

DuckDB 数据库从入门到进阶

基于 DuckDB v1.5.5 官方文档的系统性实战指南

WorkBuddy 文档生成团队

2026 年 8 月

目录

Table of Contents	6
前言	6
第一部分 基础入门	7
第 1 章 DuckDB 概述与核心思想	7
1.1 什么是 DuckDB：进程内（in-process）分析型数据库	7
1.2 设计哲学与适用场景	8
1.3 与 SQLite、ClickHouse、PostgreSQL 等系统的定位对比	8
1.4 列式存储与向量化执行的直观理解	9
1.5 本书路线图与阅读建议	9
1.6 本章小结	10
第 2 章 安装、编译与连接	10
2.1 安装方式总览	10
2.2 从源码构建	11
2.3 CLI 客户端详解	12
2.4 连接模型概述	12
2.5 第一条查询与安装验证	13
2.6 本章小结	13
第 3 章 SQL 基础：数据类型与方言	14
3.1 数值类型与布尔	14
3.1.1 定宽整数	14
3.1.2 定点小数 DECIMAL	15
3.1.3 浮点与布尔	15
3.2 字符串、BLOB 与日期时间	15
3.2.1 字符串与二进制	15
3.2.2 日期时间家族	15
3.2.3 1-based 索引	16
3.3 嵌套与复杂类型	16
3.4 类型转换与 NULL 三态逻辑	17
3.4.1 显式转换	17
3.4.2 NULL 三态逻辑	17
3.4.3 隐式转换的收紧	17
3.5 友好 SQL 与方言特性	17
3.6 本章小结	18

第 4 章 SQL 基础：表的定义与数据写入	18
4.1 CREATE TABLE 与约束	18
4.2 临时表、生成列与默认值	19
4.3 INSERT / INSERT INTO SELECT / 批量写入	19
4.4 UPDATE / DELETE / MERGE	20
4.5 视图、宏、模式与序列	21
4.6 事务与并发模型	21
4.7 本章小结	22
第 5 章 SQL 基础：查询语言	22
5.1 SELECT 与投影	22
5.2 WHERE、过滤与排序	23
5.3 聚合与分组	24
5.4 多表连接	25
5.5 集合运算	26
5.6 窗口函数	26
5.7 CTE 与递归查询	27
5.8 采样、VALUES、PIVOT/UNPIVOT	28
5.9 本章小结	29
第 6 章 SQL 进阶：函数与表达式	29
6.1 表达式与运算符	29
6.2 聚合函数	31
6.3 字符串与正则函数	31
6.4 日期时间函数	32
6.5 数组/LIST、STRUCT、MAP 操作函数	32
6.6 JSON 函数	33
6.7 条件、哈希、系统函数与元数据	33
6.8 本章小结	34
第二部分 数据工程	34
第 7 章 数据导入：文件直读	34
7.1 “直接查询文件”哲学与格式自动探测	34
7.2 CSV 读取	35
7.2.1 自动检测与采样	35
7.2.2 读取参数	36
7.2.3 读取损坏的 CSV	37
7.3 Parquet 读取与下推	37
7.3.1 读取与下推优化	37
7.3.2 用 schema 重定义读取结构	38
7.3.3 元数据查询	38
7.3.4 加密	38
7.4 JSON 读取	39
7.4.1 读取函数与格式	39
7.4.2 关键读取参数	39
7.5 多文件与 glob、分区数据集	40
7.5.1 Glob 通配符	40
7.5.2 按列名合并：union_by_name	41
7.5.3 Hive 分区与分区键下推	41
7.6 本章小结	41
第 8 章 数据导出与持久化	42
8.1 COPY ...TO：把数据写出到文件	42
8.1.1 CSV 写出与压缩	42

8.1.2 Parquet 写出、自定义元数据与压缩	42
8.1.3 JSON 写出	43
8.1.4 分区写出 PARTITION_BY	43
8.1.5 PER_THREAD_OUTPUT 与统计返回	43
8.1.6 关键限制：COPY 不做表到表复制	43
8.2 持久化数据库文件与 ATTACH	43
8.2.1 ATTACH 多库挂载	43
8.2.2 storage_compatibility_version 与存储格式版本	44
8.3 Checkpoint、WAL 与空间回收	44
8.3.1 WAL 与 Checkpoint 机制	44
8.3.2 关键误区：VACUUM 不回收空间	44
8.4 数据库迁移	45
8.5 本章小结	45
第三部分 客户端与生态集成	45
第 9 章 Python 客户端深入	45
9.1 安装与连接	46
9.1.1 安装	46
9.1.2 建立连接	46
9.1.3 线程与并发	46
9.2 关系 API	47
9.2.1 什么是 Relation	47
9.2.2 从数据源创建 Relation	47
9.2.3 链式转换与物化	47
9.2.4 一个性能警示	48
9.3 DB-API 与参数化查询	48
9.3.1 标准 DB-API 用法	48
9.3.2 参数化查询与防注入	48
9.4 与 pandas / PyArrow / Polars / NumPy 互操作	49
9.4.1 replacement scan：把 DataFrame 当表查	49
9.4.2 从查询结果回到 DataFrame	49
9.4.3 把 DataFrame 写回 DuckDB	50
9.4.4 与 Arrow 的零拷贝	50
9.5 Appender 批量写入与性能	50
9.5.1 为什么不用逐行 INSERT	50
9.5.2 使用 Appender	50
9.6 Jupyter 与 SQLAlchemy	51
9.6.1 在 Jupyter 中使用 DuckDB	51
9.6.2 通过 SQLAlchemy 连接 (duckdb-engine)	52
9.6.3 marimo：响应式笔记本	52
9.7 本章小结	52
第 10 章 其他客户端	53
10.1 CLI 进阶	53
10.1.1 输出格式：.mode 的 csv / json / markdown	53
10.1.2 批量脚本：.read 与重定向	54
10.2 C / C++ API	54
10.2.1 C API (libduckdb)	54
10.2.2 C++ API (内部接口，不稳定)	55
10.3 Node.js、ODBC、Wasm	55
10.3.1 Node.js (@duckdb/node-api)	55
10.3.2 ODBC	56
10.3.3 DuckDB-Wasm	56

10.4 第三方客户端	57
10.4.1 Primary 层级的语言绑定	57
10.4.2 Tertiary 层级与社区维护的绑定	57
10.5 本章小结	57
第 11 章 与外部数据库集成	58
11.1 ATTACH 机制与数据库集成模式	58
11.2 PostgreSQL 集成	59
RDS 与 Aurora 的 IAM 认证	59
v1.5.5 schema 变化自动检测	59
11.3 SQLite 集成	60
11.4 MySQL 集成	60
11.5 本章小结	60
第 12 章 云存储与远程数据	61
12.1 https 扩展与对象存储	61
12.1.1 统一接入模型	61
12.1.2 支持的对象存储后端	61
12.1.3 远程 DuckDB 库的只读挂载	62
12.2 CREATE SECRET 与凭证管理	62
12.2.1 Secret 类型与典型参数	62
12.2.2 持久化存储与权限	62
12.3 远程文件读取性能建议	63
12.3.1 网络延迟与并行度	63
12.3.2 大查询的落盘 (spill to disk)	63
12.3.3 下推减少传输	63
12.4 Quack 远程协议	63
12.4.1 协议模型	63
12.4.2 服务端与客户端的启动	63
12.4.3 认证、授权与安全部署	64
12.4.4 版本状态	64
12.5 本章小结	64
第四部分 扩展机制	65
第 13 章 扩展体系	65
13.1 扩展架构与加载模型	65
13.2 版本兼容性铁律与离线安装	66
13.3 扩展签名与安全	66
13.4 核心扩展一览	67
13.5 社区扩展生态	68
13.6 本章小结	68
第 14 章 关键扩展详解	69
14.1 JSON 扩展	69
14.2 Spatial	70
14.3 Iceberg 扩展	71
14.4 其他常用扩展	72
14.5 本章小结	73
第五部分 性能与运维	73
第 15 章 配置与性能调优	73
15.1 配置机制	73
15.2 内存与线程配置	74

15.3 性能最佳实践	75
15.4 执行计划分析	75
15.5 常见性能陷阱与排查	76
15.6 本章小结	76
第 16 章 运维手册	77
16.1 安装与版本策略	77
16.2 DuckDB 的足迹	78
16.3 日志	79
16.4 安全加固	80
16.5 本章小结	81
第六部分 原理剖析	81
第 17 章 DuckDB 内部原理	81
17.1 查询处理流水线	81
17.1.1 Parser：只做词法拆解，不感知语义	81
17.1.2 Binder：借助 catalog 完成语义绑定	82
17.1.3 Logical Planner 与 Optimizer	82
17.1.4 Column Binding Resolver：从“列名”到“下标”	82
17.1.5 Physical Plan Generator 与 Execution	82
17.2 向量化执行引擎	83
17.2.1 Volcano 模型的问题	83
17.2.2 DuckDB 的推送式向量化模型	83
17.2.3 DataChunk 与 Vector	83
17.2.4 向量格式	83
17.2.5 复杂类型的内存布局	83
17.3 列式存储与压缩	84
17.3.1 Row Group 与列式布局	84
17.3.2 轻量压缩算法族	84
17.3.3 存储文件头与版本兼容	84
17.3.4 磁盘占用经验值	84
17.4 查询优化器	84
17.4.1 Expression Rewriter	84
17.4.2 Filter Pushdown（谓词下推）	85
17.4.3 Join Order Optimizer	85
17.4.4 Common Sub Expressions	85
17.4.5 In Clause Rewriter	85
17.5 事务与 MVCC、乐观并发	85
17.5.1 进程内模式下的两种并发选项	85
17.5.2 单进程内的写并发规则	85
17.5.3 多进程写与文件锁	85
17.5.4 隔离级别	85
17.6 ART 索引	86
17.6.1 适用场景与单列限制	86
17.6.2 VACUUM 行为差异与实现参考	86
17.7 本章小结	86
第七部分 实战与总结	86
第 18 章 应用场景实践	86
18.1 本地数据分析与探索式查询	87
18.1.1 CLI 随手查文件	87
18.1.2 Jupyter 中的交互分析	87
18.1.3 零成本试错的工作流	88

18.2 作为 SQLite 的 OLAP 补充	88
18.2.1 分工：事务与分析各司其职	88
18.2.2 通过 sqlite 扩展桥接	88
18.2.3 一次桥接查询示例	89
18.3 轻量级数据管线	89
18.3.1 多源读取与内存计算	89
18.3.2 写出 Parquet 或持久库	90
18.3.3 调度中的批处理脚本	90
18.4 嵌入式分析中台与应用	91
18.4.1 进程内就地分析	91
18.4.2 轻量分析中台的实现	91
18.4.3 并发与连接管理	91
18.5 与数据科学生态协同	92
18.5.1 replacement scan 无缝互操作	92
18.5.2 把 DuckDB 当查询引擎驱动 DataFrame	92
18.5.3 与 Polars / Arrow / Ibis 衔接	93
18.6 本章小结	93
第 19 章 总结与展望	94
19.1 核心能力回顾	94
19.1.1 进程内、列式、向量化的 OLAP 引擎	94
19.1.2 单文件数据库与两种存储形态	94
19.1.3 丰富而现代的 SQL 能力	94
19.1.4 可插拔的扩展体系	95
19.1.5 与数据科学生态的无缝互操作	95
19.1.6 ACID 事务与 MVCC 并发控制	95
19.2 适用与不适用场景研判	95
19.2.1 典型适用场景	96
19.2.2 边界与不适用地带	96
19.3 生态与未来	96
19.3.1 DuckLake 与湖仓之路	96
19.3.2 Quack 远程协议：从进程内走向进程外	96
19.3.3 社区扩展与开源治理	97
19.3.4 路线图方向与持续学习建议	97
19.4 本章小结	97
19.5 全书收束	97
版权与免责声明	98

Table of Contents

前言

DuckDB 正在重新定义“分析型数据库”的部署形态。它不走“起一个服务、连一个集群”的老路，而是把自己做成了一个进程内的嵌入式引擎——一条 `pip install duckdb` 或一个二进制文件，就能在笔记本、数据管道、边缘设备上直接跑起列式存储与向量化执行的分析查询。当数据规模还没到需要分布式系统的程度、却又远超电子表格所能承受的体量时，DuckDB 恰好卡在那个最舒服的区间：轻量、零运维、对文件与 Arrow 生态天生友好。

本书的由来，是想给中文世界的数据库学习者与从业者一本“既能入门、又能进阶”的 DuckDB 专著。它的内容完全建立在 DuckDB 官方文档的基础之上——我们通读了 v1.5.5 版本的整套文档，并按“由浅入深、面面俱到”的原则重新编排了章节结构。你读到的每一处语法、每一个参数、每一项版本事实，都尽量回溯

到官方文档的对应出处；书中也如实标注了哪些能力属于哪个版本（例如 Quack 远程协议于 2026 年 5 月 12 日随 v1.5.3 进入 beta，DuckLake 1.0 于 2026 年 4 月发布，GEOMETRY 类型在 v1.5 成为内置类型）。

全书分为七个部分、共十九章：

- **第一部分 基础入门（第 1–6 章）**：从 DuckDB 是什么、怎么装、核心数据类型与 SQL 方言，到建表写数、查询语言与函数表达式，为后续打下地基。
- **第二部分 数据工程（第 7–8 章）**：文件直读与数据导出/持久化，这是 DuckDB“以文件为第一公民”理念最直接的体现。
- **第三部分 客户端与生态集成（第 9–12 章）**：深入 Python 客户端，概览其余语言绑定，讲解与外部数据库、云存储及远程协议（Quack）的集成。
- **第四部分 扩展机制（第 13–14 章）**：扩展的安装/加载/版本绑定，以及 httpfs、postgres、iceberg、spatial、ducklake 等关键扩展的实战。
- **第五部分 性能与运维（第 15–16 章）**：配置项调优与运维手册，覆盖生产落地的关键考量。
- **第六部分 原理剖析（第 17 章）**：从解析、优化到向量化执行与存储格式，揭开引擎内部。
- **第七部分 实战与总结（第 18–19 章）**：应用场景实践，以及对 DuckDB 演进方向的展望。

阅读建议：完全的初学者建议从第 1 章顺序读到第 6 章，把基础打牢；有一定 SQL 或数据库经验的读者，可以跳过前几章的入门铺垫，直接从数据工程或客户端集成部分切入；希望理解引擎“为什么快”的读者，第六部分值得精读。每章末尾都配有“本章小结”，便于复习与速查。

书中的代码示例以“可运行、可复现”为第一原则，默认基于 DuckDB v1.5.5；涉及版本差异处均显式标注。建议读者在自己的环境中动手敲一遍代码——DuckDB 的“免服务、单文件”特性，让这种即时实验的成本几乎为零。

最后说明：本书由 AI 辅助文档生成流程编撰而成，内容力求准确、并已尽量对照官方文档核验；但在涉及生产环境的关键决策（如安全加固、并发方案、版本升级）时，仍请以官方文档与社区的最新结论为准。

第一部分 基础入门

第 1 章 DuckDB 概述与核心思想

数据库管理系统（DBMS）种类繁多，却不存在一种能包打天下的方案——Stonebraker 等人早已指出“one-size-fits-all”在数据库领域并不成立，每一款系统都在特定取舍中找寻自己的生态位。DuckDB 同样如此：它无意取代事务型主库，而是把分析型（analytical）负载在进程内（in-process）高效地完成。本章先为 DuckDB 给出清晰定位，再拆解支撑它的设计哲学与技术直觉，最后给出全书的阅读地图。

1.1 什么是 DuckDB：进程内（in-process）分析型数据库

DuckDB 是一个关系型（table-oriented）、以表为中心（table-oriented）的数据库，使用 SQL 作为查询语言。它的两条核心标签，一是进程内（in-process），二是面向分析型负载（OLAP，Online Analytical Processing）。

“进程内”意味着 DuckDB 没有独立的服务器进程（server process）。引擎被直接编译进宿主程序，与调用方运行在同一地址空间内。这与 PostgreSQL、MySQL 这类“客户端—服务器”（client-server）数据库形成根本分野：后者需要部署、运维一个常驻服务，应用经由网络与之通信；而 DuckDB 只是一个被链接进来的库（library），你引入（import）它、调用它，数据就安放在本进程的内存中。官方甚至把整个源码树合并（amalgamation）成头文件与实现文件两个产物，零外部依赖（no external dependencies），使部署与集成极其轻量。

“分析型”意味着它的优化目标不是高频短小的事务（OLTP，如每秒数万次单行读写），而是相对长时、复杂、扫描大块数据的查询——整表聚合、多张大表 join、窗口函数分析等。文档将其负载特征概括为：查询处理数据集的显著部分，而数据变更也往往是大规模批量（一次追加或改写大段数据）。

把两条标签合起来，DuckDB 的本质便清晰了：**一个跑在你程序里、为分析而生的列存数据库**。它最常见的形态是单文件数据库（扩展名 .duckdb / .db / .ddb），亦支持纯内存模式。

下面用命令行（CLI）启动并完成第一条查询：

```
# 安装 DuckDB (Python 包附带 CLI, 或下载官方二进制)
python -m pip install duckdb
```

```
# 进入交互式 CLI, 打开或创建单文件数据库
duckdb my_first.duckdb
```

进入 CLI 后执行：

```
SELECT 42 AS answer, 'hello duckdb' AS greeting;
```

这条查询不经过任何网络往返，引擎就在当前进程内解析、向量化执行并返回结果。

1.2 设计哲学与适用场景

DuckDB 官网用一组关键词归纳其设计取向：Simple（简单）、Portable（可移植）、Feature-Rich（功能完备）、Fast（快）、Extensible（可扩展）、Free（自由）、Thoroughly Tested（经严格测试）。其中与定位最相关的三条是 Simple、Portable、Fast。

简单与嵌入式体验。 DuckDB 借用了 SQLite 最成功的两点——安装简单、进程内运行——但后来通过 Quack 协议把能力延伸到进程外。它无需安装、更新或维护任何 DBMS 服务端软件。对于其面向的分析场景，进程内还带来额外红利：宿主与数据库之间的数据搬运极快，某些情况下甚至零拷贝。例如 Python 包可以在不导入、不复制任何数据的前提下，直接对 Pandas DataFrame 运行 SQL 查询（replacement scan 机制）。

可移植与“无服务器”。 由于零依赖，DuckDB 可编译到 Linux、macOS、Windows 以及 x86、ARM 架构，从资源受限的边缘设备一路部署到数百核、数 TB 内存的服务器；借助 DuckDB-Wasm，它还能跑在浏览器乃至手机里。它提供 C、C++、Go、Python、R、Rust、Java、Node.js 等语言的 API。此处的“无服务器体验”并非云函数那种 serverless，而是“无需管理服务端”——你拿到一个库、一份数据文件，分析即刻开始。

快与面向分析。 DuckDB 针对 OLAP 负载设计：这类负载要在每个被处理的值上尽量少花 CPU 周期，手段是向量化（vectorized）或即时编译（JIT）执行引擎。DuckDB 选择了 columnar-vectorized（列式向量化）执行：查询仍是解释执行，但一次操作处理一大“批”值（一个 vector），大幅削减了 PostgreSQL、MySQL、SQLite 那种逐行（tuple-at-a-time）处理的开销。

功能完备与可扩展。 进程内的便利并未以能力缩水为代价。DuckDB 支持复杂的 SQL 查询——庞大的函数库、窗口函数、CTE、各类 join 一应俱全；通过自研、面向批处理的 MVCC 提供 ACID 事务保证；数据既可用原生单文件格式存储，也能落到 DuckLake 等湖仓（lakehouse）格式，后者可扩展到 PB 级。它还提供灵活的扩展机制：Parquet、JSON、时区（ICU）、HTTP(S) 与 S3 等核心能力本身就是以扩展形式实现的，社区扩展则进一步打开了自定义类型、函数与文件格式的空间。

适用场景由此自然浮现：本地数据分析与探索、作为 Pandas / Arrow / Polars 等数据科学生态的查询引擎、把 Parquet / CSV 直接当库来查、笔记本（notebook）中的交互式分析，以及嵌入式分析（在应用进程内就地分析，无需起服务）。它不适合充当高并发、多客户端、需要严格写并发的“主数据库”——那是 PostgreSQL 或 ClickHouse 集群的领地；单进程内的写冲突会以乐观并发失败告终，需要重跑事务。

1.3 与 SQLite、ClickHouse、PostgreSQL 等系统的定位对比

理解 DuckDB 最直观的办法，是把它放进软件谱系里看。下面四个系统常被并列提及，各自的取舍却截然不同。

系统	部署形态	存储模型	主攻负载	多进程写
SQLite	进程内 (in-process)	行式 (row-oriented)	OLTP（事务）	单写者，多进程可 读

系统	部署形态	存储模型	主攻负载	多进程写
DuckDB	进程内 (in-process)	列式 (columnar)	OLAP (分析)	单进程内多写线程；多进程写经 Quack / DuckLake
PostgreSQL	客户端—服务器	行式 (row-oriented)	OLTP 为主	多客户端并发写
ClickHouse	客户端—服务器	列式 (columnar)	OLAP (分析)	服务端分布式写

SQLite 是地球上部署最广的 DBMS，DuckDB 直接继承了它“简单、进程内”的基因，但 SQLite 是行存、为事务（OLTP）优化；DuckDB 把同样的嵌入范式搬到了分析（OLAP）阵地，换成列存与向量化。二者互补而非替代：需要轻量事务用 SQLite，需要就地分析用 DuckDB。

PostgreSQL 是经典客户端—服务器数据库，DuckDB 甚至直接复用了 PostgreSQL 的 SQL 解析器（libpg_query）。但两者定位不同：PostgreSQL 把数据放在独立服务里，靠行级处理服务高频事务；DuckDB 把引擎放进你的进程，靠批处理服务分析查询。顺带一提，DuckDB 也能通过 postgres 扩展直接扫描或挂载 PostgreSQL 库，二者是协作关系。

ClickHouse 与 DuckDB 在“列式 + OLAP”上交汇，却分属两条路线：ClickHouse 是服务端、可分布式部署的面向分析的数据库，擅长远程多节点的大规模分析；DuckDB 是进程内、单文件的嵌入式分析库，强调“零运维、随取随用”。前者是“分析服务”，后者是“分析库”。

一句话定位：DuckDB 处在“嵌入式（SQLite 的便利）× 分析型（ClickHouse / OLAP 的能力）”的交叉点。

1.4 列式存储与向量化执行的直观理解

DuckDB 有四项技术支柱：列式存储（columnar storage）、向量化执行（vectorized execution）、轻量压缩（lightweight compression）、多版本并发控制（MVCC）。本节先用直觉讲清前两柱，后两柱在后续章节展开。

列存 vs 行存。想象一张 100 万行、含 id / name / age / city 四列的表。行存（如 SQLite、PostgreSQL）把每一行的四个字段紧挨着放在磁盘上；当你只关心“各城市的平均年龄”时，仍得把 name、city 一并读出来，浪费 IO 与缓存。列存则把同一列的 100 万个 age 连续存放、100 万个 city 连续存放——分析查询往往只读少数几列，于是只需加载相关列，IO 量成数量级下降。同时，同列数据类型一致，压缩效率远高于行存。

向量化执行。传统 Volcano / Iterator 模型里，每个算子实现 next()，一次吐一行，逐行穿过整棵算子树；这带来频繁的虚函数调用、糟糕的分支预测与可怜的缓存命中率。DuckDB 改为**推送式（push-based）向量化模型**：算子以固定大小的向量（Vector）为批处理单位，整批数据在算子间流动。这个固定大小由常量 STANDARD_VECTOR_SIZE 决定，默认 **2048 行**——所有算子都针对这个批量优化。

批处理带来三重收益：① 解释与虚调用的开销被摊薄到 2048 行上；② 对整批数据做紧凑循环，利于 CPU 流水线与 SIMD 指令；③ 支持“压缩执行”——部分向量格式（如 Dictionary Vector）可在不解压的情况下直接参与计算。执行期间在算子间流动的基本单位是 DataChunk，即一组 Vector 的集合。

用一个形象的比喻：Volcano 模型像流水线工人一次只递给你一颗螺丝；向量化模型则把 2048 颗螺丝装进一个料盒一次性递过来，你用机械臂批量拧紧——单位成本被剧烈摊薄。

1.5 本书路线图与阅读建议

本书以 DuckDB v1.5.5 为版本基线，共 19 章，分为七大部分：

1. **基础入门**：安装与 CLI、SQL 基础、数据类型、单文件与内存数据库。
2. **数据工程**：CSV / Parquet / JSON 等文件格式的读写，外部数据源与云存储，与其他数据库的集成。
3. **客户端与生态集成**：Python、R、Arrow、Pandas、JDBC / ODBC 等。

4. **扩展机制**：核心扩展与社区扩展、自定义函数与扩展开发。
5. **性能与运维**：并行与超内存、索引与压缩、性能剖析（EXPLAIN / EXPLAIN ANALYZE）、故障排查。
6. **原理剖析**：查询流水线、向量化引擎、存储格式、MVCC 与并发。
7. **实战与总结**：综合案例与最佳实践。

阅读建议随背景而异：初次接触分析型数据库的读者，请按顺序通读前两部分，把“列存 + 向量化”的直觉打牢；已有 SQL 经验、只想用 DuckDB 做数据处理的，可直接跳到数据工程与客户端集成部分；打算深入内核或做二次开发的，重点研读原理剖析与扩展机制章节。无论哪条路径，都建议边读边在 CLI 或 Python 里敲一遍示例——DuckDB 的进程内特性让“随手试一条查询”几乎没有成本。

1.6 本章小结

本章给出了 DuckDB 的清晰轮廓：它是一个**进程内、面向分析（OLAP）的列式数据库**，无独立服务进程，常见形态为单文件或纯内存。它的便利性继承自 SQLite 的“嵌入式”基因，分析能力则来自列存与向量化执行。与 SQLite（行存 OLTP）、PostgreSQL（客户端—服务器 OLTP）、ClickHouse（客户端—服务器列式 OLAP）相比，DuckDB 占据“嵌入式便利 × 分析能力”的交叉生态位，定位为分析库而非主数据库。

四条技术支柱中，本章用直觉讲清了**列式存储**（按列连续存放，分析查询只读所需列、压缩率更高）与**向量化执行**（以 2048 行为一批的推送式批处理，摊薄调用开销、利好 SIMD 与压缩执行）；剩余两柱——轻量压缩与 MVCC——将在后续性能与原理章节展开。带着“它跑在我的进程里、为分析而优化”这一核心认知，即可进入下一章的动手实践。

第 2 章 安装、编译与连接

DuckDB 能在笔记本上秒开、在服务器里随取随用，根子在于它回避了传统数据库的“服务进程”范式。本章从获取这个引擎开始，依次讲清三件事：怎么把它装上、怎么从源码自己编译、以及装好之后怎样建立起第一条连接并用第一行查询验证一切正常。贯穿其中的主线是 DuckDB 的“进程内（in-process）”本性——理解这一点，安装与连接的诸多选择便不再显得随意。

2.1 安装方式总览

对绝大多数读者而言，预编译的二进制文件已经足够，无需触碰编译器。DuckDB 为常见平台都准备好了开箱即用的发行物。

最轻量的入口是官方安装脚本。在 Linux 与 macOS 上，一条命令即可把最新稳定版的 CLI 装到 `~/.duckdb/cli/latest/duckdb`：

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

装完后记得把可执行文件所在的目录加入 PATH，否则每次都得敲全路径：

```
export PATH="$HOME/.duckdb/cli/latest:$PATH"
```

这个脚本支持版本回退——通过环境变量 `DUCKDB_VERSION` 可以指定任意历史发布版（最早可回溯到 `v1.0.0`）：

```
curl https://install.duckdb.org | DUCKDB_VERSION=1.2.2 sh
```

Windows 用户有一条尚处 beta 阶段的 PowerShell 安装命令：

```
powershell -NoExit iex (iwr "https://install.duckdb.org/install.ps1").Content
```

习惯包管理器的读者还有另一条路。macOS 上可用 Homebrew，Linux 上则可走 apt 或 yum 系发行渠道；Docker 用户直接拉取官方镜像 `duckdb/duckdb`，它同时支持 ARM64 与 x86_64：

```
docker run --rm -it -v "$(pwd):/workspace" -w /workspace duckdb/duckdb
```

若想在容器里使用 DuckDB UI，需额外加上 `--net host` 参数——它会转发容器的所有端口，故仅在受信任的环境这样配置。

语言客户端走的是另一条分发通道：Python 用 `pip install duckdb`（要求 Python ≥ 3.9 ），R、Java、Node 等各自经由本语言的包管理器。需要强调的是，CLI 主要面向交互式探索，官方并不推荐把它嵌入到应用里；真正嵌入业务系统时应当调用客户端库。

无论哪种途径，DuckDB 落地的形态是一致的：一个无独立服务进程的嵌入式（embedded）分析引擎，数据库就是单个文件，扩展名任取（`.duckdb`、`.db`、`.ddb` 皆可）。这种“单文件存储格式（single-file storage format）”决定了后续所有连接方式的基调。

2.2 从源码构建

当预编译产物无法满足需求时——比如要在树莓派这类实验性平台上运行、要构建某个尚未合并的 Pull Request、或需要启用特定扩展——就得自己动手编译。

构建系统建立在 CMake 之上，要求一台装有 C++11 兼容编译器的机器（GCC、Apple-Clang 或 MSVC 均可），并强烈建议搭配 Ninja 作为构建后端以获得自动并行化带来的加速。一个值得留意版本信号：当前主线只需 C++11，但即将到来的 v2.0 会把门槛抬到 C++17，届时老编译器将被拒之门外。

DuckDB 的顶层 Makefile 封装了常用目标：

```
make          # 等价于 make release
make release  # 发布版：去断言与调试符号，性能最优
make relassert # 保留 D_ASSERT 与调试符号，比 debug 快很多
make debug    # 全调试信息，二进制偏慢，需预留  $\geq 25$  GB 磁盘
GEN=ninja make # 改用 Ninja 后端
BUILD_BENCHMARK=1 make # 一并编译基准测试套件
```

`GEN=ninja make` 这个前缀在后面几乎随处可见，它告诉构建系统放弃默认的串行 `make`、改用 Ninja。

三个主流平台在命令上略有分歧。Linux 发行版差异明显：Ubuntu/Debian 用 `apt` 装 `git g++ cmake ninja-build libssl-dev libcurl4-openssl-dev`；Fedora/CentOS/RHEL 换用 `yum`，把 `g++` 写成 `gcc-c++`；Arch 则是 `pacman -S git gcc cmake ninja openssl`。拿到依赖后统一执行 `git clone` 加 `GEN=ninja make`。macOS 最省心，一条 `brew install git cmake ninja` 之后同样 `GEN=ninja make` 即可。Windows 的处境特殊——它没有 UNIX 那层 Makefile 包装，需要直接调用 CMake，且构建期和运行期都依赖 Microsoft Visual C++ Redistributable；追求开源工具链的也可走 MSYS2 的 MinGW64 路线。

编译产物统一落在 `build/release/duckdb`（Windows 下为 `duckdb.exe`）。

自带扩展是源码构建最自然的理由。用 `BUILD_EXTENSIONS` 列出需要的扩展，分号分隔：

```
BUILD_EXTENSIONS='tpch;httpfs;fts;json;parquet' GEN=ninja make
```

若想连 Python 绑定一起编出来，加上 `BUILD_PYTHON=1` 即可（`make debug` 也适用）。`httpfs` 扩展在 Linux 上常因缺 OpenSSL 而失败，补装 `libssl-dev` 通常能解决。

反复编译时，`ccache` 能显著缩短周转。在 macOS 上：

```
brew install ccache
ccache -M 50G
```

构建完成后若要验证改动正确，应当跑测试。DuckDB 沿用源自 SQLite 的 `sqllogictest` 框架，纯 SQL 测试文件放在 `test/sql` 目录，由 `unittest` 运行器驱动。调试构建下可只挑失败的用例：

```
build/debug/test/unittest test/sql/projection/test_simple_projection.test
```

不少测试开头会带 `PRAGMA enable_verification`，它把同一条查询在“未优化”与“优化”两条路径上各跑一遍并比对结果，借以捕获优化器引入的偏差——当数据集超过百行时这条校验会变慢，故大数据集测试中一般不开。

2.3 CLI 客户端详解

CLI 是初学者与 DuckDB 打交道最直接的界面，它本质上是一个单一、无依赖的可执行文件，交互方式脱胎于 SQLite 的命令行 shell，但 SQL 语法遵循 PostgreSQL 约定。

最基本的启动方式是不带任何参数：

```
duckdb
```

此时打开的是一块临时内存数据库（transient in-memory database），进程退出数据即失。与之相对，把文件路径一并给出便得到持久库文件（persistent database file）：

```
duckdb my_database.duckdb
```

文件路径不存在则新建，存在则打开——这正是单文件存储格式带来的便利。两类数据库都支持溢出到磁盘（spilling to disk）以承载超内存的工作负载，区别只在数据是否随进程结束而消失。SQL 以分号结束，省略分号可续行；退出可用 Ctrl+D、Ctrl+C 或 .exit，退出持久库时会自动做 checkpoint、合并 .wal 文件。

CLI 的灵魂在点命令（dot commands）上——它们以小数点开头，单行输入、结尾无需分号。下表列出本章最常用的一组：

点命令	作用
.help	列出全部点命令与简要说明
.mode [MODE]	切换输出格式（csv、json、markdown、box 等）
.tables [PATTERN]	列出表，含列名、类型与行数，支持 LIKE 模式
.schema [PATTERN]	打印匹配表的 CREATE 语句
.import [FILE] [TABLE]	从文件导入数据，自动推断 schema
.read [FILE]	读取并执行文件中的 SQL 与点命令
.timer on/off	显示每条 SQL 的计时

.mode 支持十余种格式，默认 duckbox 功能最丰富；脚本里若想要机器友好输出，常用 csv 或 json。点命令支持前缀简写（如 .mo ma 等价于 .mode markdown），但正经脚本里不建议依赖这种缩写。

把 DuckDB 当作流水线的一环时，非交互用法更顺手。把 SQL 文件喂给标准输入：

```
duckdb file.db < script.sql
```

也可一行内直接执行查询：

```
duckdb :memory: "SELECT 42 AS the_answer"
```

需要隔离不可信脚本的场景，安全模式（Safe Mode）派得上用场。启动时加 -safe 参数，或在会话里敲 .safe_mode，都会禁用 .cd、.import、.read、.open、.sh 等触及外部文件系统的点命令，并把 enable_external_access 关掉——此后 ATTACH 文件、COPY 读写、read_csv/read_parquet 全部失效。要紧的是，安全模式一旦开启，同一会话内无法再关闭，这是刻意为之的纵深防御设计。

1.5.x 系列给 CLI 加了一项贴心改进：Query Completion ETA。基于 Kalman 滤波（Kalman filter），CLI 能在进度条上估算查询的剩余时间。需留意它的局限——在内存压力大、I/O 瓶颈或网络延迟明显的场景下，估算会变得不准，甚至可能不再单调递减。相关开关藏在 enable_progress_bar 与 progress_bar_time（触发进度条的最小毫秒数，默认 2000）里，用 duckdb_settings() 即可查到。

2.4 连接模型概述

“连接”在 DuckDB 语境下远比传统数据库轻。它的内核是进程内（in-process / 嵌入式）架构：数据库引擎直接运行在调用方进程之内，没有独立常驻的服务进程，也没有连接池的概念。这与 PostgreSQL、MySQL 那种客户端—服务器（client-server）模型形成根本对比。

这种差异带来一组可量化的取舍：

维度	进程内 (in-process)	客户端—服务器 (client-server)
网络开销	零，数据就在本地内存	每次往返走网络协议
写入模型	单进程读写 + 多写线程 (MVCC + 乐观并发)	独立服务端协调多客户端
多进程并发写	原生不支持	服务端天然支持
部署形态	库文件随应用分发	需单独运维服务进程

在进程内模型下，DuckDB 用 MVCC (多版本并发控制，Multi-Version Concurrency Control) 配合乐观并发控制 (optimistic concurrency control) 支撑单进程内的多线程写入：只要没有写冲突，多个并发写都能成功；当两个线程试图改动同一行时，后到者以冲突错误失败，常规做法是重跑事务。原生格式不支持多进程并发写——这条需求要靠外部方案补足。

把 DuckDB 推向客户端—服务器模型的，是名为 Quack 的远程协议。它在 2026-05-12 发布，以 beta 形态随 v1.5.3 提供，把一个 DuckDB 实例变成可通过 HTTP 访问的服务器，URI 使用 quack: 协议头，默认端口 9494。协议栈相当务实：消息走明文或经反向代理的 HTTPS，请求/响应用 application/duckdb 序列化 (与 WAL 同源的代码路径)，复杂类型无损跨线；握手之后一条查询只需一次往返，大结果经后续 FETCH 分块流式返回。由于它暴露底层 DuckDB 的完整 SQL 面，默认仅绑定 localhost 并生成随机认证 token——任何面向公网的部署都应前置 TLS 终止的反向代理。Quack 目前仍是 beta，预期在 v2.0 趋于成熟。

2.5 第一条查询与安装验证

装好之后，最该做的就是用一行查询确认一切就绪。两条路径都只需几秒钟。

CLI 路径下，直接在交互界面里执行：

```
SELECT 'Hello, DuckDB!' AS greeting;
```

或者非交互地一次性验证版本：

```
duckdb -c "SELECT version() AS v;"
```

version() 会如实回显当前引擎版本，借此可确认装的是不是预期的 v1.5.5。

Python 路径同样直白。确认 pip install duckdb 成功之后：

```
import duckdb
```

```
# 内存库上的单行查询；sql() 返回惰性求值的 Relation
result = duckdb.sql("SELECT 42 AS the_answer")
print(result.fetchall())
```

duckdb.sql() 返回的是名为 Relation 的“符号化表示”，惰性求值——直到调用 fetchall() 取数时才真正执行。若想指向磁盘库而非内存，把连接显式建出来即可：

```
con = duckdb.connect("file.db") # 持久库
con.execute("CREATE TABLE IF NOT EXISTS t(i INTEGER);")
con.execute("INSERT INTO t VALUES (1), (2), (3);")
print(con.execute("SELECT sum(i) FROM t").fetchall())
con.close()
```

能稳定打印出结果，便说明二进制、客户端库与运行环境三者已对齐，可以进入后续的正式学习。

2.6 本章小结

本章把 DuckDB 的安装、编译与连接三条线索收拢起来，几个真正值得带走的要点是：

- 预编译二进制覆盖绝大多数场景，官方脚本、包管理器与 Docker 镜像三者择一即可；语言客户端走各自的包管理器，CLI 只适合交互、不宜嵌入。
- 源码构建建立在 CMake + C++11 之上，Ninja 是推荐的并行后端；BUILD_EXTENSIONS 与 BUILD_PYTHON=1 是启用扩展和 Python 绑定的关键开关；验证改动用 sqllogictest 框架的 unittest 运行器。
- CLI 的能力集中在点命令上，.help、.mode、.tables、.schema、.import、.read、.timer 最常用；安全模式与 1.5.x 新增的 Query Completion ETA 分别服务于隔离与体验两个不同诉求。
- 连接的本质是进程内（in-process）嵌入式架构——零网络、单文件、无服务进程；多进程并发写需借助 Quack（beta）或 DuckLake。
- 验证安装不必复杂：CLI 的 version() 与 Python 的 duckdb.sql(...).fetchall() 各一行足矣。

第 3 章 SQL 基础：数据类型与方言

对任何一门数据库系统而言，类型系统（type system）都是地基。DuckDB 作为一款进程内（in-process）面向分析型负载（OLAP）的列式（columnar）数据库，其类型体系在 PostgreSQL 之上做了大量对数据分析与 DataFrame 交互友好的改造。理解了这些类型与方言细节，才能避免在建模、聚合与导入数据时踩入那些看似微小却代价高昂的陷阱。本章从标量类型讲起，逐步过渡到嵌套类型（nested type），再讨论类型转换规则与 NULL 三态逻辑（NULL three-valued logic），最后梳理 DuckDB 独有的“友好 SQL”（friendly SQL）语法糖。版本事实均以 DuckDB v1.5.5 为准。

3.1 数值类型与布尔

3.1.1 定宽整数

DuckDB 提供从 1 字节到 16 字节的定宽有符号整数，以及完整的无符号家族。常见用法中，INTEGER 在取值范围、存储与性能之间最为均衡，SMALLINT 仅在磁盘吃紧时考虑，BIGINT 与 HUGEINT 用于值域不足的场景。

