREATE SECRET az2 (
8     TYPE AZURE,
9     CONNECTION_STRING
10    'DefaultEndpointsProtocol=https;AccountName=...;AccountKey=...'
11 );
12 SELECT * FROM read_parquet('az://container/path/file.parquet');
13 SELECT * FROM
    read_parquet('abfss://container@account.dfs.core.windows.net/path/fi
    le.parquet');

```

14.3.4 Google Cloud Storage

```

1 CREATE SECRET gcs (
2     TYPE GCS,
3     KEY_ID      '...',    -- HMAC key id
4     SECRET      '...'    -- HMAC secret
5 );
6
7 SELECT * FROM read_parquet('gs://bucket/path/file.parquet');

```

GCS 通常借助 S3 兼容的互操作 API，把 endpoint 指到 `storage.googleapis.com`。

14.3.5 HTTP(S) 与 API 鉴权

```

1 -- Bearer token 或任意 HTTP 头
2 CREATE SECRET api (
3     TYPE HTTP,
4     EXTRA_HTTP_HEADERS MAP {
5         'Authorization': 'Bearer eyJhbGci...',
6         'X-Client-Id':   'myapp'
7     }
8 );
9
10 SELECT * FROM read_parquet('https://api.example.com/data.parquet');

```

基础认证也可以：

```

1 CREATE SECRET basicauth (TYPE HTTP, EXTRA_HTTP_HEADERS
    MAP{'Authorization': 'Basic dXNlcjpwYXNz'});

```

14.3.6 下载/上传缓存（重要）

对象存储的重复读取很贵（延迟 + 流量费）。DuckDB 提供本地缓存：

```
1 SET enable_object_cache = true;           -- 开启 HTTP 对象缓存
2 SET http_cache_directory = '/tmp/duckdb_cache'; -- 缓存位置（某些版本用
不同名字）
3 SET 或者: SET httpfs_client_enable_cache = true;
```

缓存相关参数在不同版本命名变化较大，请以 `SELECT * FROM duckdb_settings()`
`WHERE name LIKE '%cache%'` 查询为准。

14.4 联邦查询：读写其他数据库

14.4.1 PostgreSQL

```
1 INSTALL postgres;           -- 较新版本可能内置或仍需安装
2 LOAD postgres;
3
4 ATTACH 'host=127.0.0.1 port=5432 dbname=shop user=reader
password=xxx'
5 AS pg (TYPE POSTGRES);
6
7 SHOW ALL TABLES;           -- 看 PostgreSQL 里的表
8 SELECT * FROM pg.public.orders LIMIT 5;
9
10 -- 联邦查询: PG 的业务表 + 本地 Parquet + 本地 DuckDB 表
11 SELECT u.plan, count(*), sum(s.amount)
12 FROM pg.public.orders s
13 JOIN main.users u USING (user_id)
14 GROUP BY 1;
15
16 DETACH pg;
```

也可以只读重要配置：

```
1 ATTACH 'host=... dbname=shop' AS pg (TYPE POSTGRES, READ_ONLY, SECRET
'pg_secret');
```

14.4.2 MySQL

```
1 INSTALL mysql;
2 LOAD mysql;
3
4 ATTACH 'host=127.0.0.1 port=3306 database=shop user=root password=xxx'
5 AS my (TYPE MYSQL);
6
7 SELECT * FROM my.users LIMIT 5;
```

14.4.3 SQLite

```
1  INSTALL sqlite;
2  LOAD sqlite;
3
4  ATTACH 'app.db' AS sq (TYPE SQLITE);
5  SELECT * FROM sq.customers LIMIT 5;
6
7  -- 也可以反向: 把 DuckDB 表写进 SQLite
8  CREATE TABLE sq.copied AS SELECT * FROM main.sales;
```

14.4.4 其他

数据源	扩展	方式
Delta Lake	<code>delta</code>	<code>delta_scan('path')</code>
Apache Iceberg	<code>iceberg</code>	<code>iceberg_scan('catalog.table') / iceberg_attach(...)</code>
Google Sheets	<code>gsheet</code> (社区)	直接引用表格 URL
BigQuery	社区扩展	—

14.5 典型工作流：云端数据到本地分析

```
1  INSTALL httpfs; LOAD httpfs;
2
3  -- 1) 配置云端凭据
4  CREATE SECRET lake (
5      TYPE S3, KEY_ID '...', SECRET '...', REGION 'cn-north-1'
6  );
7
8  -- 2) 只拉取需要的分区与列，落成本地 Parquet
9  CREATE OR REPLACE TABLE recent AS
10 SELECT user_id, city_id, amount, dt
11 FROM read_parquet('s3://lake/events/dt=2026-09-*/**/*.parquet',
12                  hive_partitioning = true)
13 WHERE dt >= DATE '2026-09-10';
14
15 -- 3) 与本地维表关联，输出报表（之后所有迭代都在本地，不再走网络）
16 CREATE OR REPLACE TABLE report AS
17 SELECT d.city, count(*) AS cnt, sum(r.amount) AS gm
18 FROM recent r
19 JOIN main.dim_city d ON r.city_id = d.city_id
20 GROUP BY 1
21 ORDER BY gm DESC;
22
23 -- 4) 把结果回传云端
```

```

24 COPY (SELECT * FROM report)
25 TO 's3://lake/report/2026-09/report.parquet'
26 (FORMAT PARQUET, COMPRESSION ZSTD, OVERWRITE_OR_IGNORE);

```

关键经验：远程对象的每次访问都有网络延迟（几十毫秒起步）与流量成本。

推荐 workflow：**一次性把需要的分区/列拉成本地 Parquet → 之后所有迭代分析在本地完成 → 最后把结果写回云端。**

另一个重要技巧是只 **SELECT 需要的列**：列裁剪让网络传输量直接按列比例下降。

14.6 常见错误与排查

错误	原因	处理
IO Error: Cannot open file ... No such file or directory	未安装/加载 httpfs	INSTALL httpfs; LOAD httpfs;
Unable to connect to URL ... 403 Forbidden	凭证不足/过期	检查 Secret、Bucket Policy、临时 token
HTTP 400 Bad Request ... region	缺少 REGION	在 secret 里加 REGION
Unsupported URL scheme s3://	扩展没加载	LOAD httpfs
Connection to database failed: could not translate host	主机名/端口错误	ping / 用 IP / 检查防火墙
SSL connection failed	证书问题	检查系统 CA，或临时用 USE_SSL false（仅内网测试）
Database ... is already open / lock	另一个进程占用该文件	关闭另一端；生产见第 18 章
查询速度极慢	每次都从云端重新拉	开启对象缓存；或先落成本地 Parquet
Permission denied writing to s3	只给了 GetObject 权限	补 PutObject，或写到本地再上传

调试技巧：

```

1  -- 确认 httpfs 已加载
2  SELECT * FROM duckdb_extensions() WHERE extension_name = 'httpfs';
3
4  -- 确认 secret 生效
5  SELECT name, type, provider, scope FROM duckdb_secrets();
6
7  -- 打开更详细的日志辅助排查
8  SET enable_logging = true;
9  SET logging_level = 'DEBUG';
10 SET logging_storage = 'console';

```

调试完记得关掉 DEBUG 级日志，它会严重影响性能。

14.7 安全最佳实践

1. **不要**把访问密钥硬编码在共享的 SQL 脚本里。用：
 - 环境变量（Python 里 `os.environ`）；
 - CI/CD 的 secret；
 - IAM Role / Managed Identity（最推荐）。
2. 数据库文件（含持久化 secret）不要随意分发；
3. 生产环境给只读任务最小权限（`GetObject` 而不是 `***`）；
4. 临时 STS token 优于长期 AK/SK；
5. 对外部共享数据优先使用**预签名 URL**（presigned URL）而不是给出长期密钥。

Python 中注入环境变量的例子：

```

1  import os
2  import duckdb
3
4  con = duckdb.connect()
5  con.execute(f"""
6      CREATE OR REPLACE SECRET s3_lake (
7          TYPE S3,
8          KEY_ID '{os.environ["AWS_ACCESS_KEY_ID"]}',
9          SECRET '{os.environ["AWS_SECRET_ACCESS_KEY"]}',
10         REGION '{os.environ["AWS_REGION"]}'
11     )
12 """)

```

更好的做法是 DuckDB 直接读取环境凭据链（例如 `PROVIDER CREDENTIAL_CHAIN`），完全不必在 SQL 里拼字符串。

实例演练

实例 14-1 · ATTACH 跨库 JOIN (实测)

先造一个"维表库":

```
1 duckdb dims.db -c "CREATE TABLE regions AS
2   SELECT DISTINCT region, 'R-' || region AS region_code FROM
   read_parquet('sales.parquet');"

```

再在主库里挂载并查询:

```
1 ATTACH 'dims.db' AS dims (READ_ONLY);
2
3 SELECT database_name FROM duckdb_databases();

```

database_name
dims
examples
system
temp

```
1 SELECT s.region, d.region_code, round(sum(s.amount)) AS gmv
2 FROM main.sales s
3 JOIN dims.regions d ON s.region = d.region
4 GROUP BY 1, 2
5 ORDER BY gmv DESC
6 LIMIT 3;

```

region	region_code	gmv
华南	R-华南	17176472
华北	R-华北	17172025
华中	R-华中	17109098

维表库独立成一个文件 + 只读挂载, 是"一份维表给多个分析库复用"的实用模式。

实例 14-2 · 只读挂载的写入报错 (原文)

```
1 ATTACH 'dims.db' AS ro (READ_ONLY);
2 CREATE TABLE ro.x AS SELECT 1;

```

```
1 Invalid Input Error: Cannot execute statement of type "CREATE" on
   database "ro"
2 which is attached in read-only mode!

```

注意报错里的类型是 `CREATE`（不是某些资料写的 `CREATE_TABLE`）。

实例 14-3 · 元数据库函数的真实列名

```
1 SELECT * FROM duckdb_databases();
2 -- 实际列: database_oid, database_name, tags, cipher
```

⚠ 实测发现：`duckdb_databases()` 没有 `access_mode` 列（很多教程会写）。
想看读写模式用 CLI 的 `.databases` 或 `SHOW DATABASES`。

实例 14-4 · 云端凭据（示意，需真实账号才能跑）

```
1 INSTALL https; LOAD https;
2
3 CREATE SECRET lake (
4     TYPE S3, KEY_ID 'AKIA...', SECRET '...', REGION 'ap-northeast-1',
5     SCOPE 's3://my-bucket/'
6 );
7 SELECT count(*) FROM read_parquet('s3://my-bucket/data/*.parquet');
```

⚠ 本机没有可用的对象存储账号，本例**未实测**；Secret 的语法请以
`duckdb_secrets()` 与官方文档为准。
排错必备三条：

```
1 SELECT * FROM duckdb_extensions() WHERE extension_name =
  'https';
2 SELECT name, type, provider, scope FROM duckdb_secrets();
3 SELECT * FROM duckdb_settings() WHERE name LIKE '%http%' OR name
  LIKE '%s3%';
```

14.8 本章小结

- `ATTACH` 把外部数据库（DuckDB 文件、PostgreSQL、MySQL、SQLite）挂载成本地表一样的存在；
- `CREATE SECRET` 是统一的凭据管理机制，支持 scope、持久化、多类型（S3/Azure/GCS/HTTP）；
- 对象存储读写靠 https，**先适应范围检查 REGION / ENDPOINT / URL_STYLE**；
- 重复访问云端数据时开启对象缓存，或者干脆先落成本地 Parquet；
- 联邦查询让"PG 业务表 JOIN 本地 Parquet"成为一条 SQL 就能完成的事；
- 凭据安全第一：环境变量 / IAM Role / 最小权限 / 预签名 URL。

动手练习

1. 起一个本地 MinIO（或任意 S3 兼容服务），配置 secret 并读写一个对象。
2. 把一个本地 Parquet 通过 COPY 写到对象存储，再读回来验证。
3. 用两个 DuckDB 文件练习 `ATTACH` 跨库 JOIN；练习 `READ_ONLY` 挂载后尝试写入观察报错。
4. 若有本地 PostgreSQL/MySQL，安装对应扩展并 `ATTACH`，做一次联邦查询。
5. 查看 `duckdb_secrets()` 与 `duckdb_extensions()`，确认你的环境里有哪些可用能力。

下一章

[第 15 章 与 Python 及 Arrow 生态协同](#) —— DuckDB 为什么被称为"数据科学界的瑞士军刀"。

第 15 章 与 Python 及 Arrow 生态协同

DuckDB 之所以在数据科学圈爆发，一半功劳属于它与 **Apache Arrow** 的零拷贝互操作。本章讲清楚：

- pandas / Polars / PyArrow 与 DuckDB 之间的高效往返；
- 结果集的流式读取；
- Python UDF 让 SQL 拥有 Python 的能力（UDF 细节在第 19 章）；
- 在 Jupyter 里的典型工作流。

15.1 为什么 Arrow 是关键

传统方案："数据库 → 驱动协议 → Python 对象"：

```
1 PostgreSQL —(libpq 二进制协议)—> psycopg 解析 —> tuple —> pandas
   DataFrame
2                               每段都有序列化/反序列化开销
```

DuckDB + Arrow：数据从头到尾都是 **Arrow 列式内存格式**：

```
1 DuckDB(Arrow 内存) —— 零拷贝指针传递 ——> PyArrow —> pandas / Polars
```

带来的效果：

操作	传统	DuckDB→Arrow
1000 万行结果到 DataFrame	数秒 ~ 数十秒	几十毫秒（甚至纳秒级构造）
额外内存拷贝	通常 2~3 份	0~1 份
类型保真（Decimal/时区/嵌套）	经常丢失	完整保留

一句话：duckdb 与 pyarrow 之间没有"转换"，只有"传递指针"。

15.2 pandas ↔ DuckDB

15.2.1 直接查询 DataFrame（Replacement Scan）

```
1 import pandas as pd
2 import duckdb
3
4 df = pd.DataFrame({"a": [1, 2, 3], "b": ["x", "y", "z"]})
5
6 # 不需要 register, 不需要 COPY—写 SQL 时直接用变量名
7 duckdb.sql("SELECT sum(a) AS total FROM df").show()
8 duckdb.sql("SELECT * FROM df WHERE a > 1").df()
```

这是最省事的用法。但有**作用域限制**：它找的是调用栈上的局部变量。如果在函数内部调用另一个函数里的 SQL，可能找不到变量。工程代码请显式 `register` 或 `CREATE TABLE`。

15.2.2 结果 → DataFrame

```
1 rel = duckdb.sql("SELECT * FROM df")
2 pdf = rel.df()           # pandas DataFrame
3 print(pdf.dtypes)
```

`.df()` 内部走 Arrow，比 `pd.DataFrame(rows)` 快几十倍。

15.2.3 DataFrame → 持久表

```
1 con = duckdb.connect("analytics.db")
2
3 # 方式 1: 注册为视图（不物化）
4 con.register("my_df", df)
5 con.execute("SELECT count(*) FROM my_df").fetchall()
6
7 # 方式 2: 物化成表（推荐给反复使用的数据）
8 con.execute("CREATE OR REPLACE TABLE t AS SELECT * FROM df")
9
10 # 方式 3: 追加
11 con.execute("INSERT INTO t SELECT * FROM df")
```

建议：DataFrame 只是“数据原料”，一旦进入 DuckDB，就应该让 SQL 来处理。避免反复 `.df()` → pandas 操作 → 再回 DB 的乒乓过程。

15.2.4 pandas 的常见 JDBC-风格陷阱

```
1 # ❌ 慢：把 1000 万行拉回 Python 再过滤
2 df = con.execute("SELECT * FROM huge").df()
3 result = df[df.amount > 100].groupby("region").sum()
4
5 # ✅ 快：在 DuckDB 里做完聚合，只把 10 行结果拉回来
6 result = con.execute("""
7     SELECT region, sum(amount) AS gmv
8     FROM huge WHERE amount > 100
9     GROUP BY region
10 """).df()
```

黄金法则：让数据少动，让 SQL 多做。

15.3 Polars ↔ DuckDB

Polars 本身也是 Arrow 内核，因此互操作同样零拷贝。

```

1 import polars as pl
2 import duckdb
3
4 pl_df = pl.DataFrame({"x": [1, 2, 3]})
5
6 # 直接在 SQL 里引用 polars 变量
7 duckdb.sql("SELECT sum(x) FROM pl_df").show()
8
9 # 结果 → Polars
10 result = duckdb.sql("SELECT * FROM pl_df WHERE x > 1").pl()
11 print(result)

```

反之 (Polars 读 DuckDB 结果) :

```

1 lf = duckdb.sql("SELECT * FROM source").pl(lazy=True) # 得到
  LazyFrame
2 print(lf.collect(engine="streaming"))

```

Polars 的 `read_database` 也支持用 connection 直接查询, 但对于 DuckDB, 用 `con.sql(...).pl()` 最快。

15.4 PyArrow ↔ DuckDB

PyArrow 是"通用货币"。

```

1 import pyarrow as pa
2 import duckdb
3
4 tbl = pa.table({"a": [1, 2, 3]})
5
6 # Arrow → DuckDB
7 duckdb.sql("SELECT sum(a) FROM tbl").show()
8 con.execute("CREATE TABLE t AS SELECT * FROM tbl")
9
10 # DuckDB → Arrow
11 arrow_tbl = duckdb.sql("SELECT * FROM tbl").arrow()
12 print(arrow_tbl.num_rows, arrow_tbl.schema)
13
14 # Arrow → Parquet (也可以直接让 DuckDB 写, 见第 13 章)
15 import pyarrow.parquet as pq
16 pq.write_table(arrow_tbl, "out.parquet")

```

15.4.1 流式处理大数据: RecordBatchReader

如果要从 DuckDB 里拉出**远超内存**的结果, 不要用 `.df()`, 用流式:

```

1 con = duckdb.connect()
2 con.execute("CREATE TABLE big AS SELECT * FROM range(50_000_000)
  t(i)")
3
4 # 流式分批读取
5 reader = con.execute("SELECT * FROM big").fetch_arrow_table(100_000)
  # 或 fetch_record_batch
6 total = 0
7 for batch in reader:          # RecordBatchReader, 每批一个 Arrow
  RecordBatch
8     total += batch.num_rows
9 print(total)

```

更推荐的用法（把结果直接写成文件，避免 Python 内存参与）：

```

1 con.execute("""
2     COPY (SELECT * FROM big WHERE i % 2 = 0)
3     TO 'even.parquet' (FORMAT PARQUET, COMPRESSION ZSTD)
4 """)

```

结论：超大数据不要在 Python 里遍历。能用 SQL 写出去就用 SQL。

15.4.2 分批 INSERT 到 DuckDB

```

1 arrow_tbl = pa.concat_tables(batches)
2
3 con.register("batch_view", arrow_tbl)
4 con.execute("INSERT INTO target SELECT * FROM batch_view")
5 con.unregister("batch_view")

```

15.5 Jupyter / Notebook workflow

推荐的配置：

```

1 %load_ext autoreload
2 import duckdb
3 import pandas as pd
4
5 con = duckdb.connect("notebook.db", config={"threads": 8,
  "memory_limit": "8GB"})
6
7 # 让 pandas 显示更宽
8 pd.set_option("display.max_columns", 50)
9
10 # 便捷小工具：写 SQL，返回 DataFrame
11 def q(sql, params=None):
12     return con.execute(sql, params or []).df()
13
14 # 在 notebook 里直接渲染关系对象（DuckDB 实现了 _repr_html_）

```

```

15 con.sql("SELECT * FROM sales LIMIT 10")
16
17 # 便捷小工具：取前若干行看一眼
18 def peek(sql, params=None, n=20):
19     return con.execute(sql, params or []).df().head(n)

```

更好的方案是 **Jupyter 的 SQL magic** (依赖 `magic_duckdb` 或 `%sql` 扩展)，可以直接：

```

1 %load_ext duckdb
2 %%sql df <<
3 SELECT region, sum(amount) AS gmv FROM sales GROUP BY 1

```

或 `duckdb.execute(...).df()` 配合 `IPython.display`：

```

1 from IPython.display import display
2 re1 = con.sql("SELECT * FROM sales LIMIT 5")
3 display(re1)

```

在 Jupyter 里最舒服的方式其实是：直接用 `con.sql(query)` 让关系对象自己渲染 (DuckDB 实现了 `_repr_html_`)。

15.5.1 一个典型的分析会话

```

1 import duckdb
2 import pandas as pd
3
4 # 1) 连接内存库（分析中间结果不落盘）
5 con = duckdb.connect()
6
7 # 2) 读取（若干 GB 也没问题，因为列裁剪 + 谓词下推）
8 con.execute("""
9     CREATE VIEW events AS
10     SELECT * FROM read_parquet('lake/**/*.parquet',
11     hive_partitioning = true)
12 """)
13
14 # 3) 探索
15 con.sql("DESCRIBE events")
16 con.sql("SELECT count(*), min(ts), max(ts) FROM events")
17
18 # 4) 建模中间层
19 con.execute("""
20     CREATE TABLE daily AS
21     SELECT ts::DATE AS d, count(*) AS n, count(DISTINCT user_id) AS
22     uv
23     FROM events GROUP BY 1 ORDER BY 1
24 """)
25
26 # 5) 拉到 pandas 画图
27 daily_df = con.sql("SELECT * FROM daily").df()
28 daily_df.set_index("d")["uv"].plot(figsize=(12, 4), title="DAU")

```

```

27
28 # 6) 结果导出
29 con.execute("COPY (SELECT * FROM daily) TO 'daily.parquet' (FORMAT
    PARQUET)")

```

15.6 类型与 NULL 细节

pandas dtype	DuckDB 类型	注意
int64	BIGINT	—
float64	DOUBLE	—
object(str)	VARCHAR	—
object(混合)	VARCHAR 或报错	建议清理
bool	BOOLEAN	含 NaN 的 bool 列会出问题
datetime64[ns]	TIMESTAMP	—
datetime64[ns, tz]	TIMESTAMPTZ	—
category	ENUM / VARCHAR	DuckDB 会转成 VARCHAR (部分版本支持 ENUM)
NaN	NULL (DOUBLE 列)	NaN != NULL, 聚合时 NaN 会污染结果

处理 NaN:

```
1 | SELECT sum(CASE WHEN NOT isnan(x) THEN x END) AS s FROM df;
```

或用 pandas 先处理:

```
1 | df = df.replace({float("nan"): None})
```

15.7 常见场景模式

15.7.1 用 SQL 做特征工程, 再喂给 sklearn

```

1 | features = con.execute("""
2 |     SELECT
3 |         user_id,
4 |         count(*)                AS order_cnt,
5 |         avg(amount)            AS avg_amount,
6 |         max(amount)            AS max_amount,

```

```

7         date_diff('day', max(order_date), DATE '2026-09-17') AS
      recency,
8         list_distinct(list(category))                AS cats
9     FROM sales
10    GROUP BY user_id
11    """ ).df()
12
13 # cats 是 numpy array 列, 展开成 ML 特征
14 import sklearn.preprocessing as pp
15 mlb = pp.MultiLabelBinarizer()
16 cat_features = mlb.fit_transform(features.pop("cats"))

```

15.7.2 用 DuckDB 替代 read_csv 的内存爆炸

```

1 # ❌ 15 GB CSV, pandas 需要 ~40 GB 内存
2 # df = pd.read_csv("huge.csv")
3
4 # ✅ DuckDB: 流式读取 + 只留需要的列
5 df = con.execute("""
6     SELECT user_id, amount, ts
7     FROM read_csv_auto('huge.csv')
8     WHERE ts >= '2026-01-01'
9 """).df()

```

15.7.3 把 SQL 结果直接喂给绘图库

