KaiwuDB Lite 3.1.0 hotfix 1

用户手册

上海沄熹科技有限公司

2025 年 10 月

声明

目录

1. 产品简介

KaiwuDB Lite 是一款专为边端物联网（IoT）场景设计的轻量级时序数据库，具有轻量高效、简洁易用、稳定可靠的特点。作为 KaiwuDB 的轻量化版本，KaiwuDB Lite 提供核心的时序数据存储和查询功能，适用于资源有限的边缘设备、嵌入式系统、云边端协同等应用场景。

2. 核心功能

时序数据处理能力：内置优化的存储引擎和数据压缩技术，可高效存储和管理大规模时间序列数据。支持无模式写入，提供 Appender 功能实现高效批量写入。支持时间范围查询、聚合计算、插值查询、状态窗口分析等丰富的时序分析能力，提供标准 SQL 接口助力用户快速获取实时洞察。

轻量高效设计：采用简洁高效的架构设计，可在内存、CPU、存储资源有限的环境下稳定运行，特别适用于嵌入式设备和边缘计算场景，在性能与资源消耗之间实现良好平衡。

完善的 SQL 支持：完全兼容标准 SQL，支持 `SELECT`、`JOIN`、`GROUP BY` 等常用查询语句；支持全面的 DDL 和 DML 操作，包括 SCHEMA、TABLE、COLUMN、INDEX 等对象管理；支持 `INSERT`、`UPDATE`、`DELETE` 等数据操作语句；支持多种查询方式：简单查询、嵌套查询、关联查询、联合查询、标签查询等。

丰富的数据类型与函数：支持时间戳、数值、布尔、字符串等多种数据类型，提供灵活的类型转换机制；内置丰富的函数库，包括条件函数、聚合函数、日期时间函数、数学统计函数、字符串函数、状态窗口函数等；提供多种操作符，增强数据分析与处理能力。

多样化连接方式：支持 HTTP、PostgreSQL JDBC、PostgreSQL ODBC、Mybatis-Plus 及 EMQX 等多种连接方式，兼容主流技术栈，方便用户根据实际需求选择适配方案。

数据管理与运维能力：支持数据生命周期管理、压缩机制及许可证管理，简化运维工作；可配置线程数、内存使用和日志策略，保障不同负载下的稳定运行；提供灵活的数据导入导出机制，便于数据迁移与备份；支持用户的创建、修改、删除操作，实现精细化的访问权限控制。

3. 基础概念

概念	描述
Metric（指标）	时序数据中需要监控的具体数值，如 CPU 使用率、内存占用量、温度值等。 Metric 具有显著的时间变化特性，数值随时间推移而变化，构成时序数据的核心内容。通过记录采集对象的实时测量值，形成连续的时间序列数据，为系统监控和分析提供基础数据支撑。
Tag（标签）	用于标识和分类时序数据的静态属性信息，如服务器名称、地理位置、设备型号等。 Tag 的值相对稳定，描述数据采集对象的固有特征，能够为数据提供分组、过滤和查询定位的依据，使用户能够快速筛选特定条件下的时序数据，实现精确的数据检索和分析。
单表存储	在一张时序表中同时存储 Metrics 和 Tags 信息的存储方式。 单表存储将数据集中在同一表中，查询时无需关联操作，使用方式简单直观，降低了操作复杂度。但是，建表后不支持加减列操作，存储效率相对较低。这种存储方式适用于表结构相对固定、数据模式稳定的应用场景。
双表存储	将 Metrics 和 Tags 分别存储在两张独立表中的存储方式。 双表存储实现数据分离存储，查询时需要显式关联操作，操作相对复杂。但支持动态加减列操作，存储效率高且压缩率优异。这种存储方式适用于数据量大、表结构需要灵活变更的场景。
时序表	KaiwuDB Lite 中专门用于存储时序数据的表结构。

	时序表的第一列必须为 TIMESTAMP 类型，用于记录数据采集时间。表结构包含通过 TAGS 关键字定义的标签列和存储 Metric 值的数据字段。时序表设计充分考虑了时序数据的特点，能够高效存储和查询按时间排列的数据序列，为时序数据管理提供专业化的存储方案。
Metrics 表	双表存储模式中专门存储时序指标数据的表。 Metrics 表使用 TS 关键字创建，第一列为 TIMESTAMP 类型。该表具有高压缩率和查询效率的优势，支持动态加减列操作，为系统提供了灵活的扩展能力。Metrics 表专业化处理大量时序数据的写入和查询操作，确保系统在高并发场景下的稳定性能。
Tags 表	双表存储模式中专门存储标签信息的表。 Tags 表存储相对静态的标签属性信息，通过特定的关联字段与 Metrics 表建立关系。由于标签信息更新频率较低，Tags 表的结构设计更加灵活可变。Tags 表为数据提供分组、过滤和检索依据，实现标签与指标数据的有效分离，显著提高存储效率和查询性能。

4. 安装部署

4.1 概述

为满足不同应用场景的需求，KaiwuDB Lite 提供两种灵活的部署方式：

- 安装包部署： 采用传统的客户端-服务器架构，以独立服务进程形式运行，支持多客户端并发访问，适合企业级应用和多用户系统。
- 嵌入式部署： 将数据库引擎直接集成到应用程序中，无需启动独立服务，具有更低的资源消耗和延迟，特别适合单机应用和边缘计算场景。

4.2 安装包部署

4.2.1 部署准备

4.2.1.1 操作系统

KaiwuDB Lite 支持在以下操作系统上进行安装部署：

操作系统	版本	架构
Ubuntu	18.04 及以上	• x86_64 • arm64
KylinOS	V10	• 鲲鹏 • 海光
Windows	Windows 11	x86_64

4.2.1.2 端口

KaiwuDB Lite 默认使用 36257 端口作为客户端连接端口。如需使用其他端口，可通过修改 `kaiwudb-lite.conf` 文件中的端口配置参数进行调整。

4.2.1.3 安装包

获取系统对应的 KaiwuDB Lite 安装包，将安装包拷贝到目标机器上，然后解压安装包：

```bash
tar -zxvf <package_name>
```

解压后生成的目录包含以下文件和文件夹：

文件	说明
`kaiwudb_lite`	数据库可执行文件
`libkaiwudb_lite.so`	Linux 链接库，用于 C++ 嵌入式集成部署

kaiwudb_lite.dll kaiwudb_lite.lib	Windows 链接库，用于 C++ 嵌入式集成部署
kaiwudb-lite-default.conf	数据库默认配置文件，用户可根据需要选择： • 直接使用默认参数配置，具体默认值，参见配置文件说明章节。 • 创建 kaiwudb-lite.conf 配置文件，自定义参数值。

4.2.1.4 许可证

联系 KaiwuDB Lite 技术支持人员，获取 .lic 格式的许可证文件，并将其放置到相应目录：

- Ubuntu 和 KylinOS：/usr/local/etc
- Windows: C:\Users\Public\

4.2.2 安装部署

4.2.2.1 前提条件

- 操作系统、软件依赖和端口满足安装部署要求。
- 已获得 KaiwuDB Lite 安装包和许可证。

4.2.2.2 安装方式

4.2.2.2.1 使用默认配置

使用默认方式启动数据库时，系统会自动使用当前目录下的 data 目录作为数据目录，data/log 目录作为日志目录。如果这些目录不存在，系统会自动创建。

步骤

1. 进入可执行文件所在目录。
2. 启动数据库。

- Linux 系统（Ubuntu/KylinOS）
  - 前台启动：

```
Bash
./kaiwudb_lite
```

  - 后台启动：

```
Bash
nohup ./kaiwudb_lite &
```

- Windows 系统
  - 前台启动：双击运行 kaiwudb_lite.exe。
  - 后台启动：

```
Shell
# PowerShell
Start-Process -WindowStyle hidden -FilePath ".\kaiwudb_lite.exe"
```

4.2.2.2.2 使用自定义配置

使用自定义配置方式启动数据库时，系统将按以下顺序查找配置文件 kaiwudb-lite.conf：data 目录 → 程序目录 → 当前目录，找到后立即使用。若未找到配置文件，则使用 kaiwudb-lite-default.conf 中的默认配置启动。

步骤

1. 进入安装包目录，将 kaiwudb-lite-default.conf 配置文件复制到指定数据目录下，并重命名为 kaiwudb-lite.conf。

```
Shell
# 进入安装包目录
cd /path/to/installation/package
```

```
# 复制并重命名配置文件到数据目录
cp kaiwudb-lite-default.conf /path/to/data/directory/kaiwudb-lite.conf
```

2. 编辑 kaiwudb-lite.conf 文件，设置数据库 IP 地址、客户端连接端口、数据目录、日志目录等信息。

3. 启动数据库。

   - Linux

     ```
     Bash
     ./kaiwudb_lite [-D | --data | -l | --log | -p | --port | -v | --version ]
     ```

   - Windows

     ```
     Bash
     .\kaiwudb_lite.exe [-D | --data | -l | --log | -p | --port | -v | --version ]
     ```

   参数说明：

   - -D，--data：可选参数，指定数据文件和配置文件路径，例如 --data /var/data，表示数据存储目录为/var/data。
   - -l，--log：可选参数，指定日志文件存储目录，例如 -l /var/data/log，表示在当前目录下创建/var/data/log 目录，用于存储日志文件。
   - -p，--port：可选参数，指定数据库客户端连接端口，设置值与配置文件中的设置不一致时，系统将优先使用命令行参数。
   - -v，--version：可选参数，显示当前数据库版本。

   示例：

   - 启动数据库

     ```
     Bash
     # Linux
     # 短参数格式（无空格）
     ```

```
./kaiwudb_lite -Ddata -llog -p36257

# 短参数格式（空格分隔）
./kaiwudb_lite -D data -l log -p 36257

# 长参数格式（等号连接）
./kaiwudb_lite --data=data --log=log --port=36257

# 长参数格式（空格分隔）
./kaiwudb_lite --data data --log log --port 36257
```

- 查看数据库版本

```
Bash
# Windows
.\kaiwudb_lite.exe -v

# 系统输出
KaiwuDB-lite v3.0.0
```

4.2.2.3 配置文件说明

4.2.2.3.1 文件示例

```
bash
代码块
# KaiwuDB-Lite Server Configuration

[server]
ip          = 127.0.0.1   # 监听地址
port        = 36257     # 默认端口，有效范围：[1024, 65535]
```

```
data_dir      = data      # 数据存储位置
log_dir       = data/log   # 日志文件位置

# 性能配置（当前已注释）
# threads = 16          # 请求处理最大线程数
# memory_limit = 32GB     # 总内存分配限制

# 日志配置
enable_logging  = false      # 启用日志：false 或 true，默认为 false
logging_mode    = LEVEL_ONLY  # 设置日志过滤模式
# disabled_log_types = ''    # 输出时排除指定的日志类型
# enabled_log_types = ''     # 仅输出指定的日志类型
logging_level   = WARN       # 可用级别：TRACE, DEBUG, INFO, WARN, ERROR, FATAL
logging_storage = file       # 日志输出选项：memory, stdout, file
auto_prune = false          # 日志自动删除开关，打开后仅保留最多 20 MB 日志


# UDS 配置
# enable_uds        = false   # 是否启用 UDS 连接方式
# uds_file_path     = /tmp/kaiwudb_lite    # UDS 文件的存储路径

# 数据库配置
# ignore_db_noexists = true   # 需要连接的目标数据库不存在时，是否仍要连接数据
库


# 生命周期配置（当前已注释）
# [retention]
# 为特定模式中的所有表配置生命周期
```

```
# schemas_reg=[main.*, schema7.*, schema3.*, schema4.*]=[5 seconds]
# schemas_reg=[schema5.*, schema6.*]=[10 seconds]

# 为特定表配置生命周期
# schema.tables=[schema6.table4, schema6.table5, schema6.table6]=[30 seconds]
# schema.tables=[schema9.table1]=[1 years]
# schema.tables=[schema9.table2]=[1 months]

# 配置任务调度（cron 格式）
# cronTask=[*/5 * * * * *]
```

4.2.2.3.2 参数说明

配置文件中出现重复参数时，后面的值将覆盖前面的值。

[server]： 数据库服务器相关配置

参数	默认值	有效范围/可选值	说明
ip	127.0.0.1（本地回环地址）	有效的 IP 地址	数据库所在服务器的 IP 地址，用于建立连接，如需从其他机器连接访问，必须在启动前修改为可被外部访问的服务器 IP
port	36257	[1024, 65535]	客户端连接端口。可通过命令行参数进行覆盖配置，命令行优先级更高。
data_dir	data（）当前目录下的 data 目录）	有效的目录路径	数据存储路径。可通过命令行参数进行覆盖配置，命令行优先级更高。
log_dir	data/log	有效的目录路径	日志文件存储目录。可通过命令行

			参数进行覆盖配置，命令行优先级更高。
threads	无限制	[1, 2147483647]	数据库可使用的最大线程数，支持运行时通过 SET threads = <num>; SQL 命令在运行时动态修改。修改后立即对所有活跃会话生效，并自动更新配置文件以在重启后保留设置。 配置建议： • 4 核及以上系统：设置为 CPU 核数，充分利用多核性能 • 4 核以下系统：设置为 CPU 核数的 2 倍，以提升并发处理能力
memory_limit	系统总内存的 80%	32 MB 至 9007199254740991 字节	数据库可分配的最大内存容量。支持单位包括 KB、MB、GB、TB（十进制）和 KiB、MiB、GiB、TiB（二进制）。 支持运行时通过 SET memory_limit = '<num> <unit>'; SQL 命令在运行时动态修改。修改后立即对所有活跃会话生效，并自动更新配置文件以在重启后保留设置。 配置建议： - 内存 ≥ 8 GB：设置为总内存的 50% - 内存 < 8 GB：设置为总内存的 25%，以确保系统稳定性

enable_logging	false	• true（启用） • false（关闭）	是否启用日志功能。 支持运行时通过 SET enable_logging = [true | false]; SQL 命令在运行时动态修改。修改后立即对所有活跃会话生效，并自动更新配置文件以在重启后保留设置。
logging_mode	LEVEL_ONLY	• LEVEL_ONLY：仅按日志级别过滤 • DISABLE_SELECTED：启用黑名单 • ENABLE_SELECTED：启用白名单	日志过滤模式，用于控制是否启用日志白名单和黑名单。 支持运行时通过 SET logging_mode = [LEVEL_ONLY | DISABLE_SELECTED | ENABLE_SELECTED]; SQL 命令在运行时动态修改。修改后立即对所有活跃会话生效，并自动更新配置文件以在重启后保留设置。
disabled_log_types	-	• kaiwudb.FileSystem.LocalFileSystem.OpenFile • kaiwudb.ClientContext.BeginQuery • kaiwudb.Extension.ExtensionLoaded	日志输出黑名单。在 logging_mode 为 DISABLE_SELECTED 模式下生效，用于排除指定类型的日志。 支持指定多个日志类型，用逗号分隔，例如 disabled_log_types = 'kaiwudb.FileSystem.LocalFileSystem.OpenFile,kaiwudb.ClientContext.BeginQuery' 支持运行时通过 SET disabled_log_types SQL 命令在运行时动态修改。修改后立即对所有

			活跃会话生效，并自动更新配置文件以在重启后保留设置。
enabled_log_types	-	• kaiwudb.FileSystem.LocalFileSystem.OpenFile • kaiwudb.ClientContext.BeginQuery • kaiwudb.Extension.ExtensionLoaded	日志输出白名单。在 logging_mode 为 ENABLE_SELECTED 模式下生效，仅输出指定的日志类型。 支持指定多个日志类型，用逗号分隔，例如 enabled_log_types = 'kaiwudb.FileSystem.LocalFileSystem.OpenFile,kaiwudb.ClientContext.BeginQuery' 支持运行时通过 SET enabled_log_types SQL 命令在运行时动态修改。修改后立即对所有活跃会话生效，并自动更新配置文件以在重启后保留设置。
logging_level	WARN	• TRACE • DEBUG • INFO • WARN • ERROR • FATAL	日志记录的级别，表示记录指定级别及以上的日志信息。 支持运行时通过 SET logging_level = [TRACE | DEBUG | INFO | WARN | ERROR | FATAL]; SQL 命令在运行时动态修改。修改后立即对所有活跃会话生效，并自动更新配置文件以在重启后保留设置。
logging_storage	file	• file（存储在日志文件） • memory（存储在内存）	日志存储位置。 支持运行时通过 SET logging_storage = [file | memory | stdout]; SQL 命令在运行时动态修

		• stdout（输出到标准输出）	改。修改后立即对所有活跃会话生效，并自动更新配置文件以在重启后保留设置。
auto_prune	false	• false • true	日志自动删除开关，打开后仅保留最多 20 MB 日志。
enable_uds	false	• true（启用） • false（关闭）	是否启用 Unix Domain Socket (UDS) 连接方式，用于同一服务器上进程间的高性能通信。
uds_file_path	-	有效的目录路径	UDS 文件的存储路径，启用 UDS 时必须指定此路径。
ignore_db_noexists	TRUE	• true（启用） • false（关闭）	目标数据库不存在时的处理策略。设置为 true 时，会自动连接到 metadata 数据库；设置为 false 时，连接会失败并报错退出。

[retention]：用于配置模式和表的生命周期以及调度周期，默认关闭。具体配置说明请参见生命周期管理。

4.3 嵌入式部署（无服务部署）

KaiwuDB Lite 目前支持以下编程语言的嵌入式集成：

- C++：提供原生 C++ API，性能最优
- Python：提供 Python 绑定，支持 DataFrame 集成

4.3.1 C++ 嵌入式集成

4.3.1.1 前提条件

- 已获取操作系统对应的安装包。
- 联系 KaiwuDB Lite 技术支持人员获取头文件包。
- 已安装 C++编译器（推荐 GCC 或 Visual Studio）。
- 已安装 CMake（版本 3.10 或以上）。

4.3.1.2 步骤

1. 配置环境。
   a. 创建一个新的项目文件夹，例如 `my_kaiwudb_lite_project`。
   b. 在项目根目录创建 `CMakeLists.txt` 文件：

```
CMake
cmake_minimum_required(VERSION 3.10)
project(my_kaiwudb_app)

# 设置 C++ 标准
set(CMAKE_CXX_STANDARD 11)

# 修改为 KaiwuDB Lite 实际安装路径
# Linux 示例
set(KAIWULITE_ROOT "/home/lite-3.0/kaiwudb_lite-v3.0.0-ubuntu-20.04-amd64")

# 添加头文件搜索路径
include_directories(${KAIWULITE_ROOT}/include)

# 注意：库文件直接在根目录下，不在 lib 子目录
link_directories(${KAIWULITE_ROOT})
```

```
# 创建可执行文件
add_executable(my_app main.cpp)

# 链接 KaiwuDB Lite 库
target_link_libraries(my_app kaiwudb_lite)

# Windows 部署：构建完成后自动复制 DLL 文件到可执行文件目录
# Windows 环境下需要启用以下配置
#add_custom_command(TARGET my_app POST_BUILD
#   COMMAND ${CMAKE_COMMAND} -E copy_if_different
#  "${KAIWULITE_ROOT}/kaiwudb_lite.dll"
#   $<TARGET_FILE_DIR:my_app>)
```

2. 在项目目录创建 main.cpp 文件。

   文件示例：

```
C++
#include <iostream>
#include "kaiwudb.hpp"

using namespace kaiwudb;

int main() {
  try {
    // 创建数据库连接（使用相对路径，在当前目录创建数据库文件）
    KaiwuDB db("./test.db");
    Connection con(db);

    // 创建表
    con.Query("CREATE TABLE t1(ts timestamp, metric int);");
```

```
    // 批量插入数据
    Appender appender(con, "t1");
    for (size_t i = 0; i < 1000; ++i) {
      appender.AppendRow(kaiwudb::timestamp_t(i), i);
    }

    // 查询数据
    auto result_count = con.Query("SELECT COUNT(*) FROM t1");
    if (result_count->HasError()) {
      std::cerr << result_count->GetError() << std::endl;
    } else {
      std::cout << result_count->ToString() << std::endl;
    }

  } catch (const std::exception& e) {
    std::cerr << "Exception: " << e.what() << std::endl;
    return -1;
  }

  return 0;
}
```

3. 编译和运行。

   a. 创建构建目录：

   ```Bash
   mkdir build && cd build
   ```

   b. 生成构建文件：

```
Python
cmake ..
```

c. 编译项目：

```
Bash
# Linux
make -j4

# Windows
cmake --build . --config Release
```

4. 运行程序：

```
Bash
# Linux
./my_app

# Windows
Release\my_app.exe
```

4.3.2 Python 嵌入式集成

> 说明：Python 嵌入式集成目前只支持 Ubuntu 24.04 系统。

4.3.2.1 前提条件

- 已联系 KaiwuDB Lite 技术支持人员获取 Python 包：`kaiwudb-3.0.0-cp312-cp312-linux_x86_64.whl`。
- 已安装 Python 3.12 及以上版本。
- （可选）根据需要安装数据处理库：pandas、Polars、PyArrow。

4.3.2.2 步骤

1. 创建虚拟环境。建议使用虚拟环境来隔离项目依赖。

```bash
# 创建虚拟环境
python3 -m venv .venv

# 激活虚拟环境
source .venv/bin/activate
```

2. 安装 KaiwuDB Lite Python 包：

```python
pip install kaiwudb-3.0.0-cp312-cp312-linux_x86_64.whl
```

3. 编写示例代码：

```python
import kaiwudb
import pandas, polars, pyarrow

# 连接数据库
with kaiwudb.connect("test.db") as con:
    # 删除已存在的表
    con.sql("DROP TABLE IF EXISTS test")
    # 创建新表
    con.sql("CREATE TABLE test(a INT)")

    # 使用 pandas DataFrame
    pandas_df = pandas.DataFrame({"a": [10]})
    con.sql("SELECT * FROM pandas_df").show()
```

```
# ┌─────────┐
# │  a  │
# │ int64 │
# ├─────────┤
# │  10 │
# └─────────┘

# 使用 Polars DataFrame
polars_df = polars.DataFrame({"a": [11]})
con.sql("SELECT * FROM polars_df").show()
# ┌─────────┐
# │  a  │
# │ int64 │
# ├─────────┤
# │  11 │
# └─────────┘

# 使用 PyArrow Table
arrow_table = pyarrow.Table.from_pydict({"a": [12]})
con.sql("SELECT * FROM arrow_table").show()
# ┌─────────┐
# │  a  │
# │ int64 │
# ├─────────┤
# │  12 │
# └─────────┘

# 将不同数据结构插入表中
con.sql("INSERT INTO test SELECT * FROM pandas_df")
```

```
con.sql("INSERT INTO test SELECT * FROM polars_df")
con.sql("INSERT INTO test SELECT * FROM arrow_table")



# 查询表中所有数据
con.sql("SELECT * FROM test").show()
# ┌─────────────┐
# │  a  │
# │ int32 │
# ├─────────────┤
# │  10 │
# │  11 │
# │  12 │
# └─────────────┘

# 使用游标方式查询（可选）
cur = con.cursor()
cur.execute("SELECT * FROM test")
print(cur.fetchone())  # 仅输出第一个结果
print(cur.fetchall())  # 输出剩下的结果
```

4. 运行脚本：

```
Python
python3 example.py
```

5. 启停数据库

5.1 启动数据库

5.1.1 Linux 环境

1. 导航到 kaiwudb_lite 可执行文件所在目录。

```
Shell
cd /path/to/kaiwudb_lite
```

2. 启动数据库。

```
Shell
./kaiwudb_lite
```

5.1.2 Windows 环境

进入可执行文件所在目录，双击运行

5.2 停止数据库

5.2.1 Linux 环境

5.2.1.1 前台运行时停止

按下 Ctrl+C 组合键停止数据库。

5.2.1.2 后台运行时停止

1. 查找运行中的 KaiwuDB Lite 进程 ID：

```
Shell
ps aux | grep kaiwudb_lite
```

2. 终止进程：

```
Shell
kill <进程 ID>
```

示例：

```
Bash
# 查看进程
ps -ef | grep kaiwudb_lite
# 输出：
mgj 127008 1982 0 Mar03 pts/1 00:00:28 ./kaiwudb_lite
# 终止进程
kill 127008
```

5.2.2 Windows 环境

1. 在 PowerShell 中查找运行中的 KaiwuDB Lite 进程 ID：

```
Shell
tasklist | findstr kaiwudb_lite
```

2. 终止进程：

```
Shell
taskkill /PID <PID> /F
```

示例：

```
Bash
# 查看进程
tasklist | findstr kaiwudb_lite
# 输出：
kaiwudb_lite.exe          15136 Console                1    25,504 K


# 终止进程
taskkill /PID 15136 /F
# 输出：
```

```
成功: 已终止 PID 为 15136 的进程。
```

5.3 配置服务自启动

KaiwuDB Lite 支持在 Linux、麒麟和 Windows 环境下配置数据库服务的自启动功能，确保系统重启后服务能够自动运行。

5.3.1 Linux 和麒麟系统

Linux 和麒麟环境下推荐使用 systemd 服务管理器实现服务自启动，支持以下两种方式：

- 用户级服务：在用户登录时自动启动，无需 root 权限。
- 系统级服务：在系统启动时自动启动，需要 root 权限。

5.3.1.1 准备配置文件

步骤

1. 创建 `kaiwudb-lite.args` 参数文件，内容如下：

```
Bash
# Set the command-line arguments to pass to the server.
ARGS='-p 36257 -D data'
```

2. 创建 `kaiwudb-lite.service` 服务定义文件，内容如下：

```
TOML
[Unit]
Description=kaiwudb-lite User Database Server
After=network.target

[Service]
Type=simple
```

```
EnvironmentFile=/path-to-kaiwudb-lite-args/kaiwudb-lite.args
ExecStart=/path-to-kaiwudb-lite/kaiwudb_lite $ARGS
WorkingDirectory=/working_path
Restart=always
RestartSec=5

[Install]
WantedBy=default.target
```

参数说明：

- EnvironmentFile：参数文件的完整路径
- ExecStart：KaiwuDB Lite 可执行文件的完整路径
- WorkingDirectory：KaiwuDB Lite 的工作目录，默认在该目录下创建 data/ 和 data/log/ 目录。

> 说明
>
> KaiwuDB Lite 支持两种参数配置方式，优先级为：命令行参数 > 配置文件参数，其中：
>
> - 命令行参数通过 EnvironmentFile 参数文件指定
> - 配置文件需放置在 WorkingDirectory 指定的目录下
>
> 用户可同时使用两种方式，命令行参数会覆盖配置文件中的同名参数

5.3.1.2 配置服务

根据需要配置用户级服务或系统级服务。

5.3.1.2.1 配置用户级服务

前提条件

- 已安装 KaiwuDB Lite 服务。

- 已启用用户服务持久化。

```
Bash
# 启用用户服务持久化
loginctl enable-linger $USER

# 检查当前用户的 linger 状态是否为 yes
loginctl show-user $USER | grep Linger
```

- 已检查用户级 systemd 服务的运行环境

```
Bash
# 检查用户服务管理器状态是否为 active
systemctl status user@$(id -u)

# 验证用户级 systemctl 环境是否可用，即状态是否为 running
systemctl --user status

# 如果报错 Failed to connect to bus: Connection refused 或 Failed to connect to bus: No medium found，执行以下命令：
# export XDG_RUNTIME_DIR=/run/user/$(id -u)  export DBUS_SESSION_BUS_ADDRESS=unix:path=${XDG_RUNTIME_DIR}/bus
# 或将命令写入 .bashrc，然后执行 source ~/.bashrc
```

步骤

1. 创建用户服务目录。

```
Bash
# 注意：此处 user 为固定名字，不要替换成用户名
mkdir -p ~/.config/systemd/user
```

2. 安装服务文件。

```
Bash
mv kaiwudb-lite.service ~/.config/systemd/user/
```

3. 启用并启动服务。

```
Bash
# 重新加载 systemd 配置
systemctl --user daemon-reload

# 立即启动服务
systemctl --user start kaiwudb-lite
```

4. （可选）检查服务运行状态。

```
Bash
systemctl --user status kaiwudb-lite
```

5.3.1.2.2 配置系统级服务

前提条件

- 已安装 KaiwuDB Lite 服务。
- 用户拥有 root 权限。

步骤

1. 安装服务文件。

```
Bash
sudo cp ~/.config/systemd/user/kaiwudb-lite.service /etc/systemd/system/
```

2. 启用并启动服务。

```
Bash
# 启用开机自启
sudo systemctl enable kaiwudb-lite
```

```
# 立即启动服务
sudo systemctl start kaiwudb-lite
```

3. 检查服务状态。

```
Bash
# 检查服务状态是否为 active
sudo systemctl status kaiwudb-lite

# 检查进程
ps -ef | grep kaiwudb-lite
```

5.3.2 Windows 环境

Windows 环境下推荐使用 NSSM（Non-Sucking Service Manager）第三方工具注册系统服务，实现服务的自启动。

5.3.2.1 配置服务自启动

前提条件

- 已安装 KaiwuDB Lite。
- 用户拥有管理员权限。

步骤

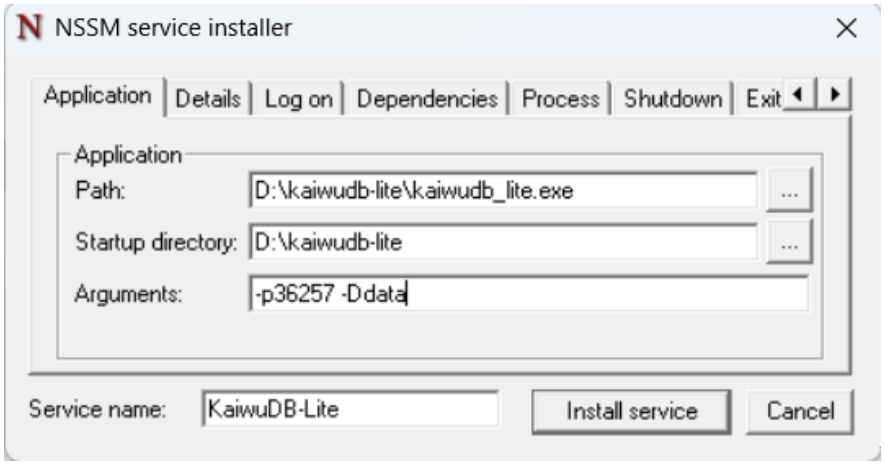
1. 访问 NSSM 官网下载安装包，将安装包解压至指定目录，例如 `D:\apps\nssm-2.24`。
2. 以管理员身份打开命令提示符或 PowerShell，然后切换到 NSSM 所在目录。

3. 安装 KaiwuDB Lite：

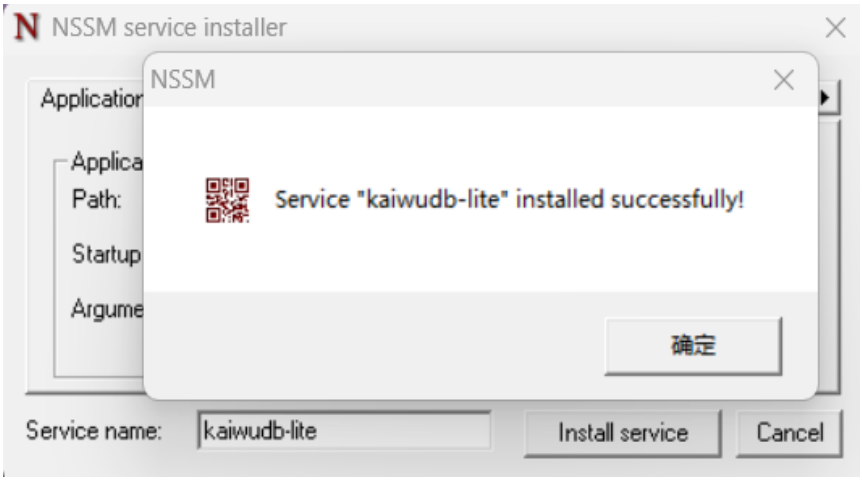
```
Bash

.\nssm.exe install kaiwudb-lite
```

4. 在弹出的配置界面，填写 KaiwuDB Lite 可执行文件的完整路径，根据需要填写启动参数，然后点击 Install service。



配置界面



配置成功

5. （可选）验证自启动配置是否成功：

   a. 按下键盘组合键 Win + R，在运行对话框中输入 services.msc，打开服务管理器。

   b. 在服务管理器窗口，查找 kaiwudb-lite 服务，确认启动类型为自动。

5.3.2.2 服务管理

使用 NSSM 完成系统服务注册后，即可通过 NSSM 进行 KaiwuDB Lite 服务管理。更多 NSSM 支持的命令，参见 NSSM 官网说明。

- 查看服务状态

```
Plain Text
.\nssm.exe status kaiwudb-lite
```

- 启动服务

```
Plain Text
# 使用 NSSM
.\nssm.exe start kaiwudb-lite

# 使用系统命令
net start kaiwudb-lite
```