类型	别名	字节	取值范围
TINYINT	INT1	1	$-2^7 \sim 2^7-1$
SMALLINT	INT2, INT16, SHORT	2	$-2^{15} \sim 2^{15}-1$
INTEGER	INT4, INT32, INT, SIGNED	4	$-2^{31} \sim 2^{31}-1$
BIGINT	INT8, INT64, LONG	8	$-2^{63} \sim 2^{63}-1$
HUGEINT	INT128	16	$-2^{127} \sim 2^{127}-1$

无符号家族以 U 前缀命名：UTINYINT、USMALLINT、UIINTEGER、UBIGINT、UHUGEINT。当数值超出 HUGEINT 上限时，可使用变长整数 BIGNUM，它用约 $0.415 \times N + 3$ 字节存储 N 位十进制数，最大值约 $\pm 4.27e20201778$ ——虽有上限，并非无限。

别名陷阱是本章最需要警惕之处。有符号别名中的数字含义并不统一：INT1/INT2/INT4 的数字表示字节数（bytes），而 INT16/INT32/INT64 的数字表示位数（bits）；无符号别名 UINT8/UINT16/UINT32/UINT64 的数字则一律表示位数。于是极易混淆的两组是：INT8 并非 8 位整数，而是 8 字节（64 位）整数，等价于 BIGINT；UINT8 也不是 8 字节无符号整数，而是 8 位（1 字节）无符号整数，等价于 UTINYINT。下面这条查询能直观看清两者的体量差异：

```
SELECT typeof(1::INT8) AS int8,
       typeof(1::UINT8) AS uint8;
-- BIGINT | UTINYINT
```

3.1.2 定点小数 DECIMAL

DECIMAL(WIDTH, SCALE) (别名 NUMERIC) 表示精确定点小数, WIDTH 为总位数, SCALE 为小数位数。这里有三点与 PostgreSQL/Oracle 不同:

- 裸 DECIMAL 并非无约束类型, 而是固定绑定到 DECIMAL(18, 3); DECIMAL() 同样如此。只给 WIDTH 时 SCALE 默认 0 (即 DECIMAL(18) = DECIMAL(18, 0), 会截断小数)。
- WIDTH $\in [1, 38]$, SCALE $\in [0, 38]$ 且 SCALE $\leq$ WIDTH, 越界时绑定器 (binder) 直接报错而非截断。
- 除法产生近似浮点结果, 而加、减、乘则返回能容纳精确结果的定点类型。

内部的存储宽度随 WIDTH 变化: 1–4 位用 INT16, 5–9 位用 INT32, 10–18 位用 INT64, 19–38 位用 INT128。由此带来一个性能陷阱: WIDTH 超过 19 的 DECIMAL 走 INT128 路径, 运算明显慢于 INT32/INT64, 除非确有需要, 建议把 WIDTH 控制在 18 以内。

```
SELECT typeof(1.5::DECIMAL)      AS "DECIMAL",
       typeof(1.5::DECIMAL())    AS "DECIMAL()",
       typeof(1.5::DECIMAL(10)) AS "DECIMAL(10)",
       typeof(1.5::DECIMAL(10,)) AS "DECIMAL(10,)";
-- DECIMAL(18,3) | DECIMAL(18,3) | DECIMAL(10,0) | DECIMAL(10,0)
```

3.1.3 浮点与布尔

FLOAT (别名 FLOAT4、REAL) 为单精度 4 字节, DOUBLE (别名 FLOAT8) 为双精度 8 字节, 通常遵循 IEEE 754。特殊值 Infinity、-Infinity、NaN 必须以引号包裹的字面量书写, 可缩写为 Inf。需要精确金额且涉及加、减、乘的场景应优先选择 DECIMAL; 浮点运算则须警惕精度陷阱, 例如 1.3::FLOAT - 0.7::FLOAT 并不等于 0.6::FLOAT。

从浮点/定点转向整数时舍入方向也不同: 浮点采用四舍五入到最近、平局向偶数 (银行家舍入), 而定点在平局时向远离零方向舍入。

```
SELECT 2.5::FLOAT::INTEGER; -- 2 (round to even)
SELECT 2.5::DECIMAL::INTEGER; -- 3 (round away from zero)
```

BOOLEAN (别名 BOOL、LOGICAL) 取值 true/false, 第三态为 NULL (SQL 中的 “unknown”)。WHERE/HAVING 中仅 true 能通过, false 与 NULL 均被过滤, 这与 3.4 节的三态逻辑一脉相承。

3.2 字符串、BLOB 与日期时间

3.2.1 字符串与二进制

VARCHAR (别名 CHAR、BPCHAR、TEXT、STRING) 是 DuckDB 唯一的字符串类型, 内部以 UTF-8 编码, 原生支持 Unicode。值得注意的是 VARCHAR(n) 中的长度限制仅作兼容用途, 不参与任何性能或存储优化; 若要约束数据完整性, 应当用 CHECK 约束:

```
CREATE TABLE strings (val VARCHAR CHECK (length(val) <= 10));
```

BLOB (别名 BYTEA、BINARY、VARBINARY) 存放任意二进制数据, 含义对数据库透明, 最大约 4 GB。对于大对象, 通常建议存于文件系统、库内只保留路径 (VARCHAR):

```
SELECT '\xAA'::BLOB;
SELECT 'AB'::BLOB;
```

3.2.2 日期时间家族

DuckDB 的日期时间类型相当丰富: DATE 为日历日期, TIME/TIME WITH TIME ZONE 表示一天中的时间, TIMESTAMP 系列包含朴素时间戳 TIMESTAMP (微秒)、TIMESTAMP_NS (纳秒)、TIMESTAMP_MS、TIMESTAMP_S, 以及时区感知的 TIMESTAMPTZ。

理解 TIMESTAMPTZ 是关键。它**并不存储时区本身**，而是存储“自 Unix 纪元 1970-01-01 00:00:00+00 以来的非闰微秒数 (INT64)”，即一个绝对时间点 (instant)；“时区感知”仅体现在算术、分箱 (binning) 与字符串格式化时会使用**配置时区** (默认系统时区，常设为 UTC)。因此跨时区运算应显式用 timezone() 函数，而非依赖隐式解释。完整的时区名需依赖 ICU 扩展 (Python/R/JDBC/ODBC 客户端已内置)：

```
SELECT
    timezone('America/Denver', TIMESTAMP '2001-02-16 20:38:40') AS aware,
    timezone('America/Denver', TIMESTAMPTZ '2001-02-16 04:38:40') AS naive;
```

DATE 与 TIMESTAMP 都支持 epoch、infinity、-infinity 等特殊值。TIMESTAMP_NS WITH TIME ZONE 并不存在——带纳秒精度又带 UTC 的 Parquet 列会被转为 TIMESTAMPTZ 并丢失纳秒精度，导入此类数据时需格外留意。

3.2.3 1-based 索引

DuckDB 对字符串、LIST 等采用**从 1 开始 (1-based)** 的索引规则，唯独 JSON 对象沿用 PostgreSQL 习惯使用 0-based 索引。这一差异在混合处理列表与 JSON 时极易出错：

```
SELECT ['first', 'second', 'third'][1];           -- 'first'
SELECT '["first","second","third"]'::JSON -> 1;   -- "second"
```

3.3 嵌套与复杂类型

DuckDB 允许嵌套类型 (nested type) 任意深度嵌套，只要遵守各自的类型规则。理解每一种嵌套类型的语义边界，是写出正确且高效查询的前提。

LIST 是变长、同类型的有序序列，每行长度可以不同，这正是多数情形下需要的“数组”。ARRAY 则相反，是**定长**、同类型的序列，所有行长度与类型必须一致，适合存放词嵌入、图像嵌入等向量。这是与 PostgreSQL 最反直觉的一点：PG 的 ARRAY 是变长的，而 DuckDB 把“定长”赋予了 ARRAY、把“变长”赋予了 LIST。

```
SELECT [1, 2, 3];           -- LIST，每行长度可不同
SELECT [3, 2, 1]::INTEGER[3]; -- ARRAY，定长 3
```

STRUCT 是命名值字典，键为字符串、各值类型可不同，且每一行的键必须相同 (键是 schema 的一部分)，类似 PG 的 ROW。MAP 是键值字典，键同类型、值同类型，但各行键可以不同，适合 schema 未知或每行不同的场景。两者另有细微差别：MAP 的键区分大小写，而 STRUCT、UNION 的键不区分；MAP 缺键返回 NULL，STRUCT 缺键则报错。

UNION 是标签联合 (tagged union)，可持有多个候选类型之一，并始终携带“tag”标识当前成员；由于不能无损地隐式转回成员类型，取值须用 union_extract 显式提取。ENUM 是字典类型，把低基数列的可能取值集中存放，列内仅保留数字引用，可显著节省存储并提速，但排序按枚举定义顺序而非字母序，比较时需注意类型安全。UUID 为 128 位唯一标识，内部以 HUGEINT 存储，可用 uuid()、uuidv4()、uuidv7() 等生成。

GEOMETRY 几何类型自 DuckDB v1.5 起成为内置类型 (此前属于 spatial 扩展)；多数收益 (存储优化、统计) 需 storage version v1.5+ 的库才能享用，而距离、面积、相交等函数仍位于 spatial 扩展中：

```
CREATE TABLE geometries (id INTEGER, geom GEOMETRY);
INSERT INTO geometries VALUES (1, 'POINT (30 10)'),
    (2, 'POLYGON ((30 10, 40 40, 20 40, 10 20, 30 10))');
```

一个隐蔽的工程风险是：对嵌套类型列执行 UPDATE 时，DuckDB 实际上执行的是“先 DELETE 再 INSERT”。若表上建有 ART 索引 (显式索引或主键/唯一约束)，这一内部重写可能触发意料之外的约束冲突，设计更新逻辑时需有所防备。

3.4 类型转换与 NULL 三态逻辑

3.4.1 显式转换

DuckDB 提供 `CAST(expr AS type)` 与 `expr::type` 两种等价的显式转换写法。`TRY_CAST` 则在转换失败（如溢出）时不报错而返回 `NULL`，适合清洗来源不可信的数据：

```
SELECT CAST(999 AS TINYINT);      -- Conversion Error: value out of range for INT8
SELECT TRY_CAST(999 AS TINYINT);  -- NULL
```

隐式转换只在“不会失败”的组合间自动发生（如 `INTEGER→DOUBLE` 可行，反向不可）；可用 `can_cast_implicitly(source, target)` 检测。需要把多种类型合并为共同父类型时（如比较、`UNION`、嵌套构造），DuckDB 会采用更宽松的“组合转换（combination casting）”。

3.4.2 NULL 三态逻辑

SQL 中 `NULL` 表示“未知”，任何与 `NULL` 的比较——包括 `NULL = NULL`——结果都是 `NULL`，而非 `true`。这正是三态逻辑的核心：谓词可能为真、为假，或为未知。判定空值必须改用 `IS NULL`，而要把 `NULL` 当作相等来处理则用 `IS NOT DISTINCT FROM`：

```
SELECT NULL = NULL;              -- NULL
SELECT NULL IS NOT DISTINCT FROM NULL;  -- true
```

聚合函数大多忽略 `NULL`（`min/max` 等），但 `first`、`last`、`list`、`array_agg` 不忽略，可用 `FILTER` 子句排除。输入含 `NULL` 的函数通常返回 `NULL`，例外是 `coalesce` 返回第一个非 `NULL` 值。

3.4.3 隐式转换的收紧

自 v0.10.0 起，DuckDB 不再允许向 `VARCHAR` 的任意隐式转换——早期版本中任何类型在绑定期都能隐式转为 `VARCHAR`，这一行为已被移除（可用 `old_implicit_casting` 配置回退，但已被标记为弃用）。数字字面量本身带有特殊的 `INTEGER_LITERAL` 类型，可隐式转为任意能容纳它的整数类型；要把它显式定型，仍需用 `::`：

```
SELECT 2.5::DECIMAL(10, 2);
```

3.5 友好 SQL 与方言特性

DuckDB 最讨数据分析者喜欢的，是大量为降低书写负担而设计的语法糖，其中部分（如 `GROUP BY ALL`）由 DuckDB 首创后被其他系统效仿。

类型推断（type inference）贯穿其中。例如省略 `SELECT` 直接以 `FROM` 开头即等价于 `SELECT *`，配合 `GROUP BY ALL` 可让分组列由 `SELECT` 列表自动推断，`ORDER BY ALL` 则对所有列排序以保证结果确定的顺序。字符串字面量也能隐式解析为目标类型——`d > '1992-01-01'` 对 `DATE` 列成立，因为 `STRING_LITERAL` 可隐式转为任意类型；但已是 `VARCHAR` 的值不能反向享受此便利，否则触发 binder 报错。

前缀别名（prefix alias）用 `x: 42` 代替 `42 AS x`，读起来更贴近“变量赋值”的直觉；尾逗号（trailing comma）在列列表、`LIST` 构造中均被允许；`INSERT INTO ... BY NAME` 按列名而非位置插入，`UNION BY NAME` 同理；`SELECT * EXCLUDE/REPLACE` 则可便捷地删减或替换列：

```
FROM tbl;                                -- FROM-first，等价于 SELECT *
SELECT * FROM t GROUP BY ALL;            -- 自动推断分组列
SELECT x: 42, y: 'hello';               -- 前缀别名
SELECT a, b EXCLUDE (c) FROM t;         -- 排除列
```

此外，列别名可在 `WHERE/GROUP BY/HAVING` 中直接引用，`COLUMNS()` 表达式可配合正则批量处理多列，可复用列别名（lateral column aliases）让后续表达式直接引用前面刚定义的别名——这些特性让 DuckDB 在处理宽表与探索性查询时格外顺手。

3.6 本章小结

本章梳理了 DuckDB 类型系统与方言的关键事实，几条真正值得牢记的要点如下：

- **别名陷阱**：INT8 是 8 字节 (= BIGINT)，UINT8 是 8 位 (= UTINYINT)，混用会酿成大错；无符号别名 UINTn 的数字表示位数，有符号 INTn (小写) 多表示字节数。
- **DECIMAL 默认 (18,3)**：裸 DECIMAL 不是无约束类型，WIDTH/SCALE 越界直接报错；WIDTH > 19 走 INT128 性能明显下降，建议控制在 18 以内。
- **TIMESTAMP TZ 不存时区**：它只存纪元微秒，显示与算术依赖配置时区；不存在 TIMESTAMP_NS WITH TIME ZONE，纳秒 +UTC 会被降级。
- **LIST 与 ARRAY 语义相反**：DuckDB 的 LIST 变长、ARRAY 定长，恰好与 PostgreSQL 相反。
- **NULL 三态逻辑**：NULL = NULL 的结果是 NULL，判等须用 IS NULL / IS NOT DISTINCT FROM；自 v0.10.0 起不再允许任意隐式转 VARCHAR。
- **嵌套类型 UPDATE 实为 DELETE+INSERT**：带 ART 索引/主键/唯一约束时需警惕意外冲突。
- **友好 SQL**：自动类型推断、FROM-first、GROUP BY ALL、前缀别名等语法糖，是 DuckDB 提升分析效率的利器。

下一章将从这些基础类型出发，进入表的创建、约束与模式的实践。

第 4 章 SQL 基础：表的定义与数据写入

数据库的价值，始于你亲手建出第一张表、并往里写入第一批数据。前三章铺垫了 DuckDB 的定位与执行模型，本章正式落地到 SQL 的 DDL (数据定义语言，Data Definition Language) 与 DML (数据操纵语言，Data Manipulation Language)：如何建表、加约束、定义临时表与生成列，如何插入、更新、删除、合并数据，乃至视图、宏、模式与序列这类复用与组织手段，最后收束于事务与并发模型——这是理解 DuckDB“进程内、单写者”性格不可绕开的一环。

4.1 CREATE TABLE 与约束

CREATE TABLE 是一切的起点。最朴素的建表只声明列名与类型：

```
CREATE TABLE t1 (i INTEGER, j INTEGER);
```

主键 (PRIMARY KEY) 既可写在列上，也可声明复合主键：

```
CREATE TABLE t1 (id INTEGER PRIMARY KEY, j VARCHAR);
CREATE TABLE t1 (id INTEGER, j VARCHAR, PRIMARY KEY (id, j));
```

其余约束——NOT NULL 与 DEFAULT、CHECK、UNIQUE、FOREIGN KEY——同处建表语句之中：

```
CREATE TABLE t1 (
    i INTEGER NOT NULL DEFAULT 0,
    decimalnr DOUBLE CHECK (decimalnr < 10),
    d DATE UNIQUE,
    t TIMESTAMP
);
```

CHECK 表达行级条件；给它命名能方便日后排查：

```
CREATE TABLE t3 (id INTEGER PRIMARY KEY, x INTEGER, y INTEGER,
    CONSTRAINT x_smaller_than_y CHECK (x < y));
```

FOREIGN KEY 引用另一表的主键或唯一列，用于维护引用完整性：

```
CREATE TABLE t2 (id INTEGER PRIMARY KEY, t1_id INTEGER,
    FOREIGN KEY (t1_id) REFERENCES t1 (id));
```

两处限制须牢记：外键不支持级联删除 ON DELETE CASCADE，也不支持自引用 (self-referencing) ——向一张引用自身的表插入会触发 “Violates foreign key constraint”。

建表还有两个常用变体：CREATE OR REPLACE 等价于先 DROP 再 CREATE；IF NOT EXISTS 在表已存在时安静跳过：

```
CREATE OR REPLACE TABLE t1 (i INTEGER, j INTEGER);
CREATE TABLE IF NOT EXISTS t1 (i INTEGER, j INTEGER);
```

CREATE TABLE 还可接受 TEMP / TEMPORARY 修饰符以创建会话级临时表，其生命周期与细节留待 4.2 展开。

CTAS (CREATE TABLE ... AS SELECT，由查询建表) 是把查询结果直接物化为新表的捷径：

```
CREATE TABLE nums AS SELECT i FROM range(0, 3) t(i);
CREATE TABLE flights AS FROM 'https://duckdb.org/data/flights.csv';
```

CTAS 只复制列名与类型，**不复制约束、默认值与索引**；它本身也**不能携带约束**（主键、CHECK 等）。仅复制结构可用 WITH NO DATA 或 LIMIT 0：

```
CREATE TABLE t1 AS FROM t2 WITH NO DATA;
```

关于性能，有一条值得记住的经验：DuckDB 在每一行插入时都会逐条校验约束，海量数据写入时这会带来可观开销。官方建议的模式是先创建不带约束的表、批量灌入数据，再用 ALTER TABLE ... ADD PRIMARY KEY 补加主键；不过须注意，除主键可用 ALTER TABLE 追加外，CHECK、FOREIGN KEY 等一般约束目前尚不支持事后增删，故应在建表时一并声明到位。

4.2 临时表、生成列与默认值

临时表 (temporary table) 用 CREATE TEMP / CREATE TEMPORARY 创建，生命周期与会话绑定——仅创建它的连接可见，连接关闭即自动删除，语义与 PostgreSQL 相近：

```
CREATE TEMP TABLE session_cache (k VARCHAR, v DOUBLE);
```

临时表位于 temp.main schema，其命名可与普通表冲突；冲突时临时表优先被解析，若要引用同名普通表需用全限定名，如 memory.main.t1。它默认驻留内存，即便底层连接的是持久化 DuckDB 文件；当内存吃紧，可通过 temp_directory 配置把溢出数据落到磁盘：

```
SET temp_directory = '/path/to/directory/';
```

生成列 (generated column) 由表达式在引用时计算，省去手动维护派生值的麻烦：

```
CREATE TABLE t1 (x FLOAT, two_x AS (2 * x));
CREATE TABLE t1 (x FLOAT, two_x FLOAT GENERATED ALWAYS AS (2 * x) VIRTUAL);
```

生成列有两种语义：VIRTUAL（不落盘，每次引用时计算）与 STORED（落盘，依赖列变化时重算）。DuckDB 当前**仅支持** VIRTUAL，且为缺省选项，类型也可省略、由表达式推断。需提醒的是，生成列不能直接 INSERT 数据，其值总是从源列派生而来。

默认值 (DEFAULT) 配合 NOT NULL 给出稳妥的初始值：

```
CREATE TABLE counters (id INTEGER NOT NULL DEFAULT 0, label VARCHAR);
```

插入时若省略该列，便取 DEFAULT；也可显式写 DEFAULT 关键字：

```
INSERT INTO counters (id) VALUES (1), (DEFAULT), (3);
```

4.3 INSERT / INSERT INTO SELECT / 批量写入

最基础的插入以 VALUES 列出元组，支持一次多行：

```
INSERT INTO tbl VALUES (1), (2), (3);
INSERT INTO tbl (i) VALUES (1), (2), (3);
```

把一张表（或查询结果）整批写入另一张表，靠 INSERT INTO ... SELECT——这是批量写入的主路径：

```
INSERT INTO tbl SELECT * FROM other_tbl;
```

列匹配默认按位置 (BY POSITION), 也可显式指定目标列。DuckDB 还提供 BY NAME, 按列名而非位置匹配, 缺失列填默认值, 列名拼错则直接报错:

```
INSERT INTO tbl BY POSITION VALUES (5, 42);
INSERT INTO tbl (b, a) VALUES (5, 42);
INSERT INTO tbl BY NAME (SELECT 42 AS b, 32 AS a);
```

冲突处理 (ON CONFLICT) 让插入在遇到主键或唯一约束冲突时有确定行为。DO NOTHING 静默跳过, INSERT OR IGNORE 是其简写:

```
INSERT INTO tbl VALUES (1, 84) ON CONFLICT DO NOTHING;
INSERT OR IGNORE INTO tbl VALUES (1, 84);
```

DO UPDATE 实现 upsert (冲突时更新既有行), EXCLUDED 虚拟表指代待插入的元组:

```
INSERT INTO tbl VALUES (1, 84) ON CONFLICT DO UPDATE SET j = EXCLUDED.j;
INSERT OR REPLACE INTO tbl VALUES (1, 84);
```

省略冲突目标时针对主键; 当存在多个唯一约束且用 DO UPDATE 时, 必须显式写出冲突目标。一个隐蔽陷阱: 走 DO UPDATE 时, DuckDB 内部先把未受影响列置 NULL 再赋值, 若这些列含 NOT NULL 约束, 会触发 “NOT NULL constraint failed” —— 设计插入逻辑时需为此留好退路。

插入亦可 RETURNING 回传写入的行, 特别适合自增主键或生成列的取值确认:

```
INSERT INTO t1 SELECT 42 RETURNING *;
INSERT INTO t2 SELECT 2 AS i, 3 AS j RETURNING *, i * j AS i_times_j;
```

4.4 UPDATE / DELETE / MERGE

UPDATE 更新既有行, 条件经 WHERE 圈定:

```
UPDATE tbl SET i = 0 WHERE i IS NULL;
UPDATE tbl SET i = 1, j = 2;
```

批量更新可借助 FROM 子句或子查询连表, 通常比逐行子查询更快:

```
UPDATE original SET value = new.value FROM new WHERE original.key = new.key;
```

同一张表自我更新时要用不同别名区分:

```
UPDATE original AS true_original SET value = (...)
FROM original AS new WHERE true_original.key = new.key;
```

DELETE 删除行, 可带 WHERE 亦可清空整表 (TRUNCATE 即 DELETE FROM 无 WHERE 的别名):

```
DELETE FROM tbl WHERE i = 2;
DELETE FROM tbl;
TRUNCATE tbl;
```

这里有一个与 PostgreSQL 不同的关键事实: DELETE 只标记删除 (marked as deleted), 并不立即回收物理空间; 空间要等 CHECKPOINT 才部分回收, 而 VACUUM 在 DuckDB 里并不回收空间。配套地, RETURNING 还能回传被删内容:

```
DELETE FROM employees RETURNING name, 2025 - age AS approx_birthyear;
```

MERGE INTO 把 “ 匹配则更新、不匹配则插入 ” 合并进单条语句, 特别适合目标表没有主键约束时的 upsert:

```
MERGE INTO people
USING (SELECT unnest([3,1]) AS id, unnest(['Sarah','John']) AS name,
        unnest([95000.0,105000.0]) AS salary) AS upserts
ON (upserts.id = people.id)
WHEN MATCHED THEN UPDATE
```

```
WHEN NOT MATCHED THEN INSERT;
```

WHEN MATCHED 可细化到 UPDATE SET salary = upserts.salary 或 DELETE ; WHEN NOT MATCHED BY SOURCE THEN DELETE 还能清理“目标有、源没有”的残留行。MERGE 的这套组合使其自然胜任缓慢变化维度 (SCD, Slowly Changing Dimension) Type 2 的建模, 自 v1.4.0 起该用法被纳入官方支持范围; 对无主键表的合并更新而言, 它比 ON CONFLICT 更灵活。

4.5 视图、宏、模式与序列

视图 (VIEW) 封装一条查询、按需复用, 且**不物化 (not physically materialized)** ——每次引用都重新执行底层查询:

```
CREATE VIEW view1 AS SELECT * FROM tbl;
CREATE OR REPLACE VIEW view1 AS SELECT 42;
CREATE VIEW view1(a) AS SELECT 42;    -- 重命名输出列
```

查看视图定义: SELECT sql FROM duckdb_views() WHERE view_name = 'view1';

宏 (MACRO) 把一段逻辑参数化封装。函数宏在调用处展开为原始表达式:

```
CREATE MACRO add(a, b) AS a + b;
CREATE MACRO add_default(a, b := 5) AS a + b;    -- 默认参数
CREATE MACRO is_maximal(a INTEGER) AS a = 2^31 - 1;    -- 类型化参数
CREATE FUNCTION main.my_avg(x) AS sum(x) / count(x);    -- FUNCTION 是 MACRO 别名
```

表宏 (table macro) 返回一张表, 关键字 AS TABLE:

```
CREATE MACRO static_table() AS TABLE
    SELECT 'Hello' AS column1, 'World' AS column2;
```

按参数数量或类型重载须一次性设定:

```
CREATE MACRO add_x (a, b) AS a + b, (a, b, c) AS a + b + c;
```

模式 (SCHEMA) 用于逻辑分组, 默认 schema 为 main:

```
CREATE SCHEMA s1;
CREATE SCHEMA IF NOT EXISTS s2;
CREATE OR REPLACE SCHEMA s2;
CREATE TABLE s1.t (id INTEGER PRIMARY KEY, other_id INTEGER);
```

序列 (SEQUENCE) 生成单调整数, 常作主键默认值:

```
CREATE SEQUENCE id_sequence START 1;
CREATE TABLE tbl (id INTEGER PRIMARY KEY DEFAULT nextval('id_sequence'), s VARCHAR);
```

取值用 nextval('serial') (下一个值) 与 currval('serial') (当前值, 须先调用过 nextval); 到达极值默认报错, 加 CYCLE 可循环。

自定义类型 (TYPE) 支持枚举、结构、联合与别名:

```
CREATE TYPE mood AS ENUM ('happy', 'sad', 'curious');
CREATE TYPE many_things AS STRUCT(k INTEGER, l VARCHAR);
CREATE TYPE one_thing AS UNION(number INTEGER, string VARCHAR);
```

4.6 事务与并发模型

DuckDB 提供完整 ACID 事务, 采用 MVCC (多版本并发控制, Multi-Version Concurrency Control) 风格的快照隔离 (snapshot isolation):

```
BEGIN TRANSACTION;  
INSERT INTO person VALUES ('Ada', 52);  
COMMIT;
```

```
BEGIN TRANSACTION;  
DELETE FROM person WHERE name = 'Ada';  
INSERT INTO person VALUES ('Bruce', 39);  
ROLLBACK;    -- 或 ABORT, 二者等价
```

隔离级别上, DuckDB 保证 snapshot isolation——按 PostgreSQL 的隔离级别定义, 即 **repeatable reads (可重复读)**: 一个事务内多次读取同一数据应得到一致快照, 违反隔离级别的事务会被中止。多语句以分号分隔批量执行时, 它们身处同一条隐式事务, 任一语句失败, 整批前序语句一并回滚。

理解 DuckDB 的并发性格, 须记住三条限制。其一, **单事务只能写入一个 attached 数据库**: 多库附加时各库各自开启事务, 单一事务跨库写入会报 “a single transaction can only write to a single attached database” ——这源于 DuckDB 不维护跨库原子性, 事务原子性仅限单个库文件内。其二, 写写冲突以乐观并发方式处理: 两个事务同时修改同一数据时, 后提交者会失败, 需要重跑事务 (retry)。其三, 外键的两处硬限制前文已述——无 ON DELETE CASCADE、不支持自引用。

SAVEPOINT 可用于事务内设置回退点 (尽管官方文档未单列专篇), 它让你在不放弃整笔事务的前提下回滚到中间状态。

4.7 本章小结

本章把 DuckDB 的“定义与写入”骨架铺开: CREATE TABLE 与各类约束、CTAS 由查询建表; 临时表与会话同生灭、位于 temp.main; 生成列当前仅 VIRTUAL 一种; DEFAULT 给出稳妥初值。写入侧, INSERT 配 ON CONFLICT 实现 upsert、BY NAME 按名匹配、RETURNING 回传结果; UPDATE/DELETE 借 FROM 连表高效批量, DELETE 仅标记删除、靠 CHECKPOINT 回收空间; MERGE INTO 则以单语句完成无主键表的合并更新, 并自 v1.4.0 起支撑 SCD Type 2。

视图、宏、模式、序列与自定义类型, 是把逻辑与结构组织、复用起来的工具箱。最后的事务与并发模型点出 DuckDB 的进程内本性: MVCC 快照隔离保证 repeatable reads, 但单事务只写一个库、写写冲突需重跑、外键不支持级联删除与自引用。带着这些约束意识, 下一章即可安心展开数据类型与单文件 / 内存数据库的细致用法。

第 5 章 SQL 基础: 查询语言

SQL 是 DuckDB 与使用者对话的唯一正式语言。DuckDB 复用了 PostgreSQL 的解析器 (libpg_query), 因此标准 SQL 的骨架在这里都能直接使用; 与此同时, 它在投影 (projection)、分组、连接、集合运算与公共表表达式 (CTE) 等若干处都做了务实的扩展, 这些扩展正是 DuckDB “为分析减负” 设计取向的集中体现。本章不重复 SQL 的通用教程, 而是围绕 DuckDB 的方言特性展开: 哪些写法比标准 SQL 更省事, 哪些语义与 PostgreSQL、SQLite 存在微妙差异, 以及哪些看似花哨的语法其实是数据分析的利器。理解本章的关键, 是先接受一个事实——**书写顺序不等于逻辑执行顺序**: SELECT 写在最前, 却在逻辑上最后落地。

5.1 SELECT 与投影

SELECT 子句决定结果包含哪些列, 它可以承载任意表达式: 算术运算、标量函数、聚合或窗口函数。每个输出表达式都会被赋予一个名字, 显式命名用 AS, 缺省时由系统自动推导。DuckDB 的列名大小写不敏感, city 与 CITY 指向同一列。

最朴素的投影是星号表达式 (star expression): SELECT * FROM tbl; 取回所有列。在列众多的宽表上, 逐个点名既啰嗦又易错, DuckDB 因此提供了两个配套修饰——EXCLUDE 与 REPLACE:

```
-- 排除某一列  
SELECT * EXCLUDE (zip) FROM (VALUES (1, 'X', '123'), (2, 'Y', '456')) a(id, city, zip);
```

-- 替换某一行 (保留列名, 改写其值)
SELECT * REPLACE (lower(city) AS city) FROM (VALUES (1, 'AMSTERDAM')) a(id, city);
EXCLUDE 在“我几乎要全部列、只丢掉少数几列”时格外顺手; REPLACE 则在不改变输出列结构的前提下就地改写某列, 省去重写整个投影列表。

当列名遵循某种命名规律 (例如 number1、number2.....) 时, 逐列罗列同样低效。DuckDB 的 COLUMNS() 接受正则表达式, 按列名批量选中:

-- 选出所有以 number 开头、后接数字的列
SELECT COLUMNS('number\d+') FROM (VALUES (10, 20, 'ignore')) t(number1, number2, other);

-- 对每一列分别套用同一个聚合
SELECT min(COLUMNS(*)) FROM (VALUES (3, 9), (1, 5)) t(a, b);
COLUMNS(*) 把“所有列”作为参数传给聚合函数, 等价于对每列各写一次 min。

别名 (alias) 有两种写法。后缀别名是 SQL 通用形式: SELECT i * i AS square FROM ...。DuckDB 额外支持前缀别名, 把名字写在表达式之前、以冒号分隔:

SELECT res: i * i, root: sqrt(i) FROM range(5) t(i);

前缀别名在投影列表很长、且表达式本身已较复杂时, 能让“输出叫什么”一眼可读, 不必滚到行尾去找 AS。列名若含空格或特殊字符, 用双引号包裹: SELECT "Some Column Name" FROM tbl;

DuckDB 还有一个背离传统 SQL 习惯、却深得数据分析者喜欢的特性——**FROM-first 语法**。它允许把 FROM 前置, 甚至可以省略 SELECT:

FROM range(5) t(i) SELECT i, i * i AS square WHERE i % 2 = 0;
FROM range(5) t(i); -- 等价于 SELECT * FROM range(5) t(i);

在交互式探索数据时, FROM-first 让思路顺着“先定位数据、再决定看什么”展开。SELECT 在此被推到最后, 恰好印证了它逻辑上最后执行的本质。

投影相关的另一个值得记下的语法是 DISTINCT ON (expr): 它为每个唯一表达式值只保留一行, 若有 ORDER BY, 则保留该排序下的第一行。对大表做关键列去重时, 它比 DISTINCT * 更省——只要排序列能唯一标识, 优化器就无需对整行哈希。

5.2 WHERE、过滤与排序

WHERE 在逻辑顺序中紧随 FROM (含 JOIN) 之后执行, 对行做过滤——注意它发生在分组之前, 因此作用于聚合之前的原始数据。过滤谓词写法与通用 SQL 一致, 字符串匹配需留意 LIKE 大小写敏感、而 ILIKE 不敏感:

SELECT i FROM range(10) t(i) WHERE i > 3 AND i < 8;

排序由 ORDER BY 承担, 它在接近查询末尾、LIMIT/OFFSET 之前执行。ORDER BY 可含多个表达式 (逗号分隔), 每个可单独指定 ASC/DESC 与 NULLS FIRST/NULLS LAST。这里藏着一个与 PostgreSQL 的分歧: **DuckDB 默认的 NULL 排序是 NULLS LAST**, 并且无论升序还是降序都保持 NULLS LAST——而 PostgreSQL 在 DESC 时会把 NULL 翻到最前。这一差异在跨数据库迁移排序逻辑时极易踩坑:

SELECT v FROM (VALUES (1), (NULL), (3)) t(v) ORDER BY v; -- NULL 在末尾
SELECT v FROM (VALUES (1), (NULL), (3)) t(v) ORDER BY v DESC; --
DuckDB 中 NULL 仍在末尾

若希望行为与 PostgreSQL 完全对齐, 可设置 SET default_null_order = 'NULLS_LAST_ON_ASC_FIRST_ON_DESC'

当需要对所有列按从左到右依次排序时, 逐个列出列名是浪费。ORDER BY ALL 一次性按全部输出列排序, 且不可再混入其他表达式:

```
SELECT * FROM (VALUES (2, 'b'), (1, 'a'), (3, 'c')) t(x, y) ORDER BY ALL;
SELECT * FROM (VALUES (2, 'b'), (1, 'a'), (3, 'c')) t(x, y) ORDER BY ALL DESC;
```

LIMIT 与 OFFSET 是逻辑上最后落地的输出修饰符：LIMIT 限定返回行数，OFFSET 跳过前 N 行，二者常用于分页或取 Top-N。DuckDB 还允许 LIMIT 接受一个百分比，按总行数比例截取——没有 ORDER BY 时结果顺序不确定，仅适合快速预览：

```
SELECT i FROM range(10) t(i) ORDER BY i LIMIT 3 OFFSET 2;    -- 取第 3、4、5 行
SELECT i FROM range(100) t(i) ORDER BY i LIMIT 10%;         -- 取约前 10 行
```

5.3 聚合与分组

GROUP BY 指定按哪些列分组后再聚合。只要 SELECT 中出现了聚合函数，查询便成为聚合查询，所有未被聚合包裹的表达式都必须出现在 GROUP BY 里。DuckDB 的 GROUP BY ALL 把这件事自动化：它按 SELECT 中所有“未被聚合包裹”的列自动分组，既减少笔误，也保证投影粒度与分组粒度一致：

```
SELECT city, avg(income)
FROM (VALUES ('A', 100), ('A', 200), ('B', 300)) t(city, income)
GROUP BY ALL;
```

需要提醒的是，GROUP BY ALL 与多维分组 (GROUPING SETS/CUBE/ROLLUP) 互不兼容，后文的多维分组须显式列出维度。

多维分组用于在**单次查询**内沿多个维度汇总，等价于把多个分组查询 UNION ALL 拼起来，超聚合 (super-aggregate) 行以 NULL 填充缺失维度：

```
-- 显式分组集；() 表示“不分组的总聚合”
SELECT city, street, avg(income)
FROM (VALUES ('A', 's1', 100), ('A', 's2', 200), ('B', 's1', 300)) t(city, street, income)
GROUP BY GROUPING SETS ((city, street), (city), (street), ());
```

```
SELECT city, street, avg(income) FROM t GROUP BY ROLLUP (city, street);    --
层级下钻：(city,street),(city),()
SELECT city, street, avg(income) FROM t GROUP BY CUBE (city, street);    --
全组合：2^2 = 4 个分组集
```

ROLLUP(a,b,c) 产出 n+1 个层级分组集，适合从细到粗的下钻；CUBE(a,b,c) 产出全部 2^n 个维度组合。当分组列本身含真实 NULL 时，“真实 NULL”与“超聚合产生的 NULL”无法区分，此时用 GROUPING_ID() (返回 BIGINT 位标志) 来识别哪些列参与了当前分组。

FILTER 子句跟在聚合函数之后，只把满足谓词的行喂给该聚合，相当于聚合内部的局部 WHERE。它在“同一查询内对多个子集分别计数”时极为自然，最典型的应用是**静态 pivot (static pivot)**——把行里的类别转成并排的列：

```
SELECT
    count(i) FILTER (y = 2022) AS "2022",
    count(i) FILTER (y = 2023) AS "2023",
    count(i) FILTER (y IS NULL) AS "missing"
FROM (VALUES (1, 2022), (2, 2022), (3, 2023), (4, NULL)) t(i, y);
-- 结果：2022 = 2, 2023 = 1, missing = 1
```

相比用 CASE WHEN 实现同样效果，FILTER 对 first/last/list/array_agg 这类“不忽略 NULL”的函数处理得更准确，因为它在聚合求值前就先筛掉了无关行。

HAVING 则在 GROUP BY 之后、对**分组后的结果**做过滤——语法与 WHERE 相仿，但作用对象已是聚合输出而非原始行：

```
SELECT city, count(*) AS n
FROM (VALUES ('A'), ('A'), ('B')) t(city)
GROUP BY city
```

```
HAVING count(*) >= 2;    -- 只保留出现不少于两次的城市
```

5.4 多表连接

连接 (join) 是把两个关系 (左、右) 水平拼接的关系运算, 结果行包含两侧的全部列; 外连接在无匹配时用 NULL 填充缺失一侧。DuckDB 支持的连接类型如下表:

连接类型	语义
INNER (默认)	仅返回两侧都能配对的行
LEFT	左表每行至少出现一次, 右无匹配补 NULL
RIGHT	右表每行至少出现一次, 左无匹配补 NULL
FULL	两侧每行至少出现一次, 缺失侧补 NULL
CROSS	笛卡尔积, 无连接条件
NATURAL	按所有同名列自动等值连接
USING (col)	仅对指定同名列做等值连接 (更简洁)
SEMI	左表中存在匹配的行 (逻辑同 IN)
ANTI	左表中不存在匹配的行 (逻辑同 NOT IN, 但忽略右表 NULL)
LATERAL	右侧子查询可引用左侧列 (相关子查询的泛化)
POSITIONAL	按物理行位置连接
ASOF	时序/有序数据的最近事件连接

SEMI 与 ANTI 是 DuckDB 对 SQL 标准的有益补充, 它们的结果行数绝不会超过左表。SEMI 等价于 WHERE 列 IN (...); ANTI 等价于 WHERE 列 NOT IN (...), 但规避了 NOT IN 的著名陷阱——一旦右侧列含有 NULL, NOT IN 的语义会让整个结果变空, 而 ANTI 直接忽略右表 NULL:

-- SEMI: 左表中能在右表找到匹配者

```
SELECT * FROM (VALUES (1), (2), (3)) a(x)
SEMI JOIN (VALUES (2), (4)) b(x) USING (x);    -- 返回 x = 2
```