```

1 import matplotlib.pyplot as plt
2
3 con.sql("""
4     SELECT region, sum(amount) AS gmv FROM sales GROUP BY 1 ORDER BY
      gmv DESC
5 """).pl().plot.bar(x="region", y="gmv") # Polars 自带 plot
6 # 或 .df().plot.bar(x="region", y="gmv") # pandas 风格

```

15.7.4 多来源 JOIN: DataFrame + 文件 + 数据库

这是 DuckDB 最独特的能力之一:

```

1 # df_local 是 pandas; csv_file 是本地文件; pg 是挂载的 PostgreSQL; s3 是远
  程 parquet
2 result = con.execute("""
3     WITH local AS (SELECT * FROM df_local),
4         remote AS (SELECT * FROM
5     read_parquet('s3://bucket/x.parquet')),
6         biz AS (SELECT * FROM pg.public.orders)
7     SELECT l.k, count(*) AS n, sum(r.amount) AS amt
8     FROM local l
9     JOIN remote r ON l.k = r.k
10    JOIN biz b ON b.id = l.biz_id
11    GROUP BY l
12    ORDER BY amt DESC
13 """).df()

```

一条 SQL，跨越四种数据源。这在 pandas 里需要好几段胶水代码，还要处理中间内存。

15.8 性能注意事项

场景	建议
结果很小	<code>.df()</code> 没问题
结果很大	用 SQL 直接 <code>COPY TO</code> 文件，或用 <code>.arrow()</code> / <code>RecordBatchReader</code> 流式处理
反复使用同一个中间结果	物化成表或 Parquet，不要每次重算
DataFrame 很大且要多次 JOIN	<code>CREATE TABLE ... AS SELECT * FROM df</code> （物化后统计信息更准）
只做一次简单过滤	直接在 SQL 里做，别拉回来用 pandas
Chinese/nested 类型	优先 Arrow，pandas 会退化成 object 列

判断 DataFrame 大小：

```
1 print(df.memory_usage(deep=True).sum() / 1024**2, "MB")
```

实例演练

⚠ 本机 Python 环境没有 pip，无法安装 `duckdb` 的 Python 包，因此本章的 Python 代码未在本机实测（SQL/CLI 部分仍是实测的）。请以 `help(duckdb)` 与官方 API 文档为准。

实例 15-1 · 最省事的用法：直接查 DataFrame

```
1 import duckdb, pandas as pd
2
3 df = pd.DataFrame({"a": [1, 2, 3], "b": ["x", "y", "z"]})
4
5 # replacement scan: SQL 里直接写变量名
6 duckdb.sql("SELECT sum(a) AS total FROM df").show()
7 duckdb.sql("SELECT * FROM df WHERE a > 1").df()
```

工程代码里建议显式 `con.register("df_view", df)`，避免作用域问题。

实例 15-2 · 关系式 API：像 pandas 一样链式写 SQL

```
1 con = duckdb.connect("examples.db")
2
3 rel = (con.table("sales")
4       .filter("amount > 500")
5       .project("region, amount, order_date")
6       .aggregate("region, sum(amount) AS gmv")
7       .order("gmv DESC")
8       .limit(3))
9
10 print(rel.sql_query()) # 打印最终生成 SQL（学习/调试神器）
11 rel.show()
```

实例 15-3 · 结果太大时的三种取法

```
1 # (A) 小结果：直接转 pandas
2 df = con.execute("SELECT region, sum(amount) gmv FROM sales GROUP BY
3 1").df()
4
5 # (B) 中等：Arrow（零拷贝，再交给任何 Arrow 生态）
6 tbl = con.execute("SELECT * FROM sales").arrow()
7 print(tbl.num_rows, tbl.schema)
8
9 # (C) 超大：别拉回 Python，直接让 DuckDB 写文件
10 con.execute("COPY (SELECT * FROM sales) TO 'out.parquet' (FORMAT
11 PARQUET, COMPRESSION ZSTD)")
```

黄金法则：让 SQL 做重活（过滤/聚合/JOIN），只把几百行结果拉回 Python 画图/建模。

实例 15-4 · 参数绑定（防注入）

```
1 # ❌ 危险
2 con.execute(f"SELECT * FROM sales WHERE region = '{region}'")
3
4 # ✅ 正确
5 con.execute("SELECT * FROM sales WHERE region = ?", [region])
```

实例 15-5 · 一个典型的 Notebook 会话

```
1 import duckdb
2
3 con = duckdb.connect(config={"threads": "8", "memory_limit": "8GB"})
4
5 con.execute("""
6     CREATE VIEW events AS
7     SELECT * FROM read_parquet('events.parquet', hive_partitioning =
8     true)
9 """)
10 con.sql("SUMMARIZE events") # 探索
11 con.execute("""
12     CREATE TABLE daily AS
13     SELECT ts::DATE AS d, count(*) AS n, count(DISTINCT user_id) AS
14     uv
15     FROM events GROUP BY 1 ORDER BY 1
16 """)
17 con.sql("SELECT * FROM daily").df().set_index("d")
18 ["uv"].plot(title="DAU")
```

15.9 本章小结

- 零拷贝的 Arrow 互操作是 DuckDB 在数据科学生态成功的根本原因；
- `duckdb.sql("... FROM df")` 直接引用 Python 变量 (replacement scan)，但工程代码建议显式 `register`；
- `.df()` / `.arrow()` / `.pl()` 三种输出，按后续用途选择；
- 大结果集：不要 `.fetchall()`，更不要 `.df()` 拉全量，用 SQL 落盘或流式 Reader；
- 让 SQL 做重活、Python 只处理最终结果，是性能与可维护性的最佳平衡；
- 一条 SQL 可以同时跨越 DataFrame、本地文件、对象存储与远程数据库。

动手练习

1. 用 `.df()`、`.arrow()`、`.pl()` 三种方式取同一结果，比较返回类型与大致耗时。
2. 造一个含 NaN 的 pandas DataFrame，观察 DuckDB 里 `sum()` 的行为，并给出处理方案。
3. 用 replacement scan 与显式 `register` 两种方式查询同一个 DataFrame，测试在函数嵌套调用时两者的差异。
4. 尝试用 `RecordBatchReader` 分批读取大结果集（例如 `range(20_000_000)`）。
5. 写一个 notebook 风格的分析：从 `sales` 做 RFM 特征，再拉到 pandas。

下一章

[第 16 章 执行引擎内部](#) —— 想知道为什么快？去引擎里看看。

第 16 章 执行引擎内部

前面的章节告诉你"怎么用"。本章与下一章告诉你"为什么快"，以及"什么时候会变慢"。

理解内部原理，意味着你能**预判**而不是**猜测**一条 SQL 的性能。

16.1 三个关键词：列存、向量化、并行

16.1.1 列式存储 (Columnar Storage)

DuckDB 持久化的数据按列存储：

```
1  逻辑表 sales(order_id, region, amount, ...)
2  物理存储：
3      column[order_id] → [1, 2, 3, ...]          独立存储 + 压缩
4      column[region]   → ['华东', '华北', ...]    字典编码
5      column[amount]   → [123.4, 55.0, ...]
```

每个列被切成 **Row Group**（默认约 122,880 行一组），每个 Row Group 内部再切成更小的存储块。

好处：

1. **列裁剪**：只 SELECT 3 列就不读其它 15 列；
2. **更好的压缩**：同类数据相邻，字典/RLE/Delta 编码效果好；
3. **Zone Map**：每个 row group 记录 min/max，**查询时可以直接跳过**不满足条件的 entire group。

16.1.2 向量化执行 (Vectorized Execution)

传统行式数据库（如 PostgreSQL）的执行模型是**一次处理一行**：

```
1  for each row:
2      evaluate(expression1)
3      evaluate(expression2)
4      ...
```

每处理一行要调用几十上百次虚函数调用解释工作 **** per row****。这被称为 "tuple-at-a-time"。

DuckDB 是**一次处理一批 (Vector)**，每批 2048 个值：

```
1  for each vector (2048 values):
2      evaluate(expression1 on all 2048 values)    ← 可能编译成 SIMD 指令
3      evaluate(expression2 on all 2048 values)
4      ...
```

优势：

	逐行	向量化
函数调用次数	$N \times \text{表达式数}$	$(N/2048) \times \text{表达式数}$
分支预测	差（每行不同分支）	好（同一批模式相近）
CPU 缓存命中	差	好（连续内存）
SIMD	无法	可用（一条指令处理 4~8 个值）
解释开销摊销	—	摊销到 2048 个值上

这就是为什么 DuckDB 即使是“解释执行”（非 JIT）也能达到接近手写 C 的速度。

DataChunk 是 DuckDB 的执行单位：一批行 $\times$ 若干 Vector（列）。

16.1.3 并行执行（Parallelism）

```
1 | SELECT sum(amount) FROM sales WHERE region = '华东';
```

执行过程被自动拆分：

```
1 | 扫描阶段：多个线程各自读不同的 Row Group          ← 并行
2 | 过滤：      每个线程对自己的块做 WHERE              ← 并行
3 | 局部聚合：每个线程算自己的 partial sum              ← 并行
4 | 合并：      combine 所有 partial 结果                 ← 单线程（很便宜）
```

DuckDB 使用 **Morsel-Driven Parallelism**（以数据块为单位的动态调度）：

- 数据被切成许多小块（morsel）；
- 线程池里的 worker 动态领取任务；
- 快的 worker 领更多任务 → 自动负载均衡（避免数据倾斜导致的长尾）。

控制线程数：

```
1 | SET threads = 8;
2 | SELECT current_setting('threads');
3 | -- 设置为 0 表示使用全部核心（默认通常是核心数）
```

16.2 查询生命周期

一条 SQL 从文本到结果，经过以下阶段：

```
1 | SQL 文本
2 | |
3 | └─ 1. Parser          (语法树)
4 | |
5 | └─ 2. Binder          (语义分析：名字解析、类型检查、catalog 查找)
6 | |
```

7		└ 3. Logical Planner	(逻辑计划: Scan/Filter/Projection/Aggregate/Join)
8			
9		└ 4. Optimizer	(一系列 rewrite rules + cost-based 选择)
10			└ Expression rewriter (常量折叠、公共子表达式消除)
11			└ Filter pushdown (谓词下推)
12			└ Column pruning (列裁剪)
13			└ Cardinality estimation(基数估计)
14			└ Join order optimizer (连接顺序重排)
15			└ Common subexpression / CTE dedup
16			
17		└ 5. Physical Planner	(选择物理算子实现: Hash Join vs Merge Join 等)
18			
19		└ 6. Executor	(Pipeline + Scan/Operator Sink; 并行执行)

16.2.1 用 EXPLAIN 观察

```
1 | EXPLAIN SELECT region, sum(amount) FROM sales WHERE order_date > DATE
  | '2026-06-01' GROUP BY 1;
```

(第 17 章会逐项解读。)

16.2.2 优化器的重要规则

规则	作用
Filter Pushdown	把 WHERE 尽量推到扫描层，减少参与上层运算的数据
Column Pruning	只保留真正被使用的列，其余不读取
Cardinality Estimation	估算每个算子的输出行数，用于选择算法
Join Order Optimization	多表连接时选择最优顺序 (DuckDB 用 DPccp 之类的算法)
Constant Folding	1+1 在编译期算成 2
Count Star Optimization	count(*) 可以不读任何实际数据 (只读元数据)
Top-N 优化	ORDER BY x LIMIT 10 用堆实现，不需要全排序
Unnest/Lateral flattening	把 unnest 结构优化为更高效的算子
Aggregate Removal	移除了被上游去重后无用的聚合
Automatic read_csv filtering	读 CSV 时也能把过滤条件下推

你可以 (在调试时) 关闭某个优化器验证效果:

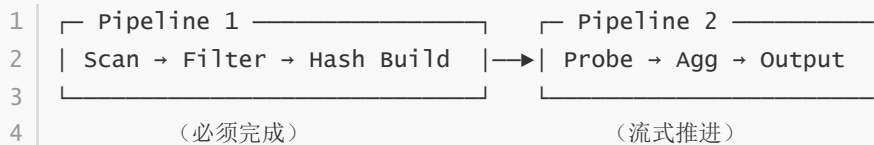
```
1 SET disabled_optimizers = 'filter_pushdown,statistics_propagate';
```

16.3 物化模式 vs 流式 (Pipelined) 执行

DuckDB 采用 **Push-based (推送)** 或者 **Pull-based** 的执行模型（内部演进到 push 为主）。

核心概念：

- **Pipeline**：一串可以流水线处理的算子（例如 `Scan → Filter → Projection`），数据像水一样流过；
- **Pipeline breaker（阻断物化点）**：必须拿到全部数据才能往下走的算子：
 - Hash Aggregate / Hash Join 的 build 端
 - 排序（ORDER BY 全局）
 - 窗口函数的分区边界



实践含义：准备留出足够内存给“阻断物化点”——它们会在内存中暂全部数据。这就是大表 GROUP BY / JOIN 时内存暴涨的原因。

16.4 关键物理算子

16.4.1 Table Scan

- 只读需要的列；
- 用 Zone Map 跳过 row group；
- 支持并行（按 row group 分配任务）；
- 支持 projection pushdown 到 Parquet 读取层（连压缩块都不解压）。

16.4.2 Filter

- 向量化执行，一次 2048 行；
- 多个谓词会被合并（AND 短路）；
- 常用常量会被特殊优化（例如 `x = NULL` 直接返回空结果）。

16.4.3 Hash Aggregate

```
1  线程1: 扫 Thread's partition → 局部 hash table → partial aggregate
2  线程2: ... (同样的)
3  合并: 把各线程的 hash table 归并
```

- 支持 **spill to disk** (内存不足时把 part 写临时文件)；

- 每个 hash table 的 key 宽度与基数决定内存占用。

16.4.4 Hash Join

两阶段：

1. **Build**：把**小表**（优化器估计的行数更少的一侧）在整个 hash table 里建立；
2. **Probe**：流式扫描大表，每行查 hash table。

优化点：

- Build 端会被物化成紧凑结构（SLOT/指针数组）；
- 支持并行 radix partitioning（把两侧数据分到多个独立分区并行处理）；
- 支持 spill（`SET temp_directory` 后自动）。

为什么"维表先去重"很重要：很小的表 $\approx$ hash table $\approx$ CPU cache 命中率，决定了 JOIN 速度。

16.4.5 Sort

- DuckDB 使用 **Sort-Merge** 与外部排序（可 spill）；
- 如果是 `ORDER BY ... LIMIT N`，会用 **Top-N heap**，避免全表排序。

```
1  -- 走 heap, 内存 O(N)
2  SELECT * FROM sales ORDER BY amount DESC LIMIT 10;
3
4  -- 必须全表排序, 内存 O(所有行)
5  SELECT * FROM sales ORDER BY amount DESC;
```

16.4.6 Window

窗口函数实现：

1. 按 `PARTITION BY` 键做 hash 分区 / 排序；
2. 每个分区内按 `ORDER BY` 排序（或复用已有顺序）；
3. 按帧逐行计算。

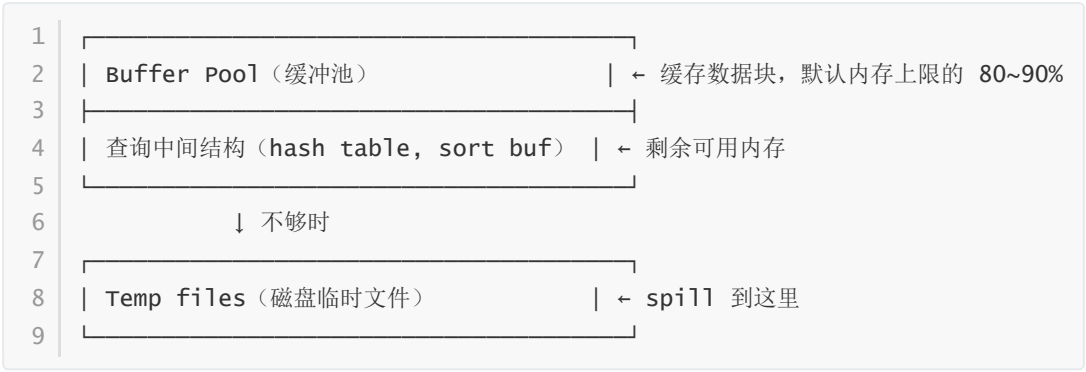
多个不同的窗口各需要一次排序/分区，成本高。所以第 10 章建议复用同一个 `WINDOW` 定义。

16.4.7 IEJoin / Piecewise Merge Join（不等值连接）

对 `a.x < b.y` 这类不等值条件，哈希连接不适用，DuckDB 使用 **IEJoin**（Inequality Join）或 similar 算法，比朴素 nested loop 快很多，但仍是 $O(n \log n)$ 级别，要警惕大表参与。

16.5 内存管理与 Spill

内存层次：



关键设置：

```
1 SET memory_limit = '12GB';  
2 SET temp_directory = '/fast/ssd/duckdb_tmp';  
3 SET max_temp_directory_size = '200GB';  
4  
5 -- 查看当前状态  
6 SELECT * FROM duckdb_memory();  
7 SELECT * FROM duckdb_temporary_files();
```

重要：发生 spill 时查询仍能完成（不会 OOM），只是变慢。**如果 spill 频繁**，请到第 17 章看调优手段。

16.6 为什么会"慢"：反模式清单

反模式	原因	修复
<code>SELECT *</code> 只用 3 列	破坏列裁剪	只写需要的列
<code>WHERE func(col) = 'x'</code>	无法下推到 Zone Map	改写为范围条件
大表 JOIN 大表且无过滤	hash table 巨大，可能 spill	先各自聚合/过滤
字符串做连接键	哈希成本高、内存占用大	换成整型代理键
高基数 GROUP BY（如 user_id 上亿）	hash table 巨大	分批、近似聚合、或预聚合
循环单条 INSERT	每行一个事务	Appender / 批量 INSERT SELECT
反复扫描同一大文件	无缓存	物化成 Parquet / 表
<code>ORDER BY</code> 无 LIMIT	全排序	加 LIMIT 或利用已有顺序
CTE 被引用多次且很重	重复计算	物化成临时表
在 Python 里循环调用 SQL	往返开销	合并成一条 SQL

实例演练：三个可亲手验证的小实验

理解原理最好的方式是**看见差异**。以下实验都可以在示例数据上跑（先执行 `examples/00-seed.sql`）。

实验一：列裁剪真的生效了吗

```
1 .timer on
2
3 -- A) 只读 1 列
4 SELECT sum(amount) FROM sales;
5
6 -- B) 读所有列（同样是为了算一个数，但把整张表数据都读出来了）
7 SELECT sum(amount) FROM (SELECT * FROM sales);
```

对比两者的执行时间，`EXPLAIN` 里也能看到 B 的 `Projections` 列出了全部 9 列，而 A 只有 `amount`。

实测（20 万行）： `EXPLAIN` 中 A 的 `Projections` 只有 `amount` 一列，B 则列出全部 9 列。耗时上 A 略快（0.006 s vs 0.007 s）——**这张表只有 9 列，差距不明显**；真实宽表（几十上百列）上差距可以达到数倍到 10 倍，因为列越多，“读不需要的列”浪费的 IO 越大。

实验二：并行度的影响

```
1 .timer on
2
3 SET threads = 1;
4 SELECT count(*) FROM sales WHERE amount > 500;
5
6 SET threads = 8;
7 SELECT count(*) FROM sales WHERE amount > 500;
```

预期：线程数 1 → 8 时耗时显著下降（但不总是线性：IO 与内存带宽会成为瓶颈）。用 `SELECT current_setting('threads')` 确认设置生效。

再看 `EXPLAIN` 里的扫描算子是否标记为并行：

```
1 EXPLAIN SELECT count(*) FROM sales;
```

实验三：Top-N 不全排序

```
1 -- A) 有 LIMIT: 走 Top-N 堆，内存 O(N)
2 .timer on
3 SELECT * FROM sales ORDER BY amount DESC LIMIT 10;
4
5 -- B) 没有 LIMIT: 必须全排序
6 SELECT * FROM sales ORDER BY amount DESC;
```

实测 (20 万行) :

查询	耗时
<code>ORDER BY amount DESC LIMIT 10</code>	0.001 s
<code>SELECT count(*) FROM (SELECT * FROM sales ORDER BY amount DESC)</code>	0.010 s

差 10 倍, 且后者要把 20 万行全部排序物化在内存里。

这也解释了: 给 BI 工具分页导出时, **一定要带 LIMIT**, 或者用第 13 章的批量导出。

16.8 本章小结

- 快的三个来源: **列存 (少 IO + 压缩 + Zone Map)**、**向量化 (2048 行一批, SIMD 友好)**、**并行 (morsel-driven 动态调度)** ;
- 查询经过 Parser → Binder → Logical Planner → Optimizer → Physical Planner → Executor;
- 优化器的两大杀手锏是**谓词下推**与**列裁剪**;
- Pipeline 中的"阻断物化点" (哈希表、排序、窗口分区) 是内存消耗大户, 也是 spill 的来源;
- Hash Join 先 Build 小表、再 Probe 大表 → 让维表小且已去重是关键;
- `ORDER BY ... LIMIT n` 走 Top-N heap, 不全排序;
- 内存不够时 DuckDB 自动 spill 到 `temp_directory`, 慢但能跑完。

思考题

1. `SELECT count(*) FROM huge_table` 为什么不需要读取任何数据? 它依赖什么元数据?
2. 为什么先 `WHERE` 再 `JOIN` 会更快? 在优化器自动做谓词下推的情况下, 什么时候还需要你手动调整?
3. 如果一张表按 `dt` 排序写入, 而另一个没有, 同样的 `WHERE dt = x` 查询性能会差多少? 为什么?
4. 为什么字符串作为连接键比整数慢? 请从哈希计算和数据宽度两个角度解释。

下一章

[第 17 章 EXPLAIN 与性能调优](#) —— 把原理变成可操作的检查清单。

第 17 章 EXPLAIN 与性能调优

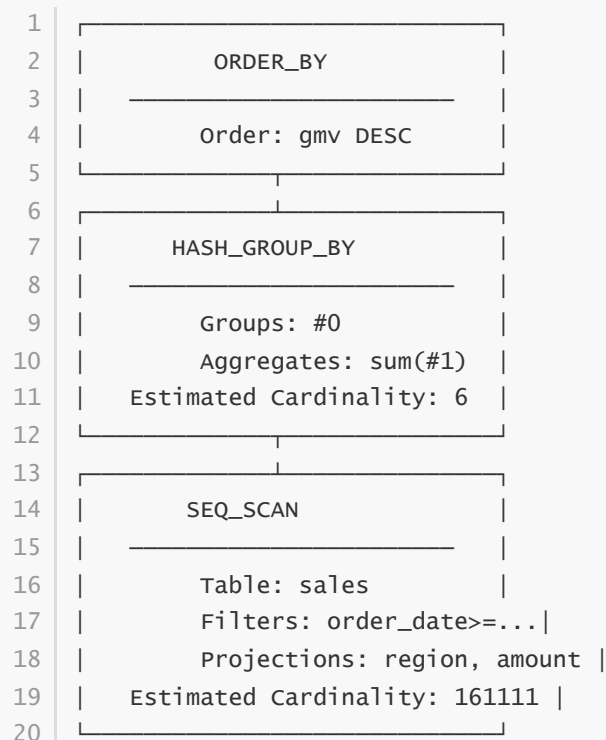
本章把第 16 章的原理变成**可操作的检查清单**：怎么读计划、怎么看真实耗时、有哪些旋钮可以调、常见慢查询怎么改。

17.1 三种查看计划的方式

17.1.1 EXPLAIN：估计的计划