- 停止服务

```
Plain Text
.\nssm.exe stop kaiwudb-lite
```

- 删除服务

```
Plain Text
.\nssm.exe remove kaiwudb-lite
```

6. 连接数据库

6.1 客户端连接

本节介绍如何使用 psql CLI 工具连接 KaiwuDB Lite 数据库。

6.1.1 前提条件

- 安装并启动 KaiwuDB Lite 数据库。有关详细信息，参见安装部署。
- 安装 psql CLI 工具：
  - Ubuntu 系统
    - 在线安装：

```
Shell
apt-get install postgresql-client
```

- 获取安装包后，离线安装：

```
Shell
dpkg -i postgresql-client-14_14.18-0ubuntu0.22.04.1_amd64.deb
```

- Windows 系统
  i. 访问 PostgreSQL 官网，下载适合的版本。
  ii. 下载后运行安装程序，在组件选择时可以只选择 "pgAdmin" 和 "Command Line Tools"。
- KylinOS 系统
  - 在线安装：

```
Shell
sudo yum install postgresql.x86_64
```

  - 下载源码编译，具体步骤，参见 PostgreSQL 官网文档。

6.1.2 连接数据库

运行以下命令，连接 KaiwuDB Lite 数据库。

```
Bash
psql -h <host> -p <port> -U <user_name>
```

参数说明：

- host：数据库所在服务器的 IP 地址，本地访问时可使用 localhost。
- port：数据库客户端连接端口，默认使用 36257。
- user_name：连接数据库的用户。默认为 admin。支持创建新的用户并使用新用户进行连接。有关创建用户的详细信息，参见创建用户。

6.2 Java

6.2.1 PostgreSQL JDBC

本节介绍如何使用 PostgreSQL JDBC 驱动程序连接 KaiwuDB Lite 数据库。

6.2.1.1 前提条件

- 安装 openJDK（1.8 及以上版本）。
- 安装 Maven（3.6 及以上版本）。
- 安装并启动 KaiwuDB Lite 数据库。有关详细信息，参见安装部署。
- 创建具有表级别及以上操作权限的用户。有关详细信息，参见创建用户。
- 获取 PostgreSQL JDBC 驱动程序。

> 说明
>
> 本节示例使用 postgresql-42.7.5.jar PostgreSQL JDBC 驱动程序。用户可按需选择 PostgreSQL JDBC 驱动程序的目标版本。

6.2.1.2 引入依赖

在项目的 pom.xml 中添加依赖，将 PostgreSQL JDBC 驱动程序引入到 Java 应用程序中。

```xml
<dependency>
  <groupId>org.postgresql</groupId>
  <artifactId>postgresql</artifactId>
  <version>42.7.5</version>
</dependency>
```

如上述依赖无法正常加载使用，运行以下命令，将 PostgreSQL JDBC 驱动程序安装到本地 Maven 仓库中。

```Shell
```

```
mvn install:install-file -Dfile=./postgresql-42.7.5.jar -DgroupId=com.kaiwudb -DartifactId=kwjdbc -Dversion=42.7.5 -Dpackaging=jar
```

6.2.1.3 配置连接

1. 为 PostgreSQL JDBC 驱动程序设置 CLASSPATH 环境变量。

```
Shell
export CLASSPATH=<path_to_your_java_file.jar>:<path_to_your_postgresql-42.7.5.jar>:. java <app_name>
```

2. 设置 KaiwuDB Lite 数据库的连接参数。

示例：

```
java
public class Test{
public static void main (){
    try{
      // 加载驱动
      Class.forName("org.postgresql.Driver");
    }catch(Exception e){
      System.out.println(e.toString());
    }
    // 数据库连接参数
    String url = "jdbc:postgresql://<host>:<port>/<database>";
    String username = "<username>";
    String password = "<password>";
      try {
        // 建立连接
        conn = DriverManager.getConnection(url, username, password);
        // 创建表
```

```
            stmt.execute("create table test(a int);");
        }catch (Exception e) {
            System.out.println("create table fail");
            System.out.println(e.toString());
        }
        return 0;
    }
}
```

参数说明：

- url：数据库所在服务器的 IP 地址，采用 jdbc:postgresql://host:port/database 格式。本地访问时可使用 localhost。
    - host：数据库所在服务器的 IP 地址，本地访问时可使用 localhost。
    - port：数据库客户端连接端口，默认使用 36257。
    - database：数据库的名称，默认使用 main。
- username：连接数据库的用户。默认为 admin。
- password：连接数据库用户的密码。如果使用 admin 用户，密码可以为空。

6.2.2 Mybatis-Plus

本节介绍演示如何在 Maven 管理的 Spring Boot 项目中，整合 KaiwuDB Lite 数据库，通过 MyBatis-Plus 实现数据访问和管理。

6.2.2.1 前提条件

- 安装 openJDK（1.8 及以上版本）。
- 安装 Maven（3.6 及以上版本）。
- 安装并启动 KaiwuDB Lite 数据库。有关详细信息，参见安装部署。

- 创建具有表级别及以上操作权限的用户。有关详细信息，参见创建用户。
- 获取 PostgreSQL JDBC 驱动程序。
- 获取 mybatis-plus-boot-starter。

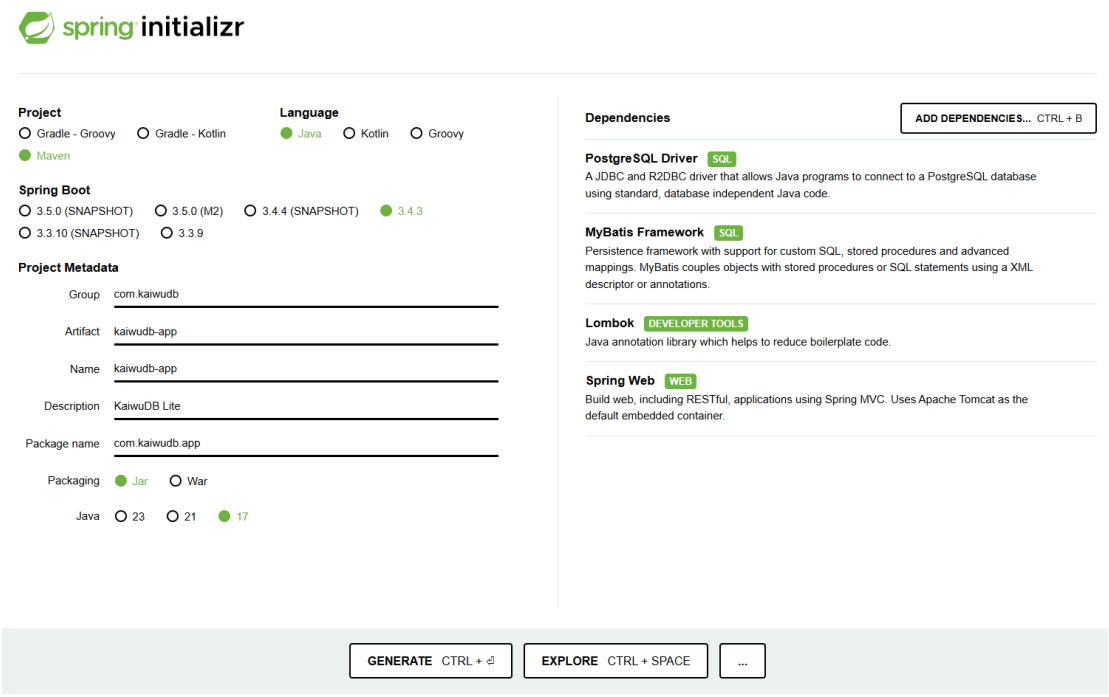
> 说明
>
> 本节示例使用以下软件版本。用户可按需选择目标软件版本。
>
> - Spring Boot: 3.4.3
> - PostgreSQL JDBC 驱动程序：42.7.5
> - MyBatis-Plus: 3.0.4

6.2.2.2 环境搭建

6.2.2.2.1 初始化应用项目

本章节使用 Spring Initializr 生成带有指定依赖的项目包。

1. 在 Spring Initializr 页面右侧部分，选择项目使用的编程语言及其版本、包管理工具、Spring Boot 版本，填写项目的基本信息。
2. 在 Spring Initializr 页面右侧部分，单击 ADD DEPENDENCES，选择并添加项目使用的依赖。
3. 单击 GENERATE，生成项目使用的压缩包。



6.2.2.2.2 引入依赖

在项目的 pom.xml 中添加依赖，将 PostgreSQL JDBC 驱动程序引入到 Java 应用程序中。

```xml
<dependency>
  <groupId>org.postgresql</groupId>
  <artifactId>postgresql</artifactId>
  <version>42.7.5</version>
</dependency>
```

如上述依赖无法正常加载使用，运行以下命令，将 PostgreSQL JDBC 驱动程序安装到本地 Maven 仓库中。

```Shell
mvn install:install-file -Dfile=./postgresql-42.7.5.jar  -DgroupId=com.kaiwudb -DartifactId=kwjdbc -Dversion=42.7.5 -Dpackaging=jar
```

6.2.2.3 配置连接

1. 在项目的 application.yml 文件中，设置 KaiwuDB Lite 数据库的连接参数。

```yaml
spring:
 datasource:
  url: jdbc:postgresql://<host>:<port>/<database> # 数据库连接 URL
  username: <user_name>                  # 数据库用户名
  password: <password>                   # 数据库用户密码
  driver-class-name: org.postgresql.Driver    # 驱动类名
```

参数说明：

- url：数据库所在服务器的 IP 地址，采用 jdbc:postgresql://host:port/database 格式。本地访问时可使用 localhost。
  - host：数据库所在服务器的 IP 地址，本地访问时可使用 localhost。
  - port：数据库客户端连接端口，默认使用 36257。
  - database：数据库的名称，默认使用 main。
- username：连接数据库的用户。默认为 admin。
- password：连接数据库用户的密码。如果使用 admin 用户，密码可以为空。

2. 启动 MyBatis-Plus 框架服务。

```Shell
mvn spring-boot:run
```

启动成功后，用户可以通过 Web 页面访问 MyBatis-Plus 的框架主页并进行简单的操作。

6.3 C++

6.3.1 PostgreSQL ODBC

6.3.1.1 前提条件

- 安装并启动 KaiwuDB Lite 数据库。有关详细信息，参见安装部署。
- 创建具有表级别及以上操作权限的用户。有关详细信息，参见创建用户。

6.3.1.2 安装 PostgreSQL ODBC 驱动

1. 安装 unixODBC。

```
Bash
apt-get install unixodbc unixodbc-dev odbcinst
```

2. 安装 PostgreSQL ODBC 驱动。

```
Bash
apt install odbc-postgresql
```

6.3.1.3 配置数据源

默认情况下，PostgreSQL ODBC 驱动文件（odbcinst.ini）和 ODBC 数据源文件（odbc.ini）存放于 /usr/local/etc 目录。

- 配置 PostgreSQL ODBC 驱动

  示例：

```
TOML
[PostgreSQL Unicode]
Description=PostgreSQL ODBC driver (Unicode version)
Driver=psqlodbcw.so
Setup=libodbcpsqlS.so
Debug=0
CommLog=1
UsageCount=1
```

- 配置 ODBC 数据源

  示例：

  ```
  TOML
  TOML
  [KaiwuDB Lite]
  Description = KaiwuDB Lite ODBC Data Source
  Driver = PostgreSQL Unicode
  Servername = localhost
  Port = 36257
  Database = main
  User = myusername
  Password = mypassword
  ```

  参数说明：

  - [DSN]：数据源的名称。
  - Description：可选项，数据源的描述信息。
  - Driver：PostgreSQL ODBC 驱动的名称。必须与 odbcinst.ini 文件中的 PostgreSQL ODBC 驱动的名称保持一致。
  - Servername：KaiwuDB Lite 数据库的 IP 地址。本地访问时可使用 localhost。
  - Port：KaiwuDB Lite 数据库的连接端口。
  - Database：KaiwuDB Lite 数据库的名称。默认使用 main。
  - User：登录 KaiwuDB Lite 数据库的用户名。默认为 admin。
  - Password：登录 KaiwuDB Lite 数据库的用户密码。如果使用 admin 用户，密码可以为空。

6.3.1.4 配置连接

以下示例说明如何根据配置的数据源连接 KaiwuDB Lite 数据库并查询数据。

1. 创建示例程序文件。

以下示例创建一个名为 demo.cpp 的示例文件。

```
Bash
vim demo.cpp
```

2. 将以下配置示例添加至 demo.cpp 示例文件。

```
c++
#include <iostream>
#include <sql.h>
#include <sqlext.h>

int main() {
  SQLHENV hEnv;
  SQLHDBC hDbc;
  SQLRETURN ret;

  // 分配环境句柄
  SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &hEnv);

  // 设置 ODBC 版本
  SQLSetEnvAttr(hEnv, SQL_ATTR_ODBC_VERSION, (void*)SQL_OV_ODBC3,
0);

  // 分配连接句柄
  SQLAllocHandle(SQL_HANDLE_DBC, hEnv, &hDbc);

  // 连接到数据库
```

```
//SERVER=localhost;DATABASE=meta.db;USER=root;PASSWORD=123456;OPTI
ON=3;
   ret = SQLDriverConnect(hDbc, NULL, (SQLCHAR*)"DRIVER={PostgreSQL
Unicode};DATABASE=main;SERVER=172.17.0.2;PORT=36257;USER=myuserna
me;PASSWORD=mypassword    ", SQL_NTS, NULL, 0, NULL,
SQL_DRIVER_COMPLETE);
   SQLHSTMT stmt;
   if (ret == SQL_SUCCESS || ret == SQL_SUCCESS_WITH_INFO) {
      // 执行 SQL 语句

      ret = SQLAllocHandle(SQL_HANDLE_STMT, hDbc, &stmt);
      // ret = SQLExecDirect(stmt, (SQLCHAR*)"create table t (a int)", SQL_NTS);
      SQLExecDirect(stmt, (SQLCHAR*)"show", SQL_NTS);

      // 处理结果集
      // ...
      SQLCHAR name[256];
     SQLINTEGER age;
     int col=7;
     while (SQLFetch(stmt) == SQL_SUCCESS) {
        for (int i =0;i<col;i++){
           SQLGetData(stmt, i, SQL_C_CHAR, name, sizeof(name), NULL);
           std::cout << name << ", ";
        }
        std::cout<<std::endl;

      //   SQLGetData(stmt, 2, SQL_C_CHAR, &age, 0, NULL);
```

```
  }
   // 关闭连接
   SQLDisconnect(hDbc);
}else {
UCHAR   sqlstate[10];
UCHAR   errmsg[SQL_MAX_MESSAGE_LENGTH];
SDWORD  nativeerr;
SWORD   actualmsglen;
RETCODE rc = SQL_SUCCESS;
   while ( rc != SQL_NO_DATA_FOUND)
{
   rc = SQLError(hEnv, hDbc, stmt,
              sqlstate, &nativeerr, errmsg,
              SQL_MAX_MESSAGE_LENGTH - 1, &actualmsglen);

    if (rc == SQL_ERROR) {
        printf ("SQLError failed!\n");
        return -1;
    }

    if (rc != SQL_NO_DATA_FOUND) {
         printf ("SQLSTATE = %s\n", sqlstate);
         printf ("NATIVE ERROR = %d\n", nativeerr);
         errmsg[actualmsglen] = '\0';
         printf ("MSG = %s\n\n", errmsg);
     }
 }
}
```

```
    // 释放句柄
    SQLFreeHandle(SQL_HANDLE_DBC, hDbc);
    SQLFreeHandle(SQL_HANDLE_ENV, hEnv);

    return 0;
}
```

3. 使用 g++ 编译器编译 demo.cpp 示例文件。

```
Makefile
pg:
    gcc -g demo.cpp -lstdc++ -lodbc -o demo
```

系统生成一个名为 demo 的二进制文件。

4. 运行示例程序。

```
Bash
./demo
```

6.4 C#

KaiwuDB Lite 支持用户通过 Npgsql 或 ODBC 方式连接 KaiwuDB Lite 数据库，进行数据增删改查操作。

6.4.1 Npgsql

6.4.1.1 前提条件

- 安装并启动 KaiwuDB Lite 数据库。有关详细信息，参见安装部署。
- 安装 .NET SDK，本示例使用版本为 8.0.121。

6.4.1.2 配置连接

1. 创建项目文件夹。

```
Bash
# 创建名为 test_npgsql 的控制台应用程序项目
dotnet new console -o test_npgsql

# 进入项目目录
cd test_npgsql
```

2. 添加项目依赖，该命令会在项目文件夹下生成 test_npgsql.csproj 项目配置文件和 Program.cs 代码文件。

```
Bash
dotnet add package Npgsql        # 示例使用 Npgsql 版本为 9.0.4
```

3. 修改 Program.cs 代码文件，配置数据库连接。

```
C#
using System;
using System.Data;
using Npgsql;
using NpgsqlTypes;

class Program
{
    // 'Server Compatibility Mode=NoTypeLoading' 一定要加
    static string DSN = "Host=localhost;Port=36257;Username=admin;Server
Compatibility Mode=NoTypeLoading";
    static string TABLE_NAME = "test";
```

```
static void Main(string[] args)
{
    try
    {
        using var conn = new NpgsqlConnection(DSN);
        conn.Open();

        Create(conn);
        Insert(conn);
        Query(conn);
        Update(conn, "tag2", 5, 5.5);
        Query(conn);
        Delete(conn, "tag1");
        Query(conn);
        conn.Close();
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
    }
}

public static void Create(NpgsqlConnection conn)
{
    try
    {
        using var cmd = new NpgsqlCommand();
        cmd.Connection = conn;
```

```
        cmd.CommandText = $"DROP TABLE IF EXISTS {TABLE_NAME}";
        cmd.ExecuteNonQuery();
        cmd.CommandText = $"CREATE TABLE {TABLE_NAME} (ts TIMESTAMP
NOT NULL, metric1 INT, metric2 DOUBLE PRECISION) tags (tag_name
VARCHAR)";
        cmd.ExecuteNonQuery();
        Console.WriteLine($"✅ Table {TABLE_NAME} created.\n");
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
    }
  }

  public static void Insert(NpgsqlConnection conn)
  {
    try
    {
      using (var transaction1 = conn.BeginTransaction())
      {
        using var cmd = new NpgsqlCommand();
        cmd.Connection = conn;
        cmd.Transaction = transaction1;
        cmd.CommandText = $"INSERT INTO {TABLE_NAME} (ts, metric1,
metric2, tag_name) VALUES (now(), 1, 1.1, 'tag1'), (now(), 2, 2.2, 'tag2')";
        cmd.ExecuteNonQuery();
        transaction1.Commit();
        Console.WriteLine($"✅ Inserted 2 rows into table {TABLE_NAME}
using non-prepared statement, transaction committed.");
```

```
        }

        using (var transaction2 = conn.BeginTransaction())
        {
            // 推荐使用 prepare 方式写入数据以获得更好的性能

            using var cmd = new NpgsqlCommand();
            cmd.Connection = conn;
            cmd.Transaction = transaction2;

            string prepare_sql = $@"INSERT INTO {TABLE_NAME} VALUES (@p1,
@p2, @p3, @p4), (@p5, @p6, @p7, @p8)";
            cmd.CommandText = prepare_sql;

            // 添加参数
            int columns_count = 4;
            for(int i = 0; i < columns_count * 2; i++) {
                if (i % columns_count == 0)
                    cmd.Parameters.Add(new NpgsqlParameter($"p{i + 1}",
NpgsqlTypes.NpgsqlDbType.Timestamp));
                else if (i % columns_count == 1)
                    cmd.Parameters.Add(new NpgsqlParameter($"p{i + 1}",
NpgsqlTypes.NpgsqlDbType.Integer));
                else if (i % columns_count == 2)
                    cmd.Parameters.Add(new NpgsqlParameter($"p{i + 1}",
NpgsqlTypes.NpgsqlDbType.Double));
                else if (i % columns_count == 3)
                    cmd.Parameters.Add(new NpgsqlParameter($"p{i + 1}",
NpgsqlTypes.NpgsqlDbType.Varchar));
```

```
            }

            // 新版本 Npgsql 可以不执行下面的 prepare 语句:
            // cmd.Prepare();

            // 设置参数值, 可循环调用
            // for (int i = 0; i < ; i++)
            cmd.Parameters[0].Value = DateTime.SpecifyKind(DateTime.UtcNow,
DateTimeKind.Unspecified);
            cmd.Parameters[1].Value = 3;
            cmd.Parameters[2].Value = 3.3;
            cmd.Parameters[3].Value = "tag1";
            cmd.Parameters[4].Value = DateTime.SpecifyKind(DateTime.UtcNow,
DateTimeKind.Unspecified);
            cmd.Parameters[5].Value = 4;
            cmd.Parameters[6].Value = 4.4;
            cmd.Parameters[7].Value = "tag2";
            int inserted_rows = cmd.ExecuteNonQuery();
            // } end for

            transaction2.Commit();

            Console.WriteLine($"✅ Inserted {inserted_rows} rows into table
{TABLE_NAME} using prepared statement, transaction committed.");
        }
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
```

```
    }
  }

  public static void Update(NpgsqlConnection conn, string tag_name, int
metric1, double metric2)
  {
    try
    {
      var sql = $"UPDATE {TABLE_NAME} SET metric1 = {metric1}, metric2 =
{metric2} WHERE tag_name = '{tag_name}'";
      Console.WriteLine(sql);
      using var cmd = new NpgsqlCommand(sql, conn);
      int updated_rows = cmd.ExecuteNonQuery();
      Console.WriteLine($" ✅ Updated {updated_rows} rows in table
{TABLE_NAME}.");
    }
    catch (Exception e)
    {
      Console.WriteLine(e.Message);
    }
  }

  public static void Delete(NpgsqlConnection conn, string tag_name)
  {
    try
    {
      using var cmd = new NpgsqlCommand();
      cmd.Connection = conn;
      cmd.CommandText = $"DELETE FROM {TABLE_NAME} WHERE tag_name
```

```
= '{tag_name}'";
        Console.WriteLine(cmd.CommandText);
        int deleted_rows = cmd.ExecuteNonQuery();
        Console.WriteLine($" ✅ Deleted {deleted_rows} rows in table
{TABLE_NAME}.");
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
    }
  }

  public static void Query(NpgsqlConnection conn)
  {
    try
    {
        using var cmd = new NpgsqlCommand($"SELECT * FROM {TABLE_NAME}
where ts > '2025-01-01 00:00:00' and ts < '2025-12-30 00:00:10'", conn);
        using var reader = cmd.ExecuteReader();
        Console.WriteLine($" 👀 table:");
        while (reader.Read())
        {
            Console.WriteLine($"ts: {reader.GetDateTime(0)}, metric1:
{reader.GetInt32(1)}, metric2: {reader.GetDouble(2)}, tag_name:
{reader.GetString(3)}");
        }
        Console.WriteLine("\n");
    }
    catch (Exception e)
```

```
    {
        Console.WriteLine(e.Message);
    }
  }

  public static void Count(NpgsqlConnection conn)
  {
    try
    {
        using var cmd = new NpgsqlCommand($"SELECT COUNT(*) FROM
{TABLE_NAME}", conn);
        var count = cmd.ExecuteScalar();
        Console.WriteLine($" 👀 Select count rows in table: {count}\n");
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
    }
  }
}
```

4. 运行程序：

```
Bash
dotnet run
```

6.4.2 ODBC

6.4.2.1 前提条件

- 安装并启动 KaiwuDB Lite 数据库。有关详细信息，参见安装部署。

- 安装 .NET SDK，本示例使用版本为 8.0.121。

6.4.2.2 配置连接

1. 创建项目文件夹。

```bash
# 创建名为 test_odbc 的控制台应用程序项目
dotnet new console -o test_odbc

# 进入项目目录
cd test_odbc
```

2. 添加项目依赖，该命令会在项目文件夹下生成 test_odbc.csproj 项目配置文件和 Program.cs 代码文件。

```bash
dotnet add package System.Data.Odbc  # 本示例使用 System.Data.Odbc 版本为 9.0.10
```

3. 修改 Program.cs 代码文件，配置数据库连接。

```c#
#define KAIWUDB_LITE
using System;
using System.Data;
using System.Data.Odbc;

class Program
{
  #if KAIWUDB_LITE
  static string DSN = $"DRIVER={{PostgreSQL Unicode}};SERVER=localhost;PORT=36257;UID=admin;Pooling=false";
```

```
  #else
  static string DSN = $"DRIVER={{PostgreSQL
Unicode}};SERVER=localhost;PORT=5432;UID=postgres;PWD=123456;Database
=postgres;Pooling=false";
  #endif
  static string TABLE_NAME = "test";

  static void Main(string[] args)
  {
    try
    {
      using var conn = new OdbcConnection(DSN);
      conn.Open();

      Create(conn);
      Insert(conn);
      Query(conn);
      Update(conn, "tag2", 5, 5.5);
      Query(conn);
      Delete(conn, "tag1");
      Query(conn);
      conn.Close();
    }
    catch (Exception e)
    {
      Console.WriteLine(e.Message);
    }
  }
```

```
    public static void Create(OdbcConnection conn)
    {
        try
        {
            using var cmd = new OdbcCommand();
            cmd.Connection = conn;

            cmd.CommandText = $"DROP TABLE IF EXISTS {TABLE_NAME}";
            cmd.ExecuteNonQuery();

            #if KAIWUDB_LITE
            cmd.CommandText = $"CREATE TABLE {TABLE_NAME} (ts TIMESTAMP
NOT NULL, metric1 INT, metric2 DOUBLE) tags (tag_name VARCHAR)";
            #else
            cmd.CommandText = $"CREATE TABLE {TABLE_NAME} (ts TIMESTAMP
NOT NULL, metric1 INT, metric2 DOUBLE PRECISION, tag_name VARCHAR)";
            #endif
            cmd.ExecuteNonQuery();
            Console.WriteLine($"✅ Table {TABLE_NAME} created.");
        }
        catch (Exception e)
        {
            Console.WriteLine(e.Message);
        }
    }

    public static void Insert(OdbcConnection conn)
    {
        try
```

```
    {
        using var cmd1 = new OdbcCommand();
        cmd1.Connection = conn;
        cmd1.CommandText = $"INSERT INTO {TABLE_NAME} (ts, metric1,
metric2, tag_name) VALUES (now(), 1, 1.1, 'tag1'), (now(), 2, 2.2, 'tag2')";
        cmd1.ExecuteNonQuery();
        Console.WriteLine($" ✅ Inserted 2 rows into table {TABLE_NAME}
using non-prepared statement, transaction committed.");



        using var cmd = new OdbcCommand();
        cmd.Connection = conn;
        string sql = $@"INSERT INTO {TABLE_NAME} (ts, metric1, metric2,
tag_name) VALUES (?, ?, ?, ?), (?, ?, ?, ?)";
        cmd.CommandText = sql;

        // 添加参数
        int columns_count = 4;
        for(int i = 0; i < columns_count * 2; i++) {
          if (i % columns_count == 0)
            cmd.Parameters.Add(new OdbcParameter("?",
OdbcType.DateTime));
          else if (i % columns_count == 1)
            cmd.Parameters.Add(new OdbcParameter("?", OdbcType.Int));
          else if (i % columns_count == 2)
            cmd.Parameters.Add(new OdbcParameter("?", OdbcType.Double));
          else if (i % columns_count == 3)
            cmd.Parameters.Add(new OdbcParameter("?",
OdbcType.VarChar));
```

```
            }

            // 设置参数值, 可循环调用
            // for (int i = 0; i < ; i++)
            cmd.Parameters[0].Value = DateTime.SpecifyKind(DateTime.UtcNow,
DateTimeKind.Unspecified);
            cmd.Parameters[1].Value = 3;
            cmd.Parameters[2].Value = 3.3;
            cmd.Parameters[3].Value = "tag1";
            cmd.Parameters[4].Value = DateTime.SpecifyKind(DateTime.UtcNow,
DateTimeKind.Unspecified);
            cmd.Parameters[5].Value = 4;
            cmd.Parameters[6].Value = 4.4;
            cmd.Parameters[7].Value = "tag2";
            int inserted_rows = cmd.ExecuteNonQuery();
            // } end for

            Console.WriteLine($" ✅ Inserted {inserted_rows} rows into table
{TABLE_NAME} using parameterized statement, transaction committed.");
        }
        catch (Exception e)
        {
            Console.WriteLine(e.Message);
        }
    }

    public static void Update(OdbcConnection conn, string tag_name, int
metric1, double metric2)
    {
```

```
    try
    {
        using var cmd = new OdbcCommand($"UPDATE {TABLE_NAME} SET
metric1 = {metric1}, metric2 = {metric2} WHERE tag_name = '{tag_name}'",
conn);
        int updated_rows = cmd.ExecuteNonQuery();
        Console.WriteLine($"✅ Updated {updated_rows} rows in table
{TABLE_NAME}.");
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
    }
  }

  public static void Delete(OdbcConnection conn, string tag_name)
  {
    try
    {
        using var cmd = new OdbcCommand();
        cmd.Connection = conn;
        cmd.CommandText = $"DELETE FROM {TABLE_NAME} WHERE tag_name
= '{tag_name}'";
        Console.WriteLine(cmd.CommandText);
        int deleted_rows = cmd.ExecuteNonQuery();
        Console.WriteLine($"✅ Deleted {deleted_rows} rows in table
{TABLE_NAME}.");
    }
    catch (Exception e)
```

```
    {
        Console.WriteLine(e.Message);
    }
}

public static void Query(OdbcConnection conn)
{
    try
    {
        using var cmd = new OdbcCommand($"SELECT * FROM {TABLE_NAME}
where ts > '2025-01-01 00:00:00' and ts < '2025-12-30 00:00:10'", conn);
        using var reader = cmd.ExecuteReader();
        Console.WriteLine($"👀 table:");
        while (reader.Read())
        {
            Console.WriteLine($"ts: {reader.GetDateTime(0)}, metric1:
{reader.GetInt32(1)}, metric2: {reader.GetDouble(2)}, tag_name:
{reader.GetString(3)}");
        }
        Console.WriteLine("\n");
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
    }
}

public static void Count(OdbcConnection conn)
{
```

```
    try
    {
        using var cmd = new OdbcCommand($"SELECT COUNT(*) FROM
{TABLE_NAME}", conn);
        var count = cmd.ExecuteScalar();
        Console.WriteLine($"👀 Select count rows in table: {count}\n");
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
    }
  }
}
```

4. 运行程序：

```
Bash
dotnet run
```

6.5 EMQX

EMQX 是一款开源的高可用分布式 MQTT 消息服务器。KaiwuDB Lite 提供了专用接口，可以将 EMQX 管理的 MQTT 消息数据直接同步写入 KaiwuDB Lite 数据库。

用户在 EMQX Dashboard 中配置 KaiwuDB Lite 的连接和请求信息后，EMQX 能够自动将接收到的 MQTT 消息转换为包含 `INSERT` 语句的请求，并通过接口写入 KaiwuDB Lite 的指定数据表。

> 说明
>
> 当前不支持在同一个连接（connection）中并行处理多个主题的事件，即一个连接不能同时执行多条 prepare 插入语句，只能用于单一主题。

6.5.1 前提条件

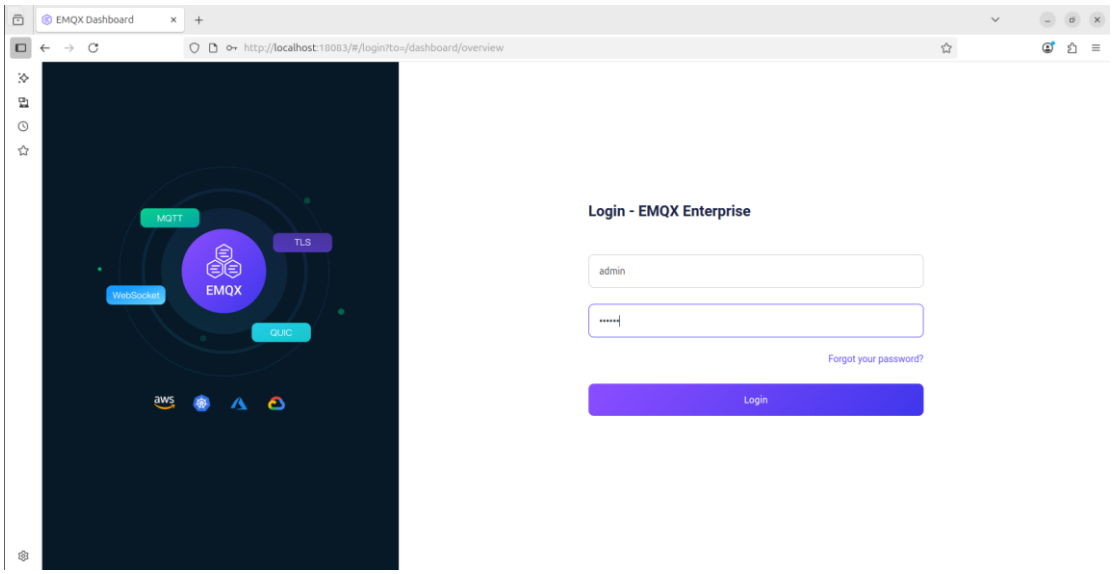
- 已安装并启动 KaiwuDB Lite 数据库。
- 已创建目标数据表，且表结构定义完整。
- 用户具备相应的数据库访问权限和数据写入权限。
- 已安装并启动 EMQX。具体安装和启动步骤，参见 EMQX 官方文档。