-- ANTI: 左表中在右表找不到匹配者; 右侧的 NULL 被忽略

```
SELECT * FROM (VALUES (1), (2), (3)) a(x)
ANTI JOIN (VALUES (2), (NULL)) b(x) USING (x);    -- 返回 x = 1, 3
```

LATERAL 让右侧子查询能引用左侧已计算的列, 是相关子查询 (correlated subquery) 的泛化——左侧每个输入值都可让右侧返回多行。DuckDB 还能自动判断何时需要 LATERAL, 关键字常常可以省略:

```
SELECT * FROM range(3) t(i), LATERAL (SELECT i + 1) t2(j);
```

POSITIONAL 连接按物理行号把两表对齐, 它始终表现为 FULL OUTER 语义: 结果长度等于较长的一侧, 短侧不足处补 NULL。它专为本身带有自然顺序的数据 (dataframe、CSV、Parquet 读入的批) 而设, 避免使用脆弱的偏移量去假装有序:

```
SELECT * FROM (VALUES (1), (2), (3)) a(x)
POSITIONAL JOIN (VALUES ('p'), ('q')) b(y);
-- 结果: (1, 'p'), (2, 'q'), (3, NULL)
```

ASOF 连接服务于时序场景: 把左表的每条记录, 与右表中“时间上最近且不超过它”的事件配对。它至少需要一个关于排序字段的不等式条件 (最常用 >=), 其余条件必须是等值 (= 或 NOT DISTINCT), 且左右表顺序敏感, 可加 OUTER 寻找无配对行:

```
SELECT t.symbol, t.when, p.price
FROM (VALUES ('A', TIMESTAMP '2024-01-01 10:00:00'),
      ('A', TIMESTAMP '2024-01-01 10:05:00')) t(symbol, when)
ASOF JOIN (VALUES ('A', TIMESTAMP '2024-01-01 09:59:00', 10.5),
            ('A', TIMESTAMP '2024-01-01 10:03:00', 11.0)) p(symbol, when, price)
```

```
ON t.symbol = p.symbol AND t.when >= p.when; -- 右表价格按时间就近匹配
```

所有连接类型都支持**自连接** (self-join), 但必须把同一张表起不同的别名, 否则会报 “Duplicate alias” 错误。下面这条查询用自连接从员工表中找出每位员工的直接上级:

```
WITH emp(id, mgr) AS (  
    VALUES (1, NULL), (2, 1), (3, 1), (4, 2)  
)  
SELECT e.id AS employee, m.id AS boss  
FROM emp e JOIN emp m ON e.mgr = m.id;
```

5.5 集合运算

集合运算把两个查询的结果纵向叠加。UNION/UNION ALL/INTERSECT/EXCEPT 传统上**按列位置合并**, 要求两侧列数相同 (类型不同会隐式转换, 结果列名取自第一个查询):

```
SELECT * FROM range(2) t(x) UNION      SELECT * FROM range(3) t(x); -- 去重合并  
SELECT * FROM range(2) t(x) UNION ALL  SELECT * FROM range(3) t(x); -- 保留重复  
SELECT * FROM range(2) t(x) INTERSECT  SELECT * FROM range(6) t(x); -- 交集  
SELECT * FROM range(5) t(x) EXCEPT   SELECT * FROM range(2) t(x); -- 差集
```

UNION 与 INTERSECT、EXCEPT 默认带集合语义 (去重), 加上 ALL 则转为包语义 (不去重、按重数计), 在已知无重复或本就希望保留重复时, ALL 能省去去重开销。

位置对齐在列顺序不一致时是个麻烦。DuckDB 的 UNION [ALL] BY NAME 改为**按列名合并**, 不要求列数或列序相同, 缺失列自动补 NULL:

```
SELECT * FROM (VALUES (1, 'x'), (2, 'y')) a(id, label)  
UNION BY NAME  
SELECT * FROM (VALUES ('p', 3), ('q', 4)) b(label, id);  
-- 两查询按 id、label 两列名对齐合并, 顺序不同也无妨; UNION BY NAME 默认去重  
UNION ALL BY NAME 则不去重。它让“结构相似但列不完全一致”的多张表合并变得轻而易举, 不必先用 SELECT 把列补齐对齐。
```

5.6 窗口函数

窗口函数 (window function) 通过 OVER 子句计算“相对其他行”的值, 关键在于它**不折叠行数**——每一行都保留, 并附上计算结果。PARTITION BY 划定分区 (类似 GROUP BY 但不合并行), ORDER BY 决定分区内的顺序:

```
SELECT  
    schema_name, function_name,  
    row_number() OVER (PARTITION BY schema_name ORDER BY function_name) AS function_rank  
FROM duckdb_functions();
```

窗口的核心是**帧 (frame)** 规范, 它划定每个当前行所“看到”的邻域。ROWS BETWEEN ... AND ... 以物理行数为界, 下面的例子对每个点取前后各一行做滑动求和:

```
SELECT i,  
    sum(i) OVER (ORDER BY i ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) AS moving_sum  
FROM range(1, 6) t(i); -- 第 1 行看到 [1,2], 中间行看到三行, 末行看到 [4,5]
```

当多个窗口函数共用同一窗口定义时, 重复书写既冗长又易错。WINDOW 子句允许把窗口命名并复用:

```
SELECT  
    row_number() OVER w AS r,  
    rank()       OVER w AS rnk  
FROM duckdb_functions()
```

```
WINDOW w AS (PARTITION BY schema_name ORDER BY function_name);
```

窗口函数的结果若想再过滤，标准做法是外包一层子查询或 CTE。DuckDB 提供 QUALIFY 子句，直接对窗口函数输出做筛选，位置在 WINDOW 之后、ORDER BY 之前——如同 HAVING 之于聚合：

```
SELECT
    schema_name, function_name,
    row_number() OVER (PARTITION BY schema_name ORDER BY function_name) AS function_rank
FROM duckdb_functions()
QUALIFY function_rank < 3;  -- 每个 schema 内只保留排名前两的函数
```

时间序列分析中常遇到零星缺失的采样点。DuckDB 的 fill() 可对 NULL 做**线性插值** (linear interpolation)，把断点之间的空缺按线性趋势补上，配合窗口即可沿时间轴逐段填补：

```
SELECT ts, val, fill(val) OVER (ORDER BY ts) AS filled
FROM (VALUES (1, 10.0), (2, NULL), (3, NULL), (4, 40.0)) t(ts, val);
-- filled 列：10, 20, 30, 40 —— 两端已知点之间等距线性补值
```

fill() 默认线性插值且不外推；需要外推时可通过其参数控制。它把“清理带洞的时序”从几十行 CASE 表达式压缩成一行调用。

5.7 CTE 与递归查询

公共表表达式 (CTE, Common Table Expression) 是用 WITH 定义的、作用域限定在单条查询内的临时视图，可相互引用、可嵌套。它让复杂查询的逻辑分层清晰，而不必落地成真实表：

```
WITH cte1 AS (SELECT 42 AS i),
     cte2 AS (SELECT i * 100 AS x FROM cte1)
SELECT * FROM cte2;
```

DuckDB 默认会**物化** (materialize) CTE——计算一次、存入临时表；在满足条件 (被引用不超过一次、不含 VOLATILE 函数、不做分组聚合等) 时则**内联** (inline, 把定义复制到引用处)。你也可以用 MATERIALIZED / NOT MATERIALIZED 显式干预——一个昂贵的 CTE 若被多次引用，强制 MATERIALIZED 可避免重复计算：

```
WITH expensive AS MATERIALIZED (SELECT count(*) FROM (VALUES (1),(2),(3)) t(x))
SELECT * FROM expensive, expensive;  -- 仅计算一次
```

WITH RECURSIVE 让 CTE 自我引用，从而表达层级、序列与图遍历。递归 CTE 的基本结构是“锚点 (anchor) 部分 UNION ALL 递归部分”。下面用斐波那契数列作最小示例：

```
WITH RECURSIVE FibonacciNumbers (RecursionDepth, FibonacciNumber, NextNumber) AS (
    SELECT 0, 0, 1                                     -- 锚点
    UNION ALL
    SELECT fib.RecursionDepth + 1,
           fib.NextNumber,
           fib.FibonacciNumber + fib.NextNumber
    FROM FibonacciNumbers fib
    WHERE fib.RecursionDepth + 1 < 10                    -- 终止条件
)
SELECT RecursionDepth, FibonacciNumber FROM FibonacciNumbers;
```

需要特别注意版本事实：自 **DuckDB 1.5.0 起**，USING KEY 递归 CTE 中弃用了递归部分的 UNION，改用 UNION ALL——因为所有行都会流入下一轮而非去重传递；**2.0.0 默认禁用**递归 UNION，到 2.1.0 将彻底移除该写法。为兼容未来版本，递归 CTE 一律使用 UNION ALL 是最稳妥的选择。

递归最自然的用武之地是图遍历，而图必有环，需要**环检测 (cycle detection)**。借助列表函数 list_position 与 list_contains 记录已访问节点，可阻止递归无限延伸：

```
WITH edges(from_node, to_node) AS (
```

```
VALUES (1, 2), (2, 3), (3, 1)          -- 含环：1 → 2 → 3 → 1
),
paths(start_node, current, path) AS (
  SELECT from_node, to_node, [from_node, to_node]
  FROM edges
  UNION ALL
  SELECT p.start_node, e.to_node, list_append(p.path, e.to_node)
  FROM paths p JOIN edges e ON p.current = e.from_node
  WHERE list_position(p.path, e.to_node) IS NULL -- 节点已访问过则不再延伸，破除环
)
SELECT * FROM paths;
```

`list_position(p.path, e.to_node) IS NULL` 保证某节点在一条路径上只出现一次；若想枚举的是两点间最短路径，可改用 `WHERE NOT EXISTS (FROM paths prev WHERE list_contains(prev.path, e.to_node))`，确保节点不被任何已有路径访问过。

`USING KEY (key_col)` 是另一种递归增强，它让 CTE 在每轮迭代中对“已存在该键的行”做**更新**而非单纯追加，相当于一张按字典键更新的并集表 (union table)。无该键时则照常追加，能显著压缩并集表体积、降低内存与耗时，常用于连通分量等图算法：

```
WITH RECURSIVE tbl(a, b) USING KEY (a) AS (
  SELECT a, b FROM (VALUES (1, 3), (2, 4)) t(a, b)
  UNION ALL
  SELECT a + 1, b FROM tbl WHERE a < 3
)
SELECT * FROM tbl;
```

此外，递归项中 CTE 名默认只指“上一轮迭代产生的行”；前缀 `recurring.` 则可访问**截至当前的全部累积行**，用于需要回顾历史并集的算法。最后提醒：DuckDB **不支持相互递归 (mutually recursive) CTE**，递归只能指向自身。

5.8 采样、VALUES、PIVOT/UNPIVOT

大数据集上，先抽样看轮廓往往比全量扫描更高效。DuckDB 提供两种几乎等效的写法：`TABLESAMPLE` 紧贴表名之后，`USING SAMPLE` 则置于表引用 (或子查询) 之后，二者都在 `FROM` 之后、`WHERE` 之前生效，支持按百分比或按行数抽样，并可指定方法 (如 `bernoulli`)：

```
SELECT * FROM (VALUES (1),(2),(3),(4),(5)) t(x) TABLESAMPLE 50%;    -- 约半数行
SELECT * FROM (VALUES (1),(2),(3),(4),(5)) t(x) USING SAMPLE 2 ROWS; -- 约两行
```

抽样结果是随机的，小数据上返回的行数并不精确，仅用于快速预览或近似统计。

`VALUES` 是表值构造器 (table value constructor)，生成固定行数的关系，可作独立语句、嵌入 `FROM`，或作为 `INSERT` 的输入：

```
SELECT * FROM (VALUES ('Amsterdam', 1), ('London', 2)) cities(name, id);
INSERT INTO cities VALUES ('Amsterdam', 1), ('London', 2);
```

`PIVOT` 把“类别在行中”的表旋转为“类别在列中”，即行转列，对列联表 (crosstab) 式报表极为实用。`ON` 指定要展开成列的列，`USING` 指定聚合方式：

```
PIVOT (SELECT country, sales
  FROM (VALUES ('NL', 100), ('NL', 200), ('UK', 150)) t(country, sales))
ON country USING sum(sales);
-- 结果：单列聚合被展开为 NL = 300、UK = 150 的一行
```

与之相对，`UNPIVOT` 执行列转行，把若干列重新堆回“键—值”两列。用 `INTO NAME ... VALUE ...` 显式声明新列名，可读性最佳：

```
UNPIVOT (SELECT * FROM (VALUES ('x', 10, 20, 30)) t(name, "2021", "2022", "2023"))
ON "2021", "2022", "2023"
INTO NAME year VALUE amount;
-- 结果：name 保持不变，原三列年份被折叠为 year / amount 两列的多行
```

PIVOT/UNPIVOT 与 5.3 节用 FILTER 手写的静态 pivot 殊途同归：前者是专为旋转设计的语句级语法，后者是通用 SQL 的表达式级写法，可按场景与可读性取舍。

5.9 本章小结

本章铺开了 DuckDB 查询语言的主干，而真正需要记住的，是它在标准 SQL 之上为分析场景做的那些“减法”与“加法”。

投影一侧，* 配合 EXCLUDE/REPLACE、COLUMNS() 正则选列，把宽表上的列操作从体力活变成一行声明；前缀别名 res：与 FROM-first 语法则是在 DuckDB 在可读性上的小执念。SELECT 逻辑上最后执行，这一点贯穿全章。

过滤与排序里，最易忽视的细节是 **NULL 排序默认 NULLS LAST 且升降序皆然**，与 PostgreSQL 的 DESC 行为不同；ORDER BY ALL 与百分比 LIMIT 则是探索数据时的顺手工具。

分组聚合的核心扩展是 GROUP BY ALL 与多维分组 GROUPING SETS/CUBE/ROLLUP；FILTER 子句把“同一查询内按不同条件分别聚合”表达得干净，并天然胜任静态 pivot；HAVING 守在分组之后做二次过滤。

连接类型中，SEMI/ANTI 补上了 IN/NOT IN 的表达缺口，其中 ANTI 自动规避 NOT IN 遇 NULL 失效的陷阱；LATERAL、POSITIONAL、ASOF 三种特色连接分别解决了相关子查询、按物理位置对齐、时序就近匹配三类真实问题。自连接务必起别名。

集合运算的 UNION [ALL] BY NAME 用列名而非位置对齐，让结构相近的表合并不再需要先手工对齐列序。窗口函数以 OVER、帧规范、WINDOW 命名复用与 QUALIFY 过滤构成完整工具链，fill() 则把时序线性插值压成一行。CTE 与 WITH RECURSIVE 是表达层级与图的利器，物化可控、环检测靠 list_position/list_contains、键式更新靠 USING KEY；版本上务必用 UNION ALL 以保证对 2.0+ 的兼容。最后，TABLESAMPLE/USING SAMPLE 抽样、VALUES 构造与 PIVOT/UNPIVOT 旋转，补足了数据整形 (reshaping) 的最后一公里。带着这些特性回看日常 SQL，多数“绕弯路”的写法都能在 DuckDB 里找到更短的直路。

第 6 章 SQL 进阶：函数与表达式

SQL 真正的表现力，并不在 SELECT、FROM 这类骨架关键字上，而在表达式 (expression) 与函数 (function) 之中。前几章已经用到了 count、sum、strftime 等函数，本章则系统梳理 DuckDB 的函数与表达式体系：从运算符优先级、类型转换、lambda 这类通用表达式机制，到聚合、字符串、日期时间、嵌套类型、JSON 等分类函数，再到条件、哈希、系统信息这类工具函数与元数据探查。读完本章，读者应能判断“该用哪个函数、为什么这样写”，而不再是零散地记忆语法。

6.1 表达式与运算符

表达式是把列、常量、运算符和函数组合成单个值的最小计算单元。DuckDB 的表达式体系有几个容易踩坑、却又极其关键的点，值得优先讲清。

运算符优先级与三值逻辑。算术运算符 (+ - * / % **) 优先级高于比较运算符 (< > <= >= = <> !=)，比较又高于逻辑运算符 (NOT、AND、OR)。例如 3 + 4 * 2 得到 11，因为乘法先算。比较运算中，只要任一操作数为 NULL，结果即为 NULL；若希望把 NULL 当作可比较的“值”来对待，应使用 IS DISTINCT FROM / IS NOT DISTINCT FROM。逻辑运算遵循三值逻辑：NULL AND false 结果为 false，NULL OR true 结果为 true。BETWEEN x AND y 等价于 x <= a AND a <= y，区间两端均含。

DuckDB 有一项与 PostgreSQL 不同的特性——**组合类型转换 (combination casting)**：不同类型比较时会自动转换，于是 `1 = true`、`1 = '1.1'` 都为真。这带来便利，却也削弱了类型严格性，编写条件时应心里有数。

CASE 与条件表达式。 CASE 是 SQL 中最通用的条件结构，可链式书写，缺省 ELSE 时返回 NULL：

```
SELECT
  CASE
    WHEN score >= 90 THEN 'A'
    WHEN score >= 60 THEN 'B'
    ELSE 'C'
  END AS grade
FROM grades;
```

COALESCE(a, b, ...) 返回参数列表中第一个非 NULL 的值；NULLIF(a, b) 在 `a = b` 时返回 NULL，否则返回 a，常用来把“无意义值”归一为 NULL：

```
SELECT COALESCE(phone, email, 'no contact') FROM users;
SELECT NULLIF(score, 0); -- 分数为 0 时返回 NULL
```

CAST 与 TRY_CAST。 类型转换用 CAST(x AS TYPE)，也支持 PostgreSQL 风格的 `x::TYPE`：

```
SELECT CAST('42' AS INTEGER);
SELECT '42'::INTEGER;
SELECT TRY_CAST('abc' AS INTEGER); -- NULL，解析失败时返回 NULL 而非报错
```

TRY_CAST 在解析失败时返回 NULL，非常适合清洗来源不可信的数据；与之同源的 TRY(expr) 也能让会抛错的表达式在该行降级为 NULL（但不能包裹易变函数、聚合或标量子查询）。

Lambda 表达式 (版本敏感)。 列表变换、列批量处理等场景会用到匿名函数 lambda。DuckDB 采用 Python 风格的双冒号写法：

```
SELECT list_transform([1, 2, 3], x : x + 1); -- [2, 3, 4]
SELECT list_transform([1, 2, 3], lambda x : x * x); -- [1, 4, 9]
```

单箭头写法 `x -> x + 1` 已在 v1.3 弃用。其弃用路线图值得记牢：v1.3 起弃用单箭头；v1.5 是最后一个默认仍可在不开开关的情况下使用单箭头的版本；v2.0 起默认禁用单箭头；v2.1 移除 lambda_syntax 开关。因此面向未来，一律推荐 `x : x + 1` 或 `lambda x : x + 1`。如需临时恢复旧写法，可用设置项 `lambda_syntax = ENABLE_SINGLE_ARROW`。

星号与 COLUMNS()。 * 及其变体是列选择的有力工具：SELECT * EXCLUDE (secret) 排除某列，REPLACE (...) 替换某列表达式，RENAME (...) 重命名；SELECT * LIKE 'col%',' * GLOB 'col*',' * SIMILAR TO 'col.' 支持模式匹配。COLUMNS(...) 则对多列批量套用同一表达式，可接收正则、列名列表，或 `lambda c: c LIKE '%num%'` 这样的列过滤：

```
SELECT * EXCLUDE (secret) FROM users;
SELECT COLUMNS('.*_id') FROM orders;
SELECT COLUMNS(lambda c : c LIKE '%num%') FROM metrics;
```

排序规则 (collation)。 字符串比较默认使用 BINARY（别名 C/POSIX），速度最快但非区域感知。内置还提供 NOCASE、NOACCENT、NFC，并可点式组合（如 NOCASE.NOACCENT）；区域相关的规则（如德语 de、中文 zh）需加载 ICU 扩展。collation 可在列级声明（VARCHAR COLLATE NOACCENT），也可用 PRAGMA default_collation 设置全局默认。

```
CREATE TABLE t (name VARCHAR COLLATE NOACCENT);
SELECT 'mühleisen' = 'muhleisen'; -- 在该列上为 true
```

dot operator 链式调用。 DuckDB 支持用点号把函数链式调用在字面量或列上，例如 `'hello'.upper()`、`name.replace('goose', 'duck')`。这一机制仅适用于标量函数 (scalar function)，不适用于表函数 (table function) —— 表函数即使无参数也必须加括号，如 `SELECT * FROM duckdb_settings();`。

6.2 聚合函数

聚合函数 (aggregate) 把多行“压”成一行，是分析查询的主力。最常用的一组是 count、sum、avg、min、max：

```
SELECT count(*), count(amount), sum(amount), avg(amount),
       min(price), max(price)
FROM orders;
```

list(x) 把所有值收集进一个 LIST，string_agg(x, sep) 则按分隔符拼接成字符串；histogram(x) 返回各取值频次构成的 MAP，是快速画像的好工具：

```
SELECT list(name) FROM users;           -- [a, b, c]
SELECT string_agg(name, ', ') FROM users; -- 'a, b, c'
SELECT histogram(category) FROM products; -- {a=2, b=1, c=5}
```

当数据规模很大、可以接受近似结果时，应使用近似聚合：

```
SELECT approx_count_distinct(user_id) FROM logs; -- 基于 HyperLogLog
SELECT approx_quantile(latency, 0.95) FROM reqs; -- 基于 T-Digest，取 95 分位
```

统计类聚合则在做数据分析时频繁出现：median 取中位数，stddev_samp 取样本标准差，corr(x, y) 取皮尔逊相关系数：

```
SELECT median(latency), stddev_samp(latency), corr(requests, errors)
FROM daily_stats;
```

以上聚合都能配合 FILTER 子句，实现“一次扫描、多口径统计”，远比写多条查询高效：

```
SELECT
  count(*)           AS total,
  count(*) FILTER (WHERE status = 'paid') AS paid,
  count(*) FILTER (WHERE status = 'refunded') AS refunded
FROM orders;
```

6.3 字符串与正则函数

文本处理是数据分析的日常。DuckDB 中字符串拼接用 || 或 concat：二者区别在于 concat 会跳过 NULL，而 || 任一操作数为 NULL 则结果为 NULL。

```
SELECT 'Hello' || ' ' || 'World';
SELECT concat('a', NULL, 'b');    -- 'ab' (自动跳过 NULL)
```

常用基础函数一览：

```
SELECT length('Hello');           -- 6 (字符数；strlen 返回字节数，此处为 4)
SELECT lower('DuckDB'), upper('DuckDB');
SELECT substring('Hello', 2, 2);  -- 'el' (起始下标为 1)
SELECT replace('hello', 'l', '-'); -- 'he--o'
SELECT contains('duckdb', 'uck'), starts_with('duckdb', 'duck');
SELECT string_split('a-b-c', '-'); -- ['a', 'b', 'c']
SELECT format('{} + {} = {}'.format(3, 5, 8)); -- '3 + 5 = 8'
```

正则能力由 RE2 引擎提供，regexp_replace 与 regexp_extract 最为常用：

```
SELECT regexp_replace('a1b2c3', '\d', '#', 'g'); -- 'a#b#c#' ('g' 表示全局替换)
SELECT regexp_extract('订单号A123', 'A(\d+)', 1); -- '123' (取第 1 捕获组)
```

regexp_replace 默认替换首个匹配，传入 'g' 选项可全局替换；regexp_extract 的最后一个参数指定捕获组编号 (0 表示整段匹配)。

6.4 日期时间函数

DuckDB 内建 DATE、TIME、TIMESTAMP、TIMESTAMP WITH TIME ZONE (需 ICU 扩展) 与 INTERVAL 等时间类型。格式化与解析是一对互补操作：strftime 把时间按格式串输出为字符串，strptime 把字符串按格式串解析为时间戳，try_strptime 在解析失败时返回 NULL 而非报错。

```
SELECT strftime(now(), '%Y-%m-%d %H:%M:%S');
SELECT strptime('2025-01-15', '%Y-%m-%d'); -- 解析为 DATE / TIMESTAMP
```

分量提取、截断与差值构成时间运算的三板斧：date_part 取某一分量，date_trunc 截断到指定精度，date_diff 求两个时间之差（单位由第一个参数指定）。

```
SELECT date_part('year', now());
SELECT date_trunc('month', now());
SELECT date_diff('day', DATE '2025-01-01', DATE '2025-01-15'); -- 14
```

epoch(x) 把时间转为 Unix 秒；make_date(y, m, d) 等构造器则从分量拼出时间值。值得特别留意 time_bucket：它把时间戳按固定间隔规整到桶起点，常用于按小时、按天汇总时序数据，自 v1.5 起增加了时区参数，使其在跨时区场景中行为更可控：

```
SELECT epoch(now());
SELECT make_date(2025, 1, 15);
SELECT time_bucket(INTERVAL '1 hour', ts) FROM events; -- v1.5 起支持时区参数
```

6.5 数组/LIST、STRUCT、MAP 操作函数

DuckDB 对嵌套类型（nested types）的支持非常深入。LIST（又称数组）可用 list_value 构造，下标从 1 开始，切片语法类 Python 且支持负数：

```
SELECT list_value(4, 5, 6); -- [4, 5, 6]
SELECT [10, 20, 30][1]; -- 10 (1 基下标)
SELECT [10, 20, 30][1:2]; -- [10, 20]
```

LIST 与 lambda 结合后能力最强：list_transform 逐元素变换，list_filter 按条件过滤，list_reduce 归约求和/求积等。它们正是上一节推荐的 $x : x + 1$ 写法的主要用武之地：

```
SELECT list_transform([1, 2, 3], x : x * 2); -- [2, 4, 6]
SELECT list_filter([1, 2, 3, 4], x : x % 2 = 0); -- [2, 4]
SELECT list_reduce([1, 2, 3, 4], (x, y) : x + y); -- 10
```

unnest(list) 把 LIST 展开成多行，是“一进多出”、改变结果集基数（cardinality）的关键操作：

```
SELECT unnest([1, 2, 3]) AS v;
-- 返回三行：1 / 2 / 3
```

STRUCT（结构体）用于把若干命名字段打包在一起，点号即可取字段：

```
SELECT struct_pack(a := 1, b := 'x').b; -- 'x'
SELECT {'name': 'duck', 'age': 5}.name; -- 'duck'
-- 也可用 struct.* 把各字段展开为独立列
```

MAP（映射）是键值对集合，按下标式的 [] 或 map_extract 查找值：

```
SELECT map {'a': 1, 'b': 2}['a']; -- 1
SELECT map_extract(map {'a': 1}, 'a'); -- 1
```

对于 UNION 类型，则用 union_tag(u) 取当前标签、union_extract(u, 'tag') 在安全匹配时取出对应值：

```
SELECT union_tag(u), union_extract(u, 'num') FROM mixed;
```

6.6 JSON 函数

JSON 已是现代数据交换的事实标准，DuckDB 通过 **json 扩展**（v1.x 中已内置并可自动加载）提供原生 JSON 处理。最基础的取值函数是 `json_extract` 与 `json_extract_string`：前者保留 JSON 片段，后者把结果作为纯文本（VARCHAR）返回：

```
SELECT json_extract('{ "name": "duck" }', '$.name');      -- { "name": "duck" } 或 "duck"
SELECT json_extract_string('{ "name": "duck" }', '$.name'); -- 'duck'
```

DuckDB 还提供两个便捷操作符：`->` 等价于 `json_extract`（返回 JSON），`->>` 等价于 `json_extract_string`（返回文本）：

```
SELECT '{ "name": "duck" }' -> '$.name';      -- JSON 片段
SELECT '{ "name": "duck" }' ->> '$.name';     -- 'duck'（文本）
```

当需要对 JSON 内部的值做变换时，可用 `json_transform`，它沿一个 JSON 路径表达式匹配目标，并以 `lambda` 改写匹配到的值：

```
SELECT json_transform('{ "a": 1 }', '$.a', x : x + 10);  -- { "a": 11 }
```

JSON 路径使用 `$.key` 形式访问对象字段、`$(i)` 访问数组元素。由于这些能力来自 json 扩展，若环境中该扩展未启用，需先 `INSTALL json` 并 `LOAD json`（在多数 v1.x 安装中会自动加载，无需手动干预）。

6.7 条件、哈希、系统函数与元数据

工具类函数虽小，却是排查与诊断的常客。**条件与判空**：除前文的 `COALESCE`、`NULLIF` 外，`IFNULL(a, b)` 等价于两参数的 `COALESCE`，`IF(cond, a, b)` 等价于单条件 `CASE`：

```
SELECT IFNULL(a, b), NULLIF(a, b), IF(score > 60, 'pass', 'fail') FROM exam;
```

类型与版本探查：`typeof(x)` 返回表达式的类型名，`version()` 返回当前 DuckDB 版本字符串，二者在调试类型推导或确认运行环境时极有用：

```
SELECT typeof(42), typeof('x'), version();
```

哈希与编码：`hash(value)` 返回 `UBIGINT`，属于非加密哈希，适合做分桶或去重判等；若需要加密级摘要，可用 `md5(s)`、`sha1(value)`、`sha256(value)`，后两者返回 `VARCHAR`：

```
SELECT hash('duckdb'), md5('duckdb'), sha256('duckdb');
```

元数据探查则有两条互补途径。`information_schema` 是一组 SQL 标准视图（基于 PostgreSQL 实现），适合用标准 SQL 读取库结构：

```
SELECT * FROM information_schema.tables;
SELECT column_name, data_type
FROM information_schema.columns
WHERE table_name = 'orders';
```

而 `duckdb_*` 表函数族是 DuckDB 自有的、信息更全的源头，多数 `information_schema` 视图本就建立在它们之上。这些表函数**即使无参数也必须加括号**：

```
SELECT * FROM duckdb_settings();  -- 全部运行期设置项
SELECT * FROM duckdb_tables();    -- 全部表
SELECT * FROM duckdb_columns();   -- 全部列
CALL duckdb_functions();          -- 全部函数（含类型、返回类型、稳定性等字段）
```

其中 `duckdb_functions()` 的 `function_type` 字段取值为 `table` / `scalar` / `aggregate` / `pragma` / `macro` / `table_macro`，是厘清“某个名字究竟是标量函数还是表函数”的权威依据——这也回扣了 6.1 节关于“表函数必须加括号、不可 dot 链式”的约束。

6.8 本章小结

本章把 DuckDB 的函数与表达式体系串成了一张地图。要点归纳为四条：

其一，**表达式的坑多于语法**。CASE/COALESCE/NULLIF 解决条件与判空，CAST 与 TRY_CAST 区分“严格转换”与“容错转换”；组合类型转换让 1 = true 为真，是 DuckDB 与 PostgreSQL 的显见差异。lambda 一律写 x : x + 1。

其二，**聚合是分析之骨**。count/sum/avg/min/max 之外，list、string_agg、histogram 提供结构化聚合；大数据下用 approx_count_distinct、approx_quantile 换速度，median、stddev_samp、corr 做统计；FILTER 子句让一次扫描产出多口径。

其三，**分类函数各司其职**。字符串用 ||/concat/replace/regexp_*/format；时间用 strf-time/strptime/date_part/date_trunc/date_diff/time_bucket（后者 v1.5 起带时区）；嵌套类型靠 list_transform/list_filter/list_reduce/unnest 与 STRUCT 点取、MAP 下标；JSON 由 json 扩展支撑，->/->> 是取值捷径。

其四，**元数据靠两个体系**。information_schema 视图走 SQL 标准，duckdb_* 表函数族信息更全、且是规范的真正源头——记住它们无参也要加括号，这是区分标量函数与表函数的硬界线。

第二部分 数据工程

第 7 章 数据导入：文件直读

数据工作者的日常，往往是从一份散落在磁盘上的文件开始——CSV 报表、Parquet 数据湖导出、JSON 接口日志。DuckDB 把“先把数据搬进数据库”这一步直接抹去：文件本身就是可查询的对象。这种“以文件为第一公民”（files as first-class query targets）的设计，让即席分析（ad-hoc analysis）的启动成本降到极低——无需建表、无需导入、无需起服务，一句 SELECT 即可从原始文件取数。本章沿着 CSV、Parquet、JSON 三条主线，讲清格式自动探测、读取参数与类型推断、下推（pushdown）优化，以及多文件与分区数据集的批量读取，构成后续数据工程章节的方法论基础。

7.1 “直接查询文件”哲学与格式自动探测

DuckDB 的核心设计理念之一是：可以像查询数据库表一样直接查询文件。用户无需先把数据 COPY 或 IMPORT 进数据库，而是直接用 SQL 读取磁盘（或远程、或压缩）上的文件。这一能力在三类最常见的数据格式上开箱即用：CSV、Parquet 与 JSON。

最直觉的用法是根据文件扩展名自动识别格式。引擎见到 .csv、.parquet、.json 后缀，便调用对应的读取器：

```
SELECT * FROM 'test.csv';      -- 自动识别为 CSV
SELECT * FROM 'test.parquet';  -- 自动识别为 Parquet
SELECT * FROM 'test.json';     -- 自动识别为 JSON
```

当文件扩展名不标准时——例如某个 Parquet 文件被命名为 test.parq——自动识别会失效，此时必须显式调用读取函数：

```
SELECT * FROM read_parquet('test.parq');
```

三种加载途径各有侧重，下表归纳其适用场景：

途径	适用场景	示例
直接文件名查询（自动识别）	快速探索、即席查询	SELECT * FROM 'flights.csv';

途径	适用场景	示例
读取函数 (<code>read_csv / read_parquet / read_json</code>)	需要传选项、类型覆盖、批量多文件	<code>SELECT * FROM read_csv('f.csv', header = false);</code>
COPY 语句	把数据装入已存在或新建的表	<code>COPY tbl FROM 'test.csv';</code>

相对路径默认相对于当前工作目录 (`working directory`) ; 也可以通过配置项 `file_search_path` 指定相对路径所扩展到的“根目录”，从而把文件组织与运行位置解耦。

压缩文件同样能直接读取。DuckDB 识别 `gzip`、`zstd` 等压缩方式，按扩展名自动探测——例如 `test.csv.gz` 会被当作 `gzip` 压缩的 CSV 透明解压后读取：

```
SELECT * FROM 'test.csv.gz';
```

把文件内容固化成持久表，最自然的方式是 CTAS (`Create Table As Select`) :

```
CREATE TABLE test AS SELECT * FROM 'test.parquet';
-- 省略 SELECT * 的 FROM-first 语法同样有效：
CREATE TABLE ontime AS FROM 'flights.csv';
```

自 DuckDB v1.3.0 起，CSV、JSON、Parquet 读取器会自动提供 `filename` 虚拟列，标明每行数据来自哪个文件，无需显式配置：

```
COPY (FROM (VALUES (42), (43)) t(x)) TO 'test.parquet';
SELECT *, filename FROM 'test.parquet';
```

早期版本需用 `filename = true` 显式开启，v1.3.0 之后该选项仅为兼容性保留。这一虚拟列在多文件合并与溯源排查时格外有用 (详见 7.5 节) 。

7.2 CSV 读取

CSV 是最常见也最容易出问题的数据格式：它没有 `schema`、方言 (`dialect`) 多样、常常含有损坏行。DuckDB 用一套 CSV sniffer (嗅探器) 自动推断配置，绝大多数场景下能直接成功读取。

7.2.1 自动检测与采样

`read_csv` 默认启用嗅探器，自动推断三类信息：一是方言——分隔符、引号规则、转义符；二是各列的数据类型；三是首行是否为表头 (`header`) 。任意一项都可在调用时显式覆盖，例如 `read_csv('file.csv', delim = '|')` 仅覆盖分隔符，其余仍自动检测：

```
SELECT * FROM read_csv('flights.csv',
    delim = '|',
    header = true,
    columns = {
        'FlightDate': 'DATE',
        'UniqueCarrier': 'VARCHAR',
        'OriginCityName': 'VARCHAR',
        'DestCityName': 'VARCHAR'
    });
```

类型检测基于文件采样，默认采样 **20,480 行**；`sample_size = -1` 表示扫描整个文件做推断。采样方式因文件而异：普通磁盘文件会“跳跃”到不同位置取样本，而 `.gz`、`stdin` 之类不可跳跃的源只在开头采样。若嗅探结果不准，可增大 `sample_size` (或直接设为 `-1`) 后再读。

嗅探器也可独立运行，返回一行诊断表，其中 `Prompt` 列给出了可直接复制执行的 `read_csv` 命令：

```
FROM sniff_csv('my_file.csv');
FROM sniff_csv('my_file.csv', sample_size = 1000);
```

方言检测的候选值为：分隔符，|；\t，引号" '（或无），转义符" '\（或无）；判定规则是选择“每行列数一致且列数最多”的方言。类型检测的候选类型按优先级从高到低为 NULL → BOOLEAN → TIME → DATE → TIMESTAMP → TIMESTAMPTZ → BIGINT → DOUBLE → VARCHAR（兜底）；VARCHAR 永远作为兜底，all_varchar = true 可整体跳过类型推断。表头检测通过比较候选表头行与其余行在类型上的差异来判断；当所有列都是 VARCHAR 时无法区分，系统会假定有表头，可用 header = false 覆盖。无表头时列名自动生成 column0, column1, ...。

7.2.2 读取参数

下表列出 read_csv 与 COPY 读取端常用的参数。标注“仅 COPY”的别名只能在 COPY ... (...) 语法中使用；COPY 语句本身始终使用 utf-8 编码，不提供 encoding 选项。

参数	类型	默认值	含义
delim / sep	VARCHAR	,	列分隔符，可为最多 4 字节的串。
delimiter	VARCHAR	,	delim 别名， 仅 COPY 。
quote	VARCHAR	"	引用值的字符串。
escape	VARCHAR	"	引号值内转义 quote 字符的字符串。
header	BOOL	false	每个文件首行是否为列名。
all_varchar	BOOL	false	跳过类型推断，所有列当作 VARCHAR。仅 read_csv 支持。
dateformat	VARCHAR	(空)	解析/写入日期的格式串。
date_format	VARCHAR	(空)	dateformat 别名， 仅 COPY 。
timestampformat	VARCHAR	(空)	解析/写入时间戳的格式串。
timestamp_format	VARCHAR	(空)	timestampformat 别名， 仅 COPY 。
nullstr / null	VARCHAR 或 VARCHAR[]	(空)	表示 NULL 的字符串。
encoding	VARCHAR	utf-8	文件编码：utf-8 / utf-16 / latin-1。 COPY 不支持 。
skip	BIGINT	0	每个文件开头跳过的行数。
max_line_size / maximum_line_size	BIGINT	2000000	最大行字节数。 COPY 不支持 。
sample_size	BIGINT	20480	自动检测用的采样行数。
columns	STRUCT	(空)	列名 + 类型 struct，指定后关闭 schema 自动检测。
types / dtypes / column_types	VARCHAR[] 或 STRUCT	(空)	列类型，按位置列表或按名 struct。
auto_detect	BOOL	true	是否启用分隔符/引号/转义/表头/类型的自动检测。
ignore_errors	BOOL	false	忽略解析错误（跳过坏行）。

参数	类型	默认值	含义
store_rejects	BOOL	false	跳过坏行并把它们存入 rejects 表。
rejects_table	VARCHAR	reject_errors	存放坏行信息的临时表名。
rejects_scan	VARCHAR	reject_scans	存放扫描参数信息的临时表名。
union_by_name	BOOL	false	跨文件按列名（而非位置）对齐。
hive_partitioning	BOOL	（自动检测）	是否按 Hive 分区路径解析。

7.2.3 读取损坏的 CSV

DuckDB 能检测并跳过结构错误，包括 CAST（列值无法转为目标类型）、MISSING COLUMNS（列数偏少）、TOO MANY COLUMNS（列数偏多）、UNQUOTED VALUE（引号未正确结束）、LINE SIZE OVER MAXIMUM（超过 max_line_size）、INVALID ENCODING（含非法字符）。错误信息包含行号、原始行、出错列、自动检测到的配置与解决建议，定位问题极为便利。

应急跳过坏行可用 ignore_errors：

```
FROM read_csv('faulty.csv',
  columns = {'name': 'VARCHAR', 'age': 'INTEGER'},
  ignore_errors = true);
```

需要系统清洗数据时，设置 store_rejects = true 会创建两个临时表：reject_scans 记录扫描参数（scan_id、file_path、delimiter、columns 等），reject_errors 记录每个坏行的详情（line、column_idx、error_type、csv_line、error_message）。相关参数 rejects_table、rejects_scan、rejects_limit（默认 0 不限）可用于自定义表名与上限。需留意 CSV 解析受投影下推影响——若只 SELECT name, age 列的 CAST 错误不会发生，被投影掉的列不会触发解析校验。