```
1 EXPLAIN
2 SELECT region, sum(amount) AS gmv
3 FROM sales
4 WHERE order_date >= DATE '2026-03-01'
5 GROUP BY region
6 ORDER BY gmv DESC;
```

预期输出结构（示意）：



读计划的要点：

- **从下往上读**：数据从底部流出，向上传；
- **Projections** 行告诉你**读了哪些列**（确认列裁剪生效）；
- **Filters** 行确认**谓词下推**到扫描层；
- **Estimated Cardinality** 是优化器的行数估计，与实际差距大时说明统计信息不准，可能导致选错算法。

17.1.2 EXPLAIN ANALYZE: 真实执行数据

```
1 EXPLAIN ANALYZE
2 SELECT region, sum(amount) AS gmv FROM sales GROUP BY region;
```

输出会包含每个算子的实际耗时占比，例如：

```
1 |
2 |
3 | || Query Profiling Information ||
4 |
5 |
6 | SELECT ... FROM sales GROUP BY region;
7 |
8 |
9 | || Total Time: 0.0126s ||
10 |
11 |
12 | ... HASH_GROUP_BY ... (98.00%) ...
13 | ... SEQ_SCAN ... (Rows Out: 200000)
```

关注：

- **每行的百分比**：哪个算子最贵；
- 每个算子的实际输出行数（可以用来发现行数爆炸）；
- 除了 profile，`EXPLAIN ANALYZE` 会**真正执行**查询（有副作用，注意不要在写操作里乱用）。

17.1.3 Profiling 到文件 & 可视化

```
1 PRAGMA enable_profiling = 'json';
2 SET profiling_output = '/tmp/plan.json';
3 SET profiling_mode = 'detailed';      -- standard / detailed
4
5 -- 跑目标查询
6 SELECT * FROM sales WHERE amount > 500 LIMIT 10;
7
8 PRAGMA disable_profiling;
```

生成的 JSON 可以导入可视化工具查看（例如官方的 query plan visualizer）。

CLI 里还可以：

```
1 .explain      -- 某些版本的快捷命令
2 .timer on    -- 显示每条语句耗时（最常用）
```

17.1.4 快速计时的土办法

```
1 | .timer on
2 | SELECT count(*) FROM sales;
3 | -- Run Time (s): real 0.010 user 0.102 sys 0.008
```

Python:

```
1 | import time
2 | t = time.time()
3 | con.execute("SELECT ...").fetchall()
4 | print(f"{time.time()-t:.3f}s")
```

基准测试提示： 同一个查询至少跑 2~3 次取中位数，避免冷缓存与页缓存影响。

17.2 关键配置项 (PRAGMA / SET)

```
1 | -- 查看所有设置
2 | SELECT name, value, description FROM duckdb_settings() ORDER BY name;
3 | SELECT * FROM duckdb_settings() WHERE name LIKE '%thread%' OR name
   | LIKE '%memory%';
```

17.2.1 常用旋钮

设置	说明	建议
<code>threads</code>	并行线程数	默认=核数；共享机器上限制一下
<code>memory_limit</code>	单查询/总内存上限	物理内存的 50%~75%（留出 OS 与其它进程）
<code>temp_directory</code>	spill 目录	指向快速 SSD；空余空间要够
<code>max_temp_directory_size</code>	spill 空间上限	防止打满磁盘
<code>enable_progress_bar</code>	进度条	CI/日志里建议关闭
<code>preserve_insertion_order</code>	是否保持插入顺序	纯 ETL 场景可设为 false 提升并行度
<code>enable_object_cache</code>	HTTP 对象缓存	反复读远程文件时开启
<code>checkpoint_threshold</code>	WAL 多大时自动 checkpoint	大量写入时可适当调大
<code>wal_autocheckpoint</code>	同上（部分版本）	—

设置	说明	建议
<code>default_order / default_null_order</code>	排序默认/null 顺序	按需要设置
<code>TimeZone</code>	会话时区	统一设为 UTC 更可复现
<code>enable_statistics_propagation</code>	统计信息传播	通常保持 true
<code>disabled_optimizers</code>	调试用关闭优化器	只在排错时用

示例：

```

1 SET threads = 8;
2 SET memory_limit = '16GB';
3 SET temp_directory = '/mnt/fastdisk/duckdb_tmp';
4 SET max_temp_directory_size = '500GB';
5 SET enable_progress_bar = false;

```

17.2.2 传统 PRAGMA 形式

部分设置也支持 `PRAGMA` 写法：

```

1 PRAGMA memory_limit = '8GB';
2 PRAGMA threads = 4;
3 PRAGMA database_size;           -- 查看库的大小信息
4 PRAGMA storage_info;           -- 每张表的存储明细
5 PRAGMA show_tables;
6 PRAGMA table_info('sales');
7 PRAGMA enable_checkpoint_on_shutdown;
8 PRAGMA disable_checkpoint_on_shutdown;
9 SET disabled_optimizers = 'expression_rewriter';

```

更推荐用 `SET / duckdb_settings()` 的体系；`PRAGMA` 更像历史兼容层。

17.2.3 查看实际资源消耗

```

1 -- 当前已用/上限内存
2 SELECT * FROM duckdb_memory();
3
4 -- 是否有临时文件 (spill)
5 SELECT * FROM duckdb_temporary_files();
6
7 -- 每张表占用
8 SELECT table_name, estimated_size, column_count, index_count
9 FROM duckdb_tables() ORDER BY estimated_size DESC;

```

17.3 调优清单：从上到下

遇到慢查询时，按以下顺序排查：

第 1 步：先缩小数据量

```
1  -- 你是否真的需要全表？
2  SELECT count(*) FROM events;           -- 先看规模
3
4  -- 是不是在扫不需要的列？
5  SELECT * FROM events LIMIT 100;       -- 改成：SELECT col1,
    col2 ...
```

第 2 步：检查是否发生 Spill

```
1  EXPLAIN ANALYZE <慢查询>;
2  -- 或者跑完后：
3  SELECT * FROM duckdb_temporary_files();
```

有 spill ⇒ 加大 `memory_limit` 或改写查询（见 17.5）。

第 3 步：看行数是否合理

在 `EXPLAIN ANALYZE` 结果里，如果某个算子的输出行数远大于输入（行数爆炸），通常是 JOIN 键不唯一导致的。

```
1  -- 检查维表唯一性
2  SELECT join_key, count(*) FROM dim GROUP BY 1 HAVING count(*) > 1;
```

第 4 步：检查连接顺序

```
1  EXPLAIN SELECT ... FROM a JOIN b ... JOIN c ...;
```

如果 DuckDB 选了“大表 build hash”，可能是统计信息过时导致。可以：

- 用 CTE 强制先过滤/聚合；
- `ANALYZE`（如果版本支持更新统计信息）；
- 手动调整 JOIN 顺序。

第 5 步：检查文件格式与布局

```
1  SELECT * FROM parquet_metadata('big.parquet') LIMIT 20;
```

如果 row group 里的 min/max 覆盖整个值域，说明数据没有排序存储 → 无法跳过 row group → 考虑按常用过滤列重排后再写。

第 6 步：检查是否需要物化

```
1  -- 反复读取同一份大 CSV => 转 Parquet
2  COPY (SELECT * FROM read_csv_auto('big.csv'))
3  TO 'big.parquet' (FORMAT PARQUET, COMPRESSION ZSTD);
```

17.4 常见慢查询改写案例

案例 1：把函数从列上挪走

```
1  -- ❌ 慢
2  SELECT count(*) FROM logs WHERE strftime(ts, '%Y-%m') = '2026-03';
3
4  -- ✅ 快
5  SELECT count(*) FROM logs
6  WHERE ts >= TIMESTAMP '2026-03-01' AND ts < TIMESTAMP '2026-04-01';
```

案例 2：EXISTS 代替 DISTINCT + JOIN

```
1  -- ❌ 先 join 膨胀再去重
2  SELECT DISTINCT u.user_id FROM users u JOIN orders o ON u.user_id =
3  o.user_id;
4
5  -- ✅ 半连接，不膨胀
6  SELECT user_id FROM users SEMI JOIN orders USING (user_id);
```

案例 3：先聚合再连接

```
1  -- ❌ 先连接所有明细行再聚合
2  SELECT d.city, sum(f.amount)
3  FROM fact f JOIN dim d ON f.city_id = d.city_id
4  GROUP BY d.city;
5
6  -- ✅ 先在事实表内部按 city_id 聚合（行数大幅减少），再连维表
7  WITH agg AS (
8      SELECT city_id, sum(amount) AS amt FROM fact GROUP BY city_id
9  )
10 SELECT d.city, a.amt FROM agg a JOIN dim d USING (city_id);
```

第二种写法减少了哈希/网络的数据量，通常在维表很大时收益明显。

案例 4：Top-N 加 LIMIT

```

1  -- ❌ 全表排序
2  SELECT * FROM sales ORDER BY amount DESC;
3
4  -- ✅ 堆 top-N
5  SELECT * FROM sales ORDER BY amount DESC LIMIT 10;

```

案例 5: COUNT(DISTINCT) 的替代

```

1  -- 可能很贵
2  SELECT count(DISTINCT user_id) FROM events;
3
4  -- 若能接受近似
5  SELECT approx_count_distinct(user_id) FROM events;
6
7  -- 或两阶段 + 物化
8  SELECT count(*) FROM (SELECT DISTINCT user_id FROM events);

```

案例 6: 窗口函数 vs 自连接

```

1  -- ❌  $O(n^2)$ : 每个员工都跟同部门所有人比较
2  SELECT a.name, count(b.name)
3  FROM emp a LEFT JOIN emp b ON a.dept = b.dept AND b.salary > a.salary
4  GROUP BY a.name;
5
6  -- ✅  $O(n \log n)$ : 窗口函数
7  SELECT name, rank() OVER (PARTITION BY dept ORDER BY salary DESC) - 1
8  AS higher
9  FROM emp;

```

案例 7: 避免 OR 导致的全扫

```

1  -- ❌ 有时优化不好
2  WHERE region = '华东' OR amount > 900
3
4  -- ✅ 拆成 UNION ALL (当两类数据各自都能用 Zone Map 时)
5  SELECT * FROM t WHERE region = '华东'
6  UNION ALL
7  SELECT * FROM t WHERE amount > 900 AND region <> '华东';

```

(注意: Oracle/Postgres 技巧不总适用, DuckDB 通常能较好处理 OR, 实测为准。)

案例 8: 分批

超大任务按时间切:

```

1 for d in dates:
2     con.execute("""
3         INSERT INTO target
4         SELECT * FROM read_parquet(?) WHERE dt = ?
5         """, [file, d])

```

17.5 内存不足 (OOM) 的处理

17.5.1 现象

```

1 OutOfMemory Error: failed to allocate data of size XXXXXX
2 Out of Memory Error: could not allocate block of x bytes

```

或者查询变慢但能跑完 (spill 中)。

17.5.2 对策 (按优先级)

1. **减少参与的数据**: 加 WHERE、少 SELECT 列、先聚合;
2. **加大 memory_limit**: `SET memory_limit = '16GB';`
3. **打开 spill**: 确保 `temp_directory` 有足够空间 (默认在系统临时目录, 可能不够大);
4. **降低并行度**: `SET threads = 2` (每个线程都要自己的中间结构, 线程越多内存越大);
5. **拆分查询**: 按日期/键分批跑;
6. **改用显式流式写出**: `COPY (SELECT ...) TO ...` 会让中间结果流式落盘, 而不是全在内存;
7. **减少同时运行的查询**: 单机上并发查询会瓜分内存。

```

1 -- 一个"稳"的配置模板
2 SET memory_limit          = '12GB';
3 SET threads               = 4;
4 SET temp_directory        = '/mnt/nvme/duckdb_tmp';
5 SET max_temp_directory_size = '300GB';
6 SET preserve_insertion_order = false;

```

⚠ `preserve_insertion_order = false` 能让某些算子更早开始输出 (不必等待顺序), 提升并行度。代价是结果行顺序不再保证稳定。如果依赖行顺序 (如导出 CSV 的顺序), 不要关。

17.6 让存储布局帮你提速

17.6.1 排序写入

```
1 COPY (SELECT * FROM sales ORDER BY order_date, region)
2 TO 'sales_sorted.parquet' (FORMAT PARQUET, COMPRESSION ZSTD);
3
4 -- 之后
5 SELECT * FROM sales_sorted.parquet WHERE order_date = DATE '2026-03-
6 15';
7
8 -- Zone Map 能跳过绝大多数 row group
```

17.6.2 分区目录

```
1 COPY (SELECT *, strftime(order_date, '%Y-%m') AS ym FROM sales)
2 TO 'sales_by_month' (FORMAT PARQUET, PARTITION_BY (ym),
3 OVERWRITE_OR_IGNORE);
```

17.6.3 Row Group 大小

```
1 COPY t TO 't.parquet' (FORMAT PARQUET, ROW_GROUP_SIZE 1_000_000);
```

- 太小 → row group 太多，元数据开销大；
- 太大 → Zone Map 粒度粗，跳过效果差；
- 常用范围：10 万 ~ 100 万行/row group。

17.6.4 合理选择压缩

- 追求读取速度优先：SNAPPY；
- 追求存储成本：ZSTD；
- 需要兼容老工具：GZIP。

17.7 基准测试方法

一个可重复的性能测量模板：

```
1 .timer on
2
3 -- 预热（把文件放进 OS 缓存）
4 SELECT count(*) FROM target;
5
6 -- 正式
7 SELECT count(*) FROM target WHERE ...;
```

Python 版本（多次取中位数）：

```

1 import statistics, time
2
3 def bench(sql, n=5):
4     times = []
5     for _ in range(n):
6         t = time.perf_counter()
7         con.execute(sql).fetchall()
8         times.append(time.perf_counter() - t)
9     print(f"median: {statistics.median(times)*1000:.1f} ms  min:
    {min(times)*1000:.1f} ms")

```

注意事项：

- 关闭其它吃 CPU 的程序；
- 数据量足够大才有意义（毫秒级结果差异可能是噪声）；
- 注意 OS page cache：第一次运行通常慢很多；
- 比较两种写法时要用同一份数据、同样的配置。

实例演练

实例 17-1 · 读懂一份真实的 EXPLAIN

```

1 EXPLAIN
2 SELECT region, sum(amount) AS gmv
3 FROM sales
4 WHERE order_date >= DATE '2026-06-01'
5 GROUP BY region;

```

实测输出（自下而上读）：

```

1 | [ PROJECTION ] | ~6 rows | ← 最终输出 6 个区域
2 | [ PROJECTION ] |
3 | [ PROJECTION ] |
4 | [ HASH_GROUP_BY ] |
5 | Groups: #0 |
6 | Aggregates: sum_no_overflow(#1) |
7 | ~6 rows |
8 | [ HASH_GROUP_BY ] |
9 | [ HASH_GROUP_BY ] |
10 | [ PROJECTION ] | region, amount | ← 列裁剪：只留这 2 列
11 | [ PROJECTION ] |
12 | [ PROJECTION ] |
13 | [ SEQ_SCAN ] |
14 | Table: sales |
15 | Projections: region, amount | ← 只读 2 列（不是全部 9 列）
16 | Filters: order_date >= '2026-06-01' | ← 谓词下推成功
17 | ~40,000 rows | ← 估计扫描 4 万行（总 20 万的
18 | 1/5 )

```

三个要看的点：`Projections`（列裁剪是否生效）、`Filters`（谓词是否下推）、行数估计（是否与实际相符）。

实例 17-2 · 慢查询改写：把函数挪下列

```

1  .timer on
2  -- ❌ 0.011 s
3  SELECT count(*) FROM sales WHERE strftime(order_date, '%Y-%m') =
   '2026-03';
4
5  -- ✅ 0.001 s
6  SELECT count(*) FROM sales
7  WHERE order_date >= DATE '2026-03-01' AND order_date < DATE '2026-04-
   01';

```

写法	耗时
函数包住列	0.011 s
范围比较	0.001 s (快 11 倍)

实例 17-3 · 慢查询改写：EXISTS/SEMI 代替 JOIN + DISTINCT

```

1  -- ❌ 0.016 s
2  SELECT count(DISTINCT u.user_id) FROM users u JOIN sales s ON
   u.user_id = s.user_id;
3
4  -- ✅ 0.004 s
5  SELECT count(*) FROM (SELECT user_id FROM users SEMI JOIN sales USING
   (user_id));

```

写法	耗时
<code>JOIN + COUNT(DISTINCT)</code>	0.016 s
<code>SEMI JOIN</code>	0.004 s (快 4 倍)

数据量增大后差距会更大——因为前者会产生“膨胀的中间结果 + 一次全局去重”。

实例 17-4 · 强制 spill（观察临时文件）

```

1 SET memory_limit = '200MB';
2 SET temp_directory = '/tmp/duckdb_tmp';
3
4 SELECT region, category, count(DISTINCT user_id)
5 FROM sales GROUP BY 1, 2;           -- 高基数去重, 容易触发 spill
6
7 SELECT * FROM duckdb_temporary_files();

```

如果 `duckdb_temporary_files()` 有行, 说明发生了落盘。查询仍能完成 (不会 OOM), 只是变慢。

实例 17-5 · 排序写入带来的加速

```

1 COPY (SELECT * FROM sales ORDER BY order_date)
2 TO 'sales_sorted.parquet' (FORMAT PARQUET, COMPRESSION ZSTD);
3
4 -- 之后再按 date 过滤, Zone Map 可以跳过整个 row group
5 SELECT count(*) FROM 'sales_sorted.parquet'
6 WHERE order_date = DATE '2026-03-15';

```

17.8 本章小结

- `EXPLAIN` 看预估计划 (关注 Filters/Projections/Cardinality), `EXPLAIN ANALYZE` 看真实耗时 (会真跑一次);
- 调参三件套: `threads`、`memory_limit`、`temp_directory`;
- 慢查询排查顺序: 缩数据量 → 看 spill → 看行数 → 看 JOIN 顺序 → 看文件布局 → 物化;
- 常见改写: 函数挪下列、EXISTS 替代 JOIN+DISTINCT、先聚合再连接、加 LIMIT、近似 COUNT DISTINCT、窗口替代自连接;
- OOM 对策: 减数据 > 加内存 > 降 threads > 分批 > 用 COPY 流式写出;
- 布局即性能: 按常用过滤列排序写入、分区、合理的 row group 大小。

动手练习

1. 对自己写的最慢的一条查询跑 `EXPLAIN ANALYZE`, 找出占比最高的算子。
2. 用两种不同的写法实现同一个统计目标 (例如先 JOIN 再聚合 vs 先聚合再 JOIN), 计时对比。
3. 故意把 `memory_limit` 设为很小的值 (比如 200MB), 观察是否 spill, 检查 `duckdb_temporary_files()`。
4. 把 `sales` 按 `order_date` 排序重写 Parquet, 对比排序前后按日期范围查询的速度。
5. 写一个 Python 的 `bench()` 函数, 测试 5 个你常用的查询, 形成基线。

下一章

[第 18 章 持久化、事务与并发](#) —— 文件格式、WAL、ACID 边界、以及"为什么它是单进程写"。

第 18 章 持久化、事务与并发

本章回答三个在生产环境一定会遇到的问题：

1. 数据是怎么存在磁盘上的？（文件格式）
2. 崩溃了会不会丢数据？（WAL、ACID、检查点）
3. 能不能多个人/多个进程同时用？（并发模型）

18.1 持久化文件里到底有什么

打开一个持久化数据库：

```
1 | duckdb learn.db
```

磁盘上会出现：

```
1 | learn.db          ← 主数据文件
2 | learn.db.wal      ← 预写日志（Write-Ahead Log），仅在有未 checkpoint 的写
   |                  入时存在
```

如果正常关闭（`exit`），`.wal` 会被删除，因为所有数据都已经合并进主文件。

常见困惑：为什么我看到了一个 `.wal` 文件？→ 因为有进程还开着这个库，或者有未完成的 checkpoint。若进程崩溃，`.wal` 会保留，下次打开时自动回放恢复。

18.1.1 内部布局

```
1 | [
2 | | Database Header (校验和与版本信息) |
3 | |-----|
4 | | Block 0, Block 1, ... (每块固定大小若干 KB) |
5 | |   └─ Row Group / Column Segment 数据 |
6 | |   └─ 每个列的压缩bitmap 与统计信息 |
7 | |   └─ 索引/元数据 (如果有) |
8 | |-----|
9 | | Catalog / Schema 元数据 |
10 | | Free block list (空闲块管理) |
11 | ]
```

DuckDB 的存储格式自 **1.0 版本起保证向后兼容**，这意味着用 1.0 创建的数据库文件可以在更高版本打开。

反过来不成立：用新版本写过的库文件可能不能被旧版本读取（新特性会引入新的物理布局）。所以生产上要**统一版本**，或用 `EXPORT DATABASE` 做逻辑迁移。

18.1.2 查看存储信息

```
1 PRAGMA database_size;           -- 总大小、块数、空闲块等
2 PRAGMA storage_info;           -- 每张表/每列的存储明细
3 SELECT * FROM duckdb_tables(); -- 每张表的估算大小
```

示例输出解读：

```
1 PRAGMA database_size;
2 /*
3 database_size  VARCHAR → '1.2 GB'
4 block_size    BIGINT  → 262144
5 total_blocks  BIGINT  → 4800
6 used_blocks   BIGINT  → 4600
7 free_blocks   BIGINT  → 200
8 wal_size      VARCHAR → '0 bytes'
9 */
```

如果 `free_blocks` 很多，说明删除大量数据后有空洞，可以做 `CHECKPOINT` 或重建表来回收（见 18.5）。

18.2 WAL 与检查点 (Checkpoint)

18.2.1 为什么需要 WAL

列式存储的写入单位是“块”。如果每次 INSERT 都随机改写主文件的多个块：

- 非常慢（随机 IO）；
- 崩溃时可能留下半写状态（损坏）。

DuckDB 的做法和所有现代数据库一致：

```
1 写入请求 → 先顺序追加到 .wal → （稍后）批量合并回主文件
2                ↑ 崩溃安全                ↑ 称为 checkpoint
```

18.2.2 自动 checkpoint

默认当 WAL 大小超过阈值（`checkpoint_threshold`，默认约 16 MiB，具体取决于版本）时会自动触发。

```
1 SELECT * FROM duckdb_settings() WHERE name LIKE '%checkpoint%';
2 SET checkpoint_threshold = '64MiB';
```

手动触发：

```
1 CHECKPOINT;
2 FORCE CHECKPOINT; -- 即使没有写入也做整理
```

关闭连接时也会触发 checkpoint（可以通过 `disable_checkpoint_on_shutdown` 关闭，但一般不要）。

18.2.3 写入流程示意

```
1 BEGIN          → 写入 WAL
2 INSERT ...     → 追加到 WAL
3 UPDATE ...
4 COMMIT        → 写 commit 标记到 WAL
5                (WAL 到达阈值时自动 checkpoint, 合并到主文件)
```

因此：一个长事务会持续撑大 `.wal` 文件。批量导入时请务必**分批提交**或在完成后手动 `CHECKPOINT`。

18.3 ACID 与隔离级别

18.3.1 DuckDB 提供什么

属性	支持情况
Atomicity (原子性)	✅ 事务内要么全做，要么全不做
Consistency (一致性)	✅ 约束会被强制检查
Isolation (隔离性)	✅ 快照隔离 (Snapshot Isolation)
Durability (持久性)	✅ 提交后落盘 (WAL + checkpoint 保证)

18.3.2 快照隔离意味着什么

DuckDB **不是** read-committed。它的语义更像：

每个事务看到的是**它开始那一刻**的一致快照。别人在你之后提交的数据，你在当前事务里看不到。