6.5.2 配置连接

以下配置步骤基于 EMQX 5.10.0 版本。不同版本的界面和配置方式可能存在差异，请以实际界面为准。

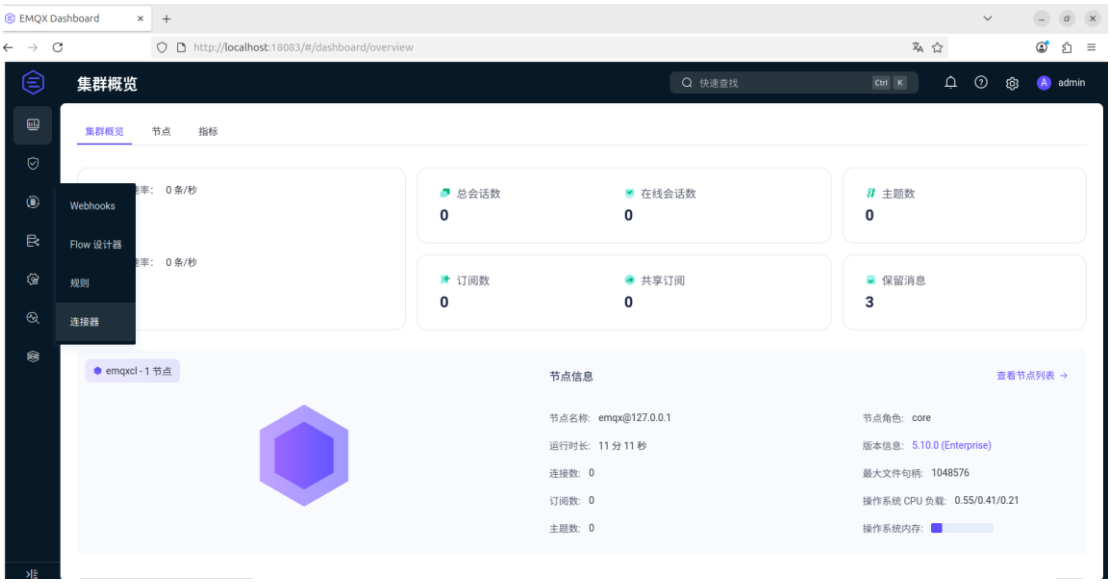
1. 登录 EMQX Dashboard。

   默认地址为 `http://localhost:18083`，默认用户名和密码为 `admin` 和 `public`。



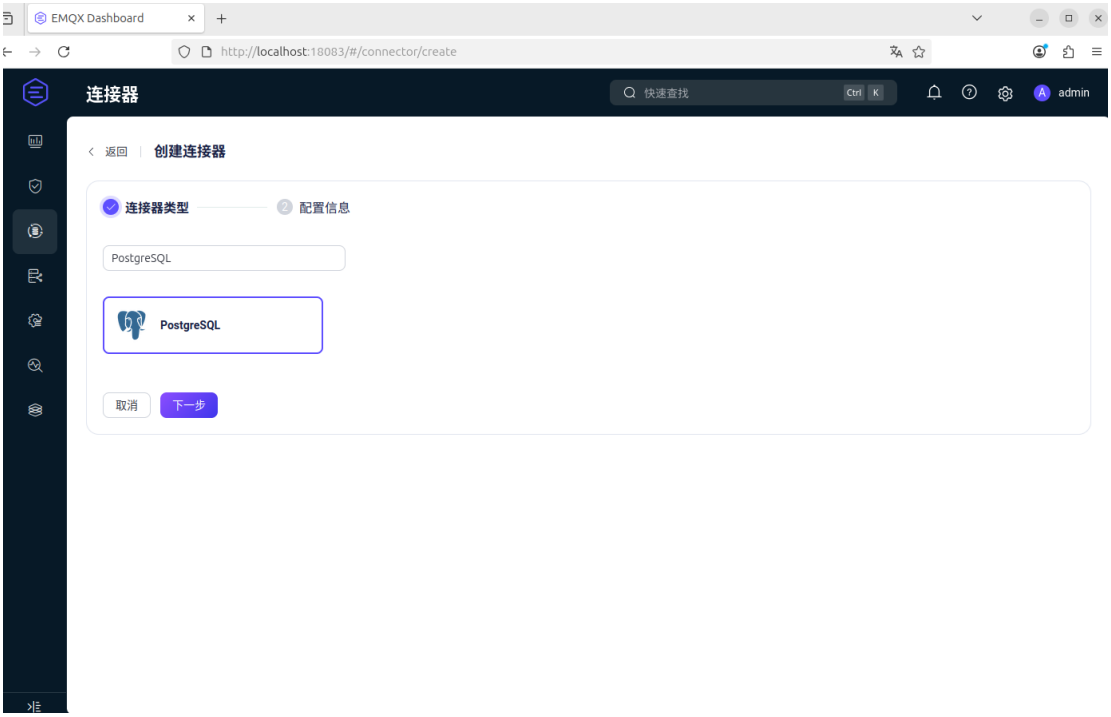
2. 创建连接器。

   a. 在左侧导航栏，选择集成 > 连接器。

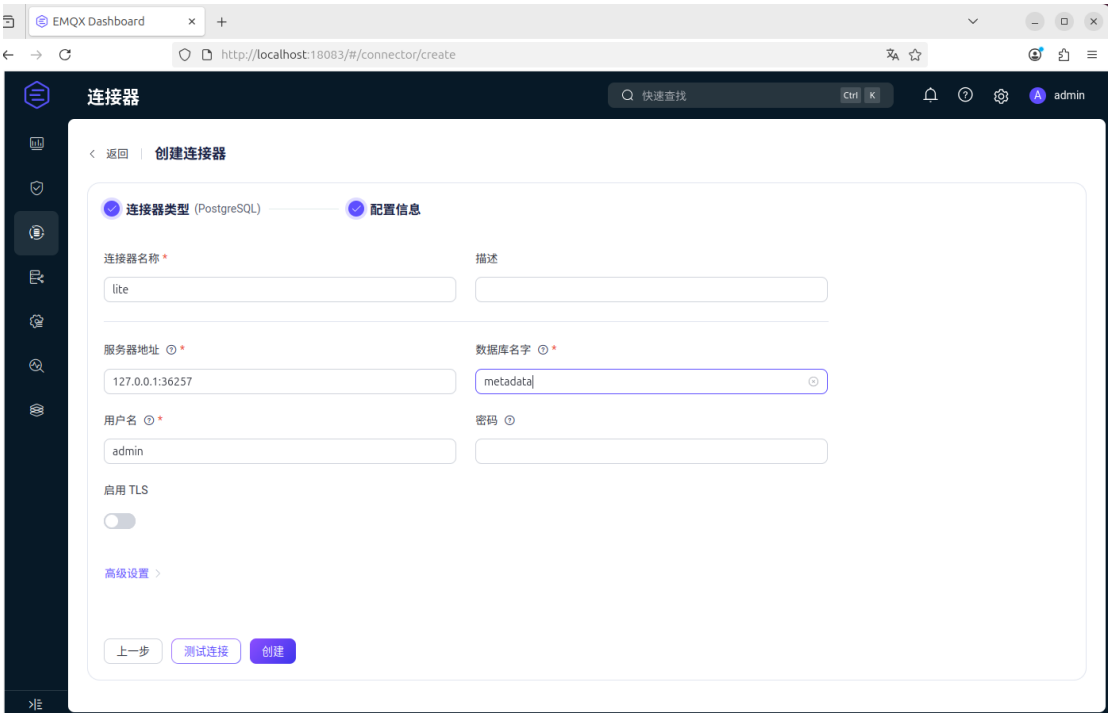


b. 单击创建按钮。

c. 选择连接器类型为 PostgreSQL，然后点击下一步。

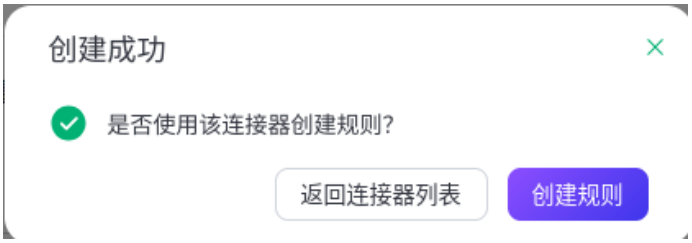


d. 在创建连接器页面，设置连接器名称、服务器地址、数据库名称（必须设置为 metadata）和用户信息，点击测试连接，确认连接成功后，点击创建。

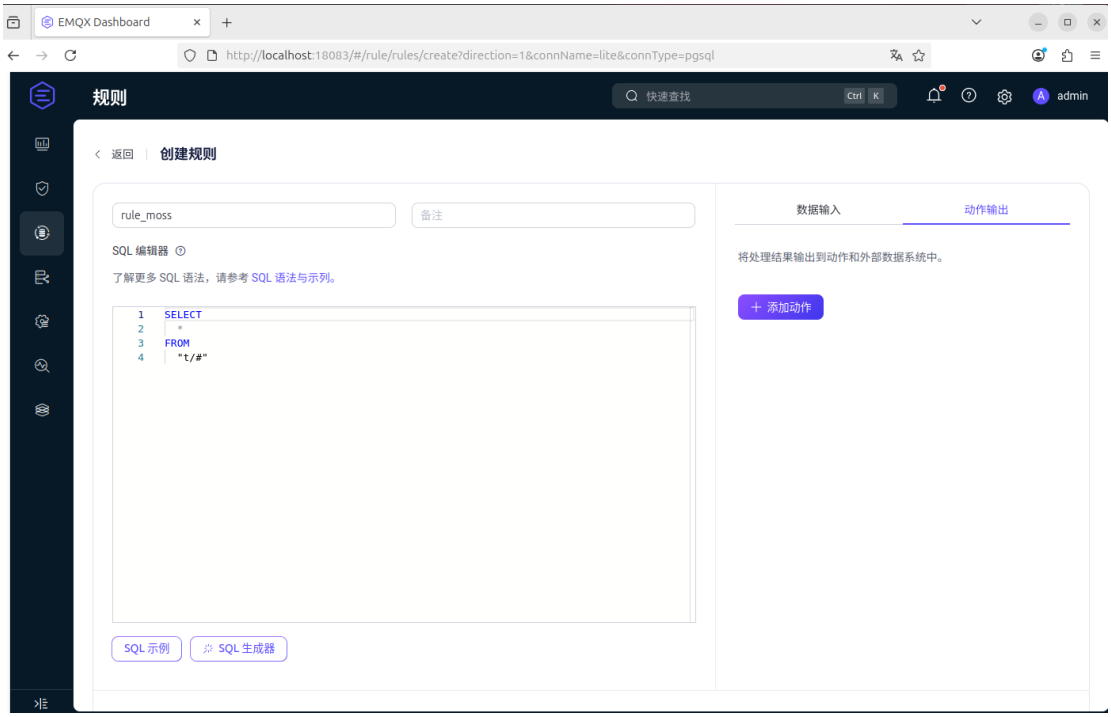


3. 创建规则，更多规则类型和对应的数据格式信息，参见 EMQX 官网文档。

   a. 点击创建成功提示框中的创建规则。

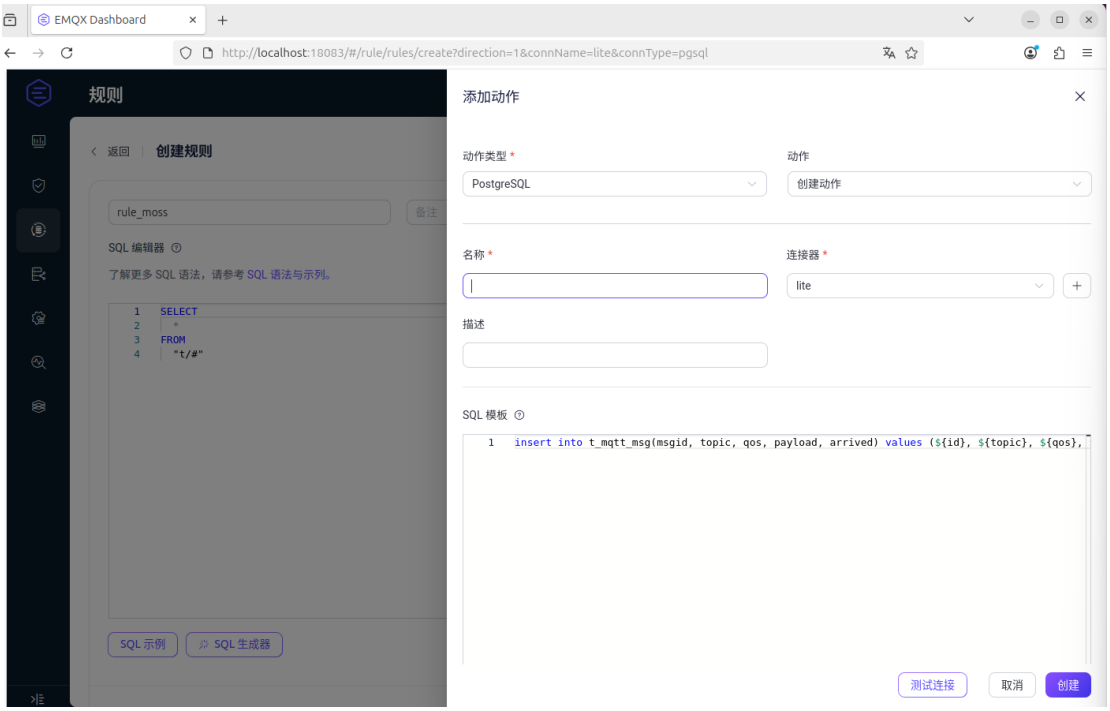


   b. 在创建规则页面，设置规则名称，点击右侧的添加动作。



c. 在添加动作页面，设置动作类型为 PostgreSQL， 自定义动作名称，选择已建好的 KaiwuDB Lite 连接器，指定写入的 SQL 规则，然后点击创建。

> 提示：
>
> 如果测试连接时提示 `resource down: {unhealthy_target,#{severity => error,error_code => <<"22000">>,error_codename => data_exception}}`, 需要在连接器的高级设置中禁用预处理语句。



4. 测试连接。

   a. 使用 MQTTX 工具发送测试消息：

```
C++

mqttx pub -i emqx_c -t t/1 "hello KaiwudbLite"
```

   b. 在 KaiwuDB Lite 中查询数据以验证消息是否成功写入：

```
Shell

select * from t_mqtt_msg;
 id |          msgid           | sender | topic | qos | retain |        payload         |
arrived
----+--------------------------------+--------+-------+-----+--------+------------------------
----------+---------
  1 | 00063C26D620607AF4450003963B0002 |        | t/1   |   0 |        | { "msg":
"hello KaiwudbLite" } |
 (1 row)


 select * from emqx_client_events;
```

```
 id | clientid |      event       |      created_at
----+----------+--------------------+-------------------------
  2 | emqx_c   | client.disconnected | 2025-08-12 07:29:51.466
  3 | emqx_c   | client.disconnected | 2025-08-12 07:31:57.387
  4 | emqx_c   | client.connected    | 2025-08-12 07:34:58.394
  5 | emqx_c   | client.disconnected | 2025-08-12 07:34:59.307
(4 rows)
```

7. SQL

7.1 数据类型

KaiwuDB Lite 支持存储和处理以下数据类型：

- 时间戳类型
- 数值类型
- 布尔类型
- 字符类型

7.1.1 时间戳类型

7.1.1.1 基础信息

时间戳包括 TIMESTAMP 和 TIMESTAMPTZ 两种类型：

- TIMESTAMP 以协调世界时（UTC） 格式存储日期和时间
- TIMESTAMPTZ 将时间戳数值从 UTC 转换为客户端会话时区。

名称	别名	存储空间
TIMESTAM	DATETIME, TIMESTAMP WITHOUT	8 字节

P	TIME ZONE	
TIMESTAMPTZ	TIMESTAMP WITH TIME ZONE	8 字节

时间戳精度

KaiwuDB Lite 支持使用 timestamp(n) 指定秒字段保留的小数位数，以控制时间戳的精度。可设置的精度包括 秒、毫秒、微秒、纳秒。例如，TIMESTAMPT(3) 表示将时间部分截断为毫秒。

名称	别名	精度
TIMESTAMP(0)	TIMESTAMP_S	秒
TIMESTAMP(1) TIMESTAMP(2) TIMESTAMP(3)	TIMESTAMP_MS	毫秒
TIMESTAMP(4) TIMESTAMP(5) TIMESTAMP(6)	TIMESTAMP	微秒
TIMESTAMP（大于 6）	TIMESTAMP_NS	纳秒

TIMESTAMPTZ 默认精确到微秒，例如：2020-02-12 07:23:25.123456。

时间戳写入方式

KaiwuDB Lite 支持以下方式写入时间戳数据：

- now() 函数：默认时间戳类型为 TIMESTAMPTZ，支持转为微秒精度的 TIMESTAMP ，暂不支持转为其他精度。
- STRING 形式的时间戳。

7.1.1.2 示例

示例 1：创建带有 TIMESTAMP 列的 metrics 表

```
SQL
CREATE TABLE timestamps (ts TIMESTAMP NOT NULL, c1 TIMESTAMPTZ);
```

示例 2：插入 TIMESTAMP 类型的数据

```
SQL
-- 插入数据
> INSERT INTO timestamps VALUES (now(), '2024-10-24 04:09:04.437');


-- 查询数据
> SELECT * FROM timestamps;
        ts         |          c1
-------------------------+---------------------------
 2025-05-15 10:07:38.767 | 2024-10-24 04:09:04.437+00
(1 row)
```

7.1.2 数值类型

KaiwuDB Lite 支持的数值类型包括整数类型和浮点类型。

7.1.2.1 整数类型

7.1.2.1.1 基础信息

KaiwuDB Lite 支持以下整数类型：

名称	别名	存储空间	取值范围
TINYINT	INT1	1 字节	-128 ~ 127
SMALLINT	• INT2 • SHORT • INT16	2 字节	-32768 ~ +32767
INTEGER	• INT	4 字节	-2147483648 ~ +2147483647

	• INT4 • INT32 • SIGNED		
BIGINT	• INT8 • INT64 • LONG	8 字节	-9223372036854775808 ~ +9223372036854775807

整数类型支持数值文本输入，例如：'42'、'-1234'。

7.1.2.1.2 示例

示例 1：创建具有整数类型列的 metrics 表

```
SQL
CREATE TABLE ints(ts TIMESTAMP NOT NULL, c1 INT2, c2 SMALLINT, c3 SHORT, c4
INT4, c5 INT, c6 INT8, c7 BIGINT, c8 LONG);
```

示例 2：插入整数类型的数据

```
SQL
-- 插入数据
> INSERT INTO ints VALUES (now(), 128, -1, 100, 1024, 21, 171, -531, 100);


-- 查询数据
> SELECT * FROM ints;
          ts          |c1 |c2|c3 | c4 |c5|c6 | c7 |c8
------------------------+---+--+---+----+--+---+----+---
 2025-05-15 10:30:32.496 |128|-1|100|1024|21|171|-531|100
(1 row)
```

7.1.2.2 浮点型

7.1.2.2.1 基础信息

KaiwuDB Lite 支持两类浮点类型：FLOAT 和 DOUBLE。

名称	别名	存储空间
FLOAT	• FLOAT4 • REAL	4 字节
DOUBLE	FLOAT8	8 字节

浮点类型支持数值文本输入，例如：'1.414'、'1234'。

7.1.2.2.2 示例

示例 1：创建具有浮点数列的 metrics 表

```
SQL
CREATE TABLE floats(ts TIMESTAMP NOT NULL, c1 FLOAT, c2 REAL,c3 DOUBLE, c4
FLOAT8);
```

示例 2：插入浮点类型的数据

```
SQL
-- 插入数据
> INSERT INTO floats VALUES (now(), 128, -1.1, 10.0, 1.024);


-- 查询数据
> SELECT * FROM floats;
        ts         | c1  | c2 | c3 | c4
-------------------------+-------+------+------+-------
 2025-05-15 10:33:09.992 | 128.0 | -1.1 | 10.0 | 1.024
(1 row)
```

7.1.3 布尔类型

7.1.3.1 基础信息

BOOL 类型用于存储布尔值 FALSE 或 TRUE。

名称	别名	存储空间
BOOL	BOOLEAN	1 字节

布尔类型包含两个预定义常量：TRUE 和 FALSE（不区分大小写）。

7.1.3.2 示例

示例 1：创建具有布尔列的 metrics 表

```
SQL
CREATE TABLE bools(ts TIMESTAMP NOT NULL,c1 BOOL,c2 BOOLEAN);
```

示例 2：插入布尔类型的数据

```
SQL
-- 插入数据
> INSERT INTO bools VALUES (now(), true, false);


-- 查询数据
> SELECT * FROM bools;
          ts          | c1 | c2
------------------------+----+-------
 2025-05-15 10:34:31.141 | true | false
(1 row)
```

7.1.4 字符类型

7.1.4.1 VARCHAR

7.1.4.1.1 基础信息

VARCHAR 是可变长字符类型，其存储长度取决于数据的实际长度。

> 提示
>
> KaiwuDB Lite 支持 VARCHAR(n)格式，但最大长度 n 实际不起作用，仅用于兼容性。

名称	别名
VARCHAR	• CHAR • BPCHAR • STRING • TEXT

7.1.4.1.2 示例

示例 1：创建带有 VARCHAR 类型的 metrics 表

```
SQL
CREATE TABLE varchars(ts TIMESTAMP NOT NULL, c1 VARCHAR, c2
VARCHAR(65536));
```

示例 2：插入 VARCHAR 类型的数据

```
SQL
-- 插入数据
> INSERT INTO varchars VALUES (now(), 'true', 'false');


-- 查询数据
> SELECT * FROM varchars;
        ts         | c1 | c2
-------------------------+------+-------
 2025-05-15 10:35:56.327 | true | false
(1 row)
```

7.1.5 类型转换

KaiwuDB Lite 支持数据类型之间的转换，类型转换方式包括显式类型转换和隐式类型转换。

- 显式类型转换：通过 CAST 函数或 :: 操作符将一种数据类型的值明确转换为另一种类型的值。
- 隐式类型转换：KaiwuDB Lite 根据上下文环境和类型转换规则，自动将数据转换为目标类型。符合转换规则的数据会在写入时自动转换为对应类型，不符合转换规则的数据则会报错。

7.1.5.1 类型转换支持

下表列出了 KaiwuDB Lite 中支持的类型转换：

From\TO	TIMESTAMP	TIMESTAMPTZ	TINYINT	SMALLINT	INTEGER	BIGINT	FLOAT	DOUBLE	BOOL	VARCHAR
TIMESTAMP	-	Y	C	C	C	Y	N	N	N	Y
TIMESTAMPTZ	C	-	C	C	C	Y	N	N	N	Y
TINYINT	Y	Y	-	Y	Y	Y	Y	Y	Y	Y
SMALLINT	Y	Y	C	-	Y	Y	Y	Y	Y	Y
INTEGER	Y	Y	C	C	-	Y	Y	Y	Y	Y
BIGINT	C	C	C	C	C	-	Y	Y	Y	Y

FLOAT	N	N	C	C	C	C	-	Y	Y	Y
DOUBLE	N	N	C	C	C	C	Y	-	Y	Y
BOOL	N	N	Y	Y	Y	Y	Y	Y	-	Y
VARCHAR	C	C	C	C	C	C	C	C	C	-

图例说明：

- Y：支持转换
- N：不支持转换
- -：不需要转换（相同类型）
- C：有条件转换（需满足特定条件）

7.1.5.2 转换规则和注意事项

7.1.5.2.1 精度丢失和数据溢出风险

以下转换可能导致精度丢失或数据溢出：

- 浮点数转整数：小数部分会被截断，例如 3.14 → 3
- 高精度整数转低精度整数：数值超出目标类型范围时会发生溢出
- 高精度浮点转低精度浮点：可能损失精度

7.1.5.2.2 有条件转换详细说明

- 时间类型转换
  - TIMESTAMPTZ → TIMESTAMP：仅支持转换为微秒精度的 TIMESTAMP，不支持其他精度
  - TIMESTAMP/TIMESTAMPTZ → 整数类型：需确保数值在目标类型范围内

- 数值类型转换
  - SMALLINT → TINYINT：需确保数值在 TINYINT 范围内（-128 到 127）
  - INTEGER → TINYINT/SMALLINT：需确保数值在目标类型范围内
  - BIGINT → 其他整数类型：需确保数值在目标类型范围内
  - BIGINT → TIMESTAMP/TIMESTAMPTZ：极值会被转换为无穷大或无穷小的时间戳；数值超出有效范围时，可能在内部计算中发生溢出错误
- 浮点类型转换
  - 浮点数 → 整数类型：需确保数值在目标整数类型范围内，小数部分会被截断
- 字符串转换
  - 字符串 → 时间类型：字符串必须符合相应的时间格式，格式不符时转换失败
  - 字符串 → 数值类型：字符串必须表示有效数字，包含非数字字符时转换失败
  - 字符串 → 布尔类型：仅支持字符串 `'0'`（转为 `FALSE`）和 `'1'`（转为 `TRUE`），其他字符串转换失败

7.1.5.2.3 布尔类型转换规则

- 数值类型 → 布尔类型
  - 数字 `0` 转换为 `FALSE`
  - 其他所有数值转换为 `TRUE`
- 布尔类型 → 其他类型
  - 布尔类型 → 数值类型：`FALSE` 转为 `0`，`TRUE` 转为 `1`
  - 布尔类型 → 字符串：`FALSE` 转为 `'false'`，`TRUE` 转为 `'true'`

7.2 函数

7.2.1 条件和类函数运算符

具有特殊评估规则的运算符如下：

运算符	描述
cast(expr as type_name)	转换 expr 为指定类型 type_name，其中 expr 是需要转换的 SQL 表达式，type_name 为目标类型名称。 例如，cast(123 as VARCHAR) 表示将数字 123 转换为 VARCHAR 类型的字符串 '123'。
ifnull(expr1, expr2)	如果 expr1 不等于 NULL，返回 expr1；否则返回 expr2。 expr1 和 expr2 必须类型相容。 例如，ifnull(NULL, 'default') 会返回 'default'，因为 expr1 为 NULL。
nullif(expr1, expr2)	如果 expr1 等于 expr2，返回 NULL；否则返回 expr1。 expr1 和 expr2 必须类型相容。 例如，nullif(10, 10) 会返回 NULL，因为 expr1 和 expr2 相等。

7.2.2 聚合函数

函数	描述
avg(expr)	计算指定列或 SQL 表达式所有非 NULL 值的均值。expr 为列名或 SQL 表达式。 例如，avg(price+3) 表示计算 price 列中所有非 NULL 值加上 3 后的均值。 数据类型必须是整数类型或浮点类型。
count()	计算表中的行数， 包括 NULL 值。
count(*)	计算表中的行数， 包括 NULL 值。
count(expr)	expr 可以为列名或数值常量： • expr 为列名时，计算指定列不为 NULL 值的行数。

	• expr 为数值常量时，计算表中的行数，包括 NULL 值。 例如，count(price) 表示返回 price 列中非 NULL 值的行数，count(1) 表示返回表中的总行数， 包括 NULL 值。
count(distinct expr)	expr 可以为列名或数值常量： • expr 为列名时，计算指定列不为 NULL 值的不同值的行数。 • expr 为数值常量时，返回 1； 例如，count(distinct price) 表示返回 price 列中非 NULL 值的不同值的行数，count(1) 返回 1。 如果查询的表为空表，均返回 0。
diff(expr)	计算表中指定列当前行与前一行非 NULL 值之间的差值。如果当前行的值为 NULL 或指定分区中仅有两行数据且首行为 NULL，则返回 NULL。如果前一行为 NULL，则使用前面最近的非 NULL 行进行计算。 语法格式为 diff (<expr>) OVER (PARTITION BY <column_list> ORDER BY <column_list>)。 其中：expr 为 SQL 表达式，指定列的数据类型必须支持减法运算，目前支持时间戳类型和数值类型。 注意：计算时的行顺序由 ORDER BY 子句决定。如果不指定 ORDER BY，则顺序随机，相同数据在不同执行时可能产生不同结果。因此必须明确指定 ORDER BY 子句以保证结果一致性。
first(expr)	计算指定列按时间戳列增序排序后的第一行非 NULL 值。expr 为列名。 例如 first(price) 表示返回 price 列中按时间戳列增序排序后的第一行非 NULL 的值。 如果查询列的所有值均为 NULL 值，则返回 NULL。
first(*)	返回所查询表所有列按时间戳列增序排序后第一行的非 NULL 值。 如果查询的表为空表，则返回 NULL。
first_row(expr)	计算指定列按时间戳列增序排序后的第一行的值。expr 为列名。

first_row(*)	返回所查询表所有列按时间戳列增序排序后第一行的值。
interpolate(expr)	插值函数，用于填补时间序列数据中的缺失值，必须与 time_bucket_gapfill() 函数配合使用。其中， • time_bucket_gapfill() 负责生成完整的时间桶 • interpolate() 负责填补相应时间桶中的 NULL 值 参数说明： expr 必须是数值类型的表达式，支持的数据类型包括：TINYINT、SMALLINT、INT、BIGINT、DECIMAL、FLOAT、DOUBLE。具体使用示例参见插值查询章节。
interpolate(expr, mode)	带模式的插值函数，提供多种填补策略。必须与 time_bucket_gapfill() 函数配合使用。其中： • time_bucket_gapfill() 负责生成完整的时间桶 • interpolate() 负责基于插值模式填补相应时间桶中的数值 参数说明： • expr 必须是结果为数值类型的聚合函数，支持的数据类型包括：TINYINT、SMALLINT、INT、BIGINT、DECIMAL、FLOAT、DOUBLE。 • mode：用于指定补值模式，取值包括 ◦ 常量值：使用指定常量值填补，常量类型必须与 expr 结果类型兼容。 ◦ prev：使用前一个非空值补全 ◦ next：使用后一个非空值补全 ◦ linear：线性插值，在 [前一个非空值， 后一个非空值] 区间内进行线性计算，支持递增和递减，如果当前值大于后值时，按递减方式插值。 ◦ null：使用 NULL 值补全，同 interpolate(expr)。 具体使用示例参见插值查询章节。
last(expr)	计算指定列按时间戳列增序排序后最后一行的非 NULL 值。expr 为

	列名。 如果查询列的所有值均为 NULL 值，则返回 NULL。
last(*)	返回所查询表所有列按时间戳列增序排序后最后一行的非 NULL 值。 如果查询的表为空表，则返回 NULL。
last_row(expr)	计算指定列按时间戳列增序排序后最后一行的值。expr 为列名。
last_row(*)	返回所查询表所有列按时间戳列增序排序后最后一行的值。
max(expr)	计算指定列或 SQL 表达式的最大值。expr 为列名或 SQL 表达式。
min(expr)	计算指定列或 SQL 表达式的最小值。expr 为列名或 SQL 表达式。
sum(expr)	计算指定列或 SQL 表达式所有非 NULL 值的和。expr 为列名或 SQL 表达式。 例如，sum(price+3) 表示计算 price 列中所有非 NULL 值加上 3 后的总和。 数据类型必须是整数类型或浮点类型。