7.3 Parquet 读取与下推

Parquet 是列式压缩（columnar compressed）格式，加载与处理都极为高效。DuckDB 对 Parquet 的读写做了深度优化，并支持把 filter 与 projection **下推（pushdown）** 到文件扫描层。

7.3.1 读取与下推优化

直接读取与建表：

```
SELECT * FROM 'test.parquet';           -- 扩展名自动识别
DESCRIBE SELECT * FROM 'test.parquet';  -- 查看列与类型
CREATE TABLE test AS SELECT * FROM 'test.parquet'; -- 建表
SELECT * FROM read_parquet('test.parq'); -- 扩展名不标准时
SELECT * FROM read_parquet(['f1.parquet', 'f2.parquet', 'f3.parquet']);
SELECT * FROM 'test/*.parquet';         -- glob
SELECT * FROM read_parquet('https://some.url/some_file.parquet'); -- 远程
```

read_parquet(path) 与别名 parquet_scan(path) 等价。两类下推是 Parquet 性能的关键：

- **Projection pushdown（投影下推）**：只读取查询实际引用的列，未引用的列不被加载。
- **Filter pushdown（过滤下推）**：对列的过滤条件下沉到扫描层，借助内建的 **zonemap（min/max 统计）** 跳过 file / row group 中不满足条件的部分；对高选择性（highly selective）过滤，Bloom filter 可进一步排除无关的 row group。每行 group 写入的 min/max 统计，正是 row group pruning（行组裁剪）的依据。

写入时若按常用过滤列预先排序，可使 min/max 范围紧致且不重叠，显著提升裁剪效果：

```
COPY (FROM 'events.parquet' ORDER BY event_time)
TO 'events-sorted.parquet' (FORMAT parquet);
```

7.3.2 用 schema 重定义读取结构

Parquet 自带 schema，但当需要重命名、增删、重排或 cast 列时，可使用 read_parquet 的 schema 参数。**行级 schema 重定义依赖 field IDs (字段标识)**：写入时需用 FIELD_IDS 为列声明 ID，读取时才能按 ID 映射。

```
-- 写 Parquet 时声明 field IDs
COPY (SELECT 42::INTEGER AS i) TO 'integers.parquet' (FIELD_IDS {i: 0});

-- 读时重命名/新增列 (按 field ID 寻址)
SELECT * FROM read_parquet('integers.parquet', schema = MAP {
  0: {name: 'renamed_i', type: 'BIGINT', default_value: NULL},
  1: {name: 'new_column', type: 'UTINYINT', default_value: 43}
});
```

schema 不可与 union_by_name = true 同时使用。

7.3.3 元数据查询

Parquet 文件内嵌丰富的统计与结构信息，DuckDB 提供一组元数据函数便于检视：

函数	作用
parquet_metadata('f.parquet')	row group / 列级统计 (stats_min、stats_max、stats_null_count、compression 等)
parquet_schema('f.parquet')	内部 schema (name、type、field_id、logical_type 等)
parquet_file_metadata('f.parquet')	文件级元数据 (num_rows、num_row_groups、format_version、encryption_algorithm 等)
parquet_kv_metadata('f.parquet')	自定义键值元数据 (key、value 均为 BLOB)
parquet_full_metadata('f.parquet')	上述四类合并为单行嵌套 struct
parquet_bloom_probe(filename, col, value)	显示某值能否用 Bloom filter 排除对应 row group (返回 bloom_filter_excludes)

这些函数都支持 glob (如 parquet_metadata('data/*.parquet')) 并行查询多文件。Bloom filter 适用于 TINYINT/UTINYINT/SMALLINT/USMALLINT/INTEGER/UINTEGER/BIGINT/UBIGINT、FLOAT/DOUBLE、VARCHAR、BLOB 类型的高选择性查询裁剪。

7.3.4 加密

自 DuckDB v0.10.0 起，Parquet 读取与写入支持 Parquet Modular Encryption 规范：

```
-- 在会话内添加密钥 ( 128/192/256 位，内存存储 )
PRAGMA add_parquet_key('key256', '01234567891123450123456789112345');

-- 写加密文件
COPY tbl TO 'tbl.parquet' (ENCRYPTION_CONFIG {footer_key: 'key256'});

-- 读加密文件
SELECT * FROM read_parquet('tbl.parquet', encryption_config = {footer_key: 'key256'});
```

当前实现仅支持用 footer_key 加密 footer 与所有列，**不支持** column_keys (逐列不同密钥)，使用者会报 Not implemented Error。加密后的 Parquet 可读取 Arrow C++ (如 PyArrow) 写的“统一加密”文件。性能上，TPC-H SF1 的 lineitem (600 万行、15 列) 读写约慢 2.5 倍。

7.4 JSON 读取

JSON 由 json 扩展支持 (多数发行版默认捆绑，首次使用时自动加载)。DuckDB 既能读取 JSON 文件，也提供一整套 JSON 处理函数。

7.4.1 读取函数与格式

基础读取与建表：

```
SELECT * FROM 'todos.json';    -- 自动识别
```

```
SELECT * FROM read_json('todos.json',
    format = 'array',
    columns = {userId: 'UBIGINT', id: 'UBIGINT', title: 'VARCHAR', completed: 'BOOLEAN'});
```

-- 自动推断建表

```
CREATE TABLE todos AS SELECT * FROM 'todos.json';
```

-- v1.3.0 起返回 filename 虚拟列

```
SELECT filename, * FROM 'todos-*.json';
```

read_json 的 format 参数决定如何解析顶层结构，取值为 auto / array / newline_delimited / unstructured：

- auto：自动推断，多数情况下无需显式设置。
- array：顶层为 JSON 数组，每个元素为一个对象。
- newline_delimited (NDJSON)：每行一个 JSON 对象。
- unstructured：顶层为多个连续书写的独立 JSON 对象。

records 选项控制是否把 JSON 对象字段拆为各列 (records = true 拆列，records = false 保留为 STRUCT 列)。若只需原始对象而不拆列，可用 read_json_objects (及其别名 read_ndjson_objects、read_json_objects_auto)。

7.4.2 关键读取参数

参数	类型	默认值	含义
format	VARCHAR	array	auto / unstructured / newline_delimited / array
maximum_object_size	UIINTEGER	16777216	单个 JSON 对象的最大字节数 (16 MB)
map_inference_threshold	UBIGINT	200	子字段数超过此值时推断为 MAP 而非 STRUCT；-1 关闭 MAP 推断
auto_detect	BOOL	true	是否自动检测键名与值类型
columns	STRUCT	(空)	指定键名与值类型；指定子集时仅返回这些列

参数	类型	默认值	含义
sample_size	UBIGINT	20480	类型检测采样的对象数； -1 扫描整个文件
maximum_depth	BIGINT	-1	类型自动检测的最大嵌套深度；-1 完全检测
dateformat / timestampformat	VARCHAR	iso	日期 / 时间戳格式，iso 时自动推断 ISO 8601
compression	VARCHAR	auto_detect	none / gzip / zstd / auto_detect
ignore_errors	BOOL	false	忽略解析错误（仅 format = 'new-line-delimited' 时）
union_by_name	BOOL	false	多文件 schema 按列名统一
hive_partitioning	BOOL	（自动检测）	Hive 分区路径解析

maximum_object_size 默认 16 MB (16,777,216 字节)，限制单个 JSON 对象的体积；超出会在解析阶段报错。map_inference_threshold 默认 200：当某个对象的子字段超过 200 个时，DuckDB 推断其为 MAP 而非 STRUCT，以避免生成过于宽大的结构类型；设为 -1 可关闭 MAP 推断。

需留意 JSON 逻辑类型 JSON 的物理存储为 VARCHAR，且对空格与键顺序敏感——'{"a":5}':::JSON 与 '{"a": 5}':::JSON 不相等，重复键（如 {"a":1,"a":2}）亦被允许。其数组索引为 0-based，与 ARRAY/LIST 的 1-based 不同（遵循 PostgreSQL 约定）。

7.5 多文件与 glob、分区数据集

DuckDB 可同时读取多种类型（CSV/Parquet/JSON）的多个文件，途径有两种：**glob 语法**或**文件列表**。

7.5.1 Glob 通配符

通配符	含义
*	匹配任意数量任意字符（含无）
**	匹配任意数量子目录（含无）
?	匹配任意单个字符
[abc]	匹配括号内任一字符
[a-z]	匹配括号内范围内任一字符

```
SELECT * FROM 'dir/*.csv';           -- 当前目录
SELECT * FROM 'dir/**/*.csv';       -- 任意深度子目录
SELECT * FROM read_csv(['f1.csv', 'f2.csv']);
SELECT * FROM read_parquet(['folder1/*.parquet', 'folder2/*.parquet']);
```

S3 等远程读取因 HTTP 编码限制**不支持** **通配符**。除 glob 外，也可直接传文件列表；列表与 glob 还能组合使用。合并的 Parquet 文件要求 schema 相同，否则改用 union_by_name。

若要查找匹配的文件名而非读取内容，可用 glob 表函数：

```
SELECT * FROM glob('*');
```

7.5.2 按列名合并: union_by_name

读取多文件时必须合并各自 schema, DuckDB 提供两种方式。默认按位置 (union by position) 对齐——以第一个文件的 schema 为准, 后续文件按列位置对齐, 适用于所有文件 schema 相同。当文件 schema 不一致 (增列或改名) 时, 用 union_by_name = true 改为按列名构建统一 schema, 缺失列补 NULL:

```
SELECT * FROM read_csv('flights*.csv');           -- 按位置
SELECT * FROM read_csv('flights*.csv', union_by_name = true); -- 按名
```

按名合并会增加内存消耗, 应在确有 schema 差异时才启用。

7.5.3 Hive 分区与分区键下推

分区是把大表按“分区键”拆成多文件、组织到多层文件夹的策略。DuckDB 原生支持 Hive 分区 (key=value 目录名) 的读取, 并支持把过滤条件**下推 (pushdown)** 到分区键, 自动跳过无关文件。默认会尝试推断路径是否为 Hive 层级 (查找 'key' = 'value' 模式), 命中则自动开启 hive_partitioning:

```
SELECT * FROM read_parquet('orders/**/*.parquet', hive_partitioning = true);
```

-- 对分区键的过滤自动下推, 仅读取匹配子目录

```
SELECT * FROM read_parquet('orders/**/*.parquet', hive_partitioning = true)
WHERE year = 2022 AND month = 11;
```

-- 仅读取 orders/year=2022/month=11/ 下文件

分区列的类型默认自动推断为 DATE、TIMESTAMP 或 BIGINT; 可用 hive_types 显式指定逻辑类型, 或用 SET hive_types_autocast = 0; 关闭自动推断。设置 hive_partitioning = false 则不附加分区列。

配合 v1.3.0 起自动提供的 filename 虚拟列, 多文件与分区读取都能精确定位每一行数据的来源:

```
SELECT *, filename FROM read_parquet('orders/**/*.parquet', hive_partitioning = true);
```

对于分区写入 (把数据落成 Hive 层级), 可用 COPY ... PARTITION_BY (...), 目标参数为根目录名; 每分区建议至少约 100 MB 数据以摊薄小文件开销, 必要时用 FILENAME_PATTERN 定制文件名、用 OVERWRITE_OR_IGNORE 控制覆盖行为。

7.6 本章小结

本章建立了 DuckDB 数据导入的主线认知: 文件即表, “直接查询文件”是贯穿全章的设计哲学。要点如下:

- **格式自动探测** 基于扩展名 (.csv / .parquet / .json) 识别读取器; 扩展名不标准时改用 read_csv / read_parquet / read_json; 压缩文件 (gzip、zstd) 透明读取, CTAS 一句话即可把文件固化为持久表。
- **CSV** 依赖 sniffer 自动推断方言、类型与表头, sample_size 默认 20,480 行 (-1 全量); 读取参数丰富, 其中 delimiter、date_format、timestamp_format 为 COPY 专用别名; 损坏文件可用 ignore_errors 跳过, 或用 store_rejects 配合 reject_errors / reject_scans 临时表做系统化清洗。
- **Parquet** 的标志性能力是下推优化——projection 与 filter pushdown 借助 zonemap (min/max 统计) 与 Bloom filter 裁剪 row group; 按 field ID 的 schema 重定义可改名增列; 元数据函数族 (parquet_metadata / parquet_schema / parquet_file_metadata / parquet_kv_metadata / parquet_bloom_probe / parquet_full_metadata) 便于检视; 自 v0.10.0 起支持 Modular Encryption。
- **JSON** 经 json 扩展读取, format 区分 auto / array / newline_delimited / unstructured; maximum_object_size 默认 16 MB, map_inference_threshold 默认 200 控制 MAP 与 STRUCT 的推断边界, union_by_name 解决多文件 schema 差异。
- **多文件与分区**: glob (* / ** / ? / [abc]) 与文件列表批量读取; union_by_name 按列名合并; Hive 分区自动检测并支持分区键下推 (partition pruning); filename 虚拟列 (v1.3.0+) 为每行标注来源文件, 是溯源与排查的有力工具。

掌握这些直读能力后，数据工程中的“取数”环节几乎零门槛；下一章起将转向数据的写出与跨源交换。

第 8 章 数据导出与持久化

DuckDB 作为进程内 (in-process) 分析型数据库，没有独立运行的服务进程，一切表、索引与数据都寄居于单一的数据库文件之中。掌握“如何把内存中的查询结果写成外部文件”以及“如何让数据库文件本身可靠地落盘、并能在多库之间挂载与迁移”，是日常工程里绕不开的两类操作。前者依赖 COPY ... TO 这条贯穿 CSV、Parquet、JSON 的导出主路径；后者则涉及持久化文件、ATTACH 挂载、WAL (Write-Ahead Log, 预写日志) 与 CHECKPOINT 的协作，以及空间回收、全库迁移等运维关切。本章逐一展开这些主题，并澄清一个从 PostgreSQL 迁移过来的用户经常踩中的认知误区。

8.1 COPY ...TO：把数据写出到文件

COPY ... TO 是 DuckDB 把表或查询结果导出为外部文件的核心语句。它的源可以是表名，也可以是一段用括号包裹的查询。目标格式由扩展名推断，也可经 FORMAT 显式指定，支持 csv、parquet、json、blob 四种。

-- 直接导出整张表

```
COPY lineitem TO 'lineitem.csv';
```

-- 导出查询结果 (用括号包裹的 SELECT)

```
COPY (SELECT l_orderkey, l_comment FROM lineitem) TO 'lineitem_part.parquet' (FORMAT parquet);
```

-- 仅导出部分列

```
COPY lineitem(l_orderkey) TO 'orderkey.tbl' (DELIMITER '|');
```

COPY ... TO 的通用选项包含 FORMAT、USE_TMP_FILE、PARTITION_BY、PER_THREAD_OUTPUT、FILE_SIZE_BYTES、FILENAME_PATTERN、RETURN_FILES、RETURN_STATS、PRESERVE_ORDER 等。其中 USE_TMP_FILE 默认开启，会先写入临时文件再改名，避免写出过程中断导致文件损坏。

8.1.1 CSV 写出与压缩

CSV 写出沿用读取端对应的选项：DELIM/SEP、QUOTE、ESCAPE、HEADER、DATEFORMAT、TIMESTAMP-FORMAT、NULLSTR、NEW_LINE 等。压缩通过 COMPRESSION 控制，取值为 none、gzip、zstd。

```
COPY lineitem TO 'lineitem.csv' (HEADER, DELIMITER '|');
```

```
COPY lineitem TO 'lineitem.csv.gz' (FORMAT csv, COMPRESSION gzip);
```

```
COPY category TO 'categories.csv' (HEADER, COMPRESSION zstd);
```

8.1.2 Parquet 写出、自定义元数据与压缩

Parquet 是列式压缩格式，导出时可用丰富的写入选项：COMPRESSION (uncompressed/snappy/gzip/zstd/brotli/lz4/lz4hc，默认 snappy)、COMPRESSION_LEVEL (zstd 取 1~22)、ROW_GROUP_SIZE (row group 最小行数，默认 122,880，下限为向量大小 2,048)、ROW_GROUPS_PER_FILE、PARQUET_VERSION (V1/V2)、FIELD_IDS、WRITE_BLOOM_FILTER 等。

```
COPY tbl TO 'result-zstd.parquet' (FORMAT parquet, COMPRESSION zstd);
```

```
COPY tbl TO 'result-v2.parquet' (FORMAT parquet, PARQUET_VERSION 'V2');
```

```
COPY (FROM generate_series(100_000)) TO 'rg.parquet'  
      (FORMAT parquet, COMPRESSION zstd, ROW_GROUP_SIZE 100_000);
```

KV_METADATA {key: 'value'} 允许往 Parquet 文件的键值元数据区写入自定义信息，便于在文件之外携带来源、作者、版本等附注：

```
COPY (SELECT 42 AS i) TO 'file.parquet'
    (FORMAT parquet, KV_METADATA {'author': 'duckdb', 'phase': 'export'});
```

写完后可用 `parquet_kv_metadata('file.parquet')` 把这些信息读回。

8.1.3 JSON 写出

JSON 写出默认生成 NDJSON (每行一个 JSON 对象); 置 `ARRAY true` 则写出为单个 JSON 数组。

```
COPY (SELECT * FROM range(5) r(i)) TO 'numbers.json' (ARRAY true);
COPY (SELECT 42 AS a, 'hello' AS b) TO 'query.ndjson' (FORMAT json);
```

8.1.4 分区写出 PARTITION_BY

`PARTITION_BY` 把数据按指定列写成 Hive 分区目录 (key=value 形式), 目标位置是根目录名而非单个文件。它不能接受表达式, 需要派生列时改用子查询。

```
COPY orders TO 'orders' (FORMAT parquet, PARTITION_BY (year, month));
```

```
COPY (SELECT *, year(timestamp) AS year, month(timestamp) AS month FROM services)
TO 'test' (PARTITION_BY (year, month));
```

每个线程会对每个分区目录写一个文件 (如 `data_0.parquet`)。可用 `FILENAME_PATTERN` 自定义文件名模式, 支持 `{i}` (序号) 与 `{uuid}` (128 位 UUID)。分区并非越细越好——每分区至少约 100 MB 数据才能摊薄目录与文件管理开销。

8.1.5 PER_THREAD_OUTPUT 与统计返回

`PER_THREAD_OUTPUT` 让每个执行线程各自写出一个文件, 在下游不在乎文件数量、只关心总吞吐时显著提升并行性能; 常与 `FILE_SIZE_BYTES` 配合, 在单文件超过阈值时切割出新文件。

```
COPY (SELECT * FROM lineitem) TO 'parts' (FORMAT parquet, PER_THREAD_OUTPUT);
COPY (SELECT * FROM big) TO 'parts' (FORMAT parquet, PER_THREAD_OUTPUT, FILE_SIZE_BYTES '1GB');
```

`RETURN_FILES` 返回本次写出的文件清单, `RETURN_STATS` 额外返回每列的统计信息, 便于在导出后立刻核对产物:

```
COPY (SELECT l_orderkey, l_comment FROM lineitem) TO 'lineitem_part.parquet' (RETURN_STATS);
```

8.1.6 关键限制: COPY 不做表到表复制

`COPY` 在 `FROM` 一侧读文件、`TO` 一侧写文件, 两端都是文件系统。它不能在两张表之间直接复制数据。若要把一张表的内容搬到另一张表, 应使用 `INSERT INTO tbl2 FROM tbl1`;。下一节的 `COPY FROM DATABASE ... TO` 看似“库间复制”, 针对的也是整个数据库文件的 schema 与数据, 而非单表对单表。

8.2 持久化数据库文件与 ATTACH

DuckDB 的数据库是一个单文件, 扩展名习惯用 `.duckdb`、`.db` 或 `.ddb`。进程内引擎在打开这个文件后, 所有建表、写入、索引都落在该文件内。需要留意的是, 默认连接的是一个纯内存数据库 (`memory`), 关闭连接即丢失; 要让数据持久化, 必须在连接时直接指向一个文件——例如在 CLI 里执行 `duckdb mydata.duckdb`, 或在 Python 里调用 `duckdb.connect('mydata.duckdb')`。

8.2.1 ATTACH 多库挂载

已连接到某个库时, 可用 `ATTACH` 把更多数据库文件挂载进同一会话, 形成多库并存的工作环境:

```

ATTACH 'file.db';
ATTACH 'file.db' AS file_db;
ATTACH 'file.db' (READ_ONLY);
ATTACH 'sqlite_file.db' AS sqlite_db (TYPE sqlite);
ATTACH IF NOT EXISTS 'file.db' AS file_db;
ATTACH OR REPLACE 'file2.db' AS file_db;

```

挂载后通过 `catalog.schema.name` 三级限定访问对象，例如 `file_db.main.t`。用 `USE` 切换默认库或 `schema`，或用 `SET search_path = 'db1, db2'` 调整解析顺序。卸载则用 `DETACH file;`。

`ATTACH` 的常用选项包括：`READ_ONLY/READ_WRITE`、`TYPE` (`DUCKDB/SQLITE/MySQL/PostgreSQL`)、`BLOCK_SIZE` (取 2 的幂，范围 16,384~262,144)、`ENCRYPTION_KEY` 与 `ENCRYPTION_CIPHER` (默认 GCM 模式的 AES-256)、`RECOVERY_MODE` (如 `no_wal_writes`)。其中远程的 HTTP/S3 路径默认只读。

一个容易忽视的事实：附加定义只存活于当前会话，不会随数据库文件持久化；每次重连都需要重新执行 `ATTACH`。

8.2.2 storage_compatibility_version 与存储格式版本

DuckDB 的磁盘存储格式会随版本演进。`storage_compatibility_version` 是一项会话级设置，决定创建或附加数据库时采用的存储格式版本——它与 `ATTACH` 的 `STORAGE_VERSION` 选项同源 (自 v1.2.0 起可显式指定，取值如 `'v1.2.0'` 或 `'latest'`)。把版本压低，可以让由新版 DuckDB 写出的文件被旧版引擎读取，换取向后兼容。

```
-- 会话级：后续新建/附加的库采用该存储格式版本
SET storage_compatibility_version = 'v1.2.0';
```

```
-- 或在挂载时显式声明
ATTACH 'legacy.db' AS legacy (STORAGE_VERSION 'v1.2.0');
```

加密功能隐含要求存储版本不低于 v1.4.0，因此一旦启用 `ENCRYPTION_KEY`，该设置会被自动抬高。

8.3 Checkpoint、WAL 与空间回收

理解 DuckDB 的磁盘布局，是避免“删了数据文件却没变小”这类困惑的前提。

8.3.1 WAL 与 Checkpoint 机制

写入操作先进入 WAL (Write-Ahead Log，预写日志)，再由 CHECKPOINT 把 WAL 的内容合并进主数据文件。系统按配置 `wal_autocheckpoint` 自动触发 checkpoint，其默认阈值为 16 MiB——WAL 累积到这一体量就会自动落盘。也可手动执行：

```

CHECKPOINT;
CHECKPOINT file_db;
FORCE CHECKPOINT; -- v1.4 起：等待获取 checkpoint 锁再执行，旧版会中止进行中的事务

```

默认情况下，若有正在运行的事务，手动 `CHECKPOINT` 会失败；`FORCE CHECKPOINT` (自 v1.4.0 引入) 改为排队等待锁释放。自 v1.4.0 起，内存表也支持 checkpoint，既能压缩内存占用，也能回收已删除行的空间。

8.3.2 关键误区：VACUUM 不回收空间

这是从 PostgreSQL 转来的用户最常误解的一点。在 DuckDB 中，`DELETE` 与 `UPDATE` 只把旧行**标记为删除 (marked as deleted)**，并不会立即归还磁盘空间。真正能回收这部分空间的手段是 `CHECKPOINT`——它会合并相邻 row group 中删除率约 25% 的数据，从而部分腾出空间。

`VACUUM` 在 DuckDB 里主要是为兼容 PostgreSQL 而保留的语句，语义与其在 PostgreSQL 中截然不同：

```
VACUUM; -- 空操作 (No-op)
VACUUM my_table(my_column); -- 重建该列的 distinct 统计, 而非回收空间
VACUUM FULL; -- 不支持, 报 "Not implemented Error"
```

VACUUM **不回收任何磁盘空间**, 默认还会跳过带 ART 索引 (Adaptive Radix Tree, 自适应基数树索引) 的表。想要彻底回收被删除数据占据的空间, 可靠路径只有两条: 一是 CHECKPOINT (部分回收), 二是用 COPY FROM DATABASE ... TO 把数据重建到一个全新的库文件 (完整回收)。DROP TABLE 同理——它释放内存, 但磁盘空间往往仅被标记为 free 块, 需借助 CHECKPOINT 或重建才能实际缩容。

8.4 数据库迁移

迁移整个 DuckDB 数据库时, 最稳妥的方式是 COPY FROM DATABASE ... TO: 先后挂载源库与目标库, 再执行整库复制。

```
ATTACH 'db1.db' AS db1;
ATTACH 'db2.db' AS db2;
COPY FROM DATABASE db1 TO db2; -- 含 schema、约束、索引、序列、宏与数据
COPY FROM DATABASE db1 TO db2 (SCHEMA); -- 仅复制 schema, 不含数据
```

整库复制会保留表结构之外的约束、索引、序列与宏, 比手工逐表重建更完整。另一种目录形式的方案是 EXPORT DATABASE 'target' (生成 schema.sql + load.sql + 数据文件) 配合 IMPORT DATABASE 'source', 适合需要把数据库转存为可审阅文本的场景。

跨版本迁移时还须留意扩展的兼容性: 扩展二进制严格绑定具体的 DuckDB 版本与平台, 升级引擎后旧扩展往往无法加载, 必须重装匹配版本。重装可用 FORCE INSTALL 强制覆盖, 或用 UPDATE EXTENSIONS 让其按 ETag 条件请求、仅对确有变更的扩展执行 FORCE INSTALL:

```
FORCE INSTALL httpfs;
UPDATE EXTENSIONS; -- 更新全部
UPDATE EXTENSIONS (name_a, name_b); -- 更新指定
```

需要提醒的是, 扩展在运行时不可热重载, 更新后须重启进程方能生效。

8.5 本章小结

- COPY ... TO 同时承担 CSV、Parquet、JSON 的文件导出, 源可以是表或括号包裹的查询; COMPRESSION 控制压缩算法, PARTITION_BY 写出 Hive 分区目录, PER_THREAD_OUTPUT 提升并行吞吐, RETURN_STATS/RETURN_FILES 回传统计与清单, KV_METADATA 写入自定义键值元数据。
- COPY 两端都是文件系统, 不能在两张表之间直接复制; 表到表应改用 INSERT INTO tbl2 FROM tbl1。
- 持久化依赖单文件数据库 (.duckdb/.db/.ddb), 默认连接为内存库, 必须连到文件才会落盘; ATTACH 挂载多库, 经 catalog.schema.name 限定访问, storage_compatibility_version 与 ATTACH 的 STORAGE_VERSION 共同决定磁盘格式版本以换取兼容。
- 写入先入 WAL, 由 CHECKPOINT 合并进主文件, wal_autocheckpoint 默认 16 MiB 自动触发; DELETE/UPDATE 仅标记删除, 空间回收靠 CHECKPOINT 或整库重建, 而 VACUUM 不回收空间——这一点与 PostgreSQL 语义不同。
- 全库迁移用 COPY FROM DATABASE ... TO (或 EXPORT/IMPORT DATABASE); 跨 DuckDB 版本时须以 FORCE INSTALL / UPDATE EXTENSIONS 重装匹配版本的扩展。

第三部分 客户端与生态集成

第 9 章 Python 客户端深入

DuckDB 作为进程内 (in-process) 分析型数据库, 与 Python 的契合不只是“能连上”那么简单。Python 是 DuckDB 的一级 (Primary) 客户端, 拥有与 SQL 语法一致、与磁盘格式互通的完整能力, 且天然衔接

pandas、PyArrow、Polars、NumPy 这一整套数据科学生态。理解 Python 客户端的内部机制——惰性求值的 `关系 API (relational API)`、参数化的 `DB-API`、零拷贝的 `Arrow` 互操作、以及最快的批量写入路径 `Appender`——是把 DuckDB 用对、用快的分水岭。本章所有示例均以 DuckDB v1.5.5 为基线，代码均可在独立进程中直接运行。

9.1 安装与连接

9.1.1 安装

DuckDB 的 Python 包名为 `duckdb`，使用 `pip` 或 `conda` 安装：

```
pip install duckdb
conda install python-duckdb -c conda-forge
```

该包对 Python 版本的要求是 ≥ 3.9 。DuckDB 与 `pandas`、`PyArrow`、`Polars` 的互操作是可选依赖——若只需基础功能，安装核心包即可；若要享受与 `DataFrame` 的无缝转换，则按需安装对应库：

```
pip install "duckdb[pandas]"      # 启用 pandas 互操作
pip install "duckdb[pyarrow]"    # 启用 PyArrow 互操作
pip install "duckdb[polars]"     # 启用 Polars 互操作
```

9.1.2 建立连接

导入包后，最朴素的用法是模块级快捷调用 `duckdb.sql()`，它复用一条隐式的全局内存连接：

```
import duckdb

duckdb.sql("SELECT 42 AS the_answer").show()
```

生产代码中更推荐显式创建连接对象，因为模块级全局连接并非线程安全，多任务并行时容易相互干扰。`connect()` 是创建连接的入口，它既可以打开内存库，也可以打开持久化的单文件数据库：

```
import duckdb

con = duckdb.connect()          # 内存库 (in-memory database)
con = duckdb.connect("file.db") # 持久库 (on-disk, 文件扩展名任意)
```

持久库的扩展名并不强制为 `.duckdb`、`.db`、`.ddb` 或任意后缀均可，因为 DuckDB 的磁盘格式与文件后缀无关。需要调优运行参数时，可通过 `config` 字典在连接建立时传入，例如限制单连接线程数：

```
con = duckdb.connect(config={'threads': 1})
con = duckdb.connect(config={'storage_compatibility_version': 'latest'})
```

连接对象支持 Python 上下文管理器 (`context manager`)，退出 `with` 块时会自动关闭并刷写，避免句柄泄漏：

```
with duckdb.connect("file.db") as con:
    con.execute("CREATE TABLE t(i INTEGER)")
    con.execute("INSERT INTO t VALUES (1), (2), (3)")
# 此处连接已自动关闭
```

多进程并发读取同一持久库时，应显式以只读方式打开，避免写操作相互干扰：

```
con = duckdb.connect("file.db", read_only=True)
```

9.1.3 线程与并发

`duckdb.connect()`、`duckdb.connect(':default:')` 与 `duckdb.sql()` 建立的都是同一全局内存连接，**非线程安全**。并行场景应让每个线程各自建立独立连接。`cursor()` 方法返回的只是同一连接上的一

一个新句柄，**不能**实现并发查询——单连接上的查询会被加锁并串行化执行。这一约束与 DuckDB 的进程内本质一致：把连接视为“一个数据库会话”，而非“一个查询槽位”。

9.2 关系 API

9.2.1 什么是 Relation

关系 API (relational API) 围绕核心类型 DuckDBPyRelation 展开，其返回的不是数据本身，而是查询的**符号化表示 (symbolic representation)**。它的关键特征是**惰性求值 (lazy evaluation)**：构造 Relation 时并不真正执行计算，只有显式取数或打印时才触发物化 (materialization)。调用 show() 或 print() 时，DuckDB 默认仅取前 10K 行用于展示。

惰性带来的直接收益是**下推优化 (pushdown optimization)**。下面的链式调用中，filter 与 project 不会逐步产生中间结果，而是被合并、重写后一次性推送到执行引擎：

```
import duckdb

rel = duckdb.sql("SELECT * FROM range(1000000) t(i)")
rel = rel.filter("i % 2 = 0")          # 惰性：未执行
rel = rel.project("i, i * i AS square") # 惰性：未执行
result = rel.limit(5).df()             # 此刻才物化，且 LIMIT 已下推
print(result)
```

由于过滤与投影都被下推，range(1000000) 中绝大多数行从未真正生成——这正是延迟计算 (deferred computation) 优于“先全量物化再处理”的地方。下推并非黑箱，可用 explain() 直接观察执行计划：

```
print(rel.limit(5).explain())
```

计划中会显示过滤条件与 LIMIT 已被合并进底层扫描算子，而非在物化后才做取舍——这正是关系 API 把“描述计算”与“执行计算”分离带来的可优化空间。

9.2.2 从数据源创建 Relation

关系 API 提供多种创建入口，覆盖文件、DataFrame、Arrow、原生表与值构造：

```
import duckdb

con = duckdb.connect()

rel_csv = con.read_csv("example.csv")          # 读取 CSV 文件
rel_parq = con.read_parquet("data.parquet")    # 读取 Parquet 文件
rel_json = con.read_json("data.json")          # 读取 JSON 文件
rel_tbl = con.table("my_table")                # 引用已存在表
rel_vals = con.values([1, 2, 3], "x")          # 由值构造
```

拿到一个 Relation 后，在不触发物化的前提下即可检视其结构：columns 列出列名，dtypes/types 给出各列逻辑类型，shape 返回 (行数, 列数)，describe 给出摘要。这些方法返回的仍是元信息，不意味着执行了查询——这正是惰性 API 的便利：先看清数据长相，再决定如何计算。

```
print(rel.columns)  # 列名列表
print(rel.dtypes)   # 各列类型
print(rel.shape)    # (估计行数, 列数)
```

9.2.3 链式转换与物化

Relation 的转换方法 (transformation) 支持链式调用且全程惰性，包括 filter、project/select、aggregate、order/sort、limit、join (inner/left/right/outer/semi/anti)、union/except_/intersect、cross、apply。转换结束后再以物化方法取回结果。

```

import duckdb

con = duckdb.connect()
con.execute("CREATE TABLE sales(region VARCHAR, amount DOUBLE)")
con.execute("INSERT INTO sales VALUES ('East', 100), ('West', 250), ('East', 150)")

rel = con.table("sales")
agg = (rel
        .aggregate("region, sum(amount) AS total", groups="region")
        .order("total DESC"))

print(agg.df())          # Pandas DataFrame
print(agg.pl())          # Polars DataFrame
print(agg.arrow())       # Apache Arrow 表 (推荐 to_arrow_reader)
print(agg.fetchnumpy())  # NumPy 字典 (各列为数组)

```

四种物化出口各有用途：`.df()` 取 pandas，`.pl()` 取 Polars，`.arrow()` 取 PyArrow（对外推荐 `to_arrow_reader(chunk_size)`，可按 `chunk_size` 分块流式读取），`.fetchnumpy()` 取 NumPy 字典，其中带 NULL 的列会以掩码数组（masked array）表达缺失。若结果集过大，可用 `fetch_df_chunk(vector_multiple)` 分块取回 DataFrame，块大小等于 `2048 × vector_multiple`——这里的 2048 正是 DuckDB 向量化执行的基本单位 `STANDARD_VECTOR_SIZE`，与底层列式执行引擎的 DataChunk/Vector 一一对应。

9.2.4 一个性能警示

关系 API 的 `sql()/query()/from_query()` 方法在传参时存在显著开销（约 5 倍处理、近 2 倍内存）。需要参数化查询时，应改用 `execute()` 做参数绑定，再借助 replacement scan（见 9.4 节）把结果接入关系 API，而非在 Relation 内部反复嵌入参数化 SQL。

9.3 DB-API 与参数化查询

9.3.1 标准 DB-API 用法

DuckDB 的 `con.execute()` 实现了 Python DB-API 约定：执行语句、用 `fetchall()/fetchone()` 取数。取回的是原生 Python 对象（而非 DataFrame），列名可通过 `description` 属性查看。

```

import duckdb

con = duckdb.connect()
con.execute("CREATE TABLE items(name VARCHAR, price INTEGER, qty INTEGER)")
con.execute("INSERT INTO items VALUES ('laptop', 2000, 1), ('mouse', 50, 3)")

rows = con.fetchall()
print(rows)          # [('laptop', 2000, 1), ('mouse', 50, 3)]
print(con.description) # 列名与类型描述

fetchone() 逐行取数，取完返回 None 后才真正关闭底层事务；fetchdf()（别名 fetch_df）则直接返回 pandas DataFrame，省去手动转换。

df = con.execute("SELECT * FROM items").fetchdf()
print(df)

```

9.3.2 参数化查询与防注入

把用户输入直接拼进 SQL 字符串是 SQL 注入（SQL injection）的根源。DuckDB 支持两种占位符：`?`（顺序占位）与 `$1`（可复用、按序号），二者均可配合参数列表执行：

```
con.execute("INSERT INTO items VALUES (?, ?, ?)", ["keyboard", 300, 2])
con.execute("SELECT * FROM items WHERE price > ? AND qty >= ?", [100, 2])
```

命名参数以字典传入，键名对应 \$ 参数名：

```
con.execute("SELECT $p AS val", {"p": 5})
```

参数化查询的底层是 prepared statement (预处理语句)，值在绑定时按类型安全转换，从根本上杜绝了字符串拼接带来的注入风险。DuckDB 还提供 executemany() 批量执行同一语句多组数据：

```
con.executemany("INSERT INTO items VALUES (?, ?, ?)",
                [[{"monitor", 800, 1}, {"cable", 20, 10}])
```

executemany() 适合小批量数据。**灌入海量数据时切勿使用它**——每条记录都会触发一次语句执行的开销，速度远不及 9.5 节介绍的 Appender 或 COPY/readers。

9.4 与 pandas / PyArrow / Polars / NumPy 互操作

9.4.1 replacement scan: 把 DataFrame 当表查

DuckDB 能直接查询当前作用域中可见的 Python DataFrame，机制是 **replacement scan (替换扫描)**：当 SQL 的 FROM 子句引用了一个 DataFrame 变量名时，DuckDB 自动把它替换为一个可扫描的数据源，无需任何显式注册。

```
import duckdb
import pandas as pd
```

```
df = pd.DataFrame({"i": [1, 2, 3], "j": ["one", "two", "three"]})
```

```
# 直接把 DataFrame 当作表查询
result = duckdb.sql("SELECT sum(i) AS total_i FROM df").df()
print(result)
```

Polars 与 PyArrow 对象同样适用，只是把变量换成对应类型：

```
import polars as pl
import pyarrow as pa
```

```
polars_df = pl.DataFrame({"x": [1, 2, 3]})
arrow_table = pa.table({"y": [10, 20, 30]})
```

```
duckdb.sql("SELECT * FROM polars_df").show()
duckdb.sql("SELECT * FROM arrow_table").show()
```

replacement scan 仅对 sql()/execute() 调用处**可见**的变量有效。若想关闭这一行为，可执行 SET python_enable_replacements=false。当同名冲突时，优先级为：register() 显式注册的视图 > 原生表/视图 > replacement scan。pandas 的 object 列在转换前会做采样分析（默认采样 pandas_analyze_sample=1000 行），若类型推断不准，可用 SET GLOBAL pandas_analyze_sample=100000 调大采样量。

9.4.2 从查询结果回到 DataFrame

把 SQL 查询结果取回为 DataFrame 是常见的“出”方向，依赖 9.2 节已经介绍过的物化方法：

```
import duckdb
import pandas as pd
```

```
con = duckdb.connect()
con.execute("CREATE TABLE t AS SELECT range AS i FROM range(5)")
```

```
df = con.execute("SELECT i, i * i AS square FROM t").fetchdf()
print(type(df))    # <class 'pandas.core.frame.DataFrame'>
```

9.4.3 把 DataFrame 写回 DuckDB

“DataFrame → DuckDB” 有两条常用落地路径。其一，先 register 注册为视图，再用 CREATE TABLE ... AS SELECT 持久化：

```
import duckdb
import pandas as pd

con = duckdb.connect()
df = pd.DataFrame({"k": [1, 2], "v": ["a", "b"]})

con.register("my_view", df)                # 注册为虚拟表（只读视图）
con.execute("CREATE TABLE persisted AS SELECT * FROM my_view")
```

其二，用关系 API 的 from_df() 拿到 Relation 后，直接 create 建表或 insert_into 写入已存在的表——这条路径语义清晰，且与查询闭环：

```
rel = con.from_df(df)                      # 得到 DuckDBPyRelation
rel.create("from_rel")                    # 新建表
# 或写入已存在的表：
# rel.insert_into("existing_table")
```

9.4.4 与 Arrow 的零拷贝

DuckDB 与 Apache Arrow 的深度集成是其与大数据生态互通的基石。从 DuckDB 取出 Arrow 表、或把 Arrow 表交给 DuckDB 扫描，都尽量走**零拷贝 (zero-copy)** 路径——内存中的列式缓冲被直接共享，省去序列化与逐行复制。对外推荐 to_arrow_reader(chunk_size) 以流式分块方式读取，避免一次性把超大结果集物化进内存。

```
import duckdb
import pyarrow as pa

con = duckdb.connect()
reader = con.execute("SELECT range AS i FROM range(1000)").to_arrow_reader(2048)
for batch in reader:
    print(type(batch))    # pyarrow.RecordBatch
```