```
1 -- 会话 A
2 BEGIN TRANSACTION;
3 SELECT count(*) FROM sales;      -- 看到 200000
4                                  -- 此刻会话 B 插入了 1000 行并提交
5 SELECT count(*) FROM sales;      -- 仍然看到 200000 (快照隔离)
6 COMMIT;
7 SELECT count(*) FROM sales;      -- 新事务，看到 201000
```

好处：分析任务不会被并发写入干扰（不会出现“同一个查询两次行数不同”）。

代价：长事务会阻止旧版本数据被清理。

18.3.3 事务控制

```
1 BEGIN TRANSACTION;           -- 或 BEGIN
2     INSERT INTO t VALUES (1);
3     DELETE FROM t WHERE id = 2;
4 COMMIT;                       -- 提交
5 -- ROLLBACK;                 -- 或回滚
```

特殊点：

- DDL (CREATE/ALTER/DROP) **可以在事务里执行** (这是 DuckDB 相比 MySQL 的巨大优势) ；
- 嵌套事务不支持 (用 SAVEPOINT 视版本而定，或避免嵌套) 。

```
1 BEGIN;
2 CREATE TABLE new_one (x INTEGER);
3 INSERT INTO new_one VALUES (1);
4 ROLLBACK;      -- 连表一起消失
```

Python 里：

```
1 con.begin()           # 或 con.execute("BEGIN")
2 try:
3     con.execute("INSERT INTO t SELECT * FROM staging")
4     con.commit()
5 except Exception:
6     con.rollback()
7     raise
```

注意 Python API 是否自动提交取决于版本：有的版本默认是 auto-commit (每条语句一个事务)，有的提供 `con.begin()`。落实到行为时请查版本文档，或显式写 `BEGIN/COMMIT` SQL。

18.3.4 写-写冲突

两个事务同时修改同一张表：

```
1 TransactionContext Error: Conflict on transaction with ... "another
  transaction depends on these tuples"
```

DuckDB 使用**乐观并发控制**：冲突时会报错让上层重试，而不是阻塞等待。

应用层应该：

```

1 import time
2 for attempt in range(5):
3     try:
4         con.execute("BEGIN")
5         con.execute("UPDATE ...")
6         con.execute("COMMIT")
7         break
8     except Exception as e:
9         con.execute("ROLLBACK")
10        if "Conflict" not in str(e):
11            raise
12        time.sleep(0.5 * (attempt + 1))

```

18.4 并发模型：单写多读（重点）

这是本章**最重要**的一节，也是 DuckDB 被用于生产时最容易误解的地方。

18.4.1 规则

场景	是否可行
一个进程（可以多线程） 读写	✓
多个进程 只读 同一个文件	✓
多个进程同时 写 同一个文件	✗（第二个进程启动会拿不到锁）
一个进程写 + 另一个进程 读	✗（读进程会收到 "file is being used by another process"）

第二个进程尝试打开一个已被打开的数据库文件时：

```

1 IO Error: Could not set lock on file "x.db": Resource temporarily
  unavailable

```

18.4.2 为什么这样设计

- DuckDB 是**进程内**数据库，没有像 PostgreSQL 那样的共享内存与后台检查点进程；
- 单进程独占写可以避免分布式锁、MVCC 清理等巨大的复杂度；
- OLAP 场景下绝大多数情况是"批处理写 → 分析读"，这个模型足够。

这是限制，也是它的简洁之源。 理解这一点，可以避免很多架构上的错误设计。

18.4.3 同一进程内的多线程

```
1 con = duckdb.connect("analytics.db")
2 # con 对象本身不应被多个线程同时使用 (safety: 每个线程自己新建 connection)
```

推荐做法：**每个线程/任务创建自己的连接**，并且最好用只读连接做查询：

```
1 import duckdb
2 from concurrent.futures import ThreadPoolExecutor
3
4 def query(region):
5     con = duckdb.connect("analytics.db", read_only=True) # 每个线程
        独立连接
6     try:
7         return con.execute(
8             "SELECT sum(amount) FROM sales WHERE region = ?",
9             [region]
10        ).fetchone()[0]
11    finally:
12        con.close()
13
14 with ThreadPoolExecutor(max_workers=8) as ex:
15     results = list(ex.map(query, regions))
```

DuckDB 的 Python 连接对象在某些版本里对互操作有保护机制 (`safe_named_locks` 等)，但**为每个线程单独建只读连接是最稳妥的模式**。

18.4.4 多进程协作的正确姿势

需求	方案
多个 BI 用户查询同一份数据	数据落成 Parquet 文件 （可被任意多个进程只读），而不是 db 文件
定时 ETL 写数据 + 别人查询	ETL 完成后 关闭连接 （释放锁），再开放给查询；或写到临时文件后用原子 rename 替换
需要并发写入	用消息队列/排队串行化；或用 PostgreSQL 作为写入端，DuckDB 只读分析
需要多作者的分布式数仓	DuckDB 不合适，考虑 ClickHouse/Trino/Spark

关键模式：写-发布分离

```

1 # 1) 在私有工作库里完成所有 ETL
2 con = duckdb.connect("build.db")
3 con.execute("CREATE OR REPLACE TABLE report AS SELECT ...")
4 con.execute("CHECKPOINT")
5 con.close() # 释放锁
6
7 # 2) 原子替换发布文件
8 import os
9 os.replace("build.db", "published.db") # 原子 rename

```

或者更简单：

```

1 # 构建期写临时文件，完成后 rename 到线上名字
2 mv report_new.parquet report.parquet

```

rename 在同一文件系统内是原子操作，读者要么看到旧文件、要么看到新文件，不会看到半成品。这是单机数据工程最常用的发布技巧。

18.4.5 只读模式

```

1 duckdb -readonly analytics.db

```

```

1 con = duckdb.connect("analytics.db", read_only=True)

```

只读的好处：

- 不会误改；
- 多个进程可以同时只读同一个文件（某些版本允许）；
- 不会写 WAL、不会 checkpoint。

若你的 BI/ODBC 工具只需要查询，请一定用只读模式。

18.5 维护与空间回收

18.5.1 删除大量数据后的空间

DELETE 不会产生立即的空间回收（和 PostgreSQL 的 VACUUM 类似）：

```

1 DELETE FROM logs WHERE dt < DATE '2025-01-01';
2 CHECKPOINT; -- 触发清理

```

或者更彻底地重建表：

```

1 CREATE OR REPLACE TABLE logs AS SELECT * FROM logs WHERE dt >= DATE
  '2025-01-01';

```

DuckDB 会自动做一些空闲块管理（`PRAGMA database_size` 里的 `free_blocks` 就是可复用空间），不必像 `VACUUM FULL` 那样频繁操作。

18.5.2 紧凑/优化

部分版本提供：

```
1 PRAGMA vacuum;           -- 视版本而定
2 PRAGMA compact;          -- 视版本而定
3 FORCE CHECKPOINT;
```

最可靠的重建手段始终是 **CTAS + rename**。

18.5.3 备份策略

三种层次：

1) 逻辑备份（推荐，跨版本安全）

```
1 EXPORT DATABASE 'backup_20260917';
```

2) 文件快照（要停机或确保已 checkpoint）

```
1 duckdb -c "CHECKPOINT" analytics.db    # 或让写入进程正常关闭
2 cp analytics.db analytics.db.bak
```

⚠ 不要在有活跃写入进程时直接 `cp` 主文件——可能拷到不一致状态（因为还需要配合 `.wal`）。

3) 复制一份干嘛？（导出 Parquet 更好）

```
1 COPY (SELECT * FROM target) TO 'backup/target.parquet' (FORMAT
  PARQUET, COMPRESSION ZSTD);
```

实践中：重要数据以 **Parquet/Delta** 形式存在对象存储里，DuckDB 的 db 文件可以视为可再生的缓存——这是最省心的方式。

18.6 什么时候用 db 文件，什么时候用 Parquet

一个常被问的问题。答案：

需求	推荐
数据需要长期归档、被多人/多进程访问	Parquet 文件 （+ 分区目录 / Delta / Iceberg）
需要事务、约束、upsert、增量更新	DuckDB db 文件
数据要分发给别人	Parquet / CSV

需求	推荐
中间结果、临时加速	db 文件或 memory 库
需要保存 SQL 视图、宏、配置	DuckDB db 文件
数据要被 BI 工具长期连接	Parquet（避免锁问题）或只读 db

最常见的新手错误：把生产数据锁在单个 `.db` 文件里，结果被并发访问问题困住。请把 Parquet 当作“数据资产格式”，把 `.db` 当作“工作区格式”。

实例演练

实例 18-1 · DDL 也能回滚

```
1 BEGIN;
2 CREATE TABLE will_rollback (x INTEGER);
3 ROLLBACK;
4
5 SELECT count(*) FROM duckdb_tables() WHERE table_name =
  'will_rollback';
6 -- 0
```

这是 DuckDB 相对 MySQL 的巨大优势：建表 + 灌数据可以作为一个原子操作。

实例 18-2 · 只读模式的写入报错（原文）

```
1 duckdb -readonly examples.db -c "CREATE TABLE z AS SELECT 1;"

1 Invalid Input Error: Cannot execute statement of type "CREATE" on
  database "examples"
2 which is attached in read-only mode!
```

实例 18-3 · WAL 文件的出现与消失

```
1 duckdb demo.db -c "CREATE TABLE t AS SELECT * FROM range(1000) x(i);"
2 ls -lh demo.db*           # 正常关闭，.wal 会被清理
3
4 # 如果进程被强杀（或事务未提交），.wal 会留下，下次打开时自动回放恢复
```

查看当前 WAL 大小：

```
1 PRAGMA database_size;  -- 看 wal_size 列
```

database_name	database_size	wal_size
examples	8.0 MiB	0 bytes

`wal_size = 0` 说明改动都已 checkpoint 进主文件。

实例 18-4 · 备份：逻辑导出 vs 文件拷贝

```
1  -- （推荐）逻辑备份：跨版本安全
2  EXPORT DATABASE 'backup_dir';
3  -- 生成 backup_dir/schema.sql、load.sql、data/*.parquet
4  IMPORT DATABASE 'backup_dir';
```

```
1  # 文件拷贝：必须先让写入方 checkpoint/关闭，否则可能拷到不一致状态
2  duckdb prod.db -c "CHECKPOINT"
3  cp prod.db prod.db.bak
```

实例 18-5 · 发布模式：构建 → 原子替换

```
1  import os
2  import duckdb
3
4  con = duckdb.connect("build.db")
5  con.execute("CREATE OR REPLACE TABLE report AS SELECT ...")
6  con.execute("CHECKPOINT")
7  con.close()                # 释放文件锁
8
9  os.replace("build.db", "published.db") # 原子 rename
```

读者要么看到旧文件、要么看到新文件，永远不会看到半截状态。这是单机数据工程最常用的发布技巧。

18.7 本章小结

- 持久化库 = 主文件 + 可选的 `.wal`；文件格式自 1.0 起向后兼容（新版本写的旧版本未必能读）；
- 写入先到 WAL、再 checkpoint 合并；`CHECKPOINT` 可以手动触发；
- DuckDB 提供完整 ACID，隔离级别是**快照隔离**（不是 read-committed）；
- DDL 也支持事务回滚，这是 DuckDB 很爽的特性；
- **单进程写、多进程读**：第二个进程无法打开已被写的库文件；
- 多进程的正确姿势是 **写 — 关闭 — 原子 rename 发布**，或干脆把数据以 Parquet 形式发布；
- 备份优先用 `EXPORT DATABASE`，或者依赖对象存储里的 Parquet；
- 把 Parquet 当作资产格式，把 `.db` 当作工作区格式。

动手练习

1. 打开一个库、插入数据，观察 `.wal` 文件的出现与消失；再执行 `CHECKPOINT` 看看变化。
2. 在两个终端里同时打开同一个库文件，观察第二个进程的错误信息。改用 `-readonly` 再试。
3. 在事务里执行 `CREATE TABLE + INSERT` 然后 `ROLLBACK`，验证表消失了。
4. 用多线程（8 个）各自建立只读连接查询，验证是否安全。
5. 写一个"构建到临时库 → `CHECKPOINT` → `os.replace` 发布"的 Python 脚本。

下一章

[第 19 章 扩展生态与自定义函数](#) —— 用扩展把 DuckDB 变成任何你想要的工具。

第 19 章 扩展生态与自定义函数

DuckDB 核心保持小而精，把大量功能交给了**扩展 (Extension)**。本章讲清楚：

- 扩展怎么安装与加载（含自动安装/自动加载机制）；
- 哪些官方扩展最值得用；
- 社区扩展的使用方法；
- 如何用 **SQL 宏**和 **Python UDF** 给 DuckDB 添加新函数。

19.1 扩展机制

19.1.1 安装与加载

```
1  INSTALL spatial;          -- 下载二进制到
   ~/.duckdb/extensions/<version>/<platform>/
2  LOAD spatial;            -- 把扩展加载进当前会话
3
4  -- 也可以在 Python 里
5  con.execute("INSTALL json; LOAD json")
```

查看状态：

```
1  SELECT extension_name, loaded, installed, install_path, description
2  FROM duckdb_extensions()
3  ORDER BY extension_name;
```

字段含义：

- `installed`：二进制是否已在本地；
- `loaded`：是否已加载到当前会话（只有 `loaded` 的函数才能用）；
- `install_path`：二进制位置。

卸载/更新：

```
1  UNINSTALL spatial;        -- 删除本地二进制
2  FORCE INSTALL spatial;     -- 强制重新下载（用于修复损坏）
3  UPDATE EXTENSIONS;       -- 更新所有已安装扩展（较新版本）
```

扩展与 **DuckDB 版本强绑定**。升级 DuckDB 后需要重新安装扩展（`INSTALL` 会下载对应版本）。

19.1.2 自动安装与自动加载

```
1 SET autoinstall_known_extensions = true; -- 用到时自动 INSTALL
2 SET autoload_known_extensions   = true; -- 用到时自动 LOAD
```

这两个通常默认开启。所以你常常不需要手写 `INSTALL httpfs`，直接：

```
1 SELECT * FROM 'https://example.com/data.parquet'; -- 会自动装载 httpfs
```

判断“是否 known”：只有官方列表里的已知扩展才能自动安装，避免任意代码执行。

19.1.3 签名与安全

官方发布的扩展都带签名。加载未签名扩展需要：

```
1 duckdb -unsigned
```

或者：

```
1 SET allow_unsigned_extensions = true;
```

只在本地测试时这么做，**生产环境务必使用签名扩展**。

19.1.4 扩展的安装位置与离线部署

默认目录：

```
1 ~/.duckdb/extensions/<duckdb_version>/<platform>/
```

离线环境（内网）：

1. 在能联网的机器上 `INSTALL`（或直接下载 `.duckdb_extension.gz`）；
2. 把文件拷到目标机器的对应目录；
3. 目标机器 `LOAD` 即可（若不能自动解压，用 `gunzip` 生成 `.duckdb_extension` 文件）。

自定义目录：

```
1 SET extension_directory = '/opt/duckdb/extensions';
```

19.2 值得掌握的官方扩展

说明：下表列出的是长期存在的核心能力，具体某个扩展是**内置**还是需要 `INSTALL` 会随版本变化。请以 `SELECT * FROM duckdb_extensions()` 结果为准。

扩展	用途	典型用法
----	----	------

扩展	用途	典型用法
<code>parquet</code>	Parquet 读写	<code>read_parquet</code> 、 <code>COPY ... TO x.parquet</code>
<code>json</code>	JSON 类型与函数	<code>read_ndjson</code> 、 <code>json_extract</code>
<code>httpfs</code>	HTTP/S3 对象存储	<code>read_parquet('s3://...')</code>
<code>autocomplete</code>	CLI 自动补全	CLI 自动加载
<code>icu</code>	国际化排序与时区	<code>ORDER BY name COLLATE NOCASE</code>
<code>tpch / tpcds</code>	生成基准测试数据	<code>dbgen(sf=1)</code> 、 <code>CALL dsdgen(sf=1)</code>
<code>fts</code>	全文检索	<code>PRAGMA create_fts_index(...)</code>
<code>spatial</code>	地理空间 (PostGIS 能力)	<code>ST_Point</code> 、 <code>ST_Intersects</code>
<code>excel</code>	读写 xlsx	<code>read_xlsx(...)</code>
<code>sqlite</code>	挂载 SQLite 文件	<code>ATTACH 'a.db' (TYPE SQLITE)</code>
<code>postgres</code>	连接 PostgreSQL	<code>ATTACH '...' (TYPE POSTGRES)</code>
<code>mysql</code>	连接 MySQL	<code>ATTACH '...' (TYPE MYSQL)</code>
<code>delta</code>	Delta Lake	<code>delta_scan('...')</code>
<code>iceberg</code>	Apache Iceberg	<code>iceberg_scan(...)</code>
<code>avro</code>	Avro 文件	<code>read_avro(...)</code>
<code>vss</code>	向量检索 (HNSW)	embedding 相似度搜索
<code>sqlsmith</code>	SQL 模糊测试 (开发工具)	随机 SQL 生成
<code>jemalloc</code>	替换内存分配器	某些负载下性能更好

19.2.1 tpch / tpcds：一键生成测试数据

```

1  INSTALL tpch; LOAD tpch;
2  CALL dbgen(sf = 1);           -- 生成 SF=1 (约 1GB) 的 TPC-H 数据集
3
4  SHOW TABLES;                -- 出现 lineitem、orders、customer ...
5  SELECT count(*) FROM lineitem;
6
7  -- 跑一个标准 TPC-H Q1
8  SELECT
9      l_returnflag, l_linestatus,
10     sum(l_quantity)           AS sum_qty,
```

```

11      sum(l_extendedprice)                AS
      sum_base_price,
12      sum(l_extendedprice * (1 - l_discount)) AS
      sum_disc_price,
13      sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) AS
      sum_charge,
14      avg(l_quantity)                    AS avg_qty,
15      count(*)                          AS count_order
16 FROM lineitem
17 WHERE l_shipdate <= DATE '1998-12-01' - INTERVAL 90 DAY
18 GROUP BY 1, 2
19 ORDER BY 1, 2;

```

这是练习性能调优的最佳素材。

19.2.2 fts: 全文检索

```

1  INSTALL fts; LOAD fts;
2
3  CREATE TABLE docs (id INTEGER, body VARCHAR);
4  INSERT INTO docs VALUES (1, 'DuckDB is fast'), (2, 'DuckDB is in-
   process');
5
6  PRAGMA create_fts_index('docs', 'id', 'body');
7
8  SELECT id, score, body
9  FROM (
10     SELECT *, fts_main_docs.match_bm25(id, body, 'in-process') AS
       score
11     FROM docs
12 ) SQ
13 WHERE score IS NOT NULL
14 ORDER BY score DESC;

```

注意: `fts_main_docs.match_bm25` 这个名字是自动生成的
(`<schema>_<table>.match_bm25`) , 具体名字以创建索引后的提示为准。

19.2.3 spatial: 地理数据

```

1  INSTALL spatial; LOAD spatial;
2
3  SELECT ST_AsText(ST_Point(1, 2))        AS p;
4  SELECT ST_Distance(ST_Point(0,0), ST_Point(3,4)) AS d;
5  SELECT ST_Contains(ST_MakeEnvelope(0,0,10,10), ST_Point(5,5)) AS
       inside;
6
7  -- 读 GeoJSON / Shapefile
8  SELECT * FROM ST_Read('roads.geojson');

```

19.2.4 vss: 向量相似度检索

```
1  INSTALL vss; LOAD vss;
2
3  CREATE TABLE items (
4      id    INTEGER,
5      emb   FLOAT[3]
6  );
7  INSERT INTO items VALUES (1, [0.1, 0.2, 0.3]), (2, [0.9, 0.1, 0.0]);
8
9  CREATE INDEX idx ON items USING HNSW (emb);
10
11 SELECT id FROM items ORDER BY array_distance(emb,
12     [0.1,0.2,0.3]::FLOAT[3]) LIMIT 3;
```

这是做本地 RAG/语义检索的常见手段。

19.3 社区扩展

除了官方维护的，社区也贡献了大量扩展。它们通过一个中心索引分发：

```
1  -- 语法（会自动从社区仓库安装）
2  INSTALL quack FROM community;
3  LOAD quack;
4
5  INSTALL ui FROM community;
6  LOAD ui;
```

查看可用列表：

- 官网的社区扩展列表页；
- 或直接问 DuckDB：

```
1  SELECT * FROM duckdb_extensions();  -- 已安装/已列出的
```

常见社区扩展：

扩展	用途
<code>ui</code>	启动一个本地 Web UI 查看执行计划与历史查询
<code>gsheet</code>	读写 Google Sheet
<code>h3</code>	Uber H3 地理网格
<code>chsql</code> / <code>prql</code>	替代查询语言
<code>sqlan</code> / <code>ldgograph</code>	各种格式/可视化
<code>crypto</code>	哈希/加密工具

扩展	用途
<code>datasketches</code>	高级近似数据结构

安装第三方扩展要谨慎：它们不受官方签名约束，运行在前述的安全边界之外。只在可信来源使用。

19.3.1 UI 扩展（很好用）

```
1 | INSTALL ui FROM community;
2 | LOAD ui;
```

启动后访问终端提示的地址（通常是 `http://localhost:4213`），可以看到：

- 漂亮的 SQL 编辑器；
- 查询历史的火焰图/执行计划可视化；
- 表浏览。

它在优化慢查询时非常方便。

19.4 用 SQL 宏（Macro）扩展 SQL

宏是"SQL 里的函数"，零依赖、跨版本稳定，是**首选的扩展方式**。

19.4.1 标量宏

```
1 | CREATE OR REPLACE MACRO is_high(x) AS x >= 500;
2 |
3 | SELECT is_high(amount), count(*) FROM sales GROUP BY 1;
4 |
5 | CREATE MACRO add_days(d, n) AS d + INTERVAL (n) DAY;
6 | SELECT add_days(DATE '2026-01-01', 30);
7 |
8 | -- 多参数是天然的
9 | CREATE MACRO pct_change(old_v, new_v) AS
10 |     CASE WHEN old_v = 0 OR old_v IS NULL THEN NULL
11 |           ELSE round(100.0 * (new_v - old_v) / old_v, 2) END;
12 |
13 | SELECT pct_change(100, 150);    -- 50.00
```

19.4.2 表宏（Table Macro）

返回一个结果集，类似表函数：

```

1 CREATE OR REPLACE MACRO top_by_region(t, n) AS TABLE
2     SELECT * FROM query_table(t)
3     QUALIFY row_number() OVER (PARTITION BY region ORDER BY amount
4     DESC) <= n;
5 SELECT * FROM top_by_region('sales', 3);

```

19.4.3 管理宏

```

1 SELECT function_name, parameters, return_type, macro_definition
2 FROM duckdb_functions()
3 WHERE function_type = 'macro';
4
5 DROP MACRO IF EXISTS add_days;

```

宏会被持久化到数据库文件里（作为 catalog 的一部分），随库一起分发。

19.5 Python UDF

当 SQL 或宏不够用时，可以用 Python 写标量函数。

19.5.1 基本用法

```

1 import duckdb
2
3 def add_one(x: int) -> int:
4     return x + 1
5
6 con = duckdb.connect()
7 con.create_function("add_one", add_one, [duckdb.typing.BIGINT],
8     duckdb.typing.BIGINT)
9
10 con.sql("SELECT add_one(41) AS answer").show()      # 42
11 con.sql("SELECT add_one(id) FROM some_table LIMIT 3").show()

```

19.5.2 类型映射