7.2.3 日期和时间函数

函数	描述
age(expr1)	计算 expr1 和当前时间之间的时间间隔。expr1 必须明确为 TIMESTAMP 或 TIMESTAMP WITH TIME ZONE 类型，如 TIMESTAMP '2025-03-05'。
age(expr1, expr2)	计算 expr1 和 expr2 之间的时间间隔。expr1 和 expr2 必须明确为 TIMESTAMP 或 TIMESTAMP WITH TIME ZONE 类型。 例如，age(timestamp '2020-02-26 15:35:00', timestamp '2025-02-26 15:35:00')表示计算 '2020-02-26 15:35:00' 和 '2025-02-26 15:35:00' 之间的时间间隔，结果为：5 years。
current_timestamp	获取当前日期和时间，包含时区。

date_trunc(part, expr)	截断日期到指定的精度。 其中，part 为字符串，指定要截取的精度，支持的值如下： • 'millennium'：格里高里千年 • 'century'：格里高里世纪 • 'decade'：格里高里十年 • 'year'：年 • 'yearweek'：ISO 周年 • 'quarter'：季度 • 'month'：月 • 'week'：周 • 'day'：格里高里天 • 'dayofweek'：一周中的天 • 'dayofyear'：一年中的天 • 'isodow'：ISO 一周中的天 • 'isoyear'：ISO 一年中的天 • 'hour'：时 • 'minute'：分 • 'second'：秒 • 'epoch'：从 1970 年 1 月 1 日起的秒数 • 'milliseconds'：毫秒 • 'microseconds'：微秒 expr 指定要操作的 SQL 日期表达式， 必须明确为 TIMESTAMP 或 TIMESTAMP WITH TIME ZONE 类型，例如 TIMESTAMP '2025-03-05'。 例如，date_trunc('hour', timestamp '2025-02-26 15:35:00') 表示将 '2025-02-26 15:35:00' 截断到小时，结果为 '2025-02-26 15:00:00'。
experimental_strftime(format_	依据指定的格式字符串对时间值进行格式化。例如 SELECT experimental_strftime('%Y-%m-%d %H:%M:%S',

string, timestamp), experimental_strftime(timestamp, format_string)	CURRENT_TIMESTAMP::timestamp); 表示将当前时间格式化为年-月-日 小时：分钟：秒的形式。 其中， • format_string 定义了输出格式的字符串，支持以下格式说明符的组合： ◦ %Y：四位数年份，例如：2025 ◦ %m：两位数月份（01-12） ◦ %d：两位数日期（01-31） ◦ %H：24 小时制小时（00-23） ◦ %M：分钟（00 - 59） ◦ %S：秒（00 - 59） ◦ %w：一周中的星期几（0 - 6，其中 0 表示星期日） ◦ %a：星期几的英文缩写，例如：Mon ◦ %A：星期几的英文全称，例如：Monday ◦ %b：月份的英文缩写，例如：Jan ◦ %B：月份的英文全称，例如：January • timestamp_value：指定要进行格式化的时间值，支持 TIMESTAMP 和 TIMESTAMPTZ 类型。
extract(field from expr)	从时间戳表达式 expr 中抽取指定的子域。 其中，field 为需要抽取的子域，支持的字段包括： • ‘era'：格里高里时代 • 'millennium'：格里高里千年 • 'century'：格里高里世纪 • 'decade'：格里高里十年 • 'year'：年 • 'yearweek'：ISO 周年 • 'quarter'：季度 • 'month'：月 • 'week'：周

	• 'day'：格里高里天 • 'dayofweek'：一周中的天 • 'dayofyear'：一年中的天 • 'isodow'：ISO 一周中的天 • 'isoyear'：ISO 一年中的天 • 'hour'：时 • 'timezone_hour'：时区偏移小时 • 'minute'：分 • 'timezone_minute'：时区偏移分钟 • 'second'：秒 • 'timezone'：时区偏移秒 • 'epoch'：从 1970 年 1 月 1 日起的秒数 • 'milliseconds'：毫秒 • 'microseconds'：微妙 expr 必须明确为 TIMESTAMP 或 TIMESTAMP WITH TIME ZONE 类型。 例如，extract('year' from timestamp '2025-02-26 15:35:00') 表示从 '2025-02-26 15:35:00' 中提取年份，结果为 2025。
now()	获取当前日期和时间，包含时区，与 current_timestamp 功能相同。

time_bucket(t_expr, i_expr)	将时间戳按指定的间隔宽度进行截断（向下取整）。
	其中：
	• t_expr：时间戳表达式，类型为 TIMESTAMP 或 TIMESTAMPTZ，通常为表列。
	• i_expr：时间间隔表达式，类型为 INTERVAL，必须指定时间单位。支持的时间单位如下：
	  ◦ MICROSECOND / MICROSECONDS（微秒）
	  ◦ MILLISECOND / MILLISECONDS（毫秒）

	◦ SECOND / SECONDS（秒） ◦ MINUTE / MINUTES（分钟） ◦ HOUR / HOURS（小时） ◦ DAY / DAYS（天） ◦ WEEK / WEEKS（周） ◦ MONTH / MONTHS（月） ◦ YEAR / YEARS（年） 例如 time_bucket(timestamp '2025-02-26 15:35:00', interval '1 hour') 表示将 '2025-02-26 15:35:00' 按 1 小时的间隔截断，结果为 '2025-02-26 15:00:00'。
time_bucket_gapfill(t_expr, i_expr)	将时间戳按指定间隔分箱并补全范围内缺失的时间点。补全行中其它列缺失值填充 NULL。 其中： • t_expr：时间戳表达式，类型为 TIMESTAMP 或 TIMESTAMPTZ，通常为表列。 • i_expr：时间间隔表达式，类型为 INTERVAL，必须指定时间单位。支持的时间单位如下： ◦ MICROSECOND / MICROSECONDS（微秒） ◦ MILLISECOND / MILLISECONDS（毫秒） ◦ SECOND / SECONDS（秒） ◦ MINUTE / MINUTES（分钟） ◦ HOUR / HOURS（小时） ◦ DAY / DAYS（天） ◦ WEEK / WEEKS（周） ◦ MONTH / MONTHS（月） ◦ YEAR / YEARS（年） time_bucket_gapfill 函数必须与 GROUP BY 一起时，补全规则取决于分组情况： • 如果分组（除 time_bucket_gapfill 外的其他分组项）中仅有一

	行数据：时间范围即为该行的 time_bucket_gapfill 值，不需补全。 • 如果分组中有多行数据：时间范围为该组 time_bucket_gapfill 值的最小值到最大值，系统会在此范围内补全缺失行。 使用要求： • 如果需要同时查询其他列信息，且待查询的列不在 GROUP BY 指定的分组项内，需要使用聚合函数来处理这些列。 • time_bucket_gapfill 必须作为查询或者查询中的顶层表达式，不能嵌套在其他函数内。
timeofday	返回包含当前日期、时间以及时区信息的字符串，例如 Thu Mar 27 03:23:06.502350 2025 +0000， 默认显示 UTC 时间。

7.2.4 数学与统计函数

函数	描述
abs(expr)	计算表达式 expr 的绝对值，expr 必须为数值表达式。 例如，abs(-5)表示计算 -5 的绝对值，返回 5。
acos(expr)	计算表达式 expr 的反余弦。expr 必须为数值表达式，且数值范围为 [-1, 1]。 例如，acos(0.5) 表示计算 0.5 的反余弦，返回 1.0471975511965979。
asin(expr)	计算表达式 expr 的反正弦。expr 必须为数值表达式，且数值范围为 [-1, 1]。 例如，asin(0.5) 表示计算 0.5 的反正弦，返回 0.5235987755982989。
atan(expr)	计算表达式 expr 的反正切。expr 必须为数值表达式。 例如，atan(1000000000) 表示计算 1000000000 的反正切，返回 1.5707963257948967。
atan2(y, x)	计算 y, x 的反正切。y 和 x 必须为数值表达式。

	例如，atan2(1000000000, 1)，表示计算点 （1000000000， 1） 的反正切，返回 1.5707963257948967。
cbrt(expr)	计算表达式 expr 的立方根。expr 必须为数值表达式。 例如，cbrt(8)表示计算 8 的立方根，返回 2.0。
ceil(expr)	计算表达式 expr 的向上取整数值。expr 必须为数值表达式。 例如，ceil(-8.5) 表示计算 -8.5 的向上取整，返回 -8。
ceiling(expr)	计算表达式 expr 的向上取整数值。expr 必须为数值表达式。 例如，ceiling(-8.5) 表示计算 -8.5 的向上取整，返回 -8。
corr(expr1, expr2)	计算 expr1 和 expr2 的相关系数。expr1 和 expr2 必须是数值类型的列名，且在数据表中包含多行记录。 例如，corr(price, amount) 表示计算 price 列和 amount 列之间的相关系数，
cos(expr)	计算表达式 expr 的余弦。expr 必须为数值表达式。 例如，cos(3) 表示计算 3 的余弦，返回 -0.9899924966004454。
cot(expr)	计算 expr 的余切。expr 必须为数值表达式，且不能为 0。 例如，cot(0.1) 表示计算 0.1 的余切，返回 9.966644423259238。
div(expr1, expr2)	计算 expr1/expr2 的整数商。expr1 和 expr2 必须为数值表达式。 例如，div(7, 2) 表示计算 7 除以 2 的整数商，返回 3。
exp(expr)	计算 expr 的自然指数值，即 e^expr， expr 必须为数值表达式。 例如，exp(1) ，表示计算 e 的 1 次方，返回 2.718281828459045。
floor(expr)	计算 expr 的向下取整数值。expr 必须为数值表达式。 例如，floor(1.8) ，表示计算 1.8 的向下取整值，返回 1。
ln(expr)	计算表达式 expr 的自然对数。expr 必须为数值表达式，且必须大于 0。 例如，ln(0.00001) ，表示计算 0.00001 的自然对数，返回 -11.512925464970229。
log(expr)	计算表达式 expr 以 10 为底的对数。expr 必须为数值表达式，且必

	须大于 0。 例如，log(1000) 表示计算 1000 的对数，返回 3.0。
mod(expr1, expr2)	计算 expr1 除以 expr2 的余数。expr1 和 expr2 必须为数值表达式。余数的符号同 expr1 的符号。余数的类型同 expr1 和 expr2 可转换到的公共类型。 例如，mod(7, 3) ，表示计算 7 除以 3 的余数，返回 1。
pow(expr1, expr2)	计算 expr1 的 expr2 次方值，即 expr1^expr2。expr1 和 expr2 必须为数值表达式。 例如，pow(2, 3) 表示计算 2 的 3 次方，返回 8.0。
power(expr1, expr2)	计算 expr1 的 expr2 次方值，即 expr1^expr2。expr1 和 expr2 必须为数值表达式。 例如，power(2, 3) 表示计算 2 的 3 次方，返回 8.0。
radians(expr)	计算度 expr 对应的弧度。expr 必须为数值表达式。 例如，radians(400) ，表示计算 400 度对应的弧度，返回 6.981317007977318 。
random()	获取区间 [0, 1) 范围内的一个随机数。
round(expr)	计算 expr 最接近的整数值。expr 必须为数值表达式。 例如，round(2.5) 表示 2.5 四舍五入为 3。
sign(expr)	计算 expr 的符号。expr 必须为数值表达式。 返回值 -1 表示负数，0 表示 0，1 表示正数。 例如，sign(-3) 返回 -1，表示 -3 为负数。
sin(expr)	计算表达式 expr 的正弦。expr 必须为数值表达式。 例如，sin(10) 表示计算 10 的正弦，返回 -0.5440211108893698。
sqrt(expr)	计算 expr 的平方根。expr 必须为数值表达式，且必须大于等于 0。 例如，sqrt(9) 表示计算 9 的平方根，返回 3.0。
tan(expr)	计算表达式 expr 的正切。expr 必须为数值表达式。 例如，tan(45) 表示计算 45 的正切，返回 1.6197751905438615。

trunc(expr)	截断 expr 到整数。expr 必须为数值表达式或数值类型的列名。 例如，trunc(3.7) 表示截断 3.7，返回 3。
width_bucket(value, min_value, max_value, num_buckets)	将数值划分到等宽区间（桶）中，并返回该数值所属的桶编号。函数会根据指定的最小值、最大值和桶数量，计算每个桶的宽度，然后确定输入值所在的桶。 其中： • value：要分配到桶中的数值，支持 DOUBLE 类型。 • min_value：区间的下限值，支持 DOUBLE 类型。 • max_value：区间的上限值，支持 DOUBLE 类型。 • num_buckets：要创建的桶数量，支持 INTEGER 类型，必须为正整数。 返回值为正整数，表示输入值所在的桶编号（从 1 开始计数）。如果输入值小于最小值，返回 0；如果输入值大于或等于最大值，返回 num_buckets + 1。

7.2.5 字符串函数

函数	描述
concat(expr, ...)	将多个表达式连接成一个字符串。表达式的数据类型会自动转换成字符串类型。 如果某个表达式为 NULL，则会被忽略。 例如，concat('Hello', ' ', 'World') 会返回 'Hello World'。
left(expr, n)	从字符串表达式 expr 中提取最左边的 n 个字符。 • 如果 n 大于 expr 的长度，返回整个 expr。 • 如果 n 为负值，提取的字符串长度为 length(expr) - abs(n)。 • 如果 n 为 0，返回空字符串 ''。 • 如果 expr 为 NULL，返回 NULL。 例如，left('abcdef', 3) 返回 'abc'，left('abcdef', 10) 返回

	'abcdef'。
length(expr)	计算 expr 中字符的数量。 • 如果 expr 为字符串，返回字符数。 • 如果 expr 为 NULL，返回 NULL。 • 如果 expr 为空字符串 ''，返回 0。 例如，length('hello') 返回 5；length('') 返回 0。
lower(expr)	将 expr 转换为小写字母。expr 必须为字符串表达式。 例如，lower('HELLO') 返回 'hello'。
lpad(expr1, n, expr2)	使用字符串 expr2 从左边填充字符串 expr1，直到字符串长度为 n。 • 如果 expr1 的长度大于 n，则从右边截断 expr1，使其长度为 n； • 如果 n 小于或等于 0，返回空字符串 ''。 • 如果 expr1 或 expr2 为 NULL，结果为 NULL。 例如，lpad('abc', 5, '0') 返回 '00abc'；lpad('abc', 2, '0') 返回 'ab'。
ltrim(expr1)	ltrim（expr1，''）的简写，表示删除 expr1 左侧的空格。 例如，ltrim(' abc') 返回 'abc'。
ltrim(expr1, expr2)	从字符串 expr1 的左侧移除字符串 expr2 中出现的字符，直到第一个不在 expr2 中的字符为止。 例如，ltrim('<<<<<<>>>>test<<', '><') 返回 'test<<'。
right(expr, n)	从字符串 expr 中提取最右边的 n 个字符。 • 如果 n 大于 expr 的长度，返回整个 expr。 • 如果 n 为负值，提取的字符串长度为 length(expr) - abs(n)。 • 如果 n 为 0，返回空字符串 ''。 • 如果 expr 为 NULL，返回 NULL。 例如，right('abcdef', 3) 返回 'def'；right('abcdef', 10) 返回

	'abcdef'。
rpad(expr1, n, expr2)	使用字符串 expr2 从右边填充字符串 expr1，直到字符串长度为 n。 • 如果 expr1 的长度大于 n，则从右边截断 expr1，使其长度为 n； • 如果 n 小于或等于 0，返回空字符串 ''。 • 如果 expr1 或 expr2 为 NULL，结果为 NULL。 例如，rpad('abc', 5, '0') 返回 'abc00'；rpad('abc', 2, '0') 返回 'ab'。
rtrim(expr1)	rtrim(expr1, ' ') 的简写，表示删除字符串 expr1 右侧的空格。 例如，rtrim('abc ') 返回 'abc'。
rtrim(expr1, expr2)	从字符串 expr1 的右侧移除字符串 expr2 中的任意字符，直到第一个不在 expr2 中的字符为止。 例如，rtrim('abc---', '-') 返回 'abc'。
substr(expr, start)	从字符串 expr 的 start 位置开始，提取字符串直到末尾。 • 如果 start 为正值，表示从左边数第 start 个字符，提取字符串直到末尾。 • 如果 start 为负值，表示从右边数第 abs(start) 个字符，提取字符串直到末尾。 例如，substr('abcdef', 2) 返回 'bcdef'；substr('abcdef', -3) 返回 'def'。

substr(expr, start, length)	从字符串 expr 的 start 位置开始，提取长度为 length 的子串，其中：
	• start:
	  ◦ 如果为正值 n，表示从左边数第 n 个字符开始。
	  ◦ 如果为负值 n，表示从右边数第 abs(n) 个字符开始（例如 -1 表示最后一个字符）。
	• length:

	◦ 如果为正值 n，表示从 start 开始向右提取长度为 n 的子串。如果 length 大于从 start 到字符串末尾的长度，则返回从 start 到末尾的所有字符。 ◦ 如果为负值 n，表示从 start 开始向左提取长度为 abs(n) 的子串。如果 length 大于从 start 到字符串开始的长度，则返回从 start 到字符串开始的所有字符。 • 如果 start 和 length 界定的范围不包含字符，则返回空字符串 '' 例如，substr('abcdef', 2, 3) 返回 'bcd'；substr('abcdef', -4, 2) 返回 'cd'。
substring(expr, start)	从字符串 expr 的 start 位置开始，提取字符串直到末尾。 • 如果 start 为正值，表示从左边数第 start 个字符，提取字符串直到末尾。 • 如果 start 为负值，表示从右边数第 abs(start) 个字符，提取字符串直到末尾。 例如，substring('abcdef', 2) 返回 'bcdef'；substring('abcdef', -3) 返回 'def'。
substring(expr, start, length)	从字符串 expr 的 start 位置开始，提取长度为 length 的子串，其中： • start: ◦ 如果为正值 n，表示从左边数第 n 个字符开始。 ◦ 如果为负值 n，表示从右边数第 abs(n) 个字符开始（例如 -1 表示最后一个字符）。 • length: ◦ 如果为正值 n，表示从 start 开始向右提取长度为 n 的子串。如果 length 大于从 start 到字符串末尾的长度，则返回从 start 到末尾的所有字符。 ◦ 如果为负值 n，表示从 start 开始向左提取长度为

	abs(n) 的子串。如果 length 大于从 start 到字符串开始的长度，则返回从 start 到字符串开始的所有字符。 • 如果 start 和 length 界定的范围不包含字符，则返回空字符串 '' 例如，substring('abcdef', 2, 3) 返回 'bcd'；substring('abcdef', -4, 2) 返回 'cd'。
upper(expr)	将字符串 expr 转换为大写字母。 例如，upper('hello') 返回 'HELLO'。

7.2.6 分组窗口函数

分组窗口函数用于对查询数据集按定义窗口进行切分，并在每个窗口内执行分组聚合计算。每个窗口会生成一行聚合结果。

函数	描述
state_window(expr)	状态窗口函数，按时间序列升序，根据状态值变化来划分窗口。具有相同状态的连续数据会归入同一窗口。当状态发生变化时，当前窗口结束并开启新窗口。支持使用 CASE-WHEN 表达式定义状态转换条件，实现复杂状态切换逻辑。 语法格式为 SELECT select_list FROM ts_table [WHERE condition] GROUP BY col_list, state_window(expr); 注意： 1. 仅适用于时序表的单表查询，不支持嵌套子查询。 2. 必须与 GROUP BY 配合使用，GROUP BY 子句中可包含主标签，但 state_window(expr) 必须是 GROUP BY 子句的最后一项。 3. expr 为划分窗口的 SQL 表达式，不能使用聚合函数，结果类型必须是整数、布尔值或字符串。 4. 查询语句中不能包含其他窗口函数。 5. 当存在重复时间戳时，无论是否按主标签分组，对应行的值

	将随机返回。

7.3 操作符

下表按优先级从高到低的顺序列出了 KaiwuDB Lite 支持的所有运算符。相同优先级的运算符采用左关联规则，即 从左向右依次计算。

操作符	描述	优先级
.	成员字段访问运算符，用于指定数据的完整路径，通过连接模式、表和字段名，明确告诉数据库系统要查询哪个模式的哪个表的特定字段。 语法格式为<schema>.<table>.<col>，例如 memory.t1.a，表示从 memory 模式中的 t1 表获取 a 字段的数据。	1
::	类型转换运算符，将字段转换为指定类型。 语法格式为<expr>::<type_name>，例如 5::FLOAT，表示将整数 5 转换为浮点数。 转换规则： • 整数与浮点数的转换： ◦ 整数转换为浮点数不会丢失精度 ◦ 浮点数转换为整数可能会丢失精度 • 整数类型转换： ◦ 从小精度整数（如 SMALLINT）转换为大精度整数（如 BIGINT）通常安全 ◦ 从大精度转换为小精度可能导致数据溢出 • 浮点数类型转换： ◦ 从小精度浮点数转换为大精度浮点数通常安全 ◦ 从大精度转换为小精度可能导致数据溢出 • 字符串和数字转换： ◦ 当字符串表示有效数字时，可以转换为相应的数字类型；	2

	如果字符串包含非数字字符，则转换会失败（例如，'123a' 转换为 INT32 会报错） ◦ 数字类型可以转换为字符串类型 • 字符串和时间戳类型转换： ◦ 符合时间格式的字符串可转换为相应的类型，格式不符时转换失败 ◦ 时间戳类型可以转换为字符串类型 • 布尔类型转换： ◦ 数字 0 转换为 FALSE，其他数字转换为 TRUE ◦ 字符串 '0' 转换为 FALSE，'1' 转换为 TRUE，其他字符串转换失败 ◦ 布尔类型可以转换为数字类型，也可以转换为字符串类型。	
-	取负运算符，对数据进行取负操作。 语法格式为-<num>，例如，-float_col 表示将 float_col 列的值取负。如果 float_col 为 5.0，则结果为 -5.0。 支持整数类型，浮点类型和时间间隔类型（INTERVAL）。	3
~	按位取反运算符，对整数类型数据进行按位取反。 语法格式为~<num>，例如，~(2::SMALLINT) 表示对 SMALLINT 类型的 2 进行按位取反。 支持 SMALLINT、INTEGER 和 BIGINT。	3
^	求幂运算符，对数字进行幂运算。 语法格式为<base_num>^<power_num>，例如，3^2 表示计算 3 的 2 次方，结果为 9.0。 支持整数类型和浮点数类型。	4
*	乘法运算符，用于乘法运算。 语法格式为<multiplicand>*<multiplier>，例如，3*2 的计算结果为 6。	5

	支持整数类型和浮点数类型。	
/	浮点除法运算符，用于浮点除法。 语法格式为<divided_num>/<divisor_num>。 支持整数和浮点数类型，返回浮点数类型。例如，6/2 的计算结果为 3.0（浮点数）。	5
//	整除运算符，执行整除操作。 语法格式为<divided_num>//<divisor_num>。 支持整数和浮点数类型。如果两个数值均为整数，结果会向下取整；如果有浮点数参与，结果为浮点数。例如，7//2 的计算结果为 3。	5
%	求余运算符，返回除法的余数。 语法格式为<divided_num>%<divisor_num> 。 支持整数和浮点数类型。如果两个数值均为整数，结果为整数；如果有浮点数参与，结果为浮点数。例如，7%2 的计算结果为 1。	5
+	加法运算符，用于求和。 语法格式为<addend1>+<addend2>。 支持以下数据类型相加： • 数值类型（整数类型和浮点数类型）相加 • 时间戳类型和 INTERVAL 类型相加	6
-	减法运算符，用于减法操作。 语法格式为<minuend>-<subtrahend>。 支持以下数据类型相减： • 数值类型（整数类型和浮点数类型）相减 • 时间戳类型和 INTERVAL 类型相减 • 时间戳类型和时间戳类型相减	6
<<	左移运算符，将二进制数字向左移动指定位置。 语法格式为<num1> << <num2>，例如，2<<3 将 2 的二进制表示向左移 3 位，结果为 16。	7

	两个操作数值都必须为非负整数。	
>>	右移运算符，将二进制数字向右移动指定位置。 语法格式为<num1> >> <num2>，例如，16>>2 将 16 的二进制表示向右移 2 位，结果为 4。 两个操作数值都必须为非负整数。	7
&	按位与操作符，对二进制表示的数字进行按位与操作。 语法格式为<num1> & <num2>，例如，5&3 结果为 1。 仅支持整数类型。	8
|	按位或操作符，对二进制表示的数字进行按位或操作。 语法格式为<num1> | <num2>，例如，5|3 结果为 7。 仅支持整数类型。	9
||	字符串拼接运算符，用于拼接字符串。 语法格式为<string1> || <string2>，例如，'Hello '||'World' 结果为 'Hello World'。 支持以下数据类型拼接： • 字符串与字符串拼接 • 字符串与数字拼接 • 数字与数字的拼接 数字会自动转换为字符串。	10
[NOT] IN	集合操作符，检查左侧表达式是否存在于右侧集合中。 语法格式为 <expr> [NOT] IN (<value1>, <value2>, ...)，例如，SELECT * FROM employees WHERE department_id IN (1, 2, 3) 查询部门 ID 为 1、2 或 3 的员工记录。 使用 NOT IN 时，查询不在指定集合中的记录。 右侧集合支持的类型包括列表和返回单列的子查询。 如果左侧表达式不在右侧集合中且右侧集合包含 NULL 值，则返回 NULL。 支持时间戳类型、数值类型、布尔类型和字符类型。	11

[NOT] LIKE	模式匹配运算符，检查字符串是否匹配指定模式。 语法格式为 <string> [NOT] LIKE <pattern> [ESCAPE <escape_character>]。 如果模式不包含 % 或 _，则等同于精确匹配（=），_表示匹配任意单个字符，%表示匹配任意长度（包括零个字符）的字符串。例如，SELECT * FROM customers WHERE name LIKE 'John%' 查询名字以 "John" 开头的客户记录。 LIKE 匹配始终应用于整个字符串，要在字符串任意位置匹配，必须在模式两端添加 %。 如果需要搜索包含 % 或 * 的实际字符时，使用 ESCAPE 子句，同时使用转义字符$，例如 SELECT 'a%c' LIKE 'a$%c' ESCAPE '$' 。 使用 NOT LIKE 时，查询不匹配指定模式的记录。 只支持字符类型。	11
[NOT] ILIKE	忽略大小写的模式匹配运算符，检查字符串是否匹配指定模式。 语法格式为 <string> [NOT] ILIKE <pattern> [ESCAPE <escape_character>]。 如果模式不包含 % 或 _，则等同于精确匹配（=），_表示匹配任意单个字符，%表示匹配任意长度（包括零个字符）的字符串。 ILIKE 匹配始终应用于整个字符串，要在字符串任意位置匹配，必须在模式两端添加 %。例如，SELECT * FROM products WHERE description ILIKE '%computer%' 查询描述中包含 "computer"（忽略大小写）的产品记录。 如果需要搜索包含 % 或 * 的实际字符时，使用 ESCAPE 子句，同时使用转义字符$，例如 SELECT 'a%c' ILIKE 'a$%c' ESCAPE '$' 。 使用 NOT ILIKE 时，查询不匹配指定模式的记录。 只支持字符类型。	11
~~	正则表达式匹配运算符，检查字符串是否匹配正则表达式。	11

	语法格式为 <string>~~<pattern>，例如，SELECT * FROM users WHERE email~~'.*@gmail\.com' 表示查询所有使用 Gmail 邮箱的用户。 只支持字符类型。	
!~~	正则表达式不匹配运算符，检查字符串是否匹配正则表达式。 语法格式为 <string>!~~<pattern>，例如，SELECT * FROM users WHERE email!~~'.*@gmail\.com' 表示查询所有非 Gmail 邮箱的用户。 只支持字符类型。	11
~~*	忽略大小写的正则表达式匹配，检查字符串是否匹配正则表达式。 语法格式为<string>~~*<pattern> 例如，SELECT * FROM articles WHERE title~~*'database' 表示查询标题中包含 "database" （不区分大小写）的文章。 只支持字符类型。	11
!~~*	忽略大小写的正则表达式不匹配，检查字符串是否不匹配正则表达式。 语法格式为<string>!~~*<pattern> 例如，SELECT * FROM documents WHERE confidential_type!~~*'confidential' 查询保密等级不是 "confidential" 的文档。 只支持字符类型。	11
=	等于，用于比较两个值是否相等。如果相等返回 true，如果不相等返回 false。 语法格式为 <value1> = <value2> ，例如，SELECT * FROM orders WHERE status = 'completed';表示查询所有状态为 completed 的订单记录。 支持数值类型、布尔类型、字符类型和时间戳类型。	12
!=	不等于，用于比较两个值是否不等。如果不等返回 true，如果相等	12

	返回 false。 语法格式为 <value1> != <value2> ，例如，SELECT * FROM inventory WHERE quantity != 0;表示查询所有数量不为 0 的库存记录。 支持数值类型、布尔类型、字符类型和时间戳类型。	
<	小于，用于比较左边数据是否小于右边数据。如果小于返回 true，如果不小于返回 false。 语法格式为 <value1> < <value2> ，例如，SELECT * FROM products WHERE price < 50.00; 表示查询所有价格小于 50.00 的产品记录。 支持数值类型、布尔类型、字符类型和时间戳类型。	12
>	大于，用于比较左边数据是否大于右边数据。如果大于返回 true，如果不大于返回 false。 语法格式为 <value1> > <value2> ，例如，SELECT * FROM employees WHERE salary > 5000;表示查询所有薪资大于 5000 的员工记录。 支持数值类型、布尔类型、字符类型和时间戳类型。	12
<=	小于或等于，用于比较左边数据是否小于等于右边数据。如果小于等于返回 true，如果不小于等于返回 false。 语法格式为 <value1> <= <value2> ，例如，SELECT * FROM events WHERE date <= '2024-12-31 00:00:00.000';表示查询所有日期在 2024 年 12 月 31 日或之前的事件记录。 支持数值类型、布尔类型、字符类型和时间戳类型。	12
>=	大于或等于，用于比较左边数据是否大于等于右边数据。如果大于等于返回 true，如果不大于等于返回 false。 语法格式为 <value1> >= <value2> ，例如，SELECT * FROM accounts WHERE balance >= 1000;表示查询所有余额大于或等于 1000 的账户记录。	12

	支持数值类型、布尔类型、字符类型和时间戳类型。	
<>	不等于，等同于 !=，用于比较左边数据是否不等于右边数据。如果不等于返回 true，如果等于返回 false。 语法格式为 <value1> <> <value2> ，例如，SELECT * FROM tasks WHERE status <> 'completed';表示查询返回所有状态不是 completed 的任务记录。 支持数值类型、布尔类型、字符类型和时间戳类型。	12

7.4 DDL 语句

DDL 语句用于创建和管理数据库对象。

7.4.1 SCHEMA

7.4.1.1 CREATE SCHEMA

CREATE SCHEMA 语句用于创建新模式。

7.4.1.1.1 语法格式

```
SQL
CREATE SCHEMA <schema_name>;
```

7.4.1.1.2 参数说明

- schema_name：新模式的名称。不支持使用 #、@ 等特殊符号。

7.4.1.1.3 语法示例

示例：创建名为 benchmark 的模式。

```
SQL
CREATE SCHEMA benchmark;
```

7.4.1.2 SET search_path

SET search_path 语句用于设置当前模式。

7.4.1.2.1 语法格式

```
SQL
SET search_path = <schema_name>;
```

7.4.1.2.2 参数说明

- schema_name：目前模式名称。

7.4.1.2.3 语法示例

示例：将 benchmark 模式设置为当前模式。

```
SQL
SET search_path = benchmark;
```

7.4.2 TABLE

7.4.2.1 CREATE TS TABLE/CREATE TABLE

KaiwuDB Lite 针对时序数据提供了两种存储方式：

存储方式	优势	缺点	适用场景
单表存储：即一张时序表同时存储 Metric 和 Tag 信息	• 查询无需显式关联操作 • 使用简单	• 建表后不支持加列/减列操作 • 存储效率相对较低	表结构相对固定的场景
双表存储：Metric 和 Tag 分别存储在 Metrics 表和 Tags 表中	• Metrics 数据存储高效，压缩率高	• 查询需要显式关联操作 • 使用相对复杂	表结构需要灵活变更的场景

	• 支持加/减列		

7.4.2.1.1 语法格式

- 单表存储（时序表）

> 说明：
> 使用以下 SQL 语句创建的表，用户无法向目标表中添加列或者从目标表中删除列。

```
SQL
CREATE TABLE <table_name> (<col_list> [col_constraint]>) [TAGS |
ATTRIBUTES] (<all_tag_list>) [PRIMARY [TAGS | ATTRIBUTES ]
(<primary_tag_list>)];
```

- 双表存储（Metrics 表和 Tags 表）

```
SQL
CREATE [TS] TABLE <table_name> (<col_list> [col_constraint]>) [, PRIMARY
KEY(<column_name>[, <column_name>])][, FOREIGN KEY(<column_name>[,
<column_name>]) REFERENCES <foreign_table>(<col_name>[,
<col_name>])]);
```