9.5 Appender 批量写入与性能

9.5.1 为什么不用逐行 INSERT

写入大量数据时，最慢的做法是用预处理语句逐行 INSERT，或等价地用 executemany 灌入上百万条记录。每一条记录的写入都要经历语句解析、绑定、事务提交的完整链路，开销被重复放大。DuckDB 文档多次强调：**大量数据务必走 Appender 或 COPY (readers)**。

9.5.2 使用 Appender

Appender 是 DuckDB 批量写入的**最快路径**。其核心思路是：建立 Appender 后，逐行追加 (append) 各列的值，行满后结束该行 (end row)，最后刷新 (flush) 或关闭 (close) 以落盘。在 Python 中，这一机制被封装为连接对象的 append 方法——把一个完整 DataFrame 追加进已存在的表：

```
import duckdb
import pandas as pd
```

```

con = duckdb.connect()
con.execute("CREATE TABLE logs(id INTEGER, msg VARCHAR)")

# 待写入的数据
new_rows = pd.DataFrame({"id": [1, 2, 3], "msg": ["a", "b", "c"]})

# Appender 批量写入 ( 最快路径 ), by_name=True 按列名对齐
con.append("logs", new_rows, by_name=True)

print(con.execute("SELECT count(*) FROM logs").fetchone())

append(table_name, df, by_name=False) 的 by_name 参数控制按位置还是按列名对齐列——
列顺序与表不一致时, 务必设 by_name=True, 否则数据会错位落列。从关系 API 一侧, 亦可用 Duck-
DBPyRelation 的 insert / insert_into 完成同样语义的追加, 其底层同样借助 Appender 避免逐语
句开销。无论走哪条封装, 关键认知一致: Appender 把“逐行 INSERT”的解析开销摊薄, 仅在缓冲满或
显式关闭时才触发一次落盘。

当数据已经以文件形式存在 ( Parquet、CSV ), 与其先读进 DataFrame 再 append, 不如直接用 COPY
... FROM 或 read_parquet/read_csv 这类 readers 让 DuckDB 自身完成摄入, 路径更短、内存占用
更平滑。例如把一批 Parquet 直接追加进表:

con.execute("COPY logs FROM 'logs_*.parquet' (FORMAT parquet)")

COPY 让 DuckDB 以内置的向量化读取器直接解析文件、分批落库, 绕开了 DataFrame 中转与逐语句开
销, 是文件来源数据摄入的推荐做法。

```

9.6 Jupyter 与 SQLAlchemy

9.6.1 在 Jupyter 中使用 DuckDB

DuckDB 的 Python 客户端无需额外配置即可在 Jupyter 中直接使用。若要获得更顺手的笔记本体验, 可借助社区生态:

- **jupysql**: 把 Jupyter 代码单元变为 SQL 单元;
- **pandas / matplotlib**: 结果可视化与下游分析;
- **duckdb-engine**: DuckDB 的 SQLAlchemy 驱动 (见下)。

原生连接方式下, 先加载 jupysql 扩展并指向一条 DuckDB 连接:

```

import duckdb
import pandas as pd
%load_ext sql
conn = duckdb.connect()
%sql conn --alias duckdb

```

此后 %sql 用于单行查询、%%sql 用于整个单元做 SQL 单元, 结果默认以 pandas DataFrame 呈现; << 可把结果赋值给 Python 变量:

```
%sql res << SELECT 'Off and flying!' AS a_duckdb_column;
```

若希望 %sql 的结果默认以 pandas DataFrame 呈现 (而非 ResultSet), 可在笔记本开头配置 jupysql 的魔法选项:

```
%config SqlMagic.autopandas = True
```

9.6.2 通过 SQLAlchemy 连接 (duckdb-engine)

duckdb-engine (由社区维护的 SQLAlchemy driver) 让 SQLAlchemy 应用能用统一接口连接 DuckDB。无需显式 import duckdb_engine, jupysql 会依据连接字符串自动探测驱动。支持三种目标：

```
%sql duckdb:///memory:          -- 内存库
%sql duckdb:///default:         -- 默认全局连接 (与 duckdb.sql 共享)
%sql duckdb:///path/to/file.db -- 文件路径持久库
```

借助 duckdb-engine, 既有的 SQLAlchemy 代码 (ORM、连接池、metadata 反射等) 几乎无需改动即可迁移到 DuckDB 之上。

9.6.3 marimo: 响应式笔记本

marimo 是与 DuckDB Python 客户端深度集成的开源**响应式笔记本 (reactive notebook)**。与传统笔记本不同, 当某个单元被运行或与界面元素交互时, marimo 会自动 (或惰性) 重算受影响的下游单元, 始终保持代码与输出一致。它的另一特点是把笔记本存为纯 Python 文件, 便于 git 版本管理与作为脚本运行。

安装与启动教程：

```
pip install "marimo[sql]"      # 或 uv add "marimo[sql]"
marimo tutorial sql            # 运行交互式 SQL 教程
```

marimo 的 SQL 单元在底层被序列化为标准的 Python 代码 `mo.sql()`, 因此不需要特殊语法或魔法命令：

```
import marimo as mo
df = mo.sql(f"SELECT 'Off and flying!' AS a_duckdb_column")
```

SQL 语句以 f-string 书写, 可用 `{}` 把 Python 变量 (包括滑块等 UI 元素的值) 插值进查询, 构成数据依赖图的一部分。若 SQL 中出现用户输入, 须自行做好净化以防注入。要接入自定义连接, 只需把 `duckdb.connect()` 创建的连接赋给一个 Python 变量, marimo 会自动在 SQL 单元的连接下拉框中发现它：

```
import duckdb
conn = duckdb.connect("path/to/my.duckdb.db") # marimo 自动识别
```

配合 UI 元素, 可写出随参数联动的响应式查询——拖动滑块时, 依赖该参数的 SQL 单元会被标记为“待刷新”, 在 lazy 运行模式下由用户决定何时重算, 从而从容处理昂贵查询与大型数据集。

9.7 本章小结

本章沿着“连接 → 计算 → 交换 → 写入 → 集成”的链路, 梳理了 DuckDB Python 客户端的核心机制：

- **连接的两种形态**：`connect()` 既能开内存库也能开持久单文件库；显式连接优于模块级全局连接，多线程务必各建连接，`cursor()` 不能带来并发。
- **关系 API 的惰性价值**：DuckDBPyRelation 是查询的符号化表示，惰性求值让 `filter/project/limit` 等被合并下推，物化时才真正计算；取数出口 `df()/pl()/arrow()/fetchnumpy()` 分别对应 pandas、Polars、Arrow 与 NumPy。
- **参数化查询防注入**：`execute()` 配合 `?/$1` 占位符与字典命名参数，是安全传参的唯一正解；`executemany` 只适合小批量。
- **replacement scan 与零拷贝**：DataFrame/Arrow 可直接当表查询，免去注册；与 Arrow 的互操作尽量走零拷贝，并推荐 `to_arrow_reader` 流式读取。DataFrame 写回可用 `register+CREATE TABLE` 或关系 API 的 `from_df().create().insert_into()`。
- **Appender 是写入最快路径**：`con.append(table, df, by_name=True)` 底层借助 Appender 摊薄逐行写入开销，海量数据应弃用逐行 INSERT 与 `executemany`，改走 Appender 或 COPY/readers。
- **生态集成**：Jupyter 中可用 jupysql 魔法命令，duckdb-engine 提供 SQLAlchemy 连接串 (`duckdb:///memory:` 等)；marimo 则以响应式、纯 Python 文件的形式，把 SQL 与 Python 无缝编织进可复现的分析流。

下一章将走出 Python 单进程，进入 DuckDB 在更大工程场景中的扩展与集成。

第 10 章 其他客户端

DuckDB 的进程内 (in-process) 本质，决定了它交付给使用者的形态是一组被编译进宿主程序的库 (library)，而非一个需要单独运维的服务器进程。前几章已围绕 CLI (第 2 章) 与 Python 客户端 (第 9 章) 展开，二者覆盖了绝大多数日常交互场景。但 DuckDB 的客户端生态远不止于此：要在 C/C++ 程序里把引擎当作嵌入式组件链接进来，要在 Node.js 服务、ODBC 桥接层或浏览器中复用同一份数据，乃至用 R、Go、Rust、Julia 等语言驱动分析任务，都需要各自的客户端 API (官方称 “drivers”)。这些客户端共享同一套 SQL 语法、同一套磁盘数据库格式 (on-disk database format)，扩展 (extensions) 也基本可移植——Wasm 是少数例外。本章逐一拆解除 CLI 与 Python 之外的其余客户端，并明确它们各自在官方支持等级 (support tier) 中的落位。

10.1 CLI 进阶

第 2 章已介绍 CLI 的启动方式、基本点命令 (dot command，如 .tables、.schema、.exit) 与交互式查询。本节把重点转向两个生产环境常用的方向：用点命令控制输出格式，以及把 CLI 当作批处理工具编排进脚本流水线。

10.1.1 输出格式：.mode 的 csv / json / markdown

CLI 默认以 duckbox 模式呈现结果 (一种功能最丰富的 Unicode 方框表格)。借助 .mode 可切换输出形态，便于把结果直接喂给其他程序。三种在集成场景中最常用的格式如下。

csv 模式输出逗号分隔值，可与电子表格或下游 ETL 工具无缝衔接；配合 .separator "|" 还能改成分隔符 (如管道符)：

```
duckdb my.db
.mode csv
SELECT id, name FROM users;
-- 1,Alice
-- 2,Bob
```

json 模式输出一个 JSON 数组，每个元素是一行记录的键值对，适合被程序解析：

```
duckdb my.db
.mode json
SELECT id, name FROM users;
-- [{"id":1,"name":"Alice"}, {"id":2,"name":"Bob"}]
```

markdown 模式输出 GitHub 风格的 Markdown 表格，常用于把查询结果直接贴进文档或笔记：

```
duckdb my.db
.mode markdown
SELECT id, name FROM users;
-- | id | name |
-- |----|-----|
-- | 1 | Alice |
-- | 2 | Bob   |
```

.mode 还支持 ascii、box、column、html、insert TABLE、jsonlines (NDJSON)、latex、line、list、table、tabs、tcl、trash (无输出) 等。命令行启动阶段也可直接以参数指定格式，例如 duckdb -csv -c 'SELECT 42 AS hello'，且参数按出现顺序逐一生效，因此可在同一条命令里用 -json 切换后续查询的格式。

10.1.2 批量脚本：.read 与重定向

把查询固化成脚本文件后，CLI 提供了多条执行路径。点命令 `.read FILE` 从文件读取并执行其中的 SQL 与点命令，是最直观的一种：

```
duckdb my.db
.read setup.sql
```

`setup.sql` 内部可以混合 SQL 与 `.mode`、`.import` 等点命令，例如：

```
-- setup.sql
.mode markdown
.import --csv raw.csv staging
SELECT count(*) FROM staging;
```

若脚本仅含 SQL，也可借助启动参数一次性运行后退出：

```
duckdb my.db -f setup.sql
```

更彻底的非交互用法是把脚本经标准输入（stdin）灌入，或把查询作为 `-c` 参数传入：

```
duckdb < select_example.sql
duckdb :memory: "SELECT 42 AS the_answer"
```

CLI 同样能接入 Unix 管道（pipeline），把上游程序的输出当作文件来读，把结果写回标准输出（stdout）：

```
cat test.csv | duckdb -c "SELECT * FROM read_csv('/dev/stdin')"
duckdb -c "COPY (SELECT 1) TO '/dev/stdout' (FORMAT CSV)"
```

这类写法让 CLI 成为数据流水线里一个轻量、无依赖的环节，既可作为临时的查询入口，也可嵌入定时任务或 CI 脚本。

10.2 C / C++ API

把 DuckDB 嵌入到自有系统中，最直接的方式是 C API。它对应 Primary（一级）支持层级，是稳定且有长期兼容承诺的接口。

10.2.1 C API (libduckdb)

C API 的头文件为 `duckdb.h`，整体模拟 SQLite 的 C API 风格，甚至提供了 SQLite API 的封装（wrapper），使得已有 SQLite 程序只需重新链接即可迁移。打开与关闭数据库的四个核心函数构成完整生命周期：

```
duckdb_database db;
duckdb_connection con;

duckdb_open(NULL, &db);           /* NULL 或 ":memory:" 打开内存库 */
duckdb_connect(db, &con);
/* ... 在此执行查询 ... */
duckdb_disconnect(&con);
duckdb_close(&db);                /* 必须显式关闭，避免句柄与内存泄漏 */
```

`duckdb_query` 执行一条 SQL 并把结果物化（materialize）到 `duckdb_result` 中；无论成败，都必须调用 `duckdb_destroy_result` 释放，以回收内部字符串与缓冲区。取数的高性能路径是 `duckdb_fetch_chunk`，它直接返回原生列式结构 `DataChunk`，与 DuckDB 内部的向量化（vectorized）执行自然衔接——文档中的 `duckdb_value` 便捷函数虽更易用，但已废弃且因类型/边界检查而偏慢。

参数化查询走 prepared statement 路径：

```

duckdb_prepared_statement stmt;
duckdb_prepare(con, "INSERT INTO integers VALUES ($1, $2)", &stmt);
duckdb_bind_int32(stmt, 1, 42); /* 参数索引从 1 开始 */
duckdb_execute_prepared(stmt, NULL);
duckdb_destroy_prepare(&stmt);

```

批量写入最快的路径是 Appender，它对行式追加做了极致优化：

```

duckdb_appender appender;
duckdb_appender_create(con, NULL, "people", &appender);
duckdb_append_int32(appender, 1);
duckdb_append_varchar(appender, "Mark");
duckdb_appender_end_row(appender); /* 逐行提交 */
duckdb_appender_destroy(&appender); /* 刷新缓冲并释放，务必调用 */

```

Appender 提供覆盖全类型的 duckdb_append_* 函数族，并支持 duckdb_appender_create_ext (按 catalog.schema.table 定位)、duckdb_appender_create_query (对 INSERT/DELETE/UPDATE/MERGE 追加) 与 duckdb_append_data_chunk (直接追加预填充的 chunk)。同进程重复打开同一文件时，可用 duckdb_create_instance_cache / duckdb_get_or_create_from_cache 复用实例缓存，避免重复初始化的开销。

10.2.2 C++ API (内部接口，不稳定)

C++ API 的头文件为 duckdb.hpp，以 DuckDB (实例)、Connection、QueryResult 等类封装了更贴近 C++ 习惯的用法：

```

DuckDB db(nullptr); /* nullptr 打开内存库 */
Connection con(db);
auto result = con.Query("SELECT * FROM integers");
if (result->HasError()) {
    /* 通过 GetErrorType / GetErrorObject 检视错误 */
}

```

但官方明确将 C++ API 归入 Secondary (二级) 并标注为内部接口：它**不保证稳定**，可能在不另行通知的情况下改动。任何面向生产的程序都应坚持使用 C API；C++ API 仅适合实验性代码或紧跟特定 DuckDB 版本的内部工具。

10.3 Node.js、ODBC、Wasm

这三类客户端面向不同的集成目标：Node.js 服务化、面向既有 BI/报表系统的 ODBC 桥接、以及浏览器内的纯前端分析。

10.3.1 Node.js (@duckdb/node-api)

Node.js 客户端属于 Primary 层级，主包 @duckdb/node-api 提供高层应用 API，底层 @duckdb/node-bindings 紧贴 C API。建立实例与连接的写法如下：

```

import { DuckDBConnection, DuckDBInstance } from '@duckdb/node-api';

const connection = await DuckDBConnection.create(); // 默认实例
const instance = await DuckDBInstance.create(':memory:'); // 或 'my.duckdb'、带 {threads:'4'}
const conn = await instance.connect();
conn.disconnectSync(); // 或 closeSync()

```

同进程内若需多个实例指向同一文件，可使用 Instance Cache 复用：DuckDBInstance.fromCache('my.duckdb') 或自建 DuckDBInstanceCache。查询侧支持 run (执行不取行)、runAndRead/runAndReadAll、stream/streamAndRead* 以及 prepare + 分步求值；取结果既能按行 (reader.getRows())、也能按

chunk(result.fetchChunk()/fetchAllChunks() , 再经 chunk.getColumnVector(i).getItem(j) 访问)。批量写入沿用 Appender 思路：

```
const appender = await connection.createAppender('tbl');
appender.appendInteger(1);
appender.appendVarchar('duck');
appender.endRow();
appender.flushSync(); // 或 closeSync()
```

10.3.2 ODBC

ODBC 驱动遵循 **ODBC 3.0 Core Interface Conformance**，跨平台（Linux / macOS / Windows）可用。它通过 Driver Manager（如 unixODBC）按 DSN（Data Source Name，数据源名）定位具体驱动，因此必须正确配置两个文件。

odbc.ini（或用户级 .odbc.ini）定义 DSN：

```
[DuckDB]
Driver = DuckDB Driver
Database = :memory:
access_mode = read_only
```

odbcinst.ini（或 .odbcinst.ini）登记已安装的驱动：

```
[ODBC]
Trace = yes
TraceFile = /tmp/odbctrace
[DuckDB Driver]
Driver = /path/to/libduckdb_odbc.dylib
```

应用只需引用 DSN 名称（上例的 DuckDB），Driver Manager 便据此找到对应的 libduckdb_odbc 动态库。各平台的路径与 Driver Manager 安装细节略有差异，需参考对应平台的专用说明。

10.3.3 DuckDB-Wasm

DuckDB 可被编译为 WebAssembly（Wasm），从而运行在任意浏览器或边缘设备。核心包为 @duckdb/duckdb-wasm（JS + WASM 库）与 @duckdb/duckdb-wasm-shell（Web shell），在线体验地址为 shell.duckdb.org。

实例化（instantiation）依赖所选的 bundle。DuckDB-Wasm 提供 **mvp** 与 **eh** 两种包：mvp 对应基础的 WASM 特性集，eh（exception handling，异常处理）则在支持该特性的环境下提供更完整的异常语义。通过 duckdb.selectBundle(...) 选定后，交由 AsyncDuckDB(logger, worker) 与 db.instantiate(...) 完成加载。查询既可物化取回，也可流式遍历 chunk：

```
const conn = await db.connect();
await conn.query(`SELECT * FROM generate_series(1, 100) t(v)`);
for await (const batch of await conn.send(`SELECT * FROM generate_series(1, 100)`)) {
  /* 逐 batch 处理 Arrow 数据 */
}
const stmt = await conn.prepare(`SELECT v + ? FROM generate_series(0, 10000) t(v)`);
await stmt.query(234);
```

Wasm 版本有两条硬限制需时刻牢记：默认**单线程**（single-threaded）执行，无法利用多核；内存受 WASM 上限约束，容量封顶 **4 GB**，部分浏览器还会施加更严格的限制。因此 Wasm 适合中等规模的交互式分析与前端嵌入，而非替代服务端的大规模 OLAP。

10.4 第三方客户端

DuckDB 的客户端版图由一套明确的三级支持等级 (support tier) 组织。厘清层级, 才能判断某语言绑定在排期与稳定性上的预期:

- **Primary (一级):** 最先获得新特性, 受社区支持 (community support)。包括 C、CLI、Java/JDBC、Go、Node.js、ODBC、Python、R、Rust、WebAssembly。
- **Secondary (二级):** 会获得新特性, 但**不受社区支持**。包括 ADBC (Arrow)、C# (.NET)、C++。
- **Tertiary (三级):** 无支持保证, 由社区维护。包括 Julia、Swift、Dart、PHP 等。

需要特别说明: R、Go、Rust、Java/JDBC 虽常被泛称为“第三方语言绑定”, 但官方将其归入 **Primary 层级**并受社区支持, 并非 Tertiary。本节的“第三方”取其宽泛含义——即除 CLI 与 Python 之外的其余语言生态; 其中真正属于 Tertiary 层级、无官方支持保证的, 是 Julia、Swift、Dart、PHP 等。

10.4.1 Primary 层级的语言绑定

R (Primary): 经 `install.packages("duckdb")` 安装, 遵循 DBI 接口, `dbplyr / duckplyr` 兼容 `dplyr` 语法; `dbWriteTable` 写表, `duckdb_register` 注册虚拟表, `memory_limit` 可限定 DuckDB 自身内存。

Go (Primary): 包 `duckdb-go` 经标准 `database/sql` 接口接入, 用 `sql.Open("duckdb", "")` 建立连接, 并提供 `NewAppenderFromConn` 做高效批量写入。

Rust (Primary): `duckdb-rs` 是基于 C API 的易用封装, `Connection::open_in_memory()` 打开内存库, `conn.execute / conn.prepare + query_map` 取结果, 亦支持 `conn.appender("foo")` 做 Appender 写入。

Java / JDBC (Primary): Maven 依赖 `org.duckdb:duckdb_jdbc`, `DriverManager.getConnection("jdbc:duckdb:duckdb:/path 连持久库, ?duckdb.read_only=true 只读");` 支持 Arrow 导入导出、`jdbc_stream_results=true` 流式结果、Appender 与 Batch Writer, `DuckDBConnection#duplicate()` 可低成本克隆连接。

10.4.2 Tertiary 层级与社区维护的绑定

Tertiary 层级由社区维护、无支持保证。代表性成员包括:

- **Julia (DuckDB.jl):** 虽由 DuckDB 团队维护, 官方仍将其列入 Tertiary; `Pkg.add("DuckDB")` 后可用 `DBInterface.connect(DuckDB.DB, ":memory:")` 连接, 并提供 `DuckDB.register_data_frame` 与 Appender API 高效写入。
- **Swift (duckdb-swift):** `Database(store: .inMemory)` 或 `.file(at:)` 建库, `database.connect()` 取连接, 结果经 `result[i].cast(to:)` 转为原生类型。
- **Dart (dart_duckdb, TigerEye 维护):** `flutter pub add dart_duckdb` 后, `duckdb.open(":memory:")` → `duckdb.connect(db)` → `connection.query(...).fetchAll()` → `dispose()`, 每连接自动管理独立后台 `isolate` 以支持并发。

完整的 Tertiary 列表还包括 Common Lisp、Crystal、Elixir (Duckdbex / QuackDB)、Erlang、Haskell、Perl、PHP、PHP(PDO)、Pyodide、Raku、Ruby、Scala、Zig 等。无论哪一层级, 所有客户端都遵循同一套跨客户端一致性约定: `:memory:` 或 `NULL` 开内存库、文件路径开持久库; 多进程并发读须 `read_only`; **Appender 始终是批量写入的最快路径** (prepared statement 的 `executemany / addBatch` 仅适合小量数据); Apache Arrow 在 Python、Java、Node、Wasm 间深度互通。

10.5 本章小结

本章 surveyed CLI 之外的 DuckDB 客户端生态, 要点可归纳为四组:

- **CLI 进阶:** `.mode csv/json/markdown` 把结果导出为可机读或文档友好的格式; `.read、-f、stdin` 重定向与 Unix 管道让 CLI 成为批处理流水线的轻量环节, 与第 2 章的基础交互形成互补。

- **C/C++ API** : C API (`duckdb_open` / `duckdb_query` / `duckdb_prepare` / `duckdb_appender`) 属 Primary 层级、稳定可用，是嵌入自有系统的首选；C++ API 仅为内部接口、随时可能变动，生产代码不应依赖。
- **Node.js / ODBC / Wasm** : Node.js 经 `@duckdb/node-api` 提供实例缓存、Appender 与 chunk 流式取数；ODBC 驱动遵循 3.0 规范，须靠 `odbc.ini` / `odbcinst.ini` 的 DSN 配置接驳；DuckDB-Wasm 有 `mvp` / `eh` 两种包，但受单线程与 4 GB 内存上限约束。
- **第三方客户端** : R、Go、Rust、Java/JDBC 实为 Primary 层级并受社区支持；Julia、Swift、Dart、PHP 等才是 Tertiary 层级、由社区维护。无论层级，Appender 都是批量写入的最快路径，且所有客户端共享同一 SQL 语法、磁盘格式与 Arrow 互操作能力。

第 11 章 与外部数据库集成

DuckDB 以进程内 (in-process) 分析引擎立足，但真实数据工程里，业务系统往往运行在独立的数据库之上：PostgreSQL 承载交易与业务元数据，MySQL 支撑高并发在线服务，SQLite 嵌在应用与移动端本地存储。把这类外部系统直接纳入 DuckDB 的分析视野、免去向中间文件搬运，正是数据库集成 (database integration) 一节的价值所在。DuckDB 借助一组 scanner 扩展——`postgres_scanner`、`sqlite_scanner`、`mysql_scanner`——把外部数据库映射为本地的 catalog (目录)，从而用统一的标准 SQL 读写远端表。本章先拆解 ATTACH 机制与集成模式，再分别落到 PostgreSQL、SQLite、MySQL 三条集成路径，并说明各自的关键函数、凭据管理与注意事项。

11.1 ATTACH 机制与数据库集成模式

ATTACH 是 DuckDB 纳入外部数据源的统一入口。其语义是把一个外部存储“挂载”为当前实例中的一个 catalog (目录)——catalog 是 DuckDB 命名空间的最高层级，默认的本地库名为 `main`，每挂载一个外部源便获得一个别名 catalog。挂载完成后，外部表以三段式 `catalog.schema.table` 寻址，与本地表在写法上无任何区别。

DuckDB 把外部数据库集成归类为“附加数据库 (attached database)”能力。对原生 DuckDB 文件，直接 `ATTACH 'file.db' AS db`；即可；对外部关系型数据库，则需对应扩展在 ATTACH 时声明 TYPE：

```
-- 把 PostgreSQL 库挂载为 catalog pg
ATTACH 'host=localhost port=5432 dbname=mydb' AS pg (TYPE postgres);
-- 把 SQLite 文件挂载为 catalog sq
ATTACH 'test.db' AS sq (TYPE sqlite);
-- 把 MySQL 实例挂载为 catalog my (只读)
ATTACH 'host=localhost user=root port=0 database=mysqlscanner' AS my (TYPE mysql, READ_ONLY);
```

TYPE `postgres` / TYPE `sqlite` / TYPE `mysql` 分别由 `postgres`、`sqlite`、`mysql` 三个扫描扩展 (scanner extension) 提供实现。这组扩展多数支持首次使用时自动加载 (autoload)，亦可显式 `INSTALL` 后 `LOAD`。由于扩展二进制与具体 DuckDB 版本、平台严格绑定，跨版本升级须重新安装匹配构建，必要时用 `FORCE INSTALL` 拉取对应版本。

只读挂载通过 ATTACH 的 `READ_ONLY` 选项实现，如上方 MySQL 示例。只读挂载既保护源库不被意外写入，也省去 scanner 在远端建立写会话的开销，是做分析取数时的推荐默认。

集成模式上，官方给出明确优先序：首选 scanner 扩展直接 ATTACH 远端库；若源系统不支持或不便暴露连接，则导出为 Parquet/CSV 再由 DuckDB 加载；再不行才用各语言客户端的 appender 逐批写入。应避免循环逐行插入——即便借助预处理语句，只要数据量不小（例如超过十万行）就不应采用。

一个与事务模型相关的约束值得强调：DuckDB 采用 MVCC 快照隔离，单笔事务只允许写入一个 attached 数据库；当两个事务同时写入同一远端库而发生写写冲突时，乐观并发控制 (optimistic concurrency control) 会使一方失败，调用方需重试。这点在把分析结果回写 PostgreSQL/MySQL 时须纳入错误处理。

scanner 在挂载时会缓存远端 schema (模式) 以提升重复查询效率。当远端表结构在外部被修改后，可用各扩展提供的 `clear_cache` 函数 (如 `postgres_clear_cache()`) 强制失效本地缓存。

11.2 PostgreSQL 集成

PostgreSQL 集成由 postgres 扩展提供，其别名即 postgres_scanner。该扩展支持自动加载，但显式加载更利于在脚本中控制初始化：

```
INSTALL postgres;  
LOAD postgres;
```

挂载整库后用三段式寻址读取：

```
ATTACH 'host=localhost port=5432 dbname=mydb' AS pg (TYPE postgres);  
SELECT * FROM pg.public.orders LIMIT 10;
```

若只需临时探查某一张表，无需挂载整库，可用 postgres_scan 直接扫描单表：

```
SELECT * FROM postgres_scan('host=localhost port=5432 dbname=mydb', 'public', 'orders');
```

ATTACH 使用的连接串为 PostgreSQL 风格 {key}={value} 键值对，可携带 host、port、user、password、dbname、passfile 等字段。

除标准 SQL 读写外，扩展暴露一组在 PostgreSQL 侧执行的辅助函数：postgres_query(attached_db, query) 在远端执行只读查询并返回结果集；postgres_execute(attached_db, query) 在远端执行任意语句，含 DDL/DML；postgres_clear_cache() 清除 schema 缓存。例如需要在 PostgreSQL 侧建索引而不希望经过 DuckDB 计划器时：

```
CALL postgres_execute('pg', 'CREATE INDEX idx_orders_cust ON orders(customer_id)');
```

连接复用由连接池（connection pool）管理，可配置池大小、空闲超时、最大存活时间与获取模式（force/wait/try），降低高频查询下的握手成本。

RDS 与 Aurora 的 IAM 认证

访问托管在 Amazon RDS / Aurora 的 PostgreSQL 时，用户名密码可用短期 IAM 令牌替代。做法是由 aws 扩展配合 postgres 扩展：aws 扩展借助 AWS credential chain 生成 IAM 令牌作为密码，并在过期前自动刷新，无需在连接串中硬编码凭据。

```
INSTALL aws; LOAD aws;  
CREATE SECRET aws_rds_secret (  
    TYPE rds, PROVIDER credential_chain, CHAIN 'env;sso',  
    REGION 'eu-west-1', RDS_USER 'postgres',  
    RDS_HOST 'database-1.xxxx.eu-west-1.rds.amazonaws.com', RDS_PORT '5432'  
);  
CREATE SECRET pg_rds_secret (  
    TYPE postgres, HOST 'database-1.xxxx.eu-west-1.rds.amazonaws.com',  
    PORT '5432', USER 'postgres', DATABASE 'postgres',  
    SSLMODE require, AWS_RDS_SECRET aws_rds_secret  
);  
ATTACH '' AS rds_db (TYPE postgres, SECRET pg_rds_secret);  
SELECT * FROM rds_db.public.tbl_name;
```

此处 TYPE rds 的 secret 持有 RDS 实例标识与 IAM 主体，TYPE postgres 的 secret 通过 AWS_RDS_SECRET 引用前者，挂载时由 SECRET 子句关联。令牌刷新对调用方透明，连接串留空（ATTACH ''）即可。

v1.5.5 schema 变化自动检测

v1.5.5 引入 pg_staleness_query_enabled 配置项（BOOLEAN，默认 FALSE）。启用后，DuckDB 可周期性探查缓存的 PostgreSQL schema 是否过期，过期则自动失效本地缓存，省去手动调用 postgres_clear_cache()。默认关闭以保持向后兼容：

```
SET pg_staleness_query_enabled = true;
```

11.3 SQLite 集成

SQLite 集成由 `sqlite` 扩展提供，别名 `sqlite_scanner` / `sqlite3`，支持自动加载。挂载一个 SQLite 数据库文件即把它作为 `catalog` 暴露：

```
INSTALL sqlite;  
LOAD sqlite;  
ATTACH 'test.db' AS sq (TYPE sqlite);  
SELECT * FROM sq.main.users;
```

SQLite 默认 `schema` (模式) 名为 `main`，故三段式寻址写作 `sq.main.users`。单表快速探查可用 `sqlite_scan`：

```
SELECT * FROM sqlite_scan('test.db', 'users');
```

SQLite 采用存储类 (storage class) 而非严格列类型，其 `INTEGER/REAL/TEXT/BLOB/NULL` 按类型亲和 (type affinity) 规则映射到 DuckDB 类型。当文件存在脏数据、列内混有非预期类型时，可开启 `sqlite_all_varchar` 选项把所有列读成 `VARCHAR`，先把数据原样取回再做清洗，避免加载因类型冲突中断：

```
ATTACH 'test.db' AS sq (TYPE sqlite, sqlite_all_varchar true);
```

该扩展亦兼容 `Turso` (libSQL) 数据库文件，可用于读取边缘与移动端落地的 SQLite 变体。

11.4 MySQL 集成

MySQL 集成由 `mysql` 扩展提供，别名 `mysql_scanner`，支持自动加载。挂载语法与 PostgreSQL 对称：

```
INSTALL mysql;  
LOAD mysql;  
ATTACH 'host=localhost user=root port=0 database=mysqlscanner' AS my (TYPE mysql, READ_ONLY);  
USE my;  
SHOW TABLES;
```

此处必须点明一个易错点：MySQL 集成的连接串同样是 PostgreSQL 风格 `{key}={value}` (database/host/password/port/socket/user)，而非原生 MySQL 客户端所用的 `user:password@host:port/database` 形式；其中的 `port=0` 表示沿用客户端默认端口 (3306)。混用 MySQL 风格串会导致解析失败。

辅助函数与 PostgreSQL 对称：`mysql_query(attached_db, query)` 在远端执行只读查询；`mysql_execute(attached_db, query)` 执行任意语句；`mysql_clear_cache()` 失效 schema 缓存。

类型映射方面，扩展提供若干开关以贴合 MySQL 的布尔惯例：`mysql_tinyint1_as_boolean` 把长度为 1 的 `TINYINT` 视作布尔，`mysql_bit1_as_boolean` 把 `BIT(1)` 视作布尔；连接安全由 `ssl_mode` 参数控制。更多行为可通过 `mysql_enable_filter_pushdown` (下推过滤)、`mysql_session_time_zone`、`mysql_time_as_time` 等设置项微调。

支持的操作涵盖 `CREATE TABLE`、`INSERT`、`SELECT`、`COPY` (含 `COPY FROM DATABASE`)、`UPDATE`、`DELETE`、`ALTER TABLE`、`DROP`、`CREATE VIEW`、`CREATE/DROP SCHEMA` 与事务。需留意 MySQL 的 DDL 是非事务性的：一条 `ALTER TABLE` 中途失败不会回滚，调用方需自行校验结构状态。

11.5 本章小结

本章梳理了 DuckDB 通过 `scanner` 扩展与 PostgreSQL、SQLite、MySQL 三类外部数据库集成的方法，要点归纳如下：

- ATTACH 是把外部源挂载为 catalog 的统一入口，三段式 catalog.schema.table 寻址让远端表与本地表无差别；只读场景应优先 READ_ONLY 挂载；单事务只能写入一个 attached 库，回写远端须处理乐观并发冲突的重试。
- PostgreSQL 路径用 postgres (postgres_scanner) 扩展，支持整库挂载、单表 postgres_scan 与 postgres_query/postgres_execute 辅助函数；RDS/Aurora 借助 aws 扩展的 TYPE rds_secret 实现 IAM 令牌自动刷新；v1.5.5 的 pg_staleness_query_enabled 可在 schema 变化时自动失效缓存。
- SQLite 路径用 sqlite 扩展挂载文件，注意类型亲和 (type affinity) 与 sqlite_all_varchar 脏数据兜底，并兼容 Turso/libSQL。
- MySQL 路径用 mysql 扩展，连接串务必采用 PostgreSQL 风格；mysql_tinyint1_as_boolean / mysql_bit1_as_boolean 贴合布尔惯例，且须记住 MySQL DDL 非事务性这一约束。

第 12 章 云存储与远程数据

DuckDB 作为进程内 (in-process) 分析型数据库，数据默认落在本地文件。真实的数据工程与科学计算场景中，待分析的数据往往并不在本地磁盘：它们可能存放在 Amazon S3、Google Cloud Storage 这类对象存储 (object storage) 之中，也可能分布在多台机器上的 DuckDB 实例里。DuckDB 不内置独立的网络服务进程，却提供了一套完整且统一的机制来跨越“本地”与“远程”的边界——httpfs 扩展负责把远程文件当作本地文件一样读写，CREATE SECRET 体系集中管理跨服务的访问凭证 (secret)，而 Quack 远程协议则把任意 DuckDB 实例变成可被其他实例连接的 HTTP 服务端。本章依次展开这三块能力，并给出远程读取的性能调优要点。

12.1 httpfs 扩展与对象存储

12.1.1 统一接入模型

httpfs 是 DuckDB 的 core extension，维护层级为 Primary，开箱可用且支持 autoload：当查询首次触及 s3://、https:// 等远程 scheme 时，DuckDB 会自动从官方仓库下载并加载它，无需手动 INSTALL/LOAD。该扩展的别名涵盖 http、https、s3，因此 LOAD httpfs；与 LOAD s3；等价。所有远程文件访问——无论是读取 Parquet、导出数据，还是挂载远程 DuckDB 库——都经由 httpfs 这一条路径，不存在第二套网络栈。

```
-- 多数情况下 autoload 已足够；显式加载仅用于明确意图或离线预装
INSTALL httpfs;
LOAD httpfs;
```

12.1.2 支持的对象存储后端

httpfs 把对象存储统一抽象为文件系统接口，覆盖 S3、Google Cloud Storage (GCS)、Azure Blob/Data Lake、HTTP(S)、Cloudflare R2、Fastly、Tigris 以及 S3 Express One，Iceberg 表也能直接落在 S3 上访问。各后端在 DuckDB 中以不同 URL scheme 与 secret 类型区分，但读写 API 完全一致。

S3 与 S3 互操作后端。 S3 及其兼容实现 (R2、Fastly、Tigris、S3 Express) 都通过 TYPE s3 的 secret 配置，差异主要体现在 ENDPOINT、URL_STYLE、REGION 等参数。

```
CREATE SECRET (TYPE s3, KEY_ID 'AKIA...', SECRET '...', REGION 'us-east-1');
-- 读取：DuckDB 只下载查询真正需要的列与行组
SELECT * FROM read_parquet('s3://bucket/path/file.parquet');
-- 写入：经 S3 multipart upload，分块大小与线程数可调
COPY my_table TO 's3://bucket/path/out.parquet';
```

Google Cloud Storage。 GCS 经 S3 互操作 API 接入，需使用 HMAC key 构建 TYPE gcs 的 secret。

```
CREATE SECRET (TYPE gcs, KEY_ID 'GOOG...', SECRET '...');
SELECT * FROM read_parquet('gs://bucket/path/file.parquet');
```

```
COPY my_table TO 'gs://bucket/path/out.parquet';
```

Cloudflare R2。R2 需要 ACCOUNT_ID 以拼出端点 `https://<acctid>.r2.cloudflarestorage.com`，由 TYPE `r2` 的 secret 承载。

```
CREATE SECRET (TYPE r2, KEY_ID '...', SECRET '...', ACCOUNT_ID '...');
SELECT * FROM read_parquet('r2://bucket/path/file.parquet');
```

Azure。Azure Blob Storage 与 Data Lake Storage (ADLS) 使用 `az/azure/abfss` scheme，凭据支持 `CONFIG`、`credential_chain`、`managed_identity`、`service_principal` 等 provider。

Fastly 与 Tigris。二者均为 S3 兼容存储，借助 TYPE `s3` 配合特定 ENDPOINT 与 URL_STYLE 接入。Fastly 需 URL_STYLE `'path'` 与端点 `us-east.object.fastlystorage.app`；Tigris 单端点 `fly.storage.tigris.dev` 服务所有区域，REGION 仅用于签名，可设为 `auto`。

S3 Express One。需显式指定端点 `s3express-<az>.<region>.amazonaws.com`（如 `s3express-use1-az5.us-east-1.amazonaws.com`）。性能上建议让 EC2 实例与 bucket 处于同一可用区，LDBC SF300 实测仅加载 4.1 GB Parquet 约 3.5 秒。

12.1.3 远程 DuckDB 库的只读挂载

除了文件，`httpfs` 还能直接挂载（ATTACH）位于对象存储或 HTTPS 上的 DuckDB 数据库文件，将其中的表当作本地表查询：

```
ATTACH 'https://blobs.duckdb.org/databases/stations.duckdb' AS stations_db;
SELECT count(*) FROM stations_db.stations;
```

这种远程挂载**仅支持只读**，不支持写回；S3/GCS 兼容 API 同样适用，GCS 上的 DuckDB 库也只能通过 ATTACH ...（READ_ONLY）附加。需要写回时，仍须在本地文件上完成变更，再导出到远端。

12.2 CREATE SECRET 与凭证管理

12.2.1 Secret 类型与典型参数

跨后端的访问凭证统一由 Secrets Manager 管理，通过 CREATE SECRET 声明，并在 ATTACH 或查询时按类型自动匹配。常用类型包括 `s3`、`gcs`、`r2`、`azure`、`quack` 等，核心参数有 KEY_ID、SECRET、REGION、ENDPOINT。S3 还支持 `credential_chain` provider，可串联 `config`（`~/aws/credentials`）、`env`、`sts`、`sso`、`instance`、`process`、`web_identity` 等来源，免去硬编码长期密钥。