```

1  import duckdb
2
3  duckdb.typing.BIGINT
4  duckdb.typing.DOUBLE
5  duckdb.typing.VARCHAR
6  duckdb.typing.BOOLEAN
7  duckdb.typing.DATE
8  duckdb.typing.TIMESTAMP
9  duckdb.typing.BLOB
10 duckdb.typing.DECIMAL(12, 2)
11 duckdb.typing.list_type(duckdb.typing.INTEGER)
12 duckdb.typing.struct_type({"a": duckdb.typing.INTEGER})

```

省略参数/返回类型时 DuckDB 会**基于样本推断**（对第一批数据推断），这有风险，**建议显式声明**。

19.5.3 NULL 处理

默认是 `null_handling="default"`：只要任一输入为 NULL，DuckDB **不会调用 Python**，直接返回 NULL（速度快）。若你希望 Python 自己看到 None 并决定返回值：

```

1  con.create_function(
2      "safe_len",
3      lambda s: len(s) if s is not None else 0,
4      [duckdb.typing.VARCHAR],
5      duckdb.typing.BIGINT,
6      null_handling="special",    # 默认为 "default"
7  )

```

19.5.4 异常与副作用

```

1  con.create_function(
2      "risky", my_func,
3      [duckdb.typing.VARCHAR], duckdb.typing.VARCHAR,
4      exception_handling="return_null",    # 出错返回 NULL 而不是中断查询
5      side_effects=False                   # 声明无副作用，允许优化器重排/常量
        折叠
6  )

```

19.5.5 性能提醒（非常重要）

Python UDF 会对每一行（或每批）调用 Python，速度比原生 SQL 慢**几十到几百倍**：

```

1  # ❌ 慢：100 万行要调 100 万次 Python
2  con.sql("SELECT my_udf(x) FROM big_table")
3
4  # ✅ 快：能用内置函数就用内置函数
5  con.sql("SELECT upper(x) FROM big_table")
6
7  # ✅ 更好：把逻辑写成 SQL 表达式（或用 CASE WHEN）

```

适用场景：调用三方库（如 jieba 分词、地理编码、调用 API）、极其特殊的业务规则。

19.5.6 一个实际例子：中文分词

```
1 import duckdb
2 import jieba
3
4 def tokenize(text: str) -> list:
5     return [w for w in jieba.cut(text) if w.strip()]
6
7 con = duckdb.connect()
8 con.create_function(
9     "tokenize", tokenize,
10    [duckdb.typing.VARCHAR],
11    duckdb.typing.list_type(duckdb.typing.VARCHAR),
12    null_handling="special"
13 )
14
15 con.sql("""
16     SELECT tokenize(content) AS words
17     FROM (VALUES ('我爱 DuckDB 教程')) AS t(content)
18 """).show()
```

这类"Python 有生态、SQL 没有"的场景，正是 UDF 的价值所在。

19.6 应用层自定义函数（其他语言）

- **C/C++**：可以编写原生扩展（正式的 extension 开发），性能最好；
- **Java/Go/Rust**：通常的用法是实现为 UDF 或者在应用层做后置处理；
- **WASM**：暂无用户级 UDF 机制，建议用 SQL 宏。

写真正的原生扩展涉及 DuckDB 的 C++ API 与构建系统，超出本书范围，可查阅官方文档的 "Extensions / Building" 部分。

实例演练

实例 19-1 · 看看你有哪些扩展

```
1 SELECT extension_name, loaded, installed
2 FROM duckdb_extensions()
3 WHERE installed OR loaded
4 ORDER BY extension_name;
```

实测（1.5.5 默认）：

extension_name	loaded	installed
autocomplete	true	true

extension_name	loaded	installed
core_functions	true	true
icu	true	true
json	true	true
parquet	true	true
shell	true	true

注意：已安装 (installed) ≠ 已加载 (loaded)。用到函数时必须 `LOAD`（或依赖自动加载）。

实例 19-2 · 在线安装一个扩展

```
1 | INSTALL httpfs;
2 | LOAD httpfs;
3 | SELECT extension_name, loaded, installed
4 | FROM duckdb_extensions() WHERE extension_name = 'httpfs';
```

extension_name	loaded	installed
httpfs	true	true

扩展与 DuckDB 版本强绑定，升级 DuckDB 后要重新 `INSTALL`。

实例 19-3 · 用 tpch 一键生成基准测试数据

```
1 | INSTALL tpch;
2 | LOAD tpch;
3 | CALL dbgen(sf = 0.01);           -- SF=0.01 ≈ 10MB
4 | SELECT count(*) AS lineitem_rows FROM lineitem;
```

lineitem_rows
60175

想练习性能调优却没有大数据？`sf = 1`（约 1 GB）就可以，TPC-H 的 22 条查询是标准考题。

实例 19-4 · SQL 宏：不写 UDF 也能扩展函数

```

1 CREATE OR REPLACE MACRO add_days(d, n) AS d + INTERVAL (n) DAY;
2 SELECT add_days(DATE '2026-01-01', 30) AS d30;           -- 2026-01-31
3
4 CREATE OR REPLACE MACRO pct_change(o, nw) AS
5     CASE WHEN o = 0 OR o IS NULL THEN NULL
6         ELSE round(100.0 * (nw - o) / o, 2) END;
7 SELECT pct_change(100, 150) AS pc;                     -- 50.00
8
9 -- 查看与删除
10 SELECT function_name FROM duckdb_functions() WHERE function_type =
    'macro';
11 DROP MACRO add_days;

```

宏会被持久化到数据库文件里，随库分发——这是“零依赖扩展”的首选方式。

实例 19-5 · Python UDF（模板，未在本机实测）

```

1 import duckdb
2
3 def mask_phone(p: str) -> str:
4     return p[:3] + "****" + p[-4:] if p else None
5
6 con = duckdb.connect()
7 con.create_function(
8     "mask_phone", mask_phone,
9     [duckdb.typing.VARCHAR],
10    duckdb.typing.VARCHAR,
11    null_handling="special",          # 让 Python 自己看到 None
12    exception_handling="return_null", # 出错返回 NULL 而不是中断查询
13 )
14
15 con.sql("SELECT mask_phone('13812345678') AS masked").show() #
    138****5678

```

⚠ Python UDF 每行都要跨越 Python 边界，比内置函数慢几十到几百倍。只在“必须调用三方库”时用。

19.7 本章小结

- 扩展是 DuckDB 的能力放大器： `INSTALL` 下载、 `LOAD` 启用、 `UNINSTALL` 删除；
- 新版通常默认支持**自动安装/自动加载**已知扩展；
- 常用官方扩展：parquet、json、httpfs、icu、fts、spatial、delta/iceberg、postgres/mysql/sqlite、tpch（测试数据）、vss（向量）；
- 社区扩展通过 `INSTALL xxx FROM community` 安装（如非常好用的 `ui`）；
- **优先用 SQL 宏**扩展能力，简单、快、可持久化；
- Python UDF 很灵活但慢：只在必需时用，务必显式声明类型、处理好 NULL 与异常；
- 扩展与 DuckDB 版本绑定，升级后要重新安装。

动手练习

1. 安装 `tpch`，生成 SF=0.1 的测试数据，跑一遍 TPC-H Q1 并观察耗时。
2. 安装 `httpfs`，读一个公开的远程 Parquet 文件。
3. 创建一个 SQL 宏，把"金额分档"的业务规则固化下来，并在多个查询里复用。
4. 写一个 Python UDF（比如手机号脱敏： `138****1234`），在 SQL 中调用。
5. 安装社区扩展 `ui`，用它观察一次慢查询的执行计划。

下一章

[第 20 章 生产实践与排错](#) —— 章节 checklist、常见错误速查、推荐架构。

第 20 章 生产实践与排错

最后这一章，把前面的知识收束成可以落地的工程实践：架构建议、可靠性 checklist、常见错误速查表。

20.1 推荐的三种生产架构

架构 A：本地分析工具（最常见）

```
1 | CSV/Parquet/日志 → DuckDB（本地进程）→ Parquet / 报表 / Python 可视化
```

- 无需服务器；
- 数据以文件形式存在；（利于共享与备份）
- 适合个人/小团队的数据分析。

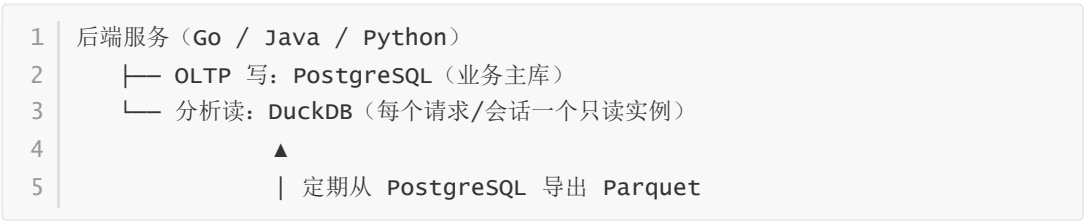
架构 B：单机 ETL 服务



关键点：

- 每次运行使用独立的临时工作库，完成后发布新文件（原子 rename）；
- 幂等：重复运行结果一致；
- 有数据质量断言；
- 有日志与告警。

架构 C：应用层嵌入式分析



关键点：

- DuckDB 实例**每次新建**（读 Parquet，不长期持有某个 db 文件的写锁）；
- 或者只读挂载一个已发布好的 db 文件；
- 控制内存（每请求太大内存会互相挤压）。

20.2 上线前的 checklist

类别	检查项
数据	是否已Parquet化（而不是让每个查询都读 CSV）？
	是否按常用过滤列排序写入？
	分区是否合理（低基数、粒度合适）？
性能	<code>threads</code> 是否与机器匹配（别吃满共享机器）？
	<code>memory_limit</code> 是否设置？留了余量给 OS？
	<code>temp_directory</code> 是否指向足够大的快速磁盘？
	是否 <code>SELECT *</code> （应该指定列）？
	慢查询是否看过 <code>EXPLAIN ANALYZE</code> ？
可靠性	ETL 是否幂等（重复运行结果一致）？
	是否有数据质量断言（行数、空值、异常值）？
	是否有重试（尤其是写-写冲突）？
	是否有日志与告警？
并发	是否只有一个写进程？
	长时间写是否释放了锁？
	多进程读取是否使用了只读模式/Parquet？
安全	凭据是否在环境变量/IAM Role 里（而不是硬编码）？
	是否最小权限？
	是否避免在生产用 <code>-unsigned</code> 加载扩展？
运维	DuckDB 版本是否统一（开发与生产一致）？
	扩展是否随版本更新重装？
	备份策略是否有效（EXPORT DATABASE 或对象存储上的 Parquet）？

20.3 常见错误速查表

20.3.1 存储与并发

错误信息	原因	处理
<code>Could not set lock on file "x.db"</code>	另一进程正在写	关闭占用进程；或改用只读；或改用 Parquet
<code>TransactionContext Error: Cannot execute statement of type "CREATE_TABLE" on database which is attached in read-only mode</code>	以只读模式挂在写	用读写模式，或写入另一个库
<code>TransactionContext Error: Conflict on transaction</code>	并发写冲突（乐观并发）	应用层重试
<code>IO Error: Cannot open file "x.db": Permission denied</code>	文件权限	chmod；检查运行用户
<code>Not a DuckDB database file / version mismatch</code>	文件损坏或由更新版本创建	用正确版本打开；从备份恢复
<code>database_size ... could not be found</code>	路径不对	检查路径与 CWD

20.3.2 数据读取

错误信息	处理
<code>Conversion Error: Could not convert string 'x' to INT32</code>	<code>sample_size=-1 / all_varchar=true</code> 后 <code>try_cast</code> / 显式 types
<code>CSV Error: value too long / exceeds maximum line size</code>	调大 <code>max_line_size</code>
<code>too many columns in line</code>	<code>ignore_errors=true</code> 或指定 columns
<code>Invalid Input Error: File not supported / unrecognized compression</code>	检查扩展名与 compression 参数
<code>Binder Error: Referenced table "x.csv" not found</code>	文件名必须加引号：' <code>x.csv</code> '
<code>Unsupported URL scheme https</code>	<code>INSTALL httpsfs; LOAD httpsfs;</code>

错误信息	处理
HTTP 403 Forbidden	secret/权限不对
报 HTTP 400 且提示 region 相关	secret 缺少 REGION，补上即可

20.3.3 SQL 语法

错误信息	处理
Parser Error: syntax error at or near "..."	缺 ;、函数名拼错、或用了别的数据库方言
Binder Error: Ambiguous column "x"	两表同名列，用表别名限定
Binder Error: No function matches the given name	函数名/参数类型不对，查 duckdb_functions()
Conversion Error: Unimplemented type for cast	用 try_cast，或先转 VARCHAR 中转
Binder Error: Referenced column "aliased_col" not found in FROM clause	在 WHERE 里用了 SELECT 别名（不允许）
Invalid Input Error: aggregate function calls cannot be nested	聚合套聚合，拆成两层 SELECT/CTE
Binder Error: window functions are not allowed in ... WHERE	用 QUALIFY 或包一层子查询

20.3.4 内存与资源

错误信息	处理
OutOfMemory Error: failed to allocate data of size ...	减小数据量 / 加 memory_limit / 降 threads / 分批 / COPY 落盘
could not write to temporary directory	temp_directory 不存在/权限不足/空间不足
查询变慢但能跑完	正在 spill，检查 duckdb_temporary_files()
Task was rescheduled ... Force Terminate	资源竞争，降低并发度

20.3.5 Python

问题	处理
<code>TypeError: ... expected 1 arguments, got 0</code>	参数要用 list/dict 传递，不是多个位置参数
UDF 报 packed type mismatch	显式声明 UDF 的参数/返回类型
大表查询把内存打爆	别用 <code>.df()</code> 拉全量；用 SQL 聚合后 <code>.df()</code> 或 <code>COPY TO</code>
DataFrame 的 replacement scan 找不到变量	作用域问题 → 显式 <code>register</code>

20.4 排错的方法论

遇到问题的标准流程：

1. **读完整错误信息。** DuckDB 的错误信息通常非常详尽（包含出错的列/行/类型），不要只看第一行。
2. **最小化重现：**把查询拆成一段段的 CTE，或者用 `LIMIT` 缩小数据量，找出具体出错的片段。
3. **看类型：** `SELECT typeof(col) FROM t LIMIT 1;`
4. **看元数据：** `DESCRIBE t;`、`duckdb_columns()`、`duckdb_functions()`。
5. **看计划：** `EXPLAIN ANALYZE`。
6. **查官方资源：** 官网文档、GitHub Issues（搜索错误信息片段命中率很高）。

一个实用的"调试 SQL"模板：

```
1  -- 1) 这个表的列与类型是什么？
2  DESCRIBE my_table;
3
4  -- 2) 有多少行？空值情况？
5  SELECT count(*),
6          count(*) - count(col_a) AS miss_a,
7          count(*) - count(col_b) AS miss_b
8  FROM my_table;
9
10 -- 3) 看几条典型数据
11 SELECT * FROM my_table USING SAMPLE 5;
12
13 -- 4) 类型确认
14 SELECT typeof(col_a) FROM my_table LIMIT 1;
15
16 -- 5) 缩小范围定位出错行
17 SELECT col_a FROM my_table WHERE try_cast(col_a AS INTEGER) IS NULL
   AND col_a IS NOT NULL;
```

20.5 资源管理建议

20.5.1 给 DuckDB 多少内存

```
1 SET memory_limit = '机器物理内存 * 0.7';
```

- 单机跑 ETL：可以给到 60~80%；
- 应用内嵌分析：给总量的 25~50%，避免影响主服务；
- 容器里：**请显式设置**，默认值会读取宿主机内存而不是容器限制，可能导致被 OOM Killer 干掉。

```
1 # Kubernetes/容器里尤其重要
2 con = duckdb.connect(config={"memory_limit": "4GB", "threads": "4"})
```

20.5.2 threads 的选择

- 离线独占机器：核数即可；
- 共享机器/多容器：限制在 2~4；
- 单条查询已经能吃满核，不需要额外加大。

20.5.3 磁盘

- `temp_directory` 必须空间足够（spill 时的临时文件可能有几百 GB）；
- 最好 SSD，特别是大数据量的排序/连接；
- 生产上给它一个专属目录并监控容量。

20.6 版本管理与升级

事项	建议
锁版本	<code>pip install duckdb==1.5.5</code> ；容器镜像固定 tag
升级前	先在测试环境跑一遍关键 pipeline
文件格式	新版本可读旧文件；旧版本可能读不了新文件
升级后	重装扩展（ <code>INSTALL</code> 会拉取对应版本）
跨版本迁移	用 <code>EXPORT DATABASE</code> / <code>IMPORT DATABASE</code>

CI 里建议加一个“冒烟测试”：运行几个关键 SQL，确认输出行数与校验和。

20.7 一个生产级脚本模板

```

1  #!/usr/bin/env python3
2  """每日 ETL 脚本示例（幂等、有断言、有日志）。"""
3  import logging
4  import os
5  import sys
6  from datetime import date, datetime, timedelta
7
8  import duckdb
9
10 LOG = logging.getLogger("etl")
11 WORK_DB = os.environ.get("WORK_DB", "etl-build.db")
12 OUT_DIR = os.environ.get("OUT_DIR", "out")
13 TARGET = date.today() - timedelta(days=1)
14
15
16 def get_con() -> duckdb.DuckDBPyConnection:
17     return duckdb.connect(WORK_DB, config={
18         "threads": os.environ.get("DUCKDB_THREADS", "4"),
19         "memory_limit": os.environ.get("DUCKDB_MEMORY", "8GB"),
20         "temp_directory": os.environ.get("DUCKDB_TMP",
21     "/tmp/duckdb_tmp"),
22         "preserve_insertion_order": "false",
23         "enable_progress_bar": "false",
24     })
25
26 def run_etl(con, target: date) -> None:
27     con.execute("""
28         CREATE OR REPLACE TABLE staging_events AS
29         SELECT * FROM read_parquet(?, hive_partitioning = true)
30         WHERE dt = ?
31     """, [f"landing/events/**/*.*.parquet", target])
32
33     con.execute("""
34         CREATE OR REPLACE TABLE mart_daily AS
35         SELECT dt, count(*) AS events, count(DISTINCT user_id) AS
uv, sum(amount) AS gmv
36         FROM staging_events GROUP BY 1
37     """)
38
39
40 def assert_quality(con) -> None:
41     checks = {
42         "rows > 0": "SELECT count(*) > 0 FROM mart_daily",
43         "no negative": "SELECT count(*) = 0 FROM mart_daily WHERE
gmv < 0",
44         "uv <= events": "SELECT bool_and(uv <= events) FROM
mart_daily",
45     }
46     for name, sql in checks.items():
47         ok = con.execute(sql).fetchone()[0]
48         if not ok:
49             raise AssertionError(f"数据质量断言失败: {name}")

```

```

50         LOG.info("check ok: %s", name)
51
52
53     def publish(con) -> None:
54         os.makedirs(OUT_DIR, exist_ok=True)
55         con.execute("""
56             COPY (SELECT * FROM mart_daily)
57             TO ? (FORMAT PARQUET, COMPRESSION ZSTD, PARTITION_BY (dt),
OVERWRITE_OR_IGNORE)
58             """, [OUT_DIR])
59         con.execute("CHECKPOINT")
60
61
62     def main() -> int:
63         logging.basicConfig(level=logging.INFO,
64                             format="%(asctime)s %(levelname)s %(
(message)s)")
65         start = datetime.now()
66         if os.path.exists(WORK_DB):
67             os.remove(WORK_DB)          # 幂等：每次从干净状态开始
68
69         con = get_con()
70         try:
71             run_etl(con, TARGET)
72             assert_quality(con)
73             publish(con)
74             LOG.info("完成, 耗时 %.1fs", (datetime.now() -
start).total_seconds())
75             return 0
76         except Exception:
77             LOG.exception("ETL 失败")
78             return 1
79         finally:
80             con.close()                # 释放文件锁
81
82
83     if __name__ == "__main__":
84         sys.exit(main())

```

要点:

- 每次从干净的工作库开始（幂等）；
- 显式配置资源上限；
- 质量断言失败就中止；
- `finally` 里 `close()` 释放锁；
- 日志可观测。

实例演练

实例 20-1 · 每步 ETL 都要有断言

```
1  -- 行数、空值、异常值、业务规则，一次查完
2  SELECT
3      count(*)                                AS total,
4      count(*) - count(order_id)              AS miss_id,
5      count(*) - count(amount)                AS miss_amount,
6      count(*) - count(order_date)            AS miss_date,
7      count(*) FILTER (WHERE amount < 0)      AS negative,
8      min(order_date)                        AS min_date,
9      max(order_date)                        AS max_date
10 FROM sales;
```

期望（在示例数据上的实测）：

total	miss_id	miss_amount	miss_date	negative	min_date	max_date
200000	0	0	0	0	2026-01-01	2026-06-29

把它封装成"失败就中止"的形式：

```
1  SELECT CASE WHEN count(*) = 0 THEN 'OK' ELSE 'FAIL' END AS no_negative
2  FROM stg_sales WHERE amount < 0;
```

实例 20-2 · 复现几个高频错误（附原文）

```
1  -- 1) 只读库写入
2  -- Invalid Input Error: Cannot execute statement of type "CREATE" on
   database
3  -- "examples" which is attached in read-only mode!
4
5  -- 2) 类型不匹配
6  SELECT CAST('abc' AS INTEGER);
7  -- Conversion Error: Could not convert string 'abc' to INT32
8
9  -- 3) 用了不存在的函数
10 SELECT width_bucket(amount, 0, 1000, 10) FROM sales;
11 -- Catalog Error: Scalar Function with name width_bucket does not
   exist!
12 -- Did you mean "time_bucket"?
13
14 -- 4) 语法不支持
15 SELECT try_cast('abc' AS INTEGER DEFAULT 0);
16 -- Parser Error: syntax error at or near "DEFAULT"
17
18 -- 5) 文件不存在
19 SELECT * FROM 'not_here.csv';
20 -- IO Error: Cannot open file "not_here.csv": No such file or
   directory
```

记住：**读完整错误信息**。DuckDB 的错误通常包含出错的列、行、候选函数，信息量很大。

实例 20-3 · 排查一条慢查询的标准动作

```
1  -- 1) 规模
2  SELECT count(*) FROM sales;
3
4  -- 2) 计划
5  EXPLAIN SELECT ...;
6
7  -- 3) 真实耗时与行数
8  EXPLAIN ANALYZE SELECT ...;
9
10 -- 4) 是否发生 spill
11 SELECT * FROM duckdb_temporary_files();
12
13 -- 5) 内存与配置
14 SELECT * FROM duckdb_memory();
15 SELECT name, value FROM duckdb_settings()
16 WHERE name IN ('threads', 'memory_limit', 'temp_directory');
```

实例 20-4 · 生产上的"稳"配置模板

```
1  SET threads                = 4;
2  SET memory_limit           = '8GB';
3  SET temp_directory         = '/tmp/duckdb_tmp';
4  SET max_temp_directory_size = '100GB';
5  SET enable_progress_bar    = false;
6  SET preserve_insertion_order = false;
```

容器里**必须显式设置** `memory_limit`——默认值读的是宿主机内存而不是容器限制，很容易被 OOM Killer 杀掉。

20.8 本章小结

- 三种典型架构：本地分析、单机 ETL 服务、应用内嵌分析；
- 上线 checklist 覆盖数据/性能/可靠性/并发/安全/运维六个方面；
- 常见错误分五类：**并发锁、文件读取、SQL 语法/类型、内存资源、Python API**；
- 排错方法论：读全错误信息 → 最小化重现 → 查类型 → 查计划；
- 容器里**必须显式设置** `memory_limit`；
- 版本要锁、升级要走测试、跨版本用 EXPORT DATABASE；
- 生产脚本的要素：幂等、配置外置、断言、日志、释放锁。

动手练习

1. 挑一个你自己的数据处理流程，用 20.2 的 checklist 逐项检查。
2. 故意触发三种错误（只读模式写、文件路径错、类型不匹配），记录错误信息与解决办法。
3. 把第 13 章的 ETL 改写成 20.7 的脚本模板形式。
4. 在容器里（或限制 `--memory` 的方式）验证 `memory_limit` 设置的效果。

下一章

[附录·速查手册](#) —— 常用函数、配置、类型、错误的快速索引。

示例数据说明

生成数据

在仓库根目录执行（也可以在任意目录，只要路径能对上）：