7.4.2.1.2 参数说明

- TS：可选关键字。当使用 TS 关键字时，创建的表为 Metrics 表，用于存储时序数据。当未使用 TS 关键字时，创建的表为 Tags 表，用于存储相对静态的标签信息。
- table_name：待创建的表的名称，支持采用 <schema_name>.<table_name> 的格式。如未指定模式，则在当前模式下建表。
- col_list：数据列列表，需添加两个以上的列定义，各列之间使用逗号（,）隔开。列定义包括列名和数据类型，支持指定 NOT NULL，默认为 NULL。

> 说明

> - 使用单表建表语句创建的时序表和使用双表建表语句创建的 Metrics 表的第一列的数据类型必须是 TIMESTAMP 且为非空。
> - 对于 Metrics 表的第一列（时间戳列），不支持 DEFAULT 表达式。

- col_constraint：列的约束，支持为一个或多个列添加约束。为目标列添加外键约束时，确保外键引用的主键存在于被引用的表中。
- column_name：待添加约束的列的名称。
- foreign_table：外键引用的表的名称。
- col_name：外键引用的表的主键的名称。
- all_tag_list：标签列表，支持添加一个或多个标签，各标签之间使用逗号（,）隔开。标签定义包括标签名和数据类型，支持指定 NOT NULL，默认为 NULL。不支持 TIMESTAMP 数据类型。
- primary_tag_list：可选参数，主标签列表，支持添加一个或多个主标签，各主标签之间使用逗号（,）隔开。主标签必须包含在标签列表内且指定为 NOT NULL，不支持浮点类型和 TIMESTAMP 数据类型。

7.4.2.1.3 语法示例

示例 1：双表模式下，创建 Metrics 表。

```
SQL
CREATE TS TABLE power (k_timestamp timestamp not null, id INTEGER, voltage INTEGER);
```

示例 2：双表模式下，创建 Tags 表。

```
SQL
CREATE TABLE students (name VARCHAR(30), age INTEGER);
```

示例 3：单表模式下，创建时序表。

以下示例在 benchmark 模式下创建名为 cpu 的时序表。

> 说明
>
> 使用以下 SQL 语句创建的表，用户无法向目标表中添加列或者从目标表中删除列。

```
SQL
CREATE TABLE benchmark.cpu (
   k_timestamp timestamp not null,
   usage_user bigint not null,
   usage_system bigint not null,
   usage_idle bigint not null,
   usage_nice bigint not null,
   usage_iowait bigint not null,
   usage_irq bigint not null,
   usage_softirq bigint not null,
   usage_steal bigint not null,
   usage_guest bigint not null,
   usage_guest_nice bigint not null，
) TAGS (
   hostname varchar(30) not null,
   region varchar(30),
   datacenter varchar(30),
   rack varchar(30),
   os varchar(30),
   arch varchar(30),
   team varchar(30),
   service varchar(30),
   service_version varchar(30),
   service_environment varchar(30)
) PRIMARY TAGS (hostname);
```

7.4.2.2 ALTER TABLE

ALTER TABLE 语句用于修改以下表信息：

- 修改表名。
- 添加列、修改列名、列的数据类型或宽度、设置列的默认值和 NOT NULL 约束、删除列的默认值和 NOT NULL 约束。
- 添加普通标签、修改普通标签的名称、数据类型或宽度、删除普通标签。

7.4.2.2.1 语法格式

```
SQL
ALTER TABLE [IF EXISTS] <table_name>
[ADD [COLUMN] [IF NOT EXISTS] <colunm_name> <data_type> [DEFAULT
<default_expr>]
|ALTER [COLUMN] <colunm_name> [[SET DATA] TYPE <new_type> [(length)] | SET
[DEFAULT <default_expr> | NOT NULL] | DROP [DEFAULT | NOT NULL]]
|DROP [COLUMN] <colunm_name>
|RENAME TO <new_table_name>
|RENAME [COLUMN] <old_name> TO <new_name>];
```

7.4.2.2.2 参数说明

- IF EXISTS：可选关键字。当使用 IF EXISTS 关键字时，如果目标表存在，系统修改目标表。如果目标表不存在，系统修改目标表失败，但不会报错。当未使用 IF EXISTS 关键字时，如果目标表存在，系统修改目标表。如果目标表不存在，系统报错，提示目标表不存在。
- table_name：待修改的表的名称 。
- ADD COLUMN: 向目标表中添加指定类型的数据列和普通标签列。
  - COLUMN：可选关键字，是否使用不影响添加列。

- IF NOT EXISTS：可选关键字，当使用 IF NOT EXISTS 关键字时，如果列名不存在，系统创建列。如果列名存在，系统创建列失败，但不会报错。当未使用 IF NOT EXISTS 关键字时，如果列名不存在，系统创建列。如果列名存在，系统报错，提示列名已存在。
- colunm_name：待添加列的名称。
- data_type：待添加列的数据类型。有关 KaiwuDB Lite 支持的数据类型，参见数据类型。
- DEFAULT <default_expr>：可选关键字，系统写入表数据时写入指定的默认值，从而不需要显式定义该列的值。对于非时间类型的数据列，默认值只能是常量。如果默认值类型与列类型不匹配，设置默认值时，系统报错。如未指定默认值，则默认填入 NULL。对于 Metrics 表，不支持为第一列（时间戳列）设置默认值。

- ALTER COLUMN: 修改数据列和普通标签列的数据类型和宽度，设置或删除数据列的默认值。
  - COLUMN：可选关键字，如未使用，是否使用不影响修改列。
  - colunm_name：待修改的列的名称。
  - SET DATA：可选关键字，是否使用不影响修改列的数据类型。
  - new_type：拟修改的数据类型。对于 Metrics 表，不支持修改第一列（时间戳列）的数据类型。
  - SET DEFAULT <default_expr>：系统写入表数据时写入指定的默认值，从而不需要显式定义该列的值。对于非时间类型的数据列，默认值只能是常量。如果默认值类型与列类型不匹配，设置默认值时，系统报错。支持默认值设置为 NULL。对于 Metrics 表，不支持为第一列（时间戳列）设置默认值。
  - SET NOT NULL：设置列的 NOT NULL 约束。
  - DROP DEFAULT：删除已定义的列的默认值，删除后将不再写入默认值。

- DROP NOT NULL：删除列的 NOT NULL 约束。对于 Metrics 表，不支持删除第一列（时间戳列）的 NOT NULL 约束。

- DROP COLUMN: 从目标表中删除数据列和普通标签列。
  - COLUMN：可选关键字，是否使用不影响删除列。
  - IF EXISTS：可选关键字，当使用 IF EXISTS 关键字时，如果列名存在，系统删除列。如果列名不存在，系统删除列失败，但不会报错。当未使用 IF EXISTS 关键字时，如果列名存在，系统删除列。如果列名不存在，系统报错，提示列名不存在。
  - column_name：待删除列的名称。对于 Metrics 表，不支持删除第一列（时间戳列）。

- RENAME
  - RENAME TO：修改表的名称。
  - RENAME COLUMN：修改数据列和普通标签列的名称。

7.4.2.2.3 语法示例

以下示例假设已创建 ts_table 表并写入相关数据。

```
SQL
-- 添加列。
ALTER TABLE ts_table ADD COLUMN c3 INT;


-- 添加列并设置列的默认值。
ALTER TABLE ts_table ADD COLUMN c6 VARCHAR(50) DEFAULT 'aaa';


-- 修改列的名称。


ALTER TABLE ts_table RENAME COLUMN c2 TO c4;
```

```
-- 修改列的数据类型。

ALTER TABLE ts_table ALTER COLUMN c3 TYPE INT8;

-- 设置列的默认值。

ALTER TABLE ts_table ALTER COLUMN c4 SET DEFAULT '789';

-- 删除列的默认值。

ALTER TABLE ts_table ALTER COLUMN c4 DROP DEFAULT;

-- 设置列的 NOT NULL 约束。

ALTER TABLE ts_table ALTER COLUMN c5 SET NOT NULL;

-- 删除列的 NOT NULL 约束。

ALTER TABLE ts_table ALTER COLUMN c5 DROP NOT NULL;

-- 删除列。
ALTER TABLE ts_table DROP c4;

-- 修改表的名称。

ALTER TABLE ts_table RENAME TO tstable;
```

7.4.2.3 DROP TABLE

DROP TABLE 语句用于删除表。

7.4.2.3.1 语法格式

```
SQL
DROP TABLE [IF EXISTS] <table_name> [CASCADE | RESTRICT];
```

7.4.2.3.2 参数说明

- IF EXISTS：可选关键字。当使用 IF EXISTS 关键字时，如果目标表存在，系统删除目标表。如果目标表不存在，系统删除目标表失败，但不会报错。当未使用 IF EXISTS 关键字时，如果目标表存在，系统删除目标表。如果目标表不存在，系统报错，提示目标表不存在。
- table_name：待删除的表的名称，格式为 <schema_name>.<table_name> 。如未指定模式名，则删除当前模式下的表。
- CASECADE：可选关键字。删除目标表及其关联对象。CASCADE 不会列出待删除的关联对象，应谨慎使用。
- RESTRICT：默认设置，可选关键字。如果其他对象依赖目标表，则无法删除目标表。

7.4.2.3.3 语法示例

示例：删除 benchmark 模式下名为 cpu 的表。

```
SQL
DROP TABLE benchmark.cpu;
```

7.4.3 COLUMN

7.4.3.1 ADD COLUMN

ALTER TABLE ... ADD COLUMN 语句用于向目标表中添加指定类型的数据列和普通标签

列。

7.4.3.1.1 语法格式

```
SQL
ALTER TABLE <table_name> ADD [COLUMN] [IF NOT EXISTS] <colunm_name>
<data_type> [ DEFAULT <default_expr>];
```

7.4.3.1.2 参数说明

- table_name：待修改的表的名称 。
- COLUMN：可选关键字，是否使用不影响添加列。
- IF NOT EXISTS：可选关键字，当使用 IF NOT EXISTS 关键字时，如果列名不存在，系统创建列。如果列名存在，系统创建列失败，但不会报错。当未使用 IF NOT EXISTS 关键字时，如果列名不存在，系统创建列。如果列名存在，系统报错，提示列名已存在。
- colunm_name：待添加列的名称。
- data_type：待添加列的数据类型。有关 KaiwuDB Lite 支持的数据类型，参见数据类型。
- DEFAULT <default_expr>：可选关键字，系统写入表数据时写入指定的默认值，从而不需要显式定义该列的值。对于非时间类型的数据列，默认值只能是常量。对于时间类型的列（TIMESTAMPTZ 或 TIMESTAMP），默认值可以是常量，也可以是 now() 函数。如果默认值类型与列类型不匹配，设置默认值时，系统报错。如未指定默认值，则默认填入 NULL。

7.4.3.1.3 语法示例

示例 1：为 ts_table 表增加一个名为 c3 的列。

```
SQL
ALTER TABLE ts_table ADD COLUMN c3 INT;
```

示例 2：为 ts_table 表增加一个名为 c4 的列并设置该列的默认值为 aaa。

```sql
ALTER TABLE ts_table ADD COLUMN c4 VARCHAR(50) DEFAULT 'aaa';
```

7.4.3.2 ALTER COLUMN

ALTER TABLE ... ALTER COLUMN 语句用于修改数据列和普通标签列的数据类型和宽度、设置或者删除数据列的默认值、设置或删除数据列的 NOT NULL 约束。

> 说明：
> - 修改列时，不支持修改 Metrics 表第一列（时间戳列）的数据类型。
> - 修改列时，不支持删除 Metrics 表第一列（时间戳列）的 NOT NULL 约束。
> - 修改列时，不支持为 Metrics 表的第一列（时间戳列）设置默认值。

7.4.3.2.1 语法格式

- 修改列的数据类型

```
SQL
ALTER TABLE <table_name> ALTER [COLUMN] <colunm_name> [SET DATA]
TYPE <new_type> [(length)];
```

- 设置数据列的默认值或 NOT NULL 约束

```
SQL
ALTER TABLE <table_name> ALTER [COLUMN] <colunm_name> SET [DEFAULT
<default_expr> | NOT NULL];
```

- 删除数据列的默认值或 NOT NULL 约束

```
SQL
ALTER TABLE <table_name> ALTER [COLUMN] <colunm_name> DROP
[DEFAULT | NOT NULL];
```

7.4.3.2.2 参数说明

- table_name：待修改的表的名称 。
- COLUMN：可选关键字，是否使用不影响修改列。
- colunm_name：待修改的列的名称。
- SET DATA：可选关键字，是否使用不影响修改列的数据类型。
- new_type：拟修改的数据类型。对于 Metrics 表，不支持修改第一列（时间戳列）的 数据类型。
- SET DEFAULT <default_expr>：系统写入表数据时写入指定的默认值，从而不需要显式定义该列的值。对于非时间类型的数据列，默认值只能是常量。如果默认值类型与列类型不匹配，设置默认值时，系统报错。支持默认值设置为 NULL。对于 Metrics 表，不支持为第一列（时间戳列）设置默认值。
- SET NOT NULL：设置列的 NOT NULL 约束。
- DROP DEFAULT：删除已定义的列的默认值，删除后将不再写入默认值。
- DROP NOT NULL：删除列的 NOT NULL 约束。对于 Metrics 表，不支持删除第一列（时间戳列）的 NOT NULL 约束。

7.4.3.2.3 语法示例

以下示例假设已创建 ts_table 表并写入相关数据。

示例 1：修改 ts_table 表中 c3 列的数据类型。

```
SQL
ALTER TABLE ts_table ALTER COLUMN c3 TYPE INT8;
```

示例 2：为 ts_table 表中 c4 列设置默认值 789。

```
SQL
ALTER TABLE ts_table ALTER COLUMN c4 SET DEFAULT '789';
```

示例 3：删除 ts_table 表中 c4 列的默认值。

```
SQL
ALTER TABLE ts_table ALTER COLUMN c4 DROP DEFAULT;
```

示例 4：为 ts_table 表中 c5 列设置 NOT NULL 约束。

```
SQL
ALTER TABLE ts_table ALTER COLUMN c5 SET NOT NULL;
```

示例 5：删除 ts_table 表中 c5 列的 NOT NULL 约束。

```
SQL
ALTER TABLE ts_table ALTER COLUMN c5 DROP NOT NULL;
```

7.4.3.3 RENAME COLUMN

ALTER TABLE ... RENAME COLUMN 语句用于修改数据列和普通标签列的名称。

7.4.3.3.1 语法格式

```
SQL
ALTER TABLE <table_name> RENAME [COLUMN] <old_name> TO <new_name>;
```

7.4.3.3.2 参数说明

- table_name：待修改列所在表的名称 。
- COLUMN：可选关键字，是否使用不影响重命名列。
- old_name：当前列名。
- new_name：拟修改的列名。新增列名不得与目标表的当前列名和标签名重复。

7.4.3.3.3 语法示例

以下示例将 ts_table 表的 c2 列重命名为 c4。

```
SQL
ALTER TABLE ts_table RENAME COLUMN c2 TO c4;
```

7.4.3.4 DROP COLUMN

ALTER TABLE ... DROP COLUMN 语句用于从目标表中删除数据列和普通标签列。

> 说明
>
> - 删除列时，系统检查目标列是否已创建索引。如果目标列已创建索引，则删除列失败，系统报错。用户需要首先删除相关索引，然后再删除目标列。有关删除索引的详细信息，参见 DROP INDEX。
> - 删除列时，不支持删除 Metrics 表的第一列（时间戳列）。

7.4.3.4.1 语法格式

```
SQL
ALTER TABLE <table_name> DROP [COLUMN] [IF EXISTS] <column_name>;
```

7.4.3.4.2 参数说明

- table_name：待删除列所在表的名称 。
- COLUMN：可选关键字，是否使用不影响删除列。
- IF EXISTS：可选关键字，当使用 IF EXISTS 关键字时，如果列名存在，系统删除列。如果列名不存在，系统删除列失败，但不会报错。当未使用 IF EXISTS 关键字时，如果列名存在，系统删除列。如果列名不存在，系统报错，提示列名不存在。
- column_name：待删除列的名称。

7.4.3.4.3 语法示例

以下示例删除 ts_table 表的 c4 列。

```
SQL
ALTER TABLE ts_table DROP c4;
```

7.4.4 COMMENT

7.4.4.1 COMMENT ON

COMMENT ON 语句用于为指定表、数据列、普通标签列、主标签列添加注释。

7.4.4.1.1 语法格式

```
SQL
COMMENT ON [TABLE <table_name> | COLUMN <column_name> ] IS
<comment_text>;
```

7.4.4.1.2 参数说明

- table_name：表的名称。
- column_name：数据列、普通标签列、主标签列的名称。支持采用 <table_name>.<column_name> 的格式，指定其他表中的列。
- comment_text：注释内容。当目标对象已有注释信息，如果新的注释消息不为空，系统将更新目标对象的原有注释信息。如果新的注释消息为空，系统将删除目标对象的原有注释信息。

7.4.4.1.3 语法示例

示例 1：为表添加注释。

```
sql
COMMENT ON TABLE power IS 'power for all devices';
```

示例 2：为列添加注释。

```
sql
COMMENT ON COLUMN power.ts IS 'auto-generated';
```

7.4.5 INDEX

7.4.5.1 CREATE INDEX

CREATE INDEX 语句用于为指定表的列创建索引。

7.4.5.1.1 语法格式

```
SQL
CREATE [UNIQUE] INDEX [IF NOT EXISTS] <index_name> ON <table_name> [USING <index_type>] {(<col_list>) | ((<expr>))} [WITH (<option_list>)];
```

7.4.5.1.2 参数说明

- UNIQUE：可选关键字。为索引列应用唯一性约束，系统在创建索引时检查重复值。在表级别应用唯一性约束，系统在写入或更新表数据时会检查重复值。
- IF NOT EXISTS：可选关键字，当使用 IF NOT EXISTS 关键字时，如果索引名不存在，系统创建索引。如果索引名存在，系统创建索引失败，但不会报错。当未使用 IF NOT EXISTS 关键字时，如果索引名不存在，系统创建索引。如果索引名存在，系统报错，提示索引名已存在。
- index_name：待创建索引的名称。该名称在数据库中必须唯一。
- table_name：待创建索引的目标表的名称。
- index_type：可选项。支持 Min-Max 索引和 ART 索引（Adaptive Radix Tree Index，自适应基数树索引）。
- col_list：待创建索引的目标列的名称。支持指定多个列，各列之间使用逗号（,）隔开。
- expr：基于表格单列或多列的表达式。通常情况下，表达式必须使用圆括号（()）包围。如果表达式采用函数调用的形式，则可以省略圆括号。
- option_list：可选项。索引选项，支持 BOOL 或键值对的形式，例如 is_cool = true 或者 my_option = 2。

7.4.5.1.3 语法示例

- 创建单列索引。

  以下示例为 re_users 表的 city 列创建一个名为 city_idx 的索引。

  ```
  SQL
  CREATE INDEX city_idx ON re_users (city);
  ```

- 创建多列索引。

  以下示例为 re_users 表的 city 和 name 列创建一个名为 cn_idx 的索引。

  ```
  SQL
  CREATE INDEX cn_idx ON re_users (city, name);
  ```

- 创建唯一索引，不允许其列中包含重复值。

  以下示例为 re_users 表的 credit_card 和 dl 列创建一个名为 cd_idx 唯一索引。

  ```
  SQL
  CREATE UNIQUE INDEX cd_idx ON re_users (credit_card, dl);
  ```

- 使用表达式创建多列索引。

  以下示例使用求和表达式为 re_users 表的 c1 和 c5 列创建一个名为 i_index 的索引。

  ```
  SQL
  CREATE INDEX i_index ON re_users ((c1 + c5));
  ```

7.4.5.2 ALTER INDEX

ALTER INDEX... RENAME TO 语句用于更改索引的名称。

7.4.5.2.1 语法格式

```
SQL
```

```
ALTER INDEX <index_name> RENAME TO <new_name>;
```

7.4.5.2.2 参数说明

- index_name：当前索引的名称。
- new_name：索引的新名称。该名称在数据库中必须唯一。

7.4.5.2.3 语法示例

以下示例将 ts_table 表的 sensor_type_index 索引重命名为 sensor_index。

```
SQL
ALTER INDEX sensor_type_index RENAME TO sensor_index;
```

7.4.5.3 DROP INDEX

DROP INDEX 语句用于删除表的索引。

7.4.5.3.1 语法格式

```
SQL
DROP INDEX [IF EXISTS] <index_name>;
```

7.4.5.3.2 参数说明

- IF EXISTS：可选关键字，当使用 IF EXISTS 关键字时，如果索引名存在，系统删除索引。如果索引名不存在，系统删除索引失败，但不会报错。当未使用 IF EXISTS 关键字时，如果索引名存在，系统删除索引。如果索引名不存在，系统报错，提示索引名不存在。
- index_name：待删除索引的名称。

7.4.5.3.3 语法示例

以下示例删除 ts_table 表中的 sensor_index 索引。

```
SQL
DROP INDEX sensor_index;
```

7.5 DML 语句

DML 语句用于对数据进行操作。

7.5.1 INSERT

INSERT 语句用于向目标表中插入数据，包括：

- 向指定时序表插入一行或多行时序数据。
- 使用 SELECT 子句将其他时序表的查询结果插入到指定时序表。

7.5.1.1 语法格式

```
SQL
INSERT INTO <table_name> [col_tag_list] [VALUES (<value_list>) | <select_stmt> ];
```

7.5.1.2 参数说明

- table_name：要写入数据的表名，格式为 <schema_name>.<table_name> 。如未指定模式名，则向当前模式下的表写入数据。
- col_tag_list：可选参数，指定一个或多个数据列和标签列。数据列或标签列之间使用逗号（,）隔开。
  - 支持指定目标表的全部或部分数据列或标签列，顺序可与表定义的列顺序不同。
  - 指定列必须包括第一列时间戳列和主标签列。
  - 指定列重复或指定列不存在时，系统报错，提示指定列重复或指定列不存在。

- value_list：要写入的数据值和标签值列表，支持添加一个或多个数据值和标签值。数据值和标签值之间使用逗号（,）隔开。
  - 指定列名时，数据值和标签值将按照列名顺序写入，数据类型须与列类型一致，数据值的数量须与指定列的数量一致。如果指定列均为标签列，系统将根据写入数据添加标签值，此时可以忽略数据列的非空约束。
  - 未指定列名时，默认插入所有列。数据值和标签值将按照表定义的列顺序写入，数据类型须与列类型一致，数据值的数量须与列的数量一致。
  - 对于未指定数据的列，默认插入 NULL 值。若该列有非空约束，则系统报错，提示违反非空约束。
  - 写入的数据必须包括时间戳数据和主标签值。
- select_stmt：SELECT 语句，具体语法，参见简单查询。SELECT 语句返回的列数必须与 INSERT INTO 语句中要插入的列数一致。

> 说明
> 目前，KaiwuDB Lite 不支持插入指定表的重复列或重复数据，例如 INSERT INTO t1 SELECT c1, c1 from t0 或者 SELECT 0,0 from t0。

7.5.1.3 语法示例

以下示例假设已经在 tsdb 模式下创建名为 tab 的表。

```
SQL
CREATE TABLE tsdb.tab (ts timestamp not null, col1 int, col2 varchar(128)) TAGS (tag1 int8 NOT NULL, tag2 float4, tag3 varchar(128)) PRIMARY TAGS (tag1);
```

示例 1：指定部分列并写入单行数据

> 说明
> 指定列名插入数据时，第一列（时间戳列）不允许为空，其他数据在满足约束的情况下可正常插入。当允许为空且不写入值时，默认为 NULL。

```
SQL
INSERT INTO tsdb.tab(ts, tag1, col2, tag3) VALUES (now(), 222, 2, 'two');
```

示例 2：指定部分列并写入多行数据

```
SQL
INSERT INTO tsdb.tab (ts, tag1, col2, tag3) VALUES (now(), 111, 1, 'one'), (now(), 222,
2, 'two'), (now(), 333, 3, 'three');
```

示例 3：不指定列写入所有列数据

```
SQL
INSERT INTO tsdb.tab VALUES (now(), 1, NULL, 111, 1.11, 'one');
```

示例 4：只指定标签列写入数据

```
SQL
INSERT INTO tsdb.tab(tag1,tag2,tag3) VALUES (222,2,'two');
```

示例 5：向时序表写入其他时序表的查询结果。

```
SQL
INSERT INTO ts_table SELECT * FROM temperature;
```

7.5.2 UPDATE

UPDATE 语句用户更新表中一行或多行数据。

7.5.2.1 语法格式

```
SQL
UPDATE <table_name> SET <col_name> = <expr>[, <col_name> = <expr>, ...]
[<where_clause>];
```

7.5.2.2 参数说明

- table_name：待更新表的表名，格式为 [<schema_name>.]<table_name> 。如未指定模式名，则更新当前模式下表中的数据。
- col_name：待更新列的列名。
- expr：SQL 表达式，指定要更新的列值。
- where_clause：where 子句，指定要表中要更新的行。格式为 WHERE <column> <operator> <value>，其中 <operator> 支持 +、-、*、/、LIKE、IN、NOT IN、=、!=、<、<=、>、>= 操作符。数值只支持常量。支持使用 AND 或 OR 指定多个过滤条件。

7.5.2.3 语法示例

以下示例假设已经在当前模式下创建 cpu 表并写入相关数据。

```
SQL
-- 1. 创建表
CREATE TABLE cpu (ts TIMESTAMP NOT NULL, cpu_usage INT, memory_usage INT)
TAGs (hostname VARCHAR(128) NOT NULL) PRIMARY TAGS (hostname);


-- 2. 写入数据
INSERT INTO cpu VALUES (NOW(), 10, 8192, '1'), (NOW(), 11, 1024, '2');
INSERT INTO cpu VALUES (NOW(), 12, 8192, '3'), (NOW(), 13, 1024, '4');


-- 3. 查看表
SELECT * FROM cpu;
   ts         | cpu_usage | memory_usage | hostname
------------------------+-----------+--------------+----------
 2025-05-19 01:41:36.886 |       10 |       8192 | 1
 2025-05-19 01:41:36.886 |       11 |       1024 | 2
```

```
2025-05-19 01:42:05.951 |      12 |       8192 | 3
2025-05-19 01:42:05.951 |      13 |       1024 | 4
(4 rows)
```

示例 1：更新表中的数据

```
SQL
-- 1. 更新 hostname 为 2 对应的 cpu_usage 值
UPDATE cpu SET cpu_usage = 14 WHERE hostname = '2';


-- 2. 查看表
SELECT * FROM cpu;
   ts         | cpu_usage | memory_usage | hostname
------------------------+-----------+--------------+----------
2025-05-19 01:41:36.886 |      10 |       8192 | 1
2025-05-19 01:41:36.886 |      14 |       1024 | 2
2025-05-19 01:42:05.951 |      12 |       8192 | 3
2025-05-19 01:42:05.951 |      13 |       1024 | 4
(4 rows)
```

7.5.3 DELETE

`DELETE` 语句用于删除表中的数据。KaiwuDB Lite 支持基于主标签和时间范围删除表中的数据。

7.5.3.1 语法格式

```
SQL
DELETE FROM [<schema_name>.]<table_name> WHERE <filter_condition>;
```

7.5.3.2 参数说明

- table_name：待删除的行所在表的表名，格式为 <schema_name>.<table_name> 。如未指定模式名，则删除当前模式下表中的数据。
- filter_condition：待删除数据的过滤条件。KaiwuDB Lite 支持基于主标签和时间范围删除表中的数据，支持使用 AND 或 OR 指定多个过滤条件。指定时间范围时，格式为<timestamp_column> <operator> <value>，其中时间戳列必须是数据所在表的第一列，操作符支持使用< 、>、=、!=、>= 或 <=； 对应的数值只支持时间戳常量，例如'2023-12-01 08:00:00'。

7.5.3.3 语法示例

以下示例假设已经在当前模式下创建 cpu 表并写入相关数据。

```
SQL
-- 1. 创建表
CREATE TABLE cpu (ts TIMESTAMP NOT NULL, cpu_usage INT, memory_usage INT)
TAGs (hostname VARCHAR(128) NOT NULL) PRIMARY TAGS (hostname);


-- 2. 写入数据
INSERT INTO cpu VALUES (NOW(), 10, 8192, '1'), (NOW(), 11, 1024, '1');
INSERT INTO cpu VALUES (NOW(), 12, 8192, '1'), (NOW(), 13, 1024, '1');
INSERT INTO cpu VALUES (NOW(), 10, 8192, '2'), (NOW(), 11, 1024, '2');
INSERT INTO cpu VALUES (NOW(), 12, 8192, '2'), (NOW(), 13, 1024, '2');


-- 3. 查询数据
SELECT * FROM cpu;
          ts          | cpu_usage | memory_usage | hostname
------------------------+-----------+--------------+----------
 2024-10-22 02:15:56.196 |        10 |         8192 | 1
 2024-10-22 02:15:56.196 |        11 |         1024 | 1
```

```
2024-10-22 02:16:22.706 |      12 |      8192 | 1
2024-10-22 02:16:22.706 |      13 |      1024 | 1
2024-10-22 02:16:33.943 |      10 |      8192 | 2
2024-10-22 02:16:33.943 |      11 |      1024 | 2
2024-10-22 02:16:45.387 |      12 |      8192 | 2
2024-10-22 02:16:45.387 |      13 |      1024 | 2
(8 rows)
```

示例 1：基于主标签删除表中的数据

```
SQL
DELETE FROM cpu WHERE hostname = '2';
```

示例 2：基于时间范围删除表中的数据

```
SQL
DELETE FROM cpu WHERE ts > '2024-10-22 02:16:00';
```

示例 3：基于主标签和时间范围删除表中的数据

```
SQL
DELETE FROM cpu WHERE ts < '2024-10-22 02:16:00' and hostname = '1';
```

7.6 SELECT

SELECT 语句用于从表中查询数据。

7.6.1 简单查询

7.6.1.1 语法格式

```
SQL
SELECT [DISTINCT] <select_list> [AS <alias>] FROM <table_name> [WHERE
<filter_condition>] [GROUP BY <groups>] HAVING <group_filter> [ORDER BY
```

```
<order_expr>] [LIMIT <n>] [OFFSET <m>];
```

7.6.1.2 参数说明

- DISTINCT：当使用 DISTINCT 关键字时，系统删除返回结果中重复的行。
- select_list：指定查询的列或表达式列表。支持指定数据列、标签列和聚合函数。目前，KaiwuDB Lite 支持以下聚合函数：first、last、avg、count、max、min 和 sum。
- alias：可选参数，为查询结果集起的别名。
- table_name：查询的表名。
- WHERE 子句：指定过滤条件，筛选出符合条件的查询结果。格式为 WHERE <column> <operator> <value>，其中 <operator> 支持 +、-、*、/、LIKE、IN、NOT IN、=、!=、<、<=、>、>= 操作符。数值只支持常量。支持使用 AND 或 OR 指定多个过滤条件。
- GROUP BY 子句：将数据集划分成小组，然后对这些小组进行数据处理。
- HAVING 子句：当 WHERE 关键字无法与聚合函数一起使用时，HAVING 子句可以用来筛选分组后的各组数据。通常情况下，HAVING 子句与 GROUP BY 子句联用，只检索表达式返回值为 TRUE 的聚合函数组。HAVING 子句的作用类似于 WHERE 子句，但适用于聚合函数。
- ORDER BY 子句：指定排序规则，可由一个或多个排序规范组成，支持使用列、函数和算术运算符（+、-、*、/）的组合，也可以通过 ASC（升序，默认）或 DESC（降序）关键字来控制排序顺序。
- LIMIT 子句：指定返回结果的最大行数。例如，LIMIT 10 表示限制查询结果最多为 10 行。
- OFFSET 子句：跳过前面的偏移量行数。OFFSET 子句通常与 LIMIT 组合使用，通过限制结果的数量，实现分页显示结果，避免一次性检索所有数据。

7.6.1.3 语法示例

以下示例假设已经在 tsdb 模式下创建 power 表，并向表中写入数据。

```
SQL
-- 1. 创建表
CREATE TABLE tsdb.power (ts TIMESTAMP NOT NULL, c1 INT) TAGs (tag1 INT NOT NULL) PRIMARY TAGS (tag1);

-- 2. 写入数据
INSERT INTO tsdb.power VALUES ('2024-10-16 01:44:26.901', 3, '1'), ('2024-10-16 01:44:26.902 ', 0, '1');
```

示例 1：一般查询

```
SQL
SELECT* FROM tsdb.power;
        ts         | c1 | tag1
-------------------------+----+------
 2024-10-16 01:44:26.901 |  3 |    1
 2024-10-16 01:44:26.902 |  0 |    1
(2 rows)
```