-- 显式长期密钥

```
CREATE SECRET (TYPE s3, KEY_ID 'AKIA...', SECRET '...', REGION 'us-east-1');
```

-- 链式凭据：从配置与 STS 临时凭据获取，并跨账号 AssumeRole

```
CREATE SECRET (
  TYPE s3,
  PROVIDER credential_chain,
  CHAIN config,sts,
  ASSUME_ROLE_ARN 'arn:aws:iam::123456789012:role/my-role',
  REGION 'us-east-1'
);
```

12.2.2 持久化存储与权限

Secret 默认仅存在于当前会话，重启后失效。可声明为持久（persistent）secret，使其跨会话保留——持久 secret 以**未加密的二进制形式**存放在用户目录 `~/duckdb/stored_secrets` 下，文件权限为 600（仅属主可读写）。这一机制便于单机复用凭据，却也意味着明文密钥落盘，须确保该目录的属主与文件系统权限受控，避免在共享或多用户主机上泄露。对安全敏感的环境，更宜依赖 `credential_chain` 的临时凭据或外部密钥管理，而非长期持久化静态密钥。

12.3 远程文件读取性能建议

12.3.1 网络延迟与并行度

httpfs 采用同步 IO：每个线程在同一时刻最多发起一个 HTTP 请求。当查询需要向远端索取大量小对象（例如海量小 Parquet 文件、深层次 Hive 分区）时，单线程会串行等待网络往返，CPU 反而空闲。此时应将 threads 提升到 CPU 核数的约 2-5 倍，用更多并发请求填满网络带宽：

```
SET threads = 16; -- 假设机器为 4 核，提升到 2-5 倍
```

12.3.2 大查询的落盘 (spill to disk)

超出内存的数据集仍可处理，DuckDB 通过 spill to disk 把中间状态溢出到临时目录（默认持久化模式下的 <db>.tmp 或内存模式下的 .tmp，可用 temp_directory 修改）。GROUP BY、JOIN、ORDER BY、窗口函数这类阻塞算子（blocking operator）都支持超内存执行，但多个阻塞算子叠加或特定聚合（list()、string_agg()）仍可能 OOM。面对远程大查询，配合 SET memory_limit 与 pre-serve_insertion_order = false 能降低内存峰值。

12.3.3 下推减少传输

远程读取的最大开销是网络传输量，而 DuckDB 的向量化执行器会把多种操作下推到扫描层，从源头削减取回的数据：

- **列裁剪 (projection pushdown)**：SELECT 只列出所需列，DuckDB 只下载这些列的数据块，而非整行。
- **过滤下推 (filter pushdown)**：WHERE 谓词被下推到 parquet 扫描，结合 row group 的 zonemap 跳过不匹配的行组，并减少回传行数。
- **范围与限制下推**：LIMIT 与范围过滤能把取数规模压到最小，远程只回传命中的少量行组。
- **Hive 分区**：按常用过滤列分区，使整目录的无关分区被直接排除。

自 v1.3.0 起 DuckDB 支持远程数据缓存（duckdb_external_file_cache()），重复访问同一远端对象可免二次下载。用 EXPLAIN ANALYZE 可直观查看一次查询发起的 HTTP 请求数与实际传输字节数，据此判断下推是否生效、并行度是否需要调整。

12.4 Quack 远程协议

12.4.1 协议模型

Quack 是一项把 DuckDB 实例转变为服务器的扩展，让其他 DuckDB 实例作为客户端通过 HTTP 连接——它补上了进程内架构天生缺失的多进程并发与远程访问能力。Quack 于 2026-05-12 发布，beta 版随 DuckDB v1.5.3 经 core 仓库提供，处于实验状态，协议与函数名仍可能变化，预期在 v2.0 达到稳定。

协议本身基于明文 HTTP 或经反向代理的 HTTPS，复用标准的负载均衡、防火墙与反向代理设施，没有自定义传输层。请求与响应以 application/duckdb 媒体类型序列化，采用 DuckDB 内部的序列化原语编码——**这条序列化路径与预写日志 (WAL, Write-Ahead Log) 共用同一套代码**，使得嵌套结构、decimal、interval 等复杂类型都能无损跨线传输。握手之后，单条查询只需一次请求-响应往返；大结果通过后续 FETCH 分块流式返回，客户端可多线程并行拉取。URI 使用 quack: scheme，默认端口 **9494**。

12.4.2 服务端与客户端的启动

服务端从已有 DuckDB 会话启动，返回监听 URI、HTTP URL 以及客户端所需的 auth_token：

```
-- 默认仅绑定 localhost
CALL quack_serve('quack:localhost');
-- 监听外部地址需显式开启，并建议前置 TLS 终止的反向代理
CALL quack_serve('quack:0.0.0.0:9494', allow_other_hostname => true);
-- 停止服务
```

```
CALL quack_stop('quack:localhost');
```

客户端有两种接入方式。其一是无状态的 quack_query，无需挂载即可执行远程查询：

```
FROM quack_query('quack:localhost', 'SELECT 42', token='TOKEN');
```

其二是 ATTACH 把远程实例作为完整 catalog 挂载，远程表、DDL、写入与事务如同本地一样使用：

```
ATTACH 'quack:localhost' AS remote_db (TOKEN 'TOKEN');  
CREATE TABLE remote_db.t AS FROM range(10) r(i);  
INSERT INTO remote_db.t VALUES (42);  
FROM remote_db.t WHERE i = 42;
```

12.4.3 认证、授权与安全部署

Quack 服务端暴露底层 DuckDB 的**完整 SQL 面**（含读写），故默认采取保守策略：启动时生成随机认证 token（auth token），客户端每次连接都必须提供；服务端仅绑定 localhost，非本地主机名需显式 `allow_other_hostname => true`；服务端自身不启用 TLS（localhost 通信加 TLS 无收益），但非本地 URI 时客户端默认假设 HTTPS。

认证（authentication）与授权（authorization）是两道可插拔的钩子，均为全局作用域设置：认证函数 `quack_authentication_function`（默认 `quack_check_token`）在连接请求时校验 token；授权函数 `quack_authorization_function`（默认 `quack_nop_authorization`，放行一切）在准备请求时按查询放行与否。二者可用 SQL MACRO 或 C++/Rust/Go 等扩展注册的标量函数替换——Python UDF 因连接作用域问题不可用作回调。推荐把 token 放入 secret 而非硬编码：

```
CREATE SECRET (TYPE quack, TOKEN 'TOKEN', SCOPE 'quack:localhost');  
ATTACH 'quack:localhost' AS remote_db (TYPE quack);
```

任何非本地部署都应前置成熟的 HTTP 反向代理来终止 TLS，常见方案为 nginx 或 Caddy 配合 Let's Encrypt。nginx 侧需关注 `client_max_body_size 256M`（PREPARE/APPEND 体较大）、`proxy_read_timeout/proxy_send_timeout` 设为 600s（长查询）、`proxy_http_version 1.1` 并保持长连以维持 `quack_connection_id` 状态，以及 `proxy_buffering off`（不缓冲流式 FETCH）。

12.4.4 版本状态

Quack 当前为 beta，在生产环境使用尚不成熟，可能包含破坏性变更，预期在 DuckDB v2.0 稳定。若遇扩展行为异常，可用 `FORCE INSTALL quack`；拉取匹配当前版本的扩展。

12.5 本章小结

本章覆盖了 DuckDB 跨越本地边界的三组能力。**其一**，**httpfs 扩展**统一了所有远程文件访问：S3、GCS、Azure、HTTP(S)、Cloudflare R2、Fastly、Tigris、S3 Express 与 Iceberg 经同一网络栈读写，远程 DuckDB 库则只能以只读方式 ATTACH。**其二**，`CREATE SECRET` 集中管理跨后端凭证，支持显式密钥与 `credential_chain` 链式临时凭据；持久 secret 以未加密二进制存于 `~/.duckdb/stored_secrets`（权限 600），明文落盘须谨慎对待。远程读取的性能关键在于用并行度抵消网络延迟、用 spill to disk 容纳超内存负载、并用列裁剪/过滤/范围/LIMIT 下推与远程缓存尽量压缩传输量。**其三**，**Quack 远程协议**借助 `application/duckdb` 序列化（与 WAL 共享代码路径）与 token 认证，把 DuckDB 变成可被 quack_query 或 ATTACH 连接的 HTTP 服务，默认端口 9494；它仍处 beta，生产部署须前置 TLS 反向代理，并预期在 v2.0 稳定。

第四部分 扩展机制

第 13 章 扩展体系

DuckDB 的一大设计特征是“可扩展 (Extensible)”：Parquet、JSON、时区 (ICU)、HTTP(S) 与 S3 等基础能力本身都以扩展 (extension) 形式实现，连引擎核心也借由扩展机制向外延展。这种“内核精简、能力外挂”的格局，让 DuckDB 既能保持二进制轻量，又能按需吸纳地理空间、向量检索、湖仓格式、跨数据库挂载等丰富功能。理解扩展的加载模型、版本绑定规则与签名安全机制，是安全、稳定地使用生态的先决条件。本章系统拆解 DuckDB 的扩展体系——从安装与加载的底层机制，到版本兼容的硬性约束，再到签名安全与社区生态，为后续自定义函数与扩展开发章节铺垫基础。

13.1 扩展架构与加载模型

DuckDB 把功能划分为两类：一类是**内置扩展 (built-in / 静态链接, statically linked)**，随二进制一同发布、无法在运行时另行安装或卸载；另一类是**可安装扩展 (installable / 动态加载, dynamically loaded)**，默认不随二进制发布，需经 INSTALL 与 LOAD 两步引入。

INSTALL 的职责是从仓库下载扩展二进制、校验其签名 (signature)，并存入本地扩展目录。默认路径因版本与平台而异，组织为 `~/.duckdb/extensions/<duckdb_version>/<platform_name>/`，例如 `~/.duckdb/extensions/v1.5.5/linux_amd64/`。仅执行 INSTALL 不会把扩展加载进当前实例——它只是把二进制落到磁盘。

```
-- 从默认仓库 (core) 安装 httpfs 扩展
INSTALL httpfs;
-- 动态加载到当前实例，使其函数与类型生效
LOAD httpfs;
```

LOAD 则把已安装的扩展**动态加载 (dynamic loading)**到当前 DuckDB 实例，让扩展注册的函数、类型、表函数与索引方法生效。多数 core 扩展支持 **autoload (自动加载)**：首次被使用时，DuckDB 会透明地从官方仓库下载并加载 (需联网)，用户无需手动 INSTALL/LOAD。例如访问 `s3://`、`https://` 路径时，`httpfs` 会自行 autoload。

INSTALL 不指定来源时默认从 core 仓库拉取；也可通过 `FROM <repository>` 指定 `core_nightly` (每夜构建仓库) 或 `community` (社区仓库) 等来源。一旦用户显式 INSTALL 过某扩展，之后即使重启实例，该扩展通常已在本地缓存，autoload 不会再触发网络下载。

少数扩展因依赖外部系统库而无法 autoload，必须显式 LOAD。典型是 `spatial` (依赖 GEOS、PROJ、GDAL 等地理空间库) 与 `odbc` (依赖 unixODBC / ODBC Driver Manager)。它们的加载语句不可省略：

```
-- spatial、odbc 不会自动加载，必须显式 LOAD
LOAD spatial;
LOAD odbc;
```

两个硬性限制需要牢记。其一，已加载的扩展在当前进程生命周期内**无法卸载 (unload) 或重新加载 (reload)**——LOAD 之后便一直存在。其二，**不存在 UNINSTALL 命令**：若要移除一个扩展，只能手动删除 `~/.duckdb/extensions/...` 下的对应文件。这一约束意味着扩展的取舍须在加载前想清楚，加载后只能重启进程来“重置”。

还有一类扩展以静态链接形式存在，连运行时加载都不支持——例如 `jemalloc` (内存分配器)，它被编译进二进制以替换系统分配器，既不能用 INSTALL 下载，也不能用 LOAD 动态加载。这类扩展的开关取决于构建方式 (如 Linux 发行版自带，macOS 需 `BUILD_JEMALLOC=1` 自行构建，Windows 不可用)。

排查扩展状态时，`duckdb_extensions()` 表函数给出当前环境可用扩展的清单，含名称、是否已安装、是否已加载、版本与仓库等字段：

```
SELECT extension_name, installed, loaded, version, repository
FROM duckdb_extensions()
WHERE extension_name IN ('httpfs', 'spatial', 'odbc');
```

通过该表函数可确认某扩展是否已被 autoloader、是否已实际加载，是诊断加载问题的第一步。

13.2 版本兼容性铁律与离线安装

扩展二进制与特定的 DuckDB 版本及平台 (platform) **严格绑定**，这是整个扩展体系最不可动摇的规则。仓库按 Target DuckDB Version 组织目录，形如 v1.5.5/linux_amd64/；每一次编译都针对某一个具体版本与某一平台。用错版本（例如拿 1.4.0 的扩展去配 1.5.5 的 DuckDB）或错平台（例如 linux_amd64 的扩展跑在 osx_arm64），加载会被直接拒绝。这条**版本兼容性铁律 (version-compatibility rule)** 意味着：升级 DuckDB 之后，旧版本对应的扩展二进制一律失效，必须重新拉取匹配新版本的扩展。

常规做法是重装匹配版本。由于默认 INSTALL 发现本地已有同版本文件便会跳过，FORCE INSTALL 用于忽略本地缓存、强制重新下载——升级后或切换仓库来源时都应使用它：

```
-- 升级 DuckDB 后，强制重新拉取匹配新版本的扩展
FORCE INSTALL httpfs;
-- 也可切换仓库来源（例如从 nightly 切回官方 core）
FORCE INSTALL iceberg FROM core_nightly;
```

网络受限或完全离线的场景，可用“直接按 URL 或本地路径安装/加载”的方式绕过仓库联网：

```
-- 离线环境：加载预先下载到本地的扩展文件
INSTALL 'path/to/httpfs.duckdb_extension';
LOAD    'path/to/httpfs.duckdb_extension';

-- 也可直接指向 HTTP/S3 上的扩展文件
INSTALL 'http://example.com/path/to/extension.duckdb_extension';
LOAD    'http://example.com/path/to/extension.duckdb_extension';
```

更系统化的离线方案是指定扩展搜索目录：通过配置项 extension_directory（单个目录）或 extension_directories（多个目录、以平台分隔符分隔）指向预先下载好的扩展文件所在位置，DuckDB 会在这些目录中查找而不再尝试联网：

```
-- 指向预下载扩展所在的多个目录（unix 以冒号分隔）
SET extension_directories = '/data/duckdb_exts:/opt/duckdb_exts';
```

与 extension_directories 对应的还有单数形式的 extension_directory，用于指定单个自定义目录；extension_directories 则是以平台分隔符分隔的多个目录列表，适合把预下载文件分散在多台机器或多种存储介质的场景。两者都让 DuckDB 优先在本地目录查找扩展，从而在完全离线的 CI 流水线或内网环境中稳定复现依赖。

UPDATE EXTENSIONS；语句可尝试按当前 DuckDB 版本与仓库批量更新已安装扩展，但同样受版本绑定约束——跨大版本升级本质上仍需 FORCE INSTALL 重新获取匹配二进制。故障排查时若出现 404，多半是版本/平台不匹配或该平台没有对应构建，此时可手动用浏览器下载对应 duckdb_version/platform 下的文件再 LOAD '...'。

13.3 扩展签名与安全

DuckDB 对来自官方仓库的扩展实施**签名 (signature) 校验**。官方仓库 core 与 community 发布的扩展均经数字签名，DuckDB 在 INSTALL 下载后会校验签名，签名不符则拒绝加载。这套机制保证扩展二进制确实出自可信来源、未被篡改。

第三方或本地自行构建的扩展可能**未签名 (unsigned)**。加载此类扩展需显式开启未签名支持——但该开关**只能在启动期确定**：要么在启动前通过配置项设定，要么以 -unsigned 命令行参数启动；一旦实例已经运行，便无法在会话中途将其开启。

```
-- 启动期开启未签名扩展支持（仅可在启动时生效）
SET allow_unsigned_extensions = true;
-- 等价地通过命令行参数启动
```

```
-- duckdb -unsigned my.db
```

allow_unsigned_extensions 的“仅启动期可开”特性，把安全开关固化在进程生命周期起点，避免运行期被动态放宽从而引入风险。部分涉及本地数据的功能（如 ui 扩展的 ui_remote_url 自定义远程地址）也只有在开启该开关时才会被尊重，理由正是远程 UI 资源可能访问本地数据。

另一个相关开关 allow_community_extensions 默认值为 true，控制是否允许从社区仓库安装扩展。它的关键行为是**自我锁定 (self-lock)**：一旦在启动期将其设为 false，运行期便无法再改回 true。也就是说，关闭社区扩展支持是不可逆的会话级决定，必须在启动阶段就规划清楚。

```
-- 默认开启；一旦启动期关闭，运行期不可再开（自我锁定）
SET allow_community_extensions = false; -- 仅可在启动期设置
```

综上，扩展安全由三层保障：签名校验确保来源可信、allow_unsigned_extensions 把未签名支持锁在启动期、allow_community_extensions 的不可逆关闭防止运行期误开社区源。三者启动期一次性定调，运行期不再可放宽。

13.4 核心扩展一览

DuckDB 共有 **28 个核心扩展 (core extensions)**，按维护方分为 Primary、Secondary 与 Third-party 三档支持层级。其中 parquet 与 json 作为**内置扩展 (built-in)** 随二进制发布、开箱即用——它们不是从仓库下载，而是直接静态链接进二进制，因此既无需 INSTALL/LOAD（显式 LOAD 也不会报错）。其余核心扩展默认不随二进制发布，需经 INSTALL/LOAD 或 autoloader 引入。

autoloader 是区分扩展“要不要手动管”的核心机制。下表列出若干代表性核心扩展及其加载特征：

扩展	主要用途	autoloader	备注
parquet	Parquet 读写	内置	无需 INSTALL/LOAD
json	JSON 读写	内置	无需 INSTALL/LOAD
httpfs	HTTP/S/S3/R2/GCS/HuggingFace	是	别名 http/https/s3
postgres	挂载 PostgreSQL	是	别名 postgres_scanner
iceberg	Apache Iceberg 表	是	支持 V3 spec (v1.5.5)
icu	排序规则与时区	是	首次区域/时区操作时触发
fts	全文检索	是	类 SQLite FTS5
inet	INET (IPv4/IPv6/CIDR) 类型	是	—
sqlite	挂载 SQLite	是	别名 sqlite_scanner/sqlite3
excel	读写.xlsx	是	不支持旧版.xls
aws	AWS 凭据集成	是	支持 RDS IAM 认证
azure	Azure Blob/ADLS	是	协议 az/azure/abfss
delta	Delta Lake 读取	是	基于 delta-kernel-rs
ducklake	DuckLake 湖仓	是	DuckLake 1.0 于 2026-04 发布
spatial	地理空间 (GIS)	否	依赖 GEOS/PROJ/GDAL，须显式 LOAD
odbc	ODBC 访问任意 DBMS	否	依赖 unixODBC，须显式 LOAD

从组织方式看，核心扩展又分 **in-tree (树内)** 与 **out-of-tree (树外)** 两类：in-tree 扩展位于 DuckDB 主仓库（如 httpfs、parquet、json、postgres、iceberg、spatial），随主仓库一起版本化与发布；out-of-tree 扩展位于独立仓库（如 duckdb-avro、duckdb-mysql、duckdb-fts），独立版本化。

在支持层级上，核心扩展分为三档：Primary 档由官方主力维护，含 httpfs、icu、json、parquet；Secondary 档为其余官方维护扩展；Third-party 档由第三方维护，含 lance、motherduck、ui、vortex。不少扩展还带别名（alias），用别名 LOAD 与用原名等价——httpfs 的别名包含 http/https/s3，postgres 别名 postgres_scanner，sqlite 别名 sqlite_scanner/sqlite3，mysql 别名 mysql_scanner，motherduck 别名 md。例如 LOAD s3；与 LOAD httpfs；指向同一扩展。

扩展的稳定性分为三级，理解其差异有助于判断能否用于生产：

稳定性级别	版本号形态	说明
unstable	基于 git hash	随每次提交变化，主要用于 core_nightly 仓库
pre-release	v0.y.z	主版本仍为 0，可能不稳定
stable	vx.y.z (x > 0)	官方 core 仓库发布的扩展通常带 stable 版本号

官方 core 仓库发布的扩展通常带 stable 版本号，可放心用于生产；core_nightly 等仓库则充斥 unstable 构建，仅适合尝鲜与测试。

13.5 社区扩展生态

除官方核心扩展外，DuckDB 还设有**社区扩展仓库 (community repository, 别名 community)**，URL 为 <http://community-extensions.duckdb.org>。该仓库自 **2024 年夏**起上线，由社区（而非 DuckDB 核心团队）维护、独立版本化、以 out-of-tree 形式存在于独立仓库。

社区扩展的使用方式与核心扩展一致，只是需显式指定 FROM community：

```
INSTALL tarfs FROM community;
LOAD tarfs;
```

社区生态中有不少广为人知的成员：tarfs、avro（先作为社区扩展发布，2025 年初转正为 core extension）、quack（Quack 远程协议）、lance、vortex、unity_catalog 等。它们由不同维护方贡献，能力覆盖归档文件系统、向量格式、远程协议、AI 湖仓格式等多元场景。

需要强调的是，社区扩展**同样受版本兼容性铁律约束**——其二进制依然绑定具体 DuckDB 版本与平台，版本错配会被拒绝加载，升级后同样要用 FORCE INSTALL 重拉匹配版本。从 community 安装后若想切回官方 core，用 FORCE INSTALL ... FROM core 即可。社区扩展也需通过签名校验，因此默认纳入 DuckDB 的安全体系，除非显式开启 allow_unsigned_extensions 加载自编译的未签名版本。

13.6 本章小结

本章系统拆解了 DuckDB 的扩展体系，核心要点如下：

- **加载模型分两步**：INSTALL 从仓库下载二进制、校验签名并存入 ~/.duckdb/extensions/<版本>/<平台>/；LOAD 将其动态加载进当前实例。多数 core 扩展支持 autoload（首次使用透明下载并加载），但 spatial、odbc 因依赖外部系统库必须显式 LOAD。已加载扩展无法 unload/reload，无 UNINSTALL；jemalloc 等静态链接扩展连运行时加载都不支持。
- **版本绑定是铁律**：扩展二进制严格绑定具体 DuckDB 版本与平台，错配即被拒加载；升级后须 FORCE INSTALL 重拉匹配版本。离线场景用 INSTALL/LOAD '<url_or_path>' 或 extension_directories 指向预下载文件。
- **安全靠三层锁定**：官方扩展经签名校验；allow_unsigned_extensions 仅启动期可开、运行期不可放宽；allow_community_extensions 默认开启但一旦关闭即自我锁定、不可逆。
- **核心扩展共 28 个**，parquet/json 为内置、开箱即用；稳定性分 unstable / pre-release / stable 三级，官方 core 仓库多为 stable。
- **社区仓库 2024 年夏上线**，由社区维护、独立版本化，使用方式一致且同样受版本约束，丰富了 DuckDB 的能力边界。

掌握扩展的安装、加载、版本与安全机制后，下一章将进入自定义函数（UDF）与扩展开发，把这套体系延伸到你自己的代码之中。

第 14 章 关键扩展详解

DuckDB 的功能大多以扩展（extension）的形式交付：核心能力如 Parquet、JSON 读写，外部系统集成如 PostgreSQL、Delta Lake 挂载，乃至向量检索、全文搜索，都遵循“按需加载”的模型。这种拆分的益处是把不常用的依赖挡在默认二进制之外，让引擎保持轻量，也让各扩展能独立版本化演进。前文已交代过扩展的 INSTALL/LOAD 机制与自动加载（autoloading）规则，本章不再复述，而是把镜头对准几类最具代表性的扩展，逐一讲清它们是什么、能做什么、有哪些边界。其中 JSON 与 Spatial 属基础数据类型与函数族，Iceberg 是开放表格式接入的典型，其余常用扩展则覆盖了从本地文件到云存储、从关系数据库到向量检索的广阔场景。另需提醒，除随二进制发布的内置扩展外，所有可安装扩展的二进制都严格绑定具体的 DuckDB 版本与平台，升级引擎后须重新拉取匹配构建——这条铁律贯穿本章每个扩展的使用。

14.1 JSON 扩展

JSON 扩展是随 DuckDB 二进制一同发布的内置扩展（built-in extension），与 parquet 并列，开箱即用。它不经由扩展仓库下载，也无需执行 INSTALL/LOAD——尽管显式 LOAD json；不会报错，只是多余。该扩展为 DuckDB 引入了原生的 JSON 类型（JSON type）以及一整套读写与构造函数，是处理接口返回、埋点日志、配置快照等半结构化数据的基石。第 6 章已系统讲解 json_extract/json_extract_string 与 ->/->> 等取值函数，本节把重心放在类型本身与读入、构造两端。

JSON 类型。 DuckDB 的 JSON 类型以半结构化方式承载 JSON 文本：它不强制 schema，也不在写入时严格校验内部结构，因此适合形态不固定的数据。当一个列声明为 JSON 时，DuckDB 不会尝试把它拆解成关系型列；需要结构化视图时，应经由 read_json 的自动推断或取值函数显式投影。换言之，JSON 类型回答的是“不规则数据如何落地”，而取结构是下游的事。与把 JSON 存成 VARCHAR 相比，JSON 类型在内部采用专为半结构化数据设计的存储布局，既保留对象层级，也让取值函数能直接在其上导航、省去文本解析开销；VARCHAR 只是原样保存字符串，每次访问都要重新解析。

读取与自动推断。 read_json 与 read_json_auto 负责把 JSON 文件读入表。二者都支持 schema 自动推断（auto-detection），能依据样本记录揣测列名与类型；遇到嵌套（nested）结构时，可整体保留为 JSON 列，也可按路径下钻成标量列。line-delimited JSON（NDJSON，每行一条记录的 JSON 流）同样受支持，这使它天然契合日志与流式导出。

```
-- JSON 为内置扩展，直接使用，无需安装/加载
SELECT * FROM read_json('events.json', auto_detect = true) LIMIT 10;
```

```
-- 读取 NDJSON（每行一条 JSON 记录）
SELECT * FROM read_json_auto('logs.ndjson');
```

嵌套结构不会迫使你立即扁平化：可以把一整个对象保留为 JSON 列，待真正需要某字段时再下钻。至于输入形态，read_json 与 read_json_auto 的差异在于后者在格式判定上更主动——它会探测输入是单个 JSON 数组还是 NDJSON 流、并据此选择读取策略；read_json 则更依赖调用方显式给出的选项。对列中出现的类型不一致，DuckDB 按兼容原则归并类型，必要时以 JSON 容纳异质值，从而让推断出的 schema 不至于因个别脏记录而整体失真。

构造与序列化函数。与读取相对，扩展提供 json_object、json_array 现结构造对象与数组，并可用 to_json/json_serialize 把 DuckDB 的结构体（STRUCT）、列表（LIST）等类型序列化为 JSON 文本，便于回写外部系统。

```
-- 构造对象与数组
SELECT json_object('city', 'Beijing', 'coords', json_array(116.4, 39.9)) AS doc;

-- 把 STRUCT 序列化为 JSON
SELECT json_serialize(STRUCT('a', 1, 'b', 'x')) AS s;
```

read_json 的自动推断、json 类型的容纳能力，正是第 6 章那些取值函数得以灵活运作的底座：前者解决“如何读入”，后者解决“不规则数据如何落下”，取值函数则在其上解构访问。

14.2 Spatial

spatial 扩展提供地理空间（GIS）函数与类型，底层依赖 GEOS、PROJ、GDAL 三套外部系统库。由于这些库独立于 DuckDB 二进制、无法由 DuckDB 自行打包，spatial 不会被自动加载（autoload），每次会话都必须显式 LOAD spatial；。这是它与绝大多数 core 扩展最显眼的区别。这三套库各司其职：GEOS 提供几何算法与空间谓词（相交、包含、缓冲、距离等），PROJ 负责坐标参考系之间的投影变换，GDAL 则负责读写数十种矢量与栅格文件格式。也正因为它们是平台相关的原生库，spatial 不能像纯 SQL 扩展那样在首次使用时透明地从仓库拉取，只能由使用者显式加载。

地理类型与版本演进。自 DuckDB v1.5 起，GEOMETRY 类型已成为内置类型——意味着即便不加载 spatial 扩展，表中也能创建并存储 GEOMETRY 列；但真正的几何运算（面积、距离、投影变换等）仍由 spatial 扩展提供。这种“类型内建、运算外置”的安排，让存储与计算的职责清晰分离：持久化一份几何数据不再强制要求 GIS 函数库在场。

空间函数概览。spatial 提供大量以 ST_ 为前缀的标量函数，例如 ST_Area（面积）、ST_AsText（转 WKT 文本）、ST_Buffer（缓冲）、ST_Distance（距离）、ST_Within（包含关系）、ST_Transform（坐标参考系变换）等，另有若干聚合函数。实践上按 ST_ 前缀检索即可定位所需函数。下列示例在创建几何列后做距离计算与面积测算：

```
LOAD spatial;
```

```
CREATE TABLE pois (name VARCHAR, geom GEOMETRY);
INSERT INTO pois VALUES
  ('Beijing', ST_Point(116.40, 39.90)),
  ('Shanghai', ST_Point(121.47, 31.23));
```

```
-- 以 WKT 形式输出，并计算到原点的距离
SELECT name,
       ST_AsText(geom) AS wkt,
       ST_Distance(geom, ST_Point(116.40, 39.90)) AS dist
FROM pois;
```

```
-- 缓冲区的面积（单位取决于坐标参考系）
SELECT ST_Area(ST_Buffer(ST_Point(0, 0), 1)) AS circle_area;
```

```
-- 判定某点是否落在另一个几何体内
SELECT ST_Within(ST_Point(116.40, 39.90), ST_Buffer(ST_Point(116.40, 39.90), 10)) AS inside;
```

坐标参考系（CRS）的换算由 PROJ 承担，ST_Transform 可在不同 CRS 间投影；涉及经纬度与投影平面距离的语义差异，务必在调用前确认数据所用参考系。ST_ 前缀源自 OGC Simple Features 规范，与 PostGIS 等成熟 GIS 系统的命名一致，便于直接迁移既有查询知识；需要提醒的是，ST_Distance 在经纬度坐标下返回的数值并非真实地表距离，若需物理距离应先经 ST_Transform 投影到合适的平面参考系再计算。

R-Tree 空间索引。为加速空间范围查询，可建 R-Tree 索引：

```
CREATE INDEX idx_pois ON pois USING RTREE (geom);
```

R-Tree 节点的默认容量为 max_node_capacity = 128；rtree_index_dump 可用于诊断索引结构。

GDAL 集成。借助 GDAL，ST_Read 能读取数十种矢量与栅格格式，COPY ... FORMAT gdal 可将结果导出到 GDAL 支持的格式，从而把 DuckDB 接入既有的 GIS 工具链。

14.3 Iceberg 扩展

iceberg 扩展用于读取乃至写入 Apache Iceberg 表——一种面向大规模分析的开放表格式 (open table format)。Iceberg 表以元数据文件 (metadata.json) 描述快照、schema 与分区, 数据本身则是 Parquet 等列式文件。该扩展会在首次使用时自动加载 (autoload)。

两种访问模式。路径模式 (path-based, 只读) 直接以 iceberg_scan('path/to/metadata.json') 读取本地或 S3 上的元数据文件, 适合一次性探查; 目录模式 (catalog-based, 读写) 则通过 ATTACH ... (TYPE iceberg, ENDPOINT_TYPE ...) 接入 Iceberg REST Catalog 后读写。可接入的目录后端包括 Amazon S3 Tables、AWS Glue / SageMaker Lakehouse、Cloudflare R2、Polaris、Lakekeeper、BigLake 等。需要牢记: 路径模式只读, 写入必须经已附加的 REST Catalog。

```
INSTALL iceberg;  
LOAD iceberg;
```

```
-- 路径模式只读探查
```

```
SELECT * FROM iceberg_scan('s3://bucket/warehouse/table/metadata/v3.metadata.json') LIMIT 5
```

```
-- 目录模式读写 (以 Polaris 为例)
```

```
ATTACH 'catalog' AS iceberg_cat (TYPE iceberg, ENDPOINT_TYPE polaris, ...);
```

```
-- 常用元数据探查函数
```

```
SELECT * FROM iceberg_snapshots('s3://bucket/warehouse/table/metadata/v3.metadata.json');
```

```
SELECT * FROM iceberg_metadata('s3://bucket/warehouse/table/metadata/v3.metadata.json');
```

V3 spec 支持。v1.5.5 的 iceberg 扩展支持 Iceberg V3 spec, 涵盖 VARIANT 类型、TIMESTAMP_NS (纳秒级时间戳)、列默认值 (default values) 以及 binary deletion vectors (二进制删除向量)。这让 DuckDB 能无损读写采用最新规范的 Iceberg 表。分区变换 (partition transforms) 支持 identity、year/month/day/hour、bucket(n, col)、truncate(n, col), 可通过 PARTITIONED BY (...) 建表、ALTER TABLE ... SET PARTITIONED BY (...) 改分区。

从使用者视角, Iceberg 的价值在于快照 (snapshot) 隔离: 每张表在任意时刻都对应一个快照, 历史快照被保留, iceberg_snapshots 能列出它们, 从而支持时间旅行 (time travel) ——回到过去某个一致性视图读取数据。这也是开放表格式相比直接堆 Parquet 文件的关键优势: 元数据层统一管理 schema、分区与快照演进, 读端无需关心文件层面的物理布局。

写入限制。写入能力虽已具备, 但约束明确:

- 仅支持 positional delete (位置删除) 文件, 不支持 copy-on-write (写时复制) 删除;
- 仅采用 merge-on-read (读时合并) 语义; 若目标表将 write.update.mode/write.delete.mode 设为非 merge-on-read, 写入会失败;
- 只能作用于非排序 (non-sorted) 表。

此外, 分区表的 write.target-file-size-bytes 与 write.parquet.row-group-size-bytes 不被遵守 (会报错), 除非显式设置 ignore_target_file_size_for_partitioned_tables / ignore_row_group_size_for_partitioned_tables。GEOGRAPHY 与 UNKNOWN 类型计划于扩展 v2.0.0 支持。表属性可用 iceberg_table_properties / set_iceberg_table_properties / remove_iceberg_table_properties 查看与改写; 跨格式迁移则可经 iceberg_to_ducklake(...) 把 Iceberg catalog 复制进 DuckLake。

在允许的约束内, 扩展支持较完整的 DML: 建 schema/表、INSERT、UPDATE、DELETE、MERGE INTO、以及 ALTER TABLE (加列、删列、改名、改类型) 均可执行; COPY FROM DATABASE duckdb_db TO my_catalog 与反向复制可做整库深拷贝。需要认清的是, 这些能力都建立在前面列出的 positional delete + merge-on-read + 非排序表这三项限制之上, 超出范围的写入会直接报错而非静默降级。

14.4 其他常用扩展

sqlite.把 SQLite 数据库文件作为外部 catalog 附加后直接读写, autoloader 为是, 别名 `sqlite_scanner / sqlite3`。

```
INSTALL sqlite;  
LOAD sqlite;  
ATTACH 'file.db' AS my_sqlite (TYPE sqlite);  
SELECT * FROM my_sqlite.main.table1;
```

类型亲和 (type affinity) 规则将 SQLite 的存储类 INTEGER/REAL/TEXT/BLOB/NULL 映射到 DuckDB 类型; 选项 `sqlite_all_varchar` 可把所有列读成 VARCHAR 以对抗脏数据。该扩展与 Turso (libSQL) 文件兼容。附加后可像普通 catalog 一样读写, 改动回写到原文件; 也可只读挂载以作探查, 使其成为 DuckDB 与既有 SQLite 应用之间的双向桥接。

excel.读写 Excel .xlsx 文件, autoloader 为是。read_xlsx 负责读取, COPY ... TO 'out.xlsx' (FORMAT xlsx) 负责写出。

```
INSTALL excel;  
LOAD excel;  
SELECT * FROM read_xlsx('report.xlsx');  
COPY (SELECT * FROM t) TO 'out.xlsx' (FORMAT xlsx);
```

限制: 仅支持 .xlsx (Office Open XML), 旧的二进制 .xls 格式不被支持。常见用途是把业务导出的报表直接入库, 或把查询结果回写供非技术同事用 Excel 打开; 多工作表场景下需逐表指定后再合并。

aws. AWS 凭据与服务的 Secrets Manager 集成 (覆盖 S3、Glue、S3 Tables、RDS 等), autoloader 为是。credential_chain 支持 config、env、sts、sso、instance、process、web_identity 多种 provider; 新增的 rds_secret_type 支持 AWS RDS IAM 认证, 临时凭据还能在过期前自动刷新 (auto-refresh)。对临时凭据而言, 自动刷新意味着长任务无需手动续期 IAM 令牌; 配合 credential_chain 的 sts / assume_role 还能实现跨账号访问, 是生产环境访问 S3 数据湖的常见路径。

```
INSTALL aws;  
LOAD aws;  
CREATE SECRET (TYPE s3, PROVIDER credential_chain, CHAIN config,sts, REGION 'us-east-1');
```

azure. 访问 Azure Blob Storage 与 Data Lake Storage (ADLS), autoloader 为是; 协议 scheme 含 az / azure / abfss。凭据 provider 有 CONFIG、credential_chain、managed_identity、service_principal。当数据落在 ADLS Gen2 时, 递归通配 (/**) 能一次扫描整个目录树; 而 Blob 存储的 glob 更偏向单层级匹配, 混用时需留意语义差异, 避免漏读或多读。

```
INSTALL azure;  
LOAD azure;  
SELECT * FROM 'az://container/path/file.parquet';
```

delta. 读取 Delta Lake 表 (基于 delta-kernel-rs), autoloader 为是。可通过 delta_scan('path') 或 ATTACH ... (TYPE delta) 接入, 并支持 time travel: SELECT * FROM tbl AT (VERSION => n) 或按时间戳回溯。

```
INSTALL delta;  
LOAD delta;  
SELECT * FROM delta_scan('s3://bucket/delta-table') AT (VERSION => 3);
```

限制: 平台仅含 Linux / macOS / Windows 的 AMD64 (linux_amd64 与 linux_arm64 也支持), 无 windows_arm64。time travel 让数据回滚与审计变得直接; 由于 Unity Catalog 扩展构建于 delta 之上, 二者共享底层读取能力, 理解 delta 即理解了 Unity Catalog 的读取底座。

ducklake. DuckLake 是 DuckDB 原生的湖仓 (lakehouse) 格式, 元数据与数据分离存储, autoloader 为是。

```
INSTALL ducklake;
LOAD ducklake;
ATTACH 'ducklake:metadata.ducklake' AS my_ducklake (DATA_PATH 's3://bucket/data');
```

版本事实：DuckLake 1.0 于 2026 年 4 月发布。其与 Iceberg 互通，可经 iceberg_to_ducklake(...) 把 Iceberg catalog 迁移或复制到 DuckLake。元数据与数据分离意味着 catalog 信息（schema、表、快照）与真实数据文件可置于不同存储后端（例如元数据留在本地、数据落在 S3），便于权限与生命周期分别治理。

vss (向量相似度搜索)。vss 基于 HNSW 索引提供向量相似度搜索，目前为实验性（experimental）。

```
INSTALL vss;
LOAD vss;
CREATE INDEX idx_vec ON vec_tbl USING HNSW (vec);
SELECT * FROM vss_match('vec_tbl', 'vec', [0.1, 0.2, 0.3]);
```

距离度量（metrics）有 l2sq（欧氏平方）、cosine（余弦）、ip（内积）；配套函数 vss_join、vss_match。限制：仅支持 FLOAT 向量，不支持其他数值类型；索引持久化仍属实验特性（开关 hnsw_enable_experimental_persistence）。HNSW（Hierarchical Navigable Small World）是一种近似最近邻（ANN）索引，查询时不保证全局最优但速度远快于暴力扫描；开启持久化后索引可落盘复用，但该能力目前仍标记为实验。