```
1 | duckdb examples.db < examples/00-seed.sql
```

脚本会：

1. 生成 7 张示例表（全部离线、结果可复现）；
2. 额外导出 `sales.parquet`、`sales.csv`、`events.parquet`、`dirty_sales.csv` 等文件供第 12 章与第 22 章使用；
3. 末尾打印一张自检表，确认每张表的行数。

表清单

表	行数	用途
<code>users</code>	800	用户维表（plan / region / reg_date / credit_score）
<code>sales</code>	200,000	订单事实表（全部案例的主力数据）
<code>events</code>	60,000	网站行为事件（含 JSON 字符串字段 <code>props</code> ）
<code>price_snapshots</code>	7	价格快照，配合 ASOF JOIN 案例
<code>trades</code>	7	成交记录，配合 ASOF JOIN 案例
<code>employees</code>	10	组织架构树，配合递归 CTE 案例
<code>sensor_readings</code>	486	带缺口的时间序列（并注入了 3 个异常高温点），配合补齐/异常检测案例
<code>daily_page</code>	~180	每日聚合小表
<code>basket</code>	~20,000 行 / 10,000 单	购物篮明细，配合关联分析（支持度/置信度/提升度）案例

为什么结果是确定的

脚本用 `hash(i * p + q) % m` 派生伪随机数，而不是 `random()`。

因此：

- 任何人重跑脚本得到的数据**完全一致**；
- 书里各案例给出的输出结构可以与你的实际结果对照；
- 做基准测试时不会因为数据不同而无法比较。

如果你想让数据"更随机", 把 `hash(...)` 换成 `random()`, 并在会话开头加一句 `SELECT setseed(0.42);`。

文件清单

```
1 | examples/
2 | └─ README.md      ← 本文件
3 | └─ 00-seed.sql    ← 数据生成脚本
```

运行第 22 章的案例时, 请先执行上面的生成命令, 然后:

```
1 | duckdb examples.db
```

再逐条粘贴案例中的 SQL。

第 22 章 实战案例集

本章是全书的可运行示例仓库。每个案例都是完整的、端到端的：有场景、有数据、有 SQL、有结果解读、有变体。

先准备数据（本书所有案例都基于它）：

```
1 | duckdb examples.db < examples/00-seed.sql
```

该脚本会生成：`users` (800行)、`sales` (20万行)、`events` (6万行)、`price_snapshots`、`trades`、`employees`、`sensor_readings`、`daily_page`，并额外导出 `sales.parquet`、`sales.csv`、`events.parquet`、`dirty_sales.csv` 供文件读取案例使用。

脚本刻意用 `hash()` 派生伪随机数（而不是 `random()`），所以每次生成的数据完全一致，你看到的输出应当和书中一致。

案例 1 · 销售日报：环比、移动平均、累计

场景：给老板做一张日报，要看 GMV、订单数、客单价、环比、7 日移动平均、累计 GMV。

用到的能力：窗口函数（第 10 章）、条件表达式（第 7 章）、`nullif` 防除零。

```
1 | WITH daily AS (  
2 |     SELECT  
3 |         order_date,  
4 |         sum(amount)           AS gmv,  
5 |         count(*)              AS orders,  
6 |         count(DISTINCT user_id) AS uv  
7 |     FROM sales  
8 |     GROUP BY order_date  
9 | )  
10 | SELECT  
11 |     order_date,  
12 |     gmv,  
13 |     orders,  
14 |     uv,  
15 |     round(gmv / nullif(orders, 0), 2) AS  
16 |     aov,  
17 |     lag(gmv) OVER w AS  
18 |     prev_gmv,  
19 |     round(100.0 * (gmv - lag(gmv) OVER w)  
20 |           / nullif(lag(gmv) OVER w, 0), 2) AS  
21 |     mom_pct,  
22 |     round(avg(gmv) OVER (ORDER BY order_date  
23 |                           ROWS BETWEEN 6 PRECEDING AND CURRENT ROW),  
24 |           2) AS ma7,  
25 |     round(sum(gmv) OVER (ORDER BY order_date  
26 |                           ROWS UNBOUNDED PRECEDING), 2) AS  
27 |     cum_gmv
```

```

23 FROM daily
24 WINDOW w AS (ORDER BY order_date)
25 ORDER BY order_date
26 LIMIT 10;

```

实测输出 (DuckDB 1.5.5, 前 3 行)

order_date	gmv	orders	uv	aov	prev_gmv	mom_pct	ma7	cum_gmv
2026-01-01	560142.74	1112	603	503.73	NULL	NULL	560142.74	560142.74
2026-01-02	561437.64	1112	602	504.89	560142.74	0.23	560790.19	1121580.38
2026-01-03	561609.71	1112	600	505.04	561437.64	0.03	561063.36	1683190.09

解读

- `WINDOW w AS (ORDER BY order_date)` 定义一次、`lag()` 复用两次，避免重复写 `ORDER BY`;
- `nullif(prev, 0)` 防止第一天除零报错;
- `ma7` 用 `ROWS BETWEEN 6 PRECEDING AND CURRENT ROW` (共 7 行)，这是**物理行窗口**；如果想按"过去 7 天" (可能有缺天) 应用 `RANGE BETWEEN INTERVAL 6 DAYS PRECEDING` (第 10.7.5 节) ；
- `cum_gmv` 的 `ROWS UNBOUNDED PRECEDING` 是 `ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW` 的简写。

变体

```

1  -- 只要异常波动的日子：环比超过 ±5%
2  SELECT order_date, gmv, mom_pct
3  FROM (上面的查询)
4  WHERE abs(mom_pct) > 5
5  ORDER BY order_date;

```

案例 2 · 帕累托 / ABC 分类

场景：哪些品类贡献了 80% 的 GMV？

```

1  WITH cat AS (
2      SELECT category, sum(amount) AS gmv
3      FROM sales GROUP BY category
4  ),
5  calc AS (
6      SELECT
7          category,
8          gmv,
9          sum(gmv) OVER (ORDER BY gmv DESC ROWS UNBOUNDED PRECEDING)
10         AS cum_gmv,
11         sum(gmv) OVER ()
12         AS total
13     FROM cat

```

```

12 )
13 SELECT
14     category,
15     round(gmv, 2) AS gmv,
16     round(100.0 * gmv / total, 2) AS pct,
17     round(100.0 * cum_gmv / total, 2) AS cum_pct,
18     CASE
19         WHEN 100.0 * cum_gmv / total <= 80 THEN 'A'
20         WHEN 100.0 * cum_gmv / total <= 95 THEN 'B'
21         ELSE 'C'
22     END AS abc_class
23 FROM calc
24 ORDER BY gmv DESC;

```

实测输出 (DuckDB 1.5.5)

category	gmv	pct	cum_pct	abc_class
服饰	25732140.47	25.18	25.18	A
食品	25571571.94	25.03	50.21	A
电子产品	25470024.52	24.93	75.14	A
图书	25403953.71	24.86	100.00	C

注意最后一行的 `abc_class` 是 **C 而不是 B**：因为它自己的累计占比已经到 100%，超过了 95% 的阈值。这就是 ABC 规则的正确行为——**最后一档永远是 C**，除非你把阈值设得更宽松。

解读：4 个品类份额接近，说明这份种子数据是均匀的；真实数据下你会看到典型的长尾。同样的写法可以套在“用户 Top 20% 贡献多少 GMV”上：

```

1  WITH u AS (SELECT user_id, sum(amount) gmv FROM sales GROUP BY 1),
2  c AS (SELECT user_id, gmv,
3         sum(gmv) OVER (ORDER BY gmv DESC ROWS UNBOUNDED PRECEDING)
4         cum,
5         sum(gmv) OVER () total,
6         count(*) OVER () n FROM u)
7  SELECT round(100.0*min(row_number)/max(n),2) AS top_users_pct,
8         round(100.0*min(cum)/max(total),2) AS gmv_pct
9  FROM (
10     SELECT *, row_number() OVER (ORDER BY gmv DESC) AS row_number FROM
11     c
12     WHERE cum >= 0.8 * total
13 );

```

这个查询回答：“贡献 80% GMV 的用户，占比多少”。是典型的 20/80 验证。

案例 3 · RFM 用户分层

场景：把 800 个用户按 Recency（最近购买距今）、Frequency（购买次数）、Monetary（累计金额）分层，用于精细化运营。

```
1  WITH rfm AS (  
2      SELECT  
3          user_id,  
4          date_diff('day', max(order_date), DATE '2026-09-17') AS  
recency,  
5          count(*) AS  
frequency,  
6          round(sum(amount), 2) AS  
monetary  
7      FROM sales  
8      GROUP BY user_id  
9  ),  
10 scored AS (  
11     SELECT  
12         user_id, recency, frequency, monetary,  
13         ntile(5) OVER (ORDER BY recency DESC) AS r_score,    -- 越近14         ntile(5) OVER (ORDER BY frequency) AS f_score,  
15         ntile(5) OVER (ORDER BY monetary) AS m_score  
16     FROM rfm  
17 )  
18 SELECT  
19     user_id, recency, frequency, monetary, r_score, f_score,  
m_score,  
20     (r_score + f_score + m_score) AS  
rfm_total,  
21     CASE  
22         WHEN r_score >= 4 AND f_score >= 4 AND m_score >= 4 THEN '重  
要价值'  
23         WHEN r_score >= 4 AND f_score <= 2 THEN '新  
客户'  
24         WHEN r_score <= 2 AND f_score >= 4 THEN '重  
要挽留'  
25         WHEN r_score <= 2 AND f_score <= 2 THEN '流  
失'  
26         ELSE '一般'  
27     END AS  
segment  
28 FROM scored  
29 ORDER BY rfm_total DESC, monetary DESC  
30 LIMIT 15;
```

按分层汇总

```

1  WITH ... <同上到 segment> ...
2  SELECT segment,
3         count(*) AS users,
4         round(avg(recency), 1) AS avg_recency,
5         round(avg(frequency), 1) AS avg_freq,
6         round(avg(monetary), 2) AS avg_monetary,
7         round(100.0 * sum(monetary) / sum(sum(monetary)) OVER (), 2)
      AS gmv_share
8  FROM scored_with_segment
9  GROUP BY segment
10 ORDER BY gmv_share DESC;

```

解读

- `ntile(5)` 做五等分；注意 `recency` 是**越小越好**，所以排序要 `DESC` 才能让最近的分到 5；
- 分层规则用 CASE WHEN 表达，一眼可读，也方便业务同学 review；
- 想做"每个分层的人群画像"，再 JOIN `users` 表看 plan / region 分布。

案例 4 · 留存矩阵（N 日留存）

场景：看 2026 年 3 月的新客在第 1/3/7 天是否还回来下单。

```

1  WITH active AS (
2      SELECT DISTINCT user_id, order_date AS d FROM sales
3  ),
4  first_day AS (
5      SELECT user_id, min(d) AS first_d FROM active GROUP BY user_id
6  ),
7  joined AS (
8      SELECT
9          f.first_d,
10         date_diff('day', f.first_d, a.d) AS day_n,
11         a.user_id
12     FROM active a
13     JOIN first_day f USING (user_id)
14     WHERE date_diff('day', f.first_d, a.d) BETWEEN 0 AND 7
15 )
16 SELECT
17     first_d,
18     count(DISTINCT user_id) FILTER (WHERE day_n = 0) AS d0,
19     count(DISTINCT user_id) FILTER (WHERE day_n = 1) AS d1,
20     count(DISTINCT user_id) FILTER (WHERE day_n = 3) AS d3,
21     count(DISTINCT user_id) FILTER (WHERE day_n = 7) AS d7,
22     round(100.0 * count(DISTINCT user_id) FILTER (WHERE day_n = 1)
23           / nullif(count(DISTINCT user_id) FILTER (WHERE day_n =
24 0), 0), 2) AS r1_pct,
24     round(100.0 * count(DISTINCT user_id) FILTER (WHERE day_n = 7)
25           / nullif(count(DISTINCT user_id) FILTER (WHERE day_n =
26 0), 0), 2) AS r7_pct

```

```

26 FROM joined
27 GROUP BY first_d
28 ORDER BY first_d
29 LIMIT 10;

```

解读

- `FILTER` 子句让"一行多指标"变得非常干净（第 8.2.4 节）；
- 分母用 `nullif(..., 0)` 防止某天零新客导致除零；
- 若只要"整体留存曲线"（不按首日分组），去掉 `GROUP BY first_d` 即可。

变体：整体留存率（不看 cohorts）

```

1 WITH active AS (SELECT DISTINCT user_id, order_date d FROM sales),
2 f AS (SELECT user_id, min(d) fd FROM active GROUP BY 1)
3 SELECT date_diff('day', f.fd, a.d) AS day_n,
4        count(DISTINCT a.user_id) AS users,
5        round(100.0*count(DISTINCT a.user_id)
6              / (SELECT count(*) FROM f), 2) AS pct
7 FROM active a JOIN f USING (user_id)
8 WHERE date_diff('day', f.fd, a.d) <= 10
9 GROUP BY 1 ORDER BY 1;

```

案例 5 · 漏斗转化分析

场景：events 表里有 view → cart → pay → complete 四步，算各步转化率和耗时。

```

1 WITH user_steps AS (
2     SELECT
3         user_id,
4         platform,
5         min(ts) FILTER (WHERE event_type = 'view') AS t_view,
6         min(ts) FILTER (WHERE event_type = 'cart') AS t_cart,
7         min(ts) FILTER (WHERE event_type = 'pay') AS t_pay,
8         min(ts) FILTER (WHERE event_type = 'complete') AS t_done
9     FROM events
10    GROUP BY user_id, platform
11 ),
12 ordered AS (
13     SELECT
14         user_id, platform,
15         (t_view IS NOT NULL) AS
16 s1,
17         (t_cart IS NOT NULL AND t_cart >= t_view) AS
18 s2,
19         (t_pay IS NOT NULL AND t_pay >= t_cart
20          AND t_cart >= t_view) AS
21 s3,
22         (t_done IS NOT NULL AND t_done >= t_pay
23          AND t_pay >= t_cart

```

```

21         AND t_cart >= t_view) AS
22     s4
23     FROM user_steps
24     WHERE t_view IS NOT NULL
25 )
26 SELECT
27     count(*) FILTER (WHERE s1) AS step_view,
28     count(*) FILTER (WHERE s2) AS step_cart,
29     count(*) FILTER (WHERE s3) AS step_pay,
30     count(*) FILTER (WHERE s4) AS step_done,
31     round(100.0 * count(*) FILTER (WHERE s2) / nullif(count(*)
32     FILTER (WHERE s1),0), 2) AS cart_rate,
33     round(100.0 * count(*) FILTER (WHERE s3) / nullif(count(*)
34     FILTER (WHERE s2),0), 2) AS pay_rate,
35     round(100.0 * count(*) FILTER (WHERE s4) / nullif(count(*)
36     FILTER (WHERE s3),0), 2) AS done_rate,
37     round(100.0 * count(*) FILTER (WHERE s4) / nullif(count(*)
38     FILTER (WHERE s1),0), 2) AS overall_rate
39 FROM ordered;

```

实测输出 (DuckDB 1.5.5)

step_view	step_cart	step_pay	step_done	cart_rate	pay_rate	done_rate	overall_rate
3199	2200	856	287	68.77	38.91	33.53	8.97

⚠ **一个容易犯的错**: 如果把 s3 只写成 `t_pay >= t_cart`、s4 只写成 `t_done >= t_pay` (不往上追溯), 会出现 `step_done > step_pay` 的怪事——因为 `NULL` 参与比较得到 `NULL`, 某用户没有 cart 事件但有 pay/complete 时, 他会算进 s4 却不算进 s3。我实测时这个版本算出的 `done_rate` 是 **100.86%** (荒谬)。漏斗的每一步都必须级联校验前一步。

分端拆解 (只看 web vs app 的差异)

```

1 SELECT platform,
2     count(*) FILTER (WHERE s1) AS views,
3     round(100.0*count(*) FILTER (WHERE s4)/nullif(count(*) FILTER
4     (WHERE s1),0),2) AS overall_rate
5 FROM ordered
6 GROUP BY platform
7 ORDER BY overall_rate DESC;

```

解读

- `min(ts) FILTER (WHERE ...)` 一次性取出每个用户在各步骤的**首次**时间, 是关键技巧;
- 用 `>= t_prev` 校验顺序, 避免"先下单后浏览"的乱序事件被误算;
- 想要"N 日内完成转化", 把 `t_pay IS NOT NULL` 换成 `t_pay <= t_view + INTERVAL 1 DAY`。

案例 6 · 会话化与行为路径

场景：把同一用户间隔 < 30 分钟的连续行为合并成一次会话，并统计每个会话的访问路径。

```
1 WITH e AS (  
2     SELECT user_id, ts, event_type,  
3           json_extract_string(json(props), '$.page') AS page  
4     FROM events  
5 ),  
6 gaps AS (  
7     SELECT user_id, ts, event_type, page,  
8           lag(ts) OVER w AS prev_ts  
9     FROM e  
10    WINDOW w AS (PARTITION BY user_id ORDER BY ts)  
11 ),  
12 flags AS (  
13     SELECT *,  
14           CASE WHEN prev_ts IS NULL  
15                 OR ts > prev_ts + INTERVAL 30 MINUTE  
16                 THEN 1 ELSE 0 END AS is_new  
17     FROM gaps  
18 ),  
19 labeled AS (  
20     SELECT *,  
21           sum(is_new) OVER (PARTITION BY user_id ORDER BY ts  
22                             ROWS UNBOUNDED PRECEDING) AS session_id  
23     FROM flags  
24 )  
25 SELECT  
26     user_id,  
27     session_id,  
28     min(ts) AS session_start,  
29     max(ts) AS session_end,  
30     date_diff('minute', min(ts), max(ts)) AS duration_min,  
31     count(*) AS events,  
32     string_agg(page, ' → ' ORDER BY ts) AS path,  
33     list(event_type ORDER BY ts) AS event_seq  
34 FROM labeled  
35 GROUP BY user_id, session_id  
36 ORDER BY events DESC, user_id  
37 LIMIT 10;
```

解读

- 三步套路：lag 求间隔 → CASE 打新会话标记 → sum(is_new) OVER (... ROWS UNBOUNDED PRECEDING) 累加得到会话号；
- string_agg(page, ' → ' ORDER BY ts) 直接产出可读路径，list() 保留结构化序列便于后续 TF-IDF/Embedding；
- 会话化后再做漏斗通常比事件级漏斗更准确（排除了跨会话的乱序）。

变体：最常见的三步转移（用 lead 求下一步）

```

1 WITH e AS (SELECT user_id, ts, event_type FROM events),
2 nx AS (SELECT user_id, event_type AS cur,
3         lead(event_type) OVER (PARTITION BY user_id ORDER BY
4         ts) AS nxt
5         FROM e)
6 SELECT cur, nxt, count(*) AS cnt
7 FROM nx
8 WHERE nxt IS NOT NULL
9 GROUP BY 1,2
10 ORDER BY cnt DESC
11 LIMIT 10;

```

案例 7 · ASOF JOIN：成交价格归因

场景：价格只在变动时落快照，想知道每笔成交发生"当时的最新价格"。

```

1 SELECT
2     t.trade_id,
3     t.sym,
4     t.ts AS trade_time,
5     p.price AS price_at_trade,
6     p.ts AS quote_time,
7     t.qty,
8     round(t.qty * p.price, 2) AS notional,
9     date_diff('second', p.ts, t.ts) AS quote_age_sec
10 FROM trades t
11 ASOF JOIN price_snapshots p
12     ON t.sym = p.sym AND t.ts >= p.ts
13 ORDER BY t.sym, t.trade_id;

```

输出结构

trade_id	sym	trade_time	price_at_trade	quote_time	qty	notional	quote_age_sec
1	AAPL	09:45	100.00	09:30	10	1000.00	900
2	AAPL	10:30	101.50	10:00	5	507.50	1800
3	AAPL	12:00	99.80	11:30	7	698.60	1800
4	AAPL	16:00	102.30	15:00	3	306.90	3600
5	MSFT	09:20	NULL	NULL	2	NULL	NULL
6	MSFT	12:30	405.25	12:00	11	4457.75	1800
7	MSFT	15:30	398.75	14:45	4	1595.00	2700

注意第 5 笔：它在 09:20 成交，而 MSFT 的第一个快照是 09:30 —— 早于任何报价，所以普通 ASOF JOIN 会丢掉这一行。

保留所有成交 (LEFT ASOF)

```

1 SELECT t.trade_id, p.price
2 FROM trades t
3 ASOF LEFT JOIN price_snapshots p
4     ON t.sym = p.sym AND t.ts >= p.ts
5 ORDER BY t.trade_id;
6 -- trade_id=5 的 price 为 NULL, 但行仍在

```

解读

- ASOF JOIN 的条件必须是"若干等值 + 恰好一个不等式 ($\geq$ $>$ $\leq$ $<$)";
- ASOF LEFT JOIN 保证左表不丢行, 适合做归因报表;
- 同样套路可用于: 汇率换算、把操作日志关联到"当时的配置版本"、把 IoT 读数对齐到最近的校准记录。

案例 8 · 递归 CTE: 组织架构与薪资汇总

场景: 展示层级路径、层级深度, 并按部门 (某个 VP 之下) 汇总薪资。

```

1 WITH RECURSIVE org AS (
2     -- 锚定: 从 CEO 开始
3     SELECT emp_id, name, mgr_id, title, salary,
4            0 AS level,
5            [name] AS path
6     FROM employees
7     WHERE mgr_id IS NULL
8
9     UNION ALL
10
11     -- 递归: 下挂下级
12     SELECT e.emp_id, e.name, e.mgr_id, e.title, e.salary,
13            o.level + 1,
14            list_append(o.path, e.name)
15     FROM employees e
16     JOIN org o ON e.mgr_id = o.emp_id
17 )
18 SELECT
19     emp_id,
20     level,
21     repeat(' ', level) || name AS tree,
22     title,
23     salary,
24     path
25 FROM org
26 ORDER BY path;

```

输出结构

emp_id	level	tree	title	salary	path
--------	-------	------	-------	--------	------

emp_id	level	tree	title	salary	path
1	0	陈总	CEO	60000	[陈总]
2	1	李VP	VP	40000	[陈总, 李VP]
4	2	张工	Engineer	25000	[陈总, 李VP, 张工]
...					

每个节点及其所有下属的成本