示例 2：使用 WHERE 子句指定过滤条件

```
SQL
SELECT * FROM tsdb.power WHERE c1 =3;
        ts         | c1 | tag1
-------------------------+----+------
 2024-10-16 01:44:26.901 |  3 |    1
(1 row)
```

示例 3：使用 GROUP BY 子句进行分组查询

```
SQL
SELECT c1 AS ct FROM tsdb.power GROUP BY c1;
 ct
----
 3
 0
(2 rows)
```

示例 4：使用 ORDER BY 子句进行分组查询

```
SQL
SELECT c1 AS ct FROM tsdb.power GROUP BY c1 ORDER BY c1 ASC;
 ct
----
 0
 3
(2 rows)
```

示例 5：使用 LIMIT 子句限制返回的记录数

```
SQL
SELECT * FROM tsdb.power LIMIT 1;
        ts         | c1 | tag1
-------------------------+----+------
 2024-10-16 01:44:26.901 |  3 |    1
(1 row)
```

7.6.2 嵌套查询

嵌套查询指在一个 SQL 查询中嵌套另一个完整的 SQL 查询，从而实现更复杂的数据检索。

KaiwuDB Lite 支持以下嵌套查询：

- 相关子查询（Correlated Subquery）：内部查询依赖于外部查询的结果，每次外部查询都触发内部查询。
- 非相关子查询（Non-Correlated Subquery）：内部查询独立于外部查询，只执行一次内部查询并返回固定的结果。
- 相关投影子查询（Correlated Scalar Subquery）：内部查询依赖于外部查询的结果，并且只返回一个单一的值作为外部查询的结果。
- 非相关投影子查询（Non-Correlated Scalar Subquery）：内部查询独立于外部查询，并且只返回一个单一的值作为外部查询的结果。
- FROM 子查询：将一个完整的 SQL 查询嵌套在另一个查询的 FROM 子句中，作为临时表格使用。

> 说明
> 当 WHERE 子句中包含多个嵌套子查询和逻辑运算符（AND、OR），且某些子查询存在语义错误时，执行可能会报错 internal error: invalid index。

7.6.2.1 语法格式

```
SQL
<select_clause> (<select_clause>);
```

7.6.2.2 参数说明

无

7.6.2.3 语法示例

示例 1：非相关子查询

```
SQL
SELECT e1 FROM ts_stable1 WHERE e1 = (SELECT avg (e1) FROM t1.stable);
e1
```

```
(0 rows)
```

示例 2：非相关投影子查询

```
SQL
SELECT first (e1) = (SELECT e1 FROM ts_stable2 limit 1) FROM ts_stable1;
?column?
-------------
t
(1 row)
```

示例 3：相关子查询

```
SQL
SELECT e1 FROM t1.stable WHERE e1 in (SELECT e1 FROM t1.stable2 WHERE
stable.e2=stable2.e2);
e1
-----
1000
2000
3000
2000
3000
4000
3000
4000
5000
(9 rows)
```

示例 4：相关投影子查询

```
SQL
SELECT sum(e1) = (SELECT e1 FROM ts_stable2 WHERE ts_stable2.e1=e1) FROM
```

```
ts_stable1;
?column?
-------------
t
(1 row)
```

示例 5：FROM 子查询

```
SQL
SELECT avg (a) FROM (SELECT e1 as a FROM t1.ts_stable1);
avg
------
1000
(1 row)
```

7.6.3 关联查询

关联查询（JOIN QUERY）从多个表中获取相关联的数据，将其联接成一个结果集，从而得到更丰富的信息。KaiwuDB Lite 支持以下关联类型：

- 内连接（INNER JOIN）
- 左连接（LEFT JOIN）
- 右连接（RIGHT JOIN）
- 全连接（FULL JOIN）

> 说明
>
> 使用 FULL JOIN 时，避免在连接条件中使用子查询。

7.6.3.1 语法格式

```
SQL
代码块
```

```
[(joined_table)
| <table_ref> CROSS <opt_join_hint> JOIN <table_ref>
| <table_ref> NATURAL JOIN <table_ref>
| <table_ref> NATURAL [FULL | LEFT | RIGHT ] [OUTER] <opt_join_hint> JOIN
<table_ref>
| <table_ref> NATURAL INNER <opt_join_hint> JOIN <table_ref>
| <table_ref> [FULL | LEFT | RIGHT ] [ OUTER ] <opt_join_hint> JOIN <table_ref>
[ ON <a_expr> | USING (<name>, <name>, ...)]
| <table_ref> INNER <opt_join_hint> JOIN <table_ref>  [ ON <a_expr> | USING
(<name>, <name>, ...)]];
```

- 内连接

```
sql
代码块
<table_ref> [ INNER ] JOIN <table_ref>  [ ON <a_expr> | USING (<name>,
<name>, ...)];
<table_ref> NATURAL [ INNER ] <opt_join_hint> JOIN <table_ref>;
<table_ref> CROSS <opt_join_hint> JOIN <table_ref>;
```

- 左连接

```
sql
代码块
<table_ref> LEFT [ OUTER ] JOIN <table_ref>  [ ON <a_expr> | USING (<name>,
<name>, ...)];
<table_ref> NATURAL LEFT [ OUTER ] <opt_join_hint> JOIN <table_ref>;
```

- 右连接

```
sql
代码块
<table_ref> RIGHT [ OUTER ] JOIN <table_ref>  [ ON <a_expr> | USING (<name>,
```

```
<name>, ...)];
<table_ref> NATURAL RIGHT [ OUTER ] <opt_join_hint> JOIN <table_ref>;
```

- 全连接

```
sql
代码块
<table_ref> FULL [ OUTER ] JOIN <table_ref> [ ON <a_expr> | USING (<name>,
<name>, ...)];
<table_ref> NATURAL FULL [ OUTER ] <opt_join_hint> JOIN <table_ref>;
```

- opt_join_hint

```
SQL
[MERGE | HASH | LOOKUP | INVERTED ]
```

7.6.3.2 参数说明

- joined_table：连接表达式。
- table_ref：表的表达式。
- opt_join_hint：可选项，连接提示。
- a_expr：ON 连接条件的标量表达式。
- name：USING 连接条件的列名。

7.6.3.3 语法示例

```
SQL
SELECT ts_table1.e1, ts_table2.e1 FROM ts_table1 LEFT JOIN ts_table2 ON
ts_table1.e1 = ts_table2.e1;
e1 | e1
----+-----
1000|1000
```

```
(1 row)
```

7.6.4 联合查询

联合查询（UNION QUERY）将具有相同列结构的多个查询结果组合成一个结果表。

7.6.4.1 语法格式

```
SQL
<select_clause> [UNION | INTERSECT | EXCEPT] [ALL | DISTINCT] <select_clause>;
```

7.6.4.2 参数说明

- UNION：将两个或多个查询结果合并成一个结果集，并自动去除重复的行。
- INTERSECT：将两个查询结果的交集作为最终结果集。
- EXCEPT：返回只存在于第一个查询结果中而不存在于第二个查询结果中的行。
- ALL：可选关键字，当使用 `ALL` 关键字时，系统不删除返回结果中重复的行。
- DISTINCT：可选关键字，当使用 `DISTINCT` 关键字时，系统删除返回结果中重复的行。

7.6.4.3 语法示例

以下示例使用 `UNION` 参数将两个查询结果集合并成一个结果集。

```
SQL
SELECT e1 FROM t1.ts_stable1 UNION SELECT e1 FROM t1.ts_stable2;
e1
------
1000
(1 row)
```

7.6.5 插值查询

时间序列数据中，有时会存在缺失和偏离的数据，影响后续数据的使用和分析。KaiwuDB Lite 提供了 time_bucket_gapfill() 函数和 interpolate() 函数，用于对指定窗口间隔的数据进行时间戳对齐，插入缺失的时间戳行，并根据需要选择是否进行补值。

使用说明：

- 分组补全规则：time_bucket_gapfill() 函数必须与 GROUP BY 一起时，补全规则取决于分组情况：
  - 单行数据分组：如果分组（除 time_bucket_gapfill() 外的其他分组项）中仅有一行数据：时间范围即为该行的 time_bucket_gapfill()值，不需补全。
  - 多行数据分组：如果分组中有多行数据：时间范围为该组 time_bucket_gapfill()值的最小值到最大值，系统会在此范围内补全缺失行。
- 其他列查询：如果需要同时查询其他列信息，且待查询的列不在 GROUP BY 指定的分组项内，需要使用聚合函数来处理这些列。
- 限制：time_bucket_gapfill()必须作为查询或者查询中的顶层表达式，不能嵌套在其他函数内。

7.6.5.1 语法格式

```
SQL
SELECT [<group_column_list>],
    time_bucket_gapfill(<t_expr>, <i_expr>),
    [interpolate(<expr>[, <mode>]),]
    [aggregate_function(<other_columns>), ...]
FROM <table_name>
[<where_clause>]
GROUP BY [<group_column_list>],
```

```
    time_bucket_gapfill(<t_expr>, <i_expr>)
[ORDER BY ...];
```

7.6.5.2 参数说明

- group_column_list：分组字段列表，支持一个或多个字段，用逗号分隔。
- t_expr：时间戳表达式，类型为 TIMESTAMP 或 TIMESTAMPTZ，通常为表列。
- i_expr：时间间隔表达式，类型为 INTERVAL，必须指定时间单位。支持的时间单位如下：
  - MICROSECOND / MICROSECONDS（微秒）
  - MILLISECOND / MILLISECONDS（毫秒）
  - SECOND / SECONDS（秒）
  - MINUTE / MINUTES（分钟）
  - HOUR / HOURS（小时）
  - DAY / DAYS（天）
  - WEEK / WEEKS（周）
  - MONTH / MONTHS（月）
  - YEAR / YEARS（年）
- expr: 结果为数值类型的聚合函数，支持 TINYINT、SMALLINT、INT、BIGINT、DECIMAL、FLOAT、DOUBLE。
- mode：补值模式，可选值包括：
  - 常量值：使用指定常量值填补，常量类型必须与 expr 结果类型兼容。
  - prev：使用前一个未缺失值补全
  - next：使用后一个未缺失值补全
  - linear：线性插值补全，在 [前一个非空值， 后一个非空值] 区间内进行线性

计算，支持递增和递减，当前值大于后值时，按递减方式插值。

- null：以 NULL 补全，同 interpolate(expr)。

7.6.5.3 语法示例

以下示例以物联网设备温度监控场景为例，演示如何处理传感器数据中的缺失值。

```
SQL
-- 创建设备温度监控表
CREATE TABLE sensor_data(
   timestamp TIMESTAMP,    -- 采集时间
   temperature DOUBLE,     -- 温度值
   device_id INT           -- 设备 ID
);

-- 插入模拟数据（部分时间点缺失）
INSERT INTO sensor_data VALUES
-- 设备 1 的数据（缺少 05-04 的数据）
('2025-05-03 09:00:00', 22.5, 1),
('2025-05-05 09:00:00', 24.8, 1),

-- 设备 2 的数据（缺少 05-04、05-08~05-19 的数据）
('2025-05-03 09:00:00', 21.2, 2),
('2025-05-05 09:00:00', 23.1, 2),
('2025-05-06 09:00:00', 23.8, 2),
('2025-05-07 09:00:00', 25.3, 2),
('2025-05-20 09:00:00', 22.9, 2);

-- 查看原始数据
SELECT * FROM sensor_data ORDER BY device_id, timestamp;
```

示例 1：识别数据缺失情况（不补值）

```
SQL
SELECT device_id,
    time_bucket_gapfill(timestamp, INTERVAL 1 DAY) AS date_bucket,
    avg(temperature) AS avg_temp
FROM sensor_data
GROUP BY device_id, time_bucket_gapfill(timestamp, INTERVAL 1 DAY)
ORDER BY device_id, date_bucket;
```

示例 2：使用前值补全

```
SQL
SELECT device_id,
    time_bucket_gapfill(timestamp, INTERVAL 1 DAY) AS date_bucket,
    interpolate(avg(temperature), prev) AS filled_temp
FROM sensor_data
GROUP BY device_id, time_bucket_gapfill(timestamp, INTERVAL 1 DAY)
ORDER BY device_id, date_bucket;
```

示例 3： 使用后值补全

```
SQL
SELECT device_id,
    time_bucket_gapfill(timestamp, INTERVAL 1 DAY) AS date_bucket,
    interpolate(avg(temperature), next) AS filled_temp
FROM sensor_data
GROUP BY device_id, time_bucket_gapfill(timestamp, INTERVAL 1 DAY)
ORDER BY device_id, date_bucket;
```

示例 4：使用常量值补全

```
SQL
```

```
SELECT device_id,
    time_bucket_gapfill(timestamp, INTERVAL 1 DAY) AS date_bucket,
    interpolate(avg(temperature), -999.0) AS filled_temp  -- 使用-999 标记缺失
FROM sensor_data
GROUP BY device_id, time_bucket_gapfill(timestamp, INTERVAL 1 DAY)
ORDER BY device_id, date_bucket;
```

示例 5：线性插值补全

```
SQL
SELECT device_id,
    time_bucket_gapfill(timestamp, INTERVAL 1 DAY) AS date_bucket,
    interpolate(avg(temperature), linear) AS filled_temp
FROM sensor_data
GROUP BY device_id, time_bucket_gapfill(timestamp, INTERVAL 1 DAY)
ORDER BY device_id, date_bucket;
```

7.6.6 标签查询

KaiwuDB Lite 支持使用 SELECT 语句查询指定表的所有标签列或特定标签列的标签值。

7.6.6.1 语法格式

- 查询指定表的所有标签列

  ```
  SQL
  SELECT * FROM kaiwudb_tags('<table_name>');
  ```

- 查询指定标签列的标签值

  ```
  SQL
  SELECT [DISTINCT] <tag_col_list> FROM <table_name>;
  ```

7.6.6.2 参数说明

- table_name：待查询的表名。
- tag_col_list：待查询的标签列列表，支持指定一个或多个标签列，多个列名之间使用逗号（,）分隔。

7.6.6.3 语法示例

示例 1：查询指定表的所有标签列

```
SQL
SELECT * FROM kaiwudb_tags('nulls');
 tag | type  | is_primary | nullable
-----+---------+------------+----------
 t1  | INTEGER | true       | false
(1 row)
```

示例 2：查询指定标签列的非重复标签值

```
SQL
SELECT DISTINCT t1 FROM nulls;
 t1
----
  1
  3
  2
(3 rows)
```

7.6.7 注释查询

KaiwuDB Lite 支持使用 SELECT 语句查询所有表、列的注释信息。用户可以使用 WHERE 子句查看指定表、列的注释信息。

7.6.7.1 语法格式

- 查看表的注释信息

```
sql
SELECT comment FROM kaiwudb_tables() [WHERE table_name =
'<table_name>'];
```

- 查看列的注释信息

```
sql
SELECT comment FROM kaiwudb_columns() [WHERE table_name =
'<table_name>' and column_name = '<column_name>'];
```

7.6.7.2 参数说明

- WHERE 子句：指定过滤条件，筛选出符合条件的查询结果。
- table_name：查询的表名。
- column_name：查询的列名。

7.6.7.3 语法示例

以下示例假设已经在 tsdb 模式下，创建 power 表并写入相关数据。

示例 1：查询指定表的注释信息

```
sql
-- 1. 为 power 表添加注释信息。
COMMENT ON TABLE tsdb.power IS 'auto-generated';
ALTER


-- 2. 查看表的注释信息。
SELECT comment from  kaiwudb_tables() where table_name = 'power';
```

```
   comment
----------------
 auto-generated
(1 rows)
```

示例 2：查询指定列的注释信息

```
sql
-- 1. 为 power 表的 c3 列添加注释信息。
COMMENT ON COLUMN power.c3 IS 'user-defined value';
ALTER


--2. 查看 c3 列的注释信息。
SELECT comment FROM kaiwudb_columns() WHERE table_name= 'power' AND
column_name = 'c3';
     comment
--------------------
 user-defined value
(1 row)
```

7.6.8 建表语句查询

KaiwuDB Lite 支持使用 SELECT 语句查询数据库中所有表的建表语句。用户可以使用 WHERE 子句查看指定表的建表语句。

7.6.8.1 语法格式

```
sql
SELECT SQL FROM kaiwudb_tables() [WHERE table_name = '<table_name>'];
```

7.6.8.2 参数说明

- WHERE 子句：指定过滤条件，筛选出符合条件的查询结果。
- table_name：查询的表名。

7.6.8.3 语法示例

示例 1：查看数据库中所有表的建表语句

```sql
SELECT SQL FROM kaiwudb_tables();
                                                  sql
----------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------
--------
 CREATE TABLE "nulls"(ts TIMESTAMP NOT NULL, price INTEGER) TAGS(t1 INTEGER
NOT NULL) PRIMARY TAGS(t1);
 CREATE TABLE nullse(ts TIMESTAMP NOT NULL, price INTEGER) TAGS(t1 INTEGER
NOT NULL) PRIMARY TAGS(t1);
 CREATE TABLE tsdb.bools(ts TIMESTAMP NOT NULL, c1 BOOLEAN, c2 BOOLEAN)
TAGS(tag1 INTEGER NOT NULL) PRIMARY TAGS(tag1);
 CREATE TABLE tsdb.power(ts TIMESTAMP, c1 INTEGER, tag1 INTEGER, c3 BIGINT, c5
VARCHAR, PRIMARY KEY(ts, tag1));
 CREATE TABLE tsdb.floats(ts TIMESTAMP, c1 FLOAT, c2 FLOAT, c3 DOUBLE, c4
DOUBLE, tag1 INTEGER, PRIMARY KEY(ts, tag1));
 CREATE TABLE tsdb.ints(ts TIMESTAMP NOT NULL, c1 SMALLINT, c2 SMALLINT, c3
SMALLINT, c4 INTEGER, c5 INTEGER, c6 BIGINT, c7 BIGINT, c8 BIGINT) TAGS(tag1
INTEGER NOT NULL) PRIMARY TAGS(tag1);
 CREATE TABLE tsdb.intss(ts TIMESTAMP NOT NULL, c1 SMALLINT, c2 SMALLINT, c3
SMALLINT, c4 INTEGER, c5 INTEGER, c6 BIGINT, c7 BIGINT, c8 BIGINT, c9 SMALLINT,
c10 INTEGER, c11 BIGINT) TAGS(tag1 INTEGER NOT NULL) PRIMARY TAGS(tag1);
 CREATE TABLE tsdb.timestamps(ts TIMESTAMP NOT NULL, c1 TIMESTAMP)
```

```
TAGS(site INTEGER NOT NULL) PRIMARY TAGS(site);
 CREATE TABLE tsdb.timestampss(ts TIMESTAMP NOT NULL, c1 TIMESTAMP WITH
TIME ZONE, c2 TIMESTAMP, c3 TIMESTAMP WITH TIME ZONE, c4 TIMESTAMP, c5
INTERVAL, c6 DATE) TAGS(site INTEGER NOT NULL) PRIMARY TAGS(site);
 CREATE TABLE tsdb.varchars(ts TIMESTAMP NOT NULL, c1 VARCHAR, c2 VARCHAR)
TAGS(tag1 INTEGER NOT NULL) PRIMARY TAGS(tag1);
 CREATE TABLE tsdb.varcharss(ts TIMESTAMP NOT NULL, c1 VARCHAR, c2 VARCHAR,
c3 VARCHAR, c4 VARCHAR, c5 VARCHAR, c6 VARCHAR, c7 VARCHAR) TAGS(tag1
INTEGER NOT NULL) PRIMARY TAGS(tag1);
 CREATE TABLE tsdb.varcharsss(ts TIMESTAMP NOT NULL, c1 VARCHAR, c2
VARCHAR, c3 VARCHAR, c4 VARCHAR, c5 VARCHAR, c6 VARCHAR, c7 VARCHAR)
TAGS(tag1 INTEGER NOT NULL) PRIMARY TAGS(tag1);
(12 rows)
```

示例 2：查看指定表的建表语句

```
sql
SELECT SQL FROM kaiwudb_tables() WHERE table_name = 'power';
                                    sql
-----------------------------------------------------------------------------------------------------------
 CREATE TABLE tsdb.power(ts TIMESTAMP, c1 INTEGER, tag1 INTEGER, c3 BIGINT, c5
VARCHAR, PRIMARY KEY(ts, tag1));
(1 row)
```

7.7 PREPARE

PREPARE 语句用于预处理 SQL 语句的 SQL 命令，允许系统将 SQL 语句准备好以供后续的执行。SQL 语句准备好后，可以使用 EXECUTE 命令执行 SQL 语句。

KaiwuDB Lite 支持用户使用 PREARE 和 EXECUTE 语句向指定表写入数据、查询指定表的数据或删除指定表的数据。

7.7.1.1 语法格式

- PREPARE

```
SQL
PREPARE <statement_name> AS <statement_sql>;
```

- EXECUTE

```
Plain Text
EXECUTE <statement_name> ( <parameter_value> );
```

7.7.1.2 参数说明

- statement_name：预处理的 SQL 语句名称。
- statement_sql：INSERT、QUERY 或 DROP 语句，语句中使用 $<number> 做占位符，例如 $1、$2。
- parameter_value：待写入、查询或删除的参数值。参数值需要对应 INSERT、QUERY 或 DROP 语句中的占位符。

7.7.1.3 语法示例

示例 1：不指定列名向时序表写入一行数据。

```
SQL
-- 1. 创建表
CREATE TABLE vehicle_gps_track (TIME timestamp not NULL, LATITUDE float,
LONGITUDE float, ALTITUDE float, SPEED float, DIRECTION varchar) TAGS (IMEI int
not null) PRIMARY TAGS (IMEI);
CREATE


-- 2. 创建 PREPARE 写入语句
PREPARE p1 AS INSERT INTO vehicle_gps_track VALUES ($1,$2,$3,$4,$5,$6,$7);
```

```
PREPARE


-- 3. 执行写入语句
EXECUTE p1 ('2024-02-06 12:00:00', 34.0522, -118.2437, 100, 60, 'North', 12345678);
 Count
-------
     1
(1 row)


-- 4. 查询写入结果
SELECT * FROM vehicle_gps_track;
       TIME        | LATITUDE | LONGITUDE | ALTITUDE | SPEED | DIRECTION |   IMEI
---------------------+----------+-----------+----------+-------+-----------+----------
 2024-02-06 12:00:00 |  34.0522 | -118.2437 |    100.0 |  60.0 | North     | 12345678
(1 row)
```

示例 2：不指定列名向时序表写入多行数据。

```
SQL
-- 1. 创建 PREPARE 写入语句
PREPARE p2 AS INSERT INTO vehicle_gps_track VALUES
($1,$2,$3,$4,$5,$6,$7),($8,$9,$10,$11,$12,$13,$14);
PREPARE


-- 2. 执行写入语句
EXECUTE p2 ('2024-02-06 12:15:00', 40.7128, -74.0060, 150, 55, 'East',
23456789,'2024-02-06 12:30:00', 51.5074, -0.1278, 80, 70, 'West', 34567890);
 Count
-------
     2
```

```
(1 row)


-- 3. 查询写入结果
SELECT * FROM vehicle_gps_track;
      TIME       | LATITUDE | LONGITUDE | ALTITUDE | SPEED | DIRECTION |   IMEI
---------------------+----------+-----------+----------+-------+-----------+----------
 2024-02-06 12:00:00 |  34.0522 | -118.2437 |    100.0 |  60.0 | North     | 12345678
 2024-02-06 12:15:00 |  40.7128 |   -74.006 |    150.0 |  55.0 | East      | 23456789
 2024-02-06 12:30:00 |  51.5074 |   -0.1278 |     80.0 |  70.0 | West      | 34567890
(3 rows)
```

示例 3：查询数据。

```
SQL
-- 1. 创建 PREPARE 查询语句
PREPARE p3 AS SELECT * FROM vehicle_gps_track WHERE imei=$1;
PREPARE

-- 2. 执行查询语句
EXECUTE p3(12345678);
         time          | latitude | longitude | altitude | speed | direction |   imei
---------------------------+----------+-----------+----------+-------+-----------+-----------
 2024-02-06 12:00:00+00:00 |  34.0522 | -118.2437 |      100 |    60 | North     | 12345678
```

示例 4：删除数据。

```
SQL
-- 1. 创建 PREPARE 删除语句
PREPARE p4 AS DELETE FROM vehicle_gps_track WHERE imei=$1;
PREPARE
```

```
-- 2. 执行删除语句
EXECUTE p4(12345678);
 Count
-------
     1
(1 row)

-- 3. 查询删除结果
SELECT * FROM vehicle_gps_track;
      TIME       | LATITUDE | LONGITUDE | ALTITUDE | SPEED | DIRECTION |   IMEI
---------------------+----------+-----------+----------+-------+-----------+----------
 2024-02-06 12:15:00 |  40.7128 |   -74.006 |    150.0 |  55.0 | East      | 23456789
 2024-02-06 12:30:00 |  51.5074 |   -0.1278 |     80.0 |  70.0 | West      | 34567890
(2 rows)
```

7.8 SHOW

7.8.1 SHOW

SHOW 语句用于查看所有模式下所有表的详细信息。

> 提示：
> 可通过\q 退出查询结果。

7.8.1.1 语法格式

```
SQL
SHOW;
```

7.8.1.2 参数说明

无

7.8.1.3 语法示例

```
SQL
 SHOW;
 database | schema |   name   |          column_names          |
column_types                          | temporary
----------+--------+----------+--------------------------------------+-----------------------------
-------------------------------------------------------------+-----------
 metadata | tsdb   | bools    | [ts, c1, c2, tag1]             | [TIMESTAMP, BOOLEAN,
BOOLEAN, INTEGER]                     | f
 metadata | tsdb   | chars    | [ts, c1, c2, tag1]             | [TIMESTAMP, CHAR(128),
CHAR(1023), INTEGER]                  | f
 metadata | tsdb   | power    | [ts, c1, tag1]                 | [TIMESTAMP, INTEGER,
INTEGER]                              | f
 metadata | tsdb   | floats   | [ts, c1, c2, c3, c4, tag1]     | [TIMESTAMP, FLOAT,
FLOAT, DOUBLE, DOUBLE, INTEGER]       | f
 metadata | tsdb   | ints     | [ts, c1, c2, c3, c4, c5, c6, c7, c8, tag1] | [TIMESTAMP,
SMALLINT, SMALLINT, SMALLINT, INTEGER, INTEGER, BIGINT, BIGINT, BIGINT,
INTEGER] | f
 metadata | tsdb   | timestamps | [ts, c1, site]             | [TIMESTAMP,
TIMESTAMP, INTEGER]                   | f
 metadata | tsdb   | varchars   | [ts, c1, c2, tag1]         | [TIMESTAMP,
VARCHAR(128), VARCHAR(65536), INTEGER]| f
 metadata | tsdb1  | order    | [ts, col1, col2, tag1, tag2, tag3] | [TIMESTAMP,
INTEGER, VARCHAR(128), BIGINT, FLOAT, VARCHAR(128)] | f
(8 rows)
```

7.8.2 SHOW SCHEMAS

SHOW SCHEMAS 语句用于查看所有模式的信息。

7.8.2.1 语法格式

```
SQL
SHOW SCHEMAS;
```

7.8.2.2 参数说明

无

7.8.2.3 语法示例

示例 1：查看所有模式信息。

```
SQL
SHOW SCHEMAS;
 schema_name
-------------
 main
 tsdb
(2 rows)
```

7.8.3 SHOW search_path

SHOW search_path 用于查看当前模式。

> 说明
> 如未提前使用 SET search_path = <schema_name> 命令设置查询路径，系统回显有可能为空。

7.8.3.1 语法格式

```
SQL
```

```
SHOW search_path;
```

7.8.3.2 语法示例

示例 1：查看当前模式的信息。

```
SQL
-- 1. 设置当前查询路径。
SET search_path = tsdb


-- 2. 查看当前模式的信息。
SHOW search_path;
 search_path
-------------
 tsdb
(1 row)
```

7.8.4 SHOW TABLES

`SHOW TABLES` 语句用于查看当前或指定模式下的所有表。

7.8.4.1 语法格式

```
SQL
SHOW TABLES [FROM <schema_name>];
```

7.8.4.2 参数说明

- schema_name：待查看表所在模式的名称，未指定时，表示查看当前模式下的表。

7.8.4.3 语法示例

示例 1：查看当前模式下的所有表。

```
SQL
SHOW TABLES;
```

示例 2：查看指定模式下的所有表。

```
SQL
SHOW TABLES FROM benchmark;
 table_name
------------
 timestamps
(1 row)
```

7.8.5 SHOW CREATE TABLE

`SHOW CREATE TABLE` 语句用于查看当前或模式下指定表的建表语句。如未指定模式，则默认为当前模式。用户也可以使用 `SELECT` 语句查询数据库中所有表的建表语句。有关详细信息，参见建表语句查询。

7.8.5.1 语法格式

```
SQL
SHOW CREATE TABLE [<schema_name>.] <table_name>;
```

7.8.5.2 参数说明

- schema_name：待查看表所在模式的名称。如未指定，则默认使用当前模式。
- table_name：待查看表的名称。

7.8.5.3 语法示例

- 查看当前模式下指定表的建表语句。

以下示例查看当前模式下 ts_table 表的建表语句。

```sql
SHOW CREATE TABLE ts_table;
 table_name |                         create_statement
------------+--------------------------------------------------------------------------------------------
----------------------------
 ts_table   | CREATE TABLE tsdb.ts_table("ts" TIMESTAMP_NS NOT NULL, c1
TIMESTAMP) TAGS(site INTEGER NOT NULL) PRIMARY TAGS(site);
(1 row)
```

- 查看指定模式下指定表的建表语句。

  以下示例查看 benchmark 模式下 cpu 表的建表语句。

```sql
SHOW CREATE TABLE benchmark.cpu;
 table_name |                              create_statement
------------+--------------------------------------------------------------------------------------------
------------------------------------------------
 cpu        | CREATE TABLE benchmark.cpu("ts" TIMESTAMP NOT NULL,
cpu_usage INTEGER, memory_usage INTEGER) TAGS(hostname VARCHAR NOT
NULL) PRIMARY TAGS(hostname);
(1 row)
```

7.8.6 SHOW COLUMNS

7.8.6.1 语法格式

```sql
SHOW COLUMNS FROM [<schema_name>.]<table_name>;
```

7.8.6.2 参数说明

- schema_name：待查看表所在模式的名称。如未指定，则默认使用当前模式。
- table_name：待查看列所在表的名称。

7.8.6.3 语法示例

以下示例查看 ts_table 表中各列的详细信息。

```
SQL
SHOW COLUMNS FROM ts_table;
 column_name | column_type | null | key | default | extra
-------------+-------------+------+-----+---------+-------
 ts          | TIMESTAMP   | NO   |     |         |
 c1          | FLOAT       | YES  |     |         |
 c2          | FLOAT       | YES  |     |         |
 c3          | DOUBLE      | YES  |     |         |
 c4          | DOUBLE      | YES  |     |         |
(5 rows)
```

7.9 EXPLAIN

EXPLAIN 语句用于显示物理执行计划，即 SQL 查询的实际执行方案。物理执行计划由一系列按特定顺序执行的算子组成，这些算子构成了一棵算子树，共同生成查询结果。

用户可使用以下 SQL 语句控制 EXPLAIN 语句的输出仅显示物理计划（默认值）、仅显示优化后的计划或显示逻辑计划、优化后的计划及物理计划。

```
SQL
SET explain_output = [physical_only | optimized_only | all];
```

7.9.1.1 语法格式

```
SQL
EXPLAIN [ANALYZE] sql_statement;
```

7.9.1.2 参数说明

- ANALYZE：可选关键字。
  - 未指定时，系统不会真正执行该 SQL 语句，因此只能看到每个算子的估算基数（Estimated Cardinality， EC）。这些估算值是基于基础表的统计信息，结合各算子的启发式规则（heuristics）计算得出。
  - 指定该关键字时，系统会实际执行 SQL 语句，因此能够提供每个算子的运行时性能数据，包括估算基数和实际基数（Actual Cardinality）。注意：输出中显示的是每个算子所消耗的累计时间（Wall-Clock Time）。当查询采用多线程并行处理时，整体查询处理时间可能小于所有算子执行时间的总和。
- sql_statement：要分析的 SQL 语句，通常为 SELECT 语句。