14.5 本章小结

- JSON 与 parquet 为内置扩展，随二进制发布、开箱即用；JSON 类型承载不规则数据，read_json 负责读入与 schema 自动推断，构造函数则把 DuckDB 结构回写为 JSON。
- spatial 因依赖 GEOS / PROJ / GDAL 外部系统库而必须显式 LOAD；GEOMETRY 类型自 v1.5 内建、运算由扩展提供，R-Tree 索引与 GDAL 集成进一步扩展了空间能力。
- iceberg 支持路径与目录两种模式，v1.5.5 兼容 V3 spec（VARIANT、TIMESTAMP_NS、默认值、binary deletion vectors），但写入仅限 positional delete + merge-on-read 且作用于非排序表，GEOGRAPHY/UNKNOWN 待 v2.0.0。
- sqlite、excel、aws、azure、delta、ducklake、vss 等扩展覆盖了从本地文件到云存储、从关系数据库到向量检索的常见场景，各自有 autoload 行为与平台/格式限制；使用前应核对扩展与 DuckDB 版本的强绑定关系，跨版本升级时以 FORCE INSTALL 重新拉取匹配构建。

第五部分 性能与运维

第 15 章 配置与性能调优

DuckDB 在大多数场景下依靠精心选择的默认配置即可达到良好性能，但当数据规模、并发环境或硬件条件偏离典型负载时，理解并主动调整配置仍是工程落地的必要能力。本章系统梳理配置机制、内存与线程资源、性能最佳实践、执行计划分析方法，以及常见性能陷阱与排查手段，目标是在不臆造命令与参数的前提下，提供可直接套用的调优指引。

15.1 配置机制

DuckDB 的行为由一组配置项（configuration options）控制，共有四种设置途径。

方式	语法 / 说明	备注
SET 语句	SET memory_limit = '10GB'; SET threads TO 1;	最常用；赋值类 PRAGMA 等价于 SET
PRAGMA 语句	PRAGMA enable_progress_bar;	源自 SQLite 扩展；赋值类 PRAGMA 等价于 SET

方式	语法 / 说明	备注
连接参数 (connection parameters)	打开数据库时以参数传入 (如 Python <code>duckdb.connect(configuration={'memory_limit': '10GB'})</code>)	进程启动前生效; 安全项 <code>low_unsigned_extensions</code> 仅能在启动时开启
环境变量	文档未列出具体的名称	主要通过前三种方式生效

SET 与 PRAGMA 是最常用的交互方式, 二者在赋值时等价。连接参数用于进程启动前就必须确定的项 (例如 `allow_unsigned_extensions` 只能在启动时开启); 环境变量在当前文档体系中并未列出具体的名称, 配置主要通过前三种方式落地。

```
SET memory_limit = '10GB';
SET threads TO 4;
PRAGMA enable_progress_bar;
```

任意一项配置若需还原为默认值, 使用 RESET 语句:

```
RESET memory_limit;
```

查询当前生效值有两种手段。其一是 `current_setting()` 函数, 按名称取单项:

```
SELECT current_setting('memory_limit') AS mem_limit;
```

其二是 `duckdb_settings()` 表函数, 可列出全部配置项及其默认值、取值范围与当前值:

```
SELECT value FROM duckdb_settings() WHERE name = 'memory_limit';
SELECT * FROM duckdb_settings();
```

配置项按作用域分为 GLOBAL 与 LOCAL 两类。GLOBAL 配置对整个数据库实例生效, 例如 `threads`、`memory_limit`、`checkpoint_threshold` 等; LOCAL 配置仅对当前会话或连接生效, 例如 `schema`、`search_path`、`enable_profiling`、`enable_progress_bar`、`streaming_buffer_size` 等。在同一进程内服务多个连接时, LOCAL 项可为不同连接定制行为而不影响全局。

15.2 内存与线程配置

内存上限由 `memory_limit` (别名 `max_memory`) 控制, 默认值为物理内存的 80%, 以字符串形式指定, 例如 `'1GB'`、`'10GB'`。一个容易被忽视的事实是, 该限制只约束 `buffer manager` (缓冲管理器) ——多数查询的中间数据由它托管, 但向量 (vectors)、查询结果以及状态复杂的部分聚合函数 (如 `list`、`mode`、`quantile`、`string_agg` 等) 所使用的内存存在 `buffer manager` 之外分配。因此实际占用可能高于设定值, 不能把 `memory_limit` 当作硬性内存上限。在内存受限或容器环境中, 可将该值降至 50%~60% 以预留缓冲。

```
SET memory_limit = '4GB';
```

并行度由 `threads` (别名 `worker_threads`) 控制, 默认等于 CPU 核数。在容器或多租户环境中, 默认占满全部核会挤占其他负载, 应显式限制; 若需要确定性结果 (并行浮点聚合可能引入约 $1e-16$ 量级的微小差异, 且集合语义不保证行序), 则设为 1。

```
SET threads = 4;
```

当内存不足以容纳中间数据时, DuckDB 将数据 `spill to disk` (溢写到磁盘), 溢出目录由 `temp_directory` 指定, 默认位于库文件同名的 `.tmp` 目录 (内存模式下为 `.tmp`), 可用 `max_temp_directory_size` 限制其体积, 默认值为可用磁盘的 90%。将溢出目录指向高速或大容量磁盘有助于提升大查询的稳定性。

```
SET temp_directory = '/path/to/temp_dir.tmp/';
SET max_temp_directory_size = '4GB';
```

持久化相关的自动检查点由 `checkpoint_threshold` (别名 `wal_autocheckpoint`) 控制, 默认值为 16 MiB——当预写日志 (WAL) 达到该体积时触发自动 checkpoint。写密集场景下可调大以减低 checkpoint 频率, 例如设为 '1GB'。

```
SET checkpoint_threshold = '1GB';
```

存储层方面, `default_block_size` 默认 262144 (256 kB), 取值范围须为 2 的幂且介于 16384~262144 之间, 仅在新建库文件时生效。`preserve_insertion_order` 默认开启, 关闭后允许系统对无 ORDER BY 的结果重排, 从而降低内存占用 (详见 15.5)。

15.3 性能最佳实践

类型选择对性能影响显著。用 VARCHAR 编码本应以数值或时间表示的列, 既浪费空间又拖慢计算。在 LDBC 社交网络数据集的 5.54 亿行 `creationDate` 列上, DATETIME 占用 3.3 GB、查询耗时 0.9 s, 而 VARCHAR 占用 5.2 GB、耗时 3.9 s。自连接测试中, 用 BIGINT 编码值的耗时为 1.2 s, 用 VARCHAR 则为 2.1 s, 差距约 1.8×; 聚合场景下 VARCHAR 的劣势往往放大到数倍。原则是优先使用 TIMESTAMP、BIGINT、DATE 等恰当类型, 内存受限时选用最受限的小类型 (如 TINYINT), 因为即便 bitpacking 压缩使磁盘占用不变, 处理时仍需按完整类型占用内存。

```
-- 推荐: 以 TIMESTAMP 存储时间, 以 BIGINT 存储整数编码, 避免 VARCHAR 编码数值
CREATE TABLE events (ts TIMESTAMP, user_id BIGINT, value DOUBLE);
```

主键、唯一、外键等约束 (constraints) 与索引 (indexes) 仅服务于数据完整性, 几乎不提升查询性能。DuckDB 的 join 与聚合并不依赖它们来加速。约束检查与索引维护反而会拖慢写入: 同一份 LDBC 5.54 亿行数据, 带主键加载耗时 461.6 s, 无主键加载仅需 121.0 s; 若先无主键加载再补加主键, 耗时为 242.0 s。因此批量加载应避免主键, 确有完整性需要时应在加载完成后追加。ART 索引 (Adaptive Radix Tree) 隐式建于约束之上或显式 CREATE INDEX 创建, 仅当单列、无表达式且查询高度选择性时才可能走 index scan (索引扫描), 且索引缓冲尚未被 buffer manager 回收, 可能占用可观内存。

压缩是另一个反直觉却关键的要点。持久化数据库默认启用压缩, 而内存表默认不压缩。TPC-H Q1 (SF30) 实测显示: 未压缩内存库耗时 4.22 s, 压缩内存库 0.55 s, 压缩持久化库 0.56 s——压缩使查询提速约 8×。因此在内存充足且追求性能时, 使用压缩的持久化库往往比未压缩的内存表更优。

并行以 row group (行组) 为单位, DuckDB 原生格式的默认 row group 大小为 122,880 行。一条查询要利用 k 个线程, 至少需要扫描 $k \times 122,880$ 行。可在 ATTACH 时调整:

```
ATTACH '/tmp/somefile.db' AS db (ROW_GROUP_SIZE 16384);
```

内存规划上, 理想状况是每线程分配 1~4 GB。聚合密集负载每线程 1~2 GB, join 密集则建议 3~4 GB。当 HyperThreading 等因素导致线程过多反而变慢时, 应手动下调 threads。远程小文件请求密集时, 由于同步 IO, 可将 threads 设为 CPU 核数的 2~5 倍以提升并发。

文件格式方面, Parquet 普遍优于直接查询 CSV (TPC-H 在 Parquet 上比原生库慢 1.1~5.0×, 但仍是首选外部格式)。理想的 Parquet row group 规模为 100K~1M 行, 单文件 100 MB~10 GB; group 小于 5000 行会使运行时间劣化 5~10×。Hive 分区 (Hive partitioning) 能使 filter pushdown (又称 predicate pushdown, 谓词下推/过滤器下推) 跨越分区边界生效, 在远程文件场景下尤为显著。

```
-- Hive 分区读取: 分区列作为过滤条件下推, 减少远程传输与扫描量
SELECT * FROM read_parquet('s3://bucket/data/dt=*/part-*.parquet');
```

15.4 执行计划分析

理解查询为何慢, 离不开对执行计划的审视。EXPLAIN 仅打印物理计划 (physical plan) 而不实际执行, 输出中给出估计基数 (EC, estimated cardinality), 用于判断优化器对数据规模的估计是否合理。

```
EXPLAIN
SELECT u.user_id, count(*)
FROM users u
JOIN events e ON u.user_id = e.user_id
```

```
GROUP BY u.user_id;
```

EXPLAIN ANALYZE 则会真正执行查询并逐算子统计累计耗时、估计基数与真实基数。在多线程下各算子时间之和会大于总时长；远程读取场景还会报告请求数与传输量，是定位网络瓶颈的直接依据。

```
EXPLAIN ANALYZE
SELECT u.user_id, count(*)
FROM users u
JOIN events e ON u.user_id = e.user_id
GROUP BY u.user_id;
```

分析时重点关注两类问题。其一是连接策略：应争取 hash join，避免 nested loop join（嵌套循环连接）。当统计信息缺失或估计严重偏差时优化器可能误选嵌套循环，此时可临时关闭相关优化器以强制连接顺序（仅调试用，生产应恢复）：

```
SET disabled_optimizers = 'join_order,build_side_probe_side';
-- 按 JOIN 书写顺序构建 left-deep join tree
SET disabled_optimizers = '';
```

其二是 cardinality explosion（基数爆炸）：多表连接或不当的笛卡尔积会使中间结果行数指数膨胀，即便单个算子看似高效，整体也会因数据量失控而极慢。通过对比 EXPLAIN ANALYZE 中的估计基数与真实基数，可快速识别估计失准的环节，辅以更精准的统计信息或重写查询来纠正。

15.5 常见性能陷阱与排查

超内存（larger-than-memory）负载依赖 spill to disk 支持，GROUP BY、JOIN、ORDER BY 以及带 OVER（PARTITION BY ... ORDER BY ...）的窗口函数均可在内存不足时落盘。但仍有若干情形会突破这一机制而引发 OOM（内存溢出）：

- 同一查询中多个 blocking operator（阻塞算子）叠加，各自需要缓冲全部输入，内存峰值叠加；
- 部分聚合函数不支持落盘，典型如 list() 与 string_agg()；
- 排序聚合（在 ORDER BY 子句中做聚合）属于 holistic 算子，需容纳全部输入；
- PIVOT 内部使用 list()，受同样的限制。

缓解 OOM 的常用手段是关闭 preserve_insertion_order 以降低内存占用，并适度下调 memory_limit（因为部分操作绕过 buffer manager，留出余量反而更稳）：

```
SET preserve_insertion_order = false;
SET memory_limit = '50%';
SET threads = 4;
```

除性能陷阱外，命令行工具存在两个版本相关的已知坑。v1.5.0 在 Linux/macOS 下以管道方式执行的脚本（如 duckdb < test.sql）不会运行，绕过方法是改用 -f test.sql 或将输出管道到 cat：

```
duckdb -f test.sql
duckdb < test.sql | cat
```

v1.5.2 中直接以命令行附加 HTTPS 数据库会失败（https 自动加载时报告 Schema with name main does not exist!），解决方式是在 URL 前加 duckdb：前缀：

```
duckdb duckdb:https://blobs.duckdb.org/databases/stations.duckdb
```

排查 OOM 时还需区分两类错误：DuckDB 抛出的 OutOfMemoryException，以及 Linux OOM reaper 直接杀死进程（可用 dmesg 查 Killed）。崩溃前若存在 WAL 文件，新会话打开即自动重放并 checkpoint 恢复，无需人工干预。

15.6 本章小结

本章梳理了 DuckDB 的配置与性能调优要点。配置上，SET/PRAGMA、连接参数与环境变量四途并用，RESET 还原默认值，current_setting() 与 duckdb_settings() 用于查询，GLOBAL 与 LOCAL 区分

实例级与会话级作用域。内存与线程方面，memory_limit 仅约束 buffer manager、实际可超，threads 默认占满核数需在容器与多租户场景中限制，temp_directory 与 max_temp_directory_size 管控 spill to disk 落盘，checkpoint_threshold 默认 16 MiB。性能实践上，恰当类型（如 TIMESTAMP、BIGINT）远优于 VARCHAR，约束与索引只保完整性、应在批量加载后追加，压缩持久化库常比未压缩内存表快约 8×，并行以 row group（122,880 行）为单位、每线程规划 1–4 GB，Parquet 宜采用 100K~1M 的 row group 与 100 MB~10 GB 单文件并借 Hive 分区助 filter pushdown。分析执行计划应善用 EXPLAIN 与 EXPLAIN ANALYZE，规避 nested loop join 与 cardinality explosion。面对 OOM 风险，须识别多 blocking operator 叠加、list()/string_agg()、排序聚合与 PIVOT 等不受落盘保护的场景，并以 preserve_insertion_order=false 与下调 memory_limit 缓解；v1.5.0 与 v1.5.2 的命令行已知坑亦需掌握对应绕过方式。

第 16 章 运维手册

DuckDB 以进程内（in-process）嵌入式数据库的形态出现，没有常驻服务进程，也不需要 DBA 团队去维护一套独立的数据库服务器。这种轻量特性让很多人误以为它“无需运维”。事实上，当 DuckDB 进入生产环境——无论是作为本地分析工具、数据管道的一环，还是被嵌入到应用程序内部——安装渠道的选择、版本演进的约束、磁盘与内存足迹（footprint）、日志记录以及安全边界，都会直接影响系统的可靠性与可重复性。本章从工程实践的角度，系统梳理把 DuckDB 跑稳、跑安全所必须掌握的四件事：版本策略、足迹管理、日志体系与安全加固。

16.1 安装与版本策略

安装 DuckDB 的渠道随使用场景而异。交互式 CLI 与嵌入到应用程序中所采用的路径并不相同，语言客户端也各有各的包管理器。

安装脚本（install.duckdb.org）

Linux 与 macOS 上，官方提供一键安装脚本，把最新稳定版 CLI 安装到 ~/.duckdb/cli/latest/duckdb：

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

安装完成后，需把 export PATH=~/.duckdb/cli/latest":\$PATH 写入 shell 配置文件，使 duckdb 命令在任意目录下可用。

版本化（versioning）是生产部署的核心诉求。安装脚本通过 DUCKDB_VERSION 环境变量支持精确锁定某一发布版本——可回溯到 v1.0.0 以来的任意历史发布版（past releases）。例如锁定 1.2.2：

```
curl https://install.duckdb.org | DUCKDB_VERSION=1.2.2 sh
```

这一机制的价值在于可重复性：在 CI、容器镜像或交付物中固定 DUCKDB_VERSION，能确保每次构建都拉取行为一致的二进制，避免因“最新版”漂移引入不预期的变更。从发布治理的角度看，把版本号显式写死，相当于为生产环境锁定一条长期稳定、可审计的基线，其作用类似于其他生态中的 LTS（Long-Term Support，长期支持）策略——差别只是 DuckDB 并不单独划定一条带名称的 LTS 通道，而是把“选哪个版本”的决定权完全交给使用者。

Windows 用户可借助 beta 阶段的 PowerShell 脚本：

```
powershell -NoExit iex (iwr "https://install.duckdb.org/install.ps1").Content
```

官方 Docker 镜像

容器化场景下，官方镜像 duckdb/duckdb 同时支持 ARM64 与 x86_64 架构：

```
docker run --rm -it -v "$(pwd):/workspace" -w /workspace duckdb/duckdb
```

该镜像把工作目录挂载为 /workspace，便于在容器内读写宿主文件。若需使用 DuckDB UI，则要附加 --net host；但这会把容器的全部端口转发到宿主网络，存在暴露风险，仅在可信环境中采用。

各语言客户端（Python、R、Java、Node 等）通过各自的包管理器安装，CLI 则专用于交互，官方不推荐将其作为嵌入载体。

升级与扩展版本匹配

扩展二进制 (extension binaries) 与具体的 DuckDB 版本、平台严格绑定。一次 DuckDB 主版本或次版本升级, 往往意味着既有扩展需要重新安装与之匹配的版本; 直接沿用旧版扩展二进制会导致加载失败或行为异常。因此, 升级 DuckDB 时必须同步重装匹配版本的扩展——把扩展版本纳入与 DUCKDB_VERSION 相同的锁定与回归测试流程, 是避免“升级即故障”的关键。

16.2 DuckDB 的足迹

DuckDB 的足迹 (footprint) 指的是它在磁盘与内存上留下的痕迹, 以及运行前需要为它预留多少资源。理解这一点, 才能在版本控制、备份与容量规划上做出正确决策。

嵌入式与单文件存储

DuckDB 是嵌入式 (embedded) 分析型数据库, 以 in-process 方式运行在调用它的进程内部, 没有独立服务进程, 也不监听端口。它的存储格式是单文件 (single-file) 的: 库文件可以使用任意扩展名, 常见的是 .duckdb、.db、.ddb。这一特性让数据库的“部署”退化为“拷贝一个文件”, 也是它适合随应用分发的根本原因。

运行时创建的辅助文件

除了库文件本身, DuckDB 在运行时会创建若干辅助文件, 分全局与工作目录两类。

全局文件位于用户 home 目录 (~):

文件 / 目录	说明
~/.duckdbrc	CLI 启动时自动执行的命令, 类似 .bashrc
~/.duckdb_history	历史命令记录, 类似 .bash_history
~/.duckdb/extensions	已安装扩展的二进制 (跨版本不共享)
~/.duckdb/stored_secrets	持久化 secret

工作目录或库文件相对路径下的本地文件:

名称	说明
<database_filename>	数据库文件 (仅磁盘模式)
.tmp/	临时目录 (仅内存模式)
<database_filename>.tmp/	临时目录 (仅磁盘模式)
<database_filename>.wal	预写日志 (WAL), 正常退出时删除, 崩溃时用于恢复

预写日志 .wal 是崩溃恢复的关键: 正常关闭数据库后它会被清理; 若进程异常终止, 下次打开时 DuckDB 会依据 .wal 回放未落盘的写入。临时目录 .tmp/ 则用于查询溢出 (spill) 与中间结果, 其大小受 max_temp_directory_size 约束 (默认可用磁盘的 90%)。

二进制体积与内存占用

DuckDB 以极小的嵌入式库著称: CLI 是单个可执行文件, 客户端库也保持轻量。内存方面, 默认上限为物理内存的 80% (memory_limit), 由 threads (默认等于 CPU 核数) 控制并行度。需要提醒的是, memory_limit 仅约束 buffer manager, 向量、查询结果以及复杂状态聚合函数 (如 list、mode、quantile) 所用的内存在其之外, 实际占用可能高于设定值, 不能把它当作硬性内存天花板。

用.gitignore 管理版本控制

把 DuckDB 库纳入 Git 仓库时, 应按“保留库文件、忽略临时产物”或“连库文件一并忽略”两种策略配置。仅忽略临时文件、保留库文件:

```
*.wal  
*.tmp/
```

若连库文件本身也要排除（以 .duckdb 为例）：

```
*.duckdb*  
*.wal  
*.tmp/
```

删除数据行后，DuckDB 不会通过 VACUUM 回收空间——这一点与 PostgreSQL 等行为不同。VACUUM 在 DuckDB 中不发生收缩；要回收删除产生的空闲空间，应执行 CHECKPOINT；。若要彻底压缩库文件、得到一个紧凑副本，可用 COPY FROM DATABASE 生成全新数据库：

```
ATTACH 'db1.db' AS db1;  
ATTACH 'db2.db' AS db2;  
COPY FROM DATABASE db1 TO db2;
```

单库文件本身没有实际大小上限（已有 15 TB 以上的库文件正常运行），但连接超大库可能耗费数秒，checkpoint 也更慢，容量规划时应对此有预期。

16.3 日志

DuckDB 内置一套可查询的日志体系，由 enable_logging（默认开启，值为 1）控制。所有日志进入名为 duckdb_logs 的特殊视图，可像普通表一样 SELECT：

```
CALL enable_logging();  
SELECT * FROM duckdb_logs;  
CALL truncate_duckdb_logs(); -- 清空当前日志  
CALL disable_logging();
```

日志级别

日志级别按严重程度由低到高排列为 ERROR < WARN < INFO < DEBUG < TRACE。配置参考中 logging_level 的默认值为 WARNING，即默认只记录警告及以上。需要排查问题时，可临时调高：

```
CALL enable_logging(level = 'debug');
```

日志存储位置

日志可写入三种存储（storage）：memory（默认）、stdout（以 CSV 形式输出到标准输出）、file（CSV 文件）。切换到标准输出：

```
CALL enable_logging(storage = 'stdout');
```

写入文件时指定路径，简写形式为 storage_path：

```
CALL enable_logging(storage = 'file', storage_config = {'path': 'path/to/logs'});  
CALL enable_logging(storage_path = 'path/to/logs');
```

duckdb_logs 视图会自动指向当前活动存储。需要注意日志有缓冲，stdout 默认禁用缓冲、而 memory/file 默认缓冲 2048 条；调试崩溃场景可设 buffer_size=0 以尽量不丢日志：

```
CALL enable_logging(storage_config = {'buffer_size': 0});
```

结构化日志类型

DuckDB 对特定子系统提供结构化（structured）日志，便于机器解析与可观测性接入。任务关键的类型包括 HTTP、FileSystem 与 Metrics：

- HTTP：记录每次 HTTP 请求的 request 与 response 结构，含类型、URL、起止时间、耗时与头信息；
- FileSystem：记录文件系统层的操作；
- Metrics：暴露内部度量指标。

此外还有 QueryLog、PhysicalOperator 等类型。按类型启用日志：

```
CALL enable_logging('HTTP');  
CALL enable_logging(['HTTP', 'FileSystem']);
```

结构化日志可用 duckdb_logs_parsed('HTTP') 宏解析，并用 DESCRIBE FROM duckdb_logs_parsed('HTTP') 查看其 schema，从而直接落地到监控系统或数据管道。

16.4 安全加固

DuckDB 的执行模型决定了它必须与 bash、Python 一样被看待：它用运行它的那个用户的权限执行 SQL。把 SQL 当作代码，而不是数据——不要在未经沙箱化的情况下执行来源不可信的 SQL。本章列出的配置项属于纵深防御（defense-in-depth），它们能收窄攻击面，但无法替代操作系统或容器级别的隔离。

最小权限的文件访问

enable_external_access 默认 true，允许加载扩展、COPY、读取 CSV/Parquet 等外部状态访问。不可信环境下应先将其关闭，再以白名单精确放行必要的目录或文件——即便外部访问被整体禁止，白名单项仍可使用：

```
SET allowed_directories = ['/tmp'];  
SET enable_external_access = false;
```

也可通过 allowed_paths 以单文件粒度放行，或用 SET disabled_filesystems='LocalFileSystem'；彻底禁掉本地文件系统。

扩展签名与供应链风险

扩展与 DuckDB 进程同权，每一次加载都做二进制签名校验，分为 core key 签名、community key 签名与无签名三类。安全级别由配置项决定：默认允许 core 与 community 扩展；设 SET allow_community_extensions = false 则只允许 core 扩展（运行不可信 SQL 的 Web 服务建议如此）。allow_community_extensions 具备自我锁定特性——关闭后运行期内无法再改回 true，报 Cannot upgrade ... while database is running。

allow_unsigned_extensions 默认 false，且只能在启动时开启，属于明确的提权风险，仅扩展开发时才考虑。社区扩展仓库本质类似 pip/npm，是“远程代码执行设计”，务必只从可信源安装。

与之相关的还有自动加载/安装开关 autoload_known_extensions 与 autoinstall_known_extensions，二者默认值均为 true：查询依赖某扩展时，DuckDB 会自动联网安装并加载。为抵御供应链（supply-chain）风险——避免不可信环境下静默拉取并执行远程代码——应显式关闭：

```
SET autoload_known_extensions = false;  
SET autoinstall_known_extensions = false;
```

锁定配置防篡改

自定义配置完成之后，可用 lock_configuration 把实例配置整体锁定，此后任何配置项都不可再改；allowed_configs 则指定锁定时仍可调的例外项：

```
SET allowed_configs = ['memory_limit', 'threads'];  
SET lock_configuration = true;
```

这与关闭自动加载/安装组合使用，能固化一个受控、可重复的安全基线。

CLI 安全模式

交互式 CLI 提供 Safe Mode（安全模式），通过启动参数 -safe 或点命令 .safe_mode 进入，禁止访问数据库文件以外的外部文件，等于把读写的边界限制在库自身：

```
duckdb -safe mydata.duckdb
```

预编译语句与权限

防止 SQL 注入的正确做法是使用预编译语句 (prepared statements)：你控制查询结构，只接收不可信数据作为值参数，从而杜绝拼接字符串带来的注入。

```
# 危险：字符串拼接
duckdb.execute("... WHERE x = " + str(42))
# 安全：预编译，值以参数传入
duckdb.execute("... WHERE x = ?", [42])
```

若允许用户提交整条 SQL，等价于允许执行任意代码，必须配合沙箱处理。

权限方面，应避免以 root (sudo) 运行 DuckDB。嵌入场景中，CLI 客户端面向交互，带有 .sh 等可直接执行 shell 命令的特性 (仅 CLI 具备，其他客户端无)；嵌入应用时应当调用客户端库 (Python/R/Java/Node 等)，而不要通过 shell 去调用 CLI，以免引入命令执行面。

16.5 本章小结

把 DuckDB 投入生产，运维重心从“养服务器”转变为“管版本、控足迹、开日志、守边界”。本章的四条主线可归纳如下：

- 安装与版本：通过 DUCKDB_VERSION 锁定发布版本，保证构建可重复；官方 Docker 镜像覆盖主流架构；升级须同步重装与版本、平台严格绑定的扩展。
- 足迹管理：单文件库无服务进程，但会产生 .wal 与 .tmp/ 等辅助文件；用 .gitignore 区分库文件与临时产物，删除空间回收靠 CHECKPOINT 而非 VACUUM。
- 日志体系：logging_level 默认 WARNING，日志可存 memory/stdout/file，HTTP/FileSystem/Metrics 等结构化类型便于接入可观测性。
- 安全加固：把 SQL 当作代码，enable_external_access 配白名单做最小权限，关闭自动加载/安装以抵御供应链风险，lock_configuration 固化基线，CLI 用 -safe 进入 Safe Mode，嵌入走客户端库而不调用 CLI。

这些配置并非银弹，它们与操作系统沙箱、容器隔离共同构成纵深防御。在不可信输入面前，限制文件访问、禁用 root、使用预编译语句，才是最贴近本质的防线。

第六部分 原理剖析

第 17 章 DuckDB 内部原理

DuckDB 之所以能在进程内 (in-process) 环境中提供接近专门分析引擎的查询性能，根源不在某一处魔法，而是一组相互咬合的设计选择：列式存储 (columnar storage)、向量化执行 (vectorized execution)、磁盘上的轻量压缩 (lightweight compression) 以及基于 MVCC 的并发模型。前三者决定了单条查询如何被高效执行，最后一项决定了多个查询与写入如何互不干扰地共存。本章沿“一条 SQL 如何被执行”这条主线，逐层拆解 DuckDB 的内部机制：从词法解析到物理计划，从 DataChunk 在算子间的流动到 Row Group 在磁盘上的压缩布局，再到优化器的改写规则与事务的乐观并发。读完本章，读者应能把书中学过的 SQL 语句与引擎内部的实际行为对应起来。

17.1 查询处理流水线

SQL 字符串从输入到产出结果，在 DuckDB 内部经历一条分层、单向的流水线。这条流水线把“文本”逐级翻译成“可在内存中批量执行的物理算子”，每一层只关心自己职责范围内的转换，且下游阶段依赖上游阶段已经完成的解析与绑定。理解这条流水线，是理解后续向量化执行、优化器与存储所有机制的前提。

17.1.1 Parser：只做词法拆解，不感知语义

Parser 的职责是把原始 SQL 字符串切成若干 token，并组装成四种中间结构：SQLStatement、QueryNode、TableRef 与 ParsedExpression。其中 SQLStatement 表示一条完整语句——一条 SQL 字符

串可以拆成多条语句；QueryNode 表示 SELECT 或集合操作（UNION / INTERSECT / DIFFERENCE）；TableRef 表示一切表来源，无论是基表、join、表生成函数还是子查询；ParsedExpression 表示表达式。

关键在于，Parser 阶段完全不感知 catalog，也不解析列类型。即使被查询的表在数据库中根本不存在，Parser 也不会报错——它只在语法层面完成 token 化。除显式 CAST 外，ParsedExpression 在此时没有类型信息，真正的类型要到 Binder 阶段才被解析。这种“无语义”的设计让 Parser 足够轻量，也为后续阶段保留了全部判断空间。

17.1.2 Binder: 借助 catalog 完成语义绑定

Binder 把 Parser 产出的节点转换为“绑定（bound）”等价物，对应关系为：SQLStatement $\rightarrow$ BoundStatement、QueryNode $\rightarrow$ BoundQueryNode、TableRef $\rightarrow$ BoundTableRef、ParsedExpression $\rightarrow$ Expression。Binding 过程通过 catalog 解析表与列、解析每个表达式的类型，并抽取聚合函数与窗口函数。只有经过 Binder，一个表达式才真正拥有了类型，一个表引用才真正指向 catalog 中存在的对象。

17.1.3 Logical Planner 与 Optimizer

Logical Planner 由 bound statement 构造 LogicalOperator 节点树，形成逻辑查询树——这一步描述“要算什么”，还不涉及“怎么算”。Optimizer 随后对逻辑查询树运行一系列改写优化（详见 17.4），目标是产出语义等价但执行代价更低的计划。

17.1.4 Column Binding Resolver: 从“列名”到“下标”

逻辑层的表达式引用的是“某表某列”，形式为 BoundColumnRefExpression。Column Binding Resolver 把这种语义化引用改写为执行引擎中的 BoundReferenceExpression，即直接引用某个 DataChunk 内具体下标的列。这一步抹去了所有表/列的符号信息，让下游算子只需按位置取数，是向量化执行得以高效运转的关键衔接。

17.1.5 Physical Plan Generator 与 Execution

Physical Plan Generator 把逻辑算子树转换为 PhysicalOperator 树，描述“具体如何执行”。Execution 阶段运行这些物理算子，产出结果。物理算子的执行采用向量化方式，DataChunk 在算子之间流动——这正是 17.2 的主题。

整条流水线用文字描述如下：

```
SQL 字符串  $\rightarrow$  Parser(token)  $\rightarrow$  Binder(bound)
   $\rightarrow$  Logical Planner(LogicalOperator 树)  $\rightarrow$  Optimizer(优化后的逻辑树)
   $\rightarrow$  Column Binding Resolver  $\rightarrow$  Physical Plan Generator(PhysicalOperator 树)
   $\rightarrow$  向量化执行(DataChunk 在算子间流动)  $\rightarrow$  结果
```

借助 EXPLAIN 可以直接观察优化后的物理计划，验证上述阶段的最终产物：

```
EXPLAIN SELECT l_orderkey, sum(l_quantity)
FROM lineitem
WHERE l_shipdate >= DATE '1995-01-01'
GROUP BY l_orderkey
ORDER BY l_orderkey;
```

EXPLAIN 的输出自上而下展示物理算子（如 SEQ_SCAN、HASH_GROUP_BY、ORDER_BY），其结构正对应 Physical Plan Generator 产出的 PhysicalOperator 树。

17.2 向量化执行引擎

17.2.1 Volcano 模型的问题

传统数据库普遍采用 Volcano (又称 Iterator) 执行模型。该模型中每个算子实现 `next()` 方法, 每次调用返回一行 (tuple-at-a-time) 数据。算子之间形成嵌套调用, 一次查询要在整棵算子树上来回穿梭、逐行推进。这种逐行模型带来三个性能短板: 频繁的虚函数调用 (virtual function call) 产生可观的解释开销; 分支走向难以预测, 破坏 CPU 流水线; 单次只搬运一行, CPU 缓存利用率低, 难以发挥 SIMD 指令的批处理能力。

17.2.2 DuckDB 的推送式向量化模型

DuckDB 采用基于推送 (push-based) 的向量化执行模型。算子的批处理单位不是单行, 而是固定大小的向量 (Vector)。整批数据在算子树间流动, DataChunk 被“推送”穿过各层算子, 而非由上层逐个拉取。所有算子都针对一个固定的向量大小优化, 这个大小由常量 `STANDARD_VECTOR_SIZE` 决定, 默认 2048 行 (元组)。

向量化模型相对 Volcano 模型的优势体现在三点: 其一, 批处理把解释与虚调用的开销摊薄到 2048 行上, 单位数据的调度成本大幅下降; 其二, 对整批数据做紧凑循环, 既贴合 CPU 流水线, 也便于编译器生成 SIMD 指令; 其三, 某些向量格式可以在不解压的情况下直接参与计算, 即“压缩执行 (compressed execution)”。

17.2.3 DataChunk 与 Vector

DataChunk 是一组 Vector 的集合, 是执行期间在算子间传递的基本单位。例如 `PhysicalProjection` 算子就用一个 DataChunk 表示一个列列表。Vector 则是执行期内存中保存单类型数据的容器。DuckDB 允许同一逻辑数据以不同物理格式 (vector formats) 表示, 从而更紧凑, 并可能实现压缩执行。

17.2.4 向量格式

DuckDB 支持多种向量格式:

- **Flat Vector (扁平向量)**: 物理上就是一段连续数组, 是标准的未压缩格式, 逻辑表示与物理表示完全一致。
- **Constant Vector(常量向量)**: 物理上只存一个常量值, 适合重复值 (如 `SELECT 1st || 'duckdb'` 中的字面量), 也由存储层从“常量压缩 (Constant Encoding)”解压时产生。
- **Dictionary Vector (字典向量)**: 由一个 child vector 加一个指向 child 的下标数组 (selection vector) 组成, 存储层从“字典压缩 (Dictionary Encoding)”解压时产生, 使数据在执行期保持压缩。
- **Sequence Vector (序列向量)**: 由一个 offset 与一个 increment 值表示, 适合存储递增序列, 通常用于行标识符 (rowid)。
- **Unified Vector Format (统一向量格式)**: 为简化编程而生——任何 Vector 都可转换为这种“通用视图”, 避免为每种向量格式的组合编写特化代码 (否则会产生组合爆炸)。

17.2.5 复杂类型的内存布局

字符串用 `string_t` 结构表示。长度不超过 12 字节的短字符串被内联 (inlined) 进结构体本身; 更长的字符串则用指针指向 auxiliary string buffer。该结构同时携带 length 字段与 4 字节 prefix 字段——prefix 用于在比较时做 early-out 优化, 前缀不符即可免追指针, 从而规避 `strlen` 与空指针检查。List、Struct、Map、Union 等复杂类型则分别由 child vector 与元信息 (如 `list_entry_t` 的 offset 与 length) 组合而成。

压缩执行正建立在这些格式之上: 字典向量与常量向量在解压后仍以压缩形态驻留内存, 算子可直接在其上计算, 省去完整解压的代价。

17.3 列式存储与压缩

17.3.1 Row Group 与列式布局

DuckDB 的数据文件采用列式存储，并按 Row Group（行组）做水平分区——这一概念等价于 Parquet 的 row group。同一列的数据在物理上连续存放，使分析查询只读取涉及的列，天然契合 OLAP 负载。

默认情况下，DuckDB 仅对持久化数据库做压缩，内存数据库不压缩；可用 `ATTACH ... (COMPRESS)` 为内存库开启压缩。Row Group 的大小可通过 `ATTACH` 选项指定：

```
ATTACH '/tmp/analytics.db' AS analytics (ROW_GROUP_SIZE 16384);
```

17.3.2 轻量压缩算法族

DuckDB 采用“轻量压缩（lightweight compression）”策略，追求压缩与解压/执行之间的低开销平衡，而非追求极限压缩率。可用的磁盘压缩算法包括：

- Constant Encoding（常量编码）
- RLE（游程编码）
- Bit Packing（位打包）
- Frame of Reference（FOR，偏移基准编码）
- Dictionary Encoding（字典编码）
- FSST（VLDB 2020 提出的字符串压缩）
- ALP（SIGMOD 2024 提出的自适应无损浮点压缩）
- Chimp（VLDB 2022 提出的浮点压缩）
- Patas（浮点压缩）
- Zstd（v1.2.0 引入的通用压缩）

具体选用哪种算法取决于存储版本（storage version），部分算法需显式设定存储版本才能启用。

17.3.3 存储文件头与版本兼容

数据文件以 `uint64_t` 校验和开头，随后是 4 个魔数字节 DUCK，再是 `uint64_t` 存储版本号。可用 `hexdump -n 20 -C file.db` 或 Python 的 `struct` 模块读取魔数与版本。

存储格式自 v0.10 起保证向后兼容（新版本能读旧版本写出的文件）；向前兼容（旧版本读新版本文件）仅“尽力而为（best effort）”。本书基线为 DuckDB v1.5.5，对应的存储版本号是 68；v1.0–v1.5 默认创建的是 storage version 64（即 v1.0.0 格式）。

17.3.4 磁盘占用经验值

压缩效果与数据类型、数据分布及所选压缩方法密切相关。经验上：约 100 GB 未压缩的 CSV 载入 DuckDB 后约需 25 GB 磁盘；约 100 GB 的 Parquet 载入后约需 120 GB（Parquet 本身已压缩，DuckDB 的列式布局在某些分布下反而需要更多空间）。这提醒我们：DuckDB 并不总比源文件小，压缩收益取决于数据特征。

17.4 查询优化器

DuckDB 的优化器对逻辑查询树运行下列优化：

17.4.1 Expression Rewriter

Expression Rewriter 简化表达式并做常量折叠（constant folding）——把能在编译期求值的部分直接算出结果，例如把 `1 + 2` 替换为 `3`，把 `WHERE year = 2020 + 1` 中的常量表达式提前求值，减少运行期重复计算。

17.4.2 Filter Pushdown (谓词下推)

Filter Pushdown 把过滤条件下推进计划树，并在等价集 (equivalency sets) 上复制过滤条件，使过滤尽早发生、减少上游算子处理的数据量。它还会剪掉那些“静态求值为 false”而必然为空的子树，直接从计划中删除无意义的扫描。

17.4.3 Join Order Optimizer

Join Order Optimizer 使用动态规划 (dynamic programming) 重排 join 顺序，具体采用论文《Dynamic Programming Strikes Back》中的 DPhyp 算法。多表 join 的排列组合随表数指数增长，DPhyp 通过动态规划在合理代价内寻找较优的连接顺序，对分析查询的多表关联尤为关键。

17.4.4 Common Sub Expressions

Common Sub Expressions (公共子表达式提取) 从 projection 与 filter 节点中抽取公共子表达式，避免同一段计算在多处重复执行。这对含有重复复杂表达式的查询能显著降低开销。

17.4.5 In Clause Rewriter

In Clause Rewriter 把大的静态 IN 子句重写为 MARK join 或 INNER join。当 IN 列表很长时，这种改写让查询能走更高效连接路径，而非逐元素比对。

17.5 事务与 MVCC、乐观并发

17.5.1 进程内模式下的两种并发选项

DuckDB 在进程内 (in-process) 模式下提供两类并发选项：读写模式 (read-write) 下一个进程可读可写；只读模式 (read-only) 下多个进程可同时读，但无人能写 (通过 `access_mode='READ_ONLY'` 设定)。读写模式的核心由 MVCC (多版本并发控制) 加乐观并发控制 (optimistic concurrency control) 支撑，支持多线程写入，但全部发生在单一写进程之内。