```

1  WITH RECURSIVE subtree AS (
2      SELECT emp_id AS root, emp_id, salary FROM employees
3      UNION ALL
4      SELECT s.root, e.emp_id, e.salary
5      FROM employees e JOIN subtree s ON e.mgr_id = s.emp_id
6  )
7  SELECT root,
8         emp.name,
9         count(*)      AS team_size,
10        sum(s.salary)  AS team_cost    -- ← 必须限定表前缀，否则
11        salary 产生歧义
12  FROM subtree s
13  JOIN employees emp ON emp.emp_id = s.root
14  GROUP BY root, emp.name
15  ORDER BY team_cost DESC;
```

⚠ **实测踩坑：**如果写成 `sum(salary)` 会报

Binder Error: Ambiguous reference to column name "salary" (use: "subtree.salary" or "emp.salary")

—— 因为 `subtree` 与 `employees` 都有这一列。JOIN 里的同名列一定要加前缀。

解读

- 递归必须有一条**收敛路径**（这里组织是棵树，天然收敛）；
- `list_append(path, name)` 携带整条路径，既可以用于打印树，也可以按 path 排序得到“先序遍历”；
- `subtree` 那个写法展示从**每个节点出发**向下聚合，是做“团队成本/团队人数”的标准套路。

案例 9 · JSON 半结构化 → 宽表

场景：`events.props` 是 JSON 字符串，要提取字段、做统计，并把每个用户的行为聚合成一个 JSON 数组交给下游。

```

1  -- 1) 提取字段 (silver 层)
2  CREATE OR REPLACE TABLE silver_events AS
3  SELECT
4      event_id,
```

```

5      user_id,
6      event_type,
7      category,
8      platform,
9      ts,
10     json(props) AS props_json,
11     json_extract_string(json(props), '$.page') AS page,
12     json_extract_string(json(props), '$.ref') AS ref,
13     json_extract_string(json(props), '$.extra.ab') AS ab_bucket,
14     try_cast(json_extract_string(json(props), '$.extra.ms') AS
INTEGER) AS cost_ms
15 FROM events;
16
17 SELECT page, count(*) AS n FROM silver_events GROUP BY page ORDER BY
n DESC LIMIT 5;

```

等价的运算符写法 (`->` / `->>`) :

```

1 SELECT json(props)->>'page' AS page,
2        json(props)->'extra'->>'ab' AS bucket
3 FROM events
4 LIMIT 3;

```

2) 每个用户聚成一个 JSON 对象 (gold 层 / 给 API 用)

```

1 SELECT
2     user_id,
3     count(*) AS events,
4     json_group_object(page, n) AS page_counts,
5     to_json({'buckets': list(DISTINCT ab_bucket),
6         'avg_ms': round(avg(cost_ms), 1)}) AS summary
7 FROM (
8     SELECT user_id, page, ab_bucket, cost_ms, count(*) AS n
9     FROM silver_events
10    GROUP BY user_id, page, ab_bucket, cost_ms
11 )
12 GROUP BY user_id
13 ORDER BY events DESC
14 LIMIT 5;

```

3) 反向: 把 JSON 展开回行

```

1 SELECT key, value
2 FROM json_each({'"region": "华东", "gmv": 1000'});

```

不同版本 `json_each` 返回 STRUCT 还是多列略有差异, 先用 `DESCRIBE SELECT *`
`FROM json_each({'"a": 1'})` 确认列名。

解读

- 入库时用 `::JSON` 存 (而不是 VARCHAR) 可以省去每次 `json()` 解析;

- `json_extract_string` 返回 VARCHAR, `->` 返回 JSON —— 按需要选择;
- 给下游 API 输出时, `json_group_object` / `to_json` 一条 SQL 就完成了"关系 → 文档"的转换。

案例 10 · 脏 CSV 清洗流水线（含断言）

场景: `dirty_sales.csv` 里金额是空串/千分位字符串、日期有两种格式 (ISO 和 MM/DD/YYYY) 。

```

1  -- 1) 先看一眼推断结果
2  SELECT * FROM sniff_csv('dirty_sales.csv');
3
4  -- 2) 全部按字符串读入, 自己掌控转换 (最稳写法)
5  CREATE OR REPLACE TABLE stg_dirty AS
6  SELECT
7      try_cast(order_id AS BIGINT)
8  AS order_id,
9      user_id,
10     region,
11     coalesce(
12         try_cast(replace(amount, ',', '' ) AS DECIMAL(12,2)),
13         0.00
14     )
15 AS amount,
16     coalesce(
17         try_cast(order_date_txt AS DATE),
18         -- ISO: 2026-03-15
19         try_cast(strptime(order_date_txt, '%m/%d/%Y') AS DATE)
20         -- 美式: 03/15/2026
21     )
22 AS order_date
23 FROM read_csv('dirty_sales.csv',
24     delim      = '|',
25     header     = true,
26     all_varchar = true,
27     nullstr    = ['', 'NULL', 'N/A']
28 );

```

3) 数据质量断言 (每步 ETL 都应该有)

```

1 SELECT
2     count(*) AS total,
3     count(*) - count(order_id) AS
miss_id,
4     count(*) - count(amount) AS
miss_amount,
5     count(*) - count(order_date) AS
miss_date,
6     count(*) FILTER (WHERE amount = 0) AS
zero_amount,
7     count(*) FILTER (WHERE amount < 0) AS
negative_amount,
8     min(order_date) AS
min_date,
9     max(order_date) AS
max_date
10 FROM stg_dirty;

```

期望结果

total	miss_id	miss_amount	miss_date	zero_amount	negative_amount	min_date	max_date
200000	0	0	0	28571	0	2026-01-01	2026-06-29

(以上为 DuckDB 1.5.5 上的实测值，可直接对照。)

`zero_amount` 不为 0 是**预料之中**：种子脚本把 `order_id % 7 = 0` 的行金额置为空串，`coalesce(..., 0)` 后就是 0。生产上这类行应该走“待人工校对”分支，而不是静默置 0。

4) 落地为 Parquet

```

1 COPY (SELECT * FROM stg_dirty ORDER BY order_date)
2 TO 'clean_sales.parquet' (FORMAT PARQUET, COMPRESSION ZSTD,
OVERWRITE_OR_IGNORE);

```

解读

- `all_varchar = true` + `try_cast` 是**脏数据的最优解**（第 12.2.2 节）；
- 多格式日期用 `coalesce(try_cast(...), try_cast(strptime(...)))` 逐个尝试；
- 断言放在流式线的每一步，而不是最后。

案例 11 · 时序补齐 + 平滑 + 异常检测

场景： `sensor_readings` 故意缺失了部分小时。要补齐缺失点、做平滑，并找出异常读数。

```

1 -- 1) 生成完整小时序列
2 WITH bounds AS (
3     SELECT min(ts) AS t0, max(ts) AS t1 FROM sensor_readings
4 ),

```

```

5  cal AS (
6      SELECT unnest(generate_series(t0, t1, INTERVAL 1 HOUR)) AS ts
7      FROM bounds
8  ),
9  joined AS (
10     SELECT
11         c.ts,
12         r.temperature,
13         r.humidity
14     FROM cal c
15     LEFT JOIN sensor_readings r USING (ts)
16 )
17 -- 2) 向前填充缺失值 + 平滑 + z-score
18 SELECT
19     ts,
20     coalesce(temperature,
21         last_value(temperature IGNORE NULLS) OVER w)
22         AS temp_filled,
23     round(avg(temperature) OVER (ORDER BY ts ROWS BETWEEN 5
24     PRECEDING AND CURRENT ROW), 2) AS temp_ma6,
25     round((
26         coalesce(temperature, last_value(temperature IGNORE NULLS)
27     OVER w)
28         - avg(temperature) OVER (ORDER BY ts ROWS BETWEEN 23
29     PRECEDING AND 1 PRECEDING)
30         ) / nullif(stddev(temperature) OVER (ORDER BY ts ROWS BETWEEN
31     23 PRECEDING AND 1 PRECEDING), 0)
32         , 2)
33         AS z_score
34 FROM joined
35 WINDOW w AS (ORDER BY ts)
36 QUALIFY abs(z_score) > 3
37 ORDER BY ts;

```

实测输出 (DuckDB 1.5.5) ——正好命中种子数据里注入的 3 个异常点

ts	temp_filled	temp_ma6	z_score
2026-06-05 04:00:00	150.0	79.73	12.50
2026-06-11 10:00:00	150.0	86.87	6.89
2026-06-17 16:00:00	150.0	91.42	9.02

✅ `last_value(x IGNORE NULLS)` 在 1.5.5 上可正常使用（用于向前填充缺失点）。若你的版本不支持，退而求其次用 `coalesce(temperature, avg(temperature) OVER (...))` 或 LATERAL 取最近观测值。

按设备分组的版本（真实场景通常是多台设备）

```

1  WITH guild AS (
2      SELECT device, ts, temperature

```

```

3      FROM sensor_readings
4  )
5  SELECT device, date_trunc('day', ts) AS d,
6         round(avg(temperature), 2) AS avg_temp,
7         round(max(temperature), 2) AS max_temp,
8         round(min(temperature), 2) AS min_temp,
9         count(*) AS samples,
10        count(*) FILTER (WHERE temperature > 70) AS hot_samples
11 FROM guild
12 GROUP BY 1, 2
13 ORDER BY 1, 2
14 LIMIT 10;

```

解读

- `generate_series` + `LEFT JOIN` 是补齐的标准写法；
- 平滑用移动平均窗口；去掉 `QUALIFY` 就能输出完整曲线（便于画图）；
- z-score 用"过去 24 小时"作为基线（`ROWS BETWEEN 23 PRECEDING AND 1 PRECEDING`，排除当前点本身）。

案例 12 · 购物篮关联分析

场景：商品之间有没有"买了 A 就会买 B"的规律？

这里用专门的购物篮表 `basket`（订单-商品明细，1 万笔订单）。种子脚本刻意设计了组合规律：

30% 订单是「面包+牛奶」、20% 是「面包+牛奶+鸡蛋」、20% 是「尿布+啤酒」、20% 是「可乐+薯片」、10% 只买鸡蛋。

```

1  WITH total AS (
2      SELECT count(DISTINCT order_id) AS n FROM basket
3  ),
4  pairs AS (
5      SELECT
6          a.item AS c1,
7          b.item AS c2,
8          count(DISTINCT a.order_id) AS both_orders
9      FROM basket a
10     JOIN basket b
11     ON a.order_id = b.order_id
12     AND a.item < b.item      -- 避免 (A,B) 与 (B,A) 重复、也避免
13                                (A,A)
14     GROUP BY 1, 2
15  ),
16  single AS (
17      SELECT item, count(DISTINCT order_id) AS orders FROM basket
18     GROUP BY 1
19  )
20 SELECT
21     p.c1, p.c2,

```

```

20     p.both_orders,
21     round(100.0 * p.both_orders / t.n, 2)           AS support_pct,
22     round(100.0 * p.both_orders / s1.orders, 2)     AS
conf_c1_to_c2,
23     round(100.0 * p.both_orders / s2.orders, 2)     AS
conf_c2_to_c1,
24     round((1.0 * p.both_orders / t.n)
25           / ((1.0 * s1.orders / t.n)
26              * (1.0 * s2.orders / t.n)), 2)         AS lift
27 FROM pairs p
28 JOIN single s1 ON p.c1 = s1.item
29 JOIN single s2 ON p.c2 = s2.item
30 CROSS JOIN total t
31 ORDER BY lift DESC, both_orders DESC;

```

实测输出 (DuckDB 1.5.5)

c1	c2	both_orders	support_pct	conf_c1_to_c2	conf_c2_to_c1	lift
可乐	薯片	2000	20.00	100.00	100.00	5.00
啤酒	尿布	2000	20.00	100.00	100.00	5.00
牛奶	面包	5000	50.00	100.00	100.00	2.00
牛奶	鸡蛋	2000	20.00	40.00	66.67	1.33
面包	鸡蛋	2000	20.00	40.00	66.67	1.33

结果完美还原了种子数据的设计：可乐/薯片与啤酒/尿布这两个“绑定组合”lift=5；牛奶面包因为还会单独出现在「面包+牛奶+鸡蛋」里，lift 被稀释到 2。

解读

- `a.item < b.item` 是去重 + 排除自配对的经典技巧；
- **support (支持度)**：同时买两者的订单比例；
- **confidence (置信度)**：买了 c1 的订单里有多少也买了 c2；
- **lift (提升度)** = $P(A \cap B) / (P(A) \times P(B))$ ：> 1 正相关，= 1 独立，< 1 替代关系；
- 粒度可以自由切换：把 `order_id` 换成 `user_id` 就是“用户级偏好”，换成 `date` 就是“同日购买”。

⚠ 一个诚实的说明：我最初的示例是基于 `sales`（商品品类字段）按用户做组合的，跑出来的 lift **全部等于 1.00**——因为种子数据里每个用户都随机买过全部 4 个品类，没有任何关联信号。所以数据不一定产生有意义的关联规则：**做关联分析前，先确认你的数据里真的存在可检验的模式**。为此我专门加了 `basket` 表。

案例 13 · Top-N 与贡献度 (QUALIFY)

场景：每个区域 GMV 最高的 2 个品类，并给出它们在区域内的占比。

```
1 SELECT
2     region,
3     category,
4     gmv,
5     round(100.0 * gmv / sum(gmv) OVER (PARTITION BY region), 2) AS
    pct_in_region,
6     rank
7 FROM (
8     SELECT
9         region, category,
10        sum(amount) AS gmv,
11        row_number() OVER (PARTITION BY region ORDER BY sum(amount)
12        DESC) AS rank
13    FROM sales
14    GROUP BY region, category
15 )
16 QUALIFY rank <= 2
17 ORDER BY region, rank;
```

更短的版本（聚合 + 窗口 + 过滤一气呵成）

```
1 SELECT region, category, sum(amount) AS gmv
2 FROM sales
3 GROUP BY region, category
4 QUALIFY row_number() OVER (PARTITION BY region ORDER BY sum(amount)
5     DESC) <= 2
6 ORDER BY region, gmv DESC;
```

每个用户的最大单笔订单

```
1 SELECT user_id, order_id, amount
2 FROM sales
3 QUALIFY row_number() OVER (PARTITION BY user_id ORDER BY amount DESC)
4     = 1
5 ORDER BY amount DESC
6 LIMIT 5;
```

解读：三个写法分别展示“显式子查询 + 别名”、“纯 QUALIFY”、“窗口函数取每组 Top-1”。第二种是日常最省事的写法。

案例 14 · 多源联邦查询（文件 + 表 + 表）

场景：直接查询 Parquet 文件里的销售数据，与数据库里的维表、事件表做联合分析。

```
1 SELECT
```

```

2      u.plan,
3      u.region                                AS user_region,
4      count(DISTINCT s.order_id)              AS orders,
5      round(sum(s.amount), 2)                 AS gmV,
6      round(avg(s.amount), 2)                 AS aov,
7      count(DISTINCT e.user_id)               AS dau_from_events
8 FROM read_parquet('sales.parquet') s
9 JOIN users u USING (user_id)
10 LEFT JOIN events e
11     ON e.user_id = s.user_id
12     AND e.ts::DATE = s.order_date
13 GROUP BY 1, 2
14 ORDER BY gmV DESC;

```

把结果直接写到新的 Parquet (一步到位 ETL)

```

1 COPY (
2     SELECT u.plan, count(DISTINCT s.order_id) AS orders, sum(s.amount)
   AS gmV
3     FROM read_parquet('sales.parquet') s
4     JOIN users u USING (user_id)
5     GROUP BY 1
6 )
7 TO 'plan_summary.parquet' (FORMAT PARQUET, COMPRESSION ZSTD,
  OVERWRITE_OR_IGNORE);

```

再加上跨文件的 JOIN

```

1 SELECT count(*) AS rows_joined
2 FROM read_parquet('sales.parquet') a
3 JOIN read_csv_auto('sales.csv') b USING (order_id);

```

解读：一个 SQL 可以同时引用文件、表、另一个文件——不需要把它们“入库”。这是 DuckDB 最有生产力的地方。

案例 15 · 精确 UV vs 近似 UV

场景：数据量很大时，`count(DISTINCT user_id)` 的内存消耗如何降低。

```

1 SELECT
2     count(*)                                AS rows,
3     count(DISTINCT user_id)                 AS exact_uv,
4     approx_count_distinct(user_id)          AS approx_uv,
5     round(100.0 * (approx_count_distinct(user_id)
6         - count(DISTINCT user_id)) / count(DISTINCT
7     user_id), 2) AS err_pct
8 FROM sales;

```

实测输出 (DuckDB 1.5.5)

rows	exact_uv	approx_uv	err_pct
200000	800	766	-4.25

⚠ 测量结果值得警惕：网上常说 HLL 误差约 2%，但在基数只有 800 的量级上，实测偏差到了 -4.25%。

原因在于 HLL 的相对误差稳定，但绝对值抖动在小基数上占比更大。

结论：

- **小基数 (< 1 万)**：老老实实用 `count(DISTINCT x)`，别为了省那点内存牺牲准确度；
- **大基数 (百万~数十亿)**：`approx_count_distinct` 的价值才真正体现（省下几个 GB 的去重哈希表）；
- 用它之前，先像上面这样测一次 `err_pct`，确认业务能接受。

另一个已实测的事实：DuckDB 的 `approx_count_distinct` 只接受 1 个参数，

`approx_count_distinct(x, 0.01)` 这种写法会报

`Binder Error: No function matches the given name and argument types`。误差精度不可调。

分区维度的大表版本（三种写法对比）

```

1  -- A: 精确，但每个分区都要独立去重
2  SELECT count(DISTINCT user_id) FROM sales GROUP BY region;
3
4  -- B: 先把 (region, user_id) 去重，再数行 -- 有时更快，且能并行
5  SELECT count(*) FROM (SELECT DISTINCT region, user_id FROM sales);
6
7  -- C: 近似，内存占用最低
8  SELECT region, approx_count_distinct(user_id) FROM sales GROUP BY
   region;
```

解读

- HLL 的内存是 $O(\log)$ 级别的，UV 上亿也不怕；
- 如果你的报表可以接受 2% 误差，**永远用** `approx_count_distinct`；
- 误差可通过第三参数调整（视版本支持情况）：`approx_count_distinct(x, 0.01)`。

案例 16 · 一批"日常随手可用"的小 SQL

这一组是每个数据分析同学每天都会用到的短句：

```

1  -- 有多少行？
2  SELECT count(*) FROM sales;
3
4  -- 一眼看懂分布
5  SUMMARIZE sales;
6
```

```

7  -- 看 5 条数据（且不扫全表）
8  SELECT * FROM sales USING SAMPLE 5;
9
10 -- 找 NULL
11 SELECT count(*) FROM sales WHERE amount IS NULL;
12
13 -- 找重复键
14 SELECT user_id, count(*) FROM users GROUP BY 1 HAVING count(*) > 1;
15
16 -- 去重统计（保留最新一条）✅ 正确写法：外层用 WHERE
17 SELECT * FROM (
18     SELECT *, row_number() OVER (PARTITION BY user_id ORDER BY
19     created_at DESC) rn
20     FROM sales
21 ) WHERE rn = 1;
22
23 -- 或者：把窗口函数直接写在 QUALIFY 里
24 SELECT * FROM sales
25 QUALIFY row_number() OVER (PARTITION BY user_id ORDER BY created_at
26 DESC) = 1;
27
28 -- 快速分组-profile
29 SELECT region, count(*) n, round(100.0*count(*)/sum(count(*) OVER
30 (),2) pct
31 FROM sales GROUP BY 1 ORDER BY n DESC;
32
33 -- 日期差、年龄
34 SELECT date_diff('day', reg_date, DATE '2026-09-17') AS
35 days_since_reg FROM users LIMIT 5;
36
37 -- 分位数阶梯
38 SELECT
39     category,
40     quantile_cont(amount, 0.25) AS q25,
41     quantile_cont(amount, 0.50) AS q50,
42     quantile_cont(amount, 0.75) AS q75,
43     quantile_cont(amount, 0.95) AS q95
44 FROM sales
45 GROUP BY 1;
46
47 -- 把结果做成 JSON 给前端
48 SELECT json_group_object(region, gmv) FROM (
49     SELECT region, round(sum(amount),2) AS gmv FROM sales GROUP BY 1
50 );
51
52 -- 一行导出 month skeleton
53 SELECT unnest(generate_series(DATE '2026-01-01', DATE '2026-12-01',
54 INTERVAL 1 MONTH)) AS m;

```

⚠️ 实测踩坑（QUALIFY 的隐藏规则）：下面这样写会报错——

```

1 SELECT * FROM (SELECT *, row_number() OVER (...) rn FROM sales)
  QUALIFY rn = 1;
2 -- Binder Error: at least one window function must appear in the
  SELECT column or QUALIFY clause

```

规则：`QUALIFY` 中使用别名（如 `rn = 1`）的前提是——**同一个查询块的 `SELECT` 列表里必须出现窗口函数**（或 `QUALIFY` 里直接写窗口函数）。上面的外层 `SELECT *` 没有窗口函数，所以被拒绝。

案例 17 · 端到端：从原始文件到发布

把前面所有案例串起来，形成一个可以每天跑的流水线（保存为 `etl_daily.sql`）：

```

1  -- ===== 0. 配置 =====
2  SET memory_limit = '8GB';
3  SET threads      = 4;
4
5  -- ===== 1. 原始数据 =====
6  CREATE OR REPLACE VIEW raw_sales AS
7  SELECT * FROM read_parquet('sales.parquet');
8
9  -- ===== 2. 清洗 =====
10 CREATE OR REPLACE TABLE stg_sales AS
11 SELECT
12     try_cast(order_id AS BIGINT)      AS order_id,
13     trim(user_id)                    AS user_id,
14     coalesce(region, '未知')          AS region,
15     try_cast(amount AS DECIMAL(12,2)) AS amount,
16     try_cast(order_date AS DATE)      AS order_date
17 FROM raw_sales
18 WHERE try_cast(order_date AS DATE) IS NOT NULL;
19
20 -- ===== 3. 维度 =====
21 CREATE OR REPLACE TABLE dim_user AS
22 SELECT
23     user_id,
24     min(order_date) AS first_order,
25     max(order_date) AS last_order,
26     count(*)        AS orders,
27     sum(amount)      AS lifetime_value
28 FROM stg_sales
29 GROUP BY 1;
30
31 -- ===== 4. 事实（含窗口指标） =====
32 CREATE OR REPLACE TABLE fct_daily AS
33 WITH daily AS (
34     SELECT order_date, sum(amount) gmv, count(*) orders,
35           count(DISTINCT user_id) uv
36     FROM stg_sales GROUP BY 1
37 )
38 SELECT

```

```
38     order_date, gmv, orders, uv,
39     lag(gmv) OVER (ORDER BY order_date) AS prev_gmv,
40     avg(gmv) OVER (ORDER BY order_date ROWS BETWEEN 6 PRECEDING AND
CURRENT ROW) AS ma7
41 FROM daily
42 ORDER BY order_date;
43
44 -- ===== 5. 断言 =====
45 SELECT CASE WHEN count(*) > 0 THEN 'OK' ELSE 'FAIL' END AS has_rows
FROM fct_daily;
46 SELECT CASE WHEN count(*) = 0 THEN 'OK' ELSE 'FAIL' END AS no_neg
FROM fct_daily WHERE gmv < 0;
47 SELECT CASE WHEN bool_and(uv <= orders) THEN 'OK' ELSE 'FAIL' END AS
uv_le_orders FROM fct_daily;
48
49 -- ===== 6. 发布 =====
50 -- 注意：目标目录必须已存在，否则报
51 -- IO Error: Cannot open file "publish/fct_daily.parquet": No such
file or directory
52 -- 单文件导出（最简单，目录已存在即可）：
53 COPY (SELECT * FROM fct_daily)
54 TO 'fct_daily.parquet' (FORMAT PARQUET, COMPRESSION ZSTD,
OVERWRITE_OR_IGNORE);
55
56 -- 若要按年分区导出，先在 shell 里建目录：mkdir -p publish/fct_daily
57 -- 再执行：
58 -- COPY (SELECT *, date_part('year', order_date) AS y FROM
fct_daily)
59 -- TO 'publish/fct_daily' (FORMAT PARQUET, PARTITION_BY (y),
OVERWRITE_OR_IGNORE);
60
61 CHECKPOINT;
```

运行：