7.9.1.3 语法示例

以下示例假设已创建表 t1 和 t2， 并向表中写入数据。

```
sql
--- 创建表 t1
create table t1(c1 timestamp not null, c2 double, c3 double) tags(tag1 int not null);

--- 创建表 t2
create table t2(c1 timestamp not null, c2 double, c3 double) tags(tag1 int not null);

--- 向表 t1 写入数据
insert into t1 values ('2025-03-05 12:12:12', 1.2, 5.6, 1);
```

```
--- 向表 t2 写入数据
insert into t2 values ('2025-03-06 12:12:12', 1.2, 5.6, 1);
```

示例 1：仅显示物理计划且不实际执行查询语句。

```
SQL
--- 控制 EXPLAIN 语句的输出仅显示物理计划
SET explain_output='physical_only';

--- 执行 EXPLAIN 语句，但不实际执行指定的 SELECT 语句
EXPLAIN SELECT t1.c2 FROM t1 JOIN t2 USING (tag1) WHERE t2.c3 > 3.0;

 explain_key |              explain_value
-------------+-----------------------------------------------------------
 physical_plan | ┌─────────────────────────────────────────────────────────┐
+
               |  │      PROJECTION      │                +
               |  │ ──────────────────── │                +
               |  │        c2            │                +
               |  │                      │                +
               |  │      ~1 Rows         │                +
               |  └──────────┬───────────┘
+
               |  ┌──────────┴───────────┐
+
               |  │      HASH_JOIN       │                +
               |  │ ──────────────────── │                +
               |  │    Join Type: INNER  │                +
               |  │                      │                +
```

```
        | |      Conditions:      |-------------------------|         +
        | |      tag1 = tag1      |         |         +
        | |                       |         |         +
        | |       ~1 Rows         |         |         +
        | |-----------------------|-------------------------|         |
+
        |
 |----------------------|-------------------------| |-----------------
---------|-------------------------| +
        | |      SEQ_SCAN         | |      SEQ_SCAN         | +
        | |  ---------------------------------------  | |
-----------------------------------------  | +
        | |      Table: t1        | |      Table: t2        | +
        | |  Type: Sequential Scan  | |  Type: Sequential Scan  | +
        | |                       | |    Projections: tag1    | +
        | |      Projections:     | |                         | +
        | |        tag1           | |                         | +
        | |         c2            | |                         | +
        | |                       | |                         | +
        | |       ~1 Rows         | |       ~1 Rows           | +
        |
 |-------------------------------------------------------------| |-----------------
-----------------------------------------| +
        |
(1 row)
```

示例 2：仅显示优化计划且实际执行查询语句。

```
SQL

--- 控制 EXPLAIN 语句的输出仅显示优化计划
```

```
SET explain_output='optimized_only';


--- 执行 EXPLAIN 语句且执行指定的 SELECT 语句
EXPLAIN ANALYZE SELECT t1.c2 FROM t1 JOIN t2 USING (tag1) WHERE t2.c3 > 3.0;
        | |         1 Rows         |                          +
        | |        (0.00s)         |                          +
        | └──────────────────────────┬──────────────────────────┘
+
        |  ┌──────────────────────────┴──────────────────────────┐
+
        | |        HASH_JOIN        |                          +
        | |   ──────────────────────────────────────────────   |
+
        | |     Join Type: INNER    |                          +
        | |                         |                          +
        | |       Conditions:       ├──────────────────────────┐                 +
        | |       tag1 = tag1       |          |               +
        | |                         |          |               +
        | |         1 Rows          |          |               +
        | |        (0.00s)          |          |               +
        | └──────────────────────────┬──────────────────────────┘          |
+
        |
 ┌──────────────────────────┴──────────────────────────┐  ┌──────────────
──────────┴──────────────────────────┐          +
        | |        TABLE_SCAN       | |        TABLE_SCAN       |          +
        | |   ──────────────────────────────────────────────   | |
──────────────────────────────────────────   |          +
        | |        Table: t1        | |        Table: t2        |          +
```

```
        | |  Type: Sequential Scan  | |  Type: Sequential Scan  |          +
        | |                         | |    Projections: tag1    |          +
        | |      Projections:       | |                         |          +
        | |         tag1            | |                         |          +
        | |         c2              | |                         |          +
        | |                         | |                         |          +
        | |        1 Rows           | |        1 Rows           |          +
        | |       (0.00s)           | |       (0.00s)           |          +
        |
 └──────────────────────────────────────────────────────────────┘ └──────────────
──────────────────────────────────┘          +
        |
(1 row)
```

7.10 系统视图

KaiwuDB Lite 提供了一系列系统视图，帮助用户查看数据库的结构信息、运行状态、依赖关系等内容，便于排查问题与系统维护。

视图名称	说明
kaiwudb_settings()	系统配置参数，涵盖内存管理、线程控制、存储设置、安全访问、扩展管理、查询优化、日志记录、文件系统、数据格式和调试选项等各个方面的系统运行配置。
kaiwudb_columns()	数据库中所有表的列信息元数据，包括列的基本属性（名称、数据类型、索引位置）、约束信息（是否可空、默认值）、数值类型精度和字符类型长度等详细的列结构信息。
kaiwudb_constraints()	数据库中所有表约束的详细信息，包括约束类型、涉及的列、约束表达式以及外键引用关系等元数据。
kaiwudb_tables()	数据库中所有表的基本信息和统计数据，包括表名、所属模

	式、表属性（是否临时表、内部表）、结构统计（列数、索引数、约束数）、存储信息（预估大小、是否有主键）以及创建表的 SQL 语句等元数据。
kaiwudb_indexes()	数据库中所有索引的详细信息，包括索引名称、所属表、索引列、索引类型（唯一性、是否主键）以及创建索引的完整 SQL 语句等元数据。
kaiwudb_views()	数据库中所有视图的详细信息，包括视图名称、所属模式、视图属性（是否临时视图、内部视图）、列数统计以及创建视图的完整 SQL 定义，涵盖了系统内置视图、元数据视图和用户自定义视图等所有视图对象。
kaiwudb_schemas()	数据库中所有模式的信息，包括模式名称、所属数据库、模式 OID、注释、标签以及是否为内部模式等属性。
kaiwudb_databases()	数据库实例中所有数据库的基本信息，包括数据库名称、唯一标识符（OID）、存储路径、注释描述、版本标签、数据库类型以及访问权限（是否只读、是否内部数据库）等属性信息。
kaiwudb_dependencies()	数据库对象之间的依赖关系信息，包含对象的类别 ID、对象 ID、子对象 ID 以及所依赖的引用对象信息和依赖类型，用于追踪表、索引、视图等数据库对象之间的关联依赖关系。
kaiwudb_functions()	数据库中所有函数的详细信息，包括函数名称、类型（表函数、标量函数等）、参数定义、返回类型、功能描述、是否有副作用、是否为内部函数以及宏定义等完整的函数元数据信息。
kaiwudb_keywords()	数据库中所有关键字的信息，包括关键字名称和分类类型（reserved 保留字、unreserved 非保留字、type_function 类型函数关键字、column_name 列名关键字），用于识别哪些词汇在 SQL 语句中有特殊含义或使用限制。
kaiwudb_memory()	数据库内存使用情况的详细统计，按不同组件类型（如基础

	表、哈希表、索引、元数据、扩展等）展示当前内存使用量和临时存储使用量，用于监控和分析数据库的内存资源消耗情况。
kaiwudb_optimizers()	数据库查询优化器中所有可用的优化规则名称，包括表达式重写、谓词下推、连接顺序优化、列裁剪、统计信息传播等各种查询优化技术，用于了解系统支持的查询优化能力。
kaiwudb_sequences()	数据库中所有序列对象的详细信息，包括序列名称、起始值、最小值、最大值、增量、是否循环、当前值以及创建序列的 SQL 语句等属性。
kaiwudb_temporary_files()	数据库临时文件的信息，包括临时文件的路径和大小，用于监控数据库在执行查询过程中创建的临时文件使用情况。
kaiwudb_types()	数据库中所有数据类型的详细信息，包括类型名称、类型 OID、存储大小、逻辑类型、类型分类（如数值型、字符串型、日期时间型、布尔型等）等属性
kaiwudb_variables()	数据库中用户定义变量的信息，包括变量名称、当前值和数据类型，用于查看和管理会话或全局范围的自定义变量。

8. 数据管理

8.1 数据导入导出

8.1.1 导入数据

8.1.1.1 导入表级别数据

KaiwuDB Lite 支持使用 `COPY ... FROM`、`CREATE TABLE ... AS FROM`、`CREATE TABLE ... AS SELECT` 语句导入指定表的数据。

- `COPY ... FROM` 语句用于将存储在本地的数据文件（`.csv` 格式）导入到 KaiwuDB

Lite 数据库中的已有表中。

- `CREATE TABLE ... AS FROM` 或 `CREATE TABLE ... AS SELECT` 语句用于在 KaiwuDB Lite 数据库中创建一个新表并将存储在本地或其他服务器上的数据文件（`.csv` 格式）导入到新创建的表中。

> 说明
>
> 将存储在本地数据文件中的数据导入到 KaiwuDB Lite 数据库已有时序表的指定列时，本地数据文件中的所有列必须在 KaiwuDB Lite 数据库已有时序表中有对应的列，否则系统报错。

8.1.1.1.1 语法格式

- 将数据导入已有表
  - 全表导入

```
SQL
COPY <table_name> FROM '<input_file_name>' [(OPTION)];
```

  - 指定列导入

```
SQL
COPY <table_name>(<column_list>) FROM '<input_file_name>' [(OPTION)];
```

> 说明
>
> 为确保与 PostgreSQL 的兼容性，KaiwuDB Lite 也支持不完全符合本文所示语法图的 `COPY ... FROM` 语句。例如，以下语句也是合法的：
>
> ```
> SQL
> COPY  <table_name> FROM '<input_file_name>' WITH DELIMITER '|' CSV
> HEADER;
> ```

- 将数据导入新表
  - 全表导入

```
SQL
CREATE TABLE <table_name> AS FROM read_csv('<input_file_name>');
```

- 指定列导入

```
SQL
CREATE TABLE <table_name> AS SELECT <column_list> FROM
read_csv('<input_file_name>');
```

8.1.1.1.2 参数说明

- table_name：目标表名，数据导入的目标表。
- input_file_name：待导入用户数据文件的名称。目前，KaiwuDB Lite 只支持导入 .csv 格式的文件。待导入的文件名必须使用单引号（'）括起来。导入其他服务器上的表时，采用 http://ip_address:port/xxx.csv 格式。
- column_list：用于指定待导入的列。支持导入多个列。多个列之间使用逗号（,）分隔。
- OPTION：导入参数。导入参数必须使用括号（()）括起来，格式为 <option1> <value1>, option2 value 2 ...。支持指定一个或多个导入参数，各导入参数之间使用逗号（,）隔开。支持以下导入参数：
  - DELIMITER or SEP：可选参数，用于指定分隔符。分隔符必须使用单引号（'）括起来。默认为逗号（,）。导入数据时，指定的分隔符需要与 CSV 文件中使用的分隔符保持一致，否则可能会出现解析后列数不匹配的情况，从而导致数据导入失败。
  - QUOTE：可选参数，用于指定包围符。使用单引号（'）作为包围符时，格式为 '''。使用双引号（"）作为包围符时，格式为 '"'。默认为双引号（"）。包围符不能与分隔符相同。
  - ESCAPE：可选参数，用于指定转义符。使用单引号（'）作为转义符时，格式为 '''。使用双引号（"）作为转义符时，格式为 '"'。默认为双引号（"）。转义符

不能与分隔符相同。

- NULLSTR：可选参数，用于指定空值的表示形式。默认不显示内容，支持指定为 NULL、null、Null 或 \N。
- HEADER：用户指定导入文件是否包含表头。默认为 true（包含表头）。支持指定为：true（包含表头）、false（不包含表头）。

8.1.1.1.3 语法示例

示例 1：将指定表的数据导入已有表

```sql
COPY power FROM 'power_local.csv';
```

示例 2：将指定表的数据导入新表

```sql
CREATE TABLE power1 AS FROM read_csv('power_local1.csv');
```

示例 3：将指定表的数据导入已有表的指定列

```sql
COPY power2(c2) FROM 'power_local2.csv';
```

示例 4：根据列名筛选表数据，并将筛选后的数据导入新表

```sql
CREATE TABLE power3 AS SELECT c1,c2 FROM read_csv('power_local3.csv');
```

示例 5：导入表数据时，指定分隔符为管道符号（|）

```sql
COPY power4 FROM 'power_local4.csv' (DELIMITER '|');
```

示例 6：导入表数据时，指定包围符为单引号（'）

```sql
```

```
COPY power5 FROM 'power_local5.csv' (QUOTE """);
```

示例 7：导入表数据时，指定转义符为双引号（"）

```
sql
COPY power6 FROM 'power_local6.csv' (ESCAPE '"');
```

示例 8：导入表数据时，指定空值表现形式为 NULL

```
sql
COPY power7 FROM 'power_local7.csv' (NULLSTR 'NULL');
```

示例 9：导入表数据时，导出文件包含表头

```
sql
COPY power8 FROM 'power_local8.csv' (HEADER);
```

8.1.1.2 导入库级别数据

IMPORT DATABASE 语句用于将指定 KaiwuDB Lite 数据库中的数据导入到另一个 KaiwuDB Lite 数据库。导入的数据库数据组织形式如下所示：

```
JSON
target_directory
|-- schema.sql
|-- load.sql
|-- t1.csv
...
|-- tn.csv
```

其中：

- schema.sql 文件包含数据库中定义结构的 SQL 语句，例如 CREATE SCHEMA（创建模式）、CREATE TABLE（创建表）等。
- load.sql 文件包含一组 COPY 语句，用于重新从.csv 文件中读取数据。该文件为数

据库模式中的每个表生成一条对应的 COPY 语句。

8.1.1.2.1 语法格式

```
SQL
IMPORT DATABASE '<target_directory>' [(OPTION)];
```

8.1.1.2.2 参数说明

- target_directory：存放待导入数据的文件夹名称。
- OPTION：导入参数。导入参数必须使用括号（()）括起来，格式为 <option1> <value1>, option2 value 2 ...。支持指定一个或多个导入参数，各导入参数之间使用逗号（,）隔开。支持以下导入参数：
  - DELIMITER or SEP：可选参数，用于指定分隔符。分隔符必须使用单引号（'）括起来。默认为逗号（,）。导入数据时，指定的分隔符需要与 CSV 文件中使用的分隔符保持一致，否则可能会出现解析后列数不匹配的情况，从而导致数据导入失败。
  - QUOTE：可选参数，用于指定包围符。使用单引号（'）作为包围符时，格式为 """。使用双引号（"）作为包围符时，格式为 ""。默认为双引号（"）。默认为双引号（"）。包围符不能与分隔符相同。
  - ESCAPE：可选参数，用于指定转义符。使用单引号（'）作为转义符时，格式为 """。使用双引号（"）作为转义符时，格式为 ""。默认为双引号（"）。转义符不能与分隔符相同。
  - NULLSTR：可选参数，用于指定空值的表示形式。默认不显示内容，支持指定为 NULL、null、Null 或 \N。
  - HEADER：用户指定导入文件是否包含表头。默认为 true（包含表头）。支持指定为：true（包含表头）、false（不包含表头）。

8.1.1.2.3 语法示例

示例 1：导入数据库

```sql
IMPORT DATABASE 'ts_db';
```

示例 2：导入数据库时，指定分隔符为管道符号（|）

```sql
IMPORT DATABASE 'ts_db1' (DELIMITER '|');
```

示例 3：导入数据库时，指定包围符为单引号（'）

```sql
IMPORT DATABASE 'ts_db2' (QUOTE '''');
```

示例 4：导入数据库时，指定转义符为双引号（"）

```sql
IMPORT DATABASE 'ts_db3' (ESCAPE '"');
```

示例 5：导入数据库时，指定空值表现形式为 NULL

```sql
IMPORT DATABASE 'ts_db4' (NULLSTR 'NULL');
```

示例 6：导入数据库时，导出文件包含表头

```sql
IMPORT DATABASE 'ts_db5' (HEADER);
```

8.1.2 导出数据

> 说明
>
> 目前，KaiwuDB Lite 仅支持本地导出。

8.1.2.1 导出表级别数据

COPY ... TO 语句用于将数据从 KaiwuDB Lite 导出到外部数据文件（.csv 格式）。通过 COPY ... TO 语句创建的数据文件都可以通过使用 COPY ... FROM 语句复制回数据库。默认情况下，导出的文件存放在 KaiwuDB Lite 的启动目录中。

8.1.2.1.1 语法格式

- 导出全表

```
SQL
COPY <table_name> TO '<target_file_name>' [(OPTION)];
```

- 导出指定列

  用户也可以使用 SELECT ... FROM 语句导出表的指定列。

```
SQL
COPY (SELECT <column_list> FROM <table_name>) TO '<target_file_name>'
[(OPTION)];
```

8.1.2.1.2 参数说明

- table_name：待导出数据的表名。待导出数据的表名必须使用单引号（'）括起来。
- target_file_name：外部文件名，数据导出的目标表。目前，KaiwuDB Lite 只支持导出 .csv 格式的文件。
- column_list：待导出的数据列的名称。支持导出多个列。多个列之间使用逗号（,）分隔。
- OPTION：导出参数。导出参数必须使用括号（()）括起来，格式为 <option1> <value1>, option2 value 2 ...。支持指定一个或多个导出参数，各导出参数之间使用逗号（,）隔开。支持以下导出参数：
  - DELIMITER or SEP：可选参数，用于指定分隔符。分隔符必须使用单引号（'）括起来。默认为逗号（,）。

- QUOTE：可选参数，用于指定包围符。使用单引号（'）作为包围符时，格式为 '''。使用双引号（"）作为包围符时，格式为 '"'。默认为双引号（"）。包围符不能与分隔符相同。
- ESCAPE：可选参数，用于指定转义符。使用单引号（'）作为转义符时，格式为 '''。使用双引号（"）作为转义符时，格式为 '"'。默认为双引号（"）。转义符不能与分隔符相同。
- NULLSTR：可选参数，用于指定空值的表示形式。默认不显示内容，支持指定为 NULL、null、Null 或 \N。
- HEADER：用户指定导出文件是否包含表头。默认为 true（包含表头）。支持指定为：true（包含表头）、false（不包含表头）。

8.1.2.1.3 语法示例

以下示例假设已创建 power 表并写入相关数据。

示例 1：导出表

```
sql
COPY power TO 'power_local.csv';
```

示例 2：根据列名筛选表数据，并导出筛选后的数据

```
sql
COPY (SELECT c1, c2 FROM power) TO 'power_local1.csv';
```

示例 3：导出表数据时，指定分隔符为管道符号（|）

```
sql
COPY power TO 'power_local2.csv' (DELIMITER '|');
```

示例 4：导出表数据时，指定包围符为单引号（'）

```
sql
COPY power TO 'power_local3.csv' (QUOTE '''');
```

示例 5：导出表数据时，指定转义符为双引号（"）

```sql
COPY power TO 'power_local4.csv' (ESCAPE '"');
```

示例 6：导出表数据时，指定空值表现形式为 NULL。

```sql
COPY power TO 'power_local5.csv' (NULLSTR 'NULL');
```

示例 7：导出表数据时，导出文件包含表头。

```sql
COPY power TO 'power_local6.csv' (HEADER);
```

8.1.2.2 导出库级别数据

KaiwuDB Lite 支持一次性导出指定数据库中所有的内容到指定文件夹。默认情况下，导出的数据库文件存放在 KaiwuDB Lite 的启动目录中。导出的数据库数据组织形式如下所示：

```JSON
target_directory
|-- schema.sql
|-- load.sql
|-- t1.csv
...
|-- tn.csv
```

其中，

- schema.sql 文件包含数据库中定义结构的 SQL 语句，例如 CREATE SCHEMA（创建模式）、CREATE TABLE（创建表）等。
- load.sql 文件包含一组 COPY 语句，用于重新从.csv 文件中读取数据。该文件为数

据库模式中的每个表生成一条对应的 COPY 语句。

8.1.2.2.1 语法格式

```
SQL
EXPORT DATABASE '<target_directory>' [(OPTION)];
```

8.1.2.2.2 参数说明

- target_directory：存放导出数据的文件夹名称。
- OPTION：导出参数。导出参数必须使用括号（()）括起来，格式为 <option1> <value1>, option2 value 2 ...。支持指定一个或多个导出参数，各导出参数之间使用逗号（,）隔开。支持以下导出参数：
  - DELIMITER or SEP：可选参数，用于指定分隔符。分隔符必须使用单引号（'）括起来。默认为逗号（,）。
  - QUOTE：可选参数，用于指定包围符。使用单引号（'）作为包围符时，格式为 '''。使用双引号（"）作为包围符时，格式为 '"'。默认为双引号（"）。包围符不能与分隔符相同。
  - ESCAPE：可选参数，用于指定转义符。使用单引号（'）作为转义符时，格式为 '''。使用双引号（"）作为转义符时，格式为 '"'。默认为双引号（"）。转义符不能与分隔符相同。
  - NULLSTR：可选参数，用于指定空值的表示形式。默认不显示内容，支持指定为 NULL、null、Null 或 \N。
  - HEADER：用户指定导出文件是否包含表头。默认为 true（包含表头）。支持指定为：true（包含表头）、false（不包含表头）。

8.1.2.2.3 语法示例

示例 1：导出数据库

```sql
EXPORT DATABASE 'tsdb';
```

示例 2：导出数据库时，指定分隔符为管道符号（|）

```sql
EXPORT DATABASE 'tsdb1' (DELIMITER '|');
```

示例 3：导出数据库时，指定包围符为单引号（'）

```sql
EXPORT DATABASE 'tsdb2' (QUOTE '''');
```

示例 4：导出数据库时，指定转义符为双引号（"）

```sql
EXPORT DATABASE 'tsdb3' (ESCAPE '"');
```

示例 5：导出数据库时，指定空值表现形式为 NULL

```sql
EXPORT DATABASE 'tsdb4' (NULLSTR 'NULL');
```

示例 6：导出数据库时，导出文件包含表头

```sql
EXPORT DATABASE 'tsdb5' (HEADER);
```

8.2 无模式写入

8.2.1 PostgreSQL JDBC

KaiwuDB Lite 支持调用 PostgreSQL JDBC 标准接口将数据无模式写入数据库。用户只需预先创建标准统一的表结构（k_timestamp,tags,cols），写入数据数据过程中不再需要增加、删除列操作。KaiuwDB Lite 采用 JSON 格式存储标签数据和用户数据。

标签列和数据列分别定义为 MAP 和 JSON 类型。

以下示例创建一个名为 cpu 的表并调用 PostgreSQL JDBC 标准接口将数据无模式写入 cpu 表。

```
sql
-- 1. 创建统一标准的表结构
create table cpu(k_timestamp Timestamp,tags MAP(VARCHAR,VARCHAR), cols
json);
-- 字段解释: tags 由于 InfluxDB 的无模式写入都为 varchar 类型。统一 tags 定义为
VARCHAR 类型
--        cols 定义为 JSON 格式。



-- 2. 直接调用 JDBC 接口写入 JSON 和 MAP 格式数据,写入标签列的 hostname 和数据
列的 e1。
String insertQuery = "INSERT Into cpu(k_timestamp,tags,cols) VALUES ('2025-03-31
12:00:00',MAP(['hostname'],['host_1']),'{\"e1\":8}')";
PreparedStatement preparedStatement =
connection.prepareStatement(insertQuery);
int rowsAffected = preparedStatement.executeUpdate();
-- 上述写入等价于 InfluxDB 无模式类型 cpu,hostname=host_1 e1=8 timestamp


-- 3. 查询无模式写入的数据标签。
admin=> select tags->'hostname',cols->>'e1' from cpu;
 ("tags" -> 'hostname') | (cols ->> 'e1')
-----------------------+-----------------
 "host_1"              | 8
(1 row)
```

```
-- 4. 增加标签列和数据列列。
String insertQuery = "INSERT Into cpu(k_timestamp,tags,cols) VALUES ('2025-03-31
12:00:01',MAP(['hostname','region'],['host_2','shanghai']),'{\"e1\":10,\"e2\":1.0101}')"
;
PreparedStatement preparedStatement =
connection.prepareStatement(insertQuery);
int rowsAffected = preparedStatement.executeUpdate();
-- 上述写入等价于无模式类型 cpu,hostname=host_2,region=shanghai
e1=10,e2=1.0101 timestamp

-- 5. 查询数据
admin=> select tags->'hostname',tags->'region',cols->>'e1',cols->>'e2' from cpu;
("tags" -> 'hostname') | ("tags" -> 'region') | (cols ->> 'e1') | (cols ->> 'e2')
-----------------------+---------------------+----------------+----------------
 "host_1"              |                     | 8              |
 "host_2"              | "shanghai"          | 10             | 1.0101
(2 rows)
```

8.2.2 Telegraf

Telegraf 是一款基于插件化的开源指标收集工具。KaiwuDB Lite 提供了专用的输出插件，可直接将 Telegraf 采集的监控数据写入数据库，无需预先定义数据结构或模式。

用户完成 Telegraf 配置文件设置后，KaiwuDB Lite 会根据配置的数据模式自动创建对应的数据表。

KaiwuDB Lite 支持两种数据存储模式：

紧凑模式（默认）：

表结构采用固定的三列设计：

- 时间戳列（默认名 ts，TIMESTAMP 类型）：存储采集时间

- 标签列（默认名 tags，MAP 类型）：以 JSON 格式存储设备、服务等维度信息
- 数据列（默认名 fields，JSON 类型）：以 JSON 格式存储所有监控指标的数值

表结构示例：

```
SQL
CREATE TABLE cpu_metrics (
  ts TIMESTAMP,
  tags MAP(VARCHAR, VARCHAR),  -- {"host": "server1", "region": "us-east"}
  fields JSON                  -- {"cpu_usage": 75.5, "load_avg": 1.2}
);
```

紧凑模式支持的完整配置示例，参见 sample.conf。

标准模式：

为每个标签和字段创建单独的列，适用于标签和字段相对固定、需要高查询性能和列级别索引的场景。

表结构示例：

```
SQL
CREATE TABLE cpu_metrics (
  ts TIMESTAMP,
  host TEXT,          -- 每个 tag 一个列
  region TEXT,
  service TEXT,
  cpu_usage REAL,     -- 每个 field 一个列
  load_avg REAL,
  memory_usage BIGINT
);
```

标准模式支持的完整配置示例，参见 sample_full.conf。

本节以紧凑模式为例介绍如何通过 Telegraf 连接 KaiwuDB Lite 并完成数据写入。

8.2.2.1 前提条件

- 安装并启动 KaiwuDB Lite 数据库。有关详细信息，参见安装部署。
- 创建具有表级别及以上操作权限的用户。有关详细信息，参见创建用户。
- 已获取 Telegraf KaiwuDB-Lite 输出插件并根据编译说明完成编译。

8.2.2.2 配置连接

1. 在 Telegraf 配置文件，添加 [[outputs.kaiwudb_lite]] 区域，配置 KaiwuDB Lite 驱动和连接信息。

```
TOML
[[outputs.kaiwudb_lite]]
 ## 数据库驱动名（必须为 kaiwudb-lite）
 # driver = "kaiwudb-lite"

 ## 数据库连接配置，兼容 psql 协议
 data_source_name = "host=192.168.10.123 port=36257 user=admin
connect_timeout=5"

 ## 启用紧凑模式架构（推荐）
 # enable_compact_schema = true

 ## 可选配置参数
 # timestamp_with_time_zone = false
 # timestamp_column = "ts"
 # tags_column_name = "tags"
 # fields_column_name = "fields"

 ## 初始化 SQL
```

```
# init_sql = ""


## 连接池配置
# connection_max_idle_time = "0s"
# connection_max_lifetime = "0s"
# connection_max_idle = 2
# connection_max_open = 0
```

参数说明：

参数	说明	默认值	必选/可选
driver	数据库驱动名，必须设置为 kaiwudb-lite	-	可选
data_source_name	数据库连接配置，兼容 psql 协议。格式为"host=IP port=<port> user=<user_name> connect_timeout=<duration>"，例如："host=192.168.10.123 port=36257 user=admin connect_timeout=5" 默认端口为 36257，默认用户为 admin，默认无密码	-	必选
enable_compact_schema	是否启用紧凑模式	true	可选
timestamp_with_ti	是否使用带时区信息的时间	false	可选

me_zone	戳。存储统一为 UTC，查询时根据时区设置展示		
timestamp_column_name	时间戳列名，对应表时的第一列，数据类型为 TIMESTAMP	ts	可选
timestamp_column	时间戳列名	"ts"	可选
tags_column_name	（仅紧凑模式）标签列名，对应建表时的标签列，以 MAP（VARCHAR， VARCHAR）类型存储维度信息和元数据标签	"tags"	可选
fileds_column_name	（仅紧凑模式）字段列名，对应建表时的数据列，以 JSON 类型存储实际的指标数值	"fields"	可选
init_sql	初始化 SQL 语句	""	可选
connection_max_idle_time	连接池中连接的最大空闲时间，"0s" 表示永不过期	"0s"	可选
connection_max_lifetime	连接的最大生存时间（无论是否被使用）。超过此时间的连接将被强制关闭。 "0s" 表示永不过期	"0s"	可选
connection_max_idle	连接池中允许的最大空闲连接数。超过此数量的空闲连接将被关闭。设置为 0 表示不限制	2	可选
connection_max_open	数据库的最大打开连接数（包括活跃和空闲连接）。达到限制时，新操作需等待可用连接。设置为 0 表示不限制	0	可选

2. （可选）根据需要在配置文件末尾添加 [outputs.kaiwudb.convert]设置数据类型转换。数据类型转换的左侧为 Telegraf 的数据类型，右侧为目标数据库数据类型。注意：目标数据类型必须是 KaiwuDB Lite 支持的有效数据类型。

```
TOML
...
 [outputs.kaiwudb.convert]
   integer          = "INT"
   uinteger         = "UINTEGER"
   bigint           = "BIGINT"
   ubigint          = "UBIGINT"
   real             = "REAL"
   double           = "DOUBLE"
   text             = "TEXT"
   timestamp        = "TIMESTAMP"
   timestamptz      = "TIMESTAMP WITH TIME ZONE"
   defaultvalue     = "TEXT"
   unsigned         = "UNSIGNED"
   bool             = "BOOL"
   json             = "JSON"
   blob             = "BLOB"
```

3. 启动 Telegraf。

```
Bash
./telegraf --config <config_path> [--debug]
```

参数说明：

- --config <config_path>：指定 Telegraf 配置文件的完整路径。
- --debug：（可选）启用调试模式，以输出详细的运行日志，便于排查问题

4. （可选） 登录数据库，检查数据是否已成功写入。

8.2.2.3 配置示例

以下示例展示了一个完整的 CPU 监控配置，Telegraf 将每秒采集 CPU 数据并每 2 秒批量写入 KaiwuDB Lite。此配置会自动创建名为 cpu 的表，包含 ts（时间戳）、tags（如 host 信息）、fields（CPU 使用率等指标）三列。

这一配置显式指定了紧凑模式下所有可选参数和数据类型转换规则，确保数据类型的精确映射。

```
TOML
# Telegraf 代理配置段
[agent]
 debug = true            # 开启调试日志，用于排查问题和监控运行状态
 interval = "1s"         # 采样频率：每秒采集一次数据
 flush_interval = "2s"      # 写入间隔：每 2 秒将数据批量写入到输出目标
 metric_batch_size = 800     # 批处理大小：每批处理 800 条指标数据
 metric_buffer_limit = 10000  # 缓存限制：内存中最大缓存 10000 条指标，防止内存溢出

# 执行外部程序采集数据的输入插件
[[inputs.execd]]          # 从外部程序/脚本采集数据的插件
 command = ["./test_insert_speed.sh"] # 执行的命令，这里是一个测试插入速度的脚本
 data_format = "influx"      # 数据格式为 InfluxDB 行协议格式

# CPU 监控输入插件配置
[[inputs.cpu]]
 percpu = true           # 采集每个 CPU 核心的详细数据
 totalcpu = true         # 采集 CPU 总体使用情况数据
 collect_cpu_time = false    # 不采集 CPU 时间相关指标
```

```
  report_active = false      # 不报告 CPU 活跃状态
  core_tags = false          # 不为每个核心添加额外的标签信息

# KaiwuDB-Lite 数据库输出插件配置
[[outputs.kaiwudb_lite]]
  ## 数据库驱动名（必须为 kaiwudb-lite）
  driver = "kaiwudb-lite"

  ## 数据库连接配置，兼容 PostgreSQL 协议
  data_source_name = "host=192.168.10.123 port=36257 user=admin
connect_timeout=5"

  ## 启用紧凑模式架构 - 使用 3 列结构
  enable_compact_schema = true

  ## 时间戳配置
  timestamp_with_time_zone = false

  ## 时间列配置
  timestamp_column = "ts"

  ## 紧凑模式列名配置
  tags_column_name = "tags"       # 标签存储为 JSON/MAP 对象
  fields_column_name = "fields"   # 字段存储为 JSON 对象

## 数据类型转换配置
[outputs.kaiwudb.convert]
  integer           = "INT"
  uinteger          = "UINTEGER"
```

```
bigint          = "BIGINT"
ubigint         = "UBIGINT"
real            = "REAL"
double          = "DOUBLE"
text            = "TEXT"
timestamp       = "TIMESTAMP"
timestamptz     = "TIMESTAMP WITH TIME ZONE"
defaultvalue    = "TEXT"
unsigned        = "UNSIGNED"
bool            = "BOOL"
json            = "JSON"
blob            = "BLOB"
```