采用这种模型的动机是把数据缓存在 RAM 以加速分析查询，并缓存函数指针、catalog 等，使同一连接的后续查询更快。catalog 在进程内被缓存正是这一动机的直接体现。

17.5.2 单进程内的写并发规则

只要没有写冲突，多个并发写都会成功；append (追加插入) 永不冲突，即便写入同一张表。多个线程可同时更新不同的表，或更新同一张表的不同子集。当两个线程试图同时编辑 (update / delete) 同一行时，乐观并发控制生效——第二个线程以冲突错误失败。常见冲突信息为 Transaction conflict: cannot update a table that has been altered!，标准应对是重跑 (rerun) 该事务。

这种乐观并发控制适合读密集的分析系统：大多数查询互不修改数据，冲突概率低；一旦发生写写冲突，以乐观失败告终并要求重试，比悲观锁的常驻开销更轻。

17.5.3 多进程写与文件锁

DuckDB 原生格式不支持多进程并发写。需要多进程写入时，须经 Quack 远程协议 (把 DuckDB 变成客户端-服务器数据库，目前为 beta) 或 DuckLake + PostgreSQL catalog (生产级稳定方案)。DuckDB 用文件锁处理并发访问请求，在网络存储或共享目录访问时需格外小心。

17.5.4 隔离级别

DuckDB 提供快照隔离 (snapshot isolation)，表现为 repeatable reads：一个事务在其生命周期内看到的是启动时刻的一致性快照，不受其他已提交事务干扰。

只读挂载的示例如下：

```
ATTACH 'shared.db' (ACCESS_MODE READ_ONLY);
```

17.6 ART 索引

DuckDB 使用 Adaptive Radix Tree (自适应基数树, ART) 作为索引的原语数据结构, 主要用于主键与等值查找等场景。其设计目标是在分析负载下保持低内存占用与高查找效率。

17.6.1 适用场景与单列限制

ART 索引有一个重要限制: 只有单列查询才能走 index scan——复合条件或涉及多列的范围扫描无法利用 ART 索引。这与 B-Tree 的多列前缀命中不同, 是 ART 在 DuckDB 中的适用边界。创建 ART 索引的语法与普通索引一致:

```
CREATE INDEX idx_orders_cust ON orders (o_custkey);
```

17.6.2 VACUUM 行为差异与实现参考

另一处与 PostgreSQL 的行为差异在于: 在 DuckDB 中, VACUUM 会跳过带有 ART 索引的表, 且不回收其空间。对建有 ART 索引的表执行 VACUUM 并不能像预期那样释放已删除数据占用的空间, 这一点与 PostgreSQL 中 VACUUM 回收空间的行为不同, 运维时须特别留意。

VACUUM;

关于 ART 的逐节点实现细节, 本书所依据的官方 internals 文档未设独立章节, 权威内容可参考《Design and Implementation of DuckDB Internals》(DiDi) 库条目及对应源码 (src/storage 等)。

17.7 本章小结

本章沿“一条 SQL 如何被执行”的主线拆解了 DuckDB 的内部机制, 可归纳为四个相互支撑的支柱:

- **分层流水线**: Parser 仅做词法拆解、不感知 catalog 与类型; Binder 借 catalog 完成表/列/类型的语义绑定; Logical Planner 产出逻辑树, Optimizer 改写, Column Binding Resolver 把列名引用降级为 DataChunk 下标, Physical Plan Generator 产出物理算子树, 最终由向量化执行引擎运行。
- **向量化执行**: 推送式模型以 STANDARD_VECTOR_SIZE = 2048 行的 DataChunk 为批处理单位在算子间流动, 相较 Volcano 的逐行 next() 显著降低虚调用开销、利于 SIMD; Flat / Constant / Dictionary / Sequence 等向量格式与 Unified Vector Format 共同支撑了“压缩执行”。
- **列式存储与轻量压缩**: 数据按 Row Group 水平分区; 磁盘上采用 Constant / RLE / Bit Packing / FOR / Dictionary / FSST / ALP / Chimp / Patas / Zstd 等低开销压缩; 100 GB CSV 约落 25 GB, 100 GB Parquet 约需 120 GB。
- **MVCC 与乐观并发**: 进程内以 MVCC + 乐观并发支持多写线程, 写写冲突以失败告终需重跑; 多进程写须经 Quack 或 DuckLake; 加上仅单列可走 index scan 的 ART 索引, 共同构成 DuckDB 在进程内分析场景下的并发与索引图景。

掌握这些内部原理, 有助于在实际调优时判断一条查询为何走某个计划、为何占用某等磁盘空间, 以及并发写入应如何组织。

第七部分 实战与总结

第 18 章 应用场景实践

前十七章沿着“安装配置 → SQL 查询 → 数据工程 → 客户端与生态集成 → 扩展机制 → 性能调优 → 原理剖析”的脉络, 逐步拆解了 DuckDB 的能力边界与内部机理。能力终究要落到具体的工作负载上才有意义。

义。DuckDB 的出身决定了它并非用来取代传统服务式数据库，而是以进程内 (in-process)、列式存储 (columnar storage)、向量化执行 (vectorized execution) 的特质，在若干类场景中提供低摩擦的分析能力：本地随手探索数据、给事务型数据库补上分析侧、搭建轻量 ETL (Extract-Transform-Load) 管线、把引擎嵌入应用做就地分析、以及与 Python 数据科学生态互为表里。本章逐一给出这些场景的落地方式与取舍，所有示例均以 DuckDB v1.5.5 为基线，代码均可直接运行。

18.1 本地数据分析与探索式查询

探索式分析 (exploratory data analysis) 的痛点往往是“想看一眼数据却要先搭一套环境”。DuckDB 的进程内特性恰好消解了这道门槛：不需要启动服务，不需要建表，把 CSV、Parquet 甚至远程文件直接当作查询对象即可。零成本试错，是这一场景最贴切的注解。

18.1.1 CLI 随手查文件

DuckDB 的命令行工具 (CLI) 开箱即用。给定一个 Parquet 文件，无需任何导入步骤，直接用 FROM 把它当表查询：

```
duckdb -c "SELECT year, count(*) AS n
          FROM 'sales.parquet'
          GROUP BY year ORDER BY year;"
```

单条 -c 参数即可执行整段 SQL 并退出，适合写进 shell 管道或定时任务。文件格式由扩展名自动识别，CSV 同样适用：

```
duckdb -c "SELECT region, sum(amount) AS total
          FROM 'orders.csv'
          WHERE order_date >= '2024-01-01'
          GROUP BY region;"
```

面对陌生的数据集，先摸清结构与分布比急着写查询更务实。DESCRIBE 看 schema，SUMMARIZE 则一次性给出每列的最小值、最大值、近似唯一值数、均值、分位数与 NULL 占比：

```
duckdb -c "SUMMARIZE SELECT * FROM 'sales.parquet';"
duckdb -c "DESCRIBE SELECT * FROM 'sales.parquet';"
```

远程文件也无需预先下载，DuckDB 会把列裁剪 (projection pushdown) 与过滤下推 (filter pushdown) 直接作用到远程读取上，只传输所需部分：

```
duckdb -c "SELECT count(*) FROM 'https://blobs.duckdb.org/data/holdings.parquet';"
```

CLI 默认以 Duckbox 表格呈现结果，配合 YouPlot 这类终端绘图工具，更能在无图形界面的环境里快速做可视化：

```
duckdb -c "COPY (SELECT region, sum(amount) AS total
                FROM 'orders.csv' GROUP BY region)
          TO '/dev/stdout' (FORMAT csv, HEADER)"
| uplot bar -d, -H -t "Region Totals"
```

18.1.2 Jupyter 中的交互分析

笔记本环境里，DuckDB 既能以原生 Python 连接工作，也能借由 jupysql 把代码单元变成 SQL 单元。原生接入只需加载扩展并指向一条连接：

```
import duckdb
%load_ext sql
conn = duckdb.connect()
%sql conn --alias duckdb
```

此后 %sql 执行单行查询、%%sql 把整段单元当作 SQL 来跑，结果默认呈现为 pandas DataFrame。需要把本地 DataFrame 也纳入查询范围时，可执行：

```
%sql SET python_scan_all_frames = true
```

这会让当前命名空间中所有可见的 pandas DataFrame 都能被 SQL 直接引用，省去手动注册的步骤。%sqlplot 则把绘图计算下推到 SQL 引擎内部完成，避免把超大数据集整体搬进 Python：

```
%sqlplot histogram --column amount --with orders.csv
```

借助 httpfs 扩展，笔记本同样能查询云端 Parquet，把远程数据探查与本地建模整合在同一工作流中：

```
con = duckdb.connect()
con.execute("INSTALL httpfs; LOAD httpfs;")
con.execute("SELECT * FROM read_parquet('s3://bucket/large.parquet') LIMIT 10;")
```

18.1.3 零成本试错的工作流

探索式分析的价值在于“快速失败、快速迭代”。DuckDB 的内存库 (:memory:) 天然契合这种节奏：每条查询独立、互不影响，试错不会产生遗留的表或文件。即便改用持久库做稍长期的草稿，单个 .duckdb 文件也方便随项目归档或删除，不构成运维负担。

当探索产出有价值的中间结果，再把它固化下来也不费事——内存库可通过 COPY FROM DATABASE 整体落盘为持久库：

```
ATTACH 'exploration.db';
COPY FROM DATABASE memory TO exploration;
DETACH exploration; -- 目标文件需事先不存在
```

这条路径把“随手试”与“正式沉淀”连成一条平滑曲线：分析师不必在探索阶段就为 schema 设计纠结，待模式稳定后再固化即可。

18.2 作为 SQLite 的 OLAP 补充

SQLite 是嵌入式事务处理 (OLTP) 的事实标准，几乎内置在所有手机与桌面软件中；它在小事务、高并发写入的场景表现稳健，却不擅长跨多表、跨多行的聚合分析。DuckDB 恰好在另一侧补足：同一个应用，让 SQLite 守住事务型数据，把分析型查询交给 DuckDB，二者通过文件与 ATTACH 桥接，无需引入额外的服务组件。

18.2.1 分工：事务与分析各司其职

典型形态是移动端或桌面应用用 SQLite 持续记录交易流水、用户行为等行级写入；运营侧要回答“各区域本月成交额”“用户留存趋势”这类问题，则把 SQLite 文件交给 DuckDB 直接扫描。SQLite 的强项在于点查、小批量写入与 ACID 保证，DuckDB 的强项在于列式扫描、向量化聚合与大于内存 (larger-than-memory) 的处理能力——两者目标不同，互补而非竞争。

需要强调的是，DuckDB 通过 sqlite 扩展读取 SQLite 文件是只读桥接，分析查询不会改写源库，这与把事务库与分析库物理隔离的安全诉求一致。

18.2.2 通过 sqlite 扩展桥接

桥接的第一步是安装并加载扩展：

```
INSTALL sqlite;
LOAD sqlite;
```

随后即可用 sqlite_scan 直接扫描某张表，或用 ATTACH 把整个 SQLite 库挂为 DuckDB 的一个 schema：

```
-- 单表扫描
SELECT * FROM sqlite_scan('app.db', 'transactions');
```

```
-- 整库桥接，之后以 schema 形式引用
ATTACH 'app.db' AS app (TYPE sqlite);
SHOW TABLES FROM app;
```

ATTACH ... (TYPE sqlite) 之后，app 下的所有表都像普通 DuckDB 表一样可查，列类型按 SQLite 的存储类自动映射。桥接是即时且轻量的，挂上即可分析，摘下 (DETACH) 即恢复 SQLite 文件的独立状态，不污染任何一方。

18.2.3 一次桥接查询示例

设想 app.db 内有一张 transactions 表，记录每笔订单的地区、金额与时间。用 DuckDB 承接其分析侧只需一次桥接：

```
INSTALL sqlite;
LOAD sqlite;
ATTACH 'app.db' AS app (TYPE sqlite);

SELECT
    region,
    DATE_TRUNC('month', order_ts) AS month,
    sum(amount)                    AS revenue,
    count(*)                      AS orders
FROM app.transactions
WHERE order_ts >= DATE '2024-01-01'
GROUP BY region, month
ORDER BY region, month;
```

这段查询对 SQLite 文件做列式读取与向量化聚合，源库保持只读。若分析结论需要回流到事务系统（例如把汇总结果写回 SQLite 的报表表），可先把结果写成 Parquet 或以 COPY 导出，再由应用侧导入——保持桥接单向，避免分析负载反过来拖累事务写入。

18.3 轻量级数据管线

ETL 管线的本质是把分散、异构的数据抽取出来，清洗转换后落到统一的存储。重型方案往往需要专门的调度服务与计算集群，而许多中小规模场景里，DuckDB 自身就能充当内存计算层 (in-memory compute layer)：读取多源文件或数据库、在进程内完成连接与聚合、再把结果写出为 Parquet 或持久库，整个流程一个二进制文件即可驱动。

18.3.1 多源读取与内存计算

DuckDB 的 read_csv 与 read_parquet 是查询级别的函数式读取器，可在一条 SQL 内同时打开不同来源并 join：

```
CREATE TABLE enriched AS
SELECT
    c.customer_id,
    c.name,
    o.order_date,
    o.amount,
    o.amount * COALESCE(e.rate, 1.0) AS amount_local
FROM read_csv('orders.csv', auto_detect = true) AS o
JOIN read_parquet('customers.parquet') AS c
```

```

    ON o.customer_id = c.customer_id
LEFT JOIN read_parquet('fx_rates.parquet')          AS e
    ON c.country = e.country
    AND o.order_date = e.rate_date
WHERE o.order_date >= DATE '2024-01-01';

```

CREATE TABLE ... AS (CTAS) 把结果物化为当前连接中的一张表，后续步骤直接引用，不必重复扫描源文件。当源 schema 偶发不一致（例如某批文件多出或少了几列），可用 UNION ALL BY NAME 技巧在转换时动态对齐列，而非手工修补每一份文件。

18.3.2 写出 Parquet 或持久库

中间结果既可以持久化为 DuckDB 单文件库，也可以导出为 Parquet 供下游继续消费。写出 Parquet 时，可显式指定压缩与 row group 大小以贴合下游读取习惯——row group 在 100K~1M 行区间对并行扫描最为友好：

```

COPY enriched
TO 'enriched.parquet'
(FORMAT parquet, COMPRESSION zstd, ROW_GROUP_SIZE 200000);

```

若下游是另一个 DuckDB 库（或同一管线不同阶段的产物需要集中管理），则直接 COPY ... TO 进已 ATTACH 的目标库：

```

ATTACH 'warehouse.db' AS wh;
COPY enriched TO wh.fact_orders;

```

数据需要增量更新而非全量重写时，MERGE 语句（自 v1.4.0 引入）可用单条语句完成按主键的插入、更新与软删除，契合缓慢变化维（SCD Type 2）等常见 ETL 语义：

```

MERGE INTO wh.fact_orders AS target
USING enriched AS source
    ON target.customer_id = source.customer_id
    AND target.order_date = source.order_date
WHEN MATCHED THEN UPDATE SET amount = source.amount
WHEN NOT MATCHED THEN INSERT *;

```

18.3.3 调度中的批处理脚本

把上面的 SQL 收进一个脚本，再交给 cron 或 Airflow 定时触发，便是一条完整的轻量管线。DuckDB 进程内、无服务的特性意味着脚本启动即算运行、结束即算退出，没有常驻进程需要看护：

```

import duckdb

con = duckdb.connect("pipeline.db")
con.execute("INSTALL httpfs; LOAD httpfs;")
con.execute("""
    CREATE OR REPLACE TABLE staging AS
    SELECT * FROM read_parquet('s3://bucket/incoming/*.parquet');
""")
con.execute("""
    INSERT INTO fact_daily
    SELECT day, sum(amount) FROM staging GROUP BY day;
""")

```

批量加载阶段应尽量避免主键、唯一等约束——它们会触发索引维护与完整性检查，显著拖慢摄入；确有完整性需要时，遵循“先加载、后加约束”的原则，加载完成再补主键，比边加载边校验快数倍。

18.4 嵌入式分析中台与应用

中台 (middle platform) 概念常被等同于一套独立的分析服务，但很多产品的分析需求其实轻到不必起服务：报表看板、运营埋点查询、本地 BI (business intelligence) 工具，都可以把 DuckDB 引擎直接嵌入应用进程，就地分析应用自身产生的数据。

18.4.1 进程内就地分析

DuckDB 不依赖任何后台进程，引擎与宿主应用共享同一地址空间。应用启动时顺带打开一个持久库，后续查询就是一次函数调用，没有网络往返、没有连接池耗尽之忧：

```
import duckdb

con = duckdb.connect("analytics.duckdb")
con.execute("INSERT INTO events SELECT * FROM read_csv('events.csv')")
rows = con.execute(
    "SELECT device, count(*) AS n FROM events GROUP BY device"
).fetchall()
print(rows)
```

持久库的默认线程数等于 CPU 核数，默认 `memory_limit` 为物理内存的 80%——分析即服务 (analytics-as-a-service) 被压缩到“嵌入一个库”的体量，部署复杂度趋近于零。需要限制资源占用时，通过连接参数或 `pragma` 显式设小即可，例如受限容器里把 `threads` 调低以匹配分配的 vCPU。

18.4.2 轻量分析中台的实现

一个内嵌的分析中台，对外暴露的往往不是 SQL 字符串，而是若干预定义的查询接口。DuckDB 的 `prepared statement` (预处理语句) 适合这类“同结构、不同参数”的反复查询，缓存解析与计划，降低小查询开销：

```
import duckdb

con = duckdb.connect("analytics.duckdb")
con.execute("PREPARE get_region AS "
    "SELECT sum(amount) FROM orders WHERE region = $1;")
for region in ("East", "West", "North"):
    print(con.execute("EXECUTE get_region('" + region + "')").fetchone())
```

查询逻辑沉淀在应用内部，数据驻留在本地单文件库，既绕开了独立数仓的运维，又保留了 SQL 的表达力。结合 `https` 与云存储 `secret`，这类中台还能透明地把远程 Parquet 纳入计算，把“本地分析”自然扩展为“本地 + 云端混合分析”而不改变代码骨架。

18.4.3 并发与连接管理

进程内并不意味着不能并发。同一进程内的多 `reader/writer` 线程，各自用 `cursor()` 拿到线程本地的连接句柄，即可并发读写同一 DuckDB 文件——DuckDB 已在单条查询内部尽力并行，多数场景下单连接已足够，多连接适合需要并行执行多个独立操作的场合：

```
import duckdb
import threading

con = duckdb.connect("analytics.duckdb")

def worker():
    cur = con.cursor()  # 线程本地句柄
    cur.execute("INSERT INTO events VALUES (1, 'click')")
```

```

print(cur.execute("SELECT count(*) FROM events").fetchone())

threads = [threading.Thread(target=worker) for _ in range(4)]
for t in threads: t.start()
for t in threads: t.join()

```

连接本身是宝贵资源，复用同一连接优于频繁断开重连；若应用是多线程服务，用连接池或在各线程内复用句柄都比每请求新建连接更经济。单事务只能写入一个 attached 数据库，且写写冲突会乐观失败 (optimistic failure)，并发写入方需对重试做好处理。

18.5 与数据科学生态协同

Python 是 DuckDB 的一级客户端，二者在数据科学生态里的协作方式是本章其余场景的技术底座。pandas、Polars、PyArrow 与 DuckDB 之间并非“谁替代谁”，而是通过 replacement scan (替换扫描) 与零拷贝 (zero-copy) 互为查询引擎与存储后端。

18.5.1 replacement scan 无缝互操作

replacement scan 让当前作用域中可见的 DataFrame 变量无需任何注册，就能直接出现在 SQL 的 FROM 子句里。DuckDB 遇到一个 DataFrame 变量名时，自动把它替换为一个可扫描数据源：

```

import duckdb
import pandas as pd

df = pd.DataFrame({"i": [1, 2, 3], "j": ["one", "two", "three"]})

```

```

# DataFrame 直接当表查，无需 register
result = duckdb.sql("SELECT sum(i) AS total_i, max(j) AS last FROM df").df()
print(result)

```

把 DataFrame 反向写回 DuckDB 同样简洁，CREATE TABLE ... AS SELECT 或 INSERT INTO 都能直接消费 DataFrame 变量名；按列名对齐可加 BY NAME，或用连接的 append 方法做批量追加：

```

duckdb.sql("CREATE TABLE my_table AS SELECT * FROM df")
con = duckdb.connect()
con.append("my_table", df, by_name=True)      # 按列名对齐追加

```

replacement scan 只对调用处可见的变量生效，同名冲突时优先级为：显式 register 的视图 > 原生表/视图 > replacement scan。它把“SQL 与 DataFrame 之间反复搬运”的心智负担降到最低——数据停留在 Python 里，查询却拥有完整的 SQL 能力。

18.5.2 把 DuckDB 当查询引擎驱动 DataFrame

数据科学工作流里，清洗与聚合往往用 SQL 表达更清晰，而上下游建模又依赖 DataFrame 接口。DuckDB 恰好充当那个“查询引擎”：DataFrame 进、DataFrame 出，中间的计算由 DuckDB 的向量化执行完成，而非逐行 Python 循环：

```

import duckdb
import pandas as pd

orders = pd.DataFrame({
    "region": ["East", "West", "East", "West"],
    "amount": [100.0, 250.0, 150.0, 80.0],
})

agg = duckdb.sql("""

```

```

SELECT region, sum(amount) AS total, count(*) AS n
FROM orders
GROUP BY region
ORDER BY total DESC
""").df()

```

```
print(agg)
```

这种“以 SQL 驱动 DataFrame”的模式把声明式查询的简洁性带给 pandas 用户，又规避了 pandas 在大表上逐行操作的内存与速度瓶颈。当结果超大，可改用 `fetch_df_chunk` 分块取回，块大小等于 `2048 × vector_multiple`——这里的 2048 正是 DuckDB 向量化的基本单位 `STANDARD_VECTOR_SIZE`。

关系 API (relational API) 则提供另一条延迟求值的路径：`from_df()` 拿到 Relation 后链式 `filter/aggregate/order/limit`，只有真正物化时才计算，且过滤与投影会被合并下推，避免中间结果全量物化：

```

import duckdb
con = duckdb.connect()
rel = con.from_df(orders)
out = rel.aggregate("region, sum(amount) AS total", groups="region").order("total DESC").df()
print(out)

```

18.5.3 与 Polars / Arrow / Ibis 衔接

DuckDB 的互操作不限于 pandas。Polars DataFrame 同样可被 SQL 直接引用，从 DuckDB 取数则用 `.pl()` (传 `lazy=True` 可得 LazyFrame，延续 Polars 的延迟求值)：

```

import duckdb
import polars as pl

p_df = pl.DataFrame({"x": [1, 2, 3], "y": [10, 20, 30]})
duckdb.sql("SELECT x, sum(y) AS s FROM p_df GROUP BY x").pl()

```

与 Apache Arrow 的互操作走零拷贝路径，列式缓冲在内存中直接共享。对外推荐 `to_arrow_reader(chunk_size)` 以流式分块读取，避免一次性物化超大结果：

```

import duckdb
con = duckdb.connect()
reader = con.execute("SELECT range AS i FROM range(1000000)").to_arrow_reader(2048)
for batch in reader:
    pass # 逐 RecordBatch 处理

```

Ibis 这类 DataFrame 库则以 DuckDB 为默认后端，提供链式、延迟求值的表达式 API，真正取数 (`to_pandas()`) 时才由 DuckDB 执行计算。从 pandas、Polars 到 Arrow、Ibis，DuckDB 在其中扮演的是“统一查询平面”：无论数据以哪种形态进入，都能用同一套 SQL 与向量化引擎完成分析，再按下游需要导回对应形态。

18.6 本章小结

本章把 DuckDB 的能力映射到五类真实工作负载，提炼出几个贯穿始终的取舍：

- **本地探索零门槛**：CLI 用 `FROM 'file.parquet'` 直接查文件，`SUMMARIZE/DESCRIBE` 快速摸底，内存库让试错无遗留负担，有价值的结果再 `COPY FROM DATABASE` 落盘沉淀。
- **SQLite 的分析侧补位**：通过 `sqlite` 扩展以只读方式 `ATTACH` 桥接，让事务库守住 OLTP、DuckDB 承接聚合分析，桥接单向、互不污染。
- **轻量 ETL 自治**：`read_csv/read_parquet` 在一条 SQL 内多源 join，`CREATE TABLE ... AS` 物化中间结果，`COPY` 写出 Parquet 或持久库，`MERGE` (v1.4.0 起) 承担增量更新；批量加载避开约束、先加载后加主键。

- **嵌入式分析中台**：引擎进程内就地分析，无服务、无连接池之忧；prepared statement 服务参数化查询，多线程借 cursor() 并发，连接复用优于频繁重连。
- **数据科学生态底座**：replacement scan 让 DataFrame 无需注册即可当表查，零拷贝贯通 Arrow；以 SQL 驱动 DataFrame 规避逐行 Python 瓶颈，Polars、Arrow、Ibis 各取所需，DuckDB 充当统一查询平面。

上述场景共享同一个内核事实：DuckDB 是进程内、列式、向量化的分析引擎。正因无服务、单文件、与生态互操作成本低，它才能在同一份代码里既做随手查询、也做管线计算、还做嵌入式中台——场景切换的边际成本，往往只是一个文件的路径。

第 19 章 总结与展望

从第一章把 DuckDB 锚定为“跑在进程里、为分析而生”的列存数据库，到各章逐一拆解它的存储、执行、扩展与生态，本书已走完从入门到进阶的全程。收尾之际，与其罗列零散要点，不如回到三个根本问题：DuckDB 究竟是什么、它该被用在何处、它正走向何方。把这三个问题想清楚，读者便能在纷繁的工具选项里为它找到恰如其分的生态位，也才能在版本更迭中辨明哪些能力是底色、哪些只是演进中的浪花。

19.1 核心能力回顾

DuckDB 官网用一组关键词概括其设计取向：Simple（简单）、Portable（可移植）、Feature-Rich（功能完备）、Fast（快）、Extensible（可扩展）、Free（自由）、Thoroughly Tested（经严格测试）。这些标签并非营销辞令，而是与其技术内核——对应。下文沿六条主线复述支撑全书的核心能力。

19.1.1 进程内、列式、向量化的 OLAP 引擎

进程内（in-process）是 DuckDB 最本质的属性。它不运行独立的服务进程（server process），引擎被编译进宿主程序、与调用方共享同一地址空间，因此省去了客户端—服务器（client-server）模型下的网络往返与序列化开销；官方甚至把整个源码树合并（amalgamation）为头文件与实现文件两份产物，零外部依赖（no external dependencies），部署轻量到几乎无感。其优化目标则是分析型负载（OLAP，Online Analytical Processing）——相对长时、复杂、扫描大块数据的查询，而非高频短小的事务。

支撑这一目标的两条技术支柱是列式存储（columnar storage）与向量化执行（vectorized execution）。列式存储让分析查询只读所需列、同列数据类型一致带来更高压缩率；向量化执行则以固定大小的向量（Vector）为批处理单位，一次操作处理一整批值。这个批量由常量 STANDARD_VECTOR_SIZE 决定，默认 **2048 行**，所有算子都针对该批量优化，执行期间在算子间流动的基本单位是 DataChunk（一组 Vector 的集合）。配合轻量压缩（lightweight compression）、多核并行（multi-core parallelism）以及超内存落盘（out-of-core，spill to disk），DuckDB 既能跑在笔记本上，也能处理远超内存的数据集。这套引擎的向量化与 Morsel 驱动并行（Morsel-driven parallelism）思路，可追溯至 MonetDB/X100 与慕尼黑工业大学的相关研究——DuckDB 站立在坚实的学术肩膀之上。

19.1.2 单文件数据库与两种存储形态

DuckDB 最常见的形态是单文件数据库，文件扩展名 .duckdb / .db / .ddb 皆可，因为磁盘格式与后缀无关；它也支持纯内存模式。两种形态共享同一套 SQL 与磁盘格式，可经由 ATTACH 与 COPY ... FROM DATABASE 相互搬运。一个常被忽视的反直觉事实是：压缩的持久库常常比未压缩的内存表更快——持久化数据库默认启用压缩，而内存表默认不压缩，TPC-H Q1 的实测显示压缩能使查询提速约 8 倍。默认运行参数也值得记住：memory_limit 取物理内存的 **80%**，threads 等于 CPU 核数，绝大多数场景下这套宽容（forgiving）的默认配置已能自动达到高性能。

19.1.3 丰富而现代的 SQL 能力

DuckDB 并非“阉割版”的嵌入式引擎，它的 SQL 覆盖面足以胜任严肃的数据分析。窗口函数（window functions）借助段树聚合（Segment Tree Aggregation）高效实现；公共表表达式（CTE）支持递归写

法，并在 v1.5 引入 USING KEY 把递归中间结果当作带键字典，显著改善性能与内存；复杂类型（complex types）如 STRUCT、LIST 及其嵌套组合让半结构化数据无需先拍平即可建模；连接（join）家族涵盖常规等值连接、时态模糊查找的 AsOf Join，以及基于 IEJoin 算法的高效不等式连接；自 v1.4.0 起还提供 MERGE 语句，可在一句话内完成缓慢变化维（SCD Type 2）式的插入、更新与软删除。其 SQL 解析器直接复用 PostgreSQL 的 libpg_query，语法与各客户端通用。下面用一段窗口函数叠加 CTE 的查询，展示这种表达力：

```
-- 窗口函数 + CTE：计算各部门薪资排名及其与部门均值的差距
WITH dept_stats AS (
    SELECT department,
           emp_id,
           salary,
           avg(salary) OVER (PARTITION BY department) AS dept_avg
    FROM employees
)
SELECT department,
       emp_id,
       salary,
       salary - dept_avg                                AS diff_to_avg,
       rank() OVER (PARTITION BY department
                   ORDER BY salary DESC)                AS rnk
FROM dept_stats
ORDER BY department, rnk;
```

19.1.4 可插拔的扩展体系

DuckDB 的灵活扩展机制允许定义新的数据类型、函数、文件格式乃至 SQL 语法。事实上，诸多关键能力本身就是以扩展形式实现的：Parquet、JSON、时区（ICU）、HTTP(S) 与 S3 协议（httpfs）均属此类。需要提醒的是，扩展的二进制产物严格绑定具体的 DuckDB 版本与平台，升级引擎时必须重装匹配版本的扩展，否则无法加载。在此之上，社区扩展（community extensions）进一步打开了自定义类型的空间——图查询（DuckPGQ，基于 SQL/PGQ）、全文检索（fts）等均来自社区贡献，使用者按需安装 INSTALL ... FROM community 后 LOAD 即可。

19.1.5 与数据科学生态的无缝互操作

DuckDB 把“就地分析”的理念贯彻到与数据科学生态的衔接上。Python 是其一级（Primary）客户端，提供与 SQL 语法一致、与磁盘格式互通的完整能力。借助 replacement scan（替换扫描）机制，pandas、Polars、PyArrow、NumPy 等对象可直接在 FROM 子句中当作表来查询，无需显式注册；与 Apache Arrow 的互操作尽量走零拷贝（zero-copy）路径，内存中的列式缓冲被直接共享。在笔记本场景里，Jupyter（配合 jupysql、duckdb-engine）、响应式笔记本 marimo，以及以 DuckDB 为默认后端的 Ibis，都让“在数据分析流中随手写一条 SQL”几乎没有成本。CLI 与 Python 共享同一套 SQL 语法，使技能在端与端之间无缝迁移。

19.1.6 ACID 事务与 MVCC 并发控制

事务安全性由 DuckDB 自研、面向批处理的 MVCC（Multi-Version Concurrency Control，多版本并发控制）提供，具备 ACID 属性。它采用快照隔离（snapshot isolation）：读操作看到的是事务开启时刻的一致性快照，写入行为彼此隔离。需要理解的一项约束是，写写冲突会以乐观方式失败，调用方需捕获并重试事务；单条事务只能写入选定的一个 attached 数据库。这套并发模型服务于“进程内、分析为主”的定位，而非高竞争的并发写入。

19.2 适用与不适用场景研判

再强大的工具也有其边界。给 DuckDB 找准位置，比罗列它的特性更有价值。

19.2.1 典型适用场景

- **本地与嵌入式分析**：当数据本就躺在分析师的机器或应用进程里，进程内引擎省去服务部署与网络开销，即取即查。
- **探索式查询 (ad-hoc analytics)**：面对一份 CSV、Parquet 或 SQLite，无需建库即可直接 SELECT，把文件当数据库来 interrogation，是 DuckDB 的招牌用法。
- **轻量 ETL 与数据整理**：借助 COPY、read_parquet/read_csv 与 Appender，可在单进程内完成中小规模的数据清洗、转换与落地。
- **湖仓与 Parquet 查询**：DuckDB 能直接扫描本地或对象存储上的 Parquet，配合 Hive 分区与 filter pushdown，把列式格式的分析价值直接释放，不必先搬运进专有系统。
- **笔记本内分析**：在 Jupyter、marimo 中，它作为兼具速度与表达力的查询内核，与 pandas、Arrow 等流畅咬合。

19.2.2 边界与不适用地带

DuckDB 无意、也不适合充当高并发、多客户端、需要严格写并发的“主数据库”。其原因根植于进程内模型：写入发生在单一宿主进程内，跨进程的多写者需要协调，而 DuckDB 原生的写并发并不为此设计——那是 PostgreSQL 这类客户端—服务器 OLTP 系统，或 ClickHouse 这类可分布式部署的分析集群的领地。下表把 DuckDB 与常被并列提及的系统放在同一坐标系里。

系统	部署形态	存储模型	主攻负载	多进程写
SQLite	进程内 (in-process)	行式 (row-oriented)	OLTP (事务)	单写者，多进程可读
DuckDB	进程内 (in-process)	列式 (columnar)	OLAP (分析)	单进程内多写线程；跨进程写经商 Quack / DuckLake
PostgreSQL	客户端—服务器	行式 (row-oriented)	OLTP 为主	多客户端并发写
ClickHouse	客户端—服务器	列式 (columnar)	OLAP (分析)	服务端分布式写

需要澄清的是，“不适用”不等于“不能写”。单进程内的多个写线程、乃至进程内多线程并发插入都是 DuckDB 支持且常见的用法；问题只出在跨进程的严格写并发上。把 DuckDB 当作分析库而非主数据库，它就处处得心应手；反过来硬塞进高竞争写入的主库角色，只会与它自身的设计取舍相抵牾。

19.3 生态与未来

回望出发点，DuckDB 押注的是“把分析带到数据所在之处”。这一赌注正沿着两条线索延展：向下扎根湖仓 (lakehouse)，向上突破进程边界。

19.3.1 DuckLake 与湖仓之路

DuckDB 的湖仓格式 DuckLake 于 **2026 年 4 月**发布 1.0 版本。与 DuckDB 原生单文件格式侧重于嵌入式便利不同，DuckLake 这类湖仓格式把元数据与数据解耦，使数据可安放于对象存储之上，分析规模可扩展至 PB 级。这让 DuckDB 的能力从“单机单文件”平滑延伸到“共享存储上的大规模分析”——本地探索与云端湖仓之间不再是非此即彼的切换，而是同一引擎下的连续光谱。

19.3.2 Quack 远程协议：从进程内走向进程外

进程内是 DuckDB 的初心，却也划定了它的天然边界。Quack 协议 (Quack protocol) 正是用来跨过这道边界的——它让 DuckDB 能够提供进程外的远程访问，补足多客户端场景下的短板。该协议已在 **v1.5.3 以 beta 形态**开放，其稳定化被规划于 **v2.0** 版本的里程碑。这意味着 DuckDB 正在谨慎地回答一个长期张力：既要守住嵌入式的零运维体验，又要回应“多人共享同一份分析后端”的现实需求。

19.3.3 社区扩展与开源治理

DuckDB 以极宽松的 MIT License 发布，知识产权由 DuckDB Foundation 持有，开发源自学术团队的公共责任意识。这种治理模式催生了活跃的社区扩展生态：从图查询 DuckPGQ 到全文检索 fts，从 shellfs 到各类自定义类型，使用者与贡献者共同推动能力边界向外生长。第三方汇总的 Awesome DuckDB 仓库则持续收录构建于 DuckDB 之上的项目，是观察生态走向的一扇窗口。与此同时，DuckDB 延续其“经严格测试”的传统——测试套件包含数以百万计的查询，并复用 SQLite、PostgreSQL、MonetDB 的用例，TPC-H 与 TPC-DS 基准亦长期运行，稳定性并非口号。

19.3.4 路线图方向与持续学习建议

从已披露的版本事实，可以读出几条清晰的方向：类型系统持续拓宽（GEOMETRY 于 v1.5 成为内置类型，VARIANT 支持 Parquet 的 shredding）；引擎向湖仓（DuckLake）与进程外（Quack）两端延伸；v2.0 还将把构建语言从 C++11 迁移至 C++17，并把 lambda 单箭头语法默认禁用。这些演进没有改变 DuckDB 的内核定位，只是把它向“更宽的数据形态、更大的协同规模”推开。

对希望持续精进的读者，建议的学习路径是：以官方文档（docs.duckdb.org）为权威基线，跟随 GitHub 仓库的 release notes 追踪版本事实而非道听途说；用真实数据集（譬如文档中常用的荷兰铁路数据集）动手复现书中的查询与调优；在需要时翻看 DuckDB 团队发表于 CIDR、ICDE、VLDB、SIGMOD 的同行评审论文，理解向量化、MVCC、ART 索引、段树聚合背后的原理；并定期浏览社区扩展与 Awesome DuckDB，保持对生态脉动的敏感。把查询计划（EXPLAIN / EXPLAIN ANALYZE）当作日常习惯，比记忆任何参数都更能让你在遇到慢查询时从容定位。

19.4 本章小结

本章把全书收束于三个根本问题：

- **它是什么**：DuckDB 是一个进程内、面向分析（OLAP）的列式数据库，以向量化执行（STANDARD_VECTOR_SIZE=2048）与轻量压缩为性能内核，以单文件数据库为常见形态，以 MVCC 提供 ACID 事务保证。它功能完备（窗口函数、CTE、复杂类型、MERGE、AsOf Join 等），可插拔、零依赖、经严格测试。
- **它用在何处**：适用于本地/嵌入式分析、探索式查询、轻量 ETL、湖仓与 Parquet 查询、笔记本内分析；不适用于高并发多客户端写入的主数据库，以及需要严格跨进程写并发的场景——后者归于 PostgreSQL 或 ClickHouse 集群。单进程内多写线程仍是其常规用法。
- **它走向何方**：DuckLake 1.0（2026-04）把它推向 PB 级湖仓，Quack 协议（v1.5.3 beta，v2.0 稳定化）把能力延伸到进程外；社区扩展与 MIT 治理让生态持续生长；类型系统与湖仓/进程外两端是可见的演进主线。

19.5 全书收束

工具更迭是数据领域的常态，今天被追捧的引擎，未必能定义明天的实践。DuckDB 真正值得记住的，是它所坚持的一个朴素判断：把分析带到数据所在之处，而不是反过来让数据为分析让路。它用进程内消除了服务与网络的距离，用列存与向量化消除了计算与硬件的距离，又用湖仓与远程协议，谨慎地把这道距离的消解推向更广阔的协同规模。

本书从安装与 SQL 起步，经数据工程、客户端集成、扩展机制、性能运维而至原理剖析，如今停在总结与展望。读者至此已握有描述 DuckDB 的词汇、调度它的机制与审视它的眼光——剩下的，是把这些放进真实的数据与真实的问题里去检验。每一次你径直对一份文件写下 SELECT 而不必先搭起一座系统，DuckDB 的设计初衷便已兑现。分析的价值不在于工具本身，而在于它缩短了多少次“从疑问到答案”的路程；愿这趟从入门到进阶的旅程，成为你缩短那路程的一步。

版权与免责声明

本书《DuckDB 数据库从入门到进阶》由 AI 辅助文档生成流程编撰，内容以 DuckDB v1.5.5 官方文档为基线整理而成，仅供学习与技术参考之用。

DuckDB 及其文档由 DuckDB 社区与相关权利人在相应开源许可（主要为 MIT）下发布；本书不主张对 DuckDB 本身的任何知识产权，相关商标与版权归原权利人所有。

免责声明：尽管本书已尽量对照官方文档核验事实与语法，但软件版本演进迅速，文中个别参数、默认值或行为可能随版本更新而变化。在涉及生产环境的关键决策（安全加固、并发方案、版本升级、容量规划等）时，请务必以官方文档与社区的最新结论为准。因使用本书内容所导致的任何直接或间接后果，编撰方不承担责任。

——编撰团队，2026 年 8 月