```
1 | mkdir -p publish/fct_daily      # 分区目录要预先创建
2 | duckdb warehouse.db < etl_daily.sql
```

本章小结

这 17 个案例覆盖了数据分析与数据工程的常见题型：

案例	核心技能点
1 销售日报	窗口函数 lag / 移动平均 / 累计
2 帕累托 ABC	累计占比 + CASE 分档
3 RFM 分层	ntile + 多维度打分
4 留存矩阵	FILTER 多指标 + date_diff

案例	核心技能点
5 漏斗转化	min(ts) FILTER + 顺序校验
6 会话化	lag → 打标 → 累加 + string_agg 路径
7 价格归因	ASOF JOIN
8 组织架构	递归 CTE + list_append
9 JSON 宽表	json_extract / json_group_object
10 脏 CSV	all_varchar + try_cast + 断言
11 时序异常	generate_series 补齐 + z-score
12 购物篮	自连接 + 支持度/置信度/提升度
13 Top-N	QUALIFY
14 联邦查询	文件 + 表混查
15 UV 近似	approx_count_distinct
16 日常短句	高频 SQL 片段
17 端到端 ETL	分层 + 断言 + 发布

动手练习

1. 给案例 1 加上"按 region 分组的日报表" (用 `PARTITION BY region`) 。
2. 把案例 4 的留存改成"周留存" (`date_diff('week', ...)`) 。
3. 在案例 6 里统计"平均会话时长最长的 10 个用户"。
4. 给案例 12 换成"同一品类在同一订单中的组合" (提示: 把 `user_id` 换成其他粒度) 。
5. 把案例 17 改写成 Python 编排的脚本 (参考第 20.7 节的模板) , 加上重试与日志。

返回目录

[回到 README 总目录](#)

附录 · 速查手册

本附录是全书的"口袋工具卡"。当你忘了某个函数叫什么、某个配置怎么写时，来这里快速查找。

A. 元数据查询（忘记结构时先看这些）

```
1 SELECT version();
2 SELECT current_setting('threads'), current_setting('memory_limit');
3
4 SHOW DATABASES;
5 SHOW TABLES;                -- 也可以用 .tables (CLI)
6 SHOW ALL TABLES;
7 DESCRIBE tbl;                 -- 列与类型
8 SHOW CREATE TABLE tbl;
9 SUMMARIZE tbl;                -- 全列统计摘要（探索神器）
10
11 SELECT * FROM duckdb_tables();
12 SELECT * FROM duckdb_columns() WHERE table_name = 'tbl';
13 SELECT * FROM duckdb_views();
14 SELECT * FROM duckdb_schemas();
15 SELECT * FROM duckdb_constraints();
16 SELECT * FROM duckdb_indexes();
17 SELECT * FROM duckdb_types();
18 SELECT * FROM duckdb_functions() WHERE function_name LIKE 'list%';
19 SELECT * FROM duckdb_settings() WHERE name LIKE '%memory%';
20 SELECT * FROM duckdb_extensions();
21 SELECT * FROM duckdb_databases();
22 SELECT * FROM duckdb_secrets();
23 SELECT * FROM duckdb_memory();
24 SELECT * FROM duckdb_temporary_files();
```

B. 数据类型速查

类别	类型
布尔	BOOLEAN / BOOL
整数	TINYINT SMALLINT INTEGER BIGINT HUGEINT
无符整数	UTINYINT USMALLINT UINTEGER UBIGINT UHUGEINT
浮点	FLOAT (REAL) DOUBLE
定点	DECIMAL(p, s) / NUMERIC(p, s)
字符	VARCHAR / STRING / TEXT / CHAR(n) (=VARCHAR)

类别	类型
二进制	BLOB / BYTEA
日期时间	DATE TIME TIMETZ TIMESTAMP TIMESTAMPTZ INTERVAL
其他	UUID ENUM
嵌套	LIST<T> STRUCT<k T> MAP<K,V> UNION<...> ARRAY/T[n]
半结构	JSON VARIANT (1.5+)

转换:

```

1 CAST(x AS T)      / x::T
2 try_cast(x AS T)
3 try_cast(x AS T DEFAULT d)
4 typeof(expr)

```

C. 常用函数

C.1 聚合

```

1 count(*) count(col) count(DISTINCT col)
2 sum avg min max median
3 quantile_cont(x, p) quantile_disc(x, p)
4 stddev variance stddev_pop var_pop skewness kurtosis
5 approx_count_distinct approx_quantile approx_top_k
6 bool_and bool_or every
7 corr covar_pop covar_samp regr_slope regr_intercept regr_r2
8 list(x) list(DISTINCT x) histogram(x)
9 string_agg(x, sep) string_agg(x, sep ORDER BY y)
10 sum(x) FILTER (WHERE cond)      -- 条件聚合

```

C.2 字符串

```

1 length(s) upper(s) lower(s) trim(s) ltrim rtrim
2 lpad(s,n,c) rpad(s,n,c) substr(s, from, len) strpos(s, sub)
3 concat(a,b,...) concat_ws(sep, ...) string_split(s, sep)
4 contains starts_with ends_with replace(s, from, to) reverse(s)
5 regexp_extract(s, pat, grp) regexp_replace regexp_matches
  regexp_split_to_array
6 s ~ 'pattern'      -- 正则匹配
7 s !~ 'pattern'     -- 不匹配
8 s LIKE / ILIKE / SIMILAR TO / GLOB
9 md5(s) sha256(s) encode(b) decode(s) printf(fmt, ...) format(fmt,
  ...)

```

C.3 日期时间

```
1 now() today() current_date current_timestamp
2 date_part(part, ts) --
  year/quarter/month/day/hour/minute/second/dow/isoyear/week/epoch/...
3 date_trunc(part, ts) -- year/quarter/month/week/day/hour/minute
4 strftime(str, fmt) -- 字符串 → 时间
5 strptime(ts, fmt) -- 时间 → 字符串
6 try_strptime(str, fmt)
7 age(a, b) date_diff(part, a, b) epoch(ts) to_timestamp(x)
  last_day(d)
8 d +/- INTERVAL n UNIT -- UNIT:
  YEAR/MONTH/DAY/HOUR/MINUTE/SECOND
9 generate_series(start, stop, step) -- 返回表
10 unnest(generate_series(DATE '2026-01-01', DATE '2026-01-10',
  INTERVAL 1 DAY))
```

常用 strftime 格式符: %Y %y %m %d %H %M %S %f %z %p %A %a %B %b

C.4 条件 / NULL

```
1 CASE WHEN ... THEN ... ELSE ... END
2 iif(cond, a, b)
3 coalesce(a, b, ...)
4 ifnull(a, b) nullif(a, b)
5 try_cast / try_strptime
6 IS NULL / IS NOT NULL / IS DISTINCT FROM
```

C.5 列表 (LIST)

```
1 len(l) list_extract(l, i) l[i] list_slice(l, a, b)
2 list_concat list_contains list_position list_distinct list_sort
  reverse
3 flatten(list_of_lists)
4 list_sum list_avg list_min list_max list_aggregate(l, 'agg')
5 list_transform(l, lambda x: ...) list_filter(l, lambda x: ...)
  list_reduce(l, lambda a, b: ...)
6 unnest(l) list_value(...) [ ... ]
```

C.6 结构体 / MAP / JSON

```

1 {'k': v}                                -- struct 字面量
2 s.field  struct_extract(s,'k')  struct_pack(k := v)
3 MAP{'k': v}                            -- map 字面量
4 map_keys(m)  map_values(m)  element_at(m, k)
5 json('...')  j->'field'  j->>'field'
6 json_extract(j, path)  json_extract_string(j, path)
7 json_valid  json_type  json_keys  json_array_length
8 json_group_array(x)  json_group_object(k, v)
9 to_json(x)  json_object(...)  json_array(...)

```

C.7 窗口函数

```

1 row_number()  rank()  dense_rank()  percent_rank()  cume_dist()
  ntile(n)
2 lag(x, n, default)  lead(x, n, default)
3 first_value(x)  last_value(x)  nth_value(x, n)
4 sum/avg/count/min/max(...) OVER (...)
5
6 OVER (PARTITION BY ... ORDER BY ... [frame])
7 frame: {ROWS|RANGE} BETWEEN start AND end
8 WINDOW w AS (PARTITION BY ... ORDER BY ...)  -- 命名窗口
9 QUALIFY <窗口条件>  -- 窗口结果过滤

```

D. 文件读写

```

1 -- 读
2 FROM 'x.csv';
3 FROM read_csv('x.csv', options);
4 FROM sniff_csv('x.csv');  -- 推断诊断
5 FROM read_csv_auto('x.csv');
6 FROM read_parquet('x.parquet', union_by_name=true,
  hive_partitioning=true, filename=true);
7 FROM read_ndjson('x.ndjson');
8 FROM read_json_auto('x.json');
9 FROM read_xlsx('x.xlsx', sheet='Sheet1');  -- excel 扩展
10 FROM delta_scan('path');
11 FROM read_parquet('s3://bucket/x.parquet');  -- httpfs
12
13 -- Parquet 元数据
14 SELECT * FROM parquet_schema('x.parquet');
15 SELECT * FROM parquet_metadata('x.parquet');
16 SELECT * FROM parquet_file_metadata('x.parquet');
17 SELECT * FROM parquet_key_value_metadata('x.parquet');
18
19 -- 写
20 COPY tbl TO 'out.parquet' (FORMAT PARQUET, COMPRESSION ZSTD,
  ROW_GROUP_SIZE 500000);
21 COPY (SELECT ...) TO 'out.csv' (HEADER, DELIMITER ',');
22 COPY (SELECT ...) TO 'out.ndjson' (FORMAT JSON, ARRAY false);

```

```

23 COPY (SELECT ...) TO 'dir/' (FORMAT PARQUET, PARTITION_BY (col),
    OVERWRITE_OR_IGNORE);

```

E. ATTACH / Secret

```

1  ATTACH 'other.db' AS o;
2  ATTACH 'other.db' AS o (READ_ONLY);
3  ATTACH 'app.db' AS sq (TYPE SQLITE);
4  ATTACH 'host=... dbname=... user=... password=...' AS pg (TYPE
    POSTGRES);
5  ATTACH 'host=... database=...' AS my (TYPE MYSQL);
6  DETACH o;
7  USE o;
8
9  CREATE SECRET s3 (TYPE S3, KEY_ID '...', SECRET '...', REGION
    '...');
10 CREATE SECRET s3p (TYPE S3, ..., ENDPOINT
    'xxx.r2.cloudflarestorage.com', URL_STYLE 'path');
11 CREATE SECRET az (TYPE AZURE, PROVIDER CREDENTIAL_CHAIN);
12 CREATE SECRET gcs (TYPE GCS, KEY_ID '...', SECRET '...');
13 CREATE SECRET api (TYPE HTTP, EXTRA_HTTP_HEADERS
    MAP{'Authorization':'Bearer ...'});
14 CREATE PERSISTENT SECRET s3 (TYPE S3, ...);
15 DROP SECRET s3;

```

F. 配置与 PRAGMA

```

1  SET threads = 8;
2  SET memory_limit = '12GB';
3  SET temp_directory = '/mnt/fast/duckdb_tmp';
4  SET max_temp_directory_size = '300GB';
5  SET enable_progress_bar = false;
6  SET preserve_insertion_order = false;
7  SET checkpoint_threshold = '64MiB';
8  SET TimeZone = 'UTC';
9  SET enable_object_cache = true;
10 SET autoinstall_known_extensions = true;
11 SET autoloading_known_extensions = true;
12 SET disabled_optimizers = 'filter_pushdown';
13
14 PRAGMA database_size;
15 PRAGMA storage_info;
16 PRAGMA table_info('tbl');
17 PRAGMA enable_profiling = 'json';
18 SET profiling_output = '/tmp/plan.json';
19 SET profiling_mode = 'detailed';
20 PRAGMA disable_profiling;

```

常用 CLI 点命令

```
1 .help          .tables          .schema [t]      .databases
2 .open [db]     .mode MODE       .headers on/off  .separator 'x'
3 .output FILE   .once FILE       .read FILE       .timer on
4 .changes on    .maxrows N       .limit N         .dump
5 .extensions    .version         .system CMD      .cd DIR
6 .exit / .quit
```

常用命令行参数

```
1 duckdb [DB] -c "SQL" -json -readonly -init FILE -csv -unsigned
2 echo "SELECT 1" | duckdb my.db
3 duckdb my.db < script.sql
4 duckdb --version
```

G. Python API 速查

```
1 import duckdb
2
3 # 连接
4 con = duckdb.connect("x.db")                # 持久化
5 con = duckdb.connect(":memory:")
6 con = duckdb.connect("x.db", read_only=True)
7 con = duckdb.connect(config={"threads": "8", "memory_limit":
8     "12GB"})
9
10 # 执行
11 rel = con.sql("SELECT ...")
12 rel.show()                                # 漂亮打印
13 rel.df()                                  # pandas
14 rel.pl()                                  # polars
15 rel.arrow()                               # pyarrow Table
16 rel.fetchall()                            # list[tuple]
17 rel.fetchone()
18 rel.describe()                            # 统计摘要
19 rel.explain()                             # 执行计划
20 rel.sql_query()                           # 打印生成的 SQL
21
22 # 关系式 API
23 (con.table("sales")
24     .filter("amount > 100")
25     .project("region, amount")
26     .aggregate("region, sum(amount) AS gmv")
27     .order("gmv DESC")
28     .limit(10))
29
30 # DataFrame 互操作
31 duckdb.sql("SELECT count(*) FROM df")      # replacement scan
```

```

31 con.register("v", df); con.unregister("v")
32 con.execute("CREATE TABLE t AS SELECT * FROM df")
33
34 # 参数绑定
35 con.execute("SELECT * FROM t WHERE k = ?", [k])
36 con.executemany("INSERT INTO t VALUES (?, ?)", rows)
37
38 # 批量写入
39 appender = con.appender("t")
40 appender.append_row(...); appender.flush(); appender.close()
41
42 # 事务
43 con.execute("BEGIN"); ...; con.execute("COMMIT") # 或
    con.begin()/con.commit()
44
45 # UDF
46 con.create_function("f", fn, [duckdb.typing.BIGINT],
    duckdb.typing.BIGINT,
47                        null_handling="special",
    exception_handling="return_null")

```

H. SQL 片段库（可直接复制）

```

1  -- 去重取最新一条
2  SELECT * FROM (
3      SELECT *, row_number() OVER (PARTITION BY key ORDER BY updated_at
4      ) QUALIFY rn = 1;
5
6  -- 每组 Top-N
7  SELECT ... FROM t QUALIFY row_number() OVER (PARTITION BY g ORDER BY
8      v DESC) <= N;
9
10 -- 同环比
11 SELECT m, v, lag(v,1) OVER w AS pv, lag(v,12) OVER w AS ly FROM t
12 WINDOW w AS (ORDER BY m);
13
14 -- 7 日移动平均
15 SELECT d, avg(v) OVER (ORDER BY d ROWS BETWEEN 6 PRECEDING AND
16     CURRENT ROW) AS ma7 FROM t;
17
18 -- 会话化
19 SELECT ..., sum(is_new) OVER (PARTITION BY user ORDER BY ts ROWS
20     UNBOUNDED PRECEDING) AS sid ...
21
22 -- 补齐日期（缺零）
23 WITH cal AS (SELECT unnest(generate_series(DATE '2026-01-01', DATE
24     '2026-01-31', INTERVAL 1 DAY)) AS d)
25 SELECT cal.d, coalesce(t.v, 0) AS v FROM cal LEFT JOIN t ON cal.d =
26     t.d ORDER BY 1;

```

```

22 -- 行转列：手工 PIVOT（用 FILTER，兼容性最好）
23 SELECT k,
24         sum(v) FILTER (WHERE cat='A') AS A,
25         sum(v) FILTER (WHERE cat='B') AS B
26 FROM t GROUP BY k;
27
28 -- 累计占比（帕累托）
29 SELECT k, v, sum(v) OVER (ORDER BY v DESC ROWS UNBOUNDED PRECEDING)
30 / sum(v) OVER () AS cum_pct
31 FROM agg;
32
33 -- 缺失率统计
34 SELECT count(*) AS n,
35         count(*) - count(c1) AS miss_c1,
36         count(*) - count(c2) AS miss_c2
37 FROM t;
38
39 -- 分层小计
40 SELECT region, category, sum(v) FROM t GROUP BY ROLLUP (region,
41 category);
42
43 -- 安全插入/更新
44 INSERT OR REPLACE INTO t BY NAME SELECT * FROM src;

```

I. 性能检查清单（一页版）

- 1 [] 只 SELECT 需要的列 （列裁剪）
- 2 [] WHERE 用范围，别把列包在函数里（Zone Map）
- 3 [] 先聚合/过滤，再 JOIN
- 4 [] JOIN 的键唯一且类型一致（最好整数）
- 5 [] 用 EXISTS/SEMI/ANTI 而不是 JOIN+DISTINCT
- 6 [] ORDER BY 配 LIMIT
- 7 [] 重复读的大文件 → 转 Parquet
- 8 [] 数据按常用过滤列排序写入
- 9 [] 分区列要低基数
- 10 [] row group 10万~100万行
- 11 [] memory_limit 与 threads 已设置
- 12 [] temp_directory 在大空间 SSD 上
- 13 [] 不用循环 INSERT（用 Appender / INSERT SELECT）
- 14 [] 慢查询看过 EXPLAIN ANALYZE
- 15 [] 检查是否 spill（duckdb_temporary_files）

J. 关键字（避免用作列名）

```

1 SELECT, FROM, WHERE, GROUP, BY, HAVING, ORDER, LIMIT, OFFSET, JOIN,
  ON, USING,
2 AS, DISTINCT, ALL, CASE, WHEN, THEN, ELSE, END, AND, OR, NOT, NULL,
  TRUE, FALSE,
3 IN, EXISTS, BETWEEN, LIKE, ILIKE, IS, ASC, DESC, WITH, RECURSIVE,
  UNION, INTERSECT,
4 EXCEPT, INSERT, INTO, VALUES, UPDATE, SET, DELETE, CREATE, DROP,
  ALTER, TABLE, VIEW,
5 PRIMARY, KEY, FOREIGN, REFERENCES, CHECK, UNIQUE, DEFAULT, INDEX,
  SCHEMA, MACRO,
6 INSTALL, LOAD, ATTACH, DETACH, COPY, EXPORT, IMPORT, QUALIFY, WINDOW,
  PARTITION,
7 OVER, FILTER, PIVOT, UNPIVOT, TYPE, ENUM, INTERVAL, DATE, TIME,
  TIMESTAMP, RETURNS

```

如果必须用关键字做列名，请用双引号："date"。

K. 实测确认的"方言差异" (DuckDB 1.5.5)

下面这些写法在部分其他数据库能用，但 **DuckDB 不支持 / 行为不同**，均已在本机实测：

你可能会写的	DuckDB 的实际情况	正确写法
<code>try_cast(x AS INT DEFAULT 0)</code>	✗ Parser Error	<code>coalesce(try_cast(x AS INT), 0)</code>
<code>width_bucket(x, lo, hi, n)</code>	✗ 函数不存在	<code>least(floor(x/step)::INT, n-1)</code> ；等频用 <code>ntile(n)</code>
<code>approx_count_distinct(x, 0.01)</code>	✗ 只接受 1 个参数	<code>approx_count_distinct(x)</code> (精度不可调)
子查询外 <code>QUALIFY rn = 1</code> (SELECT 无窗口函数)	✗ Binder Error	用 <code>WHERE rn = 1</code> 或把窗口函数写进 <code>QUALIFY</code>
<code>DATE '2026-01-01' + 3</code>	✗ 不支持 <code>DATE + 整数</code>	<code>DATE '2026-01-01' + INTERVAL 3 DAY</code>
<code>lpad(整数, 4, '0')</code>	✗ 需要 VARCHAR	<code>lpad(x::VARCHAR, 4, '0')</code>
<code>COPY ... TO '不存在的目录/x.parquet'</code>	✗ IO Error	先用 shell 建目录 (<code>mkdir -p</code>)
<code>sum(salary)</code> 在两张都有该列的表 JOIN 后	✗ Ambiguous reference	加表前缀： <code>sum(t.salary)</code>

你可能会写的	DuckDB 的实际情况	正确写法
<code>x -> x * 2</code> (旧 lambda)	⚠️ 可用但打印弃用警告	<code>lambda x: x * 2</code>
<code>FROM t WHERE ... SELECT ...</code>	❌ Parser Error	<code>FROM t SELECT ... WHERE ...</code> (SELECT 必须紧跟 FROM)
<code>INSERT INTO tgt BY NAME SELECT * FROM src</code> (src 多列)	❌ Binder Error	只选目标表有的列: <code>SELECT</code> 需要的列
<code>SELECT access_mode FROM duckdb_databases()</code>	❌ 无此列	该视图列: <code>database_oid / database_name / tags / cipher</code>
<code>read_parquet('*.parquet')</code> 混合不同 schema	⚠️ 只按第一个文件对齐	加 <code>union_by_name = true</code>

两个“看起来反直觉但正确”的行为（已实测）：

```

1 | SELECT INTERVAL 1 MONTH = INTERVAL 30 DAY;    -- true（比较时按绝对时长折
   | 算）
2 | SELECT DATE '2026-02-01' + INTERVAL 1 MONTH; -- 2026-03-01（加法走日历语
   | 义）
3 | SELECT DATE '2026-02-01' + INTERVAL 30 DAY;  -- 2026-03-03（加法走天数语
   | 义）

```

```

1 | -- 只读库写入的报错类型名是 CREATE，不是 CREATE_TABLE
2 | Invalid Input Error: Cannot execute statement of type "CREATE" on
   | database "x"
3 | which is attached in read-only mode!

```

其他实测确认可用的写法：

- ✅ `last_value(x IGNORE NULLS)` 向前填充
- ✅ `histogram(col)` 返回 MAP
- ✅ `SELECT ... GROUP BY ... QUALIFY row_number() OVER (...) <= n`
- ✅ 聚合窗口里直接引用聚合表达式: `QUALIFY row_number() OVER (ORDER BY sum(amount) DESC) = 1`
- ✅ `EXCLUDE / REPLACE / COLUMNS(lambda)、USING SAMPLE、INSERT BY NAME`

L. 学习资源

- 官方文档: <https://duckdb.org/docs/>

- GitHub Issues (搜错误信息) : <https://github.com/duckdb/duckdb/issues>
 - 扩展列表: <https://duckdb.org/docs/extensions/>
 - 社区扩展: https://duckdb.org/community_extensions/
 - 本书示例代码所在的版本: DuckDB 1.5.x
-

结语

到这里, 这本教程的全部内容就结束了。回顾一下你收获的能力:

1. 能在任何机器上装好 DuckDB 并跑通查询;
2. 能写出简洁、高效、"DuckDB 风格"的 SQL (第 4、7、8、9、10 章);
3. 能把 CSV/Parquet/JSON/Excel/数据库/云存储里的数据, 随意地组合查询 (第 12、14 章);
4. 能把结果落地成可靠的数据产品 (第 13 章), 并与 Python 生态无缝协作 (第 15 章);
5. 能预判和诊断性能问题 (第 16、17 章);
6. 知道它的边界在哪里, 以及如何在生产环境安全使用 (第 18、20 章)。

剩下的一件事: **把它用起来。**

找一份你每天都在处理的脏数据, 用一句 DuckDB SQL 替代原来的脚本。你会立刻理解为什么这本书存在。