8.2.2.4 故障排查

下表列出了 Telegraf 集成 KaiwuDB Lite 时常见的错误类型及对应的解决方案：

错误信息	错误类型	解决方案
`Error running agent: unknown driver "KaiwuDB-LITE"`	驱动名称错误	确保 `driver` 参数值为 `"kaiwudb-lite"`（全小写）
`Table "cpu" does not have a column with name "ts"`	表结构不匹配	检查数据库表结构是否包含配置的列名
`dial tcp 127.0.0.1:3626: connect: connection refused`	连接被拒绝	检查端口配置是否正确
`timeout: dial tcp 192.168.1.1:36260: i/o timeout`	连接超时	验证网络连通性，检查 IP 地址和端口配置
`User with name root does not exist!`	用户不存在	确认用户名是否正确，确保用户已在数据库中创建
`database `` does not exist`	未指定数	选择以下任一方案：

	据库名称，或指定的数据库不存在。	• 在连接字符串中指定数据库名，即在 data_source_name 中添加 database=metadata • 在数据库启动配置文件中设置 ignore_db_noexists = true

8.3 批量写入

KaiwuDB Lite 的 Appender 功能专门用于优化数据写入性能。通过自定义程序接口和批量写入机制，该功能能够显著提升数据写入效率。

Appender 通过以下方式实现高性能数据写入：

- 批量处理：采用数据块创建、批量写入和批量传输的方式优化性能
- 容错机制：当服务端发生中断时，系统会自动进行分批传输，确保已完成传输的批次数据能够正确写入
- 连接绑定：每个 Appender 绑定到特定数据库连接，追加数据时使用该连接的事务上下文
- 单表操作：每个 Appender 实例只能向数据库中的一张表追加数据

> 说明
>
> Appender 功能仅支持程序调用，不支持 psql 客户端和嵌入式部署环境。

本节将介绍如何使用 Appender 功能连接 KaiwuDB Lite 并完成数据批量写入操作。

8.3.1 前提条件

- 已安装和启动 KaiwuDB Lite 数据库。
- 已经创建需要写入数据的表，且表结构已定义。
- 用户具有相应的数据库访问和数据写入权限。

8.3.2 配置步骤

步骤

1. 创建连接对象，包含必要的连接信息：

```
C++
const char *conninfo;

// 包含密码的连接字符串
conninfo = "host=localhost user=admin password=123456 port=36257";
// 无密码的连接字符串
// conninfo = "host=localhost user=admin port=36257";

std::string table_name = "t2";
kaiwudb::Connection conn(conninfo);
```

2. 为目标表创建创建 Appender 对象：

```
C++
kaiwudb::Appender ap(conn, table_name);
```

3. 使用 Appender 接口进行数据写入：

```
C++
int rows = 100;
int cols = 10;

for (int i = 0; i < rows; i++) {
  ap.BeginRow();
  for (int j = 0; j < cols; j++) {
    ap.Append(std::to_string(i).c_str());
  }
```

```
    ap.EndRow();
}

// 关闭 Appender，提交所有数据
ap.Close();
```

8.3.3 配置示例

以下示例演示了如何连接数据库并向表中写入数据

```
C++
#include <stdio.h>
#include <stdlib.h>
#include <string>
#include <iostream>
#include <cstring>
#include <string>
#include <arpa/inet.h>
#include <unistd.h>
#include "kaiwudb/main/connection.hpp"
#include "kaiwudb/main/appender.hpp"
#include <chrono>

const char* ExecStatusTypeToString(ExecStatusType c) {
    switch(c) {
      #define ENUM_ITEM(x) case ExecStatusType::x: return #x;
      ENUM_ITEM(PGRES_BAD_RESPONSE);
      ENUM_ITEM(PGRES_COMMAND_OK);
      ENUM_ITEM(PGRES_COPY_BOTH);
      ENUM_ITEM(PGRES_COPY_IN);
```

```
        ENUM_ITEM(PGRES_COPY_OUT);
        ENUM_ITEM(PGRES_EMPTY_QUERY);
        ENUM_ITEM(PGRES_FATAL_ERROR);
        ENUM_ITEM(PGRES_NONFATAL_ERROR);
        ENUM_ITEM(PGRES_SINGLE_TUPLE);
        ENUM_ITEM(PGRES_TUPLES_OK);
        #undef ENUM_ITEM
        }
  };
int main(int argc, char **argv) {
  const char *conninfo;
  if (argc > 1)
    conninfo = argv[1]; // 命令行参数作为连接字符串
  else
    conninfo = "host=localhost dbname=metadata user=admin port=36257";
  std::string table_name="t2";
  try{
  kaiwudb::Connection conn(conninfo);
  kaiwudb::Appender ap(conn,table_name);
  int col=10;
  for (int i=0;i<100000;i++){
    ap.BeginRow();
    for (int j=0;j<col;j++){
      ap.Append(std::to_string(i).c_str());
    }
    ap.EndRow();
  }
  ap.Close();
  }catch(std::exception &e){
```

```
        std::cerr<<e.what()<<"\n";
    }
    return 0;
}
```

8.3.4 故障排查

下表列出了使用 Appender 功能写入数据时常见的错误类型及对应的解决方法：

错误信息	错误类型	解决方法
Connection Failed for missing "=" after "aa" in connection info string	连接字符串错误	检查并修正连接字符串格式，确保包含正确的键值对格式
connection to server at "localhost", port 1234 failed: Connection refused	目标 IP/Port 不可达	检查服务器是否运行，确认端口号是否正确（默认端口 36257），检查网络连接
Catalog Error: User with name u1 does not exist!	目标用户不存在	检查用户名是否存在，如不存在需先创建用户
Invalid password for user u1	目标用户密码错误	验证并提供正确的用户密码
Catalog Error: Table with name t5 does not exist!	表不存在	检查表名是否正确，确认表是否存在，或创建所需的表
Too many appends for chunk!	表列数不匹配	检查数据插入操作，确保表列数与插入数据列数匹配
server closed the connection unexpectedly	消息传输过程中断（服务端中断）	检查服务器状态，确保网络连接稳定；数据传输中断时，已完成批次的数据会被保存

8.3.5 常用接口说明

下表列出了 Appender 功能的常用接口。

接口名称	功能描述
Appender(Connection &con, const string &table_name)	通过现有连接创建 Appender 对象
Appender(Connection &con, const string &schema_name, const string &table_name)	指定 schema 创建 Appender 对象
Appender(Connection &con, const string &database_name, const string &schema_name, const string &table_name)	指定数据库和 schema 创建 Appender 对象
void BeginRow()	定义行数据开始
void EndRow()	定义行数据结束
void AppendRow(ARGS... args)	定义添加一行数据
void Close()	结束写入并销毁对象
void AppendDataChunk(DataChunk &value)	批量写入 DataChunk 数据

8.4 批量提交

KaiwuDB Lite 支持通过 group_commit_size 参数实现批量提交功能。在 PostgreSQL Extended Query Protocol 模式下，该参数可将一个 Session （会话）中的多个独立提交操作合并为一次批量提交，从而提升写入性能。该参数为 Session 级别，仅对当前会话生效。

```
Shell
SET group_commit_size = <value>;
```

参数说明：

- value：批量提交的大小设置
  - 0：关闭批量提交功能

- >0：开启批量提交功能，数值表示合并的提交数量。建议根据实际业务场景调整设置值，调大参数值有助于降低系统调用开销，提高批处理效率，但会延长提交延迟，影响实时性要求高的业务；调小参数值可以提升响应速度，但可能降低整体性能。

示例

示例 1：设置批量提交大小为 1000

```
SQL
SET group_commit_size = 1000;
```

示例 2：关闭批量提交功能

```
SQL
SET group_commit_size = 0;
```

8.5 生命周期管理

KaiwuDB Lite 提供生命周期管理功能，可定期清除过期数据，释放磁盘空间，提高数据库资源利用率。

生命周期管理功能默认关闭，用户可通过数据库配置文件（kaiwudb-lite.conf）定义模式（schema） 和表的生命周期规则，以及生命周期任务的调度周期，通过重启数据库或 SQL 语句启用生命周期管理任务。

数据库在计划内或意外停止后重启时，生命周期管理任务将自动恢复运行。

用户可通过 SQL 语句查询和删除当前执行的生命周期管理任务。

> 注意
>
> - 在数据库运行过程中修改生命周期配置后，配置将在下次调度任务执行时生效。
> - 使用 SQL 语句启用或更新生命周期管理任务时，新的设置会立即生效。

8.5.1 设置生命周期和调度周期

步骤

1. 打开自定义配置文件 kaiwudb-lite.conf 文件，配置模式和表的生命周期以及调度周期。

> 说明
>
> - 表级别的生命周期配置优先于模式级别的配置。
> - 同级别配置时，后续策略会覆盖先前策略。

示例：

```
Plain Text
...
# [retention]

# 配置指定模式下所有表的生命周期
schemas_reg=[main.*,  schema7.*, schema3.*, schema4.*]=[5 seconds]
schemas_reg=[schema5.*, schema6.*]=[10 seconds]

# 配置指定表的生命周期
schema.tables=[schema6.table4, schema6.table5, schema6.table6]=[30
seconds]
schema.tables=[schema9.table1]=[1 years]
schema.tables=[schema9.table2]=[1 months]

# 配置生命周期任务的调度周期
cronTask=[*/5 * * * * *]
```

参数说明：

- schema_reg：配置指定模式下所有表的生命周期，格式为 <schema_name>.*。
  - 支持指定一个或多个模式，模式之间使用逗号 （,）分隔。

- 支持的时间单位包括：年（years）、月（months）、日（days）、周（weeks）、小时（hours）、分钟（minutes）、秒（seconds）、毫秒（milliseconds）。

- schema.table：配置指定模式下指定表的生命周期，格式为<schema_name>.<table_name>。
  - 支持指定一个或多个表，表名之间使用逗号（,）分隔。
  - 支持的时间单位与 schemas_reg 相同。
- cronTask：定义生命周期任务的调度周期，采用 Cron 表达式，格式如下：

```
Plain Text
秒 分 时 日 月 周
```

字段	取值范围	说明
秒（Seconds）	0-59	秒
分（Minutes）	0-59	分钟
时（Hours）	0-23	小时，0 表示午夜 12 点
日（Day of month）	1-31	日期
月（Month）	1-12	月，也可使用英文缩写（JAN-DEC）
周（Day of week）	0-7	0 和 7 表示周日，也可使用英文缩写（SUN-SAT）

Cron 表达式支持以下特殊字符：

符号	描述
*	匹配所有值，例如，* 在 “分” 字段中表示每分钟执行一次。
-	指定范围，例如， 9-17 在 “时” 字段表示任务会在 9 点到 17 点之间执行。

,	列举多个不连续的值。例如，在 “周” 字段使用 MON,WED,FRI，表示任务会在周一、周三和周五执行。
/	设定步长，例如，*/5 在 “分” 字段表示每 5 分钟触发一次任务；2/5 在 “分” 字段表示从第 2 分钟开始，每 5 分钟触发一次任务。
?	用于 “日” 和 “周” 字段，表示不指定该字段，用来避免冲突。例如，0 0 12 ? * MON-FRI 表示在工作日（周一到周五）的中午 12 点触发任务。
L	用于 “日” 和 “周” 字段，表示最后一天。L 在 “日” 字段中表示该月的最后一天；L 在 “周” 字段中表示该月的最后一个星期几，例如，6L 表示该月最后一个星期五（6 代表星期五）。
W	指定离给定日期最近的工作日。例如，15W 表示离该月 15 号最近的工作日，如果 15 号是周六，则任务会在 14 号（周五）触发；如果 15 号是周日，则任务会在 16 号（周一）触发。

8.5.2 启用或更新生命周期管理任务

KaiwuDB Lite 支持用户通过 SQL 语句启用或更新生命周期管理任务，用户也可以通过重启数据库使配置生效。

前提条件

- 已完成生命周期和调度周期设置。

步骤

1. 如需启用或更新生命周期管理任务：

```
SQL
SELECT cron('call kaiwudb_retention()', '<cron_expr>');
```

参数说明：

- cron_expr：Cron 表达式，格式说明参见设置生命周期和调度周期章节。如果 SQL 语句的设置与 kaiwudb-lite.conf 中的设置不一致，系统将以 SQL 语句的

设置为准。

示例：

```
SQL
SELECT cron('call kaiwudb_retention()', ' */20 * * * * *');
```

8.5.3 查询生命周期管理任务

步骤

1. 如需查询正在执行的生命周期管理任务：

```
SQL
SELECT * FROM cron_jobs();
```

返回示例：

```
Plain Text
 job_id |       query       |  schedule  |       next_run        | status |       last_run        | last_result
--------+---------------------------+----------------+-------------------------+--------+-----------------------+-------------
 task_0 | SELECT * from retention() |  */20 * * * * | Wed Mar  5 08:07:40 2025 | Active | Wed Mar  5 08:07:20 2025 |  Success
(1 row)
```

8.5.4 删除生命周期管理任务

KaiwuDB Lite 支持用户通过 SQL 语句删除正在执行的生命周期管理任务。

> 提示
>
> - 使用 SQL 语句删除任务后，数据库在计划内或意外停止后重新启动时，生命周期管理任务将自动恢复运行。

> • 如需永久关闭生命周期管理功能，需要注释掉 kaiwudb-lite.conf 文件中的 cronTask 设置，然后重启数据库。

步骤

1. 如需删除正在执行的生命周期管理任务：

```
SQL
SELECT cron_delete('<job_id>');
```

参数说明：

- job_id：生命周期管理任务 ID， 可通过 SELECT * FROM cron_jobs(); 语句获取。

示例：

```
Plain Text
SELECT cron_delete('task_0');
```

8.6 数据压缩

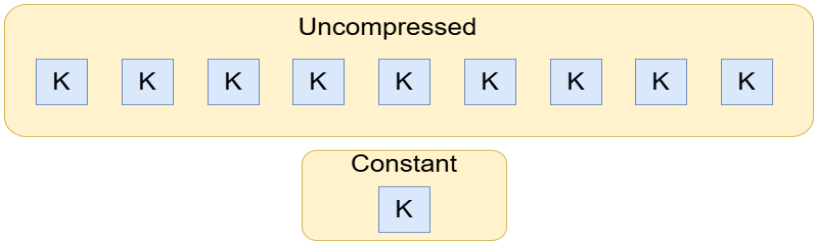
KaiwuDB Lite 采用列存方式存储数据。写入 KaiwuDB Lite 的数据按列进行存储。KaiwuDB Lite 基于列的数据类型，对于不同列的数据采用不同的压缩算法。当数据写入到 KaiwuDB Lite 时，KaiwuDB Lite 会将表拆分为不同的 RowGroup（行组）。每个 RowGroup 存储 120K 条记录。RowGroup 里面的数据按列存储在 ColumnSegment（列段）中。每个 ColumnSegment 由固定大小的 Block（数据块）组成。

KaiwuDB Lite 支持即时数据压缩。首先，KaiwuDB Lite 扫描 ColumnSegment 中的数据，判断 ColumnSegment 的数据采用的压缩算法。然后，KaiwuDB Lite 再对 ColumnSegment 中数据进行压缩，并将压缩后的数据按 Block 组织写入磁盘，进行存储。如果在压缩过程中，系统找不到对应类型的数据，KaiwuDB Lite 不对该列数据进行压缩。

KaiwuDB Lite 支持以下压缩算法：

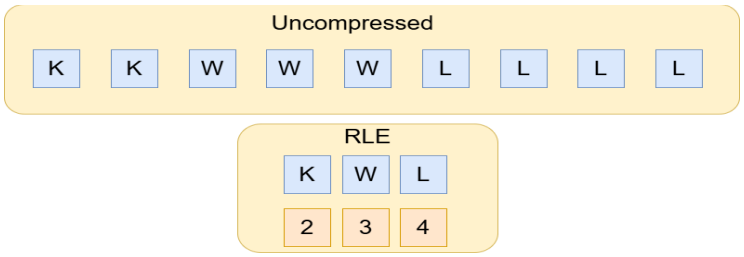
- Constant Encoding（常量编码）

常量编码用于列段中的每个值都是相同值的情况。在这种情况下，KaiwuDB Lite 只存储单个值。例如，列中可能填有 NULL 值，或者其值很少变化（如表中的 year 列）。



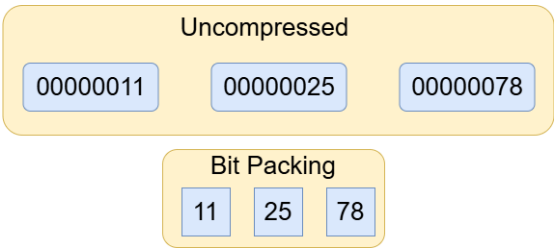
- RLE（Run-Length-Encode，行程长度编码）

RLE 是一种利用数据集中重复值的压缩算法。数据集不是存储单个值，而是分解成一对（value，count）元组，其中 count 表示值的重复频率。



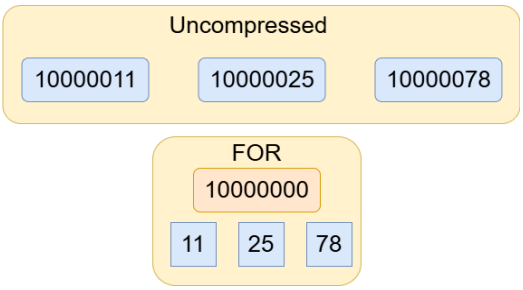
- Bit Packing

Bit Packing 压缩算法利用整数值很少跨越其数据类型的整个范围这一事实。例如，四字节整数值可以存储从负 20 亿到正 20 亿的值。通常，系统不会使用此数据类型的整个范围，而只存储少量数据。Bit packing 通过在存储值时删除所有不必要的前导零来对数据进行压缩。



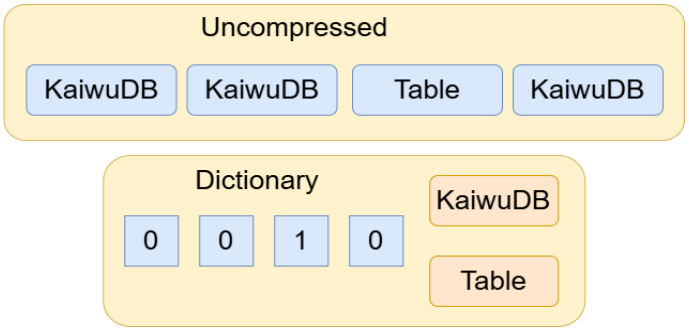
- FOR （Frame of Reference）

FOR 编码是 BitPacking 的扩展，其中还包括一个 Frame（帧）。Frame 是在数值集中找到的最小值。这些值以该帧的偏移量形式存储。这对于存储时间戳列的数据作用很大。



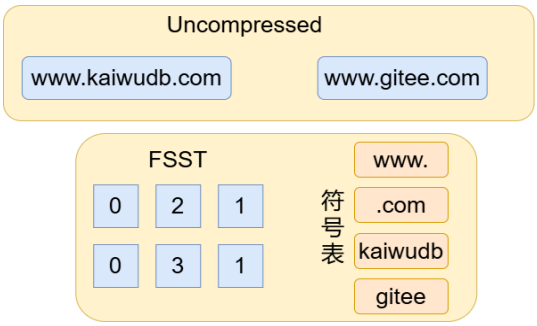
- Dictionary（字典编码）

字典编码的工作原理是将公共值提取到一个单独的字典中，然后用对该字典的引用替换原始值。



- FSST（Fast Static Symbol Table，快速静态符号表）

FSST 是字典编码的扩展。FSST 不仅提取整个字符串的重复项，还提取字符串内的重复项。这在存储本身是唯一但在字符串中有很多重复的字符串（如 URL 或电子邮件地址）时非常有效。



- Chimp & ALP（Adaptive Lossless floating-Point Compression，自适应浮点无损压缩）

Chimp 是一种非常新的压缩算法，旨在压缩浮点值。Chimp 基于 Gorilla 压缩。当浮点值一起进行 XOR 运算时，似乎会产生具有许多带有尾随零和前导零的值。Gorilla 和

Chimp 的工作原理是努力寻找一种有效的方法来存储尾随零和前导零。

9. 系统管理

9.1 用户管理

9.1.1 创建用户

CREATE USER 语句用于创建新用户，并为新用户设置登录密码和密码有效期。

9.1.1.1 语法格式

```
SQL
CREATE USER <user_name> WITH PASSWORD [<password> | NULL ] [VALID UNTIL
<timestamp>];
```

9.1.1.2 参数说明

- user_name：待创建的用户名。用户名不区分大小写，必须以字母或下划线（_）开头，只支持字母，数字或下划线（_）。
- PASSWORD <password>：设置用户密码，也支持使用 NULL 值。设置该选项的用户可以使用密码安全访问节点。密码必须采用字符串的形式，并使用单引号（'）将密码括起来。
- VALID UNTIL：可选关键字。设置密码有效期，也支持使用 NULL 值。设置的时间支持 timestamp 格式，也支持精确到毫秒级别。如果设置的为某一天，默认为 0 时 0 分 0 秒。设置时需使用单引号（'）将密码有效期括起来。到达指定日期或时间后，密码失效。失效后，系统拒绝使用该用户创建的新的连接，但已创建的连接不受影响。如未指定，则默认密码有效期为 infinity，即永不失效。

9.1.1.3 语法示例

以下示例创建 user1 用户，并为该用户设置登录 KaiwuDB Lite 服务器的密码和密码有效期。

```
SQL
CREATE USER user1 WITH PASSWORD '11aa!!AA' VALID UNTIL '2023-01-01
00:00:00+00:00';
```

9.1.2 修改用户

ALTER USER 语句用于更改用户的登录密码和密码有效期。

> 说明
>
> KaiwuDB Lite 不支持修改 admin 用户的密码有效期。

9.1.2.1 语法格式

- 修改用户密码

```
SQL
ALTER USER <user_name> WITH PASSWORD [<password> | NULL];
```

- 修改密码有效期

```
SQL
ALTER USER <user_name> WITH VALID UNTIL <timestamp>];
```

9.1.2.2 参数说明

- user_name：待修改的用户名。
- PASSWORD <password>：设置用户密码，也支持使用 NULL 值。设置该选项的用户可以使用密码安全访问节点。密码必须采用字符串的形式，并使用单引号（'）将密码括起来。
- VALID UNTIL：设置密码有效期，也支持使用 NULL 值。设置的时间支持

timestamp 格式，也支持精确到毫秒级别。如果设置的为某一天，默认为 0 时 0 分 0 秒。到达指定日期或时间后，密码失效。设置时需使用单引号（'）将密码有效期括起来。

9.1.2.3 语法示例

以下示例假设已创建 user1 用户，并为用户设置登录密码和密码有效期。

示例 1：修改 user1 用户的登录密码

```sql
ALTER USER user1 WITH PASSWORD '22aa!!bb';
```

示例 2：修改 user1 用户的密码有效期

```sql
ALTER USER user1 WITH VALID UNTIL '2025-01-01 00:00:00+00:00';
```

9.1.3 删除用户

DROP USER 语句用于删除除 admin 用户以外的任意用户（包括该连接使用的用户）。当删除正在建立连接的用户时：

- 如果删除用户操作早于用户认证操作，则建立连接失败。
- 如果删除用户操作晚于用户认证操作，则正常建立连接。

9.1.3.1 语法格式

```SQL
DROP USER [IF EXISTS] <user_name> [CASCADE | RESTRICT];
```

9.1.3.2 参数说明

- IF EXISTS：可选关键字。当使用 IF EXISTS 关键字时，如果目标用户存在，系统删除目标用户。如果目标用户不存在，系统删除用户失败，但不会报错。当未使用 IF

EXISTS 关键字时，如果目标用户存在，系统删除用户。如果目标用户不存在，系统报错，提示目标用户不存在。

- user_name：待删除的用户名。
- CASECADE：可选关键字。删除目标用户及其关联对象。CASCADE 不会列出待删除的关联对象，应谨慎使用。
- RESTRICT：默认设置，可选关键字。如果其他对象依赖目标用户，则无法删除目标用户。

9.1.3.3 语法示例

以下示例删除 user1 用户。

```
SQL
DROP USER user1;
```

9.2 资源使用限制

KaiwuDB Lite 支持用户使用 SQL 语句设置和查看数据库的线程数和内存使用情况。

9.2.1 线程数限制

9.2.1.1 设置线程数限制

9.2.1.1.1 语法格式

```
SQL
SET threads = <num>;
```

9.2.1.1.2 参数说明

num：要设置的线程数，必须大于等于 1。最大值通常为系统的 CPU 核心数。

> 说明
>
> 如果设置值大于系统的 CPU 核心数，系统不会报错，但可能无法获得最大性能。

9.2.1.1.3 语法示例

```
SQL
SET threads = 100;
```

9.2.1.2 查看线程数设置

9.2.1.2.1 语法格式

```
SQL
SELECT current_setting('threads');
```

9.2.1.2.2 参数说明

无

9.2.1.2.3 语法示例

```
SQL
SELECT current_setting('threads');
current_setting('threads')
--------------------------
                       100
```

9.2.2 内存限制

9.2.2.1 设置内存限制

9.2.2.1.1 语法格式

```
SQL
```

```
SET memory_limit = '<num> <unit>';
```

9.2.2.1.2 参数说明

- num：要设置的数值，通常为整数。有效设置范围为 32 MB 至 9007199254740991 字节（2^53-1）。
- unit：数值单位，支持以下选项：
  - 基于 1000 计算（10^3 进制）：KB， MB， GB， TB
  - 基于 1024 计算（2^{10} 进制）：KiB， MiB， GiB， TiB

> 注意
>
> - 该内存限制仅适用于缓冲区管理器，用于处理大部分查询数据。
> - 某些数据结构（如向量、查询结果）及复杂状态的聚合函数可能会使用缓冲区管理器之外的内存，因此，实际的内存消耗可能会高于设定的限制。

9.2.2.1.3 语法示例

```
SQL
SET memory_limit = '1024 MiB';
```

9.2.2.2 查看内存设置

9.2.2.2.1 语法格式

```
SQL
SELECT current_setting('memory_limit');
```

9.2.2.2.2 参数说明

无

9.2.2.2.3 语法示例

```
SQL
SELECT current_setting('memory_limit');
 current_setting('memory_limit')
 -------------------------------
 1.0 GiB
```

9.2.3 日志设置

KaiwuDB Lite 支持用户使用 SQL 语句设置数据库的日志功能及输出形式。

9.2.3.1 启用日志

启用日志语句用于控制是否记录系统日志。

9.2.3.1.1 语法格式

```
SQL
SET enable_logging = [true | false];
```

9.2.3.1.2 参数说明

- `true`：表示启用日志记录功能
- `false`：默认值，禁用日志记录功能

9.2.3.1.3 语法示例

```
SQL
SET enable_logging = true;
```

9.2.3.2 设置日志过滤模式

设置日志过滤模式语句用于配置系统输出的日志类型和过滤规则。

9.2.3.2.1 语法格式

```
SQL
SET logging_mode = [LEVEL_ONLY | DISABLE_SELECTED | ENABLE_SELECTED];
```

9.2.3.2.2 参数说明

- `LEVEL_ONLY`：级别过滤模式，仅依据日志级别进行过滤。
- `DISABLE_SELECTED`：黑名单模式，即排除指定类型的日志输出，该设置需与 `SET disabled_log_types` 语句配合使用。
- `ENABLE_SELECTED`：白名单模式，即仅保留指定类型的日志输出，该设置需与 `SET enabled_log_types` 语句配合使用。

9.2.3.2.3 语法示例

示例 1：级别过滤模式

```
SQL
SET logging_mode = LEVEL_ONLY;
```

示例 2：黑名单模式

```
SQL
-- 设置排除的日志类型
SET disabled_log_types =
'kaiwudb.FileSystem.LocalFileSystem.OpenFile,kaiwudb.ClientContext.BeginQuery
';


-- 启用黑名单过滤模式
SET logging_mode = DISABLE_SELECTED;
```

示例 3：白名单模式

```
SQL
-- 设置输出的日志类型
```

```
SET enabled_log_types =
'kaiwudb.FileSystem.LocalFileSystem.OpenFile,kaiwudb.ClientContext.BeginQuery
';


-- 启用白名单过滤模式
SET logging_mode = ENABLE_SELECTED;
```

9.2.3.3 设置日志记录级别

设置日志记录级别语句用于确定系统记录的日志详细程度，从最详细的 TRACE 到最严重的 FATAL。

9.2.3.3.1 语法格式

```
SQL
SET logging_level = [TRACE | DEBUG | INFO | WARN | ERROR | FATAL];
```

9.2.3.3.2 参数说明

- TRACE：最详细的跟踪信息
- DEBUG：调试信息
- INFO：一般信息性消息
- WARN：默认值，警告信息
- ERROR：错误信息
- FATAL：致命错误信息

9.2.3.3.3 语法示例

```
SQL
SET logging_level = WARN;
```

9.2.3.4 设置日志输出方式

设置日志输出方式语句用于确定日志信息的存储或显示位置。

9.2.3.4.1 语法格式

```
Shell
SET logging_storage = [file | memory | stdout ];
```

9.2.3.4.2 参数说明

- file：默认值，将日志存储到名为 kaiwudb-lite-<日期_时间>.log 的日志文件中。日志文件中每行为一个日志项，包含 8 个字段：时间、日志类型、日志级别、日志来源、客户端上下文 id、事务 id、线程 id。其中日志来源支持 DatabaseInstance 和 ClientContext，分别对应 DATABASE 和 CONNECTION。
- memory：将日志存储在内存中。
- stdout：将日志直接输出到标准输出（控制台或终端窗口）。

9.2.3.4.3 语法示例

```
Shell
SET logging_storage = stdout;
```

9.3 许可证管理

KaiwuDB Lite 使用许可证书管理用户登录和操作数据库的权限，以确保数据库的安全性和合法使用。用户连接到 KaiwuDB Lite 之后，可以通过以下 SELECT 语句查看许可证的相关信息，包括产品信息、客户名称、许可证到期时间、最大超期天数等重要信息。

```
SQL
select * from license_info();
```

下表列出用户在未设置有效许可证、使用有效许可证、许可证未超过最大超期天数、许可证超过最大超期天数的权限。

未设置有效许可证	许可证有效	许可证未超过最大超期天数	许可证超过最大超期天数
拒绝登录数据库。	正常登录并管理数据库。	拒绝新的会话连接。	拒绝登录数据库。服务器端自动断开已建立的会话连接。

9.3.1 配置许可证

安装部署 KaiwuDB Lite 时，用户需要配置许可证，否则部署失败。添加许可证后，系统会自动检查许可证的有效性。如果有效，则激活数据库并允许用户登录。否则，将拒绝用户登录。

1. 联系 KaiwuDB Lite 技术支持人员，获取 .lic 格式的 KaiwuDB Lite 许可证文件。
2. 将获取到的 .lic 格式的 KaiwuDB Lite 许可证文件复制到 /usr/local/etc（Ubuntu 和 KylinOS）或 C:\Users\Public\（Windows）目录中。
3. 配置并启动 KaiwuDB Lite。有关启停 KaiwuDB Lite 的详细信息，参见启停数据库。
4. （可选）查看许可证。

```
SQL
select * from license_info();
```

9.3.2 更新许可证

系统每 24 小时检查许可证设置的超期时间是否超过了当前时间。如果是，则许可证失效。用户需要在许可证到期前更新许可证，确保数据库持续运行。

1. 联系 KaiwuDB Lite 技术支持人员，获取新的 .lic 格式的 KaiwuDB Lite 许可证文

件。

2. 将获取到的 .lic 格式的 KaiwuDB Lite 许可证文件复制到 /usr/local/etc（Ubuntu 和 KylinOS）或 C:\Users\Public\（Windows）目录中。

3. 运行以下命令加载新的许可证文件。用户也可以重启 KaiwuDB Lite 加载新的许可证文件。

```
SQL
CALL load_license();